Advanced OS

github.com/martinopiaggi/polimi-notes

2023-2024

Contents

1 OS goals and definitions

The main definition, goals, and techniques of an Operating System (OS) are:

- **Resource Management**:
  - Ensures programs are created and run as if they had individual resource allocations.
  - Manages resources like CPU(s), memory, and disk.
  - Achieved through:
    - **Multiplexing the CPU**: Increases CPU utilization and reduces latency by allowing running another program when one is blocked, using preemption and context switches.
    - **Process Management**: Manages process states, indicating what a process can or can't do and the resources it uses. - Includes process control blocks containing PID, process context, virtual memory mappings, open files, credentials, signal handling info, priority, and accounting statistics.
- **Isolation and Protection**:
  - Enforces isolation for reliability.
  - Regulates access rights to resources (e.g., memory) to prevent conflicts and unauthorized data access.
  - Achieved through:
    - **Memory Management**: Utilizes a Virtual Address Space (VAS) where a program's memory locations are typically isolated but may share protected portions.
    - Employs virtual memory areas, some representing on-disk program data and others built dynamically.
    - Implements paging to extend physical memory's apparent abundance to processes
- **Portability of Applications**:
  - Uses Interface/Implementation abstractions to hide hardware complexity.
  - Ensures applications work on different systems with varying physical resources.
- **Extensibility**:
  - Creates uniform interfaces for lower layers, allowing reuse of upper layers (e.g., device drivers).
  - Hides complexity associated with different peripheral models and variants.

1.1 Architectures of OSs

The design of operating systems can encompass various architectural approaches, each with its unique characteristics:

1. **No OS - Bare Metal Programming**:
   - Direct hardware manipulation without any OS layer.
   - Often used in very simple or highly specialized embedded systems.
2. **Monolithic with Modules**:
   - Single large kernel binary.
   - Device drivers and kernel code reside in the same memory area.
   - Modules enhance flexibility.
   - Examples: Linux, Embedded Linux, AIX, HP-UX, Solaris, *BSD.
3. **Micro-kernel**:
   - Minimal core kernel components.
   - External modules for additional services.
   - Non-essential components as user-space processes.
   - Resilient to system process crashes.
   - Examples: SeL4, GNU Hurd, MINIX, MkLinux, QNX, Redox OS.
4. **Hybrid**:
   - Combines micro-kernel design with additional in-kernel code.
   - Certain services (e.g., network stack, filesystem) run in kernel space.
   - Device drivers typically run in user space.
   - Examples: Windows NT, 2000, XP, Vista, 7, 8, 8.1, 10, macOS.

5. **Library OS**:
    - Provides OS services via libraries compiled with the application.
    - An Unikernel approach for cloud or embedded environments (RTOSes).
    - Examples: FreeRTOS, IncludeOS, MirageOS.

Each architecture offers different benefits and trade-offs, influencing the performance, stability, and complexity of the OS and the applications it supports. The choice of architecture depends on the specific requirements of the system it's designed to run on.

2 Process

First, let's clarify some definitions:

- A **program** is a set of computer instructions stored and not currently being executed.
- A **process** is an instance of a program that is currently being executed and has its own isolated memory address space.
- **Task** refers to a single unit of computation and is often used interchangeably with "thread"/process in Linux. A task is made of:
    - unique program counter called `PID`
    - two stacks (one in user mode and one in kernel mode)
    - a set of processor registers
    - if kernel, an optional address space

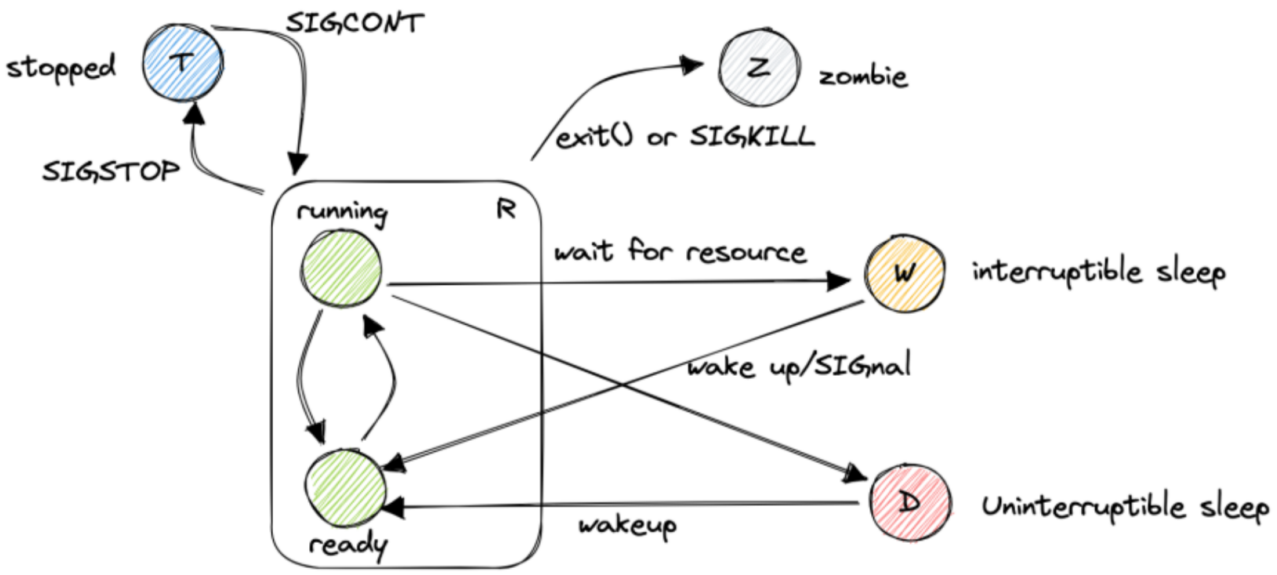
A **Task Control Block (TCB)**, also referred to as a **Process Control Block (PCB)** is the generic term used in os theory to refer to the data structure that stores all the information about a process. A TCB typically contains the following information:

1. **Process Identifier (PID)**: A unique identifier for the process.
2. **Process State**: The current state of the process (e.g., running, waiting, blocked).
3. **Program Counter**: The address of the next instruction to be executed.
4. **CPU Registers**: Includes general-purpose registers, stack pointers, and program counters.
5. **Memory Management Information**: Information about the memory allocated to the process, including base and limit registers or page tables.
6. **Accounting Information**: Includes process execution times, user and kernel mode times, and other performance metrics.
7. **I/O Status Information**: Information about I/O devices allocated to the process, open file descriptors, etc.
8. **Scheduling Information**: Priority of the process, scheduling queue pointers, and other scheduling-related data.
9. **Other Information**: Security credentials, pointers to the process's parent process, signal handling information, etc..

In the context of Linux, the PCB is implemented as the `task_struct`:

`task_struct` attributes
State {R (Running), I (Interruptible), U (Uninterruptible), D (Disk Sleep)}
PID (Process ID)
PPID (Parent Process ID)
mm (Memory Management)
fs (Filesystem Context)
files (Open Files List)
signal (Signal Handlers)
thread_struct (Processor Specific Context)

When a context switch occurs (i.e., the CPU switches from executing one thread to another), the kernel uses the information in the `task_struct` to save the state of the current thread and restore the state of the next thread to run. This is the state machine of a process:



Linux treats all threads as standard processes. Each thread has a unique `task_struct` and appears to the kernel as a normal process, threads just happen to share resources, such as address space with other processes.

System calls and exception handlers are well-defined **interfaces** into the **kernel**. A process can begin executing in kernel space only through one of these interfaces, all access to the kernel is through these interfaces.

`fork()` is a system call in Linux used to create a new process. It duplicates the current process, known as the parent, to create a child process. The new `task_struct` of the child process is a copy of the parent's, with differences in:

- **PID (Process ID)**: Unique identifier for the new process.
- **PPID (Parent Process ID)**: Set to the PID of the parent process.
- **Resources**: Some resources are duplicated or shared under certain conditions.

2.0.1 System Initialization with systemd

Systemd is a system and service manager for Linux, operating as PID 1:

- It initializes the system, manages services, mounts HDDs, and handles clean-up.
- Replaces traditional init systems like SystemV with a more efficient and unified approach.
- Configuration files in declarative language called "unit files"
- Unit files are plain text, INI-style, encoding information about services, sockets, devices, mount points, automount points, etc.

Systemctl is command-line tool for querying and controlling the systemd system and service manager:

- Start, stop, and query the status of services.
- Manage system resources and services effectively.

3 Concurrency

"Linux is probably the most complex multi-process program among all the ones that exist now."

Concurrency is when a program consists of activities that can overlap in execution. A program can be characterized by two properties:

- **Safety**: nothing bad happens.
- **Liveness**: makes progress and is not just stuck.

Regarding concurrency we can highlights these aspects:

- **Deadlock** between tasks: due to mutual exclusion, hold-and-wait, no preemption, and circular wait conditions.
- **Priority inversion** is a scheduling scenario where a high priority task is delayed by a lower priority task due to locking. The bad concurrency model inverts the priority model

Kernel concurrency is a critical aspect of Linux kernel development, distinct from user space concurrency. It involves managing and synchronizing multiple threads within the **kernel space**. This includes:

- **Interrupts**: Handling asynchronous events that can occur at almost any time.
- **Kernel Preemption**: Allowing for the interruption of kernel tasks to enhance system responsiveness and multitasking capabilities.
- **Multiple Processors**: Ensuring the kernel's ability to run on multi-processor systems, which introduces complexities in resource sharing and synchronization.

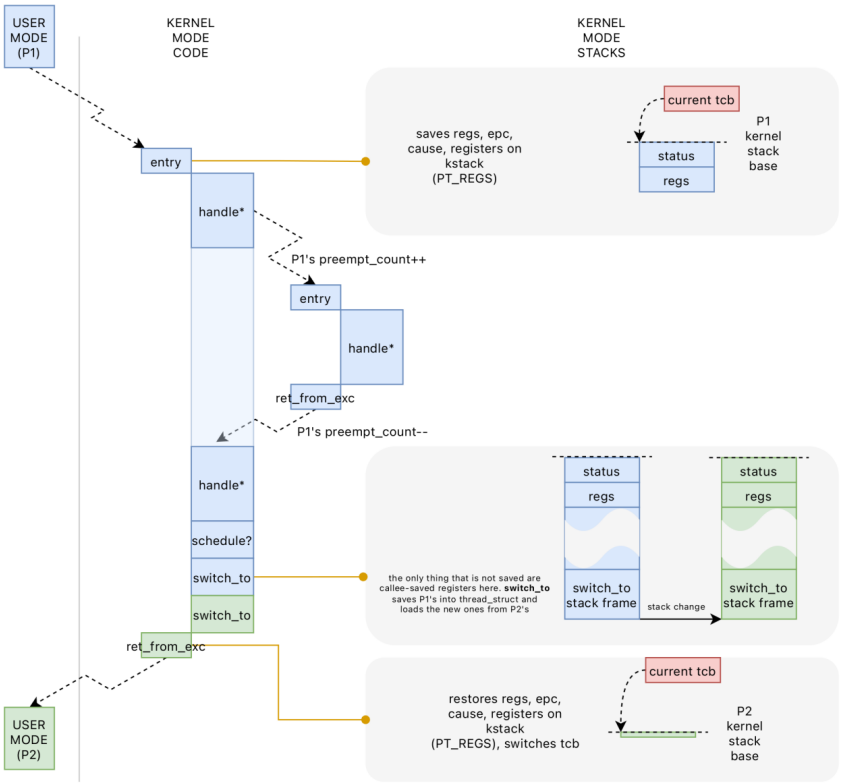
3.1 Preemption

In a preemptive kernel, the kernel allows a process to be preempted while it is running in kernel mode. This means that the kernel can interrupt a process to give CPU time to another process.

This enables higher responsiveness as the system can switch tasks even when a process is executing in kernel mode. Widely used in real-time operating systems where response time is critical.

From Linux kernel version 2.6 onward, the kernel became optionally preemptive. Preemption points **in the kernel** include:

1. When an interrupt handler exits, before returning to kernel-space;
2. When kernel code becomes preemptible again (`preempt_count=0`)
3. If a task in the kernel implicitly/explicitly calls `schedule()`
4. The kernel sets the `TIF_NEED_RESCHED` flag in the current thread's descriptor to indicate that the scheduler needs to run.



How `preempt_count`variable works actually? It is used to ensure a safe context switch that keeps track of the preemptions:

- `preempt_count=0` when the process enters kernel mode
- increase by 1 on lock acquisition (critical section)
- increase by 1 on interrupt

As long as `preempt_count > 0` the kernel cannot switch.

3.2 Synchronization

Developers with a thorough grasp of kernel concurrency can exploit advanced features like:

- **Fine-Grained Locking**: This involves locking at a more granular level, improving system performance and reducing contention.
- **Specialized Synchronization Primitives**: The kernel provides various primitives tailored for specific synchronization
- **Lockless Algorithms**: These allow for efficient data handling without traditional locking mechanisms.

3.2.1 Locking

On SMP (Symmetric Multi-Processing) machines the spinlock is the basic ingredient. **Spinning locks** continuously poll the lock until it becomes available while a sleeping lock (semaphores in Linux) (more overhead) waits until it is notified that the lock is available. The key points of spin lock:

- Useful when **lock for a short period of time**. It's wise to hold the spin locks for less than the duration of 2 context switches, or just try to hold the spin locks for as little time as possible.

- Used in interrupt handlers, whereas semaphores cannot be used because they sleep. (Processes in kernel can't sleep!)

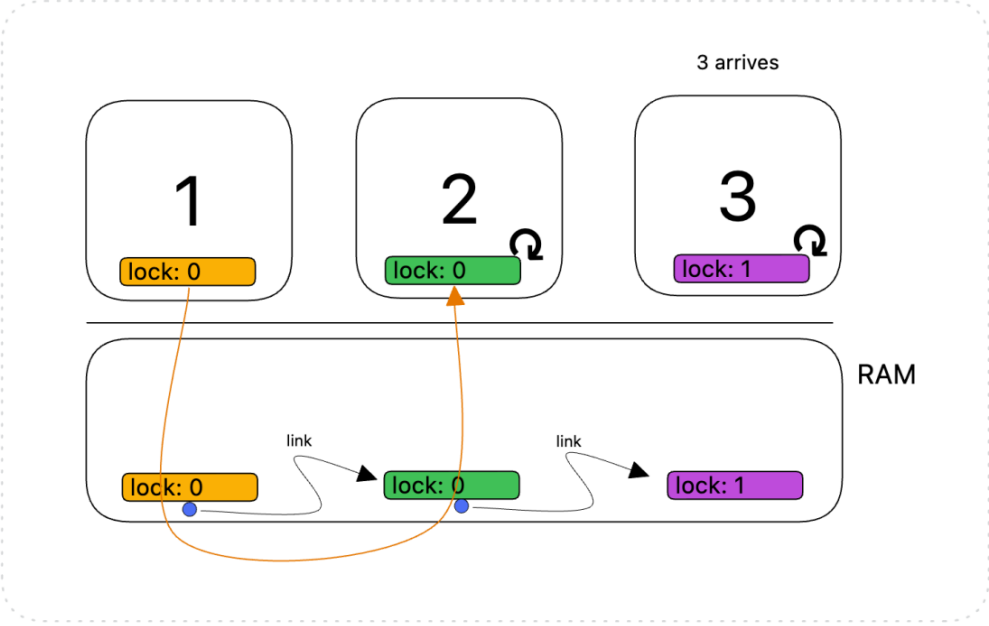
Variants of spinning locks are:

- **Readwrite locks**: distinguish between readers and writers, where multiple readers can access an object simultaneously, whereas only one writer is allowed at a time.
- **Seqlocks**: to prevent starvation of writers, a counter starting from 0 is used to track the number of writers holding the lock. Each writer increments the counter both at locking/unlocking phase. The counter permits to determine if any writes are currently in progress: if the counter is even, it means that no writes are taking place. Conversely, if the counter is odd, it indicates that a write is currently holding the lock. Similarly readers check the counter when trying to lock: if the counter is odd, it means busy wait. If even the reader does the work but before releasing, it checks if the counter changed (in case it does again the work). `jiffies` is the variable that stores a Linux machine's uptime, is frequently read but written rarely by the timer interrupt handler: a seqlock is thus used for machines that do not have atomic 64 bit read.
- **Lockdep**: it enables to report deadlocks before they actually occur in the kernel, which is equally impacted by the problem of deadlocks. When `CONFIG_PROVE_LOCKING`, **lockdep** detects violations of locking rules keeping track of locking sequences through a graph.

