Exam Preparation and Summary of Topics
Advances Systems Programming S. 5 Bachelor WS21/22
h_da Darmstadt

Jonas Weßner

February 14, 2022

Contents

1 Introduction to Systems Programming **1**
1.1 What is Systems Software? . . . 1
1.2 Systems Programming Languages . . . 2

2 Memory Management and Memory Safety **2**
2.1 Types of Memory . . . 2
2.2 Locality . . . 3
2.3 Virtual Memory . . . 3
2.4 Return Value Optimization (RVO) . . . 4
2.5 Resource Acquisition is Initialization (RAII) . . . 4
2.6 Single Owner and Shared Owner Memory Management . . . 5
2.7 Heap Containers in Rust . . . 5

1 Introduction to Systems Programming

1.1 What is Systems Software?

Important aspects of systems software:

Systems software ...

- ... closely interacts with the hardware.
- ... is concerned about efficiency.
- ... is used by other software as opposed to application software which is used by the end-user directly.

Examples for systems software:

- Operating system
- Compiler
- Game engine
- Search engine
- Programming languages virtual machines e.g. java virtual machine
- Device drivers

Examples for application software:

- Text editor

- Shopping website
- Social media apps
- Chat client

1.2 Systems Programming Languages

Systems programming languages are languages which make systems programming easy. There are three properties we are especially interested in:

1. Direct access to hardware resources:
    - Memory management
    - Network throughput
    - GPU
    - CPU (single or multi core)
    - threads and processes
2. Performance, therefore mostly compiled languages
3. It would be nice to have some useful abstractions to improve productivity (C/C++ $\rightarrow$ Rust or Go).

Examples for systems programming languages:

- C, C++
- Rust
- Go
- Assembly (rarely)

Examples for application programming languages:

- JavaScript (disgusted tone of voice)
- Python
- Java

2 Memory Management and Memory Safety

2.1 Types of Memory

Mainly the following types of memory exist in a computer ordered from small space and fast access speed to large space and slow access speed:

- CPU registers
- Cache (L1, L2, L3)
- RAM
- SSD
- HDD
- Network Storage (Cloud, Backup)

2.2 Locality

There are two principles of locality which are used by caching algorithms:
Spatial locality assumes that after a certain portion of memory has been read, the next read memory addresses are likely to be at a nearby memory address. This is true because of how e.g. arrays are designed and because of the fact that each process has an address space of consecutive memory addresses. **Temporal locality** assumes that after a certain memory location has been read it is likely that it is going to be read again in a relatively short time span. This is true because of how logic computer programs usually works. If we read something from a location we often want to write something from that location and read it again shortly after.

2.3 Virtual Memory

There are two types of memory addresses: physical addresses and virtual addresses. Physical addresses are the addresses of bytes in RAM. Virtual memory addresses are addresses that are mapped to physical addresses. The virtual memory address space is much larger than the physical address space (RAM size) which allows us to use more memory than actually available. The mapping between virtual and physical memory is stored in a data structure called page table. The reason for using pages instead of a byte to byte mapping is that the page table would be way to large if we did so. In practice pages are e.g. 4KiB on unix systems.
Whenever the CPU executes a load or store instruction the virtual address first has to be resolved. in case the virtual address maps to a physical address in RAM the instruction can be executed right away. In case the virtual address is occupied but maps to a physical address on disc, which is called *page fault*, the following things must happen:

- An unused physical address must be found. If all physical addresses are occupied one page must be swaped out of RAM and stored on disc which is called *page eviction*.
- The contents of the desired virtual memory address must be copied to RAM and the page table must be updated.
- The CPU instruction must be run again and will now not trigger a page fault.

