Lecture notes on Advanced Operating Systems

Politecnico di Milano

Claudio Migliorelli (prof. Vittorio Zaccaria)

April 2, 2023

Contents

1 Lecture 1 - Introduction to Operating Systems I

Some quick and introductory concepts

We define a *CPU* as a logic circuit able to execute instructions. A *program*, instead, is a sequence of instructions and definitions of algorithms, while a *process* is a program with associated data which is going to be actually operating on by the program itself (i.e., `process = program + ongoing data`). The difference between *user mode* and *kernel mode* is that in the former we can execute only a fraction of the instructions made available by the CPU, while the latter is a "more privileged" mode in which we can execute all instructions made available.

What is an operating system

An operating system is a software program that has several goals:

- **Resource management**: provide a single resource to several consumers in a way that the latter think they have the resource exclusively provided for them (e.g., assign CPU resources to processes)
  - In other words, the point is to allow programs to execute as they were the only ones assigned to the individual resource (e.g., their own CPU, the whole memory);
  - Another important aspect is that we want to ensure a fair and parsimonious utilization of the shared resources (i.e., avoid starvation), possibly by implying some sort of multiplexing techniques;
- **Isolation and protection**: we want to regulate/enforce access rights from applications to resources (e.g., memory) to avoid conflicts or applications clobbering one with each other
  - The operating system allows us to divide and separate the physical memory from what we call virtual memory in order for each application to see its own virtual memory without interfering with data belonging to other applications;

- **Make it easier to port and extend the entire system**: the idea is to hide the complexity of having direct access and interaction with peripherals (that may be different one with each other). The operating system is in charge of dealing with them, making the complexity transparent to the final user
  - System calls are the most important tool used by the operating system to implement that behavior;
  - The final result is to allow the same application to work on systems equipped with different physical resources (e.g., different GPUs, network interfaces, SSDs or mechanical disks, etc.).

Basically, the goals an operating system is intended to accomplish in that case are strictly related to two famous software patterns:

- *Facade*: hiding the complexity of hardware access/management from applications (abstraction to applications);
- *Bridge*: hiding the complexity associated with variants, e.g., peripheral models (abstraction to internal components).

Techniques used to implement resource management

The key idea here is to *multiplex the CPU*. Multiplexing the CPU means that we have a resource we don't want to waste and we are willing to allocate this resource fairly between processes. The operating system tries to **increase the CPU utilization** and to **reduce latency** by scheduling applications without forcing them to wait a lot of time to be executed.

This is typically done by using **interrupts**, to have a fair multiplexing when applications wait for something to happen (or even using timer interrupts to give resources to applications on a time-based approach). In other words, there are two steps needed in order to do that:

- The kernel programs *timers* to expire every x ms: writing appropriate I/O registers can be done only in supervisor mode, thus the user can't re-program these interval timers;

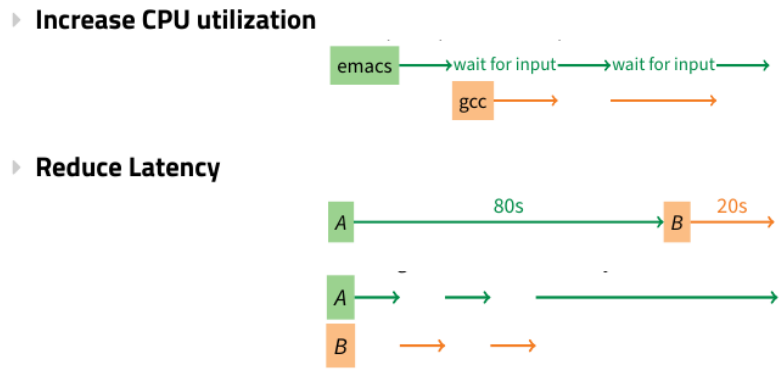


Figure 1: Increasing the CPU utilization and reducing latency

- The kernel sets interrupts to *vector back to kernel*: it regains control [1] whenever this interval timers expires and it gives, in turn, the control to another process that needs CPU resources.

As a result, there are no processes able to monopolize the CPU with infinite loops and, at worst, a single process could get $\frac{1}{N}$ of CPU with N CPU-hungry processes.

1. Processes and states A *process* is a program with associated data and its state are tracked by the operating system using a so-called **Process control block** (PCB). A PCB is a data structure (i.e., context) allocated for a each process and resides in *kernel space* and maintains:

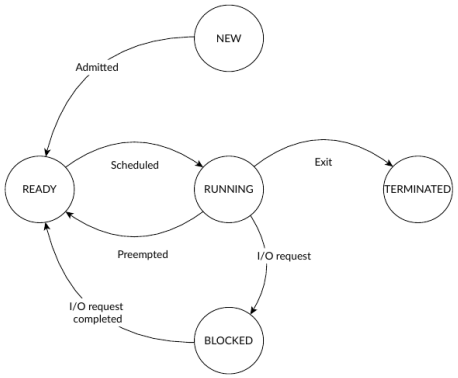
   - Machine registers related to that application;
   - Machine architectural state (e.g., user/kernel mode, virtual memory mappings);
   - Credentials (i.e., user/group ID), signal mask, controlling terminal, priority, accounting, statistics, etc.;
   - Open files (including memory mapped files).

   Processes can lie in five different states:

   - *NEW*: the process is not ready to run yet;
   - *READY*: ready to be scheduled when the CPU preempt the execution of a running process;

[1] There is no way for user code to hijack interrupt handlers, since it must be in supervisor mode to set up entry points.

- *RUNNING*: the process is running in a CPU-core until is preempted and sent to the ready state;
- *BLOCKED*: the process is waiting for an event to progress;
- *TERMINATED*: the process is terminated.



2. Context switches Operating systems that exploits interrupts to give control to applications based on time frames are called **preemptive** operating systems. Typically, when interrupt is sent to stop the execution of a process P_0, the operating system switches PCBs to give P_1 the capabilities to execute.

   A *context switch* is the activity of changing machine registers to run another process (by changing PCB and other OS-level resources). The challenge here is to select **which** other process to execute when performing a context switch (i.e., what process P_1 has to be). This problem is called **scheduling**.

3. Scheduling Scheduling is an activity used to decide which process needs to be run next. The implemented policy should balance between:

   - *Fairness*: do not make processes wait endlessly, making them starving;
   - *Throughput*: provide good overall performances;
   - *Efficiency*: minimize the overhead the scheduler itself has (take decisions in the shortest time possible);
   - *Priority*: reflect the relative importance of processes [2];
   - *Deadlines*: make applications execute for the time frame they are supposed to execute [3].

[2] Some processes, for their own nature, need to have more CPU than others. Priorities are a way to represent this and they are interpreted in different ways in different operating systems.

[3] This makes us differentiate between operating systems as general purpose and real-time.

4. Differences between general purpose and real-time OSes When we have a *general purpose* operating system, generally it employs fairness, deciding which process needs to be executed based on time intervals and priorities, in a fair way.

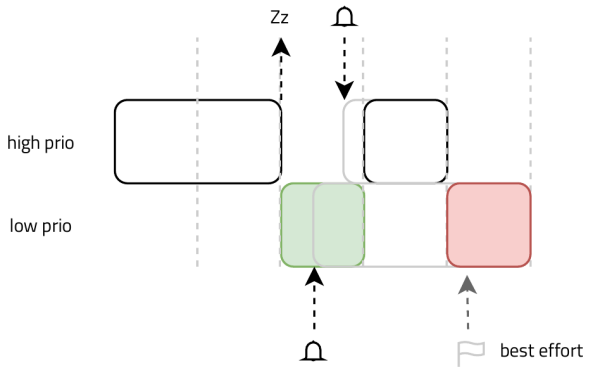


Figure 2: How general-purpose operating systems handle scheduling

We have intervals in which the operating system decides what the next running process must be and whenever a time slice finishes (and whenever the operating system decides the current process to release the CPU) we say that the current process is preempted. A general purpose operating system, as it may seem clear looking at the picture above, tries to pursue fairness, even with respect to lower priority processes and does not interrupt the current process until it reaches the end of its time slice.

Real-time operating systems try to be always predictable in terms of CPU allocation: high priority processes always have advantage on lowest priority ones.

Higher priority threads are always given access to the CPU in that scenario and the underlying assumption is that they usually do things for a limited time and then go to sleep (the idea is that we still have time to execute lower priority

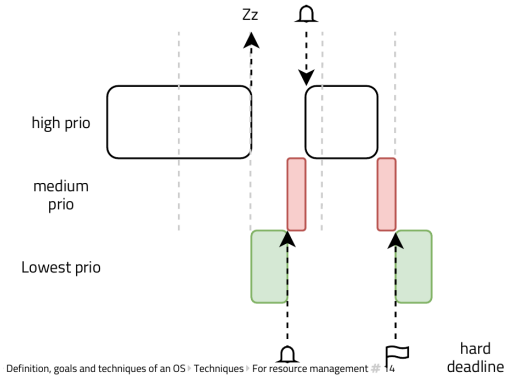


Figure 3: How real-time operating systems handle scheduling

threads). Real-time operating systems typically have more priorities to choose from with respect to general purpose operating systems.

Memory management

Whenever there is a process, we also have its *virtual address space*, which is constituted by all memory locations a program can name. These memory locations are built up from *virtual memory areas*.

Some of these memory areas are taken from the on-disk representation of the program (e.g., `text`, `rodata`, `data`, `bss`) while others are built up dynamically (e.g., stack, heap, memory mapped segments and `vdso`). There also also other memory locations which are not accessible, e.g., kernel space.

We can't however allocate the memory as it is in physical memory, since we want to avoid other processes to read areas not intended for them. For this reason, tables used for the virtual-to-memory mapping shouldn't allow pages related to different processes to refer to the same physical location, unless they are specified to do so. This is the core functionality of a key concept called **isolation**.

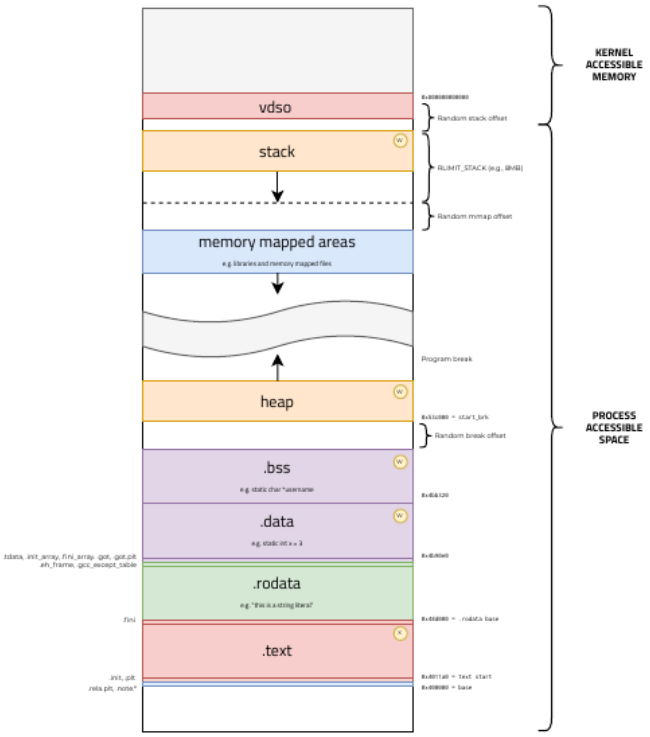


Figure 4: Process address space

2 Lecture 2 - Introduction to Operating Systems II

Introduction to tasks

Tasks are similar to programs but they are something more: they have additional data assigned to them (e.g., program counter, stack). In other words, they are a more general concept and in Linux there is a clear distinction between:

- Tasks;
- Processes;
- Threads.

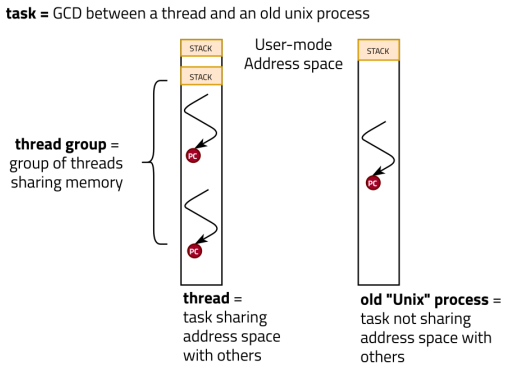


Figure 5: Task vs. process vs. thread

Threads are basically tasks that share the same memory (i.e., address space), while processes are tasks not sharing the address space with others. We could see a task as a ”super-class” that embodies processes and threads (as we said, it is a very general concept).

When the task is in user or kernel space on behalf of the process (e.g., executing a system call), we say that the kernel is in **process context**. The kernel itself can create threads which live always in kernel mode.

Kernel threads

There are also **kernel threads**: they run with kernel privileges (i.e., they see kernel's memory) and are routines and background work performed by the kernel. The address space of that threads is the entire kernel's address space. Note that these are *schedulable* and *preemptable*, which is the same for normal processes, and they are created by the kernel for specified purposes as children of the `kthreadd` kernel process.

The `task_struct`

The data structure the kernel holds to manage tasks is `task_struct` (also called Task Control Block), which forms a hierarchy within kernel memory (through the `parent` field in the data structure itself) [4]. In such a data structure, the kernel holds a description of the virtual memory address space, current and root directories, opened files, etc.

[4] The only task that does not have a parent is the first task (init).

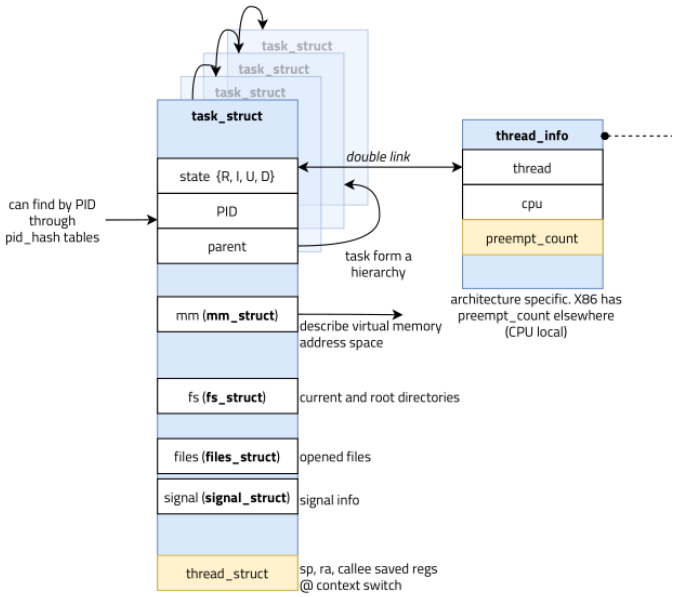


Figure 6: The `task_struct` and `thread_info` structures

Whenever the kernel wants to manipulate the structure associated to a task, this should be done right away and in an efficient way. In some architectures (e.g., RISC-V) the pointer to task structures is stored in a register always available to the kernel. In x86, instead, there is an additional structure called `thread_info` kept in the kernel-level stack, which is used by the kernel to reach the `task_struct` by computing its address. A very crucial field in the `thread_info` structure is `preempt_count`, used by the Linux kernel to understand whether, when transitioning to kernel-mode during exceptions or interrupts, it can do something. This flag prevents the kernel from messing up with interrupt nesting. The `thread_struct` field, instead, is used only during context switch to save callee-saved registers and make the `switch_to()` invocation like a "normal" function [5].

There is a macro exported by the kernel, which is called `current`, that takes care of getting the reference to the `task_struct` of the current process under execution.

[5] However, this is not a normal function: it won't return immediately but after the task is going to be rescheduled to run again.

Task states in Linux

A task can be in three main and general states (the Linux kernel uses more than these, as we will see):

- *Running*;
- *Ready*;
- *Blocked*.

For some historical reasons Linux uses the same label (i.e., running) both for a task ready but not yet running and a task currently running [6]. Whenever our process terminates, the task is put in a temporary "terminated" state: this is because its parent task wants to have a look at the final state of the task (e.g., signals, errors). The blocked state comes in two different flavors:

[6] The kernel differentiates a task already running from a ready one from the kept Task Control Block.

- `TASK_INTERRUPTIBLE`;
- `TASK_UNINTERRUPTIBLE`.

These are basically the same state (i.e., the task is waiting for something to happen), but in the first case, if some process signals the blocked task, then the latter can be waken up, even before the event had occurred. When the task is uninterruptible, instead, no signal (apart from the event occurrence) can wake up the blocked thread. An example could be a `kill` from command line towards a blocked thread: if it is uninterruptible, there would be no way to tell that process to do some task before being killed.

Finally, we could summarize task states in Linux as follows:

- `TASK_RUNNING`: the process is runnable and it is either currently running or in a run-queue waiting to run;
- `TASK_INTERRUPTIBLE`: the process is sleeping (i.e., it is blocked), waiting for some condition to exist. It wakes up and becomes runnable when receiving a signal;
- `TASK_UNINTERRUPTIBLE`: the process is sleeping (i.e., it is blocked), but it doesn't wake up when receiving a signal. This state is less often used than *interruptible* (only when the event is expected to arrive rapidly);

- `TASK_DEAD`: the process is terminated and put in this temporary state. When in this state, it will no longer be executed, while its metadata are kept in memory until its parent thread collects them (e.g., inspecting the return value). These processes are called **zombies**.

Wait queues

Linux manages events by using so-called **wait queues**: structures that maintain tasks in a wait state for a *specific event*. Tasks are added whenever a driver/module invokes functions called `wait_event()` or `wait_event_interruptible()`. Those are used by the kernel to put current tasks in specified queues to wait for specific events.

However, wait queues have a problem: whenever the event happens there would be something invoking `wake_up()`, waking up **all** tasks waiting for that event (i.e., tasks inside that wait queue). The point is that, in the majority of the cases, only the first task could consume that event (i.e., the one "winning" the race), while the others would remain inside the wait queue. This problem is called **thundering herd**.

For solving this problem, the kernel defines two kinds of sleeping processes:

- *Exclusive*: always put at the end of the queue. When the wake up reaches them, it stops;
- *Non-exclusive*: always woken up by the kernel.

The idea is to wake up threads in an exclusive way: the kernel uses an additional part of the list to wake up the first task exclusively in such an additional list. As a result, all threads without the `WG_FLAG_EXCLUSIVE` will be waken up, and only one of the tasks having that flag will wake up from being blocked. This flag is specified when calling `wait_event` APIs.

Task creation

Task creation in Linux goes by cloning: any call to `fork()` creates a new copy of the current `task_struct`. The copy *differs* from the parent only according to the following aspects:

- PID, which is unique, and PPID, which is the parent's PID and it's set to the original process (i.e., the one forking);

- Certain resources, such as pending signals, which are not inherited.

Linux uses a "copy-on-write" way of managing resources whenever we do a `fork()`: whenever the child thread changes something, it tries to share the original data in read-only. Rather than duplicate the process address space, parent and child share a single copy. If the data is written to, a duplicate is made and each process receives a unique copy. The copy-on-write is made possible by setting up writeable VMAs as read-only. In this way, the kernel can intercept on page fault and do the copy.

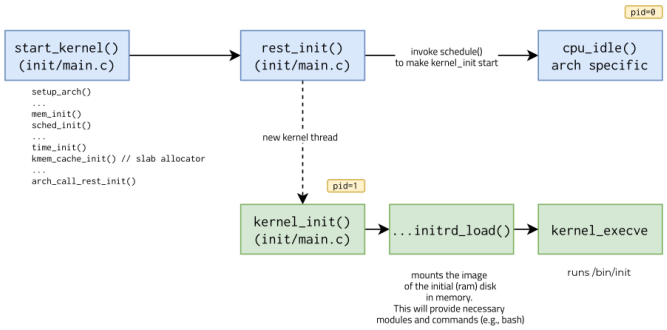
The `libc`'s function `fork()` invokes the system call `sys_clone()`, which is going to set up all the original virtual memory areas from the parent process in read-only mode [7].

[7] During this process, the `sys_clone()` shallow copies the parent's `task_struct` (and the `thread_struct`) by calling `dup_task_struct()`.

The `init` task

Let us inspect quickly the initialization of the kernel: when it starts some routines are executed to bringing it to create the first task (with `PID=1`). In particular, the kernel creates a new kernel thread that executes the `/bin/init` (first task) and a routine called `invoke_schedule()` creates another dummy task called `cpu_idle()` [8].

[8] The `cpu_idle()` routine is an always-running activity that is used for the kernel itself to pull the CPU in low-power mode when it doesn't execute any routine.



The `/bin/init` is really important because it is used to start everything else (e.g., mounting disks, starting services) and there are two main approaches to make this process to perform such a thing:

- `System V`;
- `systemd`.

1. Introduction to `System V` `System V` was released in the early '80s and specifies to the `init` process all the programs to run by collecting them in *run levels*. These run levels are basically collections of tasks to start and correspond to a specific configuration of the machine. So, at any time, the system is in one of these run levels:

   - 1: single user;
   - 2: multi-user with no net;
   - 3: multi-user;
   - 5: X11;
   - 6: reboot.

   The file `/etc/inittab` describes which processes need to be run for each run level. The format is `id:rl:action:command`, where `action` can be, e.g., `wait`, `start`, `respawn` and processes could be either simple commands or bash script to initialize subsystems. The latter are called `rc<n>` scripts and invoke subsystem scripts in `init.d` through links.

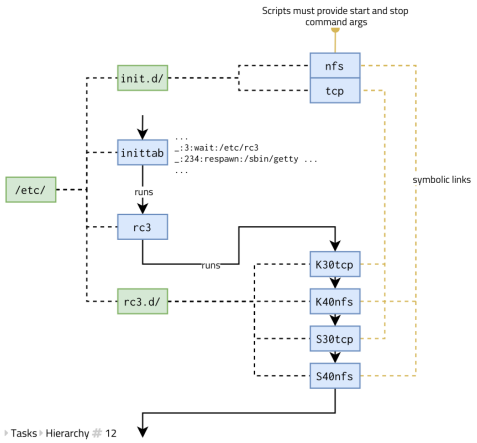


Figure 7: `System V` architecture

The most used `System V` configuration is the number `3`, which is the multi-user one. In this case, the `rc3` script is run.

`System V` is an imperative interface (with a collection of scripts) and it is basically single-threaded, everything is done sequentially [9], which is a huge disadvantage (i.e., the startup takes a lot of time). A new and parallel architecture is `systemd`.

[9] This means that, for the example in figure, `K30tcp` is set up first, then `K40nfs` etc.

2. Introduction to `systemd` `systemd` is structured by `Unit` files, which are plain text ini-style files that encode information about a service, a socket, a device, a mount point, an automount point. These files have their own semantics and they are declarative, saying for each service if there are other units to be run before that service. `Services` are the most common type of units and are described in `.service` files. What they do is encoding information about a process controlled and supervised by `systemd`. They contain the ordering of required units to be started as dependencies. `Targets` are collections of units and emulate old run-levels (e.g., `/lib/systemd/system/multi-user.target` is a file that contains links to `RL3` units).

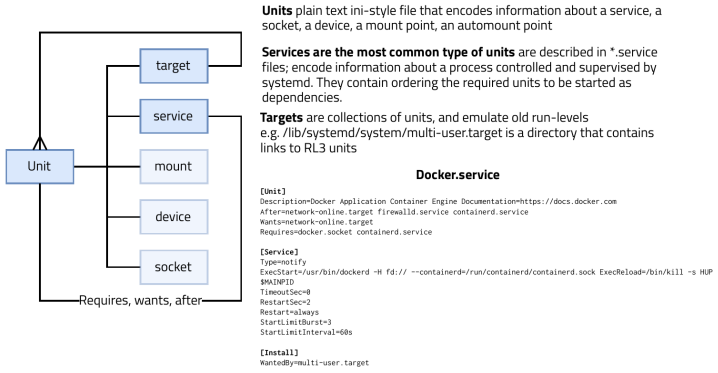


Figure 8: `systemd` architecture

Task scheduling

From its first version in 1991 to the 2.4 kernel series, the Linux scheduler scaled poorly in light of many runnable processes and/or many processors. With version 2.5, a new scheduler, called *O(1)*, introduced a *constant-time algorithm* for searching the highest priority process in a fixed-length bitmap. However, O(1) was ideal for large server workloads but performed poorly on interactive applications. Moreover, its code was complex and error prone and the

algorithm itself could led to unfairness. In next kernel's versions, the CFS algorithm was introduced to give better responsiveness.

Scheduling classes

A scheduling class is an API used internally by the kernel to specify some policy-specific way to pick up tasks from the "ready queue" (i.e., `select_task_rq`) to make them run (i.e., `enqueue_task`). Depending on the context of use, we could have several policy to encode in the implementation of these APIs.

`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()`/`clone()` system call
`task_dead(t)`	Called when thread `t` terminates

Figure 9: Scheduling class APIs for Linux 5.7 (subset)

There are scheduling classes associated with normal tasks, which doesn't have many constraints in terms of what they should do (i.e., *normal scheduling policies*). The other set of policies, used as an alternative for certain tasks, is the one associated with real-time processes (i.e., processes that must respect somehow measures of urgency and deadlines). Typically we sort these scheduling classes starting with those having a deadline to be respected, those that need to be executed up to the end (i.e., FIFO), those that are high priority but that can share the processor in a round-robin way (i.e., RR), and so on. The last two classes are `SCHED_BATCH` and `SCHED_IDLE`: the former is like CFS but has a different and longer time slice to be used to manage tasks (can be used for tasks not response-oriented), while the latter comprises tasks that can be run when the CPU is not busy.

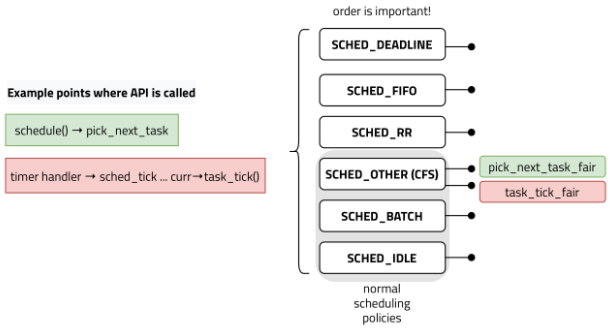


Figure 10: Scheduling classes

To summarize:

- `SCHED_FIFO`: like RR but if a task does not release the CPU, it will run indefinitely even if there are other tasks having the same priority;
- `SCHED_RR`: tasks will repeatedly go ahead of any other task having lower priority. If multiple tasks have the same priority, the RR policy will be applied;
- `SCHED_OTHER`: conventional time-shared approach. In general, it has a soft priority mechanism over the "nice" range of -20 to +19, which decides, according to the priority, which task goes first and what time slice it gets;
- `SCHED_BATCH`: has longer time slices (1.5s) thereby allowing tasks to run longer and make better use of caches, but at the cost of interactivity. This is well suited for batch jobs;
- `SCHED_IDLE`: schedule only if the CPU is `idle`. When in kernel mode, these behave like `SCHED_OTHER` to avoid soft lockups (e.g., not released semaphores);
- `SCHED_DEADLINE`: implementation of the *Earliest Deadline First* (EDF) scheduling algorithm, augmented with *Greedy Reclamation of Unused Bandwidth* (GRUB) from kernel 4.13.

However, if multiple policies have a runnable thread, Linux has a simple fixed-priority list to determine the order:

```
deadline -> realtime -> fair -> idle
```

When a thread is scheduled out, the scheduler subsystem will iterate over this list and call the `pick_next_task()` function of each class until a thread is returned.

The scheduler performs the so-called *load balancing*: it tries to migrate threads between cores in order to even the number of threads running on all cores. Load balancing is done using a "work stealing" approach: on each core, the kernel does its own balancing and tries to steal threads from the busiest core on the system.

Completely Fair Scheduler (CFS)

Completely Fair Scheduler has been introduced in 2007 for non real-time processes to solve previous problems. We still have completely separated sets of processes with a certain normal priority π:

- *Real-time processes*, $\pi \in [0, 99]$: they belong to `SCHED_FIFO` and `SCHED_RR` scheduling classes;
- *Non real-time processes*, $100 \leq \pi(\nu) \leq 139$: the value $\pi(\nu)$ depends on a *nice* value $\nu \in [-20, +19]$:

  $$\pi(\nu) = 120 + \nu.$$

  Processes following this approach belong to the `SCHED_NORMAL` scheduling class.

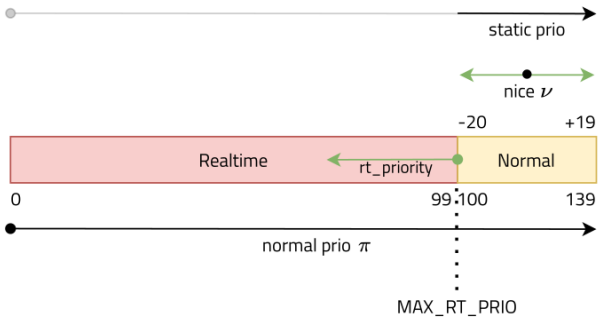


Figure 11: CFS priorities and scheduling classes

To inspect processes and their associated "nice" value, we could use the following command:

```
$ ps ax --format comm,pid,pri_baz,nice,cls
COMMAND             PID BAZ  NI CLS
systemd               1 120   0  TS
kthreadd              2 120   0  TS
rcu_gp                3 100 -20  TS
rcu_par_gp            4 100 -20  TS
netns                 5 100 -20  TS
kworker/0:0H-ev       7 100 -20  TS
mm_percpu_wq          9 100 -20  T
```

Linux run queues

Run queues are central data structures used by the core scheduler to manage active processes. Multiple run queues (one per CPU) are needed to avoid contention over task selection among multiple processors. Practically speaking, each scheduling class has a run queue that contains runnable tasks. These tasks are grouped in a general `struct rq` for each core, structured as follows:

```
/*
 * This is the main, per-CPU runqueue data structure.
 *
 * Locking rule: those places that want to lock multiple runqueues
 * (such as the load balancing or the thread migration code), lock
 * acquire operations must be ordered by ascending &runqueue.
 */
struct rq {
/* runqueue lock: */
raw_spinlock_t __lock;

/*
 * nr_running and cpu_load should be in the same cacheline because
 * remote CPUs use both these fields when doing load calculation.
 */
unsigned int nr_running;

// ...

struct cfs_rq cfs;
struct rt_rq rt;
```

```
struct dl_rq dl;

// ...


/*
 * This is part of a global counter where only the total sum
 * over all CPUs matters. A task can increase this counter on
 * one CPU and if it got migrated afterwards it may decrease
 * it on another CPU. Always updated under the runqueue lock:
 */
unsigned int nr_uninterruptible;

struct task_struct __rcu *curr;
struct task_struct *idle;
struct task_struct *stop;
unsigned long next_balance;
struct mm_struct *prev_mm;
// ...
}
```

The `cfs_rq` contains CFS-related statistics and attributes and it is defined as follows:

```
/* CFS-related fields in a runqueue */
struct cfs_rq {
struct load_weight load;
unsigned int nr_running;
unsigned int h_nr_running;      /* SCHED_{NORMAL,BATCH,IDLE} */
unsigned int idle_nr_running;   /* SCHED_IDLE */
// ...

/*
 * 'curr' points to currently running entity on this cfs_rq.
 * It is set to NULL otherwise (i.e when none are currently running).
 */
struct sched_entity *curr;
struct sched_entity *next;
struct sched_entity *last;
struct sched_entity *skip;

// ...
}
```

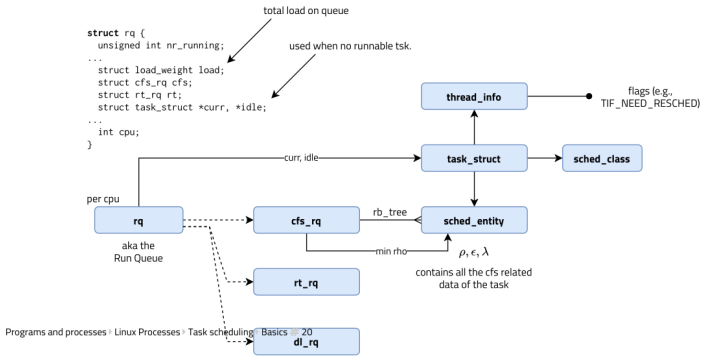
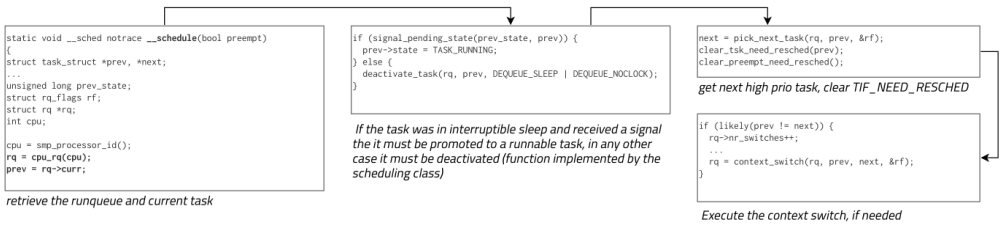


Figure 12: Run queues structure

Linux `schedule()` function

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/interrupts or when a thread does some explicit blocking, with mutexes/semaphores or when it is put inside a wait queue).



