Assignment 1: COL818

SHIVAM VERMA (2020CS50442)

Implement the universality of consensus algorithm in C++

1. Implement the lock-free and wait-free variants.
2. Write your own consensus object using the compare and swap (exchange) instruction (available in both x86 and ARM). Use the __asm directive to implement the consensus object.
3. Implement a concurrent stack and queue using the consensus object that you created.
4. Show the correctness of your algorithm.

   The submission is for machines with ARM architecture.

Files in directory

1. `LF-universalConsensus.cpp`: Lock Free variant of universal consensus object.
2. `WF-universalConsensus.cpp`: Wait Free variant of universal consensus object.
3. `Makefile`: To compile the files
4. `checker.py`: To test the correctness of execution log. To see whether property of stack and queue are maintained?
5. `execution_log.txt`: Final execution log is printed in this file, after all threads have completed their job.
6. `run.sh`: Shell script to run the task
7. `Readme`: You are currently reading this!
8. `test-consensus.cpp`: Created to test the consensus object.

How to Run?

1. In `Makefile` either use either `SRC = LF-universalConsensus.cpp` or `SRC = WF-universalConsensus.cpp`
2. Set number of threads in the `run.sh` file.
3. `chmod +x run.sh`
4. Run `./run.sh`
5. Verification result will be printed in terminal and logs can be seen in `execution_log.txt` file.
6. Additional `CHECK` flag can be set to 1 in main file, to check which all threads are competing for a particular sequence number concurrently.

REPORT

The `LFUniversal` (and `WFUniversal` respectively) are the main objects in the code. It contains a linked list of the nodes and a pointer to the very first node of this linked list.

The `apply` function of this object takes an `invocation` and `tid` as argument and tries to add a new node to the end of this linkedlist with this invocation using the consensus game.

Each `node` object contains a user called invocation function, sequence number and the pointer to the next node.

Consensus Function

The consensus function is implemented using a CAS protocol. We have used the `__asm` directive to implement it. The `ldaxr` and `stlxr` instructions are used.

```
template <typename T>
static inline int CAS(T *ptr, T *oldValPtr, T *newValPtr) {
    int ret;
    int res;
    T oldVal = *oldValPtr;
    T newVal = *newValPtr;
    __asm__ __volatile__("1:\n"
                         "ldaxr %w0, [%2]\n"
                         "mov %w1, %w0\n"
                         "cmp %w0, %w3\n"
                         "b.ne 2f\n"
                         "mov %w0, %w4\n"
                         "stlxr %w1, %w0, [%2]\n"
                         "cbnz %w1, 1b\n"
                         "2:\n"
```

```
                                    : "=&r"(ret), "=&r"(res), "+r"(ptr)
                                    : "r"(oldVal), "r"(newVal)
                                    : "cc", "memory");
    return ret;
}
```

The above code is a C++ template function implementing a Compare-And-Swap (CAS) operation, typically used in concurrent programming for synchronization.

1. **Input Parameters**:
   - `ptr`: A pointer to the memory location where the CAS operation will be performed.
   - `oldValPtr`: A pointer to the old value which is expected to be found at the memory location `ptr`.
   - `newValPtr`: A pointer to the new value that will replace the old value if the CAS operation is successful.
2. **Local Variables**:
   - `ret` and `res`: These are integer variables used for return values and intermediate results in the assembly code.
   - `oldVal` and `newVal`: These variables store the values pointed to by `oldValPtr` and `newValPtr`, respectively.
3. **Inline Assembly Code**:
   - The `__asm__ __volatile__` statement introduces inline assembly code, which directly embeds assembly instructions within the C++ code.
   - The assembly code implements the CAS operation. It typically loads the current value at the memory location pointed to by `ptr`, compares it with `oldVal`, and if they match, updates the memory location with `newVal`.
   - It uses specific ARM assembly instructions (like `ldaxr`, `mov`, `cmp`, `stlxr`, `cbnz`) for atomic load, store, and conditional branching, which are common in ARM architecture.
4. **Output and Input Constraints**:
   - The `=r` constraints indicate that the corresponding variables (`ret`, `res`, `ptr`) are both inputs and outputs and will be stored in registers.
   - The `r` constraints indicate that the variables `oldVal` and `newVal` will be stored in registers but are only inputs.
5. **Clobber List**:

- The clobber list (`"cc", "memory"`) informs the compiler that the assembly code might modify condition flags (`cc`) and memory (`memory`), so it should not make any assumptions about their values before and after the assembly block.

6. **Return Value**:

   - The function returns `ret`, which indicates the success of the CAS operation. Typically, it returns 0 if the operation was successful and non-zero otherwise.

STACK and QUEUE

Finally using these universal consensus object, I have implemented a concurrent `stack` and `queue` for demonstration.

Both the objects support 2 functions:

1. `PUSH`: push value in stack (enqueue value in queue)
2. `POP`: pop value from stack (deque value from the queue)

For testing, I have invoked multiple threads trying to randomly call either `PUSH` or `POP` function on these objects with equal probability.

Verification: To verify the correctness, after each thread has finished its job, the main thread makes a copy of the object and simulate the whole linkedlist of invocations on this local object. The logs are captured for each invocation and then this log is examined through a python script whether it follows `stack` (or `queue` respectively) behaviour.