Passwortwiederherstellung

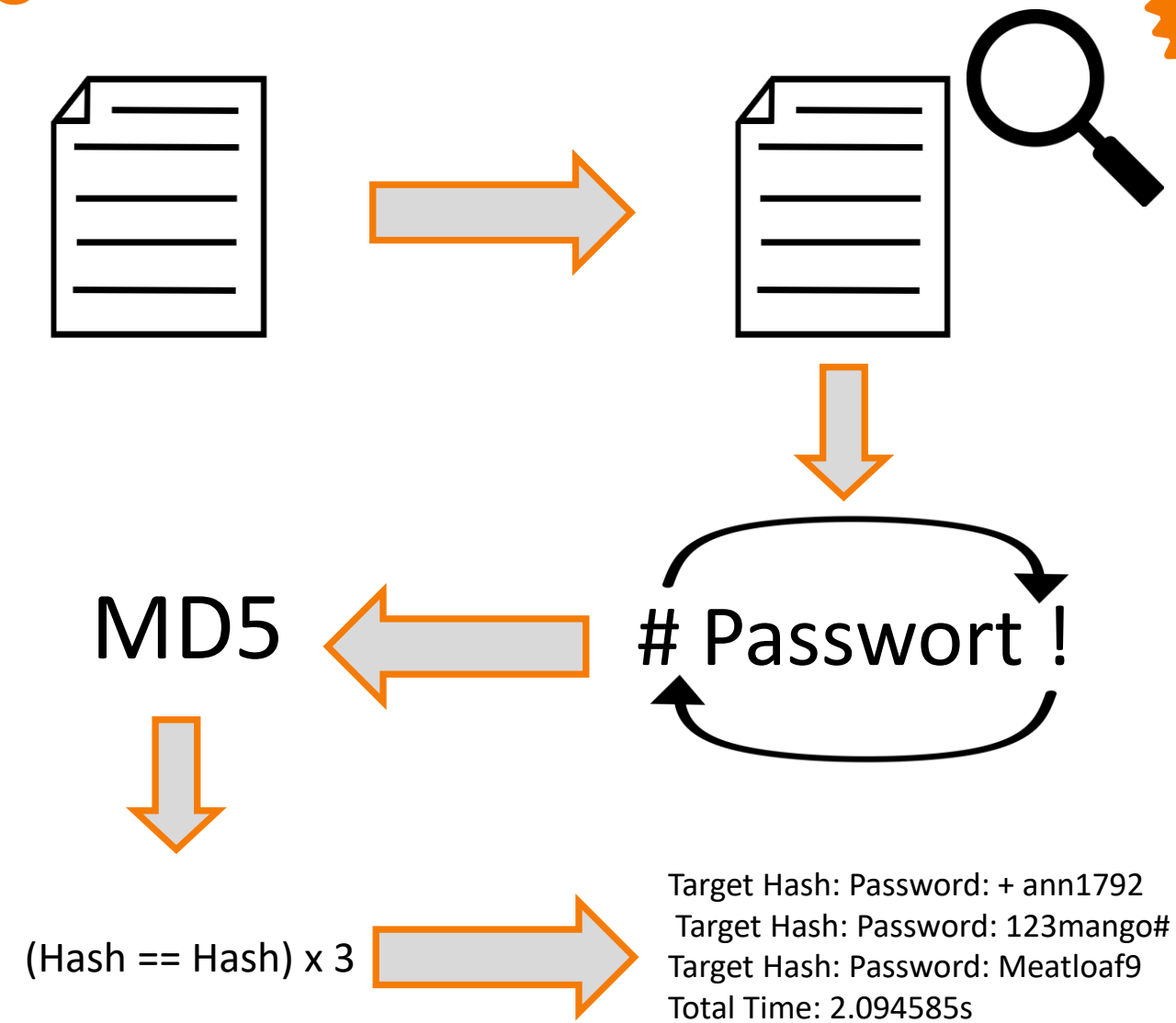
Luis Heider

Inhaltsverzeichnis

1. Demo
2. Codeübersicht
3. Unsafe Rust
4. Nightly Rust
5. Memory Mapping
6. Parallelisierung mit Rayon
7. MD5 Hashing
8. Performance