3.2.1.1 Multi-CPU concurrency: MCS lock

MCS (Mellor-Crummey and Scott) Locks are a type of synchronization mechanism particularly noted for being **cache-aware spinlocks**: MCS locks solve the cache ping-pong problem by using a **queue system**.

1. **Queue-Based Locking Mechanism**: MCS locks maintain a queue of waiting threads. Each thread in the queue spins on a **local** variable in the **local cache**, reducing the need for frequent memory access, which is common in traditional spinlocks.
2. **Localized Spinning**: In an MCS lock, a thread waits for a signal from its predecessor in the queue. This means that **it only needs to monitor a local variable**, which is likely to be in its cache line.



3.2.2 Lock-free

This part delves into the realm of lock-free algorithms, which are essential for high-performance concurrent systems.

3.2.2.1 Primitives and basic problems

Let's break down each part for a clearer understanding:

1. **Per-CPU Variables**: In a multi-core or multi-processor environment, the Linux kernel maintains data specific to each CPU using per-CPU variables: thereby eliminating the need for synchronization mechanisms like locks.
2. **The Atomic API**: Atomic operations are fundamental in concurrent programming, especially in kernel development. An `atomic` operation is executed entirely as a single, indivisible step, which is crucial in multi-threaded contexts. The atomic API in the Linux kernel provides a set of functions and macros that perform atomic operations, such as adding to, subtracting from, or setting a value. These operations are used to manipulate shared data safely between different threads or interrupt handlers without causing race conditions: the operation will be atomic, meaning that it will be completed in its entirety, or not at all. Atomic operations that can be used to safely increment a shared counter, such as `atomic_inc()`, `atomic_add()`, and `atomic_add_return()`. These operations are implemented using special CPU instructions such as x86's "lock" prefix or ARM's "LDREX" and "STREX" instructions.
3. **CAS (Compare-and-Swap) Primitive**: Compare-and-Swap is a powerful synchronization primitive used in multi-threaded programming. It is a single atomic operation that compares the contents of a memory location to a given value and, only if they are the same, modifies the contents of that memory location to a new given value. This atomicity is crucial to ensure that no other thread has altered the data between the steps of reading and writing. CAS is widely used in the implementation of lock-free data structures and algorithms.
    - **The ABA Problem**: The ABA problem is a classical issue encountered in concurrent programming when using CAS operations. It occurs when a memory location is read (A), changed to another value (B), and then changed back to the original value (A) before a CAS operation checks it. The CAS operation sees that the value at the memory location hasn't changed (still A) and proceeds, unaware that it was modified in between. This can lead to incorrect behaviour in some algorithms, as the change-and-reversion can have side effects or meanings that the CAS operation fails to recognize.

3.2.2.2 Readers/writers

The readers/writers problem is a common challenge in managing access to shared data by multiple threads where some threads read data (readers) and others modify it (writers).
One synchronization technique used in the Linux kernel is the **Read Copy Update** (RCU) mechanism.

The core idea of RCU is to allow readers to access data without any locks. In this way readers don't need to acquire locks, they can access shared data with minimal overhead, which is particularly advantageous in high-concurrency environments.

When a writer needs to update a data structure, instead of modifying the structure in-place and potentially disrupting readers, RCU creates a copy of the structure and applies the changes to this copy. Once the updated copy is ready, the system switches pointers so that subsequent readers see the updated structure. The old data structure is not immediately freed; instead, it is marked for later reclamation once all pre-existing readers that might be accessing it have finished.

In the Linux kernel, RCU is widely used for various purposes, such as managing network routes, file system mounts, and other data structures where the pattern of frequent reads and infrequent updates is common.

3.2.2.3 Memory models

Memory models define how memory operations in one thread are seen by others in a multiprocessor environment. Different models have varying levels of order enforcement and visibility, influencing how multi-threaded programs behave and are synchronized.

- **Sequential Model**: Theoretically the strongest but impractical, ensuring program and memory order are the same.
    - Defined by Leslie Lamport's in the 1979 paper: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs"
    - Define $<_p$ the program order of the instructions in a single thread and $<_m$ as the order in which these are visibile in the shared memory (also referred to as the happens-before relation).

  - A multiprocessor is called sequentially consistent iff for all pairs of instructions $(I_{p,i}, I_{p,j})$ you have that $I_{p,i} <_p I_{p,j} \rightarrow I_{p,i} <_m I_{p,j}$
  - In practice the operations of each individual processor appear in this sequence in the order specified by its program.
- **Total Store Order (TSO)**: Used in x86 intel processors, involves FIFO store buffers allowing delayed writes.
  - Processors use a local write queue, or **store buffer**, to mitigate the delays caused by memory latency when they issue a store command. Once a write operation is completed and the data reaches the shared memory, it becomes visible to all processors.
  - Despite the sequence of operations,**loads may appear to be reordered in relation to stores** due to the buffering of write operations. This means that reads can sometimes retrieve updated values before the writes are fully propagated through the system.
  - To ensure sequentially consistent behavior, especially at crucial points within a program, processors can use specific instructions known as "memory barriers" like the `mfence` instruction. The `mfence` instruction acts as a full memory barrier, ensuring that each thread completes its pending write operations to memory before initiating any read operations.
- **Partial Store Order (PSO)**: Used in ARM, treats memory as if each processor has its own copy. **Writes can be reordered**, lacking a mechanism for simultaneous write visibility across processors.
  - The PSO model can allow data races due to the absence of a forced happens-before relation. Synchronization instructions like *release* and *acquire* are necessary to commit writes to shared memory and avoid data races.

PSO is the linux memory model: the least common denominator all CPU families that run the Linux kernel. The Linux Kernel Memory Model (LKMM) it's mainly based on the primitives `smp_store_release (W[release])` and `smp_load_acquire(R[acquire])`, which are used to create happens-before arcs between different threads.

3.2.2.4 Data races Compilers can reorder instructions, affecting perceived memory models. High-level languages provide mechanisms to enforce certain memory orderings:

- **Fences/Barriers**: These are special instructions that prevent certain types of reordering of memory operations, ensuring that operations are completed as perceived in the program's order.

Fences help in avoiding incorrect read operations that might arise due to data races in concurrent environments.

3.2.2.4.1 Great powers, great responsibilities

- **Performance Consideration**: Overuse of fences can lead to performance degradation as they restrict the processor's ability to reorder instructions for efficient execution.
- **Strategic Placement**: It's crucial to strategically place fences where necessary, rather than using them indiscriminately, to balance correctness and performance.

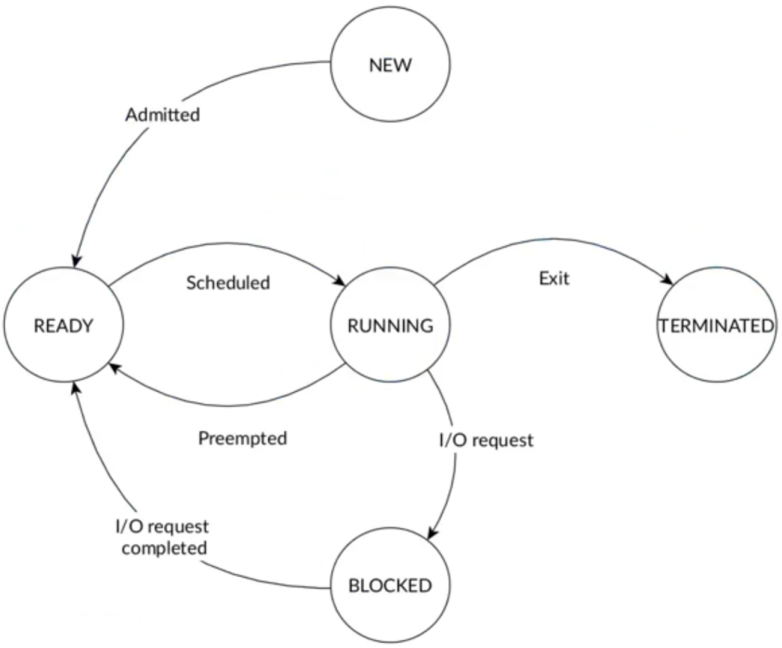
3.2.2.5 LKMM The linux memory model is the least common denominator all CPU families that run the Linux kernel. At its core, it's **essentially, PSO.**

The LKMM employs specific C language instructions, known as primitives, to manage memory ordering **across different threads**. Two such critical primitives are `smp_store_release` (notated as `W[release]`) and `smp_load_acquire` (notated as `R[acquire]`).

These primitives play a pivotal role in establishing **happens-before** "relationship arcs" between concurrent threads.

4 Scheduling

The scheduler is responsible for determining the order in which tasks are executed ("task" can have different meanings and can be used interchangeably with "thread" in these notes). A simplified view of task states is:



Actually the reality in Linux is much more complex and there are a lot of possible things that can happen to the process like errors, special states etc.

When studying the scheduling of processes we have a couple of metrics to consider:

- `a_i`: **Arrival time**, which is when a task becomes ready for scheduling.
- `s_i`: **Start time**, which indicates when a task begins execution.
- `R_i`: **Response time**, which is the time from arrival to **first quantum slice** ends.
- `W_i`: **Wait time**, which is the **total** time spent waiting in the queue by a task.
- `f_i`: **Finishing time**, which denotes when a task completes its execution.
- `C_i`: **Computation time** or burst time, which represents the duration required for the processor to execute the task without any interruptions.
- `Z_i`: **Turnaround time**, which is the overall time taken from when a task becomes ready until it completes its execution. It is given by the equation `Z_i = f_i - a_i`. Note that `Z_i` is not necessarily equal to `W_i + C_i` because **interruptions** could occur.

Based on the nature of the operations performed by a task, we categorize processes as either CPU-bound or I/O bound:

- CPU-bound processes primarily spend their time executing computations. In this case, `Z_i` is approximately equal to `W_i + C_i`.
- I/O bound processes spend most of their time waiting for I/O operations. Here, `Z_i` is significantly higher than `W_i + C_i`.

Scheduling in an OS is a critical task involving the decision of which process to run next. The scheduling policy should aim to balance several factors:

- **Fairness**:

  - Ensure the scheduling is fair and that no process is starving.
- **Throughput**:
  - Aim for maximum process completion rate.
- **Efficiency**:
  - Minimize the resources used by the scheduler itself.
  - Optimize CPU usage reducing context switching overhead.
- **Priority**:
  - Reflect the relative importance or urgency of processes.
- **Deadlines**:
  - Meet time constraints for time-sensitive operations like real-time tasks like multimedia playback or similar.

Note that OS scheduling strategies are **balancing conflicting goals** like deadlines and fairness. For this reason we must make a distinction **domain-Specific Scheduling**:

- General-Purpose OSes **GPOS**: Balance throughput, fairness, user response; utilize time-sharing, dynamic priorities.
- Real-Time Operating Systems **RTOS**: Prioritize deadlines, predictability; apply RMS, EDF algorithms.

Additionally, **user** and **kernel mode** processes may have different priorities. But also **I/O-Bound** and **CPU-Bound** processes need a distinction for resource efficiency. **Multicore/Multiprocessor** environments add scheduling complexities and **adaptive Scheduling** adjusts priorities and nakes decisions based on system load and process activity.

4.1 Scheduling algorithms

The algorithm used by the scheduler to determine the order is called the **scheduling policy**. Computing an optimal schedule and resource allocation is an NP-complete problem. To increase the complexity we have to keep in mind that these objectives often conflict with each other:

- maximize processor utilization
- maximize throughput: number of tasks completing per time unit
- minimize waiting time: time spent ready in the wait queue
- ensure fairness
- minimize scheduling overhead
- minimize turnaround time
- and many more: energy, power, temps, ...

Also there is the problem of **starvation**. Schedulers can be categorized into different types based on their characteristics.

- **Preemptive vs Non-preemptive**:
  - Preemptive is the ability to interrupt tasks and allocate the CPU to another task ensuring responsiveness.
  - Non-preemptive schedulers minimizes overhead but can impact responsiveness.
- **Static vs Dynamic**:
  - Static schedulers make decisions based on fixed parameters and are not be realistic in general-purpose systems.
  - Dynamic schedulers make decisions based on runtime parameters.
- **Offline vs Online**:
  - Offline schedulers are executed once before task activation, and the resulting *inflexible* schedule remains unchanged.
  - Online schedulers are executed during task execution at runtime, allowing for the addition of new tasks.
- **Optimal vs Heuristic**:

- Optimal schedulers typically come with higher overhead and complexity.
- Heuristic schedulers are not optimal but are usually more efficient in terms of overhead.

What are the menu offerings?

Name	Target (Goal)	Preemptive
`FIFO`	turnaround	No
`SJF`	waiting time	No
`HRRN`	waiting time	No
`SRTF`	waiting time	Yes
`Round-robin`	response time	Yes
`CFS`	CPU fair share	Yes

4.1.1 First-In-First-Out (FIFO)

Simplest scheduling algorithm possible, also known as First Come First Served (FCFS). FIFO blueprint:

- Tasks are scheduled in the order of arrival
- Non-preemptive
- Very simple
- Not good for responsiveness
- Long tasks may monopolize the processor
- Short tasks are penalized

4.1.2 Shortest Job First (SJF)

Shortest Job First (SJF) scheduler aims to minimize the **waiting time** of processes. Also known as Shortest Job Next (SJN). SJF blueprint:

- It selects the process with the shortest computation time `C_i` and executes it first.
- Non-preemptive
- Starvation for long tasksthe main disadvantage
- How the fuck you know C_i in advance?

This makes SJF less ideal in environments with a high variance in task length or where fairness among tasks is a crucial requirement.

A possible alternative that mitigates starvation is the Highest Response Ratio Next (HRRN) scheduler.

4.1.3 Shortest Remaining Time First (SRTF)

SRTF uses the **remaining** execution time instead of the total C_i to decide which task to run. SRTF blueprint:

- Improve responsiveness for all tasks compared to SJF
- Starvation for long tasks
- We need to know C_i in advance as SJF

4.1.4 Highest Response Ratio Next (HRRN)

HRRN selects the task with the highest **Response Ratio**:

$$RR_i = (W_i + C_i) / C_i$$

HRRN blueprint:

- Non-preemptive
- Prevent starvation
- We need to know C_i in advance

4.1.5 Round Robin (RR)

RR is very popular and very simple and also very adopted in modern OS. Tasks are scheduled for a given time slice q and then preempted to give time to other tasks. RR blueprint:

- Preemptive: when the time quantum expires, the task is moved back to the ready queue.
- Computable maximum waiting time: $(n-1)*q$
- No need to know C_i in advance
- Good to achieve the fairness and responsiveness goals
- No starvation is possible
- Turnaround time worse than SJF

Tasks in a ready queue are added based on FIFO policy. If an executing task gets preempted while a new task has been added in the ready queue, the new task has precedence in the queue over the preempted task. In Linux, the default time quantum for the Round Robin (RR) scheduler is stored in `/proc/sys/kernel/sched_rr_timeslice_ms`, with a default value of 100ms.

The total overhead depends on the number of preemptions, and so it can be reduced by increasing the quantum q. Obviously, the disadvantage of choosing a larger quantum value is the increasing of the average turnaround and waiting time.

4.1.6 CFS (Completely Fair Scheduler)

CFS attempts to balance a process's virtual runtime with a simple rule: CFS picks the process with the smallest virtual runtime (`vruntime`), which represents the time a task should run on the CPU. CFS uses a red-black tree to find the task with the smallest `vruntime` efficiently. If here're no runnable processes, CFS schedules the idle task.

In Linux, the transition from the $O(1)$ scheduler to the CFS marked a significant evolution in process scheduling, emphasizing fairness and dynamic adaptability.