3 Lecture 3 - Concurrency I

Recapping some definitions

We define as *program* a set of instructions stored somewhere (e.g., disk) but that are not doing anything at that time (i.e., instructions are not executed). *Applications*, instead, are software whose main goal is to interact with the user, typically with a GUI. Finally, a *process* lives in memory and it is currently in execution. Note that an application is more user-oriented, while processes are more on the operating system side.

We define *thread* as the smallest schedulable unit of execution in a way that the latter can't be divided further. A *task* is the most general term and has a no specific definition. It usually refers to a single unit of computation at a conceptual level. In practice, it can be mapped to:

1. A process;
2. An application;
3. A thread.

In Linux, and not in all operating systems, task is a synonym of thread.

Processes vs. threads

Definitions may vary, but, looking at the Linux view, a *process* can be defined as an instance of a program, which has its isolated memory address space. *Threads*, instead, are multiple in the context of a process, thus they belong to the address space of the process. Moreover, a thread shares the address space with other threads belonging to the same process and they run sequentially on the CPU (with exceptions related to *hardware parallelism*.

Multi-threaded programming

Threads can communicate one with each other and they synchronize themselves when accessing shared data (heap, data, etc.). They can access variables belonging to other threads, if they know their addresses.

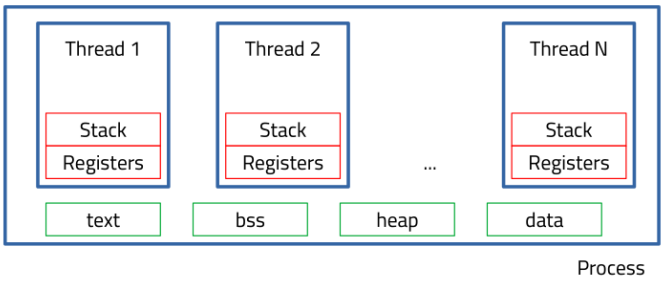


Figure 13: Multi-threaded environment

Multi-process programming

Given that two different processes have different address spaces, we need an intermediate layer (i.e., the operating system) to communicate one with each other [10]. This activity is called **inter-process communication** (IPC).

[10] This intermediate layer has the responsibility of forwarding data between processes.

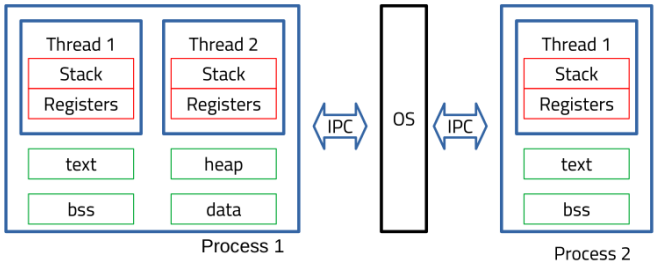


Figure 14: Performing IPC to share data between processes

Forking

The starting point of a process, from a programming perspective, is given by the library function `fork()`, which is based on the system call `sys_clone()`. The idea is to clone the program and to make the computation start right after the `fork()` itself (i.e., starting from the next instruction after the call).

When a `fork()` is called, the underlying operating system creates a new address space and all the variables share the same value, except the `fork()`'s return value. On the "physical" side, the *copy-on-write* strategy is applied to all pages.

1. The process tree Every process has exactly one parent and may have an arbitrary number of children. The only exception is the `init` process, which is the ancestor of all other processes.

   Every process has an unique identifier called PID and a parent identifier known as PPID. We can get both the current PID and PPID executing, respectively, `getpid()` and `getppid()`. In Linux, the PID is of `pid_t` type (defined in `<sys/types.h>`), which is currently a 32-bit integer [11].

[11] The PID is limited by the special file `/proc/sys/kernel/pid_max`. When the `pid_max` is reached, no other process can be created. Note that this is a sort of DDoS attack.

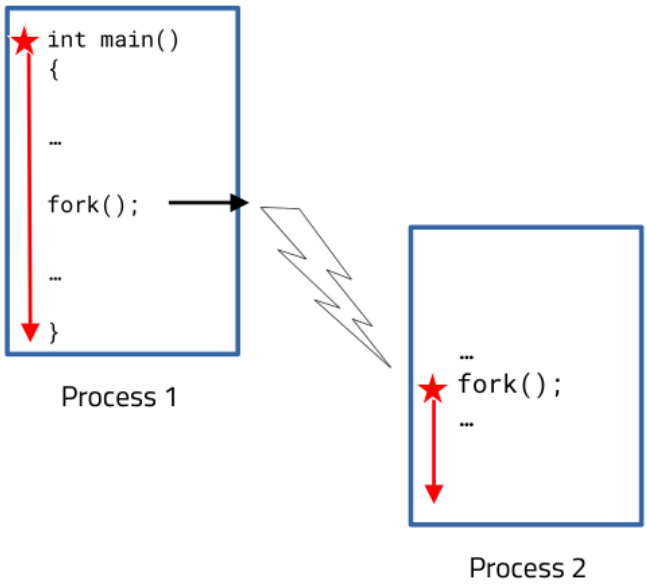


Figure 15: The forking technique

The boot process on Linux

The boot process on Linux works in the way described below:

1. Power on: the system is powered on, the firmware (BIOS/UEFI) is loaded in memory;
2. BIOS: the firmware performs checks and launches the *bootloader*;
3. Bootloader: the bootloader mounts file-systems and loads the kernel image (e.g., `/boot/vmlinuz-*`);
4. Linux kernel: The kernel starts running;
5. Low-level initialization: The first code initializes the CPU, programs the MMU, does the transition to 64-bit mode, etc.;
6. Kernel image decompression: the kernel image is decompressed
7. `start_kernel()`: non-architecture specific initialization: interrupt setup, memory configuration, scheduler initialization, etc.;

8. Start `init` process: the `init` process is started, by default is searches for `/sbin/init`.

As we just said, `init` is the first user-space process launched at the end of kernel boot. This always has `PID = 1` and `PPID = 0` since it is the ancestor of all user-space processes in the system. What `init` does is to start all enabled services during startup (e.g., DHCP, web server, graphic server, NTP). There are two common implementation of the `init` system:

- `System V` (legacy): loads all services from `/etc/init.d` and `/etc/rc.d`;
- `systemd`: loads all services according to `/etc/systemd`.

Executing

The `fork()` by itself is not useful to have a complete operating system and set of overlying applications: we need a way to load and run another executable. For that matter, we have the `exec*()` family of functions: they load a new program and replace the current process image with it by calling the `execve()` system call.

The `fork()` is typically used only to generate the new address space, while this address space is populated with a new process image using `exec*()` functions.

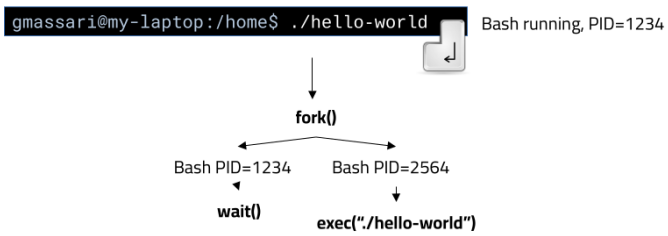


Figure 16: `fork()` and `exec()` mechanism

All functions in the `exec*()` family take the executable path as the first argument:

```
#include <unistd.h>
int execl (const char *path, const char *arg, … );
int execlp(const char *file, const char *arg, … );
int execle(const char *path, const char *arg, …, char * const envp[]);
int execv (const char *path, char * const argv[]);
int execvp(const char *file, char * const argv[]);
int execve(const char *path, char * const argv[], char * const envp[]);
```

Each function has slight variations based on arguments:

- `-l` functions accept list of `NULL`-terminated parameters;
- `-v` functions accept an array of `NULL`-terminated strings;
- `-p` functions search the `PATH` environment variable;
- `-e` functions allow to specify new environment variables.

Parent-child basic synchronization

When we `fork()` a process and we execute the same function both in parent and child, we have no assurance that they will perform operations "in order". In other words, *scheduling order* is unpredictable: we need some **synchronization** mechanisms. The simplest one for parent-child relation is the `wait()` based mechanism:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

These are two functions performing a simple parent-child synchronization:

- The `wait()` suspends the execution until one of the children terminates;
- The `waitpid()` suspends the execution until a specific child process terminates (or changes state).

The parameter `status` is a pointer to a variable where to write the return value (and it can be `NULL`), `pid` is the PID of the child to wait and `options` are extra-option flags.

How to avoid zombies

Zombies are dead processes indeed, but they are still there: they consume memory and resources because a parent process forgot to use the `wait()` function [12].

When a process terminates, the Linux kernel changes its state to "zombie" and saves the return value. In particular, the kernel returns the child's return value to its parent, when the latter calls `wait()`, or `waitpid()`, on the child process. If the parent terminates before calling `wait()`, the child is adopted by `init`, which

[12] Processes return to the parent only if the latter used a `wait()` function indeed.

performs `wait()` on all children, freeing the memory and the PID number.

As a rule of thumb, persistent zombie processes in the system are a sign of a programming error.

Inter-Process Communication (IPC)

Since different processes have their own address space, we must have a way to exchange information between them. This problem is called *Inter-Process Communication*. In Linux, there are two libraries providing IPC:

- *POSIX*, which is newer and thread-safe;
- *System V*, which is basically a legacy software and it is not thread-safe.

Signals

Signals are unidirectional and their information content is only the ”signal type”. This means that there is no data transfer being performed, and, moreover, they are basically asynchronous. Examples of signals could be elapsed timers, I/O operation successes or failure events, exceptions, user-defined events, etc.

Remember that signals are communication methods between processes, *not threads*, and they are sent by the underlying operating system. Some examples could be:

- A terminating child process sending a `SIGCHLD` to its parent;
- An user pressing `Ctrl+C` on the keyboard, which implies a `SIGINT` to the process;
- A process trying to execute an illegal instruction, implying a `SIGILL` from the operating system;
- A segmentation fault, which means that a `SIGSEGV` signal is sent.

POSIX signals	Number	Default action	Description
SIGHUP	1	Terminate	Controlling terminal disconnected
SIGINT	2	Terminate	Terminal interrupt
SIGILL	4	Terminate and dump	Attempt to execute illegal instruction
SIGABRT	6	Terminate	Process abort signal
SIGKILL	9	Terminate	Kill the process
SIGSEGV	11	Terminate and dump	Invalid memory reference
SIGSYS	12	Terminate and dump	Invalid system call
SIGPIPE	13	Terminate	Write on a pipe with no one to read it
SIGTERM	15	Terminate	Process terminated
SIGUSR1	16	Terminate	User-defined signal 1
SIGUSR2	17	Terminate	User-defined signal 2
SIGCHLD	18	Ignore	Child process terminated, stopped or continued
SIGSTOP	23	Suspend	Process stopped

Figure 17: Most common POSIX signals

4 Lecture 4 - Concurrency II

More on signals

Sending a signal

To send a signal we can use the `kill()` function:

```
#include <signal.h>
#include <sys/types.h>
int kill(pid_t pid, int sig);
```

We have that `pid` is the PID of the receiver process and `sig` is the signal to send (i.e., a `SIG_<something>` constant). The `kill()` return value is `0` on success and `-1` on error.

Handling a signal

To receive and handle a signal we can use the `sigaction()` function:

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act,
      struct sigaction *oldact);
```

We have that `signum` is the signal to catch (i.e., a `SIG_<something>` constant), `act` is a data structure containing new settings to apply in order to register a handler function and `oldact` is an output variable, used to save the old set settings if not `NULL`. The return value could be `0` on success or `-1` on error.

To register a signal handler we must deal with the `sigaction` data structure:

```
struct sigaction {
  void (*sa_handler)(int);
  void (*sa_sigaction)(int, siginfo_t *, void *);
  sigset_t sa_mask;
  int sa_flags;
  void (*sa_restorer)(void);
};
```

The structure's fields are the following:

- `sa_handler` is a handler function (or `SIG_IGN`);
- `sa_sigaction` is an alternative handler;
- `sa_mask` is a mask to be set to block certain signals;
- `sa_flags` contains various options;
- `sa_restorer` is not intended to be used by user applications.

POSIX Real-Time Extension

The POSIX real-time extension, which is available on any modern Linux, introduces the following functions:

- `sigqueue()` to send a queued signal;
- `sigwaitinfo()` to synchronously wait a signal;
- `sigtimedwait()` to synchronously wait a signal (for a given time).

The field `sa_signaction` in the `sigaction` data structure, allows to specify a handler that accepts an input data. Flags must contain `SA_SIGINFO`.

Masking a signal

Signals can be masked (i.e., blocked) to avoid disrupting our code execution. The masking process is similar to `SIG_IGN`, but instead of being dropped, they are enqueued and managed later when the process unmasks the signal. Note that `SIGKILL` and `SIGSTOP` can't be masked.

We have the following function prototype:

```
int sigprocmask(int how, const struct sigset_t *set,
struct sigset_t *oldset);
```

Where `how` could be either `SIG_BLOCK` (i.e., add to the mask), `SIG_UNBLOCK` (i.e., remove a signal from the mask), `SIG_SETMASK` (i.e., replace the mask), then `set` is the set of signals and `oldset` is the output value, which is the previous set of signals.

Unnamed pipes

Unnamed pipes are a POSIX-defined mechanism to let parent and child processes communicate. To let general processes communicate with each other there is another POSIX-defined mechanism called *named pipes* (FIFO) that we are going to discuss later on.

Unnamed pipes are based on the producer/consumer pattern: one producer writes and one consumer reads (i.e., the communication is unidirectional). Data is written/read in a first-in-first-out (FIFO) fashion, as shown in figure.

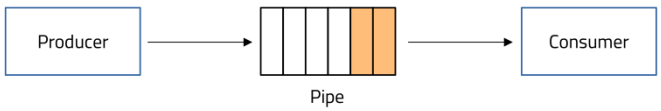


Figure 18: How unnamed pipes work

In Linux, the operating system guarantees that only one process at a time can access the pipe, using a so-called *implicit synchronization*.

Creating a pipe

To create a pipe we can use the following functions:

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

int pipe(int pipefd[2]);
int pipe2(int pipefd[2], int flags);
```

We have that `pipefd` is an array of two integers to be filled with two file descriptors:

- `pipefd[0]` is the file descriptor of the *read end* of the pipe;

- `pipefd[1]` is the file descriptor of the *write end* of the pipe.

The `flags` parameter, instead, can be either:

- `O_CLOEXEC`: close the file descriptor if `exec*()` is called;
- `O_DIRECT`: perform I/O in ”packet mode”;
- `O_NONBLOCK`: avoid blocking read/write in case of empty/full pipe.

Using a pipe

To use a pipe we can either do it directly with low-level I/O functions, such as

```
ssize_t write(int fildes, const void *buf, size_t nbyte);
ssize_t read(int fildes, void *buf, size_t nbyte);
```

or we can transform our file descriptor to a stream, using

```
#include <stdio.h>
FILE *fdopen(int fildes, const char *mode);
```

and then exploit all the `f*` functions (e.g., `fwrite()`, `fread()`, `fprintf()`, `fscanf()`).

Named pipes (FIFO)

Named pipes have basically the same behavior of unnamed ones: one producer writes and one consumer reads (i.e., the communication is unidirectional). In other words, data is written/read in a first-in-first-out (FIFO) fashion (as unnamed pipes indeed), as shown in figure.

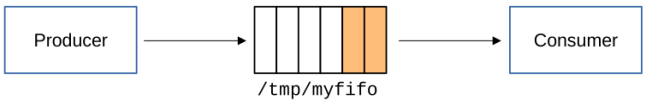


Figure 19: How named pipes work

Named pipes, differently from unnamed ones, are based on special files created in the file-system and not on file descriptors. Data is transferred in the same way as reading/writing to a disk file, but no actual I/O operation is involved: the operating system passes data from one process to another.

Creating a FIFO

To create a FIFO, we can use

```
#include <sys/types.h>
#include <sys/stat.h>

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

Where `pathname` is the usual `path + filename` of the file to be created, `mode` is related to file permissions we want the file to have (e.g., `S_IWUSR` are file permissions for the owner `0200`). Then, we only need to just use `open()`, `write()` or `read()` to access the FIFO.

Message queues

Message queues are an IPC method suitable for multiple readers and multiple writers. They are based on a priority-queue and works as shown in figure.

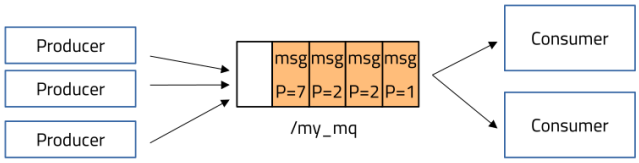


Figure 20: How message queues work

The status of the message queue can be observable and all special files related to the queues are in a system-wide directory, which is `/dev/mqueue`. When using message queues we are required to link our executables to the POSIX real-time extension library [13].

[13] This means that we should compile with `gcc` using the flags `-lrt`.

Opening/creating a message queue

To open/create a message queue we can use

```
#include <mqueue.h>
mqd_t mq_open(const char *name, int oflag, mode_t mode,
      struct mq_attr *attr);
```

where `name` is a unique name for the message queue (starting with `/`), `oflag` is an opening flag (e.g., `O_RDONLY`, `O_WRONLY`, `O_CREAT`), `mode` is related to file permissions to give to the file (only for `O_CREAT`), and `attr` are attributes:

```
struct mq_attr {
  long mq_flags; // 0 or NON_BLOCK
  long mq_maxmsg; // max nr. messages in the queue
  long mq_msgsize; // max message size in bytes
  long mq_curmsg; // nr. messages currently in the queue
}
```

There are also other management functions, such as the following:

```
#include <mqueue.h>
int mq_close(mqd_t mqdes);
int mq_unlink(const char* name);
int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
int mq_setattr(mqd_t mqdes, const struct mq_attr *newattr,
       struct mq_attr *oldattr);
```

Message queue input/output

To send messages over a message queue we could use:

```
#include <mqueue.h>
int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
    unsigned int msg_prio);
```

We have that `mqdes` is the message queue descriptor, `msg_ptr` is the pointer to the message to be sent, `msg_len` is the length of the message (in bytes) and `msg_prio` is a non-negative priority value in the range $[0, 31]$ [14]. The higher `msg_prio` value is, the higher the priority, while for messages with the same priority the FIFO approach is applied.

[14] The maximum priority value can be larger in some implementations (POSIX requires at least 32 priority values).

To receive messages from the queue we can use:

```
#include <mqueue.h>
int mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
       unsigned int *msg_prio);
```

We have that `mqdes` is the message queue descriptor, `msg_ptr` is the output parameter (pointer to a buffer to fill), `msg_len` is the length (in bytes) of the buffer and `msg_prio` is the output parameter (the priority of the message).

Shared memory

Shared memory is an IPC mechanism to allow two processes to share a memory segment. In POSIX, the shared memory is based on the **memory mapping** concept. To use shared memory, we should link, when compiling, to the POSIX real-time extension library using `gcc` with flags `-lrt`.

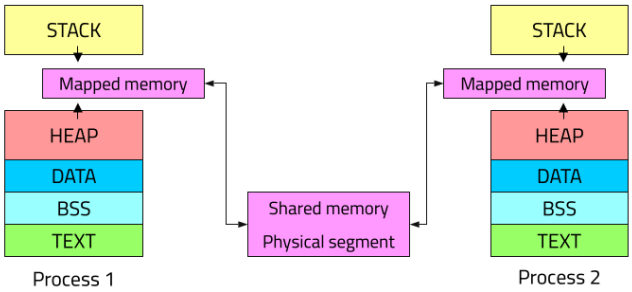


Figure 21: The shared memory

Like message queues, opening/creating shared memory segment is referenced by a name. In Linux, a spacial file is created under `/dev/shm/<name>`.

Opening/creating a shared memory

To open a shared memory region we can use:

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/fcntl.h>
int shm_open(const char *name, int oflag, mode_t mode);
```

Where `name` is a unique name for the shared memory (starting with `/`), `oflag` is the opening flag (e.g., `O_RDONLY`, `O_WRONLY`, `O_CREAT`) and `mode` is related to file permissions to give to the file. This function returns a file descriptor.

After creating a shared memory object, we should specify the size of the special file to be created:

```
#include <unistd.h>
#include <sys/types.h>
int ftruncate(int fd, off_t length);
```

We have that `fd` is the file descriptor to truncate and `length` is the size in bytes.

Mapping the memory

We can map a file or a device to the virtual address space of the calling process:

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,
   int fd, off_t offset);
```

We have that `addr` is the starting virtual address [15], `length` is the size of the mapped segment, `prot` is related to memory protection flags (i.e., `PROT_EXEC`, `PROT_READ`, `PROT_WRITE`, `PROT_NONE`), `fd` is the file descriptor, `offset` is the offset to skip to starting from the beginning of the file descriptor and `flags` is related to the visibility of the updates w.r.t. other processes [16].

[15] When this address is `NULL` we leave the kernel to select it for us.

[16] For instance, we could have:

- `MAP_SHARED`: updates are visible to other processes and carried out through the underlying file;
- `MAP_PRIVATE`: enable copy-on-write, which means that updates are not visible to others and are not carried out through the underlying file;
- etc.

Cleaning stuff

To clean memory regions we mapped, we can use the following:

```
#include <sys/mman.h>
int munmap(void *addr, size_t length);
int shm_unlink(const char *name);
```

We have that `munmap()` deletes the mappings for the specific address range, while `shm_unlink()` removes the shared memory object created by `shm_open()`.

How to synchronize processes

We have seen that it is possible to synchronize two processes using a `wait()` or `waitpid()` approach, but this is clearly very limited. Anyway, synchronization is needed in a multi-process scenario to avoid race conditions on shared resources (shared memory almost always requires synchronization).

For that matter, POSIX provides inter-process *semaphores*, which have a very simple general logic:

- `semaphore_counter = 0`, then `WAIT`;
- `semaphore_counter > 0`, then `PROCEED`.

When a counter can only be either `0` or `1`, it is called *binary semaphore* and behaves similarly to a mutex. We have two main atomic functions:

- `wait()`: block until `counter > 0`, then decrement it and proceed;
- `post()`: increment the counter.

Similarly to pipes, POSIX semaphores can be *named* or *unnamed*. They require the POSIX thread library, which means that we need to use the `-pthread` flag when compiling with `gcc`.

Unnamed semaphore initialization/destruction

To initialize a semaphore we can use:

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

We have that `sem` is the output parameter (i.e., semaphore data structure to initialize), `pshared` establishes whether the semaphore is shared among threads (equals to `0`) or not (different from `0`) and `value` is the initial value. The function returns `0` on success and `-1` on error.

To destroy a semaphore we can use:

```
#include <semaphore.h>
int sem_destroy(sem_t *sem);
```

Named semaphore initialization/destruction

To initialize a named semaphore we can use:

```
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflags);
sem_t *sem_open(const char *name, int oflags, mode_t mode,
unsigned int value);
```

We have that `name` is the POSIX object name, `oflags` are opening flags (`O_CREAT`, `O_EXECL`), `mode` is related to file permissions and `value` is the initial value. Those functions return the pointer to the semaphore object or `SEM_FAILED` in case of errors.

Functions

```
#include <semaphore.h>
int sem_close(sem_t *sem);
int sem_unlink(const char *name);
```

are functions to, respectively, close the named semaphore or remove the named semaphore object.

Synchronization functions

The following are functions used to synchronize processes based on POSIX semaphores:

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *timeout);
```

We have that `sem` is the semaphore object and `timeout` is the time limit to wait if the semaphore counter is equal to zero. The `sem_trywait()` is the non-blocking version of `sem_wait()`. All functions return `0` for success or `-1` in case of error.

5 Lecture 6 - Concurrency III

Terminology

Task has no unique definition, but we can refer to it as a synonym of *thread*: it represents a computation performed by the processor in a sequential fashion. The *scheduler* is the operating system component in charge of establishing the execution order of tasks (called *schedule*). In that context, the ordering algorithm is called *scheduling policy*. The activity of *dispatching*, instead, is about allocating a processor to a specific task.

Introduction to scheduling

Typically, we have a set of tasks that at some point will be executed. In order to do that, they start the *activation* phase and they are moved to the operating system **ready queue**. The scheduler, in turn, picks a task from this ready queue, allowing it to access its resources to be executed. It is likely that this task will be preempted (because, for some reason, it was executing for too long), thus the scheduler de-allocates its resources and picks another task from

this ready queue to be executed. This preemption is performed via a **context switch**. The crucial thing to understand is that when a task is preempted it doesn't mean that it was blocked (i.e., preemption and blocking are two different concepts): a task can voluntarily suspend its execution to wait for I/O data (e.g., keyboard input, file input/output). In that case, the task is put into the **I/O queue** and the CPU will become available to run other tasks.

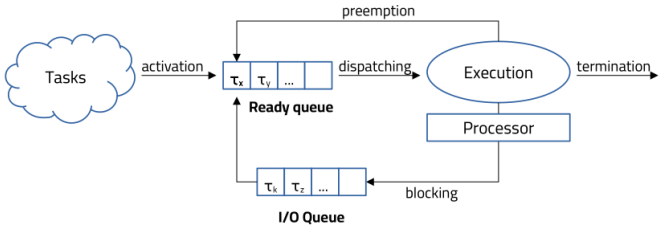


Figure 22: Scheduling conceptual diagram

Task states are not so straightforward, as we have seen during the previous lectures, since, in the Linux kernel, there are a lot more states with different purposes [17]:

[17] See also https://elixir.bootlin.com/linux/v5.19.12/source/include/linux/sched.h#L83.

```
/* Used in tsk->state: */
#define TASK_RUNNING 0×0000
#define TASK_INTERRUPTIBLE 0×0001
#define TASK_UNINTERRUPTIBLE 0×0002
#define __TASK_STOPPED 0×0004
#define __TASK_TRACED 0×0008
/* Used in tsk->exit_state: */
#define EXIT_DEAD 0×0010
#define EXIT_ZOMBIE 0×0020
#define EXIT_TRACE (EXIT_ZOMBIE | EXIT_DEAD)
/* Used in tsk->state again: */
#define TASK_PARKED 0×0040
#define TASK_DEAD 0×0080
#define TASK_WAKEKILL 0×0100
#define TASK_WAKING 0×0200
#define TASK_NOLOAD 0×0400
#define TASK_NEW 0×0800
/* RT specific auxilliary flag to mark RT lock waiters */
#define TASK_RTLOCK_WAIT 0×1000
#define TASK_STATE_MAX 0×2000
```

```
/* Convenience macros for the sake of set_current_state: */
#define TASK_KILLABLE (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
#define TASK_STOPPED (TASK_WAKEKILL | __TASK_STOPPED)
#define TASK_TRACED __TASK_TRACED

#define TASK_IDLE (TASK_UNINTERRUPTIBLE | TASK_NOLOAD)
```

The task model

Let us define the *task model* before talking about scheduling policies. Consider a set of tasks $T = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Those tasks have several parameters:

- The *arrival time* (or *request time*) a_i, which is the time instant at which the task is ready for execution and put into the ready queue;
- The *start time* s_i, which is the time instant at which the execution actually starts;
- The *waiting time* W_i, which is the time spent waiting in the ready queue. In other words

$$W_i = s_i - a_i;$$

- The *finishing time* (or *completion time*) f_i, which is the time instant at which the execution terminates;
- The *computation time* (or *bust time*, or *execution time*) C_i, which is the amount of time necessary for the processor to execute the task **without interruptions**;
- The *turnaround time* Z_i, which is the difference between finishing time and arrival time, namely

$$Z_i = f_i - a_i.$$

Note that Z_i is not necessarily $W_i + C_i$, since in case of preemption or suspension, Z_i contains also the interferences from other tasks and the I/O waiting time (which is the time the task has been interrupted).

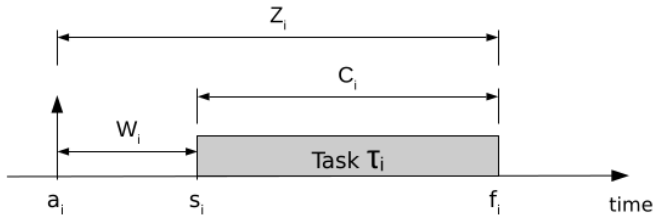


Figure 23: Task's parameters

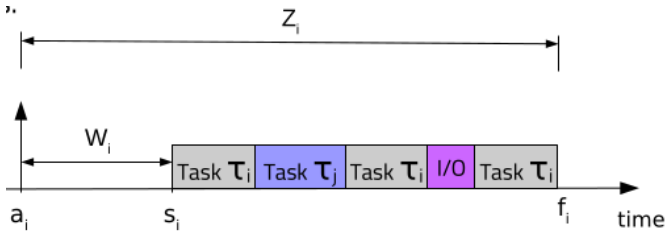


Figure 24: Task's parameters and timeline

Task boundness

Depending on the type of operations dominating the lifetime of a task, we may identify its "boundness" [18] as *CPU-bound* and *I/O bound*. CPU-bound means that the task spends most of its time executing operations:

$$Z_i \approx W_i + C_i,$$

not considering preemptions.

[18] The boundness could be, in general, of three different types: I/O boundness, CPU boundness and memory boundness, which is a more recent term. In this chapter we will consider only the first two types of boundness.

Figure 25: A CPU-bound task

I/O-bound means that the task spends most of its time waiting for I/O operations:

$$Z_i \gg W_i + C_i,$$

not considering preemptions.

Platform model

A computing system is composed of:

- m *processing elements* (PE), namely CPU $= \{\text{CPU}_1, \ldots, \text{CPU}_m\}$;
- s *additional resources*, namely $R = \{R_1, \ldots, R_s\}$.

Each PE, at each time t, is assigned to zero or one task, namely $A_c(\text{CPU}_k, t) = \tau_i$ or $\emptyset$. A task τ_i can execute at time t only if $\exists\, A(CPU_k, t) = \tau_i$.

Moreover, each resource, at each time t, is assigned to zero or more tasks, namely $A_r(R_k, t) = \{\tau_1, \tau_2, \ldots\}$ or $\emptyset$. Depending on the type of the resource, R_k could be exclusive: in that case, $A_r(R_k, t)$ can't contain more that one task. A task τ_i can execute at time t only if $\tau_i \in A_r(R_k, t)\, \forall R_k$ required to run the task.

Problem statement

The problem statement is the following: given a set of n tasks, $T = \{\tau_1, \ldots, \tau_n\}$, a set of m PES, CPU $= \{\text{CPU}_1, \ldots, \text{CPU}_m\}$ (if $m > 1$ we have a **multi-core** or **multi-CPU** system), a set of resources, $R = \{R_1, \ldots, R_n\}$, and an optional set of precedent relationships and constrains (e.g., τ_5 must execute before τ_2), we have to compute an *optimal* schedule (i.e., tasks order) and allocations (i.e., $A_c(R_k, t),\ A_r(R_k, t)$). However, most of these problems (i.e., they reduce to the *knapsack* problem) are NP-complete.

Scheduling metrics

The scheduler aims at optimizing one (or more objectives) depending on the domain in which the processor operates (e.g., embedded systems, servers, laptops). There are several metrics to consider:

- Processor utilization: percentage of time the CPU is busy;
- Throughput: number of tasks completing their execution per time unit;
- Waiting time (average): average time the task spent in the ready queue;
- Fairness: whether tasks have a fair allocation of the processor;
- Overhead: amount of time spent in taking scheduling decisions and context-switches;

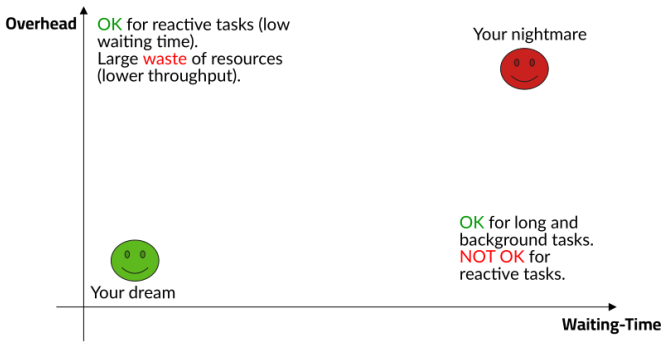


Figure 26: Scheduling metrics and tradeoffs

- Other metrics: energy, power, temperature, reliability, etc.

Usually we want to maximize the CPU utilization, the throughput and the fairness, while we want to minimize the turnaround time, the waiting time, the completion time and the overhead.

Something we definitely don't want to have is **starvation**, an undesirable perpetuated condition in which one (or more) tasks can't e executed due to the lack of resources. Whatever is the objective of a scheduler, a good scheduling algorithm should guarantee that all tasks are served. For instance, in a fixed-priority scheduling policy, the situation in which a starvation could occur is when we have a high priority task keeping busy the CPU and not allowing other lower priority tasks to execute.

Classification of scheduling algorithms

Now, let's try to classify scheduling algorithms based on their characteristics.

We have *preemptive* ones, where running tasks can be interrupted (by the operating system) at any time to allocate the processor to another active task. This is necessary if we need a responsive system. *Non-preemptive* schedulers, instead, are such that, once started, a task is executed until its completion (i.e., *run-to-completion*). Scheduling decisions can be taken only when a task terminates: this guarantees minimum overhead, but this is not good for a responsive system.