1. Demo
2. Codeübersicht
3. Unsafe Rust
4. Nightly Rust
5. Memory Mapping
6. Parallelisierung mit Rayon
7. MD5 Hashing
8. Performance

1. Demo



1. Demo
2. Codeübersicht
3. Unsafe Rust
4. Nightly Rust
5. Memory Mapping
6. Parallelisierung mit Rayon
7. MD5 Hashing
8. Performance

2. Codeübersicht

```
struct FoundResult {
    candidate: [u8; MAX_LEN + 1], // Staatisch Allokierter Puffer
    len: usize,                   // Länge des Strings
    duration: std::time::Duration,
}
```

```
const TARGET_DIGESTS: [u128; 3] = [
    0x32c5c26e20908ebd80269d32f51cb5bb,
    0x648d5d9cc7cafe536fdbc6331f00c6a0,
    0xd31daf6579548a2a1bf5a9bd57b5bb89,
];
```

1. Demo
2. Codeübersicht
3. Unsafe Rust
4. Nightly Rust
5. Memory Mapping
6. Parallelisierung mit Rayon
7. MD5 Hashing
8. Performance

2. Codeübersicht

```
30  unsafe fn store_variant(...)
56  unsafe fn check_digest_variant(...)
85  unsafe fn compute_suffix_hash(...)
93  unsafe fn compute_prefix_hash(...)
102 fn main()
```

1. Demo
2. Codeübersicht
3. Unsafe Rust
4. Nightly Rust
5. Memory Mapping
6. Parallelisierung mit Rayon
7. MD5 Hashing
8. Performance

3. Unsafe Rust

- Keine Compilersicherheit

Ownership

Borrowing und Mutability

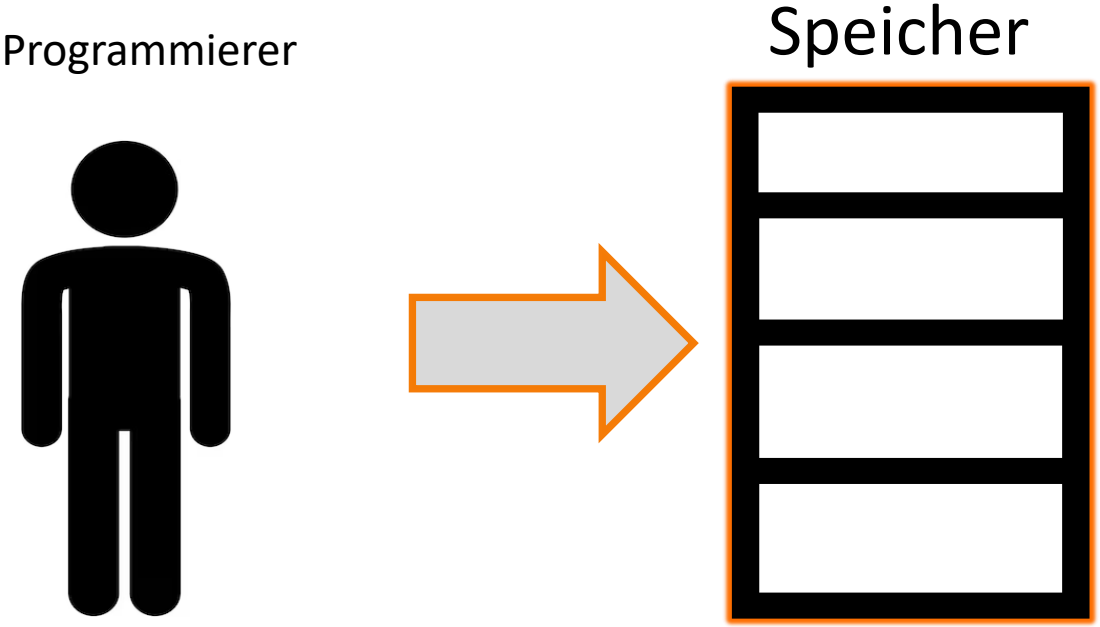
Lifetime

Borrow Checker

1. Demo
2. Codeübersicht
3. Unsafe Rust
4. Nightly Rust
5. Memory Mapping
6. Parallelisierung mit Rayon
7. MD5 Hashing
8. Performance