The $O(1)$ scheduler (known for its constant time complexity) offered quick scheduling decisions but struggled with **fair** CPU time distribution, especially for long-running tasks. This was due to its reliance on fixed timeslices, which could lead to task starvation.

CFS dynamically adjusts time slices in proportion to the task's priority. All it's based on the 'nice' value $\nu \in [-20, +19]$

The nice value it's then used in this exponential formula:

$$\lambda_i = k \times b^{-\nu_i}$$

(current values k= 1024, $\mathbf{b} = \mathbf{1}.25$) The exponential formula is then used for compute the weight derived from the task's nice value (ν_i), influencing its share of CPU time:

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

The targeted latency τ reflects the desired period within which all runnable tasks receive CPU time, while the minimum granularity (μ) ensures a lower bound on the timeslice, preventing excessive preemption.

Then for each process p, its time-slice is computed as:

$$\tau_p = f(\nu_0, \ldots, \nu_p, \ldots, \nu_{n-1}, \bar{\tau}, \mu) \sim max(\frac{\lambda_p \bar{\tau}}{\sum \lambda_i}, \mu)$$

4.1.6.1 Cgroups CFS alone is not enough to guarantee optimal CPU usage, especially when there are multiple threads from different user: for example, if user `A` with 2 threads and user `B` with 98 threads, user `A` will only be given 2% of the CPU time, which is not ideal. Each user should be given an equal share of the CPU, which is then divided among their threads. Cgroups is a mechanism for guarantee fairness and optimal cpu usage when there are multiple users: it allocates CPU usage based on groups rather than individual threads.

4.1.6.2 Load balancing in CFS Load balancing in CFS is done using a work stealing approach, where each idle core balances its workload by attempting to steal threads from the busiest core (also known as the designated core).

4.2 Scheduling classes

A scheduling class is an API (set of functions) that include an unique scheduling algorithm/policy-specific code. This allows developers to implement thread schedulers without reimplementing generic code and also helps minimizing the number of bugs. Which are these scheduling class in linux?

- `SCHED_DEADLINE`
- `SCHED_FIFO`
- `SCHED_RR`
- `SCHED_OTHER`
- `SCHED_BATCH`
- `SCHED_IDLE`

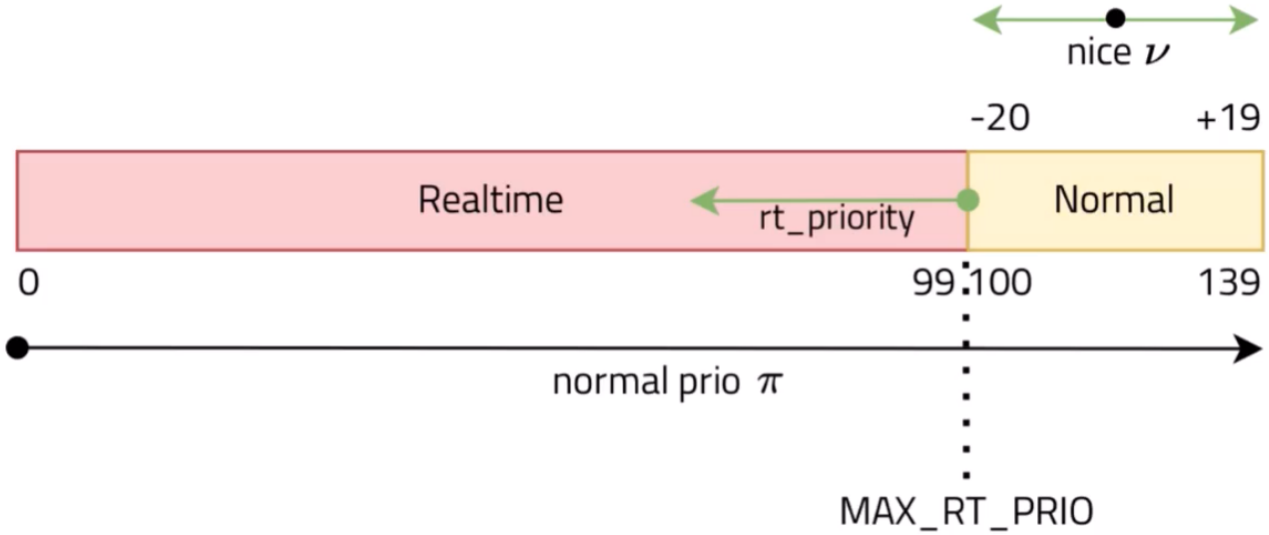
Or in more detailed way:

Scheduling Class	Description	Scheduling Algorithm	Type of Target Processes
SCHED_DEADLINE	Deadline-based	(EDF) Earliest Deadline First	Real-time
SCHED_FIFO	Soft real-time processes, continue to run until higher priority task is ready.	First-In-First-Out	Real-time
SCHED_RR	Share with timeslice	Round-Robin	Real-time
SCHED_OTHER / SCHED_NORMAL	Variable-priority	Completely Fair Scheduler (CFS)	Not real-time
SCHED_BATCH	Low-priority	CFS with idle task prioritization	Not real-time

Real-time processes $\pi \in [0, 99]$; they belong to scheduling class `SCHED_FIFO` or `SCHED_RR` or `SCHED_DEADLINE`.

`SCHED_NORMAL` and `SCHED_BATCH` are implemented through CFS. The difference is that `SCHED_BATCH` has a longer timeslice (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.

Non real-time processes $100 \leq \pi(v) \leq 139$ which depend on a nice value $v \in [-20, +19] : \pi(v) = 120 + v$.



The `nice` value is only applicable to non-real-time processes, specifically those in the `SCHED_OTHER` (also known as `SCHED_NORMAL`) scheduling class. The `nice` value ranges from -20 (highest priority within this class) to +19 (lowest priority within this class). These `nice` values are used by the Completely Fair Scheduler (CFS) to adjust the share of CPU time that processes get, with lower `nice` values giving a process more priority, hence more CPU time.

Regarding process priorities, Linux uses a priority range from 0 to 139, where:

- Priorities 0 to 99 are reserved for real-time priorities (higher value, higher priority), used in scheduling classes like `SCHED_FIFO`, `SCHED_RR`, and `SCHED_DEADLINE`.
- Priorities 100 to 139 are for non-real-time tasks, with the `SCHED_OTHER` (or `SCHED_NORMAL`) and `SCHED_BATCH` classes. Within this range, the effect of the `nice` value is evident.

It might seem counterintuitive, but within the Linux kernel's scheduling system, a lower priority number means a higher priority for getting CPU time. This is particularly true for real-time tasks where a priority of 0 is the highest possible priority. This scheme allows real-time tasks (with priorities 0 to 99) to always preempt non-real-time tasks (with priorities 100 to 139) regardless of their `nice` values.

So, in summary:

- `nice` values are used to adjust priorities within the non-real-time priority range (100 to 139).
- Real-time tasks, which ignore `nice` values, have priorities in the range of 0 to 99, where a lower number means higher priority.

Regarding the rest, Linux chooses a simple fixed-priority list to determine this order (deadline → real-time → fair → idle).

Priority is selected depending on the workload type:

- CPU-bound tasks have low priority (high quantum value)
- I/O-bound tasks have high priority (low quantum value)

How to know if a task is CPU-bound or I/O-bound?

- A **run-time feedback mechanism**: a a new task is always placed in the highest priority queue with the lowest quantum value. If the quantum expires, the task is progressively moved in queues with longer time quantum.
- Or manually set by the user

4.3 Multi-Processor Scheduling

In a multi-processor system, the scheduler must decide not only which task to execute but also on which processor to assign. This can be a challenging decision due to various factors such as the occurrence of task synchronization across parallel executions and the difficulty of achieving high utilization of all processors or CPU cores. Additionally, managing correctly cache memory which can significantly enhance overall performance by enabling faster access to frequently used data, reducing the reliance on slower main memory access.

- **Load balancing**: evenly distributing tasks across different queues to positively impact power consumption, energy efficiency, and system reliability. It's typically performed via **task migration** which can be implemented mainly in 2 ways:
  - **push model**: a dedicated task periodically checks the lengths of the queues and moves tasks if balancing is required.
  - **pull model**: each processor notifying an empty queue condition and picking tasks from other queues.
- **Hierarchical queues**: a hierarchy of schedulers can be implemented to manage task dispatching in a global queue and local ready queues. Improved scalability with maybe more complex.

5 Multi-processing and Inter-Process Communication

The use of multiple processes in a program (multi-process programming) and Inter-Process Communication (IPC) are essential for developing applications that require multiple processes to work together.

- **Forking**: the forking operation allows a process to create a new process that is a copy of itself. this new process runs concurrently with the original process. The virtual address space is copied, and most of the physical pages in memory are marked as "copy-on-write". The new process has the same variables as the original process, **except for the fork's return value**. No direct access to parent-child variables, so IPC is needed.
- **Parent-Child basic synchronization**: A Parent-Child basic synchronization is the use of `wait()` which suspends the parent until one of the children terminates. Also `waitpid()` suspends execution until a **specific** child process terminates or changes state. If the parent terminates before calling `wait()` the os ends up with zombie processes which are "adopted" by the `init` process which performs `wait()` on all its children (remember that it's the ancestor of any process, freeing memory and PID numbers.
- **IPC**
  - **Signals**
  - **Pipes and FIFO**
  - **Messages Queues**
  - **Shared Memory**
  - **Synchronization**

5.1 Inter-Process Communication (IPC)

Linux has two main libraries: POSIX and System V. POSIX is the newer library, while System V is considered legacy.

5.1.1 Signals

Let's see the very first example of inter-process communication, which is the use of **Signals**. Signals are communication methods between processes, and they can be sent by processes or by the OS. They have the following characteristics:

- Unidirectional: one process sends a signal without expecting a reply.
- No data transfer: simply indicate an event or request to stop a process.
- Asynchronous

To send a signal, we use the `kill()` function with the PID of the receiver process and the signal to send as arguments. Due to its name, there's a common misconception that `kill` is only for terminating processes. In reality, it's a general-purpose tool for signal sending:

- `SIGHUP(1)`: Controlling terminal disconnected.
- `SIGINT(2)`: Terminal interrupt.
- `SIGILL(4)`: Attempt to execute illegal instruction.
- `SIGABRT(6)`: Process abort signal.
- `SIGKILL(9)`: Kill the process.
- `SIGSEGV(11)`: Invalid memory reference.
- `SIGSYS(12)`: Invalid system call.
- `SIGPIPE(13)`: Write on a pipe with no one to read it.
- `SIGTERM(15)`: Process terminated.
- `SIGUSR1(16)` and `SIGUSR2(17)`: User-defined signals 1 and 2 respectively.
- `SIGCHLD(18)`, which is used when a child process terminates, stops, or continues.

The return value is `0` on success and `-1` on error. To handle a signal, we use the `sigaction()` function. It allows us to register a handler function for a specific signal. The return value is same as before.

There are additional functions introduced in modern Linux systems that support POSIX real-time:

- `sigqueue()`: Sends a queued signal
- `sigwaitinfo()`: Synchronously waits for a signal
- `sigtimedwait()`: Synchronously waits for a signal for a given time

Signals can be masked using `sigprocmask()` to prevent them from interrupting the execution of code. Masked signals are enqueued and managed later when the process unmasks them. However, certain signals like `SIGKILL` and `SIGSTOP` cannot be masked.

By default, most signals result in the termination of the receiving process. However, custom behavior can be implemented by registering signal handlers using the `sigaction` function. The `sigaction` data structure contains function pointers for defining signal handlers.

5.1.2 Pipes and FIFO

We can use pipes as a mechanism of IPC. The concept of a pipe is similar to an actual pipe, where data can flow from one end to the other. Pipes are commonly used in the producer-consumer pattern, where one producer writes and one consumer reads, creating a unidirectional flow of data. The data is written and read in a First-In-First-Out (FIFO) fashion. To create a pipe:

- Use the `pipe` function to create an array of two integers, `pipefd`, which will be filled with two file descriptors.
- `pipefd[0]` represents the file descriptor of the read end of the pipe.
- `pipefd[1]` represents the file descriptor of the write end of the pipe.

Functions like `read` and `write` permit to directly interact with the file descriptors.

Alternatively, you can transform your file descriptor into a stream using functions like `fwrite` and `fscanf`.

5.1.2.1 Named pipes Named pipes have similar behavior to unnamed pipes but are based on special files created in the filesystem. Data is transferred between processes as if reading/writing to a disk file. To create a FIFO:

- Specify the pathname (path + filename) for creating the FIFO.
- Set appropriate permissions.

Any process can use `open/write/read` functions to access and manipulate data in the FIFO.

5.1.3 Messages Queues

When dealing with multiple writers and multiple readers we need a different mechanism for establishing communication. This is where message queues come into play. Message queues are more complex than pipes:

- multiple producers and consumer
- message queues have state logic. The os keeps track of the queue in a special file in `dev/mqueue/`
- priorities messages and a bunch of attributes like flags etc

To create a message queue, we utilize the `mq_open` function, which has the following parameters:

- `name`: a unique name for the message queue, starting with "/"
- `oflag`: a flag related to blocking behavior or non-blocking behavior
- `mode`: file permissions to give to the file (only for O_CREAT)
- `attr`: attributes

The function returns a message queue descriptor (`mqd_t` data type) or `-1` in case of error.

The attribute struct also enables us to specify the maximum number of messages allowed in the queue (`mq_maxmsg`) and the size of each individual message (`mq_msgsize`).

To send a message with `mq_send()` the following parameters are used:

- `mqdes`: message queue descriptor
- `msg_ptr`: pointer to the message to send
- `msg_len`: length of the message in bytes
- `msg_prio`: non-negative priority value in the range `[0 ; 31]`

On receiving end of a message queue, we use `mq_receive()` function which takes similar arguments as `mq_send()`:

- `mqdes`: message queue descriptor
- `msg_ptr`: output parameter - pointer to a buffer to fill with the received message
- `msg_len`: length of the buffer in bytes
- `msg_prio`: output parameter - priority of the received message.
- **Processes**: Process A (Sender) and Process B (Receiver).

5.1.4 Steps:

1. **Message Queue Creation**:
   - A message queue is created, which can be accessed by both Process A and B.
   - Often done using `msgget` in Unix-like systems.
2. **Define Message Structure**:
   - A structure for messages is defined, including a message type and content.
   - Example Structure in C:
     ```c
     struct message {
         long msg_type;
         char msg_text[100];
     };
     ```
3. **Process A (Sender)**:
   - Prepares a message and sends it to the queue.
   - Uses `msgsnd` to send the message.
   - Example Code:
     ```c
     struct message msg;
     msg.msg_type = 1; // Message type
     ```

```
strcpy(msg.msg_text, "Hello Process B");
msgsnd(queue_id, &msg, sizeof(msg.msg_text), 0);
```

4. **Process B (Receiver)**:
   - Listens for messages on the queue.
   - Uses `msgrcv` to receive messages.
   - Example Code:
     ```
     struct message msg;
     msgrcv(queue_id, &msg, sizeof(msg.msg_text), 1, 0);
     printf("Received Message: %s\n", msg.msg_text);
     ```

5.1.5 Shared memory

Shared memory is a different IPC mechanism where multiple processes access the same piece of memory.

To create or open a shared memory segment, we use the `shm_open()` function. This function takes parameters such as the name of the shared memory (starting with /), opening flags (e.g., `O_RDONLY`, `O_WRONLY`, `O_CREAT`), and file permissions. It returns a file descriptor.

When using shared memory, it is recommended to call `ftruncate` to specify its size. This function takes the file descriptor and the desired size in bytes as parameters.

To map the shared memory into our process's address space, we use `mmap()`. This function takes parameters:

- The starting virtual address (usually set as `NULL`)
- The size of the mapped segment
- Memory protection flags (`PROT_EXEC`, `PROT_READ`, `PROT_WRITE`, `PROT_NONE`)
- The file descriptor obtained from `shm_open()`
- An offset which is used when mapping files rather than creating a new object. Indeed another use case for memory mapping is accessing large files more efficiently by loading them into memory instead of reading from disk. The offset is used to specify the point from which to start reading.
- Visibility flags (`MAP_SHARED` or `MAP_PRIVATE`).