Static scheduling means that decisions are based on fixed parameters, whose values are known before task activation. In this

case, strong assumptions are required. *Dynamic* schedulers, instead, are such that their decisions are based on parameters that changes at runtime, during the task execution. They may need a run-time feedback mechanism to adapt their parameters during the execution. Modern general purpose operating systems are usually a mix of the two categories: for instance, task priority is usually known before execution, but the completion time is unknown.

Going further, *offline* schedulers are such that they execute on a set of known tasks before their activation. Their output is the sequence of tasks to execute, called **schedule**. Clearly, they need to be static (since we need to know all the parameters) thus they are very limited, but often necessary when we want to provide some formal guarantees. *Online* schedulers execute at runtime and can schedule new and previously unknown tasks. Modern general purpose operating systems are typically online.

Finally, we have *optimal* schedulers, which are based on algorithms optimizing a given cost function, defined over a task set. The algorithm my be too complex, with an high overhead. *Heuristic* schedulers, instead, are based on heuristic functions and they tend to optimal scheduling, but without any guarantee about achieving it. Generally, they are much faster than optimal algorithms.

Scheduling algorithms

Let's now see the most famous scheduling algorithms and their characteristics.

First-in-first-out (FIFO)

FIFO schedulers are such that tasks are scheduled in order of arrival. They are non-preemptive and they are also known as *first-come-first-served* (FCFS). They are very simple and do not require any knowledge of the processes. As drawbacks, they are not good for responsiveness: long tasks may monopolize the processor and short tasks are penalized (and may even starve).

Shortest Job First (SJF)

SJF schedulers are such that tasks are executed in ascending order of computation time C_i. They are non-preemptive and also known as *shortest-job-next* (SJN). SJF algorithms are optimal non-preemptive ones with respect to the minimization of the average

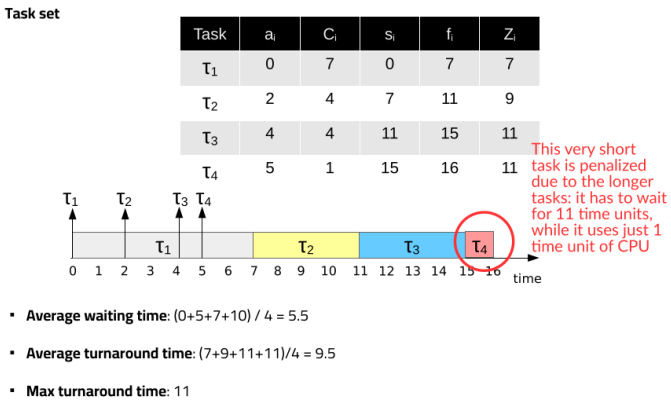


Figure 27: FIFO example

time, but we have a risk of starvation for long tasks and we need to know the execution time C_i in advance, which is not common in general purpose operating systems.

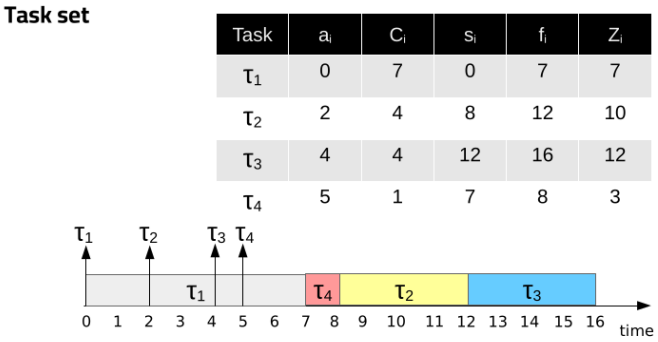


Figure 28: SJF example

Shortest Remaining Time First (SRTF)

SRTF is the preemptive variant of SJF and uses the remaining execution time instead of C_i to decide which task to dispatch. For this reason, it improves the responsiveness for all tasks compared

to SJF. As drawbacks there is a risk of starvation for long tasks and, moreover, we need to know C_i in advance.

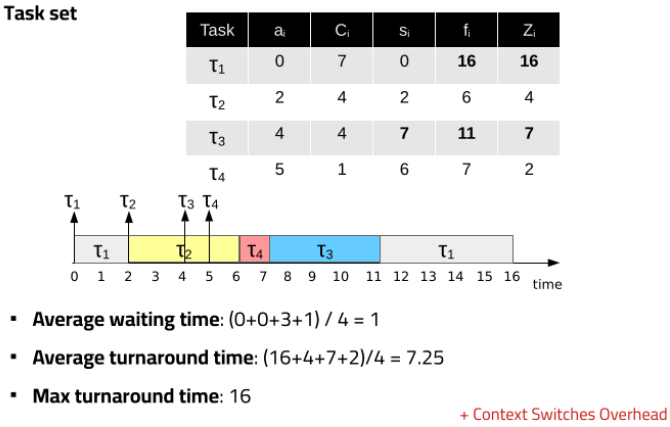


Figure 29: SRTF example

Highest Response Ratio Next (HRRN)

HRRN is a non-preemptive scheduling algorithm and it works in the following way: it selects the task with the *highest response ratio*:

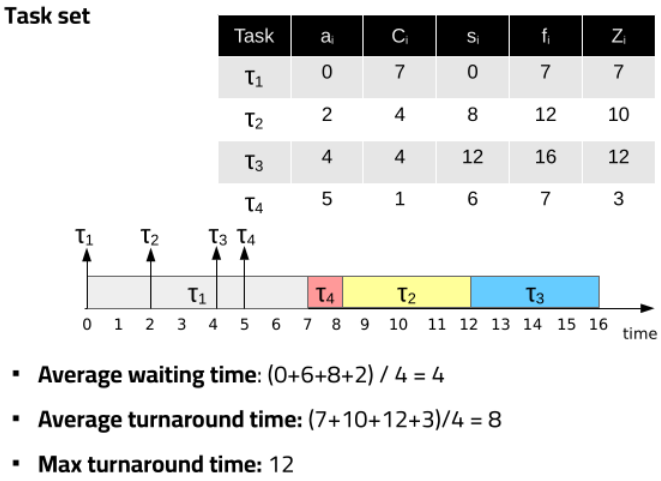
$$\text{RR}_i = \frac{W_i + C_i}{C_i}. \tag{1}$$

This algorithm prevents starvation with respect to SJF, but again we need to know C_i in advance.

1. SJF vs. HRRN Let us compare, for instance, SJF and HRRN with a different task set. When we have small tasks, the longest one risks starvation in SJF, while in HRRN it increases its response ration, gaining priority over the short tasks.

Round Robin (RR)

RR is a *preemptive* scheduling algorithm such that tasks are scheduled for a given time quantum q (also called *time slice*). When the time quantum expires, the task is preempted and moved back to the ready queue.



- **Average waiting time**: (0+6+8+2) / 4 = 4
- **Average turnaround time:** (7+10+12+3)/4 = 8
- **Max turnaround time:** 12

Figure 30: HRRN example

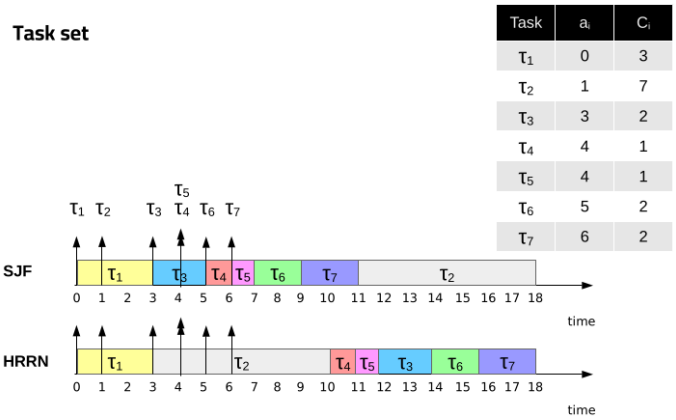


Figure 31: SJF vs. HRRN with an example

In such a setting, the maximum waiting time can be computed in the following way

$$W_i^{\max} = (n-1) \cdot q, \forall \tau_i. \tag{2}$$

Moreover, there is no need to know C_i in advance. RR is good to achieve the fairness and responsiveness goals and no starvation is possible. As disadvantage, we have that the turnaround time is worst than SJF.

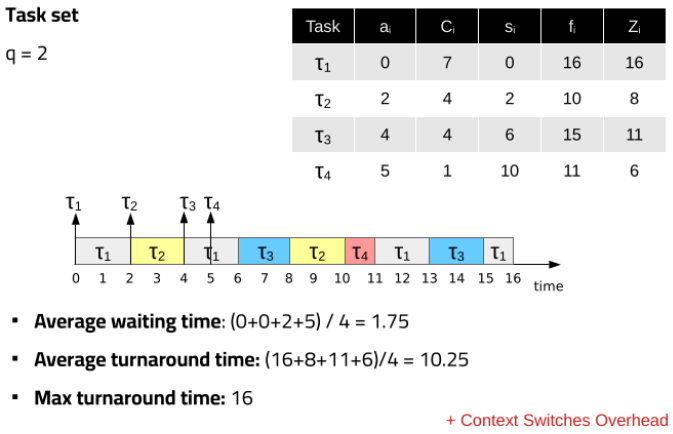


Figure 32: RR example

Note that we have a lot of context switches, thus depending on the system we are int it could penalize the execution time a bit.

In Linux, to inspect the quantum slice allocated for RR policies we can type the following:

```
$ cat /proc/sys/kernel/sched_rr_timeslice_ms
90
```

1. Choosing the quantum value The activity of choosing the RR quantum value is critical: choosing a ”long” quantum will make the scheduler to tend to a FIFO one and will favor CPU-bound tasks. This will imply a low overhead, since there will be less context switches. Choosing a ”short” quantum, instead, will reduce the average waiting time, favoring I/O-bound tasks. This would be good for responsiveness and fair scheduling. However, we would have a high overhead, since there will be more context switches.

6 Lecture 7 - Concurrency IV

Priority-based scheduling

The *priority* P_i is a task parameter through which we can specify the importance of a task, and it can be **fixed** (i.e., known at design-time) or **dynamic** (i.e., changes at run-time). The priority is usually expressed using an integer value: the lower the integer value, the higher the priority (and vice versa).

We define as *multi-queue scheduling* a scheduling system such that, for each queue, we can specify a different scheduling algorithm (e.g., RR or FCFS). The first task to schedule is picked from the topmost non-empty queue, which also has the highest priority. Another crucial aspect is that tasks can't be moved from one ready queue to another. Finally, the queue scheduling is *preemptive*.

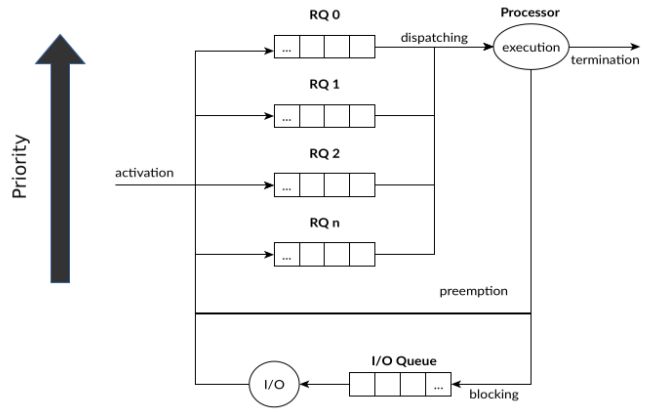


Figure 33: Multi-level queue scheduling scenario

In such a scenario, clearly there is a risk of **starvation**: whilst highest priority queues are populated by new tasks, the scheduling of those tasks belonging to lower priority queues is delayed.

Priority-based with Round Robin policy

Let us consider a priority-based multi-level queue scheduling assigning RR to each queue with different quantum slices. Suppose that the priority is selected on the workload type as follows: CPU-bound tasks have a low priority (therefore a high quantum value), I/O-bound tasks have high priority (therefore a low quantum value) [19]. This priority scheme guarantees **responsiveness**, but starvation

[19] There are several ways to determine if a task is CPU-bound or I/O-bound. For instance, we could distinguish it using information provided both by the user or by the program itself (using a rum-time feedback mechanism), or we can use a *multi-level feedback queue* scheduling.

is always a concrete risk, as said previously.

Multi-level feedback queue scheduling

This scheduling model works like the previous multi-level RR scheme, but the priority is now **dynamic**: the new/activated task is moved to the highest priority queue (i.e., lowest quantum value), then if the quantum of the running task expires, the task is moved to the next queue with lower priority (i.e., higher quantum value). This means that CPU-bound tasks are progressively moved in queues with longer time quantum.

Anyway, also in this case we didn't solved the problem of starvation. There are basically two ways to solve starvation in a multi-level scheduling scenario:

- Time slicing;
- Aging.

1. Multi-level feedback queue scheduling with time slicing In this scenario, each queue gets a **maximum** percentage of the available CPU time it can use to schedule the task, which determines a *time quota*. If the time quota expires, the remaining tasks in the queue are skipped, and the scheduler start picking tasks from the next (lower priority) queue.

   For instance, we could have the following:

Queue	Quota	Period
Q_1	80 ms	100 ms
Q_2	15 ms	100 ms
Q_3	5 ms	100 ms

   So, Q_1 has the highest priority, but it can't schedule tasks for more than 80 ms. The value $\sum_i \text{quota}_i$ can be larger than the period but it is guaranteed to have no starvation if $\sum_i \text{quota}_i \leq \text{period}$. However, we can still have starvation due to the scheduling policy of one of the queues.

   a) Multi-level queue scheduling: the case of the Linux scheduler The Linux scheduler is a *multi-level queue scheduler* with several queues with a quota and period defined in the following files:

```
quota = /proc/sys/kernel/sched_rt_runtime_us
period = /proc/sys/kernel/sched_rt_period_us
```

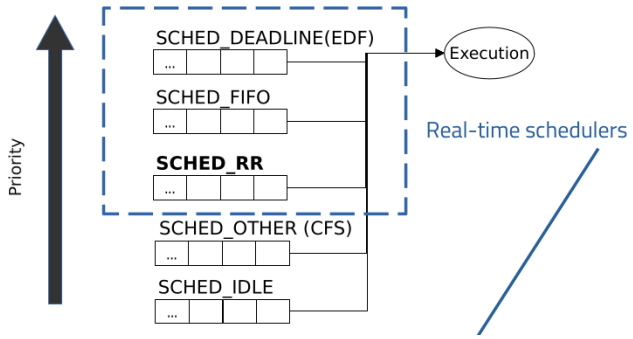


Figure 34: The Linux scheduler

The scheduling policy is set by the user/system administrator. Note that if `IDLE` is chosen for a task, the latter may starve.

2. Multi-level feedback queue with aging In this case, the priority of the task is increased dynamically, as long as it spends time in the ready queue (i.e., it gets older). The idea is that, at a certain point, the task will be served. The aging time is basically a defined period after which a task will gain priority and jump into a more prioritized queue. This approach prevents a task from being indefinitely postponed by new coming higher priority tasks. In other words, it avoids starvation.

Multi-processor scheduling

Suppose that we have a multi-processor machine and we want to design a scheduler taking into account the underlying architecture. The scheduler, indeed, must choose between the task to execute and the processor to which such task is assigned.

Multi-processor scheduling is hard

This scenario introduces some new problems, such as the following:

- Task synchronization may occur across parallel executions;

- Difficulties in achieving a high-level of utilization of the whole set of processors (or CPU cores);
- Simultaneous (not only concurrent) access to shared resources, which is sometimes hierarchical and multi-level (e.g., cache memories).

This is sufficient to prove that multi-processor scheduling is a *very hard* problem. For instance, in a modern architecture (with a L1 exclusive cache and a L2/L3 shared cache) there could be problems related to loading pages from main memory to cache: tasks may interfere one with each other, thus slowing down their execution [20]. Decide on which processor a task should be executed is crucial to avoid such a scenario: make tasks colliding being executed on cores having a L2 not shared between them is key.

[20] The reason why this happens is that some memory locations may end up in the same cache line, colliding.

Design choices

There are basically two main design choices:

- Single queues vs. multiple queues;
- Single scheduler vs. multiple per-processor schedulers.

1. Choosing the number of queues A *single queue* approach is such that all the ready tasks wait in the same **global queue**, as shown in figure.

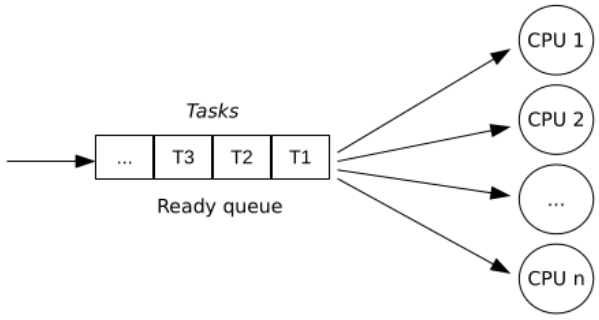


Figure 35: Single queue approach

This implies a simple design and it turns out to be good for fairness and for managing the CPU utilization. However,

it has obvious problems of scalability: the scheduler runs in any processor and requires a synchronized version of the ready queue (e.g., using a semaphore/mutex).

Using *multiple queues*, instead, requires to have a ready queue for each processor and each of these have two options:

a) To keep, in turn, several queues one for each priority;
b) To keep a data structure to efficiently reorder tasks inside a single queue and schedule them accordingly.

Moreover, for each queue we could use different scheduling policies.

This solutions sounds like a more scalable approach, but also has potentially more overhead (i.e., more data structures to handle). Using a multiple queue approach it could be easier to exploit data locality (i.e., processor affinity) and there is the possibility to adopt a single global scheduler or per-CPU schedulers. Finally, this approach requires *load balancing*.

a) Load balancing Unbalanced ready queues have a very negative impact for several reasons: from a **CPU utilization** perspective, processors may be idle (due to empty queues), while other queues may have a lot of waiting tasks, from a **performance** perspective, waiting times and response times can be reduced by moving tasks in a different queue (e.g., with a lower number of tasks). Unbalanced ready queues have a negative impact also on **thermal management**: balancing the ready queue levels the temperature distribution, and this impacts power consumption, energy efficiency and reliability.

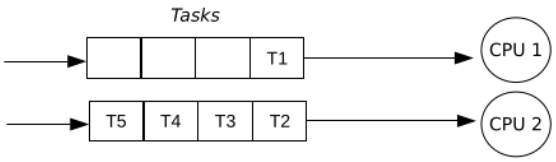


Figure 36: A scenario in which load balancing is needed

b) Task migration Load balancing is typically performed via task migration: tasks are moved to a different queue

(usually containing less tasks) and this as a consequence on the cache overhead and interference, as we have already seen. There are two possible implementations:

- *Push model*: a dedicated task periodically checks the queues' lengths and moves tasks if balancing is required;
- *Pull model*: each processor notifies an empty queue condition and picks tasks from other queues.

For instance, **work stealing** is an example of pull model-based approach: each per-CPU scheduler can "steal" a task from other queues in case of an empty one. This is scalable in theory, but we need to protect the concurrent access to the ready queue with locks.

c) Hierarchical queues *Hierarchical queues* is an approach involving multiple schedulers. A global queue dispatches tasks in local ready queue with a hierarchy of schedulers. In this way, we can have a better control over the CPU utilization and load balancing, along with a good scalability. Of course, this model is way more complex to implement.

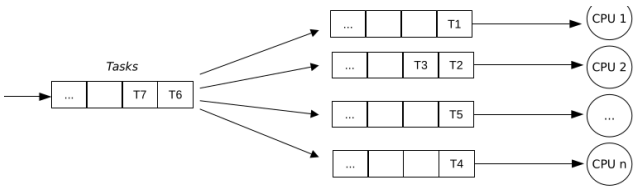


Figure 37: Hierarchical queues scheme

7 Lecture 8 - Concurrency V

Introduction to concurrency

Concurrency is a situation in which a program composed by activities such that one of them **can** start before the previous has finished. In other words, if that happens, the program would be correct even if those instruction execute in an overlapped fashion.

Consider that concurrency is a "possibility". Different part or units of a program can be executed in an overlapped way *without affecting the final outcome*, but ensuring to have different program

counters, stacks and registers for variables. In particular, ”can” is not equal to ”must”: we can execute concurrent activities without overlapping them. However, they will not meet expected deadlines or response times and this can be impossible when these activities are infinite loops.

One way of overlapping these activities is through the **threading model**: the operating system allows to stop one activity, save its state and start from another one by restoring its state.

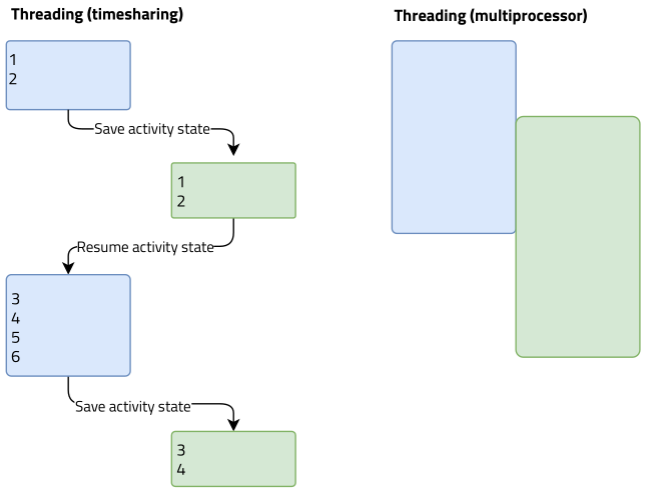


Figure 38: Threading model as concurrency paradigm

Besides the well-known thread execution model, we could want its benefits but in a more real-time scenario [21], so there is another execution model family called **lightweight execution models**: we want concurrency but we also to avoid the overhead resulting from context switches. Lightweight execution models are typical of languages supporting *co-routines* and *generators*. In case we have an application based on external events (e.g., server), we can use a subset of lightweight execution models: *continuation passing* (i.e., callbacks - e.g. Nodejs) and languages supporting asynchronous wait [22].

Historically, operating systems supported through a uniform interface the thread execution model. Now, it is more and more common to have support to other mechanisms as well, such as event-based I/O.

[21] What we do not typically want is to waste time in context switching.

[22] C++ recently adopted asynchronous wait, but Java and Rust have it for a long time now.

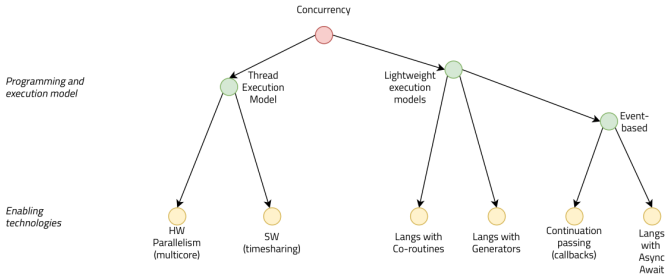


Figure 39: Concurrency taxonomy

Hardware parallelism and its boundaries

Software parallelism is crucial nowadays, because of hardware parallelism's limits: the frequency of operations (in MHz) hit a wall, mainly because of power consumption (which impacts the cost of the chip itself, because of cooling mechanisms). Therefore, single-threaded performance reached a plateau, nonetheless we have more and more transistors [23].

Originally, frequency was used as the main knob to obtain more performance. Increasing frequency implies also an increase in power consumption and, since 20 years, as shown in picture, we reached a power threshold which makes it impractical to increase even more.

[23] According to the Moore's law the number of transistors doubles every two years.

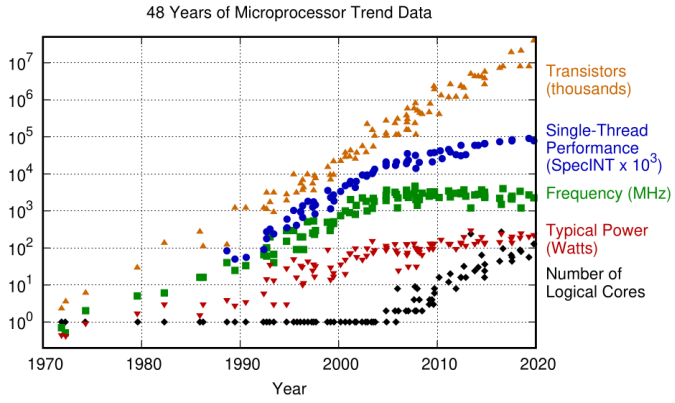


Figure 40: Reaching the power wall

Micro-architectural techniques allowed for some time to increase the single thread performance by better exploiting avail-

able transistors while keeping everything within its power envelope. Nevertheless, the more transistor are available the more we have resources to be exploited by multiple threads the more an operating system becomes important. However, having a parallel program may lead to some issues, as we are going to see.

Concurrency issues and prevention

When we have a concurrent program, we can characterize it using two concepts:

- *Safety* (i.e., correctness);
- *Liveness* (i.e., progress).

The idea of liveness is that, at some point, our program is going to terminate, reaching a final state (this also means that our program never blocks in a state but can always progress). However, our program can change states and never block only if we do not have deadlocks.

Deadlocks are not the only problem, but there are others like **priority inversion** (a high priority thread is delayed by a lower priority one) which surely needs some attention.

Safety-related issues can be categorized in several ways, as we are going to see: they are mainly non-deadlocks bugs and the majority of them are **atomicity** and **order violation** (i.e., **data races**).

Data races

Data race is the first condition that can impact on safety: two instructions of two different threads that access the same variable should synchronize to avoid conflicts. In other words, when our program's substantive behavior depends in an uncontrolled way from the memory model an the timing of involved threads, if this is undesirable, we are in front of a data-race-like problem. The actual data race happens whenever we have two instructions, say `a` and `b` (in different threads), where at least one of them is a `write` and when there is nothing that enforces either `a` $<_m$ `b` or `b` $<_m$ `a` [24].

[24] We are not sure on the order in which the two instructions are going to execute.

1. Atomicity violation *Atomicity violation* also impacts on safety. Let us consider the following block of code:

```
// Thread 1::
if (thd->proc_info) {
  fputs(thd->proc_info, ... );
}
...
// Thread 2::
thd->proc_info = NULL;
```

In the example above, without any mutual exclusion, we might have a sequence of instructions being scheduled such that the nullification of the pointer happens before the other thread reaches the `if` condition.

2. Order violation Let us consider the following example:

```
void thread1() {
   ...
  mThread = PR_CreateThread(mMain, ... );
   ...
}
...
void thread2() {
  mState = mThread->State;
}
```

Order violation is such that, if we don't ensure that `thread1` assigns the value before `thread2` accesses it.

Deadlocks

A deadlock is a situation in which a series of threads can't take any action because of a some sort of circular dependency between those. More precisely, the situation is such that, no task $\tau_i \in T = \{\tau_0, \ldots, \tau_n\}$ can take action because it is waiting for another task $\tau_j \in T$ to take action.

However, in a concurrent program setting, we have a deadlock (i.e., the program can't progress towards its end) also for the following conditions:

- **Mutual exclusion**: two threads can't act on the same resource at the same time;
- **Hold-and-wait**: threads hold resources allocated while waiting for additional resources;

- **No preemption**: resources (e.g., locks) can't be forcibly removed from threads that are holding them;
- **Circular wait**: there exists a circular chain of waiting threads [25].

[25] The simplified definition of deadlock coincides with a circular wait, but there are also other ways in which a deadlock may occur.

1. Mutual exclusion Mutual exclusion can be prevented using certain data structures. In particular, there are data structures to be used for avoiding the use of locks but without give up on mutual exclusion. Let us consider the following example:

```
void add_atomically(int *value, int amount) {
  int old;
  do {
    old = *value;
  } while (_atomic_compare_xchg(value, old, old + amount) == 0);
}
```

   This function uses a procedure called `compare_and_xchg()`, which checks in an atomic way a comparison between two values and updates it when the two compared values are equal.

   We can also use the same approach to insert a new value into the head of a linked list:

```
void insert(int value) {
  node_t *n = malloc(sizeof(node_t));
  assert(n ≠ NULL);
  n->value = value;
  do {
    n->next = head;
  } while (_atomic_compare_xchg(&head, n->next, n) == 0);
}
```

2. No preemption and hold-and-wait Both no preemption and hold-and-wait can be prevented: through appropriate APIs, a task can voluntarily release the resource if the acquisition of further ones fails. Given a programming interface that allows to write tasks in the following way:

```
// TASK 1
lock(M1);
if (!try_lock(M2)) {
  release(M1);
}
else {
  update(R1);
  update(R2);
  release(M1, M2);
}

// TASK 2
lock(M2);
if (!try_lock(M1)) {
  release(M2);
}
else {
  update(R1);
  update(R2);
  release(M1, M2);
}
```

We can use APIs (such as `try_lock()`) to check whether also other resources *can* be locked as well. Thus, threads can be programmed in such a "gentle" way to release resources whenever they need another resource already locked by someone else. This is a way to avoid the *no preemption* condition.

a) POSIX `pthreads` In the POSIX `pthread` library, we have the `pthread_mutex_trylock()` function which does exactly what we saw before: either grabs the lock (if available) and returns success or returns an error code indicating that the requested lock is held by another thread. In particular, we would have:

```
top:
pthread_mutex_lock(L1);
if (pthread_mutex_trylock(L2) ≠ 0) {
  pthread_mutex_unlock(L1);
  goto top;
}
```

However, there is a problem with that scenario: if we have two threads checking for the other one in the

similar ways, they wouldn't progress towards the end. There would be situations when they just attempt to acquire the second lock without succeeding at all (i.e., circular wait condition).

3. Circular waits Let us introduce a representation of the circular wait as follows: assume there are two philosophers sitting around a table, and assume that they share two forks (suppose that they need all the two to eat). If each philosopher happens to grab the fork on their left before any philosopher can grab the fork on their right, each will be stuck holding one fork and waiting for another, forever.

   The problem can be solved as follows: introduce *ordering* so that no cyclical wait arises [26]. When total lock ordering may be difficult to achieve (given the size of the project), **partial ordering** can be used.

   Some operating systems have lock order rules to be complied with (e.g., xv6, linux). Linux has also validation/debug features [27] to detect circular locks, which have to be enabled at compile time.

4. Priority inversion To understand how *priority inversion* works, let us introduce the following example: assume to have a preemptive scheduler which schedules always the highest priority task and three tasks having priorities $\pi_1 > \pi_2 > \pi_3$, where T_1 and T_3 share a resource with a critical section.

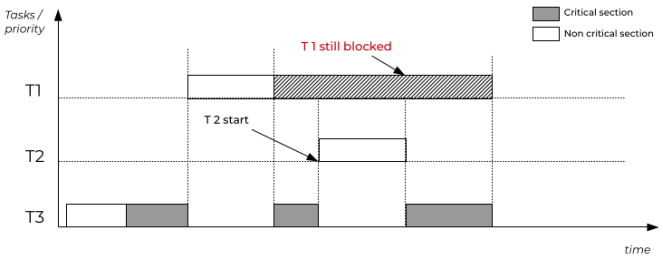


Figure 41: The priority inversion scenario

   In such a scenario, we have a lower priority thread, which is T_3, locking a resource and making T_1, a higher priority thread, to miss some deadlines, even if it should be scheduled as soon as possible due to its priority.

[26] The example of the dining philosophers can be solved by establishing an order such that, the first philosopher takes the left fork first, and the second philosopher takes the right fork first: `T_1: (l, r); T_2: (r, l)`.

[27] To better understand how the strategy used by Linux works, see also https://lwn.net/Articles/185666/, https://www.kernel.org/doc/html/v4.13/kernel-hacking/locking.html.

a) Solving the priority inversion There are three main techniques used to solve priority inversion:

- *Highest Locker Priority* (HLP);
- *Priority Inheritance* (PIP);
- *Priority Ceiling* (PCP).

They are all based on changing the priority of the tasks w.r.t. the nominal one.