3. Unsafe Rust

- Keine Compilersicherheit
- Direkter Speicherzugriff




```
ptr::copy_nonoverlapping(  candidate.as_ptr(),
                           res_box.candidate.as_mut_ptr(),
                           candidate.len());
```

1. Demo
2. Codeübersicht
3. Unsafe Rust
4. Nightly Rust
5. Memory Mapping
6. Parallelisierung mit Rayon
7. MD5 Hashing
8. Performance

3. Unsafe Rust

- Keine Compilersicherheit
- Direkter Speicherzugriff
- Intrinsics

```
1  #![feature(core_intrinsics)]
```

```
55 #[inline(always)]
```

```
64  if likely(digest == TARGET_DIGESTS[0]) {
65      if store_variant(candidate, start, found_count, &results[0]) {
66          hit = true;
67      }
68  }
```

1. Demo
2. Codeübersicht
3. Unsafe Rust
4. Nightly Rust
5. Memory Mapping
6. Parallelisierung mit Rayon
7. MD5 Hashing
8. Performance

3. Unsafe Rust

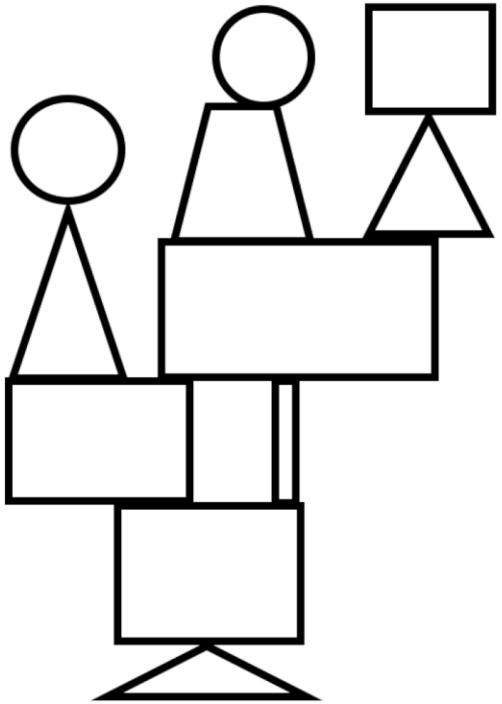
- Keine Compilersicherheit
- Direkter Speicherzugriff
- Intrinsics
- Wieso direkte Speicherzugriffe?

  - **Effizienz**: Keine Sicherheitscheck beschleunigen den Kopiervorgang
  - **Flexibilität**: Fehlende Sicherheitschecks sorgen für mehr Kontrolle
  - **Atomare** Operationen: Bei Parallelität sorgt es für korrekte Synchronisation

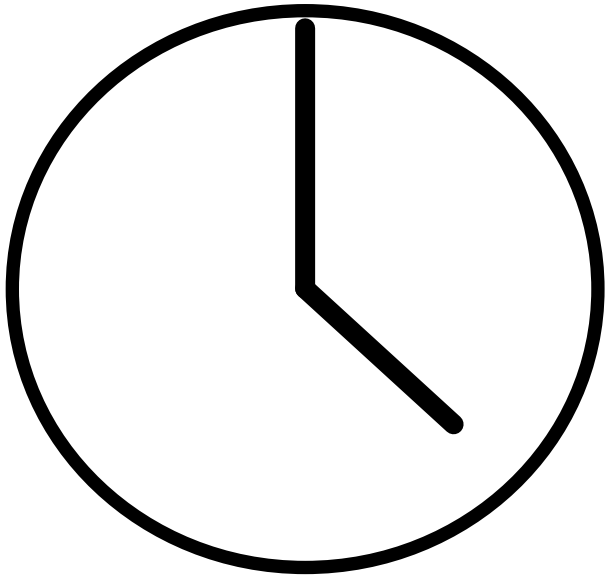
1. Demo
2. Codeübersicht
3. Unsafe Rust
4. Nightly Rust
5. Memory Mapping
6. Parallelisierung mit Rayon
7. MD5 Hashing
8. Performance

4. Nightly Rust

- Was ist Nightly Rust?