To clean up and release resources associated with shared memory, we can use `munmap()` and `shm_unlink()`. The `munmap()` function deletes mappings for a specified address range, and `shm_unlink()` removes the shared memory object created by `shm_open()`.

5.1.6 Synchronization

An efficient synchronization mechanism is necessary to avoid inefficient and power-consuming CPU busy loops where multiple processes access shared data simultaneously inefficiently. One IPC mechanism to achieve effective synchronization is the use of **semaphores**.

Semaphores act as a bridge between processes using a counter that determines whether a process should wait or proceed. When the semaphore counter is `0`, the process waits. When the counter is greater than `0`, the process proceeds. In the case of binary semaphores, where the counter can only be `0` or `1`, they behave similarly to a mutex.

There are two atomic functions available for synchronization:

- `wait()`: Blocks until the counter is greater than `0`, then decrements the counter and allows the process to proceed.
- `post()`: Increments the counter.

For unnamed semaphores, they can be initialized using the `sem_init()` function and destroyed using the `sem_destroy()` function. **Named semaphores**, which have POSIX object names, require different functions:

- `sem_open()` is used to initialize named semaphores

- `sem_close()` is used to close them
- `sem_unlink()` is used to remove them from memory.

Synchronization functions are:

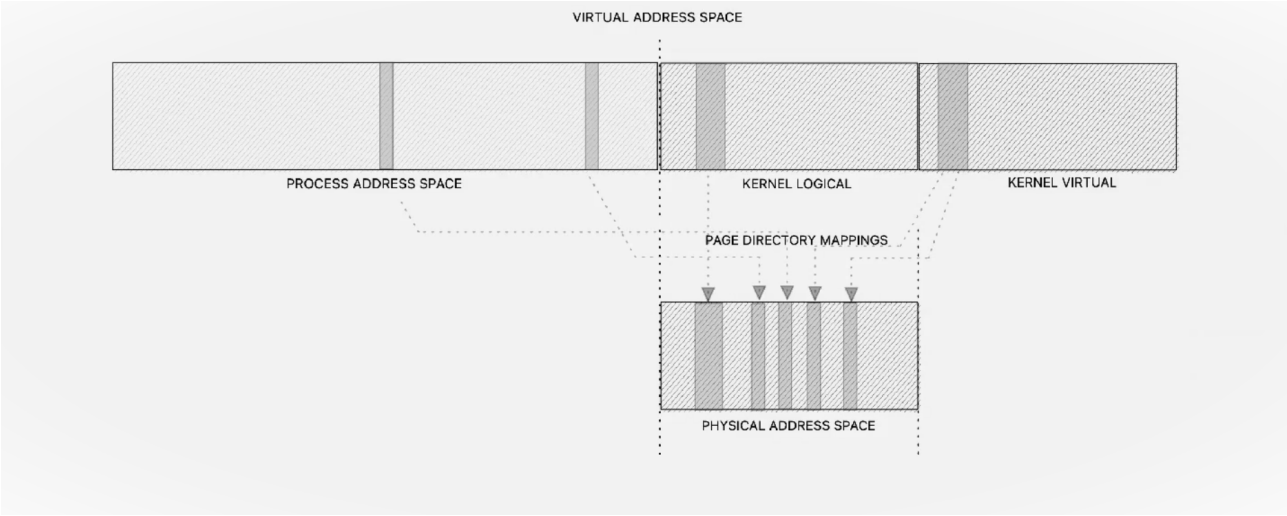
- `sem_wait()` to wait until the semaphore counter is greater than `0`. It has a timeout parameter to specify how long to wait if the counter is `0`.
- Alternatively, `sem_trywait()` can be used, which is a non-blocking version of `sem_wait()`.

All these synchronization functions return `0` for success or `-1` for error.

Mutexes and condition variables are are further known synchronization mechanisms commonly used in multi-threading programming. While we won't cover them in detail here, you will come across them in your studies. For more information, refer to the POSIX documentation, specifically the `manpages-posix-dev`.

6 Virtual memory

Linux, similar to other OSs, uses virtual memory to manage memory. Virtual memory creates the perception of a larger memory space, even when physical memory is limited: a computer can overcome memory shortages by temporarily moving data from RAM to disk storage.



The kernel treats physical pages as the basic unit of memory management.

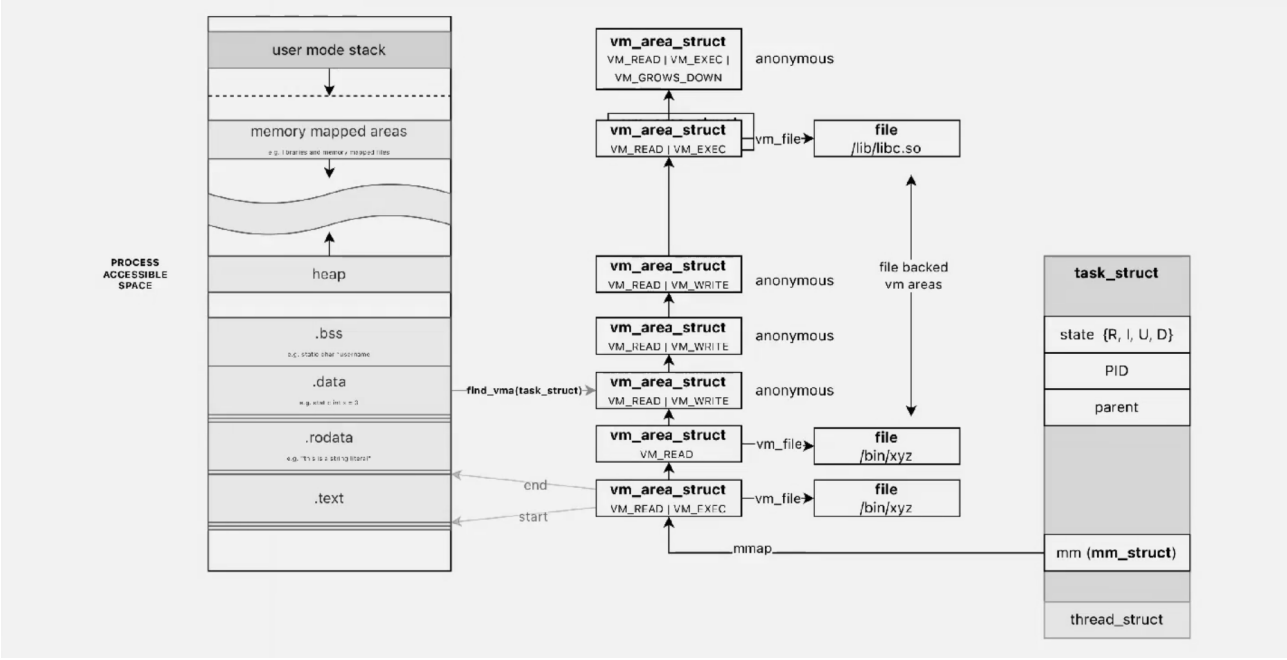
The kernel represents every physical page with a `struct page` structure which doesn't describe the data contained in it , but the properties of the physical page.

The important point to understand is that the page structure is associated with physical pages, not virtual pages. Therefore, what the structure describes is transient at best. Even if the data contained in the page continues to exist, it might not always be associated with the same page structure because of swapping and so on.

Some main concept to keep in mind:

- **Virtual Memory Area** (VMA) is a region in a process's virtual address space which main purpose is to handle mappings of virtual memory into physical memory.
  - VMA can be mapped to a file (with backing store) or be anonymous (e.g., stack, heap).
  - Anonymous areas (heap, stack, bss) initially map to a zero page and employ Copy-On-Write (COW) if written.
  - Backing store areas are derived from PT_LOAD segments in the ELF file.
  - shared or private.

  - Permissions: Readable, writable, executable.
  - `VM_IO` flag indicates mapping of a device's I/O space.
  - VMAs are created by a process using the `mmap()` function.
- **Paging** is a memory management scheme that eliminates the need for contiguous allocation of physical memory. It allows the physical address space of a process to be non-contiguous, which helps in efficiently utilizing memory and simplifies memory management.
- Demand Paging: Pages are loaded into memory only when they are accessed, not all at once.
- Page faults occur when a process accesses a page not present in memory.
- Linux uses a red-black tree of VMAs for efficient.
- **Process Address Space** is the range of addressable memory for each process which includes text (code), data, heap, and stack segments.



6.1 Kernel address space

Kernel Logical Addresses are a subset of kernel addresses that are directly mapped to physical memory. They are used for memory that is frequently accessed or needs to be accessed quickly, such as memory used by DMA (Direct Memory Access).

- Directly mapped to physical addresses starting from 0.
- Correspond to contiguous physical memory.
- Accessible by Direct Memory Access (DMA).
- Managed using the `kmalloc` function.

For the most part, only hardware devices require physically contiguous memory allocation but for performance, most kernel code uses `kmalloc()`.

Kernel Virtual Addresses are not directly mapped to physical memory. They are helpful in situations where the kernel needs to allocate large buffers but can't find a continuous block of physical memory. The allocation and management of these addresses are done using the `vmalloc` function.

- Not contiguous in physical memory.
- Ideal for large buffers where contiguous physical memory is scarce.
- Managed using the `vmalloc` function.

6.2 Page Allocation

In the Linux kernel, each memory zone is characterized by the total size of its pages and an array of lists of free page ranges. When the kernel requires contiguous pages and free pages are available, it utilizes the **buddy algorithm** and a couple of others.

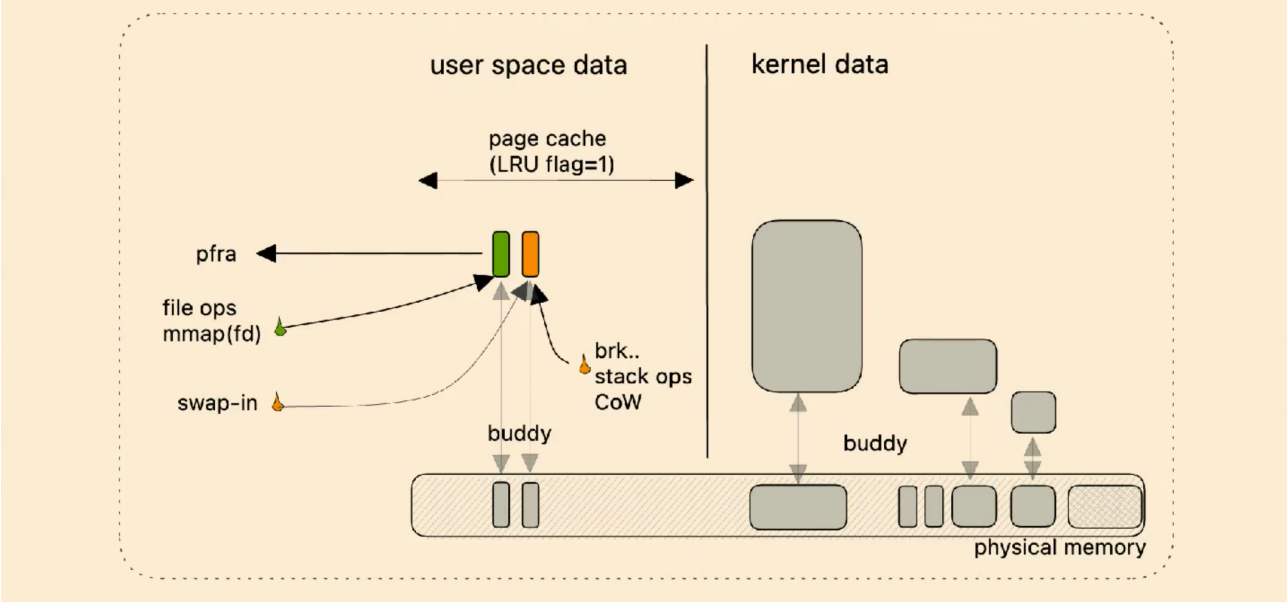
6.2.1 Buddy Algorithm

The buddy algorithm aims to reduce the need for splitting large free memory blocks when smaller requests are made. This is important for two reasons:

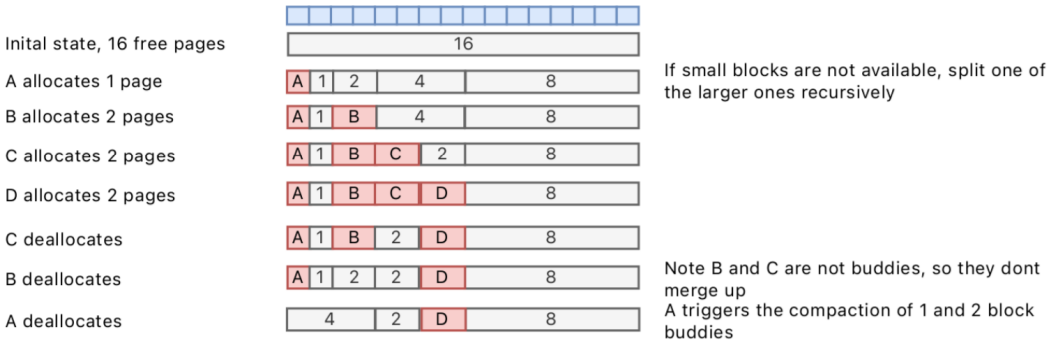
1. The kernel often needs contiguous pages for Direct Memory Access (DMA).
2. Using larger, 4 MB pages instead of smaller ones decreases the frequency of Translation Lookaside Buffer (TLB) misses.

This algorithm manages the merging/splitting of memory blocks:

- **Allocation**: To allocate a block of a given size, the kernel first checks the free list of the requested size and higher. If a block of the requested size is available, it is allocated immediately. If not, a larger block is split into two smaller blocks, with one half being allocated and the other returned to the free list. This process may recur as necessary.
- **Deallocation**: Upon freeing memory, the kernel checks if there is a free 'buddy' block adjacent to the one being freed. If so, the two blocks are merged into a single larger block. This process is recursive.



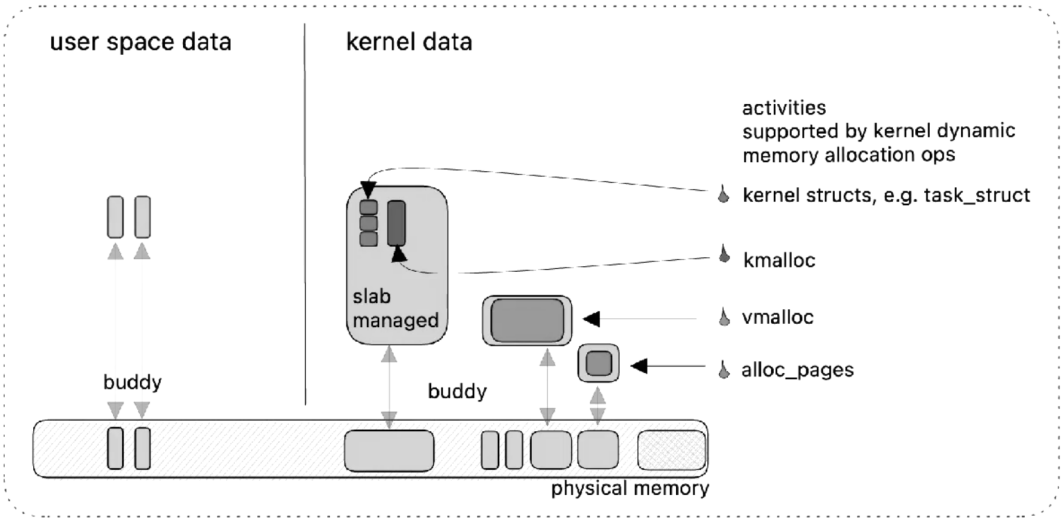
- It works by dividing memory into blocks of various sizes, which are **powers of 2**. When a request for memory is made, the Buddy System finds the smallest block that will satisfy the request. If a block is larger than needed, it's split into "buddies."
- The Buddy System is efficient for managing varying sizes of memory requests and for minimizing fragmentation.