Let us consider $R_0, \ldots, R_m$ shared resources guarded by semaphores $\Sigma = \{S_0, \ldots, S_m\}$, P_i and p_i as, respectively, the nominal and active priority value of task T_i (lower values indicates higher priority), $z_{i,k}$ as the critical section in task T_i accessing R_k, and finally $\varsigma_i \subseteq \Sigma$ as a set of semaphores used by T_i.

i. Highest locker priority protocol The idea of HLP is to avoid preemption during the execution of any critical section by raising the priority of the task accessing R_k to

$$p_i(R_k) = \min_i\{P_i | T_i \text{ using } R_k\}. \tag{3}$$

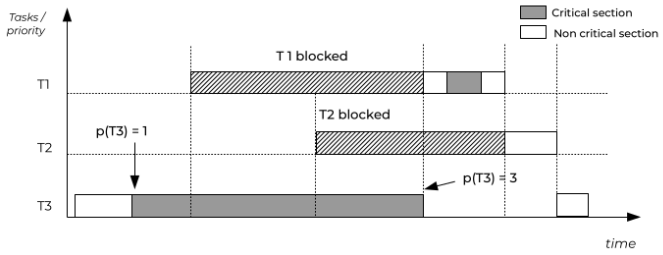


Figure 42: Highest locker priority protocol example

ii. Priority inheritance protocol The idea of PIP is such that, when T_i enters $z_{i,k}$ and R_k is already held by T_j, the latter assumes the active priority of T_i, namely

$$p_j(R_k) = min_i\{P_j, P_i | T_i \text{ blocked on } R_k\}. \tag{4}$$

iii. Priority ceiling protocol A *priority ceiling* of a semaphore S_k is the highest priority among the tasks that can

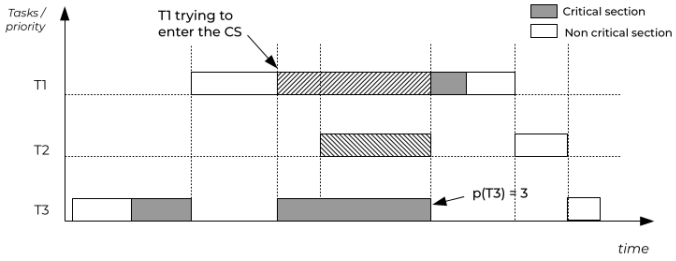


Figure 43: Priority inheritance protocol example

lock it:

$$C(S_k) = \max_i P_i | S_k \in \varsigma_i. \tag{5}$$

According to the *priority ceiling protocol*, a task T_i is allowed to enter a critical section only if its priority is higher than all the priority ceilings of the semaphores currently locked by other tasks.

b) Priority inversion avoidance in POSIX There are ways to define which approach to use with POSIX mutexes:

- `PTHREAD_PRIO_NONE`: no protocol;
- `PTHREAD_PRIO_INHERIT`: priority inheritance;
- `PTHREAD_PRIO_PROTECT`: priority ceiling [28].

In particular, using priority inheritance:

[28] We can use `pthread_mutex_setprioceiling` to set the ceiling priority.

```
int main( ... ) {
  pthread_mutex_t mutex;
  pthread_mutexattr_t mattr;
  pthread_mutexattr_setprotocol(&mattr, PTHREAD_PRIO_INHERIT);
  pthread_mutex_init(&mutex, &mattr);
  return 0;
```

8 Lecture 9 - Concurrency VI

Introduction to futexes

Linux locks are based on a concept called *futex* (i.e., fast user level lock). The idea is to avoid as many system calls as possible [29]. We also want to avoid thundering herd problems in which all threads are woken up at the same time. Note that, whenever we use POSIX

[29] The reason why is better to avoid system calls is that, doing this, we also avoid state saving overhead and context switching, which are costly in general.

NPLT or PTHREAD libraries we use this kind of locks (i.e., they are built-in).

Futexes architecture

The idea is that on traditional kernels we always need to go directly in kernel mode to obtain a lock (and eventually wait in a queue). From a futex perspective, instead, we could have an uncontended case (no threads are overlapping in the use of the lock), and a contended case. In the former, there won't be any syscall, but everything is resolved in the runtime environment, in the latter, instead, a system call interaction is required.

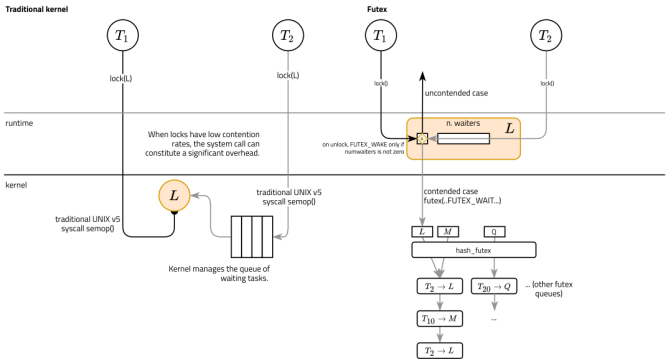


Figure 44: Futexes architecture

The lock structure is moved into the runtime itself, as a 32 bit integer, divided into 31 bits encoding the number of waiters plus a bit indicating if the resource is locked or unlocked.

When a task is redirect into the kernel and the lock representation's first bit is set to 1, it is put inside a queue and waken up later. The kernel maintains a data structure containing for each task the lock this task is waiting for.

Locking with futexes

Let us inspect the function used to lock resources using the futex mechanism:

```
void futex_based_lock(int *mutex) {
  int v;
  if (atomic_bit_test_set(mutex, 31) == 0) return;
```

```
  atomic_increment(mutex);
  while (1) {
    if (atomic_bit_test_set(mutex, 31) == 0) {
      atomic_decrement(mutex);
      return;
    }
    v = *mutex;
    if (v >= 0) continue;
    futex(mutex, FUTEX_WAIT_v); /* sleeps only if mutex still has v */
  }
}
```

In this function, the Linux runtime tries as much as possible not to call the kernel, in fact it checks for a bit atomically, and then checks it again in a loop (seeing if in the meantime something changed). The `if (v >= 0)` control checks if the first bit of the integer is either set or no (remember that the first bit is the sign in the binary representation).

The unlock function, instead, is structured as follows:

```
void futex_based_unlock(int *mutex) {
  if (atomic_add_zero(mutex, 0x80000000))
    return;
  futex(mutex, FUTEX_WAIT, 1); // wake up only one thread
}
```

This approach is used not only for mutexes but also for conditions variables. In a broadcast scenario, all the threads waiting for a condition variable in a `COND_WAIT` state, would be waken up. With futexes, instead, Linux tries to avoid the thundering herd problem.

Event-based concurrency

Event-base concurrency occurs when multiple activities has their progress strictly related to external events. This paradigm is heavily used in GUI-based applications and internet servers (e.g., Nodejs).

The idea is that there is a wait of something (i.e., an ”event”) to occur (or to be ”posted” by an external entity). When this event occurs, there is a check on what type of event arrived, identifying which activity it belongs. Then, the routines do the *small* amount of work it is required [30]. Finally, the routine is put into a *rinse and repeat* state.

[30] This work may include issuing I/O requests, or other events for future handling, etc.

Different types of concurrent activities

In a more traditional event-based concurrency scenario, we have applications executing instructions and then blocking (or polling) for a certain amount of time. We can use `pthreads` to develop applications such that a thread is spawned for each activity that needs to be handled. However, using `pthreads` to handle activities in a web server may not be the best idea, since we should avoid context switches as much as possible.

There are basically two techniques to avoid context switches:

- Event loops;
- Callbacks.

Event loops is a technique involving a single thread, giving the control to a function repeatedly checking for external events (i.e., the *event loop*). This function then should restore the state of the activity itself once the waiting is done (e.g., Javascript has the concept of *closure*). The modern incarnation of this behavior are callbacks, a closure able to restore the state of the activity.

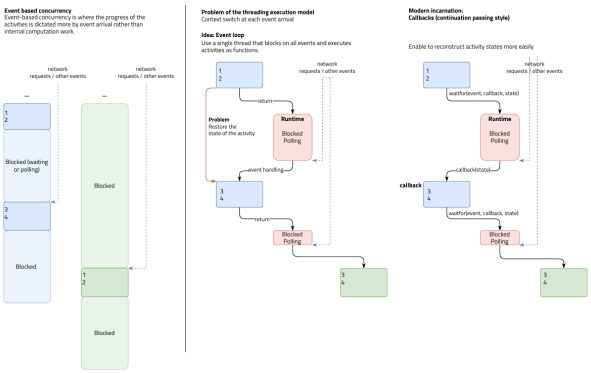


Figure 45: Event-based concurrency techniques

1. Event loop In an event loop scenario, when a handler processes an event, it is the only activity taking place in the system. Thus, no locking is needed: this task can modify shared data without worrying about other activities. In this way, we do not have context switch overhead, except for resuming each activity from where it stopped.

The code associated to an event loop could be the following:

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

Let's now focus on events based on network requests. On UNIX/Linux we have the `select()` and `poll()` APIs, which check whether there is any incoming network I/O that should be attended to. For example, `select()` examines the first `nfds` I/O descriptor sets checking if there is data ready (to read), or if the write buffer still has space, or if there is an exceptional condition pending:

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

Consider now the following usage example:

```
void main() {
  while(1) {
    int wfd[] = {10, 20, 30};
    fd_set readFDs;
    FD_ZERO(&readFDs);
    for (int i = 0; i < 3: i++)
      FD_SET(wfd[i], &readFDs);
    int rc = select(3, &readFDs, NULL, NULL, NULL);
    for (int i = 0; i < 3; i++) {
      if (FD_ISSET(wfd[i], &readFDs)) processFD(wfd[i]);
    }
  }
}
```

We have three streams (i.e., `wfd`) from which we are waiting for an event, then we create a set of descriptors, specifying that inside them we want to read the streams. The `select()` will block and returns when there is one or more request ready on one of the incoming streams specified before. Finally, for each of the stream we check whether this still has something inside, if so we invoke the `processFD()`.

The problem with this approach is that `processFD()` can't block on any activity (e.g., in Javascript callbacks we can't have infinite loops). If an event handler issues a call that blocks, the entire server will do just that: it blocks until the call completes. When the event loop blocks, the system sits idle, and thus this is a huge potential waste of resource [31]. To solve this, we could run blocking file I/O operations in a thread pool and make the `processFD()` register a callback for when the data is ready (libuv creates a nice abstraction on this). We could also use Linux/POSIX `AIO` library, which provides native functions for launching asynchronous file I/O operations and be notified in several ways.

[31] This might happen if the `processFD()` function invokes an e.g., `fread()`.

Using `AIO`, one state machine is created dynamically for each client, which, in turn, can send an action char `(0, ... , 3)` to change a client-specific state `(a, ... , d)` in the server. The state transition table is the following:

Current	0	1	2	3
a (initial state)	a	b	c	d
b	a	a	b	d
c	a	a	c	d
d (exit state)	d	d	d	d

Linux kernel space concurrency

There are basically three main characteristics of the Linux kernel that might introduce concurrency within the activities performed:

- *Kernel preemption*;
- *Interrupts*;
- *Multiple processors*;

Kernel preemption is about multiple threads allowed to share the same resource in the kernel, while interrupts are another aspect of concurrent activity. Interrupts can occur asynchronously at almost any time, interrupting the currently executing code in kernel mode. If the interrupt and the interrupted task are using the same resource, then we must regulate access. Code that is safe from concurrent access from an interrupt handler is said to be **interrupt-safe** [32]. The last source of concurrency is having multiple processors with tasks running at the same time.

[32] When an interrupt arises, the program flow goes into a state called **interrupt context**, given that the original application started from **process context** and might end up in **kernel context**.

Multi-processing

Multi-processing support implies that kernel code must simultaneously run on two or more processors. Code in the kernel, running on two different processors, can thus simultaneously access shared data at exactly the same time. Multi-processing is also called, for this reason, *true concurrency*.

Code that is safe from concurrency on symmetrical multi-processing machines is *SMP-safe*. Note that writing code that is SMP-safe is really tricky, since processors do not have the same view of the memory.

Kernel preemption

Remember that, on certain conditions, we can have also nested interrupts. In *preemptive kernels*, we can switch to another task instead of resuming a kernel activity that was executing before the interrupt came. In other words, even kernel activities can be preempted and we could have context switches also between kernel tasks before the preempted one finished [33]. The idea is that, in preemptive kernels, context switches may happen in places different from the end of process context, so to switch to other tasks different from the preempted one (maybe because of a higher priority) or even to kernel tasks.

[33] Preemptive kernels also require special care to keep kernel code preempt-safe, without leading it in incostitent states. See also https://www.kernel.org/doc/Documentation/preempt-locking.txt.

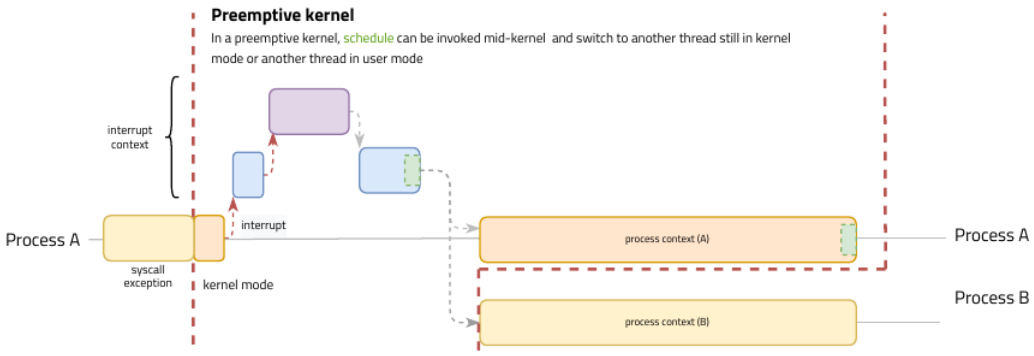


Figure 46: Preemptive kernel scenario

1. Preemption points From its 2.6 version, the Linux kernel became preemptive. This comes with the definition of *preemptive points*, which are the following:

   - The **end** of **interrupt/exception handling**, when `TIF_NEED_RESCHED` flag in the thread control block has been set (i.e., *forced process switch*);
   - When a task in the kernel **explicitly blocks** (which results in a call to `schedule()` - i.e., *planned process switch*) [34].

   [34] However, it is always assumed that the code explicitly calling `schedule()` knows it is safe to reschedule.

   The `TIF_NEED_RESCHED` flag is defined as follows:

   ```
   #define TIF_NEED_RESCHED 1 /* rescheduling necessary */
   ```

   Moreover, it is set/unset in the `task_struct` in the following way:

   ```
   static inline void set_tsk_need_resched(struct task_struct *tsk)
   {
   set_tsk_thread_flag(tsk,TIF_NEED_RESCHED);
   }

   static inline void clear_tsk_need_resched(struct task_struct *tsk)
   {
   clear_tsk_thread_flag(tsk,TIF_NEED_RESCHED);
   }
   ```

2. Preemption count In the Linux kernel, the `preempt_count` variable is increased for each coming interrupt. As soon as this variable is not equal to zero, the kernel can't give control to other threads. When this variable is zero, since we are assuming to have a preemptive kernel, the latter may give control to another task and not to the one executing before the first interrupt arrived. This strategy is typically employed to allow the kernel to manage nested interrupts correctly, knowing how deep the current chain of interrupts is.

3. The real-time patch (`PREEMPT_RT`) The key point of the `PREEMPT_RT` real-time patch is to minimize the amount of kernel code

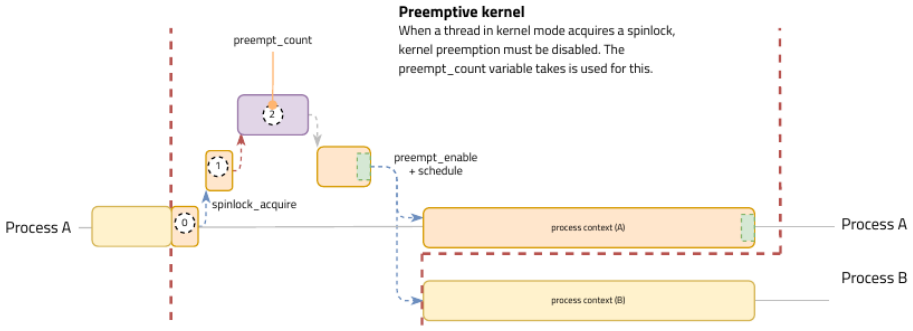


Figure 47: `preempt_count` usage

that is non-preemptible, while also minimizing the amount of code that must be changed in order to provide this added preemptibility. However, Linux will never provide the lowest latencies as possible, since it wasn't designed with this in mind. This means that it is not very suitable for safety critical applications.

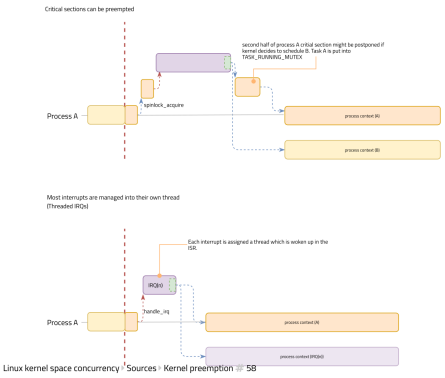


Figure 48: PREEMPT$_{RT}$ structure

9 Lecture 10 - Concurrency VII

More on preemption

To have additional responsiveness in a RT scenario, the Linux kernel has the `PREEMPT_RT` patch, which maximizes the preemptable code within the kernel. In this case, critical sections, even if in kernel mode, can be preempted: a task in kernel mode that has previously acquired a lock on a critical section can be preempted, although the lock is not released yet, and the control, once the interrupt handler finishes, can be given to another task.

The `PREEMPT_RT` must be managed carefully

This case must be certainly managed carefully, since interrupting critical sections may be dangerous, depending on what the newly scheduled task will do (remember that we still have a lock taken by a preempted task). When the critical section is postponed, which means that a new task has been scheduled before the one that acquired the lock released it, the latter is put into the `TASK_RUNNING_MUTEX` state.

Threaded IRQs

When a interrupt comes in process context, the interrupt is managed in a way such that some work is scheduled to be done in process context later on. Thus, each interrupt is assigned to be executed in its own thread.

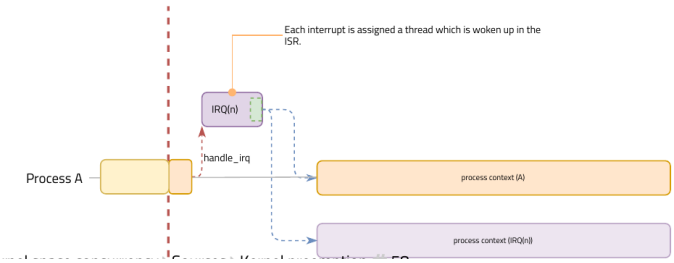


Figure 49: The IRQ mechanism

Locking techniques

Spinlocks

Spinlocks are the most used technique when dealing with multiple processes: each of them can be either `1` or `0`. An example of code using spinlocks could be:

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

The `spin_lock()` operation is an active loop, checking repeatedly if the requested lock becomes free. In the Linux kernel [35], spinlocks are defined as mutexes and initialized accordingly:

[35] To inspect the source code see also https://elixir.bootlin.com/linux/latest/source/tools/include/linux/spinlock.h#L9.

```
#define spinlock_t pthread_mutex_t
#define DEFINE_SPINLOCK(x) pthread_mutex_t x = PTHREAD_MUTEX_INITIALIZER
```

The macro `PTHREAD_MUTEX_INITIALIZER` is defined in the `pthread.h` library. In particular, `spin_lock()` and `spin_unlock()` are defined as follows [36]:

[36] These functions are taken from the source code; see also https://elixir.bootlin.com/linux/latest/source/tools/include/linux/spinlock.h#L13.

```
#define spin_lock(x) pthread_mutex_lock(x)
#define spin_unlock(x) pthread_mutex_unlock(x)
```

Also in this case, both `pthread_mutex_lock()` and `pthread_mutex_unlock()` are defined in the `pthread.h` library.

On uni-processor machines, these locks are discarded after the compilation and become simple calls to `preempt_disable()`.

We might have variations of lock acquiring and release operations, which are:

```
spin_lock_irqsave(&mr_lock, flags);
spin_unlock_irqrestore(&mr_lock, flags);
```

These are typically used whenever our kernel code must share data with an interrupt handler, since they disable local interrupts and save the previous state into `flags`.

1. Basic implementation of a spinlock The basic implementation of a spinlock would boil down to a sort of `_atomic_compare_xchg`, as follows:

```
lock:
 ... ; prepare '1' in %edx (new)
 ... ; prepare '0' in %eax (expected)
 ... ; prepare 'ptr' as &lock

; the actual atomic exchange is called `cmpxchg`
; (%eax F= *ptr) ? {*ptr = %edx, %eax = *ptr} : {%eax = *ptr}

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

Readwrite locks

Readwrite locks distinguish between readers and writers [37]. When a writer arrives, there is a check whether there is at least one reader active. These locks come with `irq_save` and `irq_restore` variants, as seen before. In particular, their use could be as follows:

[37] Remember that, if we have multiple readers in the critical section, without anyone writing, there is no problem at all. Somehow, this is what is exploited in by readwrite locks.

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

write_lock(&mr_rwlock);
/* critical section (read and write) ... */
write_unlock(&mr_lock);
```

Seqlocks

We may also have many readers and few writers. For this reason, the kernel has another locking paradigm: *seqlocks*. These locks try to minimize the effort that readers do to access data, but without blocking them and avoiding writers to be put aside (i.e., they may starve). Let us consider an example:

```
write_seqlock(&mr_seq_lock); // increment seq. counter
/* write lock is obtained ... */
write_sequnlock(&mr_seq_lock); // increment seq. counter
do {
  seq = read_seqbegin(&mr_seq_lock);        // ^
  /* read/copy data here                       | check if seq. counter equal. */
} while (read_seqretry(&mr_seq_lock, seq)); // V
```

The `read_seqbegin()` checks whether the counter is odd, then, if so, it spins waiting for this state to change. The `read_seqretry()` checks if some writer appeared in the meanwhile. In this case, writers are not blocked, readers, instead, are forced to redo their work if some new writer arrived in the meantime.

Read-copy update (RCU)

Read-copy update (RCU) is a scalable high-performance synchronization mechanism implemented in the Linux kernel. The final goal RCU tries to achieve is the following: readers shouldn't be blocked by anything and writers shouldn't disrupt the operations of readers. Readers, in particular, should tolerate concurrent writes (as in seqlocks), but in this case there is a whole infrastructure making readers not repeating their operations. As a result, readers avoid locks and should tolerate concurrent writes (might be multiple concurrent versions) [38], and writers create a new version (copy) of the data structure they are working on and they publish the new version with a single atomic instruction [39].

[38] This means also that readers could see an old version of a given datum for a limited time.

[39] This means that writers do their work offline, committing only the final value at the end of their operation.

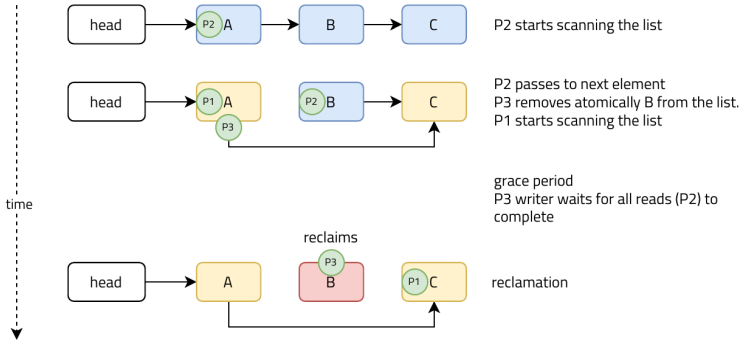


Figure 50: RCU functioning

In particular, task `P3` removes atomically node `B`, thus task `P2` will not be aware of this change, until it re-scans this linked list again. In case of reclamation, `P3` needs to wait until every reads are completed.

For its very nature, RCU can be effective in avoiding deadlocks and making read lock acquisition a breeze. However, for the same reason, it might delay the completion of destructive operations.

More on the underlying idea of RCU

The basic idea behind RCU is to split a *destructive* operation into two parts:

1. Remove references to data items within a data structure, running concurrently with readers (i.e., **removal**) [40];
2. Carry out the actual destruction (i.e., **reclamation**).

Between these two steps, a so-called *grace period* must elapse, and this must be long enough so that any reader accessing the item being deleted has since dropped its references (i.e., dropper `rcu_read_lock()`). Since the read section can't block, one can wait until all CPUs have executed the context switch. This is done by waiting a context switch in `call_rcu` (or `synchronize_rcu`).

[40] This takes advantage of the fact that writes to single aligned pointers are *atomic* and must consider memory ordering.

Examples of RCU usage

Let us inspect this example of a reader (which scans the process list) and a writer (which removes a process from the list):

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

// WRITER
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
}
```

MCS locks

In SMP systems, every attempt to acquire a lock requires moving a cache line containing that lock to the local CPU. For contended locks, this cache line bouncing can hurt performance significantly. In 1990, Mellor-Crummey and Scott proposed a CPU-aware type of lock which tries to avoid the ping pong game between caches.

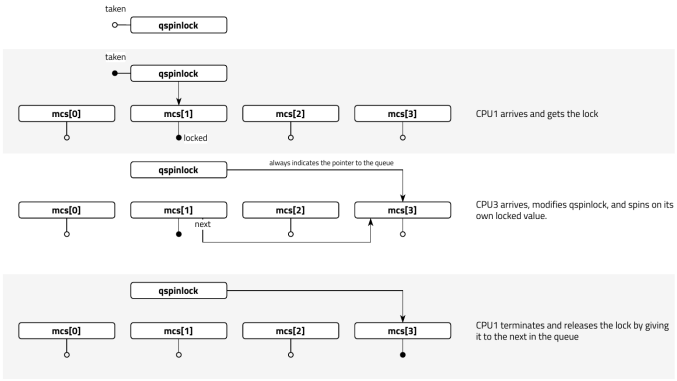


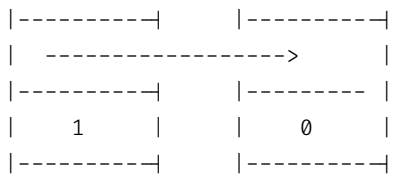
Figure 51: How MCS locks work internally

The idea of MCS locks is the following: all processors will have their particular version of the lock (which it could be either `0` or `1`). This lock is represented as a `mcs_spinlock` data structure in Linux, defined as follows:

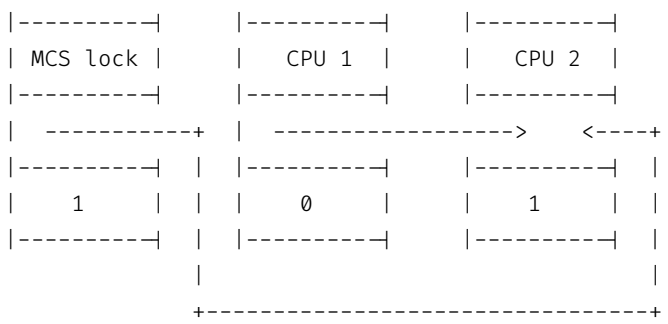
```
struct mcs_spinlock {
  struct mcs_spinlock *next;
  int locked; /* 1 if lock acquired */
}
```

Starting with a not taken spinlock (which means that the lock itself would have its data structure with `locked = 0`), whenever a CPU 1 arrives and takes the lock, it allocates its own *processor-specific* data structure and updates atomically both the pointer and the `locked` field (setting it equals to `1`) of the spinlock data structure. The situation will be like the following:

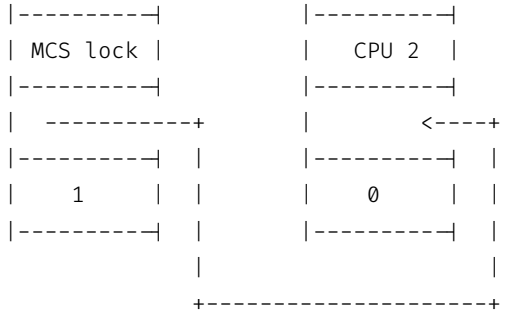
```
|----------|      |----------|
| MCS lock |      |   CPU 1  |
```



Whenever a second processor CPU 2 arrives, the `spinlock` structure (as always) points to the last arrived processor, and the new one does the exact same things the previous one did: it allocates its own version of `mcs_spinlock` and spins over its local `locked` field (which is set equals to `1`). Moreover, it atomically changes the pointers of the `next` field both in the spinlock and CPU 1 data structures. The situation would be as follows:



When CPU 1 finally finishes with the lock, it will do a compare-and-swap operation on the main lock, trying to set the next pointer to `NULL` (which was the original value CPU 1 saw in the beginning) on the assumption that this pointer still points to its own structure. If that operation succeeds, the lock was never contended and the job is done. If some other CPU has changed that pointer as shown above, though, the compare-and-swap will fail. In that case, CPU 1 will not change the main lock at all; instead, it will change the locked value in CPU 2's structure and remove itself from the situation:



In Linux, MCS locks are called *queued spinlocks*, since the pointers between `mcs[1]` and `mcs[3]` in figure, constitute a queue.

10 Lecture 14 - Linux Virtual/Physical Address Space I

The virtual address space

Operating systems introduced a remapping of physical pages into a virtualized version of them, which are called *virtual pages*. The `task_struct`, that we already discussed before, contains a field called `mm`, which is a data structure containing all the information related to the process address space [41].

Virtual memory is used both by kernel code and user code, with the latter being positioned at lower addresses, while the Linux kernel uses higher addresses.

[41] This data structure is also called *memory descriptor* and it is defined under `include/linux/mm_types.h`.

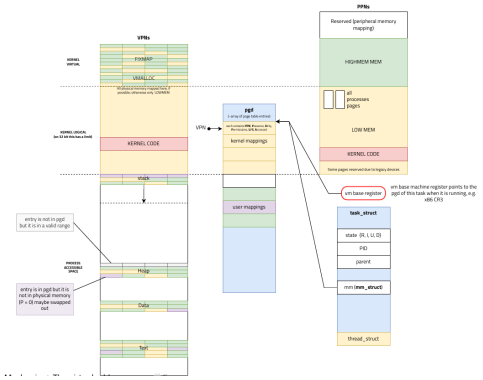


Figure 52: Virtual/physical address space mappings and data structures

Note that, after process switch, the user-space mappings change while kernel ones remain unvaried. Kernel *logical* memory is a remapping of the physical one and contiguous virtual page allocations end up in contiguous physical pages. The kernel *virtual* memory zone corresponds to additional remapping used whenever it should allocate huge memory areas which can't be allocated contiguously in memory [42].

[42] In other words, those pages allocated in the virtual memory zone, will be contiguous from a virtual memory perspective, but not in physical memory.

The `pgd` field (i.e., *page directory*) inside the `mm_struct` is basically a pointer to the (first level) page table. Note that, there is also a *VM base register*, which is a machine register that points to the `pgd` of the current running task (e.g., in x86 `CR3`).

There could be also valid pages in the virtual memory which are not present in the physical one [43]. In other words, their entries are in the `pgd` but not in physical memory (they were maybe swapped out). Furthermore, we could also have pages whose entries are not even inside the `pgd`, but they are in a valid range. This is what is called *demand paging*. To understand its behavior, consider the heap and the `brk()` system call to extend its memory area, Linux acknowledges that the size of the heap has been increased and those new pages (demanded) are treated differently from the others.

[43] They could be present both in the swap space or in the disk itself.

Valid and non-valid ranges are defined using the so-called Linux *VMAs* (i.e., Virtual Memory Areas). VMAs are pointed in the `mm_struct`: they are collected using a list, reporting the valid ranges of the address space. VMAs are represented as `vm_area_struct`, containing `start` and `end` of the corresponding address space region (e.g., `.text`, `.bss`) and those regions might be also related to a file (e.g., where the code has been read). In particular, we say that the structure has a so-called *backing store*. This is important because not all pages occupy corresponding frames in the physical memory (e.g., not all code/data resides always in memory), so they would be directly loadable from the original file. VMAs have flags for `rwx` capabilities, as usual.

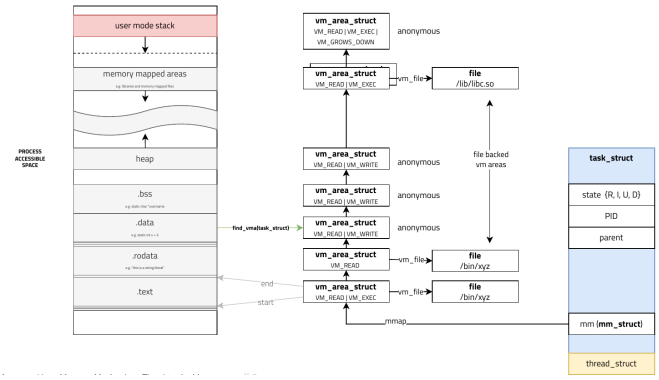


Figure 53: VMAs in the Linux kernel

There are memory areas not having any backing store (e.g., `.data`, heap) and these are called *anonymous* virtual memory areas. These areas will be written in the swap space. Given the address space of a process, there could also be implicitly/explicitly mapped pages which are related to other parts of the system (e.g., libraries such as `libc`). Dynamic libraries may end up in several address spaces (of several different processes), but at the end, those pages will map into the same physical memory region. In other words, some pages may be shared among different processes. To inspect those things on our command line, we can use

```
$ cat /proc/118286/maps
   start         end      prm   offset   dev   inode                      backing store
55ffded63000-55ffded65000 r--p 00000000 fe:00 16074806                   /usr/bin/emacsclient
55ffded65000-55ffded68000 r-xp 00002000 fe:00 16074806                   /usr/bin/emacsclient
55ffded68000-55ffded6a000 r--p 00005000 fe:00 16074806                   /usr/bin/emacsclient
55ffded6a000-55ffded6b000 r--p 00006000 fe:00 16074806                   /usr/bin/emacsclient
55ffded6b000-55ffded6c000 rw-p 00007000 fe:00 16074806                   /usr/bin/emacsclient
55ffded6c000-55ffded6d000 rw-p 00000000 00:00 0
55ffe04a4000-55ffe04c5000 rw-p 00000000 00:00 0                          [heap]
7f2dd88ba000-7f2dd88bd000 rw-p 00000000 00:00 0
7f2dd88bd000-7f2dd88df000 r--p 00000000 fe:00 15994155                   /usr/lib/libc.so.6
7f2dd88df000-7f2dd8a3a000 r-xp 00022000 fe:00 15994155                   /usr/lib/libc.so.6
7f2dd8a3a000-7f2dd8a91000 r--p 0017d000 fe:00 15994155                   /usr/lib/libc.so.6
7f2dd8a91000-7f2dd8a95000 r--p 001d4000 fe:00 15994155                   /usr/lib/libc.so.6
7f2dd8a95000-7f2dd8a97000 rw-p 001d8000 fe:00 15994155                   /usr/lib/libc.so.6
7f2dd8a97000-7f2dd8aa4000 rw-p 00000000 00:00 0
7f2dd8ae0000-7f2dd8ae2000 rw-p 00000000 00:00 0
7f2dd8ae2000-7f2dd8ae3000 r--p 00000000 fe:00 15994143                   /usr/lib/ld-
linux-x86-64.so.2
7f2dd8ae3000-7f2dd8b09000 r-xp 00001000 fe:00 15994143                   /usr/lib/ld-
linux-x86-64.so.2
7f2dd8b09000-7f2dd8b13000 r--p 00027000 fe:00 15994143                   /usr/lib/ld-
linux-x86-64.so.2
7f2dd8b13000-7f2dd8b15000 r--p 00031000 fe:00 15994143                   /usr/lib/ld-
linux-x86-64.so.2
7f2dd8b15000-7f2dd8b17000 rw-p 00033000 fe:00 15994143                   /usr/lib/ld-
linux-x86-64.so.2
7ffdb5b9e000-7ffdb5bc0000 rw-p 00000000 00:00 0                          [stack]
7ffdb5bfa000-7ffdb5bfe000 r--p 00000000 00:00 0                          [vvar]
7ffdb5bfe000-7ffdb5c00000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

Pages with *inode* equals to 0 correspond to sections that do not have a backing store (i.e., anonymous pages). Backing store areas come from the `PT_LOAD` segments in the ELF file and they are, initially, code, read-only data and initialized writable data (i.e., `.data`). Writable initialized data is put read only at first, but copy-on-write is then used if written [44]. Anonymous areas (i.e., heap, stack and `.bss`) are originally mapped into a zero page maintained by the operating system and mapped as read only. A `VMA_GROWSDOWN` area is initialized with a certain number of zero pages and it can grow when the last page is accessed: this area is used for the stack. For other anonymous VMAs, the kernel expects processes to use `brk()` or `sbrk()` explicitly.

