Soukarya Ghosh (5447593)
Jacob Olsson (6545270)
CESE4015 Software Systems
Assignment # 1 : Concurrency

1) Arc vs Rc in std

Both Rc and Arc are Reference counted. This means that they count the references to the memory, and keep it in memory as long as there are at least one reference to the memory. The difference between them is simply that Arc is atomic, while Rc is not, making Arc thread safe, and Rc must stay within a single thread.

However, the simple approach of just counting references doesn't work that well, since it allows for loops of references, eg. object one point to object two, and object two point to object one. In the simple case of just count the references, neither can ever not be referenced, so it will "forever" stay in memory, and never be freed. This is why there is a strong count, and a weak count. A strong count is a reference that owns the value, and can easily access it. The weak counter does not own the value, but it knows where it is, and can potentially get a value. If a weak reference whats to use the value, it may run upgrade. This will return the value if it still exists, and None otherwise.

When you create a Rc or Arc, there is automatically one strong pointer to the Rc. When you clone the Rc, you increment the strong count. You can downgrade this to a weak reference. That will decrement the strong count and increment the weak count. This will stop memory leaks and reference cycles from forming, since we will not allow two strong pointers to point to each other. When a reference is dropped, it decrements the counter, meaning a strong reference decrements the strong counter and the weak decrements the weak. When the strong counter is 0, the value in the Rc is dropped, freeing the memory [1], but the pointer still exists if the weak counter is at least 1. When both counters are 0, the Rc is completely dropped, and all references become invalid, and the memory is freed. In the case of a parent pointing to a child with a strong reference, and the child pointing to the parent with a weak reference, the moment the last strong reference to the parent is dropped, the value of the parent is also dropped, but the reference is still valid since the weak count is still 1. The upgrade of a weak reference will return None, since the value is dropped. Since the value of the parent is dropped, the strong reference to the child is also dropped, decrementing the strong count of the child. If the strong count is now 0, it will also drop the value, and the weak pointer to the parent, decrementing its weak count [2]. Now both counters of the parent is 0, so the Rc is fully dropped and all memory is freed.

Now the difference between Rc and Arc is that Arc is thread safe. Firstly, Arc makes the Rc atomic, so it prevents two threads to change the count at the same time. But Arc also makes sure that the value is also safe to use between threads. Arc¡T¿ will inherit send and sync, if T is send and sync. Send means that it is safe to move it to another thread. Sync means that it is safe to move a reference of it to another thread (T is sync if &T is send) [3][p. 16]. This makes sure that all values shared between threads will not allow data races. All primitive types, like i32, bool and str are both send and sync. But sharing these between threads with Arc, they are not mutable. If you need to change the value, you need to make sure it is safe to change it without data races. To do this you can wrap it in something like a mutex or a Rwlock. This is send and sync as well, and it is now safe to share and change the value between threads.

2) **Mutex vs RwLock in std**

Both Mutex and RwLock are primitives in Rust that are used to allow safe sharing of data between threads, but each primitive is optimized for different concurrency patterns.

A `Mutex<T>` is a lock that gives access to only one thread at a time. No other thread can enter the critical section while the mutex is locked. A `RwLock<T>` (read–write lock), on the other hand, allows multiple threads to access the data at the same time, but only one thread may write, and writing cannot happen while readers are active. Because of this, a mutex is simpler, while a read–write lock can be faster when the data is read often and written occasionally [3].

A mutex is required most when updates to the data happen frequently or when the protected code is small. A read–write lock is useful when reads happen much more often than writes, since many readers can proceed in parallel without blocking one another.

A simplified structure of `Mutex<T>` looks like:

```
pub struct Mutex<T> {
    locked: AtomicBool,
    data: UnsafeCell<T>,
}
```

In the simplified model above, the field `locked` is an atomic flag showing whether the lock is currently held, and when a thread uses the `lock()` function, it attempts to set this flag. If another thread already holds it, the caller waits. Once the lock is acquired, a `MutexGuard<T>` is returned and you are given access to the data [3]. And lastly, the data being wrapped in `UnsafeCell<T>` also enables interior mutability.