6.2.2 Beyond the Buddy Algorithm

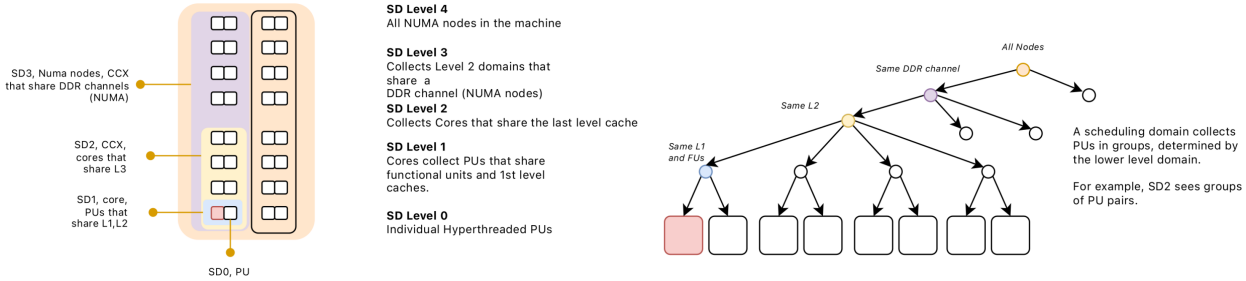
Allocating and freeing data structures is one of the most common operations inside any kernel. The Buddy System, while efficient for larger allocations, is not ideal for these smaller structures due to potential internal fragmentation and the need for synchronization via locks. Therefore, Linux kernel buffers its request through two additional fast allocators:

1. **Quicklists**: used to optimize the allocation of frequently used small objects, particularly for per-CPU allocations.
   - They are essentially a list of pre-allocated memory pages or blocks for certain types of objects, allowing for rapid allocation without having to frequently query the general-purpose memory allocators.
   - Quicklists can reduce contention on global resources in a multiprocessor system.
2. **Slab Allocator**: the slab layer acts as a generic data structure caching layer for kernel objects such as file objects, task structures, etc.:
   - When an object of that type is needed, it can be quickly allocated from a pre-existing memory chunk called slab, reducing the overhead of frequent allocations.
   - The Slab Allocator is not a replacement for the Buddy System but rather a higher-level memory manager that works on top of it. It uses the Buddy System for obtaining larger chunks of memory, which are then subdivided into slabs. The slab allocator provides two main classes of caches:
     - **Dedicated**: These are caches that are created in the kernel for commonly used objects (e.g., `mm_struct`, `vm_area_struct` ,etc. . .).
     - **Generic** (size-N and size-N(DMA)): These are general purpose caches, which in most cases are of sizes corresponding to powers of two.



6.2.3 Zonal page allocation in Linux

Physical memory can be organized into **NUMA** banks or nodes.



Zonal Page Allocation is a memory management strategy which aims to allocate memory physically near the CPU requesting it. For this the memory is organized hierarchically in zones. **Zones** contain information about the total size and lists of free page ranges.

Zone	Description	Physical Memory
ZONE_DMA	DMA-able pages	$<$ 16MB
ZONE_NORMAL	Normally addressable pages	16 $\sim$ 896MB
ZONE_HIGHMEM	Dynamically mapped pages	$>$ 896MB

6.3 Physical Address Space in Linux

Unlike file-mapped VMAs, anonymous VMAs don't have a direct file backing on the disk. **Common Uses are**:

- **Heap**: For dynamically allocated memory in a process (e.g., using `malloc` in C).
- **Stack**: To store function call frames, local variables, etc.
- **System Calls**: In scenarios like using `mmap` without a file descriptor or with the `MAP_ANONYMOUS` flag.

Obv anonymous pages are typically initialized to zero upon first access, contrasting with file-backed pages, which are initialized with the corresponding file content.

Anonymous pages can be moved to a swap area on the disk to free up physical memory, and they are frequently used with COW, particularly during process forking (`fork` system call), where the child process initially shares the same pages as its parent, and duplication of pages occurs only upon modification by either process.

6.3.1 User space page caching

Linux's User Space Page Caching goal is to cache data **from the filestystem** in RAM. Disk data, upon being read, is cached in the Page Cache, represented as a collection of `struct page`, each describing a memory page's properties and status.

This `struct page` describes **physical page** (where actual data is) attributes in memory, containing essential details like mappings, counters, and flags.

```
// Representation of a page of memory in the Linux kernel
struct page {
    unsigned long flags;         // Status flags for the page
    atomic_t _count;             // Reference count
    struct address_space *mapping; // Pointer to the address space
    pgoff_t index;               // Offset within the address space
    struct list_head lru;        // LRU list linkage
};
```

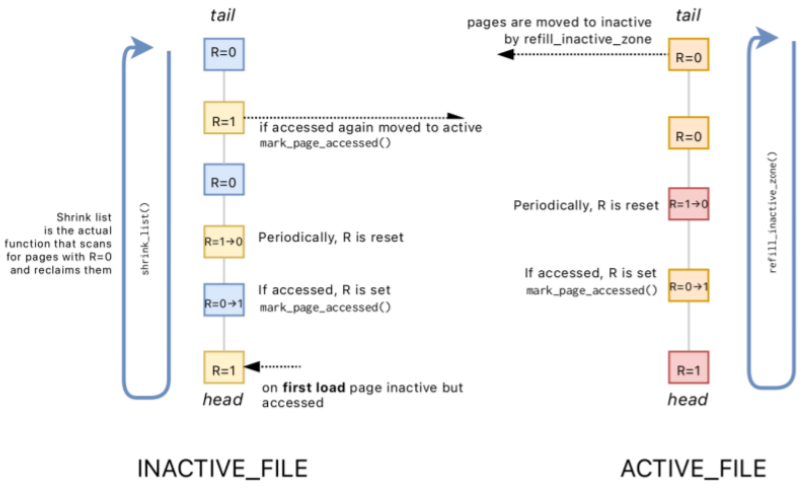
Exits two **mapping mechanisms:**

1. **Forward Mapping (`file descriptor + offset -> struct page`):** Enables direct access to the physical page containing a file's data at a specified offset.
2. **Backward Mapping (`struct page -> [VMA]`):** Facilitates invalidation of page table entries for shared or CoW (Copy-on-Write) pages across different processes, crucial for managing file-backed pages.

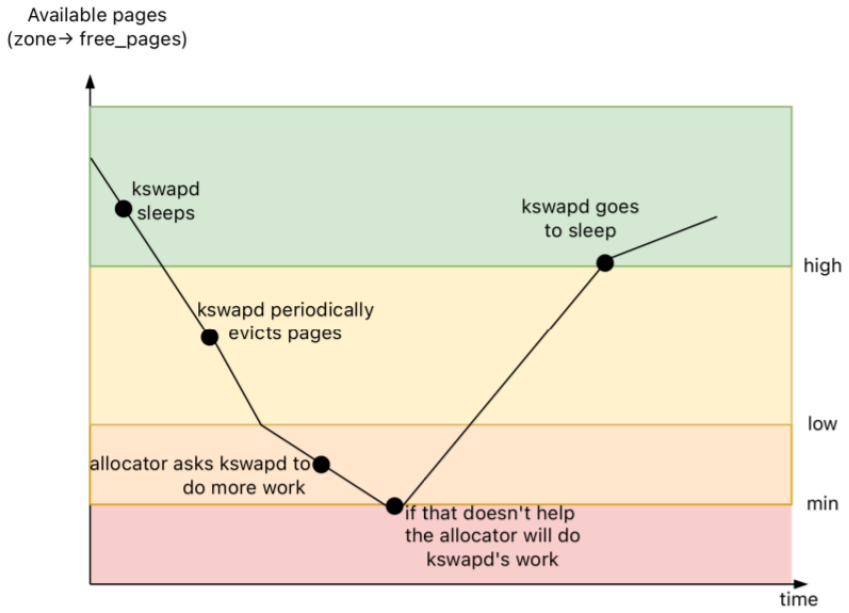
```
graph TD
        subgraph Backward Mapping
        VA[struct page] --> VMA
    end

    subgraph Forward Mapping
        FD2[File Descriptor + Offset] --> PTE1b[struct page]
        PTE1b --> PA2[Physical Address]
    end
```

To manage memory pressure (no infinite RAM), Linux employs a **Page Frame Reclaim Algorithm** which organizes pages based on their activity into 'active' and 'inactive' lists within each memory zone. Linux uses the **clock algorithm** as PFRA, which approximates well the LRU (Least Recently Used) strategy (ideal LRU is not achievable in practice because you would have to precisely track the last usage timestamp). Clock algorithm is simple:

- Pages are organized in 2 circular lists: - **Inactive List**: Pages initially reside here. Pages move to the 'active' list after two accesses, indicating they are used regularly. - **Active List**: Contains pages that have been accessed multiple times.
- If a page's reference bit (R) is 0, indicating it hasn't been used recently, that page is evicted. If the reference bit is 1, it is cleared, and the clock hand moves to the next page.



The PFRA is triggered under certain conditions based on a **zone-based watermark** model to ensure enough free pages are always available for requests.



Simplifying:

- **Pages High**: If free pages fall below this level, `kswapd` periodically runs the PFRA.
- **Pages Low**: When free pages hit this level, it triggers a deferred `kswapd` invocation.
- **Pages Min**: At this critical level, the buddy allocator itself will invoke the PFRA to free up pages.

7 Virtualization

A Virtual Machine (VM) is an effective, isolated replication of an actual computer designed for running a specific operating system (OS). It's based on a virtual machine monitor (VMM), also known as a **hypervisor**. There are two types of hypervisors:

- **Type 1 Hypervisor**: Also known as a native or bare-metal hypervisor, it operates directly on the hardware without an underlying host OS.
- **Type 2 Hypervisor**: This runs on top of a host operating system, like KVM or VirtualBox.

A mention also goes to paravirtualization, which is when the guest os is modified to work closely with the hypervisor, leading to improved performance and reduced virtualization overhead. The VMM has total control over system resources, ensuring:

- **Fidelity**: The VM behaves in the same way as the real machine.
- **Safety**: The VM is restricted from bypassing the VMM's management of virtualized resources.
- **Efficiency**: Programs running within the VM experience little to no significant drop in performance.

The reasons for using a virtual machine are numerous:

- **Consolidation**: Multiple VMs can run on a single physical machine, maximizing hardware utilization by running one machine at full capacity instead of multiple machines at partial capacity.
- **Adaptability**: VMs can quickly adjust to changing workloads.
- **Cost Reduction**: Data centers can reduce both hardware and administrative expenses.
- **Scalability**: VMs can be easily scaled horizontally to meet increased demands.
- **Standardization**: They provide a standardized infrastructure across different environments.
- **Security and Reliability**: VMs offer secure sandboxing for applications and can enhance fault tolerance.

7.1 Popek and Goldberg theorem

First of all, the difference between sensitive and privileged instructions:

- **Sensitive**:
  - **Controls Sensitive**: Directly changes the machine status, like enabling or disabling interrupts.
  - **Behavior Sensitive**: Operates differently depending on whether it's executed in user or supervisor mode, which can impact fidelity.
- privileged: privileged (which are restricted and can cause a trap if executed in user mode).

A privileged instructions affect the state within the virtual CPU, represented in the Virtual Machine Control Block (VMCB):

- Guest state: the state of the processor that is saved and restored when exiting and entering the virtual machine.
- Host state: the state of the processor that is restored upon exiting the virtual machine.
- Execution control: specifies how to handle interrupts when the guest operating system is running.
- Specifies whether certain instructions, such as manipulating cr3, should be considered sensitive.
- Exit reason: indicates why the virtual machine exited, such as due to I/O access and which register was involved.
- Enter and exit control: used, for example, when the machine is in root mode and receives an interrupt but the guest has disabled interrupts. It informs the VMM to trap when interrupts are reenabled.

Regarding the Popek and Goldberg theorem:

> "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." theorem, Popek and Goldberg

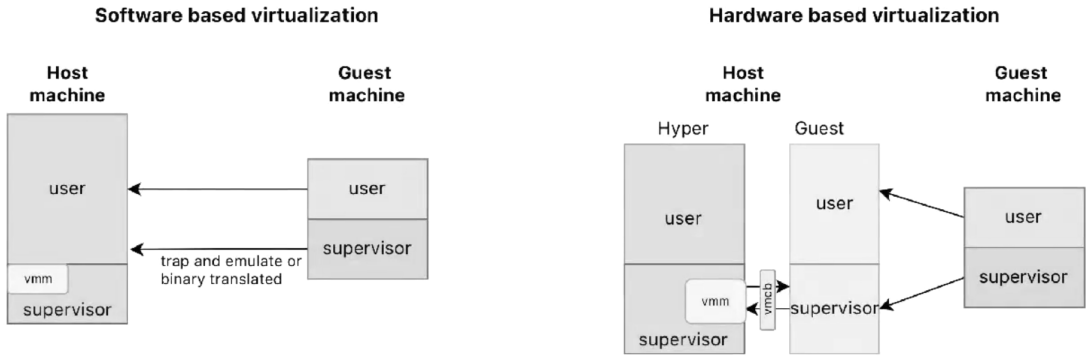
This theorem states that for any conventional computer, a VMM can control the execution of guest OS (by intercepting and emulating privileged instructions) only if all sensitive instructions are also privileged: this so that any attempt by the guest OS to execute a sensitive instruction will cause a trap to the VMM, allowing it to handle the instruction appropriately.

This theorem is a **sufficient** condition, not a necessary one as some mechanism exist(ed) to achieve virtualisation for x86 processors not equipped with VT-x.

7.2 Virtualization techniques

Virtualization can be achieved through:

1. **Software-Based** (de-privileging):
   - "Trap and emulate"
   - Together with shadowing it is a basic constituent of software virtualisation is **ring deprivileging**: the process of reducing the privilege level of a virtual machine by moving it from a higher privilege ring to a lower one, such as from Ring 0 to Ring 3 and 1.
2. **Hardware-Assisted**:
   - Modern processors offer built-in support for virtualization, making it easier and more efficient. These processors have additional modes for guest and hypervisor operations.
   - Some instructions aren't trapped, fetched, and executed by the hypervisor, but directly transformed into emulation routines.



7.3 Depriviliging

Deprivileging is when a virtual machine functions with lesser access to hardware and system resources than it would if it ran on a physical system. Deprivileging might pose a problem in terms of fidelity because, if not taken care of, a guest could under- stand that it is running in unprivileged mode.

This is resolved (in software virtualization) through just in time compilation: the JIT compiler detects sensitive or privileged instructions in the guest code that need special handling when the guest OS is deprivileged. JIT makes:

- **Translation**: Instead of executing these instructions directly, the JIT compiler translates them into a sequence of safe, unprivileged instructions that the host system can execute without compromising security or stability.
- **Caching**: Translated instructions are cached so that subsequent executions of the same instructions can use the optimized versions without needing retranslation, improving performance.

It's all about to deprivilege an instruction from ring 0 (highest privilege) to run in ring 1 or another less privileged ring. The rings :

1. **Ring 0**: Kernel mode, full privileges.
2. **Ring 1**: (Rarely used) Driver mode, intermediate privileges.
3. **Ring 2**: (Rarely used) Driver mode, intermediate privileges.
4. **Ring 3**: User mode, least privileges.

7.4 KVM

KVM (Kernel-based Virtual Machine) is a virtualization technology integrated into the Linux kernel, which allows Linux to be a hypervisor, converting it into a type 1 hypervisor.

For higher performance, KVM also supports device pass-through, allowing VMs to directly access physical devices. This is facilitated by the VFIO (Virtual Function I/O) driver, which provides secure direct access to devices using IOMMU (Input-Output Memory Management Unit).

IOMMU (Input-Output Memory Management Unit) provides a mechanism to apply virtual-to-physical address translation and access control for device DMA, similar to how the MMU does for CPU access to memory. This allows devices to use "virtual addresses" in their operations, which the IOMMU then translates to physical addresses according to translation tables it maintains.

7.5 Translation of physical addresses

Regarding the translation of physical addresses from the guest OS to physical addresses on the host:

- in **software based virtualization** the guest physical pages are actually host virtual pages and the whole mapping guest virtual to hardware physical host is called "Shadow Page Table". The guest OS believes it controls the page tables, but the hypervisor actually intercepts and manages the mappings. The hypervisor intercepts these updates, translates them, and updates the shadow page tables accordingly (all of this, under the hood, it's done marking the guest page table as read-only so that then it manages any corresponding traps) that arise out of these modifications).
- in **hardware based virtualization**, as the name suggests all this stuff is managed at hardware level by a mechanism called "Extended Page Tables" . Introduced by Intel VT-x or AMD RVIK (Rapid Virtualization Indexing), EPT is an "extended part" which keeps track of the mapping from the guest physical address to the host physical address.