To summarize, a VMA can be classified as follows:

- **Mapped** to a file (with *backing store*) or **not mapped** (i.e., anonymous pages - e.g., stack, heap);
- **Shared** or **private**;
- **Readable**, **writable** or **executable** (e.g., `.text` section of a process is typically mapped with `VM_READ` and `VM_EXEC` but not `VM_WRITE`);
- `VM_IO` specifies if that area is a **mapping of a device's I/O space**. This field is typically set by device drivers when `mmap()` is called on their I/O space.

[44] Remember that this is exactly what happens also for a child of a forked process.

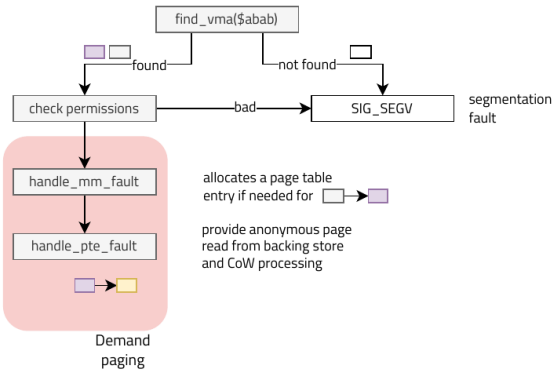
Page fault management

Consider now a process that accesses a page not present in memory. The first thing that Linux does is to invoke a function called `find_vma()` passing an address. The latter scans all the list of VMAs (a list of `vm_area_struct`) and it distinguishes between two cases:

- The page is *not found*, then the address belongs to a page related to a non-mapped region $\Rightarrow$ `SIG_SEGV`;
- The page is *found*, then the page could be in two different states: i) in the page table but not in physical memory; ii) valid but it doesn't have any page table entry. In this case, firstly the permissions are checked (e.g, `rwx` on pages with different capabilities), if something goes wrong, a `SIG_SEGV` is raised. Otherwise, Linux tries to resolve the address by using the `handle_mm_fault()` function [45], checking if some

[45] Linux recognizes that this page is in a "demand paging scenario", so it creates an entry in the page table.

page needs to be allocated in the page table, given that the address is valid. Finally, it will handle the fault by loading, if needed, the data from the disk by using `handle_pte_fault()`.



The activity of loading data only when there is an instruction reading that data (and not before), is called *demand paging*.

Checking the entire VMAs list by the `find_vma()` would have been a costly operation if this list was implemented as a "simple" list. VMAs are represented as a *red-black tree*: a way of arranging elements of a set by considering, in this case, the ending address of that particular `vm_area_struct`. Elements are ordered so to have their value being less than the value of their parent, if they are present in the left-hand subtree, greater if they lie on the right-hand subtree. Linux tries to make this tree balanced to reach the VMA of offending instructions in logarithmic time (i.e., $\mathcal{O}(\log n)$) instead of linear time.

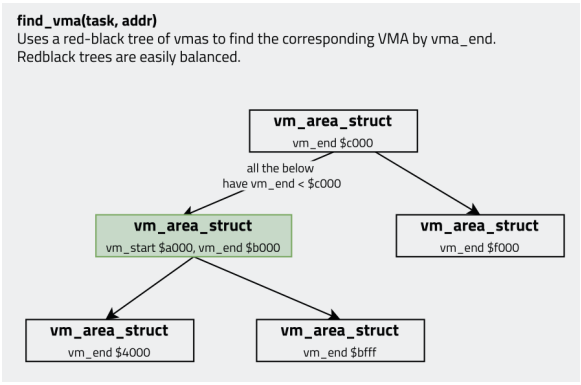


Figure 54: The `find_vma()` primitive

Demand paging

Now, going more into the details of demand paging, consider a program calling `brk()` to enlarge the heap and suppose that this process has a page in the heap which is in the page table but not in the physical memory, and one mapped in physical memory. The `brk()` enlarges the virtual memory area of the process, but new pages will be valid and *non present*. In this way, the physical memory will remain unvaried. Whenever the program has a load/store instruction involving one of the new memory areas, the provoked page fault will be handled and Linux, at the end, will allocate a physical page for the just mapped ones. The example described above is also depicted in figure.

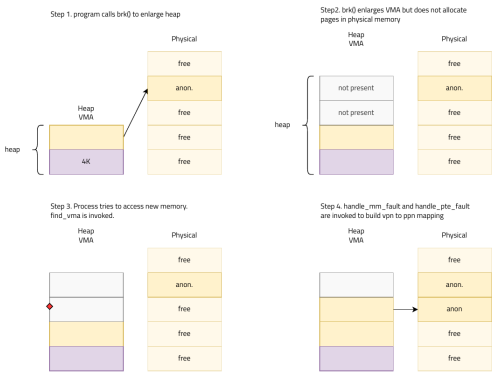


Figure 55: An example of demand paging

User space memory mapping

In user space, virtual memory areas can be created explicitly by a process using the `mmap()`, which goes as follows:

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

In order for programs to read files, these will have some/all of the data stored into physical pages and typically, whenever user-space

processes want to read that data, there will be data-copying (using system calls) and data-returning of that specific data. However, this is not always the best and fastest way to read files. The alternative way to do it is by using `mmap()`, which creates and additional VMA and mappings between the virtual memory space of the process and the physical pages that contain the data in kernel memory. These pages are mapped directly from the *page cache* and can access the file directly.

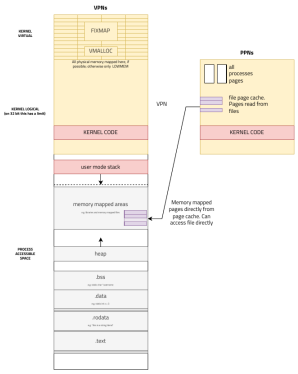


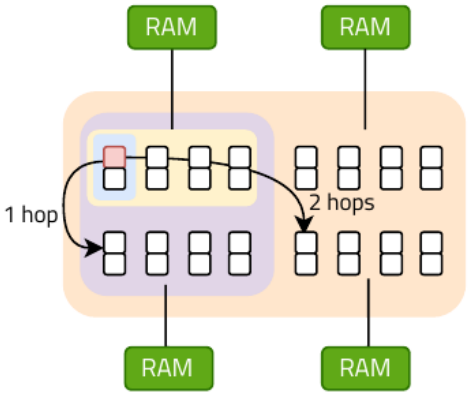
Figure 56: The page cache

At the end of the day, `mmap()` just returns a pointer to the beginning of a region of virtual memory where the contents of the file are located. Actual physical allocation happens only when a process reads or writes that specific page. The operating system tries to find space for the page and make it accessible by updating the page table of the process accordingly (i.e., *demand paging*).

How Linux manages physical pages

Modern systems are structured with non-uniform access memory, such that physical threads are grouped into cores (grouped, in turn, as NUMA nodes) sharing the same RAM. Different NUMA nodes have different RAM banks, so that access latency to data may be different based on which RAM (or NUMA node) it belongs to.

For each NUMA node, Linux keeps a `pg_data_t` structure (all structures are contained in the `pgdat_list`) and maintains three types of information:



Non uniform access memory (NUMA)
A machine with 32 cores, 4 NUMA nodes (eight cores per node sharing a last-level cache),
and pairs of cores sharing a floating point unit.

Figure 57: The NUMA architecture

- `ZONE_DMA`;
- `ZONE_NORMAL`;
- `ZONE_HIGHMEMORY`.

The RAM is divided in zones and each zone is linked to a `free_area` structure, which keeps track of free pages inside the specific area. This data structure is used by Linux to allocate/deallocate pages from the physical memory and it is basically an array of lists, pointing to page descriptors (which can be either free or allocated). In particular, the `free_area` points to set of contiguous pages (e.g., 1 contiguous page, 2 contiguous pages, 4 contiguous pages, etc. up to 2^9) belonging to a list. Furthermore, we have `watermarks` that are values kept by Linux to understand, given the number of free pages, if it should go around and find other pages in case the number of free ones is becoming low. The `pg_data_t` has a pointer to another data structure called `lruvec`, used by the page frame reclaim algorithm to keep track of pages belonging to the array described before (see the picture below for further details).

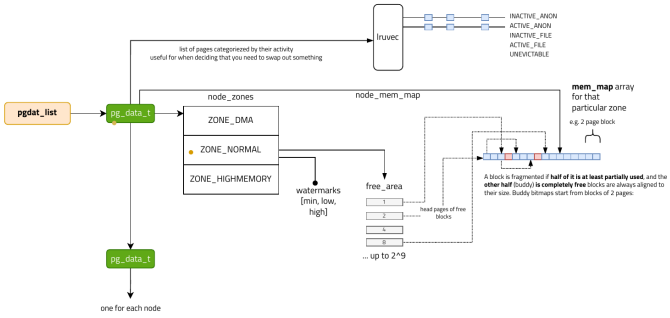


Figure 58: Zones and their data structures in Linux

Page allocation

Consider now a zone, the number of available pages in that zone and the `watermarks` value. Then, we could derive a plot based on Linux allocation policies, as depicted in picture.

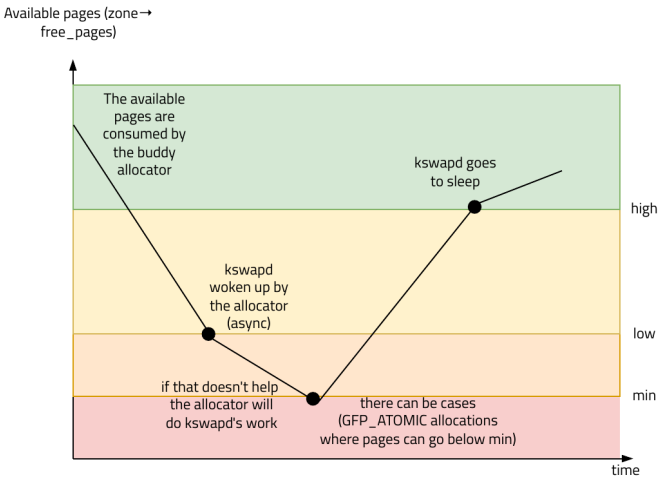


Figure 59: Page allocation

In other words, when the number of available pages goes under a certain threshold, the kernel allocator wakes up a daemon called `kswapd`, which starts to apply the page frame reclaim algorithm, selecting inactive pages and storing them in the backing store or in the swap region. When the number of free pages goes under the `low` value, the allocator (i.e., the page fault handler) will do the `kswapd` thread's job. There can be situations in which the

number of pages might be lower than the `low` value. The hope is that, sooner or later, the number of free pages will go back to its normal state and the daemon will go to sleep.

The Buddy algorithm

There is a well-known memory management problem called *external fragmentation* that the Linux kernel tries to avoid as much as possible. The idea is that frequent releases and requests of groups of contiguous page frames of different sizes may lead to have small blocks of free page frame being scattered inside blocks of allocated page frames. Having contiguous page frames is really important in some circumstances (e.g., buffers for DMA processors), so Linux uses the well-known *buddy algorithm* to avoid external fragmentation.

All page frames are grouped into 11 lists of blocks that contain groups of 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 and 1024 contiguous page frames respectively. The largest request of 1024 page frames corresponds to a chunk of 4MB of contiguous RAM.

Assume that there is a request for a group of 256 contiguous page frames ($256 \cdot 4096 = 1\text{MB}$). The algorithm checks if there is a free block in the 256-page-frame list. If there is no such a block, it looks for the next larger block (i.e., a free block in the 512-list-frame list). If such blocks exists, the kernel allocates 256 of the 512 page frames to satisfy the request and inserts the remaining 256 page frames into the list of free 256-page-frame blocks. Note that if there is no such free block in the 512-page-frame list either, the algorithm checks for it in the 1024-page-frame list, etc. The reverse operation is the one giving the name to algorithm. The kernel attempts to merge pair of free buddy blocks of size b together into a single block of size $2b$ if the following conditions are met:

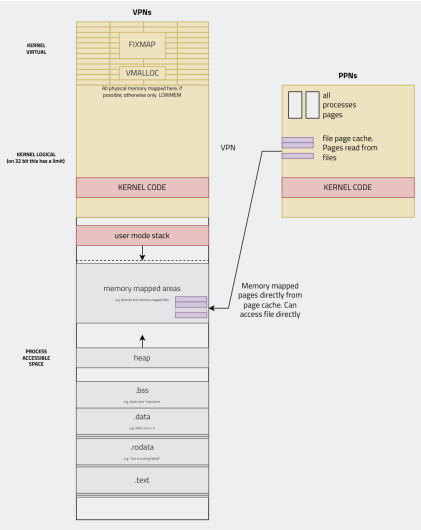
- Both blocks have the same size, say b;
- They are located in contiguous physical addresses;
- The physical address of the first page frame of the first block is a multiple of $2 \times b \times 2^{12}$.

11 Lecture 15 - Linux Virtual/Physical Address Space II & Memory Models I

Page cache

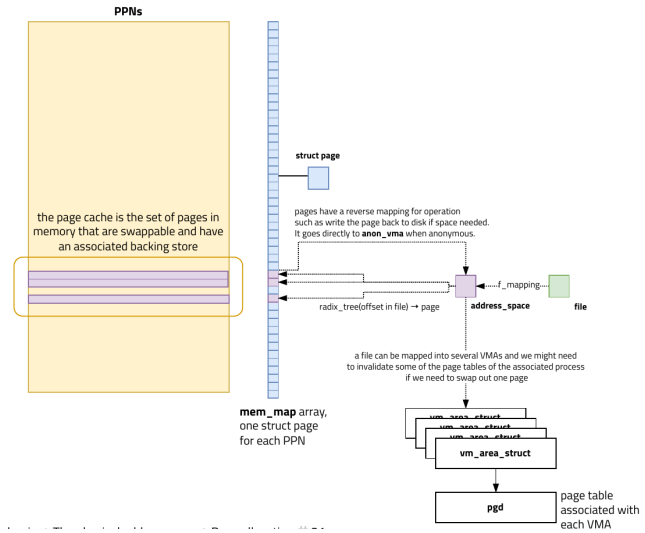
Assume we have multiple processes accessing the same file. To know whether the data requested by some process is already in memory, Linux maintains a *per-file page cache*. The page cache is a set of physical pages that are the result of reads and writes of regular files lying in the file-system. This set of physical pages is stored in a structure called `address_space`. This structure maps `file_descriptor + offset` → `physical_page`, but works also in a reverse way `physical_page` → `[VMA]` [46].

[46] The reverse mapping is useful when we want to invalidate page table entries of shared pages in different processes.



Since the idea of page cache is to implement the `file_descriptor + offset` → `physical_page` mapping, each `address_space` structure has a unique radix tree stored as `page_tree` [47]. To search the cache, the kernel goes through the pages associated with the `address_space` of a file which contains a mapping for `*offset -> physical_page` (through the function `find_get_page(addrsp, offset)`). Also anonymous pages are mapped to the `swapper_space` address space before being swapped out, if necessary.

[47] A radix tree is a type of binary tree and enables quick searching for the desired page, given only the offset.



When a page does not exist in the cache and the memory is full, Linux manages the address space (possibly evicting pages) through its own version of the LRU algorithm.

Page descriptor

Some physical pages can be shared between VMAs (e.g., file-backed page of files shared between processes, copy-on-write page - a child's page which is copied only when written). For that matter, Linux keeps a `struct page` that contains:

- How many such sharings there are (i.e., `_mapcount`);
- How to reach these pages in the case we want to invalidate them (`address_space`, then into each `pdg` pointed by VMAs).

Page Frame Reclaim Algorithm

The PFR algorithm is based on the clock algorithm, which is an approximation of LRU: we flag pages based on the fact that they has been accessed recently or not.

There is a variation for which, if a page is dirty, it is skipped once if `R = 0` since writing things back to disk has a non-minimal cost. Anonymous pages are contained in two lists belonging to the `lruvec` per-CPU data structure, which are `INACTIVE_ANON` and

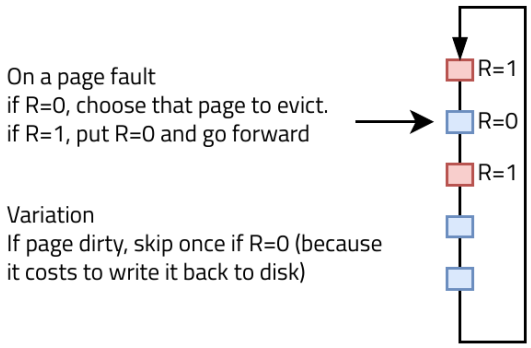


Figure 60: Clock algorithm for evicting pages

`ACTIVE_ANON`, with pages, respectively, considered to eviction and still referenced by something else. However, the clock algorithm has a known problem: we could have pages with `R = 1` but accessed only once and no more since then. So, Linux uses a modified version of the clock algorithm and tries to balance `INACTIVE_ANON` and `ACTIVE_ANON` pages. In this way, page referenced only once will be sooner or later considered for eviction.

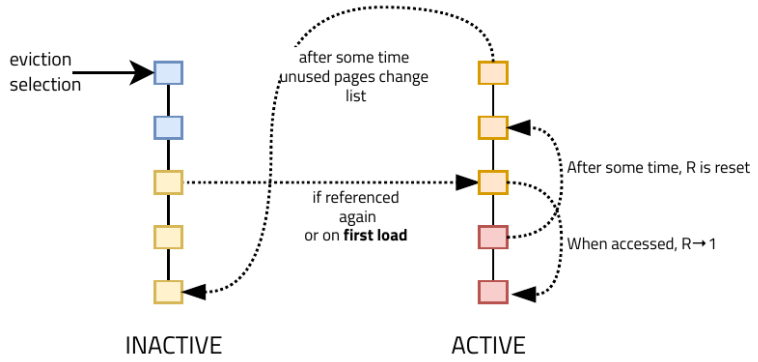


Figure 61: Linux management of the two LRUs

In practice, whenever the page is loaded or referenced, it is put into the `ACTIVE` list with the reference bit set to `0`. When this page is accessed again, it will be moved as a tail and the reference will

be set to 1. In the ACTIVE list, the more pages will be near the tail, the more they are far from being evicted. After some time, active pages with the reference bit set to 1 are reset to a state such that R = 0. After some time, pages in such a state are moved into the INACTIVE list. If they are not accessed anymore, they are put at the top of the INACTIVE list, where they are more likely to be chosen for eviction.

In other words, Linux manages the space (possibly evicting pages) through its own version of the LRU algorithm, which works as a zone basis:

- Keep two lists of clean pages for each zone, active and inactive (kept in the page file);
- Victim are taken from the inactive list and must be not dirty;
- Pages go into the ACTIVE list only after two accesses;
- Linux periodically moves the pages from ACTIVE to INACTIVE, trying to keep lists balanced;
- The heuristic refill_inactive when the INACTIVE list is small.

Object allocation

In the kernel, there is also a need to allocate memory regions which are smaller than a page. In general, within the kernel, fixed size data structures are very often allocated and released.

However, the Buddy System we presented before clearly does not scale: internal fragmentation can rise too much [48] and the Buddy System on each NUMA node is protected by a spinlock. Even if the object we want to allocate has the size of a page, it is not always efficient to use the Buddy System. For instance, allocation and release of page tables requires a frequent allocation and de-allocation of the same fixed size structures through the following functions:

- pgd_alloc(), pmd_alloc() and pte_alloc();
- pgd_free(), pmd_free and pte_free().

[48] One page for a single object is an enormous waste of resources.

Fast allocators

There are two fast allocators in the kernel:

- *Quicklists*: used only for paging;
- *Slab allocator*: used for other buffers.

Quicklists are used to implement the page table cache (i.e., pages that we need for the data of the page table itself). For the functions `pgd~/~pmd~/~pte_alloc()` we have three quicklists `pgd~/~pmd~/~pte_quicklist` per-CPU. Each architecture implements its own version of quicklists, with the same underlying principles. Quicklists use a LIFO approach: during the allocation, one page is popped off the list and, during the 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.

The Slab allocator has as its underlying idea to have caches of commonly used objects kept in an initialized state, available for use by the kernel. This allocator consists of a variable number of caches, linked together by a double linked list called `cache_chain`. Every cache manages objects of a particular kind (e.g., `mm_struct`) and maintains a block of contiguous pages in memory called *Slab*.

1. More on the Slab allocator There have been different implementations of the Slab allocator to this day. Nowadays, on most distributions, the default Slab allocator is the SLUB allocator.

   Let's recap what we need to have for such an allocator: we need object caches. We can't use simple page allocation for each e.g., `struct` since there will be a considerable internal fragmentation [49].

   A *slab* is an intermediary structure which consists of one page frame containing both allocate and initialized (but free) objects. We can call the slab to obtain new objects:

   ```
   void *kmem_cache_alloc(kmem_cache_t *cachep, int flags);
   ```

   For instance, when a file is opened, the `cachep` used is the `filp`. The Slab tries to find a slot in the active `cpuslab`. However, if this is full, a partial one is taken and switched into the active one. When later the object is freed, it is left in an initialized state so that the next allocation will be faster.

   Note that even `kmalloc()` is based on the slab: it allocates generic size from a set of caches (`malloc_sizes`) that are searched dynamically.

[49] If we want a 65-page chunk, we have to ask for and get a 128-page chunk. Or, if we want to allocate a `task_struct`, which is much less than 4K, we need to ask for 4K.

The Slab allocator provides two main classes of caches:

- *Dedicated*: these are caches created in the kernel for commonly used objects (e.g., `mm_struct`, `vm_area_struct`, etc.). Structures allocated in this cache are initialized, and, when they are freed, they remain initialized for the next allocation to be faster;
- *Generic* (*size-N* and *size-N(DMA)*): these are general purpose caches, which, in most of the cases, are of sizes corresponding to powers of 2. This separation can be seen in the dedicated file for slab in the `proc` file-system.

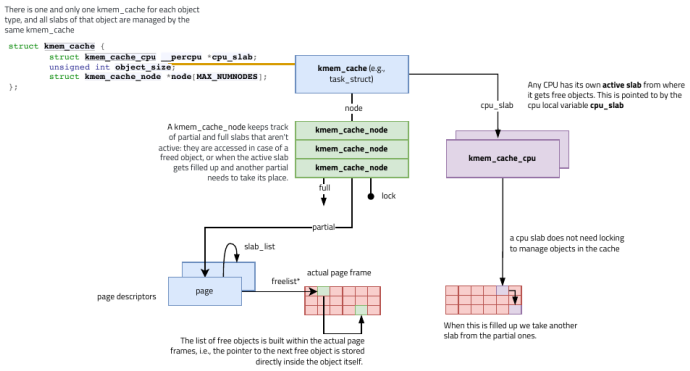


Figure 62: The SLUB allocator

Linux memory security

Buffer overflows and ASLR

Buffer overflows are not so rare when programming in a low-level programming language as C. To overcome to this fact, the Linux kernel came with address space layout randomization, which randomizes the stack at every execution of a binary. However, buffer overflows may occur also in kernel space, so Linux introduced kernel ASLR (KASLR), which randomizes at each startup the kernel's `.text` section. This is done mainly because one of the easiest way to get root privileges is to execute two simple functions:

```
commit_creds(prepare_creds(NULL));
```

Knowing where symbols of interest live in the kernel's address space might give the attacker a chance to change *credentials* [50].

However, KASLR currently randomizes only where the kernel code (i.e., `.text`) is placed at boot time. Moreover, the amount of randomization that can be applied to the base address of the kernel image is rather small, due to the address space size and hardware constraints related to memory management.

[50] Provided that this attacker knows how to force their execution when getting into kernel space. To know more on credentials see also https://www.kernel.org/doc/Documentation/security/credentials.txt.

Meltdown

Kernel memory can contain sensitive information from all sorts of applications running in the system (e.g., passwords). In principle, kernel memory is *all* the memory. The Meltdown attack demonstrated that processor speculation can leak kernel memory into user mode long enough for it to be captured by a side-channel cache attack. For instance, consider the following piece of code:

```
char my_userspace_buffer[BUFSIZE]; // attacker created
// ...
clear_up_cache();
long_instruction();
char t = my_userspace_buffer[ ((*kernel_address_to_be_analyzed) & 1) << BUFSIZE];
// measure time to load my_userspace_buffer[0] and my_userspace_buffer[BUFSIZE]
```

In the example above:

- `clear_up_cache()` ensures no entry in `my_userspace_buffer` is cached;
- `long_instruction()` forces the CPU to begin out of order issue and execution of subsequent instructions (i.e., it starts loading `*kernel_address_to_be_analyzed`);
- Considering the x86 architecture, the fault associated with this load is triggered only when retiring from the ROB. In the meantime, `*kernel_address_to_be_analyzed` is used for successive instructions to load speculatively from `my_userspace_buffer`;
- To perform the attack, the cache state is examined (i.e., it measures how much time it gets to load `my_userspace_buffer[0]` and `my_userspace_buffer[BUFSIZE]`). If `my_userspace_buffer[0]` is faster, then the low bit of `kernel_address_to_be_analyzed` was 0.

Meltdown is known to have been used in any actual attack, even if is potentially disruptive. For older Intel CPUs, KPTI is one possible solution.

Kernel Page Table Isolation (KPTI)

On Intel machines, the `CR3` processor points to a first-level page table. The latter indeed is a hierarchical structure working as shown in picture.

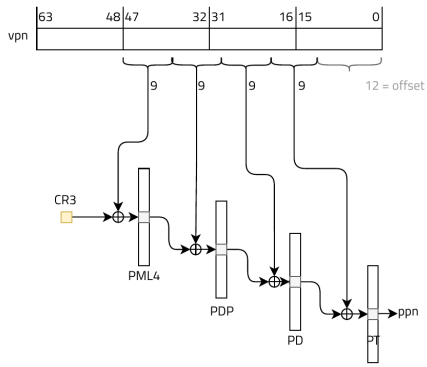


Figure 63: 4-level Intel page table

In Linux, when the user code is running, its page table does not contain all the mappings for the kernel, but only a fraction of them [51]. When a context switch happens, Linux switches into a full blown kernel memory address space and implements specific actions around user-space code (i.e., *protection against execution* and *protection against invalid references*). In practice, we would not be able to run user-space code when in kernel context.

[51] This minimum number of pages is the one used to manage a context switch (to invoke system calls essentially).

Before KPTI, Linux kept a page table which had both user and kernel mappings, so that most context switches-related overheads (e.g., TLB flush, page-table swapping) could be avoided. KPTI, instead, uses two separate user-space and kernel-space PGD:

- One PGD includes both kernel-space and user-space addresses (as it was originally), but it is only used when the

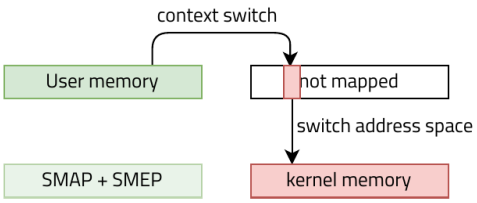


Figure 64: KPTI functioning

system is running in kernel mode. Protection against execution (SMEP) and access invalid references (SMAP) are both in place;

- The second PGD, for use in user mode, contains a copy of user-space and a minimal set of kernel-space mappings providing the information needed to enter or exit from system calls, interrupts and exceptions.

KPTI comes with a *measurable run-time cost*, estimated at about 5%. The kernel code needed to deal with interrupts will no longer exist in the address space. So there must be enough kernel code mapped in user mode to switch back to the kernel PGD and make the rest available. All entries to the kernel from ring 3 must have a set of trampolines to switch page tables and jump into the kernel (i.e., `cpu_entry_area`). None of this is accessible directly from ring 3, but ring 3 code can jump into it by using CPU-mediated mechanisms allowing to change privilege levels (e.g., the CPU's `syscall` instruction) [52].

[52] For more details on KPTI see also https://lwn.net/Articles/741878/.

Memory models on SMP machines

On SMPs, different threads could have a non-aligned and different view of the activity of other threads over the memory. Threads indeed could also have a non coherent view wrt. operations done in memory, compared to the original program flow.

The *memory model* defines the visibility's behavior (and consistency) of operations done by one thread, from the point of view of the other threads executing on the SMP machine. Due to write buffering, speculation and cache coherency protocols of modern

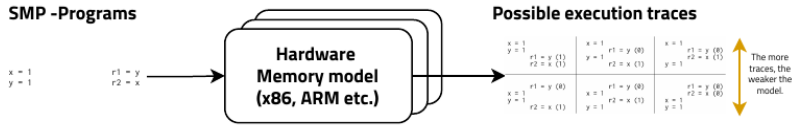


Figure 65: SMP programs and execution traces

processors, the order of which memory accesses are seen by other threads might be different from the order in the issuing thread.

For example, consider the program:

```
// Thread 1
x = 1;
done = 1;

// Thread 2
while (done == 0) { /* loop */ }
print(x);
```

On certain processors, this works intuitively. For instance, on an Intel x86 multi-processor, this will always print 1. While, on an ARM processor it can print 0, if no mitigation is adopted.

Sequential consistency

There are two orders of operations: the one dictated by the program and another one for which operations are visible to other threads. The order in which operations are visible is called *memory order*, while the other one is called *program order*.

To formalize, we define $<_p$ as the *program order* of the instructions in a single thread and we define $<_m$ as the order in which these instructions are visible in the shared memory (either order is also referred to as the "happens-before" relation), or *memory order*.

A multi-processor is called *sequentially consistent* if and only if, for all pairs of instructions $(I_{p,i}, I_{p,j})$, we have that

$$I_{p,i} <_p I_{p,j} \Rightarrow I_{p,i} <_m I_{p,j}. \tag{6}$$

In practice, operations of each individual processor appear in this sequence with an order specified by its program. This is clearly the strictest memory model.

Consider the following example:

```
// Thread 1
x = 1
y = 1

// Thread 2
r1 = y
r2 = x
```

On a sequentially consistent processor, this program can't end with `r1 = 1, r2 = 0` since only the six interleavings in figure are valid.

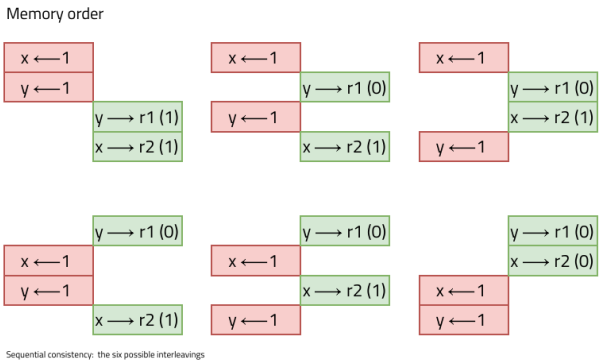


Figure 66: The six possible interleavings

However, the are architectures for which this memory model doesn't hold anymore. In particular, sequentially consistent processors are too expensive to be built and they require an intrinsic complexity which is not affordable at all.