An `RwLock` keeps track of how many readers currently hold the lock, whether a writer holds the lock, and which threads are waiting. A simplified state representation is:

```
struct RwLockState {
    state: AtomicUsize,
    data: UnsafeCell<T>
}
```

An RwLock allows multiple readers or one writer. From the the simplified model above, the field `state` writes down both the reader count and a writer flag, using bits inside the `AtomicUsize`. When a reader acquires the lock, the reader count is incremented. A writer must wait until reader count is zero, and then the writer bit is set to gain access to data [3]. And lastly, the data being wrapped in `UnsafeCell<T>` also enables interior mutability.

Rust normally requires a unique reference (`&mut T`) to change data. However, both `Mutex<T>` and `RwLock<T>` allow modification through a shared reference. This is possible because these types use the concept of **interior mutability**. Interior mutability means that a value can be changed even when only a shared reference exists [4]. This is implemented using `UnsafeCell<T>`, which is the only type in Rust that allows mutable access to its contents through a shared reference.

1 Benchmarking and Performance Comparison: Theory

We evaluated the performance of our `mygrep` implementation against GNU `grep` using the recursive search pattern `[mM]icrodevice` on a large directory. All tests used `/usr/bin/time`, capturing **real time**, which is the wall-clocl time, **user time**, which captured CPU time in user-space, and **system time**, which captures CPU time in kernel-space.

To evaluate performance, the **real,user, and sys** times of our `mygrep` implementation and the `grep` implementation were compared. For each implementation, the **real,user, and sys** times were read 10 times, and then averaged at the end using the following benchmark scripts; average values were computed.

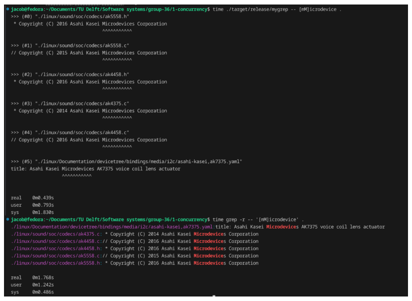
mygrep Benchmark Script:

```
#!/usr/bin/env bash
runs=10; sum_real=0; sum_user=0; sum_sys=0
for i in $(seq 1 $runs); do
  t=$( { /usr/bin/time -f "%e %U %S" \
    ./target/release/mygrep '[mM]icrodevice' . \
    >/dev/null; } 2>&1 )
  read real user sys <<< "$t"
  echo "Run $i: real=$real user=$user sys=$sys"
  sum_real=$(echo "$sum_real + $real" | bc -l)
  sum_user=$(echo "$sum_user + $user" | bc -l)
  sum_sys=$(echo "$sum_sys + $sys" | bc -l)
done
echo "Average: real=$(echo "$sum_real/$runs" | bc -l)"
```

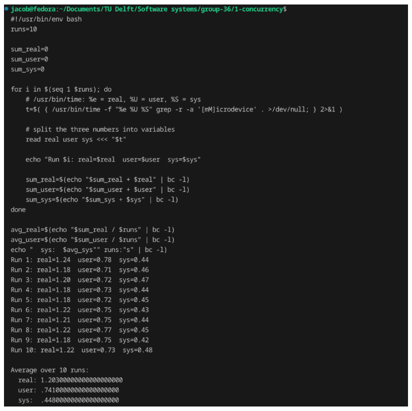
GNU grep Benchmark Script:

```
#!/usr/bin/env bash
runs=10; sum_real=0; sum_user=0; sum_sys=0
for i in $(seq 1 $runs); do
  t=$( { /usr/bin/time -f "%e %U %S" \
    grep -r -a '[mM]icrodevice' . >/dev/null; } 2>&1 )
  read real user sys <<< "$t"
  echo "Run $i: real=$real user=$user sys=$sys"
  sum_real=$(echo "$sum_real + $real" | bc -l)
  sum_user=$(echo "$sum_user + $user" | bc -l)
  sum_sys=$(echo "$sum_sys + $sys" | bc -l)
done
echo "Average: real=$(echo "$sum_real/$runs" | bc -l)"
```