Experimentelle Features



Tägliche Updates

1. Demo
2. Codeübersicht
3. Unsafe Rust
4. Nightly Rust
5. Memory Mapping
6. Parallelisierung mit Rayon
7. MD5 Hashing
8. Performance

4. Nightly Rust

- Was ist Nightly Rust?
- Warum hier Nightly?

```
#![feature(core_intrinsics)]
#![allow(internal_features)]
```

```
if likely(digest == TARGET_DIGESTS[0]) {
    if store_variant(candidate, start, found_count, &results[0]) {
        hit = true;
    }
}
```

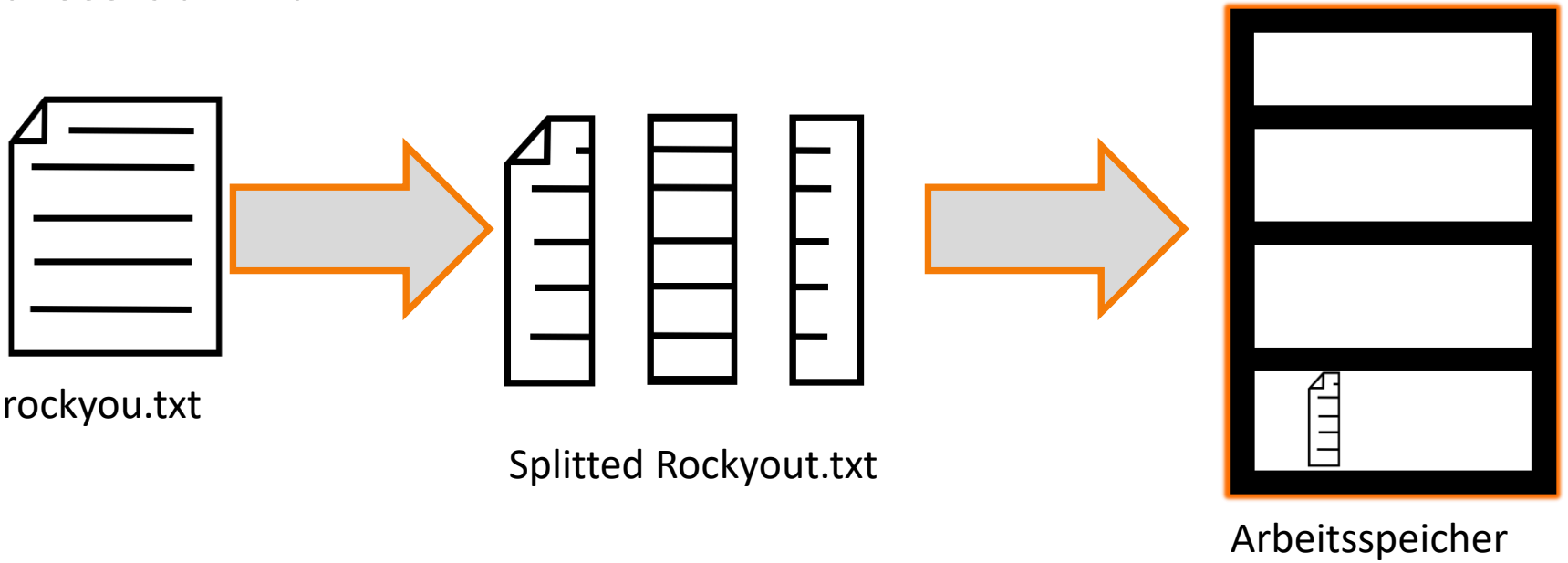
1. Demo
2. Codeübersicht
3. Unsafe Rust
4. Nightly Rust
5. Memory Mapping
6. Parallelisierung mit Rayon
7. MD5 Hashing
8. Performance

