System Performance Engineering CW2

- Yifan Wu(yw5920)

1 Synchronization primitives

1.1 Execution order(1)

```
// all orderings:
1122
1212
1221
2112
2121
2211
```

1.2 Execution order(2)

```
// all orderings
1122
2211
```

1.3 User-level implementation

```
mysem::mysem(uint32_t init_value) {
    counter.store(init_value, std::memory_order_seq_cst);
}

void mysem::acquire() {
    while (true) {
        uint32_t v = counter.load(std::memory_order_seq_cst);
        if (v > 0) {
            if (counter.compare_exchange_weak(
                    v,
                    v - 1,
                    std::memory_order_seq_cst))
                break;
            // else: thread was scheduled out between
            // 2 atomic ops, so we should try again
        }
    }
}

void mysem::release() {
    counter.fetch_add(1, std::memory_order_seq_cst);
}
```

1.4 Hybrid implementation

```
#include <linux/futex.h>
#include <sys/syscall.h>
#include <unistd.h>

void mysem::acquire() {
    uint32_t v;
    uint32_t* counter_ptr = reinterpret_cast<uint32_t*>(&counter);
    while (true) {
        v = 0;
        for (int i = 0; i < 100; ++i) {
            if ((v = counter.load(std::memory_order_seq_cst) > 0))
                break;
        }
        if (v == 0)
            syscall(
                SYS_futex,
                counter_ptr,
                FUTEX_WAIT,
                0, 0, 0, 0
            );
        for (int i = 0; i < 100; ++i) {
            if ((v = counter.load(std::memory_order_seq_cst)) > 0) {
                if (counter.compare_exchange_weak(
                        v,
                        v - 1,
                        std::memory_order_seq_cst))
                    return;
            }
        }
    }
}

void mysem::release() {
    uint32_t* counter_ptr = reinterpret_cast<uint32_t*>(&counter);
    counter.fetch_add(1, std::memory_order_seq_cst);
    syscall(
        SYS_futex,
        counter_ptr,
        FUTEX_WAKE,
        1, 0, 0, 0);
}
```

1.5 A better hybrid implementation

We can maintain another `std::atomic<uint32_t>` which is increased by one atomically before calling `FUTEX_WAIT` and decreased by one atomically after `FUTEX_WAIT` returns. This object indicates the number of threads which are trapping in the futex at the moment. Therefore, in `mysem::release`, we can call `FUTEX_WAKE` only if the value of the object is greater than 0.

2 Networked Service Design

2.1 Network modeling

On average, in order to handle one request, the NIC receives $(1+512)\times 0.9+(1+512+1024)\times 0.1=615.4\text{Bytes}$ and transmits $1024\times 0.9+1\times 0.1=921.7\text{Bytes}$, from which we can see that the transmitting end is the bottleneck.

The bandwidth of the transmitting end is $10\text{Gbps}=1.25\times 10^9\text{Bytes/s}$.

Therefore, the maximum throughput is $\frac{1.25\times 10^9\text{Bytes/s}}{921.7\text{Bytes/req}}=1356189.65\text{req/s}$.

2.2 CPU modeling

On average, each CPU takes $200\times 0.9+500\times 0.1=230\mu\text{sec}$ to handle one request, which means that one CPU can handle $\frac{1\text{sec}}{230\mu\text{sec/req}}=4347.83\text{req/sec}$.

Therefore, at least $\lceil\frac{130000\text{req/sec}}{4347.83\text{req/sec}}\rceil=30$ CPUs are needed.

2.3 Cutting corners

- 1000req/sec, 100% GETS->average latency: 200μsec

  It is the same as the ideal lock-free implementation. This is because the gap between 2 requests is $1\text{ms}=1000\mu\text{sec}$, which is greater than $200\mu\text{sec}$, which we takes to handle one GET request. This means that we always have enough time to handle each request, so the latency is also ideal.

- 100000req/sec, 100% GETS->average latency:207μsec

  This is slightly larger than the lock-free implementation. In this case, a new request arrives every $10\mu\text{sec}$, but we need $200\mu\text{sec}$ to handle it, which means that we handle 20 requests in parallel at a time. Thus, it is possible that there are 2 requests falling in the same bucket at a time, if that happens, there will be contention between them since they all need to update the counter inside the `std::shared_mutex`, which brings a larger overhead than the lock-free implementation.

- 100000req/sec, 50% GETS, 50% SETS->average latency:475μsec

It is worse than the lock-free implementation, which could achieve a latency at $350\mu\text{sec}$ if GETS and SETS are equally distributed. The reason is that it is possible that 2 GETS and SETS accessing the same bucket can be in flight at the same time. In this case, the later one has to wait for the earlier one to be completed before it starts, leading to the result that its latency is even larger than the latency when there are only GETS requests.