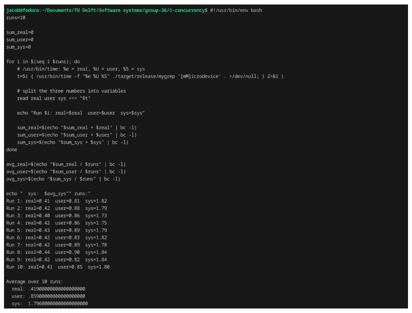
2 Benchmarking and Performance Comparison: Results



```
jacob@fedora:~/Documents/TU Delft/Software systems/group-36/1-concurrency$ time ./target/release/mygrep -- [mM]icrodevice .
>>> (#0) "./linux/sound/soc/codecs/ak5558.h"
 * Copyright (C) 2016 Asahi Kasei Microdevices Corporation
                                 ^^^^^^^^^^^

>>> (#1) "./linux/sound/soc/codecs/ak5558.c"
// Copyright (C) 2015 Asahi Kasei Microdevices Corporation
                                 ^^^^^^^^^^^

>>> (#2) "./linux/sound/soc/codecs/ak4458.h"
 * Copyright (C) 2016 Asahi Kasei Microdevices Corporation
                                 ^^^^^^^^^^^

>>> (#3) "./linux/sound/soc/codecs/ak4375.c"
 * Copyright (C) 2014 Asahi Kasei Microdevices Corporation
                                 ^^^^^^^^^^^

>>> (#4) "./linux/sound/soc/codecs/ak4458.c"
// Copyright (C) 2016 Asahi Kasei Microdevices Corporation
                                 ^^^^^^^^^^^

>>> (#5) "./linux/Documentation/devicetree/bindings/media/i2c/asahi-kasei,ak7375.yaml"
title: Asahi Kasei Microdevices AK7375 voice coil lens actuator
                   ^^^^^^^^^^^


real    0m0.439s
user    0m0.793s
sys     0m1.830s
jacob@fedora:~/Documents/TU Delft/Software systems/group-36/1-concurrency$ time grep -r -- '[mM]icrodevice' .
./linux/Documentation/devicetree/bindings/media/i2c/asahi-kasei,ak7375.yaml:title: Asahi Kasei Microdevices AK7375 voice coil lens actuator
./linux/sound/soc/codecs/ak4375.c: * Copyright (C) 2014 Asahi Kasei Microdevices Corporation
./linux/sound/soc/codecs/ak4458.c:// Copyright (C) 2016 Asahi Kasei Microdevices Corporation
./linux/sound/soc/codecs/ak4458.h: * Copyright (C) 2016 Asahi Kasei Microdevices Corporation
./linux/sound/soc/codecs/ak5558.c:// Copyright (C) 2015 Asahi Kasei Microdevices Corporation
./linux/sound/soc/codecs/ak5558.h: * Copyright (C) 2016 Asahi Kasei Microdevices Corporation

real    0m1.768s
user    0m1.242s
sys     0m0.486s
```



```
jacob@fedora:~/Documents/TU Delft/Software systems/group-36/1-concurrency$
#!/usr/bin/env bash
runs=10

sum_real=0
sum_user=0
sum_sys=0

for i in $(seq 1 $runs); do
    # /usr/bin/time: %e = real, %U = user, %S = sys
    t=$( { /usr/bin/time -f "%e %U %S" grep -r -a '[mM]icrodevice' . >/dev/null; } 2>&1 )

    # split the three numbers into variables
    read real user sys <<< "$t"

    echo "Run $i: real=$real  user=$user  sys=$sys"

    sum_real=$(echo "$sum_real + $real" | bc -l)
    sum_user=$(echo "$sum_user + $user" | bc -l)
    sum_sys=$(echo "$sum_sys + $sys" | bc -l)
done

avg_real=$(echo "$sum_real / $runs" | bc -l)
avg_user=$(echo "$sum_user / $runs" | bc -l)
echo "  sys:  $avg_sys"" runs:"s" | bc -l)
Run 1: real=1.24  user=0.78  sys=0.44
Run 2: real=1.18  user=0.71  sys=0.46
Run 3: real=1.20  user=0.72  sys=0.47
Run 4: real=1.18  user=0.73  sys=0.44
Run 5: real=1.18  user=0.72  sys=0.45
Run 6: real=1.22  user=0.75  sys=0.43
Run 7: real=1.21  user=0.75  sys=0.44
Run 8: real=1.22  user=0.77  sys=0.45
Run 9: real=1.18  user=0.75  sys=0.42
Run 10: real=1.22  user=0.73  sys=0.48

Average over 10 runs:
  real: 1.20300000000000000000
  user: .74100000000000000000
  sys:  .44800000000000000000
```



```
jacob@fedora:~/Documents/TU Delft/Software systems/group-36/1-concurrency$ #!/usr/bin/env bash
runs=10

sum_real=0
sum_user=0
sum_sys=0

for i in $(seq 1 $runs); do
    # /usr/bin/time: %e = real, %U = user, %S = sys
    t=$( { /usr/bin/time -f "%e %U %S" ./target/release/mygrep '[mM]icrodevice' . >/dev/null; } 2>&1 )

    # split the three numbers into variables
    read real user sys <<< "$t"

    echo "Run $i: real=$real  user=$user  sys=$sys"

    sum_real=$(echo "$sum_real + $real" | bc -l)
    sum_user=$(echo "$sum_user + $user" | bc -l)
    sum_sys=$(echo "$sum_sys + $sys" | bc -l)
done

avg_real=$(echo "$sum_real / $runs" | bc -l)
avg_user=$(echo "$sum_user / $runs" | bc -l)
avg_sys=$(echo "$sum_sys / $runs" | bc -l)

echo "  sys:  $avg_sys"" runs:"
Run 1: real=0.41  user=0.81  sys=1.82
Run 2: real=0.42  user=0.88  sys=1.79
Run 3: real=0.40  user=0.86  sys=1.73
Run 4: real=0.42  user=0.86  sys=1.75
Run 5: real=0.43  user=0.89  sys=1.79
Run 6: real=0.42  user=0.83  sys=1.82
Run 7: real=0.42  user=0.89  sys=1.78
Run 8: real=0.44  user=0.90  sys=1.84
Run 9: real=0.42  user=0.82  sys=1.84
Run 10: real=0.41  user=0.85  sys=1.80

Average over 10 runs:
  real: .41900000000000000000
  user: .85900000000000000000
  sys:  1.79600000000000000000
```