Total Store Order

Intel's x86 processors are Total Store Order ones. Consider a shared memory scenario, so these processors are not directly connected to this shared memory, but they interact with a component in between called *Store Buffer*, as shown in picture. This component is here to hide memory latency and commits values to the shared memory in an asynchronous way.

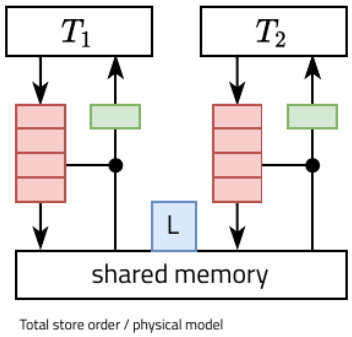


Figure 67: Total Store Order physical model

Since data is not immediately written into the shared memory, values take some time to be seen by other threads. When processor T_1 in the example wants to read something which is in the Store Buffer, this data is taken directly from the buffer and read (and will not be taken from the shared memory). A single processor can see effects of its own writes before others. In general, this is a problem.

There are some programs (not the one shown previously) that can have different effects based on whether they run on sequentially consistent or TSO processors.

Consider now the program already seen before:

```
// Thread 1
x = 1
y = 1

// Thread 2
r1 = y
r2 = x
```

On sequentially consistent processors, this program can't end with `r1 = 0, r2 = 0`, but on TSO processor this may happen.

This happens before the two writes are executed but their effects will be seen later on, since data needs to flow inside the write buffer.

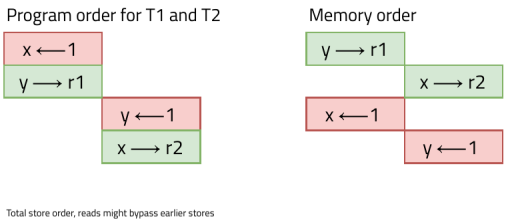


Figure 68: Read might bypass earlier stores

12 Lecture 16 - Memory Models II & Virtualization I

More on memory models

In the last lecture we saw two memory models: sequentially consistent machines and Total Store Order processors (e.g., Intel's x86). Now, there is one last memory model to be analyzed, which is typical of ARM processors.

Partial Store Order

Partial Store Order is the ARM processor's memory model and it is even weaker than TSO. Each processor reads from and writes to its own complete copy of the shared memory and each write propagates to other processors independently, with reordering allowed as the writes propagate.

Hardware threads can perform out-of-order writes, or even speculatively (i.e., before preceding conditional branches are resolved). Any local reordering is allowed, unless otherwise specified. Note that the reordering of writes by a single processor means that T_1's writes may not be observed by other threads in the same order. Moreover, reads might be delayed after writes.

On ARM architectures, within a single hardware thread, there are (implicit) dependencies to enforce orderings (i.e., address, control and data dependencies) that make programming more intuitive. However, the memory system does not guarantee that a write becomes visible to all other hardware threads simultaneously (this behavior is called *non-multicopy-write-atomicity*).

Consider now the example we already saw last time:

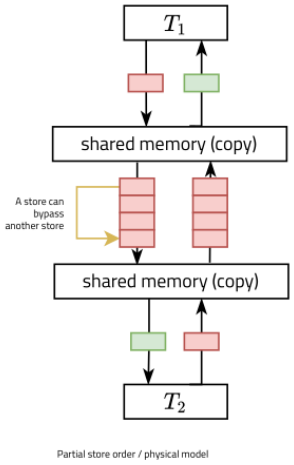


Figure 69: Partial Store Order physical model

```
// Thread 1
x = 1
y = 1

// Thread 2
r1 = y
r2 = x
```

On a PSO machine, this program can end with `r1 = 1` and `r2 = 0`. There are little programs that can tell us whether one system is either complying with one memory model or not, which are called *litmus tests*.

Data races

Data races may occur, as we already saw before. To reason more intuitively about the memory behavior of TSO and PSO one can use synchronization instructions. In computer science, the act of synchronize means to cause something to happen in a planned way (i.e., by enforcing some order).

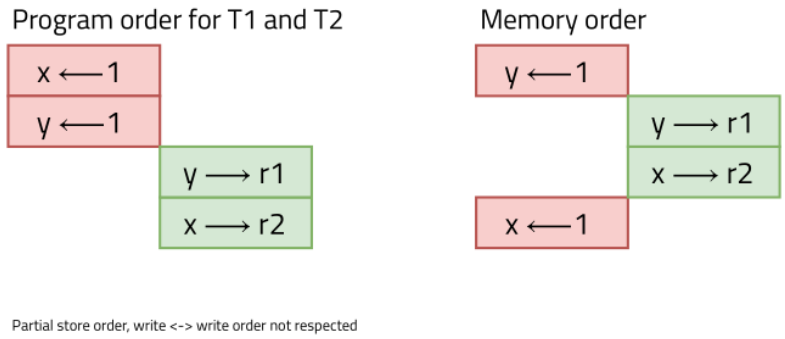


Figure 70: On a PSO machine, writes order could be not respected

Memory barriers

Fences (or *barriers*) are synchronization instructions that enforce the *program order* $<_p$ into the *memory order* $<_m$. These instructions might flush write buffers to make the memory up to date.

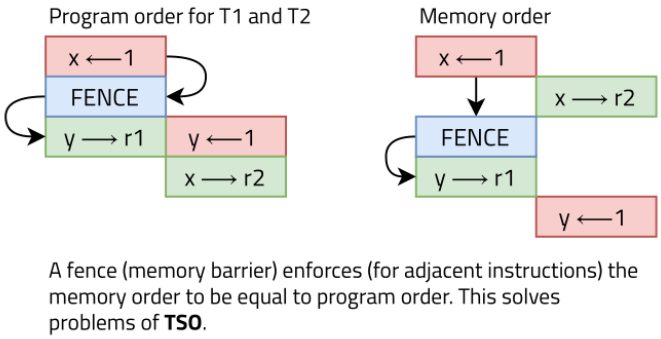


Figure 71: Memory barriers

There are several ways in which we can use fences, for instance:

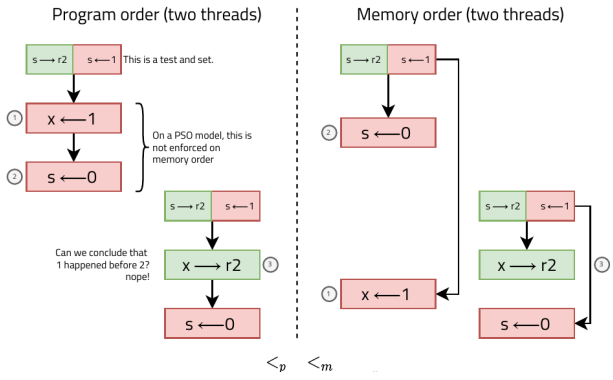
- Enforce sequential consistency by putting fences everywhere, which is the overly conservative thing we can do. In this way, it is easier to reason about the effect of fences, but we'll

force processors (and compilers) to avoid optimizations, lowering the overall performance;

- Identify when this order is actually needed. Programmers use structured programming where one thread must acquire the ownership of a resource before working on it. This line of reasoning can be exploited to enforce the order only when it is needed.

Hardware memory models

A *hardware memory model* defines the behavior and the visibility (along with the consistency) of operations of a machine language program done by one thread from another thread. In this process, no compiler is involved. We already defined data races in some sense, but they are situations such that our program substantive behavior depends (in an uncontrolled way) from the memory model and timing of the involved threads. When this thing is undesirable, it is considered to be a bug. In other words, there is a data race whenever we have two instructions `a` and `b` (in different threads) where at least one of them is a write and when there is nothing to enforce either `a` $<_m$ `b` or `b` $<_m$ `a`.



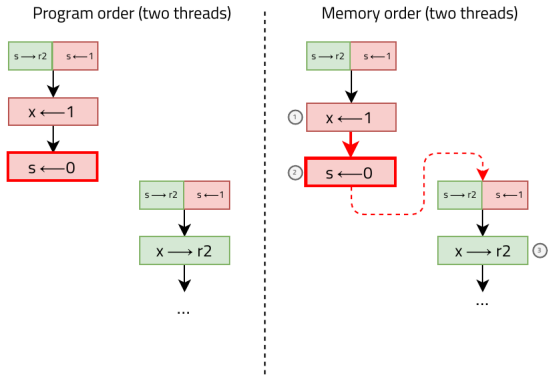
In this example, we have a read (3) and a write (1) for which we can't say that (1) happened before (3). These are concurrent. Since at least one of them is a write, this is called *data race*.

There are two operation's categories we must deal with when concurrency is involved:

- *Synchronization operations*: operations that can be used to enforce the memory order of other operations. They can be of two types:

1. *Ownership release operations*: synchronization operations that, when observed, make the observer conclude that all previous writes have been completed. This is generally a store that ensures that the next write happens after any previously-executed reads or writes;
2. *Ownership acquire operation*: synchronization operations that must be used by the observer to recognize the release operation. This is a load that ensures that the next read happens before any subsequent reads or writes [53];

- *Data operations*: normal reads and writes.



In the above example, we have a synchronization release (2) that ensures all previous writes have been completed. It creates a new, cross-thread arc (i.e., "synchronizes-with"), which allows us to say that (1) happens before (3). In particular, to enforce the reordering we should have a release operation (the one performed by T_1 in the example), followed by an acquire operation (the one performed by T_2 in the above example): this series of instructions is key to enforce the "happens-before" relationship.

In case we don't have data races, hardware thread T_2 will never see the reordering of (1) and (2). The observability of T_1's operations is thus limited. The basic tenet of memory consistency theory is that, if a program is data-race-free, then it will appear as if it was sequentially consistent all along.

[53] Most importantly, if an operation P with release semantics synchronizes with an operation Q with acquire semantics, then operation P happens before operation Q, even if they happen in different threads.

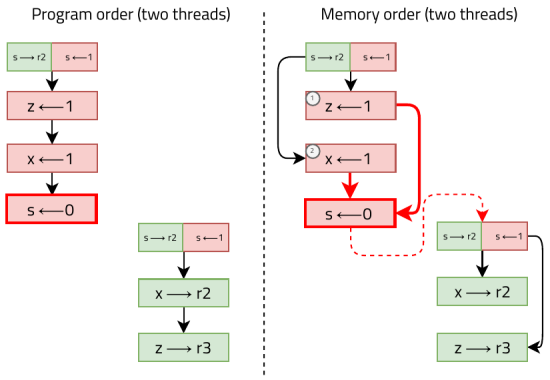


Figure 72: The happens before relationship has been enforced. Even having reordered instructions in each thread, the program will behave as a sequentially consistent one.

Software memory models

Compilers can introduce additional reordering of instructions that might appear as if the machine had a weaker memory model. This adds another layer of complexity to hardware memory models. Higher level languages must give to the programmer a way to enforce the ordering of happens-before relations just as it is done at the ISA level. This is called *language memory model*. These must work portably across SC/TSO/PSO hardware platforms and account for low level differences. For example, we could write a TSO program that might run on a PSO machine. When writing data-race-free programs, we will get that they will behave like sequentially consistent ones, even exploiting hardware and software optimization.

In particular, a *software memory model* defines the behavior and the visibility (along with the consistency) of operations of high-level language programs done by one thread from another thread. For example, a compiler might reorder writes to unrelated variables. This is perfectly fine for single-threaded applications but could make a multi-threaded program to behave like it was running on a PSO machine. Compilers may also introduce even weaker memory models (i.e., they can even break the coherence, making a thread to read the same variable twice with a different history [54]).

[54] For more details, see https://research.swtch.com/plmm.

The Linux Kernel Memory Model (LKMM)

As we saw before, a memory model is a set of instruments with which a programmer can enforce happen-before relationships. Linux has its own memory model, which is basically the least common denominator of the guarantees of all memory models related to all CPU families running in the Linux kernel. The LKMM provides non synchronized (relaxed) access to data, with `READ_ONCE` and `WRITE_ONCE` forcing the compiler to:

- Prevent reordering of reads (writes);
- Omitting reads (for known values) or doing too many (when register spilling is needed).

This memory model naturally provides access to synchronized data: `smp_store_release` and `smp_store_acquire` are the APIs that could be used [55]. Moreover, Linux has the `atomic_t` (32 bit) and `atomic64_t` types. Operations on these types are always guaranteed to be not interruptible. For instance:

[55] Both of them could be implemented with `READ_ONCE` and `WRITE_ONCE` plus memory barriers.

```
atomic_t v = ATOMIC_INIT(0);

atomic_set(&v, 4);
atomic_add(2, &v);
atomic_inc(&v);
```

RMW instructions on `atomic_t` are implemented through lockless techniques (which might involve loops), and/or instructions with the `LOCK` prefix [56]. Some of them have acquire-release semantics.

[56] For more details and to inspect the actual source code, see https://elixir.bootlin.com/linux/v5.15-rc1/source/arch/x86/include/asm/atomic64_64.h#L184.

Introduction to Virtualization

To introduce the concept of *Virtualization*, we should start with the definition given in 1974 by Popek and Goldberg: a virtual machine is

> an *efficient*, *isolate* duplicate of the real machine,

dedicated to an operating system. This VM is based on a *virtual machine monitor* (or *hypervisor*) that:

- Creates an environment for an operating system which is essentially identical with the original machine (having only minor speed decrease);

- It is in complete control of system's resources.

There are several reasons why to use a virtual machine. The first reason is that it allows to consolidate and partition the hardware [57] and to react to variable workloads (with a reduced hardware and administration costs for data centers, providing horizontal scalability). Having virtual machines also allows to have a standardized infrastructure (e.g., have a firewall configured in the same way the virtual machine is configured), simplifying software distribution of complex environments. This standardized infrastructure also allows to have isolated network and storage. Last but not least, VMs allow security sandboxing and fault tolerance (e.g., *checkpointing* and *rollbacks*).

[57] The idea here is to use one physical machine at 100% instead of two at 50% each. Moreover, in some cases an entire machine is exaggerated for the specific use case.

Some terminology and theorems

The following is a list of common terminology which is crucial to have in mind when discussing virtualization:

- *Host system*: the operating system where the virtual machines run;
- *Guest system*: the operating system that runs on top of the virtual machine;
- *Virtual machine monitor* (VMM) or *hypervisor*: software program that translates/mediates accesses to physical resources such as interrupts or sensitive processor state and ensures isolation;
- *Instruction types*: unprivileged and privileged. The latter are those that trap in kernel mode;
- *Virtualization idea*: run privileged instructions in a de-privileged mode;
- *Instruction sensitivity*: an instruction is virtualization-sensitive if it is both control sensitive and behavior sensitive [58].

[58] Being *control sensitive* means that the instruction modified directly the machine status (e.g., enabling or disabling interrupts, modifying the interrupt vector table) and being *behavior sensitive* means that these instructions behave differently when used in either user or supervisor mode. This might affect fidelity.

There are two types of hypervisors:

- *Type 1 hypervisors* (also called *native* hypervisors): they run on bare metal without any operating system abstraction;
- *Type 2 hypervisors*: they run in the context of another operating system (e.g., KVM and VirtualBox).

According to a theorem by Popek and Goldberg

> For any conventional computer, a virtual machine monitor may be built if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.

This doesn't mean that the virtualization can't be implemented otherwise. As a matter of fact, old x86 processors (non VT-x) did not respect the Popek and Goldberg's theorem originally (for this reason VMWare's engineers needed to get around this limitation).

Hardware vs. software virtualization

There are two different types of virtualization: software and hardware based.

The software based one is generally done through *deprivileging*:

- All guest software must be run at a privilege level less than the supervisor;
- VMM runs in supervisor mode as a collection of fault handlers;
- On x86 hosts, the user can be in two different privilege modes (i.e., ring 1 and ring 3).

There are two kinds of software-based virtualization:

- *Full system virtualization*: we have a real abstraction of the underlying hardware and VMs run unmodified operating systems (such as the ones running on real machines);
- *Operating-system virtualization* ("containers"): we have an abstraction of the underlying operating system (and we do not abstract the hardware) but only in its user-space part [59].

[59] The unmodified user-space runs in the container.

In other words, software based virtualization ensures that the guest user code runs on the processor, while making guest privileged instructions being intercepted and emulated (i.e., *trap-and-emulate*) by the hypervisor. This is the classical definition of *full-system virtualization* (1974). A classical VMM executes guest operating systems directly, but at a reduced privilege level [60].

[60] For more details, see also https://www.vmware.com/pdf/asplos235_adams.pdf.

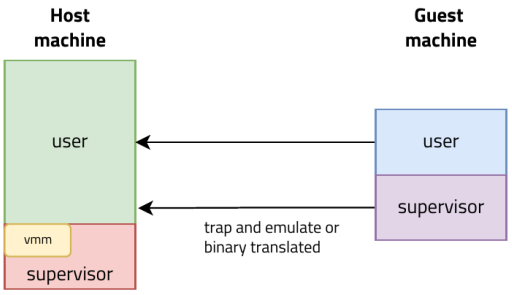


Figure 73: Software based virtualization

Hardware based virtualization, instead, introduces new modes to avoid the problems of *deprivileging*:

- Introduces an orthogonal guest-mode where the guest's supervisor works as in a traditional supervisor (with no ring aliasing);
- Maintains a control block where the state of the guest can be explicitly and comprehensively saved and resumed (used for some shadow structures);
- VMM still is invoked on traps (called *exits*).

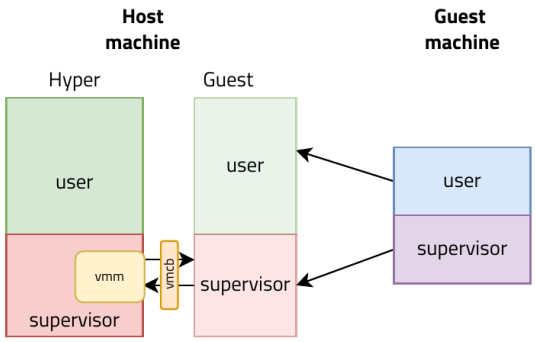


Figure 74: Hardware based virtualization

In other words, there are some processors that provide hardware assistance for CPU virtualization. Hardware assisted virtualization reduces hypervisor intervention to the minimum possible. Processors provide two additional guest/hypervisor modes which are orthogonal to user/supervisor modes, as shown in picture. Guest/supervisor mode runs almost at native speeds and only on certain events it enters hypervisor/supervisor mode (typically on page table manipulation).

Hypervisor architecture

Always according to Popek and Goldberg, there are three requirements for virtualization:

- *Fidelity*: the behavior of the VM must be reasonably equivalent to the one of the real machine (the VM could be slower);
- *Safety*: the VM doesn't have control of the physical resources, but only of the virtualized ones. Virtual resources are assigned to the VM by the VMM and this control can't be overridden;
- *Efficiency*: programs should "show at worst only minor decreases in speed".

1. Arriving to the hypervisor concept Consider now a simple memory model implementing segmentation as "relocate and bound":

   ```
   L = lowest accessible address
   B = virtual memory size
   S = physical memory size
   ```

   Whenever there is a memory access, the operating system does the following:

   ```
   read(A):
       if A + L > B then ERROR
       if A > S then ERROR
       read from memory address A + L
   ```

   Now, let ask ourselves whether this machine is virtualizable, by seeing L and B registers and splitting the memory in different areas (and splitting them further). Consider

```
(L0, B0) = memory area assigned to the VM
(L1, B1) = L and B registers of the VM
```

so using this model, we should also have

```
(L01, B01) = L and B register while the VM runs: (L0+L1, min(B1, B0-
L1))
```

Figure 75: The example of a simple memory model for VMs

In such a setting, if the VMM can guarantee that the virtual machine can't modify the L and B registers pointing to an out-of-bound memory area, then it should be enough to have all the requirements. Right now, since the virtual machine can write L and B arbitrarily, we can't virtualize this machine.

We could consider, instead, two modes of execution: user mode and supervisor mode. In such a way, instructions can "trap" to supervisor mode:

```
trap:
    MODE = sup
    L = 0
    B = S - 1
    M[0] = PC
    PC = 0
```

However, this is not enough to make the machine being virtualizable. The reason why we need user and supervisor mode is that are some instructions called *sensitive*:

- *Control-sensitive* instructions affect the availability of resource (overwriting the register L we could make memory not accessible before as accessible);

- *Behavior-sensitive* instructions behave differently depending on the configuration of the machine (reading the L register will lead the VM to have something it doesn't expect).

An efficient VMM is possible if all sensitive instructions are privileged, since control-sensitive instructions affect safety and behavior-sensitive instructions affect fidelity [61]. The basic mechanism for virtualization is called *trap-and-emulate*.

2. Trap-and-emulate mechanism The idea of the *trap-and-emulate* mechanism is to make the VM run **always** in user mode and to maintain a copy of privileged CPU registers (i.e., processor flags and segmentation/paging registers). The VMM traps the sensitive instructions, emulates them and goes back to the virtual machine [62]. In particular, the execution of the VM by the hypervisor is the following:

```
for (;;) {
    if (VM has interrupt) {
        setup VM registers for interrupt delivery
    }
    store hypervisor registers
    load VM registers
    return to user mode
  trap_vector:
    store VM registers
    load VM registers
    act on processor trap
}
```

This is a very old technique (1965, IBM CP-40) and

> "each virtual machine program is actually executed in problem state (i.e., user mode). The effects of privileged instructions are reproduced by CP within the virtual machines".

This technique works but it implies huge costs and it is rather slow, not suitable for today's standards. It suffers to trap amplification (i.e., each trap becomes 6/7 traps). Moreover, virtualization of paging structures is quite difficult.

[61] There are examples of sensitive instructions such as read/write processor flags (e.g., supervisor bit, disable/enable interrupts), read/write segmentation registers, MMU root (page tables), read/write interrupt vectors, invalidate caches, etc.

[62] In particular, the VMM executes privileged instructions in hypervisor context and go back to the virtual machine. There is also the possibility to inject a fake trap into the virtual machine.

3. Moving to modern approaches: shadow IDTs Software-based virtualization nowadays implies to use two different techniques:

   - *De-privileging*;
   - *Shadowing*.

   We are going to describe both of them. Let's firstly inspect the architecture shown in figure below.

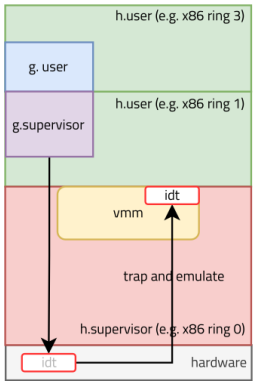


Figure 76: The shadow IDT

In a virtualized setting, `g.supervisor` is translated into `h.user`, which is reasonable: the virtual machine (even on its supervisor counterpart) is executed in user mode. Privileged instructions or memory accesses produces an interceptable trap. Moreover, `h.supervisor` installs its own (shadow) structures instead of those dictated by `g.supervisor`. For example, x86 uses ring 1 instead of ring 0 for guest's ring 0 [63]. Applications (running at ring 3) can't alter the state of the guest operating system (running at ring 1) and the latter can't access data structures of the host operating system. Finally, exceptions or interrupts are captured by the VMM (at ring 0) and must be properly handled (e.g., by reflecting them into ring 1 tasks).

[63] This technique is called *0/1/3 virtualization*.

For instance, on x86 architectures, the VMM can trap the following instructions:

- `hlt`, `lidt`, `lgdt`, `invd`, `mov %crx`, since they modify the machine status;
- `cli`, `sti`, since they enable or disable interrupts.

The shadow IDT shown in picture contains wrappers invoking the VMM, which in turn decides if `g.supervisor` must be called. System calls, for instance, are managed simply with a *trampoline setup* between the two IDTs.

4. Shadow page tables A number of key data structures used by processors need to be *shadowed*, which means that there is an actual physical data structure (derived from the one seen by the guest) that is actually used by the host. There are several shadow data structures used mainly for virtualization. For instance, when dealing with the page table:

   a) The VMM intercepts the guest operating system and sets the virtual CR3;

   b) The VMM iterates over the guest page table, constructs a corresponding shadow page table;

   c) In the shadow page table, every guest physical address is translated into the host physical address (i.e., machine address);

   d) Finally, the VMM sets the real CR3 to point to the shadow page table.

   Of course, the guest operating system should not modify its page table directly, rather the VMM needs to intercept this attempt and update the shadow page table accordingly. The underlying idea is to make guest page table pages as read-only (in the shadow page table) and manage corresponding traps. This technique is called *memory tracing*.

Now, consider the case in which there is a page fault involved. The three actors participating will be `g.user`, triggering the actual page fault, `g.supervisor`, acting as an intermediary between `g.user` and `h.supervisor`, and `h.supervisor`, updating the shadow page table. Moreover, on a ”simple” page fault, many privileged instructions are involved:

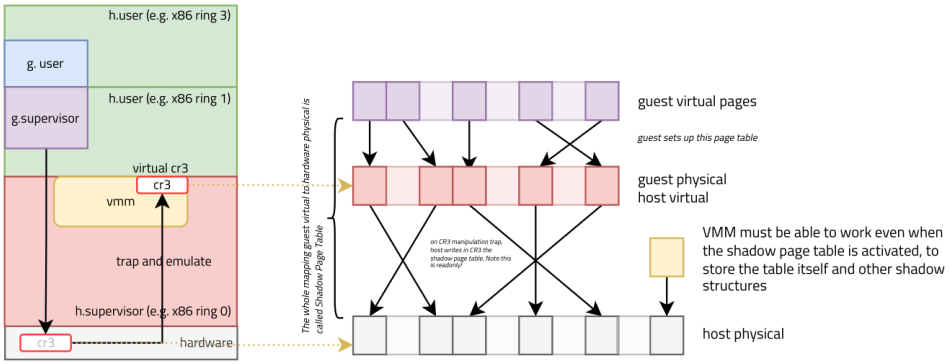


Figure 77: Shadow page table structure

- Set the MMU root;
- Invalidate the TLB entry;
- Modify the page table.

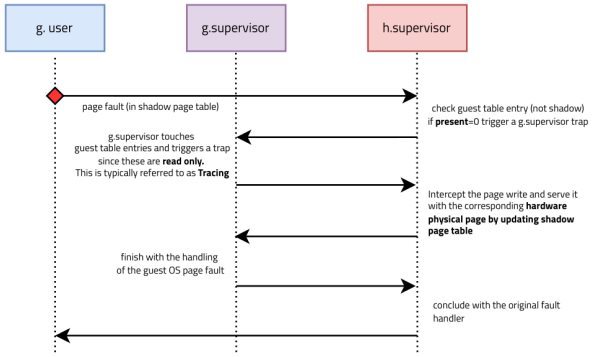


Figure 78: An example of page fault

However, a security problem might arise in a scenario like the one depicted in figure. In such a setting, `g.user` could read `g.supervisor` memory, since such an action will not produce a trap. However, we can notice at least a change of privilege (i.e., `g.user` $\rightarrow$ `g.supervisor`), so we can keep two

shadow page tables, one for `g.user` and one for `g.supervisor` and the former will not have all the mappings for `g.supervisor`.

13 Lecture 17 - Virtualization II

Problems with trap-and-emulate

Trap-and-emulate lets the virtualization setting to work, but has known problems. There are, as we pointed out, some architectures having virtualization-sensitive unprivileged instructions. For instance, in Intel x86:

- Instructions manipulating the interrupt flags, `h.supervisor` can't track the state of interrupts correctly;
- Reading and writing segment descriptors and registers. For instance, `g.supervisor` can see that it has been de-privileged by reading the current privilege level (CPL).

On Intel x86 we have 4 kinds of unprivileged virtualization-sensitive instructions [64]:

- `pushf`, `popf`, `iret`: instructions manipulating the interrupt flag (i.e., `%eflags.if`) are `nop` instructions if executed in user mode. This means that they do not trap. The guest operating system would think it has disabled interrupts, when it didn't;
- `lar`, `verr`, `verw`, `lsl`: provide visibility into segment descriptors in the global/local descriptor table. These instructions would access the VMM's tables (rather than the one managed by the operating system), thereby confusing the software;
- `pop <seg>`, `push <seg>`, `mov <seg>`: manipulate segment registers. This is problematic because 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 behavior if `push %cs` was 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

[64] For more details, see http://www.cs.columbia.edu/~cdall/candidacy/pdf/Bugnion2012.pdf.

not those specified by the virtual machine's operating system.

Thus, there are several problems with pure trap-and-emulate:

- *Ring aliasing*: a guest operating system could easily determine that it is not running at supervisor privilege level;
- *Address space compression*: operating systems expect to have access to the processor's full virtual address space. However, the VMM must have a minimal amount of pages allocated in the guest address space to manage traps;
- *Excessive faulting*: on x86, `sysenter` and `sysexit` are used for each system call, but they trap into the VMM any time they are executed by the guest operating system. When trapping into the VMM, they will raise interrupts themselves, yielding to excessive trapping.

Binary translation as a solution

The first idea of virtualization providers was to use a so-called *binary translator*, which converts an input binary instruction sequence into a second binary instruction sequence that can execute natively on the target system. When VMWare Workstation first shipped, the binary translator consisted of approximately 27 thousands line of C source code (45% of the total VMM line count).

A dynamic binary translator performs the translation at runtime by storing the target sequences into a buffer called *translation cache*.

It consists in a loop where a dispatch function is invoked:

- The dispatch function looked up the location in the translation cache corresponding to the current state of the virtual CPU;
- If none was found, it invoked the translator, which first decoded a short instruction sequence (no longer than a basic block) starting at the current virtual CPU instruction pointer, generated a corresponding (but translated) instruction sequence and stored it in the translation cache;

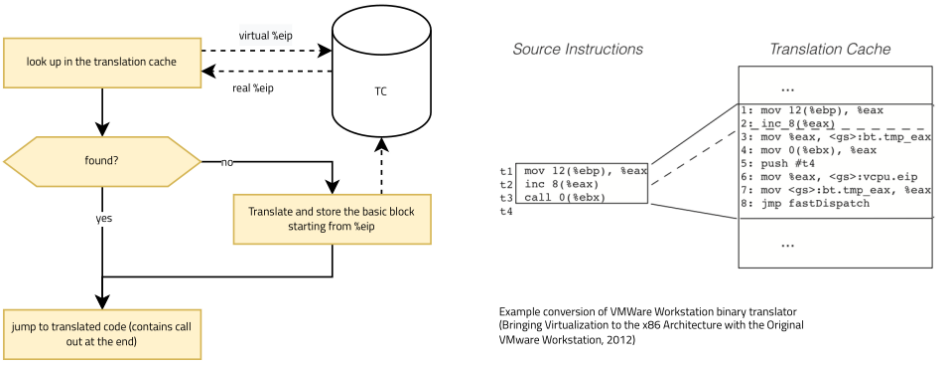


Figure 79: Binary translation in VMWare

- The dispatch function then transferred control to the location in the translation cache. The translated code consisted of native x86 instructions within the translation cache, and could encode calls to support routines. These callouts typically invoked the dispatcher again, to close the loop.

In particular, virtualization sensitive instructions are translated into specialized inlined sequences.

Hardware assisted virtualization

The idea behind hardware assisted virtualization is such that sensitive (but not privileged) instructions become privileged, in order for the hypervisor to handle all the corner cases we already discussed. Let's first analyze the main goals of hardware assisted virtualization:

- Avoid problems of deprivileging, which means that `g.supervisor` works as a traditional supervisor (with no ring aliasing), by adding new modes;
- Allow the state of the guest to be explicitly and comprehensively saved and resumed (used in some shadow structure).

Secondary goals for hardware assisted virtualization are the following:

- On x86, finally obey to Popek and Goldberg's requirements;

- Improve performance by reducing the number of traps from system calls and interrupts and by avoiding the shadow paging overhead.

In particular, they introduced a new mode using hardware facilities to choose which privileged instructions are actually sensitive and which, instead, can be managed without an hypervisor call.

Solutions proposed by vendors

Vendors proposed their own way to offer hardware assisted virtualization. On ARM and PowerPC the hypervisor mode has its own rule. In particular, the hypervisor lives in its own privileged mode, which is different from the supervisor and user one. On ARM there is a different page table format, while in PowerPC there is no paging at all. Intel (on VT-x) introduced two orthogonal privilege modes mirroring the original ones, while RISCV does an hybrid between the two (the hypervisor level is shared between ”non virtualized” and ”guest”).

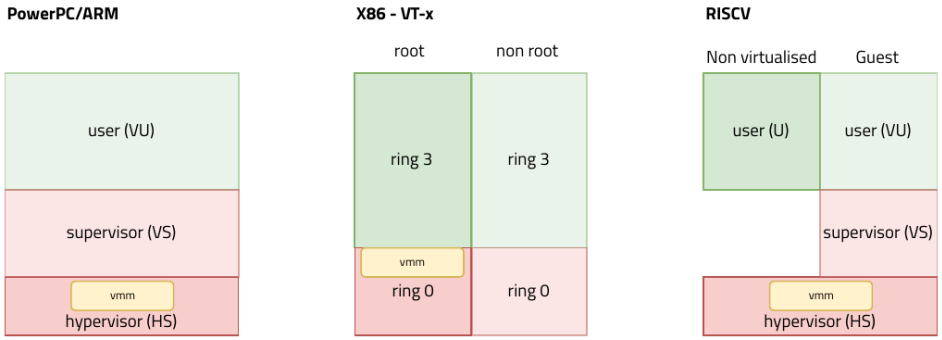


Figure 80: Hardware assisted virtualization proposed by different vendors

Note that *”non root”* in x86 is equivalent to ”guest” operating system. In such a setting, whenever the user code does a system call, given that levels are explicit now, those system calls can be

managed without incurring in the VMM. On x86 and RISCV, hypervisor mode is a superset of regular supervisor mode. Moreover, in x86 a virtual machine control block holds the state of the entire guest, so that context switches are performed by the processor with a special instruction. On ARM, PowerPC and RISCV, special processor registers hold minimal information on the guest.

