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

Jonas Weßner

February 15, 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 **3**
2.1 Types of Memory . . . 3
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 . . . 4
2.7 Heap Containers in Rust . . . 5

3 Zero Overhead Abstractions **6**
3.1 Product Types and Sum Types . . . 6
3.2 Iterators . . . 6
3.3 Algorithms . . . 7

4 Error Handling **8**

5 Systems Level I/O **8**
5.1 Files . . . 8
5.2 Interprocess Communication . . . 8

6 Fearless Concurrency with Rust **9**
6.1 Ways of Parallelism . . . 9
6.2 The Fork Join Pattern . . . 9
6.3 The rayon parallelism crate . . . 9
6.4 Asynchronous I/O . . . 10

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
- Any kind of server

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.

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) 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.

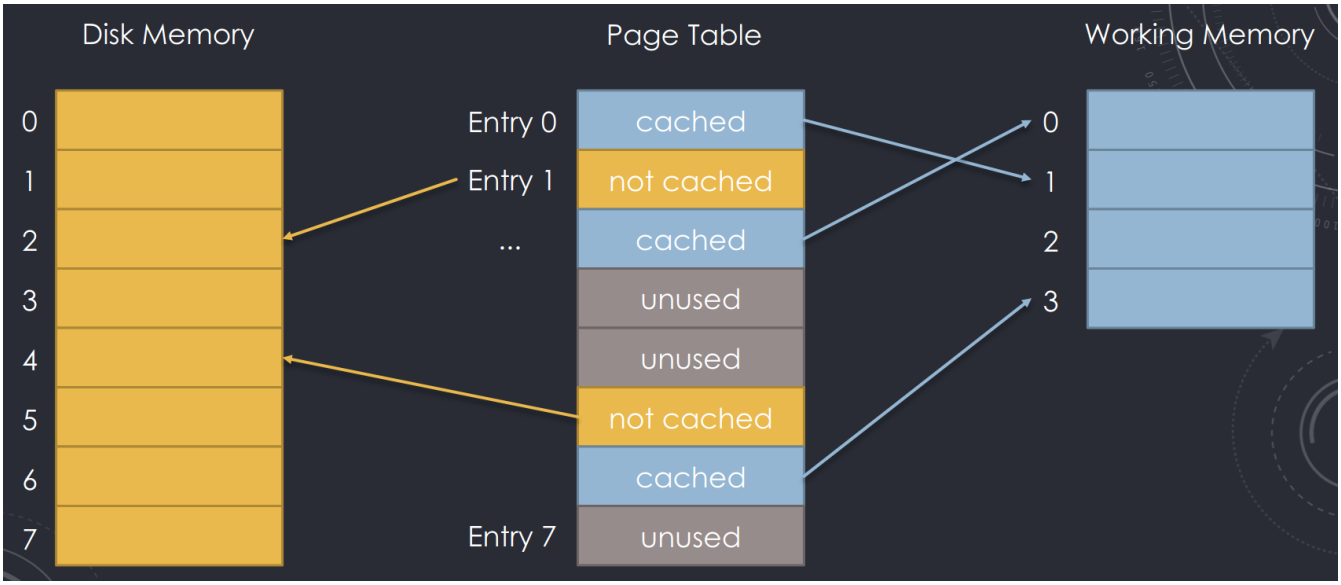

Figure 1: Visualization of virtual memory

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

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.

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.

