

Excercise Sheet 6

Solution

Lecture Parallel Processing

Winter Term 2025/26

Exercise 1: Lock-Free Data Structures

```
class Entry { // One entry in the event trace buffer
public:
    int valid; // Indicates that the entry is written completely
    double time; // time stamp in us
    int thread; // thread ID
    int iteration; // the current iteration in the thread
    int status; // some status of computation
    Entry(double time, int thread, int iteration, int status) {
        this->valid = 0;
        this->time = time;
        this->thread = thread;
        this->iteration = iteration;
        this->status = status;
    }
    Entry() : Entry(0, 0, 0, 0) {}
};

class EventBuffer {
private:
    std::vector<Entry> buffer; // event buffer
    int read_index; // index of the event to be returned by the next call to pop_front()
    int write_index; // index of the next 'free' entry in the buffer
public:
    EventBuffer(int size) : buffer(size) {
        read_index = 0; // if read_index == write_index, the buffer is empty
        write_index = 0;
    }
    void push_back(int thread, int iteration, int status) {
        auto now = std::chrono::high_resolution_clock::now();
        double time = (now - starttime) / std::chrono::nanoseconds(1) / 1000.0;
        int wr = __sync_fetch_and_add(&write_index, 1); // Reserve buffer element
        buffer[wr] = Entry(time, thread, iteration, status); // Write entry into buffer
        buffer[wr].valid = 1; // Mark entry as completely written
    }
    Entry pop_front() {
        // For simplicity, we do busy waiting when the buffer is empty.
        // We also have to wait until the entry is completely written.
        while ((read_index == write_index) || !buffer[read_index].valid);
        return buffer[read_index++];
    }
};
```

Exercise 2: Sokoban with Lock-Free Data Structures

```
bool BFSQueue::lookup_and_add(uint64_t conf, uint64_t predIndex, uint64_t box)
```

```

{
uint64_t bitmask = (uint64_t)1 << bsBitPos(conf);
uint64_t i1 = bsIndex1(conf);
uint64_t i2 = bsIndex2(conf);

if (bitset[i1] == NULL) {
    // Several threads may allocate a new block
    uint64_t * p = new uint64_t[BLOCKSIZE] ();
    // However, only for one of them, the compare_and_swap succeeds
    if (!__sync_bool_compare_and_swap(&bitset[i1], NULL, p))
        delete[] p;
}

// Atomically fetch and set the bit in the bitmask
if ((__sync_fetch_and_or(&bitset[i1][i2], bitmask) & bitmask) != 0)
    return false;

// Reserve the next element in the write queue
uint64_t wrPosOld = __sync_fetch_and_add(&wrPos, 1);
uint64_t wr = depth % 2;
uint64_t n1 = qIndex1(wrPosOld);
uint64_t n2 = qIndex2(wrPosOld);

if (queue[wr][n1] == NULL) {
    // Several threads may allocate a new block
    Entry * p = new Entry[BLOCKSIZE] ();
    // However, only for one of them, the compare_and_swap succeeds
    if (!__sync_bool_compare_and_swap(&queue[wr][n1], NULL, p))
        delete[] p;
}

queue[wr][n1][n2].set(conf, wrPosOld + (rdLength - predIndex), box);

return true;
}

bool DFSDepthMap::lookup_and_set(uint64_t conf, uint64_t newDepth)
{
uint64_t i1 = index1(conf);
uint64_t i2 = index2(conf);

if (depth[i1] == NULL) {
    // Several threads may allocate a new block
    unsigned char * p = new unsigned char[BLOCKSIZE] ();
    // However, only for one of them, the compare_and_swap succeeds
    if (!__sync_bool_compare_and_swap(&depth[i1], NULL, p))
        delete[] p;
}

unsigned char old;
bool success;
do {
    old = depth[i1][i2];
    if ((old != 0) && (old <= (unsigned char)newDepth))
        return false;
    // We must only store into depth[i1][i2], if newDepth is still smaller than depth[i1][i2].
    // Otherwise, some other thread updated depth[i1][i2], before we got the chance.
    success = __sync_bool_compare_and_swap(&depth[i1][i2], old,
        (unsigned char)newDepth);
} while (!success); // If another thread has updated depth[i1][i2], repeat the operation.
if (old != 0)
    __sync_fetch_and_add(&nConfigs[old], -1); // Atomic decrement
}

```

```
    __sync_fetch_and_add(&nConfigs[newDepth], 1); // Atomic increment  
    return true;  
}
```