Comparing with software assisted virtualization

Hardware and software assisted virtualizations have their own strengths and drawbacks [65]. In particular, hardware virtualization has several advantages:

- When having many system calls, hardware prevails because such calls run without the VMM intervention;
- Recovering the guest state to manage traps is easier with support from hardware;
- Having no binary translation means less overhead and less memory.

These strengths come with costs and drawbacks:

- Some instructions need to trap, be fetched and executed by the VMM, while in software they can be directly transformed into an emulation routine (e.g., memory tracing);
- I/O requires a full VMM-guest round trip while in software they can be operated on a virtual chipset;
- The control path on page faults (which is similar for both hardware and software) could have a higher overhead in hardware with respect to software.

[65] For mode details, see https://www.vmware.com/pdf/asplos235_adams.pdf.

Speeding virtualization up

Now let's analyze all those techniques that have been used to speed up the performance in virtualization.

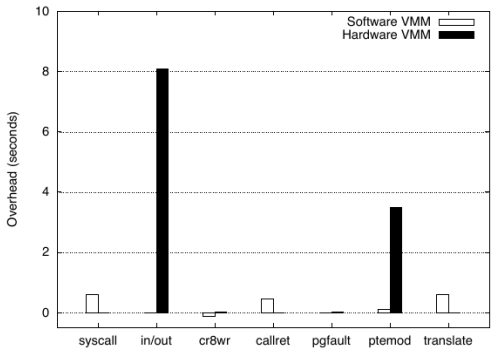


Figure 81: Overhead between software and hardware VMM

Extended page tables

Using the traditional way of managing page tables, we would have only a single key register which is used: the MMU consults just the PGD contained in the host `CR3`; the guest `CR3` is just an illusion. The PGD pointed by `CR3` thus contains the full translation from the guest operating system's virtual pages down to host's physical pages. Whenever the guest operating system wants to modify something in its own page table translation, the VMM traps this interaction to keep the entire virtual memory view consistent. Note that this technique also applies to hardware based virtualization. In some sense, this is equivalent, in terms of overhead, to have a shadow pagle table to be consistent with the view of the guest operating system.

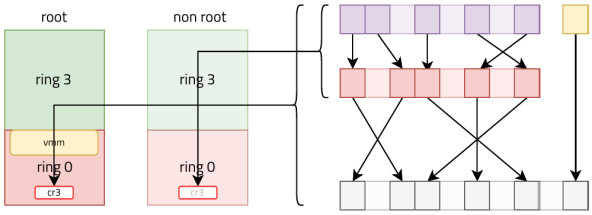


Figure 82: The traditional way of managing page tables

With hardware based translation introduced by Intel, there is also a new way for avoiding to trap always into the VMM whenever

dealing with page tables. This new technique is called *extended page table*. There will be two actual key registers in the machine and the host MMU now holds two pointers:

- `g.cr3`, containing guest-virtual to host-virtual (i.e., guest physical) mappings;
- `h.cr3`, containing **only** host-virtual to host-physical mappings.

In such a setting, the guest's page table pointer has more authority: `g.cr3` is manipulated directly by the guest, with no intervention from the VMM. The PGD pointed by `h.cr3` stores the translation from guest's physical pages down to host's physical pages [66]. This means that guest's physical pages don't need to be protected in terms of reads and writes.

[66] In such a setting, host's physical pages are basically treated as host's virtual pages by the host's PDG itself.

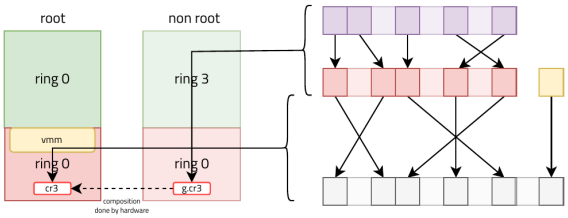


Figure 83: Extended page tables

I/O passthrough

Virtual machines often make use of direct device access (i.e., "device assignment") when configured for the highest possible I/O performance. From a device and host perspective, this simply turns the VM into a userspace driver, with the benefits of significantly reduced latency, higher bandwidth, and direct use of bare-metal device drivers.

In particular, to exploit such a functionality, we should have the root Linux kernel booted with `io_mmu = on`, detach the specific device from its traditional driver and bind it to a *VFIO* driver. When launching the "guest", the VMM must configure it to use such "vfio" interface to connect to the device by setting up the *IOMMU* (using `ioctl()` on `sysfs`). Interrupts are remapped by

VFIO to an `eventfd` object, which is more or less a socket used for evented IPC. The VFIO driver exposes direct device access to userspace, in a secure IOMMU protected environment. In other words, this allows safe non-privileged, userspace drivers.

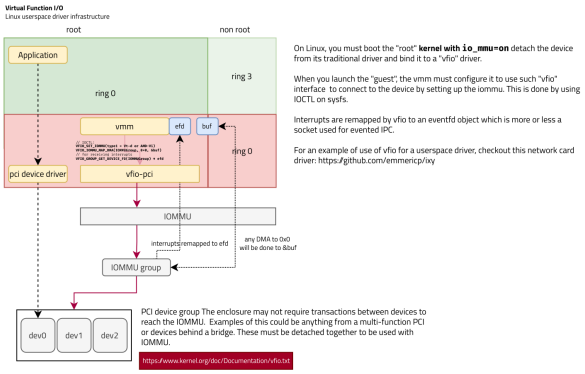


Figure 84: I/O passthrough mechanism

Many modern systems now provide DMA and interrupt remapping facilities to help ensure I/O devices behave within the boundaries they are provided with. This includes x86 hardware with AMD-Vi and Intel VT-d and PowerPC systems with Partitionable Endpoints (PEs).

KVM

KVM is a loadable kernel module that takes away the problem of creating our own VMM [67]. It consists of a module that provides the core virtualization infrastructure and a processor specific module (e.g., `kvm-intel.ko`, `kvm-amd.ko`). KVM was originally a forked version of QEMU and was provided to launch guests and deal with hardware emulation that isn't handled by the kernel. That support was eventually merged into the upstream project.

[67] For a practical example, see https://lettieri.iet.unipi.it/virtualization/2018/kvm.cc.

Paravirtualization

Paravirtualization has as its main goal to go even beyond virtualization and deals with changing the target guest and replacing the

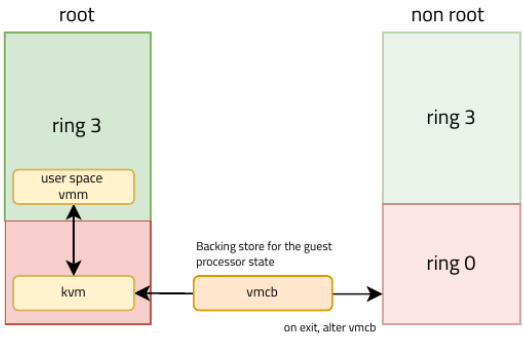


Figure 85: The KVM architecture

difficult-to-virtualize instructions with something else. Rewriting all the target's software from scratch is out of question, but porting a well designed kernel to a new architecture is a matter of replacing the system-specific routines and recompile the rest (e.g., instead of accessing the MMU, the paravirtual kernel could call the VMM for write access to page tables). This technique was first introduced in the XEN hypervisor, but nowadays it is mostly used within drivers in the guest.

VirtIO

VirtIO is a specification for writing device drivers targeting the guest that access directly the host. We introduce it by looking at differences between the standard way of emulating a device and the paravirtualized case.

Consider now a Network Interface Controller (NIC) and the way in which it would be traditionally handled in virtual machines as an emulated peripheral. The *non-root* machine will behave as a *root* one, not knowing that there is an underlying host. Given packet queues handled by the kernel, at some point the kernel will write into some hardware registers, which are the ring buffer head/tail hardware registers (i.e., `TDT|TDH` and `RDT|RDH`). However, any read and write to this hardware registers triggers a VM-exit (to write into the actual hardware registers accessed by the host). In particular, KVM must read registers, reconstruct packets and send them to the host network, which is not minimal as far as the overhead is concerned. This case is depicted in figure below.

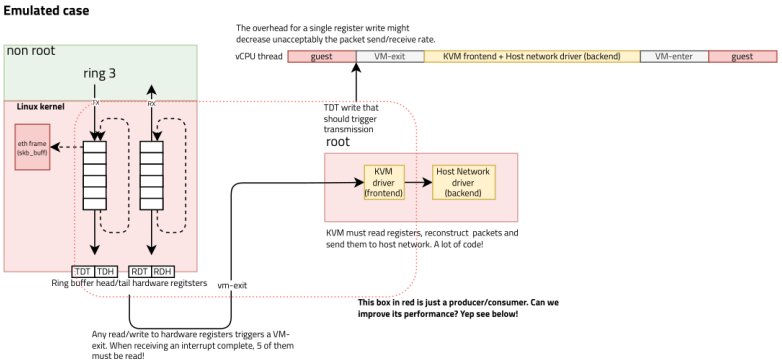


Figure 86: The emulated case

Now, speaking about the *VirtIO* case, we have a new driver called *virtio-net* that communicates to the host through an interface called *guest virtio*. In particular, it has this direct connection by writing into a queue (i.e., *virtqueue*) which is different from what has been done before (writing to registers). This write raises a trap, to inform the host that a message is coming. The host operating system reads that specific message and write it to the host network driver. Doing this, we are minimizing the time of the *vCPU thread* waiting for the packet to be sent to the host peripheral. This description is depicted in figure below.

Paravirtualization is a must when speeding up the virtual machine.

Containerization

Containers are a way to isolate a set of processes and make them think 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). However, keep always in mind that **containers are not virtual machines**: processes running inside a container are normal processes executing on the host kernel,

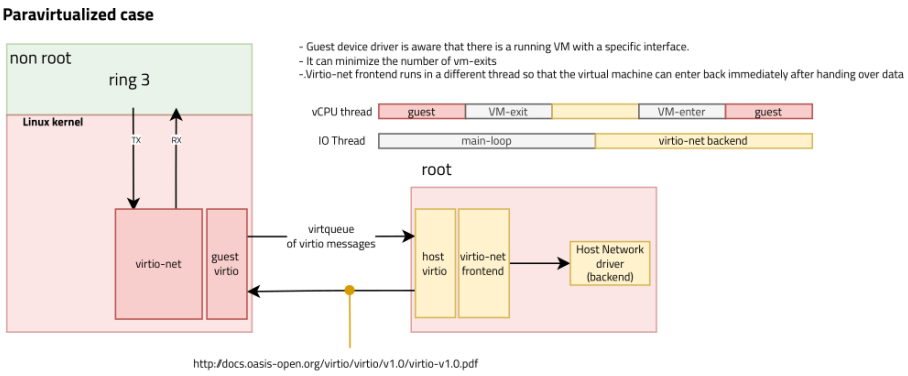


Figure 87: The paravirtualized case

thus there is no guest kernel running inside the container. This also means that we can't run an arbitrary operating system in a container, since the kernel is shared with the host (i.e., Linux in our case).

The most important advantage of containers with respect to virtual machines is performance: there is no performance penalty in running an application inside a container compared to running it on the host. Linux containers are implemented using two distinct kernel features: **namespaces** and **control groups**.

In terms of *security*, there is a huge debate between the virtual machines being more secure than containers and vice versa. Virtual machines are considered (by some) to be more secure than containers, since they have a smaller *attack surface* [68], which is the *hypervisor* (vs. the *entire kernel* for containers). Taking KVM, it is a kernel module and uses some facilities from the rest of the Linux kernel (e.g., for scheduling, virtual memory etc.). However, this is still less than the amount of features and code used to implement containers.

[68] With *attack surface* we mean the amount of code and features that a malicious attacker may probe to find exploitable bugs.

Namespaces

Namespaces provide a means to segregate system resources. In some sense, this is similar to what the (by now, old) `chroot()` syscall does:

- For each process, the kernel remembers the inode of the process *root directory*;

- This directory is used as a starting point whenever the process passes the kernel a filesystem path that starts with `"/"` (e.g., in a `open()`);
- 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 [69];
- Only *root* can call `chroot()`;
- The process root directory is inherited by its children.

[69] In other words, we are just introducing a boundary that can't be climbed. This boundary the root from where all the path inspection of directories starts.

To summarize, by using `chroot()` we can make a subset of the filesystem look like it was the full filesystem for a set of processes. *Chroot* environments, however, are not full containers. Contrary to popular belief, in fact, **not everything is a file in Unix** (e.g., network interfaces, network ports, users and processes are not files). While we can have as many instances as we want of, say, `/etc/passwd`, each different and living in its own *chroot* environment, 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), and user and process IDs will have the same meaning in all chroot environments.

Now, *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 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. 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.

Control groups

While namespaces can be used to hide and create private copies of all the system entities, they are not sufficient in isolating sets of processes so that they cannot interfere with each other. Processes may interfere also by abusing the system resources, e.g., allocating

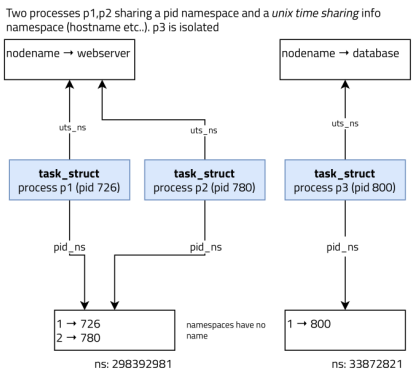


Figure 88: Namespaces functioning

too much memory, using too much CPU time, or disk and network bandwidth. To properly implement containers, therefore, we also need to limit the usage of resources by processes living in the container. Control groups are groups explicitly created by the administrator, who can later assign processes to them. The administrator may setup the system so that processes cannot escape their control group. Control groups can be organized in a *tree-shaped* hierarchy: each process in the system must belong to exactly one control group in the hierarchy, and therefore the hierarchy is a *partition* of the system processes. When a process creates a child process, the child inherits the control group of its parent. Note that some cgroups may be empty. Indeed, it is good practice to put processes only in the root cgroup and in the leaf cgroups, leaving all intermediate *cgroups* empty [70]. Once we have the ability to reliably group processes in one or more hierarchy, we can start controlling their resource usage. To this aim, hierarchies can be linked to so-called *subsystems* (*subsystem controllers* would be a better name). Subsystems are used to control the resources assigned to the cgroups in the linked hierarchy. Some examples of existing subsystems are:

- *Memory*: limits the amount of main memory used by each *cgroup*;
- *CPU*: limits the maximum fraction of CPU that each *cgroup* may use and may schedule the CPU based on *cgroups* weights;
- *CPUacct*: this is not much of a controller, since it only provides accounting information about the CPU usage of the

[70] This procedure has become mandatory in the newest version of *cgroup*, which is *version 2*.

cgroups;

- *CPUset*: on multi-CPU systems, limits the set of CPUs that may be used by each *cgroup*;
- *PIDs*: limits the number of processes that can be created in a *cgroup*.

The are also other subsystems that control device access, block I/O etc. To have a practical overview of how to use namespaces, see also https://lettieri.iet.unipi.it/virtualization/2018/containers.pdf.

14 Lecture 18 - I/O Devices and Drivers I

Introduction to hardware peripherals

In a more classical view of system device architecture, the CPU is connected to a shared bus to interact with other devices (e.g., memory and peripherals), but this turns out to be not feasible in practice: the shared bus would introduce several delays for the CPU to interact with external peripherals. What we have, instead, is a more point-to-point approach: the CPU is connected to the memory using a *DDR* interface and to graphics cards with *PCIe* connections [71]. There is also an external hub called *I/O chipset* (connected to the CPU with a *DMI* interface) which manages the connection to all the other devices both through PCIe connection (e.g., NVMe SSD, network cards) and through SATA (e.g., SATA SSDs and HDDs). The PCI specification provides a way to the operating system to look at the topology of the system itself, which is very useful in terms of peripherals management. The I/O chipset is also used for USB connection, which is a different standard, and serial ports (i.e., RS232).

[71] For its very own nature, PCI is a serial bus and has several lanes to connect, e.g., graphic cards to the CPU.

In terms of communication, the CPU read/writes from/to peripherals using their so-called *device ports*, which can be characterized both by a certain memory address to which the CPU writes or reads (with normal `load` and `store` instructions) or by specific I/O ports addresses that are manipulated by the CPU with specialized instructions.

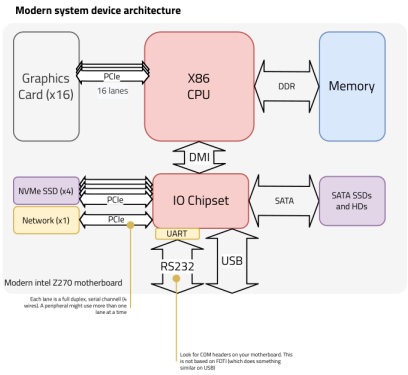


Figure 89: Modern system device architecture

Now, speaking about serial ports, we have two wires that connect the device to the actual motherboard, but concerning the CPU the device port (UART) is seen as a set of registers and, in turn, these are seen as *memory locations* or *port locations*, as said before. Apart from the transmit and receive registers (i.e., *THB* and *RBR*) there are several configuration registers. Among these there is the *IIR* register that sends an interrupt to the CPU on character received and/or sent.

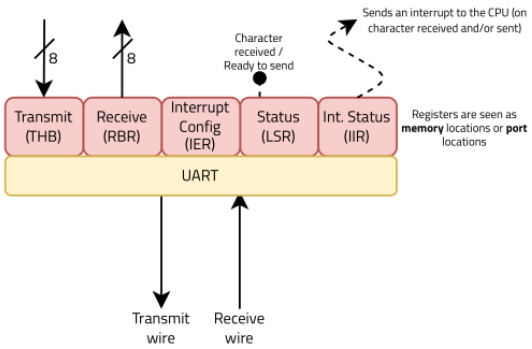


Figure 90: Example of a device port (UART)

Mechanisms to interact with devices

There are basically two ways in which the CPU can interact with external devices:

- *I/O ports*: they provide explicit I/O instructions [72]
- *Memory-mapped I/O*: the hardware makes device registers available as they were (virtual) memory locations, without the need to use privileged instructions to interact with them.

[72] On Intel x86 processors, the `in` and `out` (which are privileged) instructions can be used to communicate with devices. Each device is assigned to a `port` number in the I/O address space, which names the device.

I/O instructions

For example, to read the character received by UART into `al` we could do:

```
in RBR, %al   ;; RBR = 0×2f8
```

The port number in `RBR` is limited to be enclosed in range $[0, 2^{16}]$, which is quite limited in modern architectures. With memory-mapped I/O we can overcome this limitation.

Memory-mapped I/O

As we said before, `in/out` instructions are slow and clunky: the instruction format restricts what registers can be used and only allows 2^{16} different port numbers. Moreover, per-port access control turns out not to be useful (any port access allows to disable all interrupts). Devices can achieve the same effect with physical addresses: the operating system must map physical to virtual addresses, ensure non-cachable pages and to assign physical addresses at boot to avoid conflicts. For instance, we could have an interaction implemented as follows:

```
volatile int32_t *device_control = (int32_t *) (0×c0100 + PHYS_BASE);
*device control = 0×80;
int32_t status = *device_control;
```

In this way, we are pointing to a device port to write to it (by writing a pointer), and then we read the status from it. This is all managed using pointers and not exploiting privileged instructions accessing device registers directly.

Device to CPU communication

The device can communicate data to the CPU in two fundamental ways:

- *Polling*;
- *Interrupts*.

Polling

Polling is basically a loop performed by the CPU (i.e., the CPU spins) to wait for data or action to be finished. For instance,

```
while((inb(LSR) & (1<<5)) == 0)
// wait until the UART is ready to send;
outb(c, THR); // send character
```

In this case we are waiting for a certain register (i.e., `LSR`) to contain some bits indicating that the device is ready to accept data. When the device is slow, polling wastes cycles. If the device is fast, instead, it is quite inexpensive: for instance, there is no saving of registers.

Interrupts

Instead of polling the device repeatedly, the operating system can issue a request, put the calling process to sleep, and context switch to another task. When the device is finally finished with the operation, it will raise a *hardware interrupt*, causing the CPU to jump into the operating system at a predetermined *interrupt service routine* (ISR), or more simply, an *interrupt handler*.

Interrupt makes sense for slow devices, otherwise the cost of interrupt handling might be more than what we expect [73]. When there are too many interrupts, the operating system might **livelock**, that is, find itself only processing interrupts and never allowing a user-space process to run and actually serve requests. In fact, some devices generate events faster than one per microsecond (e.g., a gigabit Ethernet can deliver 1.5 million small packets per second): an old approach was that every event caused an interrupt (simple hardware, smarter software), but nowadays the hardware completes lots of work before interrupting. To enable interrupts, an interrupt controlled needs to be programmed. Program an interrupt controller means to set up how the system should interpret

[73] Note that an interrupt takes on the order of a microsecond (we need to save/restore the state, deal with cache misses, etc.).

signals from devices (edge triggered, level triggered, etc.) and how hardware signal numbering should be mapped to interrupt handlers.

Directy Memory Access (DMA)

When using programmed I/O (PIO) to transfer a large chunk of data to a device, the CPU is overburdened with a rather trivial task and might waste time. This is why and additional device that can orchestrate transfers between devices and memory without CPU intervention has been introduced. This device is called *Direct Memory Access* (DMA). In figure below, there is the original interaction between an external device and the CPU. This will be useful when compared with the DMA approach.

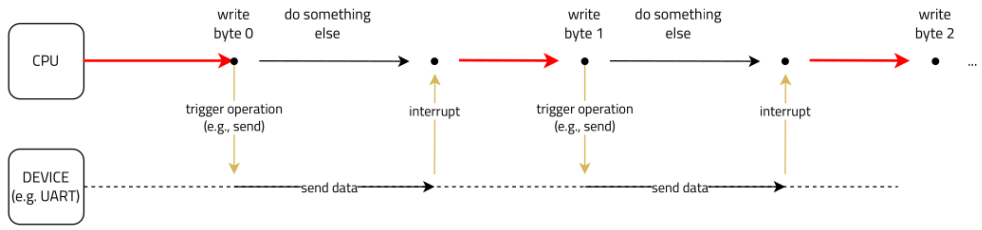


Figure 91: The original CPU-device interaction

To wrap up, in the original interaction the CPU writes a single byte triggering the device to send the actual data. In the meantime the CPU does something else, until an interrupt arrives, and so on and so forth.

In DMA, instead, we have the CPU sending a command such as "send x bytes from memory location M to byte register TX of UART" to the DMA controller. The entire interaction is now handled by the latter, sending signals to the CPU on *half-transmit complete* (HTC) and *transmit-complete* (TC) only. PCI allows any peripheral to have a DMA and this goes under the name of *first party DMA*.

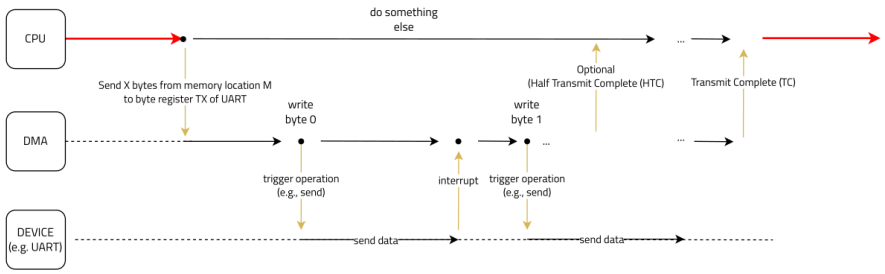


Figure 92: The DMA approach

Linux interrupts

There are basically two different types of interrupts:

- *Asynchronous interrupts* (i.e., I/O interrupts), which are generated by other hardware devices at arbitrary times with respect to the CPU clock signals. These interrupts could be *maskable* and *unmaskable*:
  - *Maskable* interrupts are all Interrupt Requests (IRQs) issued by the I/O. A masked interrupt is ignored by the control unit as long as it remains masked;
  - *Non-maskable* interrupts are, instead, always recognized by the CPU;
- *Synchronous interrupts* (i.e., exceptions), which are produced by the CPU's control unit while executing instructions. They are called synchronous because the control unit issues them only after terminating the execution of an instruction. These exceptions can be divided in:
  - *Fault exceptions*: they imply to correct and re-execute the faulting instruction;
  - *Traps*: they imply to not re-execute the excepting instruction.

Interrupts are issued by interval timers and I/O devices (e.g., the arrival of a keystroke from a user sets off an interrupt). Exceptions, on the other hand, are caused either by programming errors or by anomalous conditions that must be handled by the kernel.

Note that the code executed by an interrupt or by an exception is not a process. Rather, it is a kernel control path that runs at the expense of the same process that was running when the interrupt occurred (i.e., it runs *on behalf of a process*).

Finally, some examples of maskable interrupts could be non-recoverable hardware errors, as well as a corruption in system memory such as parity and ECC errors or data corruption detected on system and peripheral buses. Exceptions, instead, on an Intel x86 machine could be division by zero, debug [74], breakpoint (i.e., `INT3` instruction), invalid opcode, protection fault, page fault, etc.

[74] The address of an instruction or operand may fall within a range of an active debug register.

The interrupt flow

On Linux, each interrupt or exception is identified by a number ranging from 0 to 255 (called **vector**). In particular, on Intel's x86 machines, each vector $32 + n$ is called **IRQ n** and is associated with I/O interrupts. IRQs are external interrupts and constitute a subset of interrupt vectors. They are remapped by the Programmable Interface Controller (PIC) into a vector number, as we already said. This vector number has a corresponding entry into a table, called *Interrupt Descriptor Table* (IDT). This table contains handlers for all vectors and is pointed by the `idtr` machine register. Each entry identifies a segment selector and offset (i.e., `entry_ptr = segment_selector + offset`) pointing to a so-called Interrupt Service Routine (ISR), a function whose job is to handle the interrupt. As already said, this service routine is within a segment specified in the GDT. Thus, for each vector Linux registers a routine called `do_irq(n)` that invokes all the actions that device drivers have registered to be executed when that specific interrupt request comes. Typically, the first part of this routine disables all interrupts and Linux avoids nested interrupts of the same number by using a method of PIC actions called `ack()`. Then, the handler is actually executed, followed by a call to `end()` which re-enables interrupts (i.e., it is the revert of `ack()`). Certainly, all these actions are executed in kernel-mode.

Note that, on interrupt arrival, the processor switches to supervisor, in the context of the current task's supervisor stack. In the new stack, it saves:

- The interrupt vector number;

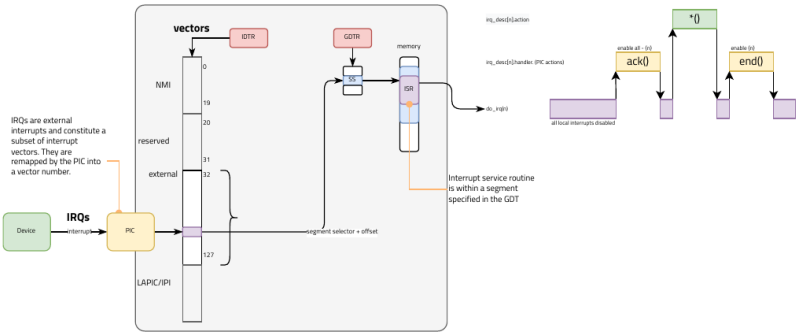


Figure 93: The Linux interrupt flow

- The return instruction pointer;
- The old CS segment selector;
- The old values of SS and ESP.

The following piece of code is an example of registering an action within the IRQ subsystem:

```
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)
{
// allocate space for mydata
ret = request_irq(irqnum, handler, IRQF_SHA, mydata);
// ...
}
```

Programmable Interrupt Controller

The PIC is an external circuit for which Linux has appropriate interfaces to interact with. Legacy PIC had a number of pins that could be used to inform it that there is a device wanting to generate an interrupt to be executed on the CPU. The CPU would receive that interrupt through a single signal (single line) and once it received the interrupt it used a very specific interrupt bus (directly connected to the PIC) to read data associated with the interrupt itself. We could even use multiple PIC of this type in our architecture at that time. To summarize, legacy PICs were allowed to use only 1 CPU interrupt pin (`INTR`) to manage 8 interrupts and could be used in a chain (master/slave) of two, increasing IRQs to 15. Note that, upon a multiple signals arrival through the input lines, the PIC decided to choose the one with the lower interrupt number.

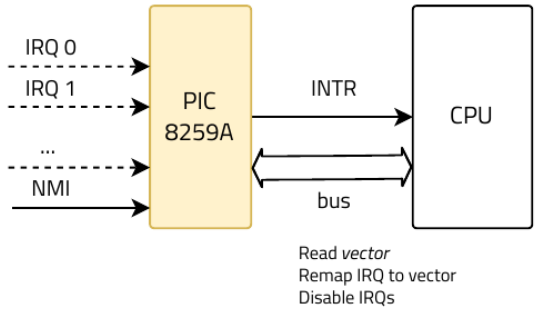


Figure 94: Legacy PIC architecture

Having 8 to 15 IRQs became very soon a limit and has been surpassed even because we started to have multiple processors dealing with multiple devices. The Advanced Programmable Interrupt Controller (APIC) was introduced for that matter. With APIC, we could have 24 interrupts divided as follows:

$$\begin{cases} \text{ISA} \rightarrow [0, \dots, 15] \\ \text{PCIdevices} \rightarrow [16, \dots, 23]. \end{cases} \tag{7}$$

With *xAPIC* specification we could even reach 256 different interrupts. These interrupts are structured as follows: each CPU has its own circuitery for managing interrupts (called *Local Programmable Interrupt Controller* - LPIC) connected to a APIC bus so that we could program the main APIC as a router to map interrupt request to specific CPUs. In this way we can also program the APIC to distribute the workload when managing a certain interrupt. In other words, it acts as a router with respect to local APICs and can be programmed. Moreover, for logical/low priority+ operations it can deliver interrupts to multiple cores in a round robin fashion. Each CPU could interact with other CPUs through the generation of *inter-processor interrupts*, enabling a sort of message passing between CPUs. Interrupts may also arrive unordered with respect to writes done by a device through DMA [75].

[75] This brings a severe consistencty problem.

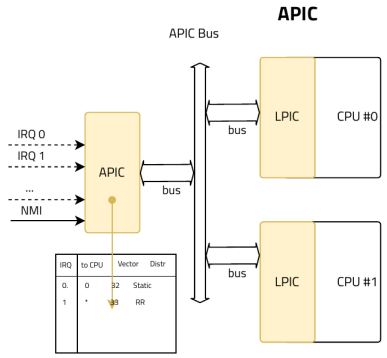


Figure 95: APIC architecture

The APIC architecture, for some interrupts, has been substituted by a messaging mechanism through PCIe, whose idea is to remove physical wires from devices and use PCIe buses to write messages instead. Each device can produce up to 32 interrupt types and these are synchronous with respect to data read/write to memory.

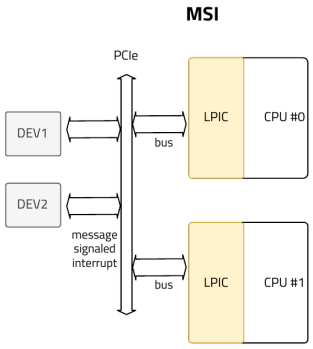


Figure 96: MSI architecture

Deferring work

When executing interrupts, we want to avoid as much as possible to do all the necessary work right after the interrupt had arrived. The idea of *deferring work* is to move the non-critical mangement of interrupts to a later time. It might also benefits from aggregation, in the sense that we would have many small operations merged into a single one.

Originally, every interrupt management was divided into two levels:

- *Top half*: executes a minimal amount of work which is mandatory to later finalize the whole interrupt management [76] and schedules some deferred work (i.e., *deferred functions*);
- *Bottom half*: finalizes the work by deferred functions from a queue and executes them. Bottom halves are invoked in particular *reconciliation points* in time.

[76] This works in a **non-interruptible** scheme.

There are basically three ways to perform the deferring work:

- SoftIRQs (main mechanism, rarely used alone);
- Tasklets;
- Work queues.

15 Lecture 23 - Seminar on Software Verification I

Introduction to Software Verification

Software verification is a very classical problem: we write a program and we want to be sure that it is *correct*. From the theory we know that this problem is *undecidable* (from Rice's theorem). However, we actually need to prove correctness in particular situations (in *safety-critical* systems), where errors can cost a huge quantity of money, or even lives. The most used technique to have some guarantees is *testing*, but of course is very limited: it can only prove that a system has errors, not its correctness.

