Citadel *2021 May17, Gasper Azman*

Question 1 - What is Epoll?
How does kernel implement epoll so that it can achieve O(1) complexity? What data structure is needed inside epoll for storing the file descriptors? Please read *"The method to epoll's madness"* by Cindy Sridharan.

Question 2 - Posting list intersection (Modification on merge sort)
Given a map (dataset) with string as key (just a name of identifier), sorted integer sequence as value

- the string is short (says 8 characters)
- the number of integer sequence can be large
- the integer sequence can be long (16 million ~ 2^{24})
- the integer belongs to range [0 - 2^{43}]
- the dataset is immutable once setup, hence precalculation is allowed

Given a query, which is just a subset of the names (subset size is about 10), find the intersection of the 10 selected integer sequences in the most efficient way.

Solution
Since slow precalculation is allowed, vector is used to store the integer sequences, which is more cache friendly than list. There are approaches : pairwise merge or all-in-one batch merge. Pairwise merge is easier to implement, more readable and less error prone (recall the pattern for implementation of `std::merge`).

```
template<typename IN_ITER0, typename IN_ITER1, typename OUT_ITER>
void pair_merge(IN_ITER0 begin0,  IN_ITER0 end0, IN_ITER1 begin1, IN_ITER1 end1, OUT_ITER begin)
{
	static_assert(std::is_same< typename std::iterator_traits<IN_ITER0>::value_type,
	                            typename std::iterator_traits<IN_ITER1>::value_type>::value,
	                            "IN_ITER0::value_type / IN_ITER1::value_type are not consistent");

	IN_ITER0 iter0 = begin0;
	IN_ITER1 iter1 = begin1;
	OUT_ITER iter  = begin;
	while(iter0!=end0 && iter1!=end1)
	{
		if (*iter0 < *iter1) ++iter0;
		else if (*iter0 > *iter1) ++iter1;
		else
		{
			iter = *iter0;
			++iter0;
			++iter1;
			++iter;
		}
	}
}

template<typename IN_ITER, typename OUT_ITER>
void K_merge_pairwise(const std::vector<std::pair<IN_ITER, IN_ITER>>& sequences, OUT_ITER begin)
{
	// *** Deep of first sequence *** //
	auto seq = sequences.begin();
	std::vector<typename std::iterator_traits<IN_ITER>::value_type> in(seq->first, seq->second);

	// *** Merge with next sequence *** //
	for(++seq; seq != sequences.end(); ++seq)
	{
		std::vector<typename std::iterator_traits<IN_ITER>::value_type> out;
		pair_merge(in.begin(), in.end(), seq->first, seq->second, std::back_inserter(out));
		in = std::move(out);
	}

	// *** Final output *** //
	for(const auto& x:in)
	{
		*begin = x;
		++begin;
	}
}
```

However interviewer required to implement in all-in-one batch way. This is error-prone and not readable. I failed to find out how the loops should be implemented and resulting in infinity loop. Besides he tried to make things cumbersome by introducing coroutine. He also gave a hint that we should keep check of `std::make_pair(v.begin(), v.end())` for each vector as there is no `zip` in C++ (which seems to be a misleading hint, and confuses my thought).

Here is a workable version that I did after the interview, please note :

- the following two increments are different
  `++seq0` vs `++(seq0.first)`
  `++seq1` vs `++(seq1.first)`
- the function should quit whenever one sequence reaches the end

```
template<typename IN_ITER, typename OUT_ITER>
void K_merge_allinone(const std::vector<std::pair<IN_ITER, IN_ITER>>& sequences, OUT_ITER begin)
{
	auto seqs = sequences;

	while(true) // loop all elements in sequences
	{
		bool match_exists = true;
		auto seq0 = seqs.begin();
		auto seq1 = seqs.begin();
		++seq1;

		while(seq1!=seqs.end()) // loop all sequences
		{
			if (*seq0->first < *seq1->first)
			{
				// break and redo : with seq0 = first sequence
				++(seqs.begin()->first);
				if (seqs.begin()->first == seqs.begin()->second) return;
				match_exists = false;
				break;
			}
			else if (*seq0->first > *seq1->first)
			{
				// continue : with same seq0 and seq1
				++(seq1->first);
				if (seq1->first == seq1->second) return;
			}
			else
			{
				// continue : with next seq0 and seq1
				++seq0;
				++seq1;
			}
		}

		if (match_exists)
		{
			*begin = *(seqs.begin()->first);
			++begin;

			// increment each sequence iterator
			for(auto& seq:seqs)
			{
				++(seq.first);
				if (seq.first == seq.second) return;
			}
		}
	}
}
```