7.6 hardware/software virtualizations techniques

- KSM (Kernel Same-page Merging) is a feature in Linux designed to save memory across multiple processes or VMs. It works by scanning for physical pages with identical content and remaps all virtual pages pointing to these physical pages to a single copy. This allows the system to free up the surplus physical pages, setting them as copy-on-write.
- Ballooning allows a hypervisor to dynamically adjust the amount of physical memory available to VMs. A special driver installed in the guest operating system allocates memory to itself forcing the guest OS to swap or release less frequently used memory pages back to the host, allowing the hypervisor to reallocate this memory to other VMs.
- Memory overcommitment in virtualization allows assigning more memory to virtual machines (VMs) than the physical memory available. When the actual memory usage approaches or exceeds the physical memory, techniques like ballooning are used to reclaim memory.

7.7 Containerization

Containers are a way to isolate a set of processes and make them think that they are the only ones running on the machine: - Processes running inside a container are normal processes running on the host kernel. - There is no guest kernel running inside the container

Containers are based mainly on 2 technologies:

- Namespaces are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources. In this way each process in Linux has its own network namespace, pid namespace, user namespace and a few others.
- CGroups / control group (met at this chapter)

8 I/O

8.1 Memory and port mapped devices

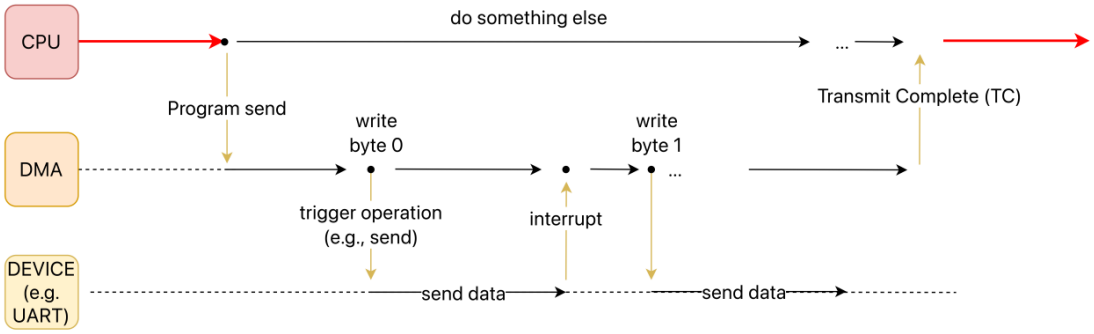
A **bus** is an hardware connection channel that facilitates communication between the CPU and other devices. Communication between the CPU and devices can happen over two buses:

- **Memory bus:** Peripheral device registers are integrated into the CPU's address space. This setup allows the CPU to read from and write data to the peripherals, as well as send commands, using standard memory access instructions. It simplifies the interaction between the CPU and the peripherals by treating peripheral I/O as regular memory reads and writes. Memory-mapped I/O is more used than port-based I/O
- **Port bus:** the port bus typically has its own address space separate from the main memory, necessitating specialized commands for data transfer and control.

8.2 Devices and CPU communication

3 main ways to communicate:

- **Polling** involves the CPU constantly checking the status of devices. While simple, it is inefficient as it consumes significant CPU resources.
- **Interrupts**: Allow devices to notify the CPU of important events, prompting immediate attention. This is more efficient than polling. We can say there are different types of Interrupts:
  - **Asynchronous Interrupts**: Generated by hardware devices.
  - **Maskable Interrupts**: Can be ignored by the CPU.
  - **Non-Maskable Interrupts**: Always recognized by the CPU.
  - **Synchronous Interrupts**: Produced by the CPU itself during instruction execution.
  - The **Interrupt Handling Process** prioritizes the interrupts and addresses an interrupt event by executing an interrupt handler stopping what the CPU is doing.
  - In **Multi-Core Systems**, interrupts are managed by a dedicated interrupt controller that assigns them to the appropriate CPU core based on priority.
- **DMA** (Direct Memory Access) which involves using a DMA controller. The DMA controller is like an additional device that independently manages data transfers between different devices and memory. It does so without needing the CPU's involvement, thereby **decoupling** it from the transfer process.



8.2.1 Interrupts

8.2.1.1 Interrupts and you

1. Why we need interrupts?
2. The only way we know up to now to check whether some hardware condition happened is to check status bits.

3. Repeatedly checking status bits is called **polling**.
4. There is an inherent trade-off between responsiveness to events, and polling period.
5. Some events may be sporadic.
6. Interrupts can be thought as letting hardware call a software function when an event occurs.
7. When the interrupt occurs, the CPU is executing some other userspace or kernelspace code.
8. Interrupts can pause the normal code execution in between any two assembly instructions and jump to a function, the "interrupt service routine (ISR)".

8.2.1.2 Interrupt Service Routine (ISR) Interrupt handlers or Interrupt Service Routine (ISR), the function kernel runs in response to a specific interrupt. What differentiates interrupt handlers from other kernel functions is that: ISRs live in a special context called **interrupt context**, also called atomic context sometimes, because code executing in this context is unable to block.

Top halves versus bottom halves: Because the 2 goals:

- ISR executes quickly
- ISR performs a large amount of work

conflict with each other, it's smart split each ISR into two part:

- top half: most urgent aspects of the interrupt.
- bottom half: defer work until any point in the future when the system is less busy and interrupts again enabled. Often the bottom halves run immediately after the interrupt returns. The key is that they run with all interrupts enabled.

8.2.1.3 Deferring work In Linux, the concept of "**deferring work**" involves postponing the execution of a task until a later time. Before diving in the common methods for deferring work in Linux, let's talk about **reentrancy** which is: a property of code that allows it to be safely called again before its previous execution is complete.

Reentrant Code:

- Does not rely on shared data.
- Uses local variables or ensures exclusive access to shared resources (e.g., through locking mechanisms).

Non-reentrant Code:

- Uses static or global variables without proper synchronization.
- Relies on state that might be altered by concurrent executions.

Three methods are available for deferring work:

1) **SoftIRQs**: difficult to program them directly, mainly used only by networking and block devices directly in their interrupt handlers. Same type of SoftIRQs can run simultaneously on several processors and for this reason it must be reentrant.
2) **Tasklets**: offer an easier interface for defining bottom-half processes.
   - the same tasklet **cannot** run on multiple CPUs simultaneously by design
   - cannot sleep.
3) **Work Queues**: Work queues accommodate deferred tasks that may need to **sleep** or **wait** for resources, something that SoftIRQs and Tasklets aren't capable of. It's the general mechanism for submitting work to a worker kernel thread: operate in **process context** rather than in interrupt context. This means that:
   - they can sleep
   - less time-pressured

8.2.1.3.1 Tasklets example Tasklets could be created statically or dynamically:

```
DECLARE_TASKLET(packet_tasklet, packet_tasklet_handler, 0);
```

The tasklet handler must match the following prototyte:

```
void tasklet_handler(unsigned long data);
```

Scheduling the tasklet:

```
tasklet_schedule(&my_tasklet);  /* mark my_tasklet pending */
```

As with softirqs, tasklets cannot sleep, so you cannot use semaphores or other blocking functions in a tasklet. Tasklets are rapresented as a list of `tasklet_struct` :

```
struct tasklet_struct {
    struct tasklet_struct *next;
    unsigned long state; /* 0, scheduled or running */
    ...
    void (*func)(unsigned long);
    unsigned long data;
};
```

8.3 Linux Device Management

The inclusion of devices into the file system is a key idea that makes device management easier. Highlights of this approach:

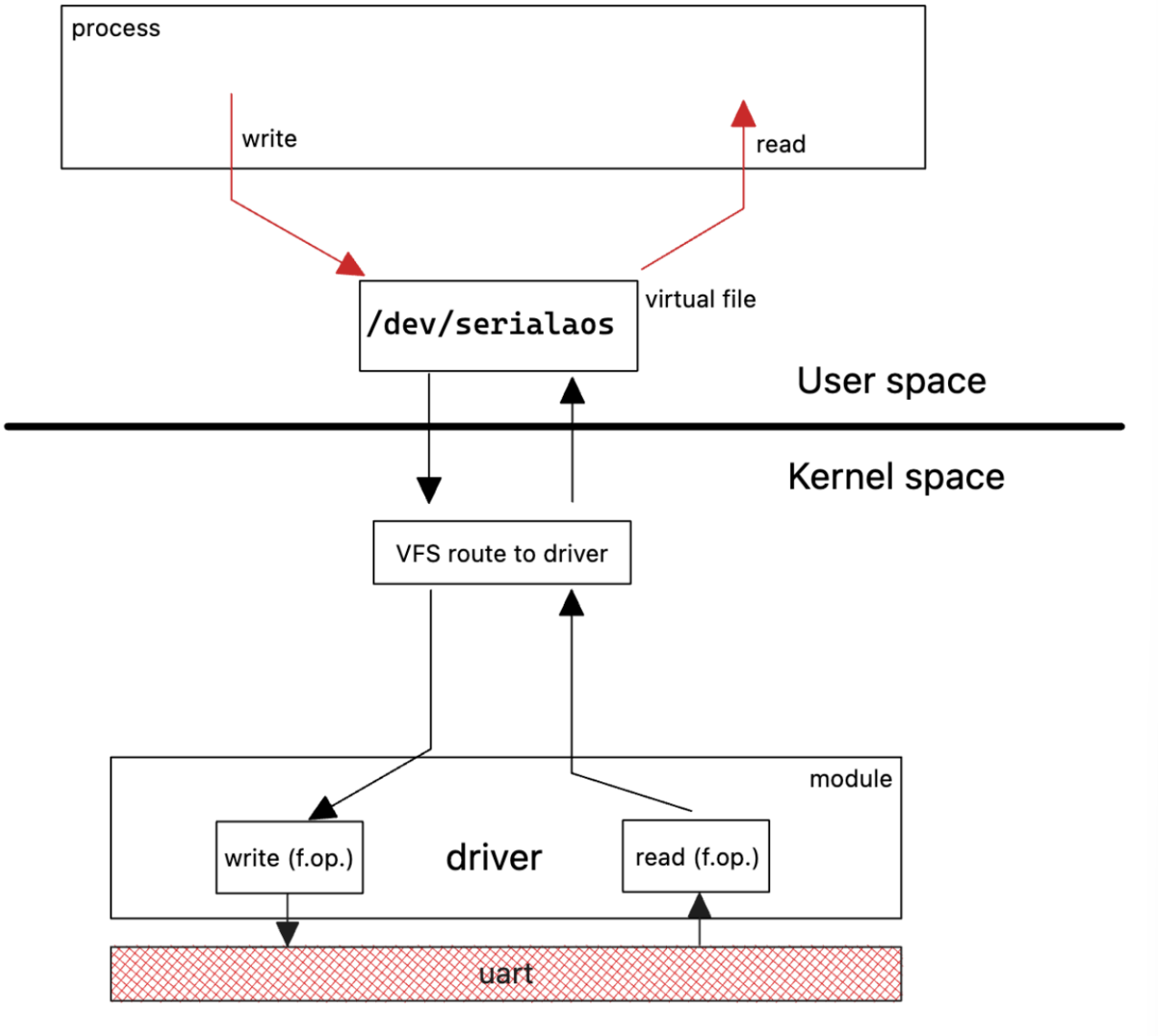
- **Special Files for Devices**: Linux treats devices like files, represented as special files within the file system. This approach is based on the Unix philosophy of "everything is a file". Each device is assigned a path name, typically located in the `/dev` directory.
- These files are identified by **major** and **minor** numbers:
  - **Major Device Number**: This is used to identify the driver associated with the device.
  - **Minor Device Number**: When a driver manages multiple devices of the same type, each device is assigned a unique minor number.

In summary, modern Linux systems handle all of this using:

- `udev` for managing device nodes in `/dev`
- `sysfs` for exporting device their attributes, and their relationships to user space.

Behind this there is the Virtual File System (VFS): it exposes information about devices and drivers, as well as the relationships and hierarchy among different components, to user space.

`devfs` is now deprecated and removed in favor of `udev`. `udev`, which is a user-space daemon handles the creation and removal of device nodes, and manages device permissions and symlinks dynamically.



8.3.1 Device Categories

Linux categorizes devices based on their type and function into:

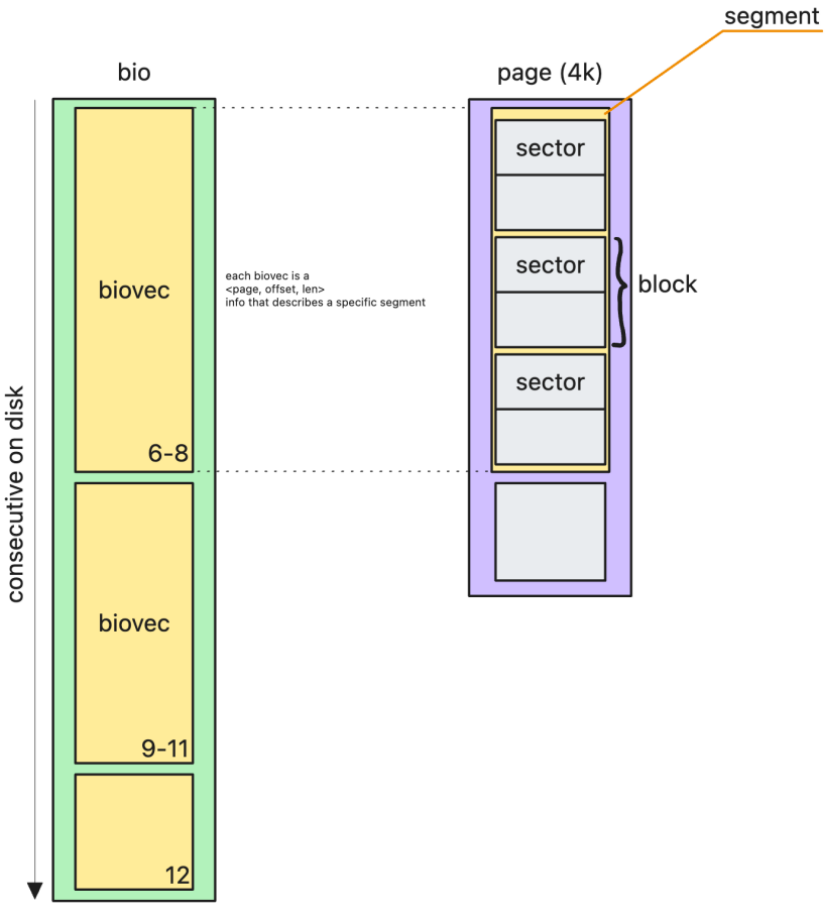
- **Character Devices**:
  - **Functionality**: Operate with a stream of characters, accessed one at a time. Interaction with these devices is immediate, making them ideal for hardware that requires prompt data transfer.
  - **Examples**: Serial ports, keyboards, terminal devices.
  - **Access**: Via special files in `/dev/`.
  - **Key Point**: They do **not use buffering**, directly affecting the device.
- **Block Devices**:
  - **Functionality**: Organized in blocks for random access, employing buffering and caching, suitable for large data storage and retrieval.
  - **Examples**: Hard drives, SSDs, USB drives.
  - **Access**: Through special files in `/dev/`.

- **Network Devices**:
  - **Functionality**: Handle data packet transmission and reception over network interfaces, critical for network communication.
  - **Examples**: Ethernet adapters, wireless interfaces.
  - **Access**: Managed through network configuration tools, not directly via `/dev/`.