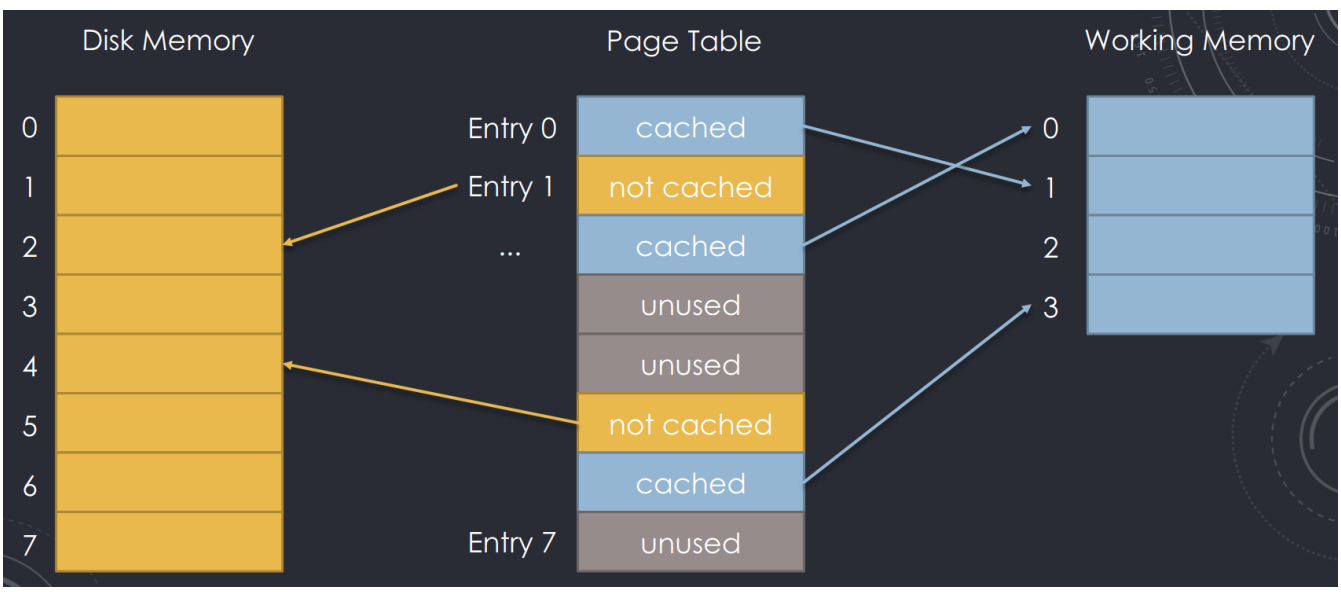

Figure 1: Visualization of virtual memory

A virtual address is split into two parts: the page number (yellow) and the offset within the page (orange). With an address like shown in figure 2 we can therefore address $2^4 = 16$ virtual pages of the size of $2^{12}bit = 2^2 \cdot 2^{10}bit = 4KiB$.

A page table entry as shown in figure 3 must then map the virtual page number (VPN) to a physical page number (PPN). As physical and virtual pages are of the same size the virtual page offset (VPO)

Figure 2: A single virtual address with page number in yellow and offset within page in orange

and the physical page offset (PPO) are identical and must not be mapped. When mapping a virtual to a physical address like shown in figure 4 therefore first the cached flag (yellow) is checked. If it is equal to zero a page fault has occurred and the physical address must first be loaded into RAM and then the page table must be updated. Now the physical address is build by replacing the virtual page number (VPN) with the physical page number (PPN) in the virtual address.

Figure 3: One page table entry with cached flag (yellow), virtual page number VPN (gray) and physical page number PPN (green)

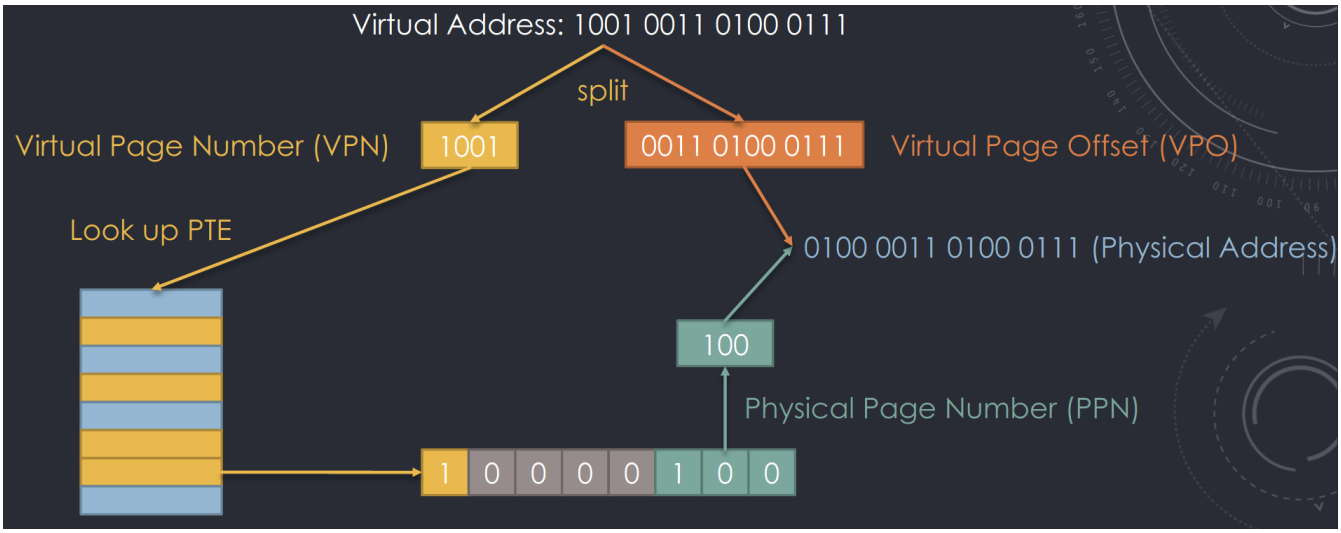

Figure 4: Visualization of the process of mapping a virtual address to a physical address

