Bachelor's Thesis

Automated Testing of Atomic Instruction (`LR`/`SC`) Implementations in Selfie

by

Luis Thiele

submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science
in Informatics

Department of Computer Science
Paris Lodron Universität Salzburg
Salzburg, Austria

supervised by

Univ.-Prof. Dipl.-Inform. Dr.-Ing. Christoph Kirsch

December, 2022

Abstract

The selfie system is an educational self-compiling C* to RISC-U compiler and self-executing RISC-U emulator (C* is a subset of C and RISC-U is a subset of RISC-V). It contains an autograder that is written in python and considers source code submitted by students and runs tests against it. One of the assignments requires students to extend the selfie system such that the compiler supports the Treiber stack - a concurrent thread-safe and lock-free stack. This Treiber stack must be implemented using the atomic `LR` (load-reserved) and `SC` (store-conditional) instructions. They must also be implemented to be supported by both the compiler and the emulator.

This thesis is about extending the autograder to grade an assignment called `treiber-stack` where students must implement the Treiber stack using the atomic instructions `LR` and `SC`. The primary challenge of the new tests added to the autograder was about forcing specific interleavings of `LR` and `SC` run by the emulator the students extended. Some interleavings could be forced directly by the source code of the run test but some could only be forced by having the thread scheduler timeout and force a switch of the thread that is run. A timeout happens when the thread has run the set maximum amount of instructions sequentially without there ever being a switch to another thread in between.

Additionally, this thesis also describes the correct implementations of the `LR` and `SC` instructions in the emulator and a correct implementation of the Treiber stack in order to pass the entirety of the autograded `treiber-stack` assignment. All prerequesite requirements are also explained, namely a correct implementation of support for threads (`threads` assignment), as well as any concepts and details about the emulator one must understand such as paging or system calls.

Contents

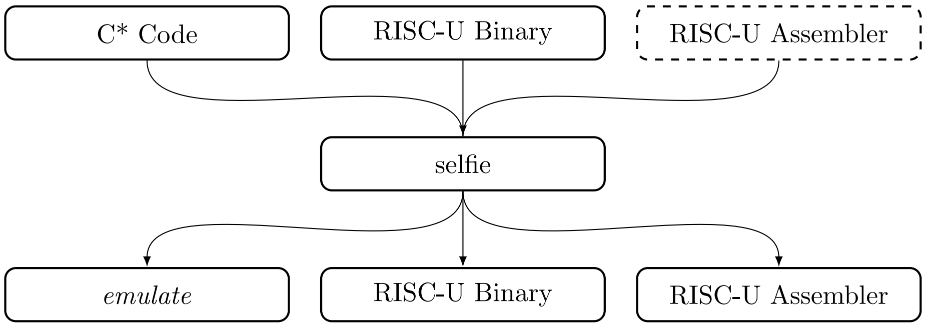

Figure 1: Capabilities of the selfie system.

1 Introduction

The selfie system is an eduactional self-compiling C* to RISC-U compiler and a self-executing RISC-U emulator (C* is a subset of C and RISC-U is a subset of RISC-V). To pass different classes students must extend selfie with certain features and then test their implementations using an autograder which comes as part of the selfie system.

This thesis describes the extension of the autograder with new tests such that implementations of a Treiber stack - a concurrent thread-safe and lock-free stack - in the selfie system are being tested. This Treiber stack must be implemented by using the atomic instructions `LR` (load-reserved) and `SC` (store-conditional) which are also to be implemented and are being tested by the new grading tests. Using these atomic instructions helps avoid the "aba" problem that would otherwise occur by using compare-and-swap instructions instead.

The automated grader already included the `treiber-stack` assignment and tests for a Treiber stack implementation. This thesis will go into detail on how this assignment was split into two different assignments (`treiber-stack` and `threadsafe-malloc`) and how it was greatly extended and improved upon.

Subsequently, the prerequesites and tasks of these assignments are explained followed by what a correct implementation might look like and what the challenges are. In the end, all the grading tests are explained in detail with emphasis on what the problems are and what the idea behind each and every test is but also not leaving out an actual explanation of their semantics.

1.1 selfie

The selfie system primarily serves as an educational software for students for classes of compilers, systems, and emulators [1]. It is written in C* which is a subset of the programming language C and can compile C* code into RISC-U binaries - a subset of RISC-V - and emulate such binaries among other things (Figure 1 - making selfie accept RISC-U assembler code is a required student

```
$ ./selfie -c selfie.c -m 2 -c some_code.c -m 1
```

Figure 2: Running code on selfie running on selfie.

assignment). The system comes with a set of assignments for students which require them to extend it in different ways ranging from adding different syntax to the compiler to introducing threads to the emulator which otherwise only executes code as a single process.

The emulator of selfie, called Mipster, runs on a fetch-decode-execute loop. First the instruction the program counter is pointing to is fetched, then it is decoded from it's binary format, followed by it being executed. The execution of an instruction comes with an increase (or change) of the program counter and thus the next fetch will be done at a different address.

Testing and grading of assignment submissions is done by an automated grader written in Python that comes with the system. This grader usually runs a program written in C*, but possibly including new syntax the student is required to add support for, on top of Mipster and then checks and matches the output to the expected output. The grader can also test RISC-U binaries for new instructions among other things.