8.3.1.1 Block Devices

In the context of the block layer:

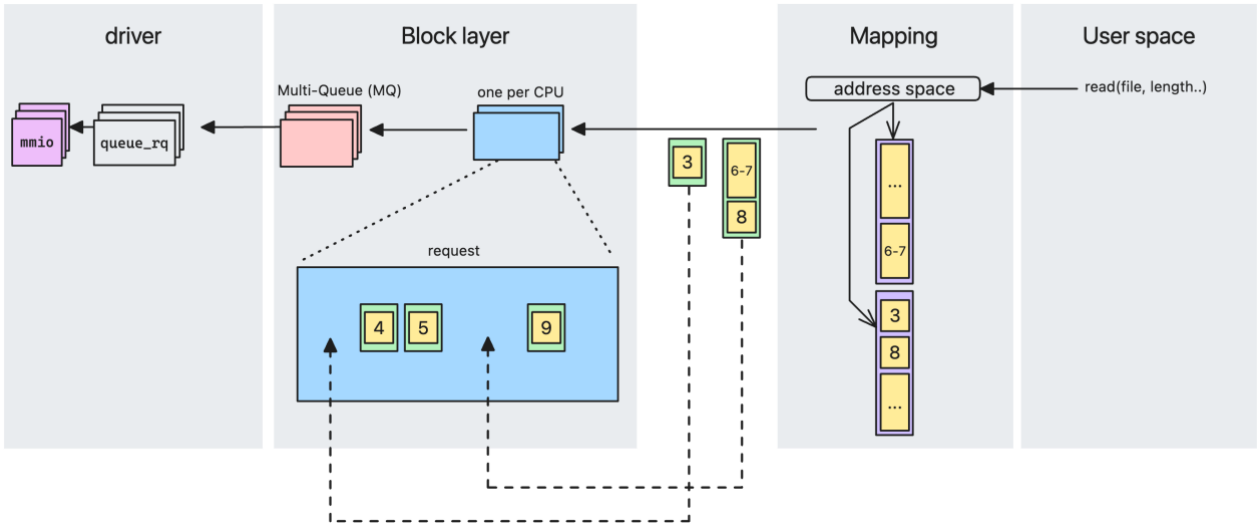
- **Segments** refer to the parts of a page that correspond to contiguous sectors on the disk.
- Internally, the mapping layer works with multiples of sectors called **blocks**.
- **bio**: This structure represents a block I/O operation. A bio can contain multiple segments (bio_vec), each pointing to a **contiguous** area in memory where data resides or will be placed after an I/O operation.
- **bio_vec** : Represents a segment with:
  1. The page where the data should be read or written.
  2. The offset within that page.
  3. The length of this segment.



request_queue: This structure represents the queue of pending I/O requests for a block device. It helps in scheduling and optimizing these requests before they are dispatched to the actual device driver. Techniques such as merging adjacent requests are employed to enhance performance through the "hw schedulers". There also I/O software schedulers on top of hw schedulers to optimize everything:

- kyber

- noop: mainly used for SSDs
- MQ-deadlines
- Budget Fair Queue



`queue_rq()` is a function which queues any request (rapresented in the kernel by a `struct request`).

8.3.2 High-Level Device Management (The Device Model)

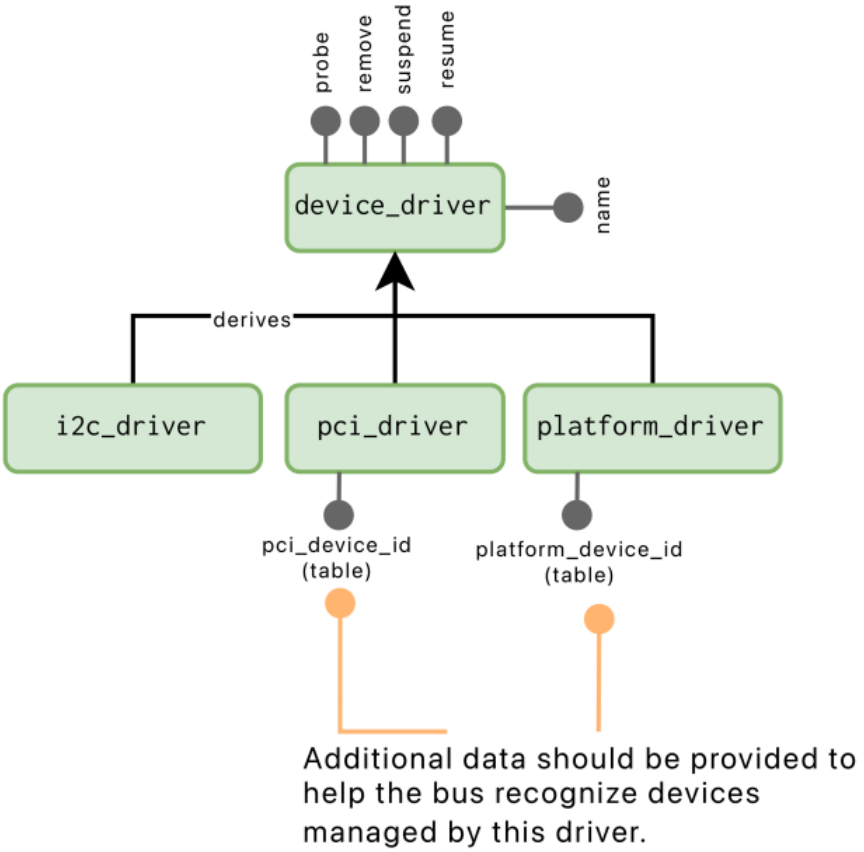
The Linux device model is an abstraction layer which aims to **maximize code reuse** between different platforms in the Linux kernel. This is achieved by the kernel providing a framework and a set of APIs that enable consistent and efficient management of devices. The goals of this kind of frameworks are:

- **device representation** which gives an unified view of a clear structure
- **driver binding**
- **power management**
- **hot plugging**
- expose it to user space (through sysfs and udev)
- **event notifications**

The core components (logically) of this device model are:

- devices
- drivers
- buses

all of these are structures which are extended with custom data. For example, the `pci_driver` extends the generic device structure adding PCI-specific infos.



To conclude, just remember that **kobjects** are the underlying foundation of the device model: they represent kernel objects such as bus, devices, drivers, and modules to user space via the `sysfs` virtual filesystem:

- (kobject -> directory, attributes -> file)

An important feature of kobjects is their ability to emit **uevents**. These are notifications sent to user-space tools like udev, signaling any changes in the Kobjects. Sysfs exports information about these objects to user space, providing structured access to hardware details.

9 Booting

The **Unified Extensible Firmware Interface**(UEFI) is a modern replacement for the traditional BIOS: it's an interface between the system firmware and the OS, enhancing the boot process.

When the system powers on:

- UEFI, written in C, modular and runs on various platforms quickly takes control to initialize system hardware and load firmware settings into RAM.
- UEFI uses a dedicated FAT32 partition, known as the EFI System Partition (ESP), to store bootloader (startup) files for various operating systems.

UEFI uses GPT (GUID Partition Table) overcomes the size limitations of BIOS and allows more flexible partitioning (supports disks larger than 2.2TB and can handle up to 9.4 zettabytes with 64-bit logical block addressing).

During machine boot-up, the firmware is loaded into main memory, performs necessary checks and launch the bootloader which mounts the filesystem and loads the kernel image.

During these system checks, it's also validated the signature of the bootloader. This is called **secure boot** (which is a security feature of UEFI standard) and ensures that a device will only boot using software trusted by the (Original Equipment Manufacturer (OEM)). Its primary purpose is to protect the boot process from attacks such as rootkits and bootkits, by using a set of cryptographic keys to authenticate allowed software. These keys are stored in the TPM (hardware component).

In Linux, the **init process** is started at the end of the kernel boot. The init process has a PID of 1 and PPID of 0. It is the ancestor of all user-space processes and starts all enabled services during startup. The two common implementations of init are SysVinit and **systemd**. The last one is very popular:

- It is responsible for starting/stopping services and managing their dependencies.
- systemd provides a way to define system services using unit files.

9.0.1 Discoverability

When booting, the operating system (OS) needs to be aware of several things:

1. The devices that are already present on the machine.
2. How interrupts are managed.
3. The number of processors available.

To address these needs, two standards have been developed to provide the kernel with all the necessary platform data:

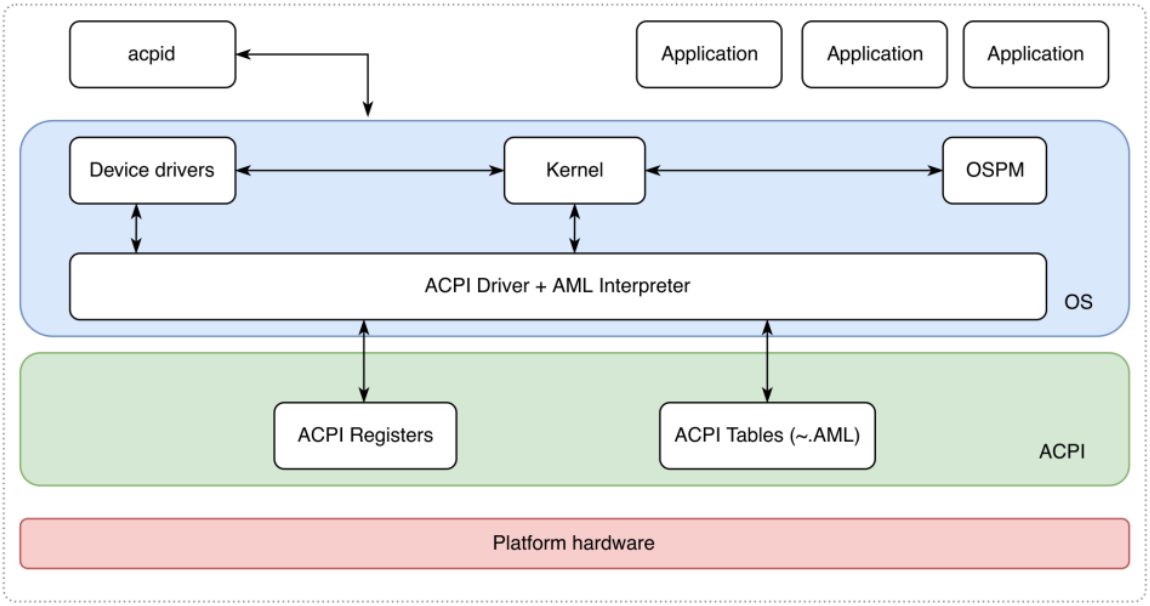
- **Advanced Configuration and Power Interface** (ACPI): This standard is primarily used on general-purpose platforms, particularly those utilizing Intel processors.
- **Device trees**: This standard is mainly used on embedded platforms. Device trees aid in discoverability and provide information about the platform's configuration.

9.1 ACPI: Advanced Configuration and Power Interface

ACPI helps with discoverability, power management, and thermal management. It was developed by Intel, Microsoft, and Toshiba in 1996. The key benefits of ACPI are:

- It provides an open standard for operating systems to discover and configure computer hardware components.
- It enables power management features such as putting unused hardware components to sleep.
- It supports auto-configuration, including Plug and Play and hot swapping.
- It facilitates status monitoring.

There is no need to include platform-specific code and a separate binary kernel for each platform.



9.1.1 Power managament using Device states

Power management refers to the various tools and techniques used to consume the minimum amount of power necessary based on the system state, configuration, and use case.

The power management in modern computing is a critical challenge linked to the Dennard scaling: - Named after Robert Dennard - Observed that smaller transistors keep power density constant - Enabled higher frequency without significant power increase

Breakdown of Dennard scaling: - Occurred around mid-2000s - Issues controlling leakage current and other non-ideal effects - Shrinking transistors no longer proportionately reduced power usage

- **Power management in modern computing:**
  - Critical challenge linked to Dennard scaling
- **Dennard scaling:**
  - Named after Robert Dennard
  - Observed that smaller transistors keep power density constant
  - Enabled higher frequency without significant power increase
- **Breakdown of Dennard scaling:**
  - Occurred around mid-2000s
  - Issues controlling leakage current and other non-ideal effects
  - Shrinking transistors no longer proportionately reduced power usage

This led to the concept of the "power wall," where the industry shifted towards multicore processors to improve performance. In this context, addressing the power management problem is challenging to do mathematically. The primary source of power consumption is related to the frequency and voltage of each system device. Key Questions:

1. What voltage should be used for each device?
2. What frequency should be used for each device?
3. How does switching between different frequencies and voltages impact the system?

Power management is complex due to the interdependence of multiple variables. To simplify this, ACPI uses device states.

ACPI defines a hierarchical state machine to describe the power state of a system. There are five different power state types: G-type, S-type, C-type, P-type, and D-type.

- G-type states group together system states (S-type) and provide a coarse-grained description of the system's behavior. They can indicate whether the system is working, appearing off but able to wake up on specific events, or completely off.
- S-type states, also known as system or sleep states, describe how the computer implements the corresponding G-type state. For example, the G1 state can be implemented in different ways, such as halting the CPU while keeping the RAM on, or completely powering off the CPU and RAM and moving memory content to disk.
- C-type states, specific to the CPU (CPU State), refer to the power management capabilities of the CPU. These states allow the CPU to reduce its clock speed (C0) or even shut down completely when idle to save energy (C1-C3). C-states can only be entered in an S0 configuration (working state) and are usually invoked through a specific CPU instruction, such as MWAIT on Intel processors.
- P-type states (Performance State) are a power management feature that allows the CPU to adjust its clock speed and voltage based on workload demands. P-states only make sense in the C0 configuration (working state).
- D-type states (Device State) are the equivalent of C-states but for peripheral devices. They represent the power management capabilities of the devices.

Above all this system, at the end, the orchestra director (kernel) uses OSPM (Operating System-directed configuration and Power Management) to set the appropriate sleep state for a CPU when it's idle, through ACPI.

An example: when the scheduler identifies that more performance is required, it communicates this desired change to the CPUFreq module in OSPM. CPUFreq then interacts with ACPI and the Performance Control Machine register is modified.

Also the user-space can interact with the kernel's ACPI through an ACPI daemon, like acpid in Linux, which listens for ACPI events and executes predefined scripts in response.

10 Cpp

10.1 Classes

A class in C++ is defined using a specific syntax and is typically placed in a header file (`.h` or `.hh`) that needs to be imported in implementation files (`.cpp` or `.cc`).

Each class has special member functions:

1. **Constructor**: initializes an object that can take parameters. If not defined, the compiler will generate a default one.
2. **Copy constructor**: initializes an object using another object of the same class as an argument. If not defined, the compiler will generate a default one.
3. **Move constructor**: initializes an object from an value. Unlike a copy constructor, which creates a new object as a copy of an existing object, a move constructor transfers resources from a source object to a new object, effectively "moving" the data.
4. **Destructor**: called implicitly when the object goes out of scope. If not defined, the compiler will generate a default one. It does not take any parameters and includes cleanup/resource release code.

To instantiate an object of a class:

- Use `Car c;` which is equivalent to `c();` for constructors with no parameters.
- Use `Car c(<params-list>);` for constructors with parameters.

- Use `Car c{<params-list>};` as a more modern approach where the compiler checks for and disallows narrowing conversions. Example:

```
int x{7.9};  // Error: narrowing conversion from 'double' to 'int'
char c{999}; // Error: narrowing conversion might occur
```

10.1.1 Encapsulation, Inheritance and polymorphism

Object-oriented programming defines three fundamental properties for a class type data: encapsulation, inheritance, and polymorphism. In the context of C++, these properties are implemented in specific ways.

Encapsulation means to properly restrict the access to some of the members We can set a member as:

- **public**: can be accessed by code outside of the object
- **private**
- **protected**: permits to derived classes to access protected members of the base class

```
#include <iostream>

class Vehicle {

    public:
        // Constructor with an initializer list
        Vehicle() : nr_wheels(4), x(0) {}

        virtual void move_one_step_forward() {
            this->x += 1;
        }

        int num_of_wheels() const {
            return nr_wheels;
        }

    protected:
        int nr_wheels;
        int x; // Position
};


class Car : public Vehicle {

    public:
        void move_one_step_forward() override {
            this->x += 100; // Moves the car 100 units forward
        }
};


int main() {
    Car my_car;
    // Move car one step forward
    my_car.move_one_step_forward();

    // Print number of wheels
```

```
    std::cout << "New position: " << my_car.num_of_wheels() << std::endl;

    return 0;
}
```

Some member functions may be pure **virtual**, which means that derived classes must provide their own implementation. An abstract class refers to a type of class whose member functions are all pure virtual. This is useful when defining a general interface but leaving the implementation to the specific case.