5. Memory Mapping

OS teilt Bereich im virtuellen Adressraum zu

Datei wird vom OS in „Pages“ unterteilt

Nur geladene Pages werden in den Arbeitsspeicher geladen



1. Demo
2. Codeübersicht
3. Unsafe Rust
4. Nightly Rust
5. Memory Mapping
6. Parallelisierung mit Rayon
7. MD5 Hashing
8. Performance

5. Memory Mapping

Warum Memory Mapping?

- **On-Demand Laden:** Nur benötigte Pages werden geladen
- **Optimierung durch das BS:** Das OS verwaltet das Page loading effizient
- **Weniger Kopieraufwand**: Kein explizites Kopieren in einen separaten Puffer

1. Demo
2. Codeübersicht
3. Unsafe Rust
4. Nightly Rust
5. Memory Mapping
6. Parallelisierung mit Rayon
7. MD5 Hashing
8. Performance

5. Memory Mapping

```
5 use memmap2::Mmap;
```

```
111 let file = File::open("rockyou.txt")?;
112 let mmap = unsafe { Mmap::map(&file)? };
113 let data = mmap.as_ref();
114 let lines: Vec<&[u8]> = data.split(|&b| b == b'\n').collect();
```

Unsafe: Os Systemressourcen werden Abgerufen
-> Keine Rust Sicherheit

1. Demo
2. Codeübersicht
3. Unsafe Rust
4. Nightly Rust
5. Memory Mapping
6. Parallelisierung mit Rayon
7. MD5 Hashing
8. Performance

6. Parallelisierung mit Rayon

Rayon:

- **Einfache Parallelisierung**: Iterator macht Parallelisierung einfach
- **Thread Pool**: Nutzt alle verfügbaren CPU
- **Work Stealing**: Bei beendung der Aufgabe übernimmt der Thread die Aufgabe des anderen

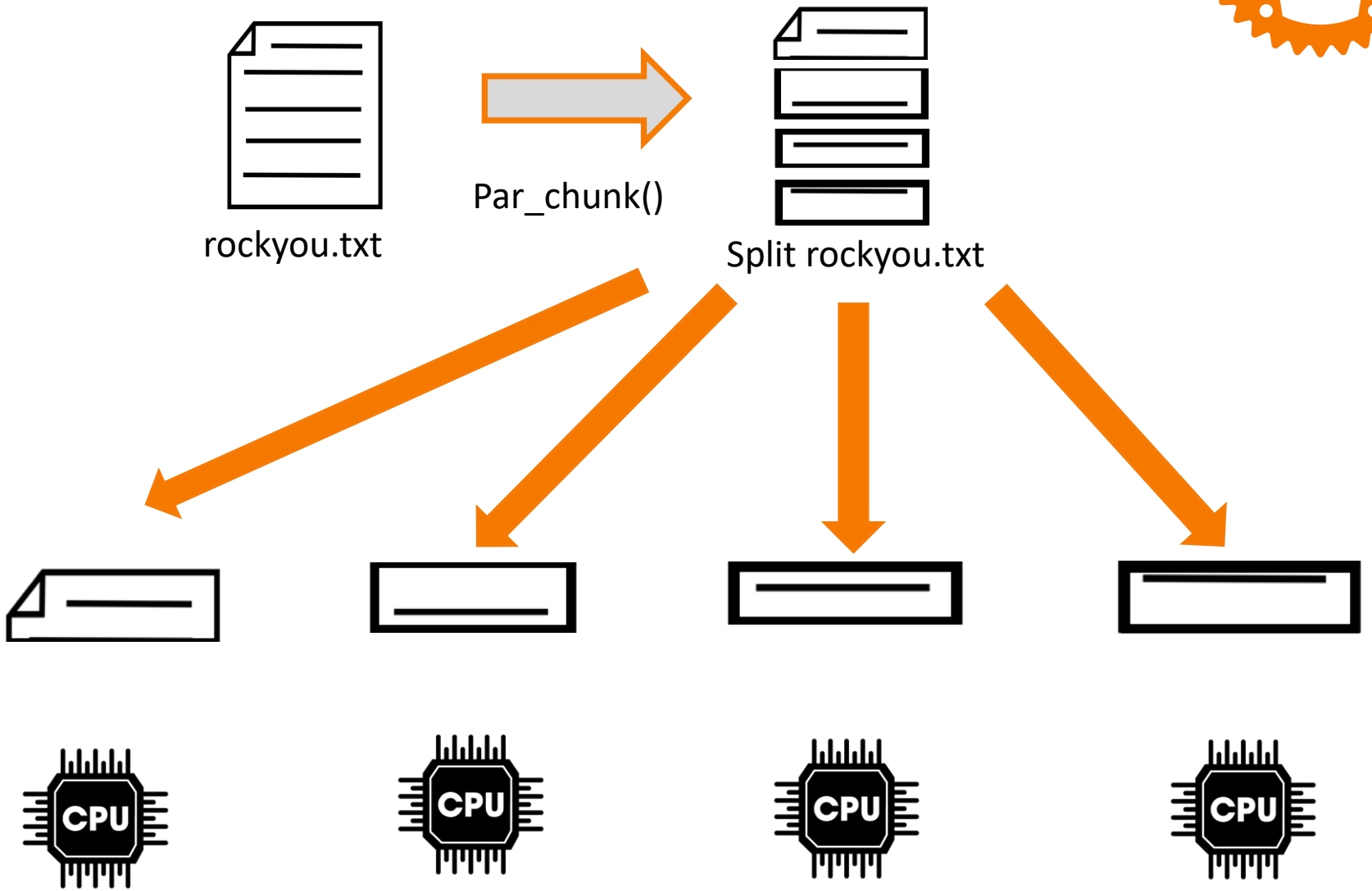
Code:

```
9   use rayon::prelude::*;
```

```
121 lines.par_chunks(30000).for_each(|chunk|{...
```