2.4 Return Value Optimization (RVO)

A common thing to do in C++ is to return a local variable from a function and assign it to a local variable of the calling function like shown in figure 5. In this example if we assume we return a stack variable from the function `add()`. Then if we resolve function calls we would end up with a local variable on the stack of the `add()` function wich is written to the register R0 as a return value. Then the return value is copied into the variable `result` by the `main()` function. To prevent this unnecessary copying of stack variables the compiler in C++ is allowed to apply *return value optimization (RVO)* and alter the normal behavior of stack cleanup to avoid the copying.

2.5 Resource Acquisition is Initialization (RAII)

C++ introduced a term called RAII which stands for *resource acquisition is initialization*. This refers to the behavior that if we use the `new`-operator to allocate memory on the heap, we also automatically call a constructor which will initialize that memory for us. Also when freeing that memory location with the `delete`-operator the destructor of the object is called to perform necessary cleanups. In pure C that was different as `malloc` and `free` do not initialize and cleanup their memory location. RAII is useful, can prevent frequent bugs and is of course also used in Rust.

```
int main(){
    int result = add(2, 2);
}

```

Figure 5: Use case for return value optimization in C++

2.6 Single Owner and Shared Owner Memory Management

In a single owner memory model like a `vector` the object allocates memory once it is created and cleans it up once it goes out of scope. However with a shared memory management an object allocates the memory only if it is the first instance being created for a specific memory segment and only cleans up if it is the last object pointing to that memory segment. This behavior allows multiple owners of heap memory and prevents *double free errors*. To be able to achieve this we need smart pointers with reference counting i.e. pointer types that check in their destructor if they are the last instance pointing to specific address.

2.7 Heap Containers in Rust

In figure 6 a short summary of containers and types is shown to work with the heap in rust. Furthermore I would like to give some more details to the individual containers here:

- `Rc<T>`: is a reference counting pointer. Its construction allocates memory on the heap. Using clone creates another instance of `Rc<T>` pointing to the same memory location. Once the last instance of `Rc<T>` pointing to that location goes out of scope it triggers the Drop trait for `T`.
- `Weak<T>`: is a pointer to a memory location which does not guarantee that memory location is still valid. It can be obtained via `Rc::downgrade` and be converted to an `Rc<T>` by calling `upgrade`. When a `Weak<T>` gets out of scope it never triggers the Drop Trait of `T`. Also when all `Rc<T>` pointing to one instance of T have been dropped `Weak<T>`s to that location will return `None` on `upgrade` as they point to freed memory locations. `Weak<T>` is useful to avoid circular dependencies between `Rc`s.
- `Cell<T>`: allows multiple owners of the same data. However, when retrieving that value the value will be copied. Therefore if we retrieve the value mutably inside the `Cell<T>` twice in the same scope, this is possible, but also results in two independent variables.
- `RefCell<T>`: allows multiple mutable owners of the same data. When retrieving a reference to the inner value it is checked that there is currently no other access to the inner value, otherwise the `borrow_mut` function panics. This way the borrow checking rules are enforced at runtime. This means we will never have two mutable references to the same data at the same time, because we otherwise panic. The programmer must now make sure he will not get a panic. This is not thread safe.
- `Arc<T>`: is an atomic reference counting pointer. It can be used across threads in order to have multiple threads accessing the same data. However it requires the wrapped value to be Sync, which means it must also be thread safe to access that. This can be ensured by wrapping the inner value inside of a `Mutex<T>` or a `RwLock<T>`.

Memory-Model	Rust Type(s)	C++ Type(s)	Low-level equivalent
Single value on heap/stack, borrowed	&T and &mut T	const T& and T&	T*
Single value on heap, single owner	Box<T>	std::unique_ptr<T>	T*
Single value on heap, multiple owners	Rc<T> and Weak<T>, potentially with Cell<T> or RefCell<T>	std::shared_ptr<T> and std::weak_ptr<T>	T*
Multiple adjacent values on heap/stack, borrowed	[T] in general and str for Strings	std::span<T> in general and std::basic_string_view<T> for Strings	T* + size_t length;
Multiple adjacent values on heap, single owner	Vec<T>	std::vector<T>	T* + size_t length;
Multiple adjacent values on heap, multiple owners[5]	Rc<Vec<T>>	std::shared_ptr<std::vector<T>>	T* + size_t length;

Figure 6: Use case for return value optimization in C++