Pure virtual member function definitions must include the virtual keyword as a qualifier and an "=0" termination.

10.1.2 Operators

In addition to member data and functions, we can define and implement different operators in C++.

- The copy assignment operator is basically used for updating an object so it matches another object. By default, the compiler provides us with an implementation of this operator.
- The move assignment operator "moves" the state of the right-hand object into the assigned object, leaving the right-hand object empty. e.

The Rule of Five in modern C++ states that we should define or delete the following five member functions for classes that deal with resource management:

- the copy constructor
- the move constructor
- the copy assignment operator
- the move assignment operator
- the destructor.

10.1.3 Namespaces

Namespaces in C++ are used to prevent name conflicts by creating "named scopes" for the declaration of symbols, such as functions and classes. Symbols with the same name from different namespaces do not introduce conflicts.

The built-in namespace std includes the definitions of most of the C++ library symbols. Additionally, it is possible to define nested namespaces.

10.1.4 Templates

Templates in C++ are a useful feature for implementing functions and classes that can be used with different data types. They allow for code reuse and avoid writing the same code for each specific data type.

The data type for a template is determined during compilation, and the compiler generates the appropriate code to handle different data types. This means that we can have a single implementation that works for multiple cases.

Template code is typically placed in header files, making it easy to use and reuse in different parts of a program.

However, using templates has some drawbacks. Compiling code with templates can be slower and result in larger executable files. Modifying a template class often requires recompiling a significant portion of the project. Debugging can also become more complex due to lengthy error messages from the compiler.

```
template<typename T>
T duplicate(T param) {
    return param*2;
}
//Calling template function
```

```
void testTemplate() {
    cout<<duplicate(2)<<endl; //T=int
    cout<<duplicate(2.5)<<endl; //T=double
}
```

10.1.4.1 Standard Template Library (STL)

The std namespace is probably the most important namespace in the library that encapsulates all the classes and functions defined by the STL library.

The STL (Standard Template Library) is an important part of the `C++` language. It includes the definition of a set of template-based containers.

Container	Notes	Access Efficiency	Insert/Delete Efficiency
Vector	Contiguous storage; reallocates when capacity exceeded.	O(1)	O(n) at middle or start
List	Double-linked list; extra memory for pointers. Good for sorting.	O(n)	O(1)
Forward List	Single-linked list; less memory than `list`.	O(n)	O(1)
Array	Fixed-size; contiguous storage.	O(1)	Not applicable
Map	Stores key-value pairs; implemented as a binary search tree.	O(log n)	O(log n)
Set	Stores unique elements; like `map` but only keys.	O(log n)	O(log n)
Unordered Map	Stores key-value pairs; uses hashing. Faster than `map`/`set`.	O(1) or O(n)	O(1) or O(n)

Containers are used to build collections of objects. When choosing a container, it is important to consider design aspects such as:

- **Read access patterns**
  - How frequently we need to access objects? Do we need a direct access or can we tolerate an iteration over the entire collection?
- **Write access patterns**
  - Does the content of the collection change frequently? (many add/remove operations)
- **Memory occupancy**
  - Is the additional overhead introduced by the container an issue?

10.1.5 Smart pointers

Feature/Issue	Description
Raw Pointers	Traditional way of dynamic memory allocation using `new`/`delete`. Prone to memory leaks, segmentation faults, and unclear ownership.
Smart Pointers	Introduced to automate memory management and prevent common issues associated with raw pointers. Defined in `<memory>` header.

Feature/Issue	Description
std::unique_ptr	Represents exclusive ownership of a dynamically allocated object. Not copyable, only moveable. Automatically deallocates memory when the pointer goes out of scope.
std::shared_ptr	Allows for shared ownership of an object through reference counting. Memory is automatically freed when the last shared pointer to an object is destroyed or reset.
std::weak_ptr	Complements `std::shared_ptr` by providing a non-owning "weak" reference to an object managed by `shared_ptr`, preventing cyclic references (cyclic references are a problem because they keep alive indefinitely each other).

Example of `shared_ptr` :

```cpp
#include <iostream>
#include <memory>

class Resource {
public:
    Resource() { std::cout << "Resource acquired\n"; }
    ~Resource() { std::cout << "Resource released\n"; }
};

int main() {
    std::shared_ptr<Resource> res1 = std::make_shared<Resource>(); // Resource acquired
    std::shared_ptr<Resource> res2 = res1; // Both res1 and res2 now own the Resource

    std::cout << "res1 use count: " << res1.use_count() << '\n'; // Outputs 2
    res2.reset(); // res2 releases ownership; Resource is still alive
    std::cout << "res1 use count: " << res1.use_count() << '\n'; // Outputs 1

    // res1 goes out of scope here, automatically deleting the Resource
    return 0; // Resource released
}
```

10.1.6 Functor or Function object

A function object (functor) is an object hat can be called as if it were a function. This is achieved by defining an `operator()` within the class. Functors are useful because, unlike regular functions, they can have state.

```cpp
#include <iostream>

class Increment {
private:
    int num;
public:
    Increment(int n) : num(n) {} // Constructor initializes the value to add

    // The functor's call operator
    int operator()(int i) const {
        return num + i; // Adds the stored value to the given parameter
    }
};

int main() {
```

```
    Increment incFive(5); // Create a functor that adds 5
    Increment incTen(10); // Create another functor that adds 10

    std::cout << "Increment 5 to 3: " << incFive(3) << std::endl; // Outputs: 8
    std::cout << "Increment 10 to 3: " << incTen(3) << std::endl; // Outputs: 13

    return 0;
}
```

10.1.6.1 Bind expressions

`std::bind` (found in the `<functional>` header) that lets you prepare a function call in advance. It's like setting up a function with some of its parameters pre-filled.

You can specify some arguments now and leave placeholders for others to be filled in later when the function is actually called.

```
int main() {
    // Creating a function that always multiplies its argument by 2
    auto multiply_by_2 = std::bind(multiply, 2, std::placeholders::_1);

    // Now, you can use 'multiply_by_2' like a regular function
    std::cout << "3 multiplied by 2 is " << multiply_by_2(3) << std::endl; // Outputs: 6

    return 0;
}
```

10.1.7 Lambda expressions

Lambda expressions in C++11 let you write quick, unnamed (anonymous) functions right where you need them.

- **Basic Structure:** `[] () {}` is the simplest form of a lambda.
- **Capturing Variables:** The part inside `[]` is about grabbing variables from the surrounding area so you can use them in the lambda. - `[]`: Captures nothing. - `[&]`: Captures everything around by reference. - `[=]`: Captures everything around by making copies. - `[=, &foo]`: Copies everything but references `foo`. - `[foo]`: Only captures `foo` by copying it. - `[this]`: Captures `this` pointer, so you can use the class's member variables and functions.
- **Return Type:** Usually, C++ can figure out what a lambda returns, so you don't have to spell it out.

Examples:

- **Capture Nothing**: `[]() { std::cout << "Hello, World!"; }`
- **Capture by Reference**: `[&]() { ++counter; }` (Imagine `counter` is a variable defined outside the lambda)
- **Capture by Value**: `[=]() { return value + 1; }` (Here, `value` is a variable from outside)

Lambdas are handy for quick operations, like when you're using functions that need other functions as parameters, such as `std::sort` or `std::for_each`.

10.2 Modern C++ threading

`C++11` introduced the class `<thread>` :

Member function	Return value	Description
get_id()	thread::id	Returns an unique identifier object
detach()	void	Allows the thread to run independently from the others

Member function	Return value	Description
join()	void	Blocks waiting for the thread to complete
joinable()	bool	Check if the thread is joinable
hardware_concurrency0	unsigned	An hint on the HW thread contexts (often, number of CPU cores)
operator=		Move assignment

`C++11` defines also the namespace `std::this_thread` to group a set of functions to access the current thread.

Function	Return value	Description
yield()	void	Suspend the current thread, allowing other threads to be scheduled to run
sleep_for()	void	Sleep for a certain amount of time
sleep_until()	void	Sleep until a given timepoint

10.2.1 Mutex

A `mutex` is a common tool used for synchronizing access to data in order to ensure that only one thread can access it at a time. To use a mutex, you can utilize the member function:

Member function	Description
lock	Locks the mutex. If already locked, the thread blocks.
try_lock	Try to lock the mutex if not already locked
try_lock_for	Try to lock the timed_mutex for a specified duration
try_lock_until	Try to lock the timed_mutex until a time point
unlock	Unlock the mutex
release	Release ownership the mutex, without unlocking
operator=	Unlocks the owned mutex and acquire another

A key concept ts the **RAII (Resource Acquisition Is Initialization):** it ensures resources are acquired on initialization and released on destruction, applicable to all mutex types mentioned. Here more advanced mutex wrappers which follows the RAII pattern:

- **`lock_guard`**:
- Automatically acquires a mutex when constructed and releases it when the scope ends, ensuring minimal overhead.

```
#include <mutex>

std::mutex myMutex;

void safeFunction() {
    std::lock_guard<std::mutex> lock(myMutex); // The door is locked
    // Critical section: only you can access the shared resource here
} // The door automatically unlocks here when `lock` goes out of scope
```

- **`unique_lock`**:

- Supports deferred locking, transferring lock ownership, and manual lock/unlock operations.
- Has more overhead compared to `lock_guard` due to its additional features.

```
#include <mutex>

std::mutex myMutex;

void flexibleFunction() {

    {
    std::unique_lock<std::mutex> lock(myMutex);
    // Do some work...
    }

    //outside scope, so lock automatically unlocked


    {
    std::unique_lock<std::mutex> lock(myMutex);
    // Do some work...
    lock.unlock(); // You can manually unlock
    // Do some work without the lock...
    }

}
```

The additional `{` and `}` create a new scope, which is used here to define the critical section more precisely. By limiting the scope of the lock to just the necessary operations on the shared resource, the code reduces the likelihood of contention and ensures that other threads can access the shared resource sooner, improving the overall efficiency of the multithreaded program.

`scoped_lock` is similar to `lock_guard`, but capable of locking **multiple mutexes** at once.

- Helps avoid deadlocks by managing multiple locks simultaneously.
- Adheres to the RAII pattern, automatically releasing all acquired mutexes when the scope ends.

```
#include <mutex>

std::mutex mutex1, mutex2;

void multiLockFunction() {
    std::scoped_lock lock(mutex1, mutex2); // Both doors lock together
    // Critical section: safe access to resources protected by both mutex1 and mutex2
} // Both doors automatically unlock here
```

- `shared_lock` allows multiple threads to hold a "read" lock (shared ownership) simultaneously, but only one thread can hold a "write" lock (exclusive ownership) at a time.
- You use `unique_lock` when writing and `shared_lock` when reading with `shared_mutex`

10.2.2 Condition variables

Condition variables are used when a thread needs to wait (block) for a specific condition to become true. Often used in conjunction with a mutex to synchronize the access to a resource and wait for specific conditions on that resource.

Member function	Description
`wait(unique_lock<mutex> &)`	Blocks the thread until another thread wakes it up. The Lockable object is unlocked for the duration of the wait
`wait_for( unique_lock<mutex>&) const chrono::duration<..> t)`	Blocks the thread until another thread wakes it up, or a time span has passed.
`notify_one()`	Wake up one of the waiting threads.
`notify_all()`	Wake up all the waiting threads. If no thread is waiting do nothing.

Spurious wakeups occur when a thread wakes up from waiting, even without a proper signal. To handle spurious wakeups, it is recommended to use a `while` loop instead of an `if` statement. If we use an `if` statement, there is a chance that the thread may proceed even after a spurious wakeup, leading to incorrect behavior.

```
while (queue.size() < minFill)
    cv.wait(lock);
```

10.3 Design patterns for multithreaded programming

Now let's see most used design patterns in multithreaded programming context using

10.3.1 Producer/Consumer

The Producer-Consumer pattern is a classic example of a multi-threading design pattern used to coordinate work between these two types of threads.

A `SynchronizedQueue` is an implementation detail in this pattern: a thread-safe buffer between producers and consumers (it can be a FIFO data structure). Obviously to to protect the access to the queue, a Mutex (`std::mutex`) is used.

A condition Variable (`std::condition_variable`) is crucial for coordinating between producers and consumers without occupying the CPU.

10.3.2 Active Object

This pattern allows for threads to be treated as objects with their own methods, making it easier to manage. This approach is more organized and secure than using global variables for thread communication.

```
class ActiveObject {
public:
    ActiveObject() : quit(false) {
        // Constructor just starts the thread
        t = std::thread(&ActiveObject::run, this);
    }

    virtual ~ActiveObject() {
        // Destructor signals the thread to stop and waits for it to finish
        quit = true;
        if (t.joinable()) {
            t.join();
        }
    }
```

```
private:
    virtual void run() = 0; // Pure virtual function to be implemented by derived classes

protected:
    std::atomic<bool> quit; // Atomic flag to safely signal the thread to stop
    std::thread t; // The encapsulated thread
};
```

10.3.2.1 `std::atomic<bool>` `std::atomic<bool>` is often used to ensure thread-safe signaling:. You can initialize it just like a regular boolean and both use explicit assignments or `store` and `load` methods

```
quit = true; // or
quit.store(true);

bool shouldQuit = quit; //or
bool shouldQuit =  quit.load()
```

`store` and `load` are member functions provided by the `std::atomic` template class in C++, which are used to safely set and retrieve the value of an atomic variable in a multithreaded environment.

10.3.3 `store` Method:

The `store` method sets the value of the atomic variable. It ensures that the write operation is atomic, meaning it cannot be interrupted or divided into smaller operations that other threads can observe partway through. This atomicity guarantees that other threads always see the variable as either fully updated or not updated at all, with no intermediate states.

- **Syntax:** `atomic_variable.store(desired_value, memory_order);`
- **Example:** `quit.store(true);` sets the atomic boolean `quit` to `true`.

The `memory_order` argument is optional and specifies the memory ordering semantics for the operation. If omitted, it defaults to `memory_order_seq_cst`, which provides sequential consistency, the strongest memory ordering.

10.3.4 `load` Method:

The `load` method retrieves the current value of the atomic variable, ensuring that the read operation is atomic. This means that the value read is consistent and not a partial value that could result from simultaneous write operations in other threads.

- **Syntax:** `value = atomic_variable.load(memory_order);`
- **Example:** `bool shouldQuit = quit.load();` reads the value of `quit` into `shouldQuit`.

Like `store`, `load` also accepts an optional `memory_order` argument that dictates the memory ordering semantics of the read operation. If not specified, it defaults to `memory_order_seq_cst`.

10.3.5 Use in Multithreading:

`store` and `load` are essential for thread-safe operations on shared variables in concurrent code, preventing data races and ensuring visibility of changes across threads. They are particularly useful for flags and counters that are accessed and modified by multiple threads, such as loop control variables or state indicators.

For the `displayInit()` function in an embedded context, using a spinlock might not be necessary unless you're dealing with a multi-threaded environment where other threads might modify the LCD controller registers concurrently. In many embedded applications, especially in initialization routines, simple polling is sufficient and commonly used due to its simplicity and the single-threaded nature of most initialization sequences.

10.3.6 Reactor

The Reactor pattern allows for the decoupling of task creation and execution by packaging a function and its arguments to be called later. This pattern utilizes:

- **Active objects and synchronized queues:** These components are combined to manage tasks and ensure thread-safe communication.
- **Task queue:** the central place where tasks are stored, waiting to be executed by the reactor thread.
- **Execution order:** tasks are typically executed in a FIFO order.

10.3.7 ThreadPool Pattern Explained

While the Reactor pattern is suited for scenarios where tasks can be processed sequentially, the ThreadPool pattern optimizes performance employing multiple executor threads:

- **Multiple Worker Threads:** These threads concurrently pick up and execute tasks from one or more shared task queues.
- The **number of worker threads** (ideally the number of available CPU cores) and **task allocation** are design considerations.

Both the Reactor and ThreadPool patterns are foundational in concurrent programming, providing mechanisms to handle tasks efficiently in multi-threaded environments.