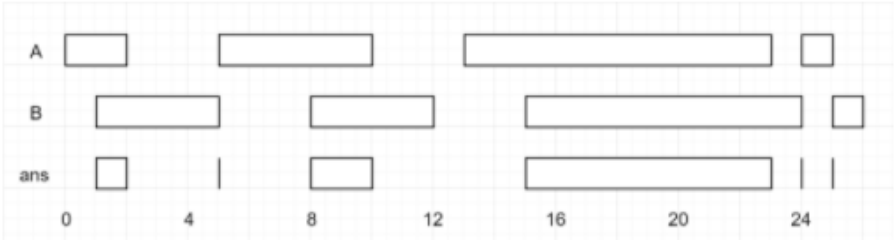
Interviewer argued that this is still not fast enough. As the bottleneck is memory read bandwidth (CPUs are idle waiting for data to process, hence concurrency or SIMD doesn't help. In order to speed up further, we need integer compression. Typical integer compression is ***bit-packing*** (like what I did for `CDOI::HSI` market data).

- compression of sequence of integer
= serialization
= composing and parsing

- optimum compression (minimum size) is not the most efficient compression (fastest compose and parse)

Question 3 - Interval list intersection (modification on merge sort)

This question is from Leetcode. An interval `[a,b]` is defined as a set of consecutive integers including `a` and `b`. 2 sorted list of disjoint intervals are given as `list0=A` and `list1=B`, find their intersection.




```
class interval_list_intersection
{
public:
    std::vector<std::pair<int,int>> solve(std::vector<std::pair<int,int>>& list0, std::vector<std::pair<int,int>>& list1)
    {
        std::vector<std::pair<int,int>> output;
        auto iter0 = list0.begin();
        auto iter1 = list1.begin();
        while(iter0!=list0.end() && iter1!=list1.end())
        {
            if (is_strictly_less(*iter0,*iter1))
            {
                ++iter0;
            }
            else if (is_strictly_less(*iter1,*iter0))
            {
                ++iter1;
            }
            else
            {
                auto new_interval = overlap(*iter0,*iter1);
                output.push_back(new_interval);

                if (iter0->second == new_interval.second) ++iter0;
                if (iter1->second == new_interval.second) ++iter1;
            }
        }
        return output;
    }

private:
    bool is_strictly_less(const std::pair<int,int>& interval0, const std::pair<int,int>& interval1)
    {
        if (interval0.second < interval1.first) return true;
        return false;
    }

    std::pair<int,int> overlap(const std::pair<int,int>& interval0, const std::pair<int,int>& interval1)
    {
        std::pair<int,int> output;
        output.first = std::max(interval0.first, interval1.first);
        output.second = std::min(interval0.second, interval1.second);
        return output;
    }
};
```

Question 4 - Interval list union (modification on merge sort)

This question is from Leetcode. Union is more difficult than intersection, as it may merge multiple intervals into a single interval. To simplify the question, we consider the special case such that : `list1.size()==any` while `list1.size()==1`. Given a sorted-and-disjoint interval list, plus one interval, find the union.

```
class interval_list_union
{
public:
    std::vector<std::pair<int,int>> solve(std::vector<std::pair<int,int>>& intervals, std::pair<int,int>& new_interval)
    {
        bool new_interval_added = false;
        auto iter = intervals.begin();
        while(iter!=intervals.end() && !new_interval_added)
        {
            if (is_less(*iter, new_interval))
            {
                output_push_back(*iter);
                ++iter;
            }
            else
            {
                output_push_back(new_interval);
                new_interval_added = true;
            }
        }

        while(iter!=intervals.end())
        {
            output_push_back(*iter);
            ++iter;
        }
        if (!new_interval_added)
        {
            output_push_back(new_interval);
            new_interval_added = true;
        }
        return output;
    }

private:
    bool is_less(const std::pair<int,int>& interval0, const std::pair<int,int>& interval1)
    {
        if (interval0.first < interval1.first) return true;
        return false;
    }

    void output_push_back(const std::pair<int,int>& new_interval)
    {
        if (output.size()==0)
        {
            output.push_back(new_interval);
        }
        else if (output.back().second < new_interval.first)
        {
            output.push_back(new_interval);
        }
        else if (output.back().second < new_interval.second)
        {
            output.back().second = new_interval.second;
        }
    }

    std::vector<std::pair<int,int>> output;
};
```