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

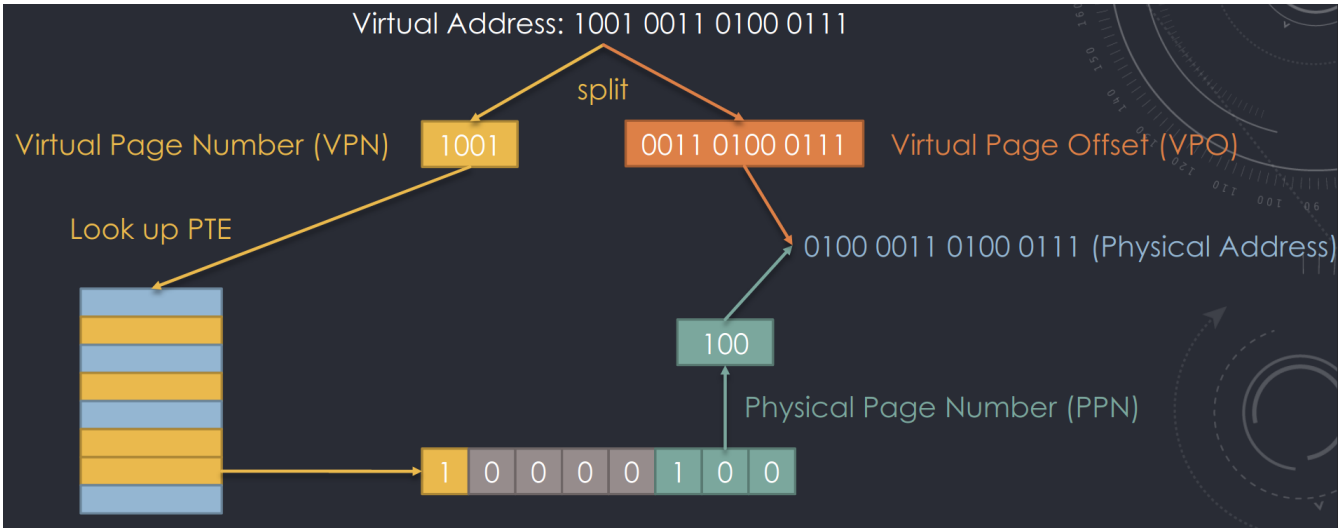

Figure 4: Visualization of the process of mapping a virtual address to a physical address

2.7 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>`.

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

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

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[2]	Rc<Vec<T>>	std::shared_ptr<std::vector<T>>	T* + size_t length;

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

3 Zero Overhead Abstractions

Zero Overhead abstractions are abstractions that have no performance overhead at runtime compared to the most optimal handwritten solution for the problem.

3.1 Product Types and Sum Types

Each primitive type has a finite set of possible values it can take on. However, we can define product types and sum types that build an abstraction on top of primitive types.
Product types are types whose set of possible values is of the size of the product of the amount of possible values of all contained types, an example are structs or classes.
Sum types are types whose size of the set of possible values is the sum of the amount of distinct values of all contained types, an example are rust enums.

3.2 Iterators

Iteration can be defined as a sequence of the following three operations:

- Dereferencing: Get access to current element
- Incrementing: Move to next element
- Comparing: Check if we are at the end of collection

In Rust all these three properties are usable with only one method which is called `next()`. It retrieves the current element if there is one, otherwise returns `None` and increments the iterator.

In contrast to C++, rust uses stateful iterators, which means that the iterator knows its current position inside a collection and will return `None` once it has reached the end.

3.3 Algorithms

In the following we will summarize some of the useful algorithms that are supplied for working with rust iterators:

1. `all`: returns true if all elements meet a certain condition
2. `any`: returns true if at least one element meets a certain condition
3. `is_partitioned`: returns true if there is a boundary in the container such that all elements before the boundary are either true or false and after the boundary the oposite.
4. `filter`: Returns an iterator over all elements that meet a given condition.
5. `filter_map`: returns an iterator over all elements of Type U for which a filter closure returns Some(U)
6. `position`: returns the Some(index) of the first element matching a condition
7. `find`: returns Some(T) for the first element of type T matching a condition.
8. `collect`: returns a collection storing all elements over which has been iterated
9. `count`: Returns the number of elements over which is being iterated.
10. `for_each`: calls a function without return value taking each element as an input.
11. `max`: returns the maximum element
12. `min`: returns the minimum element
13. `fold`: takes a start value of Type U and a closure with 2 parameters T and U with return value U which will be called for each element. Returns a value of type U which is the result of calling the closure with the accumulated value and the current value of the iterator for each element.
14. `reduce`: Combines two elements of type T to one repeating this for each element and returning the resulting scalar value T.
15. `sum`: fold with start value 0 and an addition as closure.
16. `product`: fold with start value 1 and multiplication as closure.
17. `step(n)`: returns an element over every n-th element
18. `peekable`: returns a peekable iterator which is an iterator which can call .peek() which means returning the current element without incrementing the position.
19. `map`: converts each element of type T into an element of type U by calling a closure.
20. `flatten`: converts an iterator over iterators (nested iterator) into an iterator over elements of the inner type (like flattening a 2d-array to a 1d array).
21. `intersperse(val)`: returns an iterator over the elements separated by the given val.
22. `cycle`: returns a never ending iterator that returns the elements of the given iterator in a cycle.
23. `chain(other)`: concats iterators
24. `enumerate`: returns an iterator over tuples of (index, element).
25. `fuse`: returns an iterator that will always return None after having returned None once on calling `next`.
26. `rev`: returns an iterator over the elements in reversed order
27. `zip(other)`: returns an iterator over tuples of the elements of both iterators.

Note that rust iterators are online iterators, which means they work with one piece of data at a time and therefore do not support sorting, however collections in rust often have sort methods.