1. Demo
2. Codeübersicht
3. Unsafe Rust
4. Nightly Rust
5. Memory Mapping
6. Parallelisierung mit Rayon
7. MD5 Hashing
8. Performance

6. Parallelisierung mit Rayon



1. Demo
2. Codeübersicht
3. Unsafe Rust
4. Nightly Rust
5. Memory Mapping
6. Parallelisierung mit Rayon
7. MD5 Hashing
8. Performance

6. Parallelisierung mit Rayon

Beenden des Prozesses:

```
121 lines.par_chunks(30000).for_each(|chunk| {
122     for line in chunk {
123         if unlikely(found_count.load(Ordering::Relaxed) >= num_targets) {
124             break;
125         }
```

```
44 if slot.compare_exchange(ptr::null_mut(), res_ptr, Ordering::SeqCst,
Ordering::SeqCst).is_ok() {
45     found_count.fetch_add(1, Ordering::Relaxed);
```

1. Demo
2. Codeübersicht
3. Unsafe Rust
4. Nightly Rust
5. Memory Mapping
6. Parallelisierung mit Rayon
7. MD5 Hashing
8. Performance

7. MD5 Hashing

MD5-Hasher Methoden:

- .update(inhalt) -> Aktualisiert den Hasher mit dem Inhalt
- .finalize_reset() -> Beendet den Datenstrom und gibt den Hashwert zurück
- .into() -> Gibt den Hashwert in einem anderen Datentyp zurück

1. Demo
2. Codeübersicht
3. Unsafe Rust
4. Nightly Rust
5. Memory Mapping
6. Parallelisierung mit Rayon
7. MD5 Hashing
8. Performance

7. MD5 Hashing

```rust
use md5::{Md5, Digest};
```

```rust
const TARGET_DIGESTS: [u128; 3] = [
    0x32c5c26e20908ebd80269d32f51cb5bb,
    0x648d5d9cc7cafe536fdbc6331f00c6a0,
    0xd31daf6579548a2a1bf5a9bd57b5bb89,
];
```

```rust
let mut hasher = Md5::new();
hasher.update(candidate_slice);
let orig_digest_arr: [u8; 16] = hasher.finalize_reset().into();
let orig_digest = u128::from_be_bytes(orig_digest_arr);
```