Operating systems are among the most critical examples of software and so verification is often used for plenty of aspects of an operating system. There are two typical properties which are checked:

- *Security*: for instance, we want to check if the operating system is vulnerable to some attacks (e.g., code injection, buffer overflow);
- *Safety*: for instance, we want the operating system to be reliable and free of crashes, or that it is responsive enough in some specific real-time environment (e.g., a control system of an aircraft).

Main approaches

There are basically two main approaches to the problem:

- *Model checking*: totally automatic, but has necessary limitations on the expressivity of used notations (remember that the problem is non-decidable);
- *Automated theorem proving*: it is more general, but kind of "user-driven" [77].

[77] These tools are often called "proof-checkers".

1. Model checking This technique was introduced in the '80s for a "push-button" verification. Its basic idea is that we have a model M of the system and one of its properties, called ϕ (usually written $M \models \phi$). Our model M is still an abstraction of the original system we have. Traditionally, M is written in some variant/extension of *finite state automata* (thus

very limited) in some temporal logic, typically *linear temporal logic* (LTL). With these limitations, we are working with *regular languages*, hence we could leverage on their various nice closure properties.

The hard part is the logic: ϕ is written, for instance, in LTL, which corresponds to an *aperiodic regular languages* (or *star-free languages*, i.e., regular expressions without the "$*$" operation), which is a subset of regular languages. There are algorithms to translate $\neg\phi$ in an equivalent automaton A_ϕ, albeit with a (exponential) *state blow-up*, from going from LTL to non-deterministic FSA. Having the FSA, the remaining part is easier: we build the automaton $M \cap A_\phi$ and we check whether the resulting language is empty:

- If yes, then we proved that $M \models \phi$;
- If no, this means that $\exists\, x \ \in \ L(M \cap A_\phi)$ and x is a *counter-example*.

This procedure turns out to be *PSPACE-complete* (polynomial deterministic), which means that it is exponential in time. Note that, typically we have a very small ϕ with respect to the entire model M, thus the exponential growing is limited.

There was a lot of research on these techniques, with two main aims:

- To cover *more expressive* notations;
- To get faster and/or use *less memory*.

The typical problem was that the resulting automaton was too big to fit in memory. The main idea was to build up only *portions* of it (i.e., *on-the-fly*). Another approach was the so-called *bounded model-checking*, according to which, instead of using automata, we could translate everything in some kind of weaker logic, by adding a temporal bound:

- Originally, it was *propositional logic*, and then we used a SAT-solver;
- More recently, a *satisfiability modulo theory* (SMT), then we use a SMT-solver (e.g., Z3).

The idea to use automata has been extended by using *pushdown automata* (PDA), which is more suitable to represent programs than a FSA, thanks to their stack. The most successful regular language is the *visibility pushdown* (VP): a subclass of deterministic context-free languages which enjoy most of the nice properties of regular languages. Hence, they were proposed for model checking procedural programs. However, there is a lot of theory behind this notations, but a lack of useful tools to use it.

More recently, there has been the rise of a class of languages called *operator precedence* (OP) that generalizes VP, retaining all their properties. OP are much more expressive than VP and can model complex stack behavior such as those, for instance, of exceptions and continuations, where part of the stack can be discarded because of some operations.

2. Automated theorem proving The other big family of approaches are based on the idea of actually building a general *formal proof* that the software has a particular property (as a theorem). This is indeed a quite natural approach and the offered logic is very expressive, but very often *undecidable*. For this reason, we need *user guidance*. In practice, it is the user that actually writes the proof, but some tools can automate simpler passages and can check every step (i.e., *proof checker*).

   There is a connection between writing programs and writing proofs: many of the recent proof checkers are systems for functional programming languages (they are based on this languages) and they are dependently typed. In this settings, types are very expressive and can be used to write *complex properties*: if we write the code of a function f with a type expressing some property ϕ, then we are assured that f enjoys ϕ [78]. The interesting connection between proving theorems and writing programs is formalized by the *Curry-Howard Isomorphism*.

[78] Note that this is also true for standard typed languages, but, not surprisingly, the properties we can express are quite limited: for instance, we could use propositional logic for simple types, a very basic predicate logic for parametric types.

Memory safety

Memory safety characterizes programs where, in any possible scenario, memory allocation/use and de-allocation can never compromise the functionality of the program. Memory access is always well defined. The problem with C++ and Rust is that they

need to manage memory directly. In particular, system-level languages can't afford a garbage ollector and thus they are usually not memory-safe. However, unlike C++, Rust has default *compile-time* and *run-time* mechanisms for ensuring memory safety. Both problems related to temporal memory safety (e.g., use-after-free, double-free) and to spatial memory safety (e.g., buffer overflow) are ensured by Rust.

Memory safety bugs can always be seen as ownership problems. When a variable has a pointer to a region of memory we say that **it owns it**. Large programs might use several variables owning the same region of memory at the very same time. In general, this is not a bug but we need to ensure to deallocate only when the last owner exited. Otherwise, we would get double-free or use-after-free bugs. Moreover, some programs might have memory regions not owned by a variable. These are called *memory leaks* because they can't be reclaimed.

Memory leaks

A memory leak is a pointer to a memory region with no owner. Essentially, the region will never be de-allocated because the program forgot about it [79]. Consider now the following example:

[79] Of course, one way to solve it is to use the `free()` or `delete` function, respectively in C and C++, to de-allocate the memory region.

```
int vuln(void)
{
char *s1 = (char *) malloc(0x14);
// ...
}
// the buffer is never freed
```

The buffer is never freed, which means that if calling this function multiple times we will end up having a lot of memory areas that won't be ever freed.

C++ introduced *RAII*, which stands for *Resource Acquisition Is Initialization* and it is a form of mitigation for memory leaks. In particular, it is a programming idiom that ties *automatic variables* (i.e., variables allocated on the stack) to the acquisition and release of resources on the heap. De-allocation is typically done implicitly when the variable goes out of scope. Such type of variables can be seen in forms of smart pointers (i.e., `shared_ptr` and `unique_ptr`).

To summarize, the idea is to link some allocated objects to automatic variables, so that when the code ends the automatic vari-

able is deleted, along with the allocated variable. To better understand the point, let us consider the following example:

```
int main(void)
{
unique_ptr<Rectangle> P1(new Rectangle(10, 5));
cout << P1->area() << endl; // This will print 50
return 0; // here the smart pointer destructor will deallocate the rectangle object
}
```

Large programs might need to copy the same pointer into other variables. If the lifetimes of these variables do not overlap, we can enforce single ownership without any additional overhead. In fact, even if syntactically we have multiple variables, semantically we are just **moving ownership** from one to another. C++ has special functions for complying with this semantics and exposes a new constructor (i.e., `&&`) to build such copies. For instance, consider the following piece of code:

```
int main(void)
{
unique_ptr<Rectangle> P1(new Rectangle(10, 5));
cout << P1->area() << endl; // This will print 50
unique_ptr<Rectangle> P2;
P2 = move(P1); // P1 becomes invalid here
cout << P2->area() << endl;
return 0; // only de-allocates the pointer in P2
}
```

In this case, `P1` is not the owner of the `Rectangle` object anymore, it becomes invalid. Moving owernship is a way to ensure that, at any point in time, there is a single variable carrying a pointer to memory. This is opt-in in C++, but it is not in Rust, as we are going to see.

In C++ a move is a shallow copy of an object which invalidates the source object:

```
string s("Original string");
string d(std::move(s));

// uses the following constructor where that == s

string(string&& that) { // string&& is a rvalue reference to a string
data = that.data;
```

```
that.data = nullptr;
}
```

Since `that` is moved-from object, it will not be used anymore and we can just use the original pointer (i.e., `data`) in our new object.

Multiple ownership (i.e., aliasing)

Programs could have pointers pointing to the same regions of memory, in a legitimate fashion. This is called *multiple ownership*, or *aliasing*. As we said, sometimes aliasing is perfectly fine, for istance:

- We want to work temporarily on elements of subvectors of a heap-allocated vector;
- We want several threads working on parts of the same heap-allocated object.

However, in liberal languages such as C++, unintuitive/wrong behavior (i.e., bugs) might happen (e.g., double-free, use-after-free). For double frees, we can use *shared pointers* in C++ that uses reference counting. They use this counter which is increased with every copy and decreased whenever a pointer is destroyed. This is useful for building containers that last over the invocation of a function:

```
auto s1 make_shared<String>("Hello");
auto s2 = s1;
```

Use-after-frees are probably one of the nastier things that we could find in C++ and might appear in several ways. The following C++ valid sequence has a memory bug:

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

The problem with this is that we are creating a pointer to an internal region of the vector and the `push_back()` operation (pushing an element at the end of the vector), might modify or de-allocate the vector memory region (allocating a new one), which means that the print afterwards could break the program, printing random stuff in memory.

Rust and memory safety

Solutions to problem we described before could be the following:

- Try to enforce a single owner and make it commit to manage memory properly;
- When multiple owners are needed (aliasing of pointers), either use a reference counter or strictly control the creation and destruction of such aliases (i.e., *borrowing*).

Single ownership

Rust uses RAII basically everywhere. For instance, when creating a string as an automatic variable on the stack internally this allocates a pointer to the heap where the string is actually allocated:

```
fn main() {
    let s = String::from("hello"); // s is valid from this point forward
    // do stuff with s
} // here memory is automatically de-allocated
```

In this way, memory leaks can't happen anymore.

We can produce explicitly an object on the heap by creating a "boxed value", which is effectively a smart pointer (think of it as `unique_ptr` in C++):

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

Note that, when we exit the scope, `b`'s data is deallocated from the heap.

Move semantics

In Rust, *move semantics* is the default. Variables are in charge of freeing their own resources. Remember that, in Rust, resources can only have one owner. In the following example, `s1` will not be usable after `s2` is created, it will produce a compile time error:

```
let s1 = String:from("Hello");
let s2 = s1;
// ...
```

```
// s1 can't be used anymore
// ...
```

Function invocation moves ownership to the called function:

```
// caller's code
fn func() {
    let s = String::from("Hello");
    // ...
    consume(s); // s looses ownership
    // ...
    println!("{}", s); // compiler error
}

// callee's code
fn consume(r: String) { // r acquire ownership
    // implicit drop(r);
}
```

Multiple ownership

In Rust, the equivalent of *shared pointers* is the `Arc` type (which stands for *atomic reference counted variable*):

```
let s1 = Arc::new("Hello");
let s2 = s1.clone(&s1);
```

This uses an atomic reference counter which can be used to share safely the pointer across threads. There exist also a simple `Rc<T>` trait (not atomic). However, this is a very resource intensive way of dealing with multiple ownership (remember that a reference counter is always kept when using `Arc`). This is solved by borrowing.

Borrowing

Contrary to C++, Rust has a way to deal at compile time with use-after-free bugs. The mechanism is based on references. *References* (or *borrows*) are pointers that do not own the value. Creating a reference is called borrowing: given a variable `v` we create a reference with `&v`. Aliasing through references is controlled: Rust ensures that if there is a live borrowing we can't modify the original value.

Let's see the original (buggy) program in C++ now rewritten in Rust:

```
let mut v = vec![10, 11];
let vptr : &'a mut i32 = &mut v[1];
v.push(12);
println!("{}", *vptr); // error
```

In particular, the mutable borrow `vptr` freezes the original object `v` until it is alive. We use the symbol `a'` to indicate all the lines in which the variable is still alive (i.e., its lifetime). At line 3 in the code above, `v` is used while an outstanding live borrow exists so the compiler produces an error. Actually, the compiler sees that there is an overlap between a modification of an object (i.e., `v.push(12)`) and an outstanding borrow (i.e., `println!("{}", *vptr);`). The other important thing to notice is that this is a so-called *zero cost abstraction*.

In some sense, borrowing a value is like creating a lock on an object: whenever we have a pending lock in a reference to an object, we can't modify the original object itself. Creating non-mutable borrows is like creating read locks: we can have multiple non-mutable borrows outstanding, but we can't modify the object anyhow.

16 Lecture 24 - Seminar on Secure Boot I

Secure boot basics

UEFI Secure Boot (SB) is a verification mechanism for ensuring that code launched by a computer's UEFI firmware is trusted. It is designed to protect a system against malicious code being loaded and executed early in the boot process, before the operating system has been loaded. SB works using cryptographic checksums and signatures. Each program that is loaded by the firmware includes a signature and a checksum, and before allowing execution the firmware will verify that the program is trusted by validating the checksum and the signature. When SB is enabled on a system, any attempt to execute an untrusted program will not be allowed. This stops unexpected / unauthorised code from running in the UEFI environment.

17 Lecture 25 - I/O Devices and Drivers II

Deferring work

The underlying idea behind work deferring is to move non-critical management of interrupts to a later time (i.e., doing a minimal work immediately and let some work to be performed after some time). This might also benefit from aggregation, since we collected some work to be done later [80]. Originally, the Linux interrupt management was divided into two levels:

- *Top-half*: executes minimal amount of work which is mandatory to later finalize the whole interrupt management. Such a scheme works in a **non-interruptible** fashion and schedules some deferred work (or deferred functions);
- *Bottom-half*: finalizes the work using deferred functions from a queue and executes them. These are invoked in particular **reconciliation points** in time.

[80] In some sense, we would have many small operations merged into a single one.

In the Linux kernel there are some abstractions, used to perform the work deferring. *SoftIRQs* are basically tasks to be executed at a later time (i.e., at the reconciliation points) and can be categorized in several types:

- `HI_SOFTIRQ`;
- `TIMER_SOFTIRQ`;
- `TASKLET_SOFTIRQ`;
- etc.

Unless using a very specific type of SoftIRQs, these actions won't be ever interrupted on a CPU when executing, but they could run at the same time on a different CPU [81]. There is another (simpler to use) abstraction called *Tasklets*, which are treated as a specific type of SoftIRQs. By scheduling multiple instances of the same tasklet, we are sure that these instances can't run in parallel, but they are serialized instead.

[81] Thus, some mutual exclusion primitives must be used to avoid concurrency problems. Note that we want to avoid locks as much as possible.

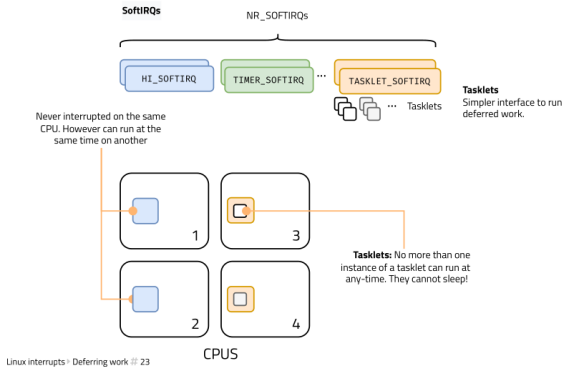


Figure 97: Deferring work

SoftIRQs

Suppose that an handler is being executed. Interrupts of the same IRQs are disabled, but other interrupts can arrive. Whenever another interrupt arrives, if it has some work to do it marks the SoftIRQ as **pending** (i.e., using the `raise_softirq()`). This deferred work will be executed at the next reconciliation point. Note that the handler being executed originally could itself register some deferred work to be done with the `raise_softirq()`. When exiting from the interrupt context (on `irq_exit`), the `do_softirq()` function signals a reconciliation point: on return from hardIRQs or within `ksoftirq/<cpu>` kernel threads, some of the deferred work is executed. Note that all interrupts are enabled when executing softIRQs.

SoftIRQs might be raised at high rates (such as during heavy network traffic) and they might reactivate themselves either. To avoid starving the user space, the kernel's strategy is not to immediately process re-activated SoftIRQs. Instead, if the number of SoftIRQs grows excessively, the kernel wakes up a family of kernel threads to handle the load (i.e., `ksoftirqd` threads). The `ksoftirqd` kernel thread repeatedly checks for pending SoftIRQs and handles them by always ensuring to re-schedule if a higher priority thread comes around.

SoftIRQs are **statically allocated** at compile time and they are an array of `NR_SOFTIRQS` function pointers:

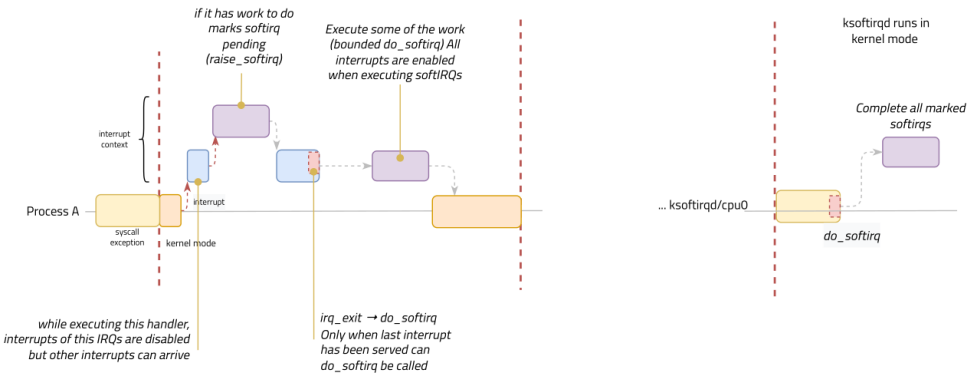


Figure 98: SoftIRQs functioning in the Linux kernel

```
void softirq_handler(struct softirq_action *)
```

Linux ensures that, when a SoftIRQ is running on a CPU, it can't be preempted on that CPU. Another CPU, however, could run the same handler at the same time. This makes them difficult to be programmed correctly.

The kernel defines several SoftIRQs. In general, we want to use only one of this type called `TASKLET_SOFTIRQ` and `HI_SOFTIRQ`:

Name	Prio	Description
`HI_SOFTIRQ`	0	High-priority tasklets
`TIMER_SOFTIRQ`	1	Timers
`NET_TX_SOFTIRQ`	2	Send network packets
`NET_RX_SOFTIRQ`	3	Receive network packets
`BLOCK_SOFTIRQ`	4	Block devices
`TASKLET_SOFTIRQ`	5	Normal priority tasklets
`SCHED_SOFTIRQ`	6	Scheduler
`HRTIMER_SOFTIRQ`	7	High-resolution timers
`RCU_SOFTIRQ`	8	RCU locking

Tasklets

Tasklets are one of the many SoftIRQs available within the Linux kernel. In particular, they have a simpler interface and there can't be more than one instance of the same tasklet running over all the CPUs. This implies no problems with *reentrancy* [82]. A tasklet is basically a function plus some data and it is represented by the kernel with a list of structures as:

[82] A subroutine is called reentrant if multiple invocations can safely run concurrently on multiple processors.

```
struct tasklet_struct {
struct tasklet_struct *next;
unsigned long state; /* 0, scheduled or running */
// ...
void (*func)(unsigned long);
unsigned long data;
};
// ...
DECLARE_TASKLET(my_tasklet, my_tasklet_handler, my_data);
// ...
tasklet_schedule(&my_tasklet);
```

In particular, during the reconciliation points, the kernel goes through the list and looks for tasklets that are scheduled but not running on another CPU: it sets them to the running state and runs the handler (i.e., `func`). Remember that `tasklet_handler` can't sleep and must:

- Take care of **locking data** if they are accessing data shared with other processors;
- **Disable interrupts** if they share data with an interrupt handler;
- **Avoid to block** for any reason.

As shown in the example, we could declare tasklets statically or dynamically in our interrupt handlers:

```
// creates a my_tasklet variable
DECLARE_TASKLET(my_tasklet, my_tasklet_handler, my_data);
```

To schedule a tasklet for execution, we invoke `tasklet_schedule(&my_tasklet)` and it will run once. If the task is re-scheduled before running the first time, it will only run once.

Work queues

It may happen that a deferred action must block. For instance, we could have work allocating a lot of memory, obtaining a semaphore or performing a blocking I/O operation. Thus, we want to use so-called *work queues* to do that. A work queue is a schedulable entity that runs in process context to execute our bottom halves. In particular, this is a general mechanism to submit work to worker

kernel threads called `events/n`. To create work for the `events/n` thread we could use:

```
DECLARE_WORK(work, void (*func)(void *), void *data);
```

To enqueue the work we could use:

```
schedule_work(&work);
```

In particular, the worker thread enters **an infinite loop** and goes to sleep. When the work is queued, the thread is awakened and processes the work. When there is no work left to process, it goes back to sleep.

Timers

There are two main types of timers:

- *System timer*: programmable piece of hardware that issues an interrupt on a fixed frequency. The interrupt handler for this timer (called *timer interrupt*) updates the system time and performs periodic work such as updating the timeslice consumption of a process. Without the system timer, the entire time sharing mechanism in Linux couldn't work (and Linux couldn't work as well);
- *Dynamic timers*: they are used to schedule events that run once after a specified time slice has elapsed [83]. For instance, we could set that, after a certain period of inactivity, a device is turned off.

[83] Note that dynamic timers are multiple ones, while the system timer is just one global timer with fixed tick rate (e.g., 50 to 1000 Hz).

The system timer invokes an interrupt handler called `tick_period()` that updates the `jiffies` global variable, taking into account the time elapsed from the startup of the system. Note that the system timer **is not** the real-time clock. The real-time clock (RTC), instead, provides a non-volatile device for storing the system time. The RTC continues to keep track of time even when the system is off, by using a small battery which is typically included on the system board. On boot, the kernel reads the RTC and uses it to initialize the wall time (stored in the `xtime` variable). On x86, the primary system timer is the programmable interrupt timer (PIT). The PIT exists on all PCs and has been driving interrupts since the days of DOS. The kernel programs the PIT on boot to drive the system timer interrupt (interrupt zero) at the Hz frequency. However, it is a simple device with limited functionality, but it gets the

job done. Other x86 time sources include the local APIC timer and the processor's time stamp counter (TSC).

Consider this definition of a dynamic timer:

```
struct timer_list my_timer;
init_timer(&my_timer);
// ...
my_timer.expires = jiffies + delay; /* timer expires in delay ticks */
my_timer.data = 0; /* zero is passed to the timer handler */
my_timer.function = my_function; /* function to run when timer expires */
add_timer(&my_timer); /* start the timer */
```

The kernel code often needs to delay the execution of some function until a later time (e.g., bottom-halves). Timers are constantly created and destroyed and there is no limit in the number of timers. The kernel checks for expired timers at the end of the system timer; if any has expired, the kernel raises a `TIMER_SOFTIRQ` interrupt.

Linux device management

Linux integrates devices into the file system as **special files**. In particular, each device is assigned a path name (usually in `/dev`). For example, the printer might be `/dev/lp`. Devices are divided into two categories:

- A *character device* is characterized by a character *stream*. Writing or reading from the file has direct impact on the device itself. No buffering is performed in this case;
- A *block device* is seen as a sequence of numbered blocks. Each block can be individually addressed and accessed through a cache (with random access).

Each device has a special *driver* that handles it and has what is called a **major device number** that servers to identify it. If a driver supports multiple devices, say, two disks of the same type, each disk has a **minor device number** that identifies it.

`devfs`, `sysfs` and `udev`

Linux used to expose entries under `/dev` even for devices not currently attached [84]to the system (it used to contain thousands of devices). There have been an evolution of such a scenario, going

[84]Linux used to contain all devices that at some point in time would be attached and used in the system.

under the name of *devfs*, according to which we had only those devices currently plugged into the system.

There were still some problems. For instance, it forced a few constraints (especially on the name of some devices) and implied to have a big database of names in kernel memory. In particular, the name given to a device could change depending on how we connected the device to the system [85]. Each name corresponded to a major and minor number, used by the kernel to determine which hardware device to talk to. Names assigned to devices, as we just said, might change if the user changed the topology (e.g., plug USB devices in a different way). Another new system has been introduced to solve some problems and goes under the name of `udev`. This system substituted `devfs`'s `/dev` directory and gives complete customizability on the naming policy to the user. This new file system structure is based on `sysfs` information, which is an orthogonal and complementary architecture to keep track how devices are connected to the system itself. The `sysfs` part of the file-system (under `/sys`) shows to the file-system how the original devices were connected to the system. This introduces two views [86]: one flattened one (`/dev`) and a hierarchical one (`/sys`).

[85] For instance, changing the port at which the device was connected implied to change the device name as well.

[86] A metaphor is that `/sys` provides access to the packaging, while `/dev` provides access to the content of the box.

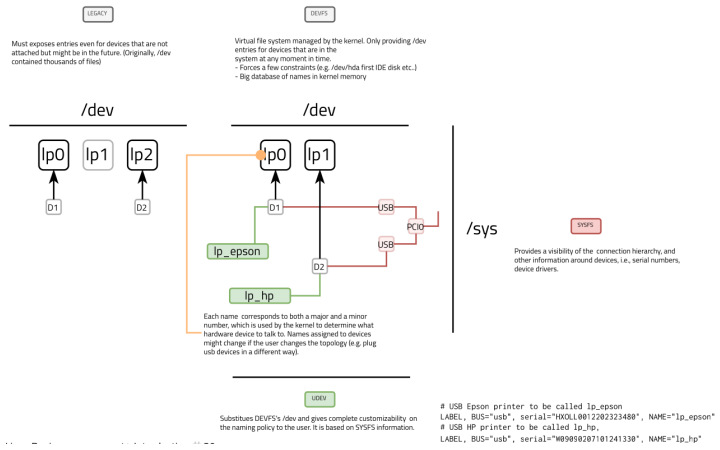


Figure 99: `devfs`, `sysfs` and `udev` functioning

Note that `sysfs` exposes how the kernel models the current state and structure of the system (taking into account devices currently added to the system), in terms of:

- How they are in terms of power management;
- What bus they are attached to;
- What drivers they have, along with the structure of buses, devices and drivers in the system.

The kernel provides a representation of its model in userspace through the `sysfs` virtual file system (mounted under `/sys`), while in `/dev` it allows programs to access devices themselves.

Creating a new character device

Suppose we want to create a new character device. This means that we want to follow a specific pattern within the Linux kernel, including the creation of such a device under `/dev`. Our device must interact with the Linux system to create files. There is a strict connection between the file itself and the operations to be registered and that can be done by the device. The operations are defined using the `file_operations` structure, which is an array of function pointers that define how the kernel will handle various operations (e.g., `read()`, `write()` and `ioctl()`) on a device. This structure allows device drivers to register functions that will be called for each of these operations. By standardizing the interface for device drivers, the kernel is able to support a wide variety of devices without requiring any special code for each one.

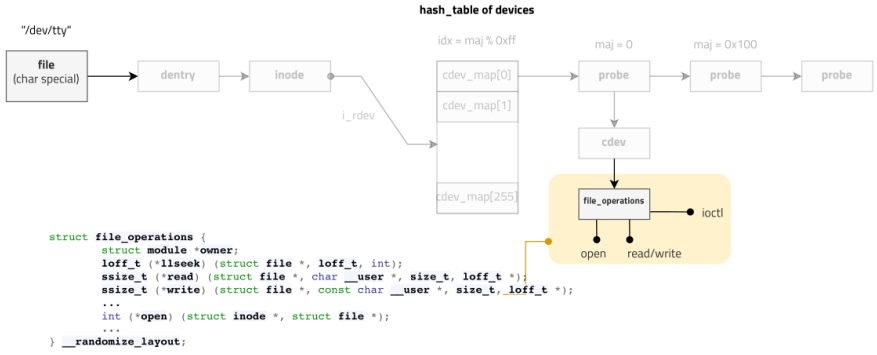


Figure 100: Writing a character device driver

The difference between character and block devices

When creating block devices, instead, we follow a different pattern: we don't interact directly with the device. However, when we want to register a block device we instantiate and specify a structure called `block_device_operations` (along with a `gendisk` structure), defining operations as functions that work just like their char driver equivalents. They are called whenever the device is opened and closed. A block driver might respond to an `open()` call by spinning up the device, locking the door (for removable media), etc. It should also count how many users are using it to safely release the device (i.e., spinning down the device). Block devices can provide an `ioctl()` method to perform device control functions (e.g., return device-specific data such as geometry).

The virtual file-system interacts with block device drivers to read and write data thanks to the `gendisk` data structure: this structure creates the actual block device itself and defines a way to register requests to the device for reading or writing blocks.

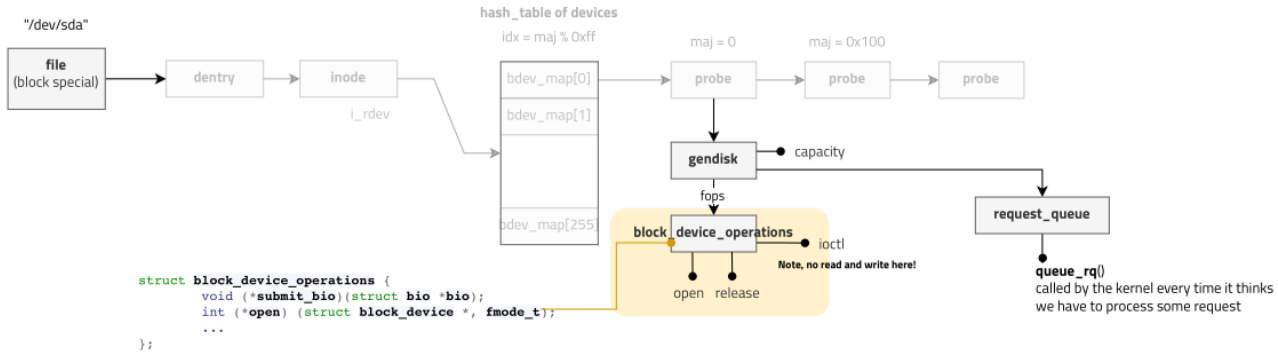


Figure 101: Writing block device drivers

Managing block devices

Suppose we have a process that access a disk with `read()` and `write()` operations involving a certain number of bytes. The VFS is a module that handles two different aspects:

- It verifies that data is already in memory (i.e., 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. Finally, it creates a request to the block layer through a structure called `bio` (i.e., block I/O).

Then, the block I/O layer tries to merge the list of `bio` structures into the least amount of actual requests to the driver.

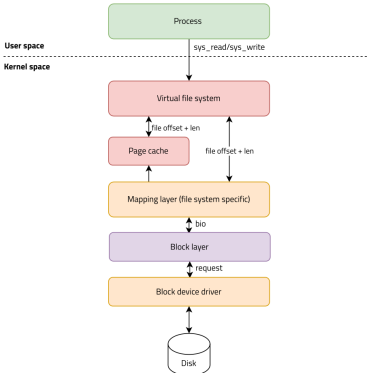


Figure 102: Block device interactions

More specifically, if the page cache does not contain the data, the mapping layer identifies the `segments` of a page that correspond to contiguous sectors on disk. Internally, the mapping layer works with multiple sectors called *blocks* (for simplicity assume that 1 sector is 1 block), then it 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.

Once created, one or more `bios` are sent to the block layer which will create the actual `request` to be sent to the device driver.

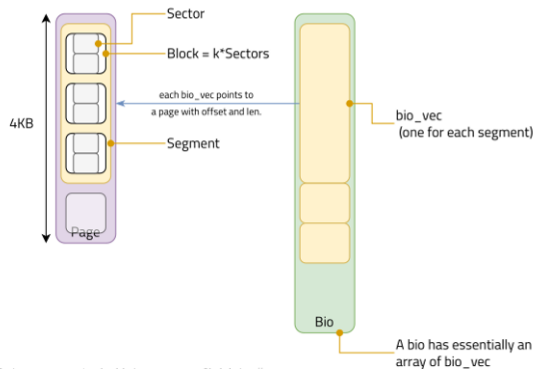


Figure 103: Block request's management

Before sending it, the block layer tries to merge several `bio` requests into single requests whenever possible. 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. 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.

I/O schedulers

I/O schedulers can have many purposes depending on the goal. Common purposes include the following:

- To minimize time wasted by hard disk seeks (still relevant in some cases);
- To prioritize I/O requests coming from certain processes;
- To give a share of the disk bandwidth to each running process;
- To guarantee that certain requests will be issued before a particular deadline.

Since 2013, Linux supports fastest storage devices on large systems, which have multiple hardware queues. This allows to have

multiple requests flying concurrently and to have out-of-order completions. The kernel must support efficient request tagging to distinguish which one has finished. Each CPU has its own software queue to avoid contention.

1. NOOP I/O scheduler The goal of the NOOP I/O scheduler is to pursue efficiency by doing as little as possible. Global I/O requests are ordered in a FIFO fashion. The scheduler just merges requests to adjacent sectors to maximize the throughput. This scheduler is suitable for storage devices not including mechanical parts.

2. Budget Fair Queuing I/O scheduler The goal of the Budget Fair Queuing I/O scheduler is to assign a fair amount of disk bandwidth. In particular, it assigns an I/O budget to each process (i.e., the number of sectors 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. BFQ 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.

3. Kyber I/O scheduler The goal of the Kyber I/O scheduler is to maximize the throughput. This is a simple scheduler that allows for request merging and some simple policies. It is intended for fast multi-queue devices (flash) and lacks much of the complexity found in BFQ.

4. MQ-deadline I/O scheduler The goal of MQ-deadline I/O scheduler is to try to do some merges but then prioritize long starving reads. In particular, it assigns an expiration time to any incoming I/O request. The MQ-deadline I/O scheduler has four queues, two separate FIFO queues READ and WRITE requests and two sector-wise sorted queues for READ and WRITE for merging requests.