4 Error Handling

There are three common ways of error handling:

- Error Codes like in C:
  - + easy and not complicated
  - - abuse of return type makes certain values unusable for non-error cases
  - - easy to ignore errors returned by a function
  - - error codes have no names and must be transated using documentation
  - - specific error data must be read from global variables
- Exceptions like in C++:
  - + convenient for user as the function call is separated from the catch statement
  - - not transparent which function throws what errors
- Result Type like in Rust:
  - + error must be handled explicitly
  - + errors can deliver error details to the caller
  - + it is clear to the caller what error can be returned by e.g. using an enum as error type inside a result type.

5 Systems Level I/O

5.1 Files

A file is a sequence of bytes. It can be stored persistently on disc or also just in memory. Unix was build on the idea that the following operations are everything needed to work with files:

- Opening
- Closing
- Changing position inside a file
- Reading a range of bytes from current position
- Writing a range of bytes to current position

In unix all types of input and output are modeled using files such as:

- Reading from and writing to disk
- Reading from and writing to network streams
- Reading from and writing to the terminal

5.2 Interprocess Communication

Two types of communication:

1. Process control: with Signals
   - Signals inform processes about low level system events having happened and are handles using a signal handler
   - Signals can be send using the *kill* command
   - signals are basically just an integer encoding an event, they have no payload -¿ not suitable for information exchange

2. Information exchange: with I/0 or shared Memory
    - We can share some pages of memory between processes with the *mmap* function in C.
    - Shared memory fast but difficult and most of the time inconvenient
    - Standard way is using I/O such as TCP streams

Command Line Arguments

In rust we can access the command line arguments using `st::env::args()` which returns an iterator over all cli arguments.

Environment Variables

Environment variables are key value pairs that can be set on the operating system for one process. There are some predefined variables. When using `int execve(const char *filename, char *const argv[], char *const envp[]);` to spawn a new process, the environment variables of the parent process are passed to the child process via the third argument to it's main function `int main(int argc, char *argv[], char *envp[])`

6 Fearless Concurrency with Rust

6.1 Ways of Parallelism

1. Instruction Level Parallelism: Pipelining
2. Hyperthreading: Duplicating certain parts like the ALU inside a single processor to run multiple instructions at once. This is possible because one instruction may only need some parts of the CPU and another instruction may need other parts of the CPU.
3. SIMD: perform same operation on multiple words -¿ data level parallelism
4. Using multiple Cores with multithreading or multiple processes.

Threads in Rust

Via the interface `std::thread::spawn(FnOnce + Send) -> JoinsHandle<T>` we can start a new thread executing a closure which will return a join handle. Join handles are used to join or detach threads later on. All variables captured in the closure must be owned by it i.e. must be moved into the closure using `std::thread::spawn(move || ...`.

6.2 The Fork Join Pattern

The fork join pattern says that a task can be parallelized by first splitting the task into n disjunct sub tasks by the master thread, then spawning $n-1$ worker threads and executing the tasks and eventually merging the result into one single result by the master thread again.

6.3 The rayon parallelism crate

Rayon provides parallel iterators which are very easy to use and can parallelize tasks on big containers:

```
1    fn search_parallel_rayon(text: &str, keyword: &str, parallelism: usize) -> usize {
2    let chunks = chunk_string_at_whitespace(text, parallelism);
3    chunks
4        .par_iter()
5        .map(|chunk| search_sequential(chunk, keyword))
6        .sum()
7    }
```

6.4 Asynchronous I/O

Threads are great for saturating the CPU with a lot of work. But if we have many small tasks then it might be better to use asynchronous programming. The challenge that asynchronous code comes to solve is how to handle multiple streams of tasks at the same time when using one thread per task is not suitable because most of the times the threads would wait for tasks. I.e. how to handle multiple connections where most of the time on each connection is spend waiting for a message. This comes eventually down to the idea that we observe a pool of actions often called Futures with only one thread and want to make sure we always handle the first Future that is ready and execute the corresponding code. This can be implemented with interfaces like `poll` in C, however, is not very convenient. Asynchronous patterns in rust with async and await add a layer of abstraction on top of that. We can define that a function is async which means it returns a Future which can be awaited. Because async functions can only be called inside of other async functions that comes down to the main function being async. Then we need some kind of executor which is essentially a runtime to execute our async main function. Executors are supplied by libraries like tokio. The executor then calls await on the outer most future which will call poll on all futures until one of the polls returns Poll::Pending. This call also schedules a waker which will notify the executor when the future has new data and shall be polled again.