1. Demo
2. Codeübersicht
3. Unsafe Rust
4. Nightly Rust
5. Memory Mapping
6. Parallelisierung mit Rayon
7. MD5 Hashing
8. Performance

7. MD5 Hashing

```
if likely(orig_digest == TARGET_DIGESTS[0]
    || orig_digest == TARGET_DIGESTS[1]
    || orig_digest == TARGET_DIGESTS[2])
{
    let _ = store_variant(candidate_slice, start,
&found_count, &results[0]);
    continue;
}
```

1. Demo
2. Codeübersicht
3. Unsafe Rust
4. Nightly Rust
5. Memory Mapping
6. Parallelisierung mit Rayon
7. MD5 Hashing
8. Performance

7. MD5 Hashing

```
115 let affixes: &[u8] = b"!#+0123456789";
```

!Passwort, #Passwort, +Passwort …

Passwort!, Passwort#, Passwort+ …

```
142 let mut variant_buf = [0u8; MAX_LEN + 1];
143 variant_buf[..candidate_slice.len()].copy_from_slice(candidate_slice);
144 let mut base_hasher = Md5::new();
145 base_hasher.update(candidate_slice);
```

1. Demo
2. Codeübersicht
3. Unsafe Rust
4. Nightly Rust
5. Memory Mapping
6. Parallelisierung mit Rayon
7. MD5 Hashing
8. Performance

7. MD5 Hashing

```
for &aff in affixes {
    if unlikely(found_count.load(Ordering::Relaxed) >= num_targets) {
        break;
    }
    variant_buf[candidate_slice.len()] = aff;
    let variant_slice = &variant_buf[..candidate_slice.len() + 1];
    let digest_u128 = compute_suffix_hash(&base_hasher, aff);
```

7. MD5 Hashing

1. Demo
2. Codeübersicht
3. Unsafe Rust
4. Nightly Rust
5. Memory Mapping
6. Parallelisierung mit Rayon
7. MD5 Hashing
8. Performance

```rust
unsafe fn compute_suffix_hash(base_hasher: &Md5, aff: u8) -> u128 {
    let mut h = base_hasher.clone();
    h.update(&[aff]);
    let d_arr: [u8; 16] = h.finalize_reset().into();
    u128::from_be_bytes(d_arr)
}
```

7. MD5 Hashing

1. Demo
2. Codeübersicht
3. Unsafe Rust
4. Nightly Rust
5. Memory Mapping
6. Parallelisierung mit Rayon
7. MD5 Hashing
8. Performance

```
153 if likely(digest_u128 == TARGET_DIGESTS[0]
154     || digest_u128 == TARGET_DIGESTS[1]
155     || digest_u128 == TARGET_DIGESTS[2])
156 {
157     let _ = check_digest_variant(digest_u128, variant_slice, start,
&found_count, &results);
158 }
```

1. Demo
2. Codeübersicht
3. Unsafe Rust
4. Nightly Rust
5. Memory Mapping
6. Parallelisierung mit Rayon
7. MD5 Hashing
8. Performance

8. Performance

Änderung	Gesparte Zeit	Zeit nach Implementierung
Erste Version	-	72-76 Sekunden
Vergleich von Binärwerten	~ 21 Sekunden	51-55 Sekunden
Aufteilung in Chunks	~ 17 Sekunden	34-38 Sekunden
Dopplungen verhindern	~ 8 Sekunden	26-30 Sekunden
Mapping der Textdatei & direkter Speicherzugriff	~ 16 Sekunden	8-12 Sekunden
Intrinsics und MD5 Optimierung	~ 5 Sekunden	4-7 Sekunden

Finaler Wert: 2.1-5.5 Sekunden

Speichernutzung: 360-380 mB

Schnellst gefundenes Wort in 299.654 ms