Figure 1: Comparison of `grep` and `mygrep` performances and timing statistics.

The figure to the left shows our mygrep implementation working successfully, and also compares the execution time of our concurrent mygrep implementation against GNU grep. The figures to the middle and right show the averaged statistics of the 10 runs, showing the 10-run averaged user time, real time, and system time for the GNU grep and the mygrep respectively. As shown in the figures, our mygrep implementation ran almost three times faster than the GNU grep (in terms of real time). However, our mygrep implementation's system time was about 4 times slower than the GNU grep (in terms of sys time). This is because of the parallelism structure that our mygrep uses. Real time is reduced by parallelism, but the system time increases as the kernel must manage that parallelism.

References

[1] The Rust Project Developers, *Cloning an rc increases the reference count*, https://doc.rust-lang.org/book/ch15-04-rc.html, 2025.

[2] The Rust Project Developers, *Preventing reference cycles: Turning an rc¡t¿ into a weak¡t¿*, https://doc.rust-lang.org/book/ch15-06-reference-cycles.html, 2025.

[3] M. Bos, *Rust Atomics and Locks: Low-Level Concurrency in Practice*. Cambridge, MA: O'Reilly Media, 2023, ISBN: 978-1098119447.

[4] loudsilence. "Understanding mutex and rwlock in rust. " [Online]. Available: https://loudsilence.medium.com/understanding-mutex-and-rwlock-in-rust-55974cc163c0.