Selfie allows you to emulate RISC-U binaries on x86 hardware. You compile selfie into an x86 binary (bootstrapping) and then run it passing your RISC-U binary as argument for emulation. You can also "stack" another selfie instance on the emulator first and then make the later emulate your binary (Figure 2).

1.2 System calls

System calls in selfie are handled by the emulator. They are initiated by the `ECALL` instruction. The emulator then looks up the value in the `REG_A7` register which represents the syscall to be executed. Then the appropriate syscall is emulated. This of course means that the fetch-decode-execute loop is broken and then restarted again. Supported system calls include `exit`, `read`, `write`, `open`, and `brk` (increase program break).

1.3 `malloc()`

Selfie's memory allocation system is implemented as bump-pointer allocator. A pointer stores the last allocated address and whenever new memory is allocated the pointer is increased ("bumped") by the size of the newly required memory and the new pointer is returned to the `malloc()` call. It is trivial to see that used memory addresses are lower than the current bump-pointer while free memory addresses are above said pointer.

Any memory allocated by `malloc()` of course persists and as such is not allocated in the stack but rather in the heap segment. Therefore the bump-pointer

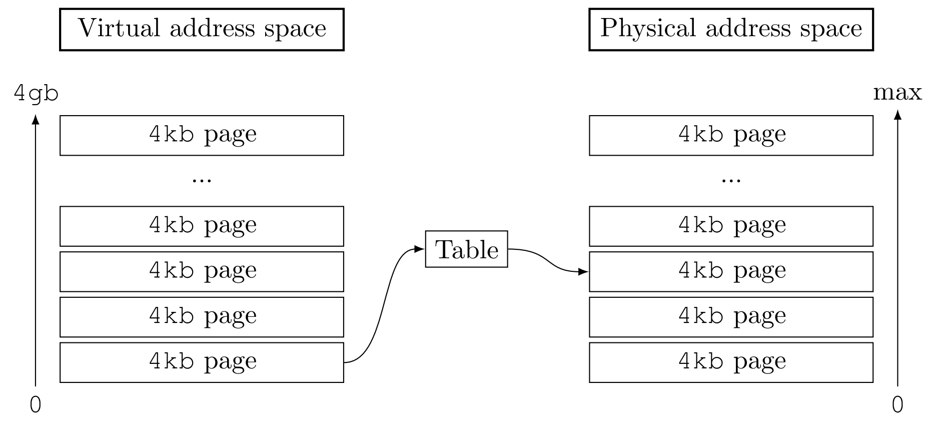

Figure 3: Address look-up involving paging.

also represents the top of the current heap segment, the program break. So whenever the bump-pointer is increased the program break must be adjusted as well. As mentioned, this is accomplished by the `brk` system call which is integrated into any `malloc()` call.

To go further into detail, `malloc()` is written as an instructions macro. Predefined instructions are emitted, which include `LD` (load the bump-pointer), `ECALL` (the `brk` system call), and `SD` (store the increased bump-pointer) in this order. Other predefined methods may be implemented as pure system call.

1.4 Paging

The memory management of Mipster follows the concept of paging. The virtual address space is divided into chunks with each of those chunks mapped onto a physical address page where all content is stored. Of course, the actual memory doesnt have to be ordered according to their virtual addresses. This concept is demonstrated in Figure 3. The arrow represents an address look-up.

Whenever there is an attempt at loading from or storing at an address (virtual address) that has not been mapped yet, Mipster issues a page fault. This is immediately handled by allocating a physical page and then mapping the virtual address to it. The mappings are stored in a table.

Otherwise, if said address has already been mapped there is an entry in the table already and Mipster applies it to the address and looks up the contents at the physical address.

```
Compiler Assignments:
|- hex-literal
|- bitwise-shift-compilation
| |- bitwise-shift-execution
|- bitwise-and-or-not
|- array
| |- array-multidimensional
|- struct-declaration
| |- struct-execution
|- for-loop
|- logical-and-or-not
| |- lazy-evaluation
```

Figure 4: A list of assignments involving the compiler construction class.

```
Systems Assignments:
|- assembler-parser
| |- self-assembler
|- processes
| |- fork-wait
| | |- fork-wait-exit
| | | |- threads
| | | | |- threadsafe-malloc
| | | | | |- treiber-stack
|- lock
```

Figure 5: A list of assignments involving the systems class.

2 Assignments

Students have to finish different assignments which require them to extend the selfie system. These can be dependent on one another, with later assignments being dependent on earlier ones. In this section the prerequesite tasks are going to be explained as well as the newly added ones as part of the project of this thesis.

These assignments are split into different groups and can depend on one another as shown in Figures 4 and 5.

2.1 Prerequisites: Processes and threads

This sub-section will explain all the assignments that are required for those implemented by this project.

2.1.1 `processes` assignment

The selfie system introduces an assignment called `processes`. This assignment requires students to implement a process scheduler and to add an execution flag with which a code is run multiple times in said scheduler, instead of only once. The process structure here exists as multiple different processes all started at the same time and running independently. If one of them exits so does Mipster.

Context switching is already implemented in selfie. Every fetch-decode-execute loop is forcibly broken after a set amount of instructions. Additionally, system calls also break said loop and as such force a context switch. This side effect of system calls is in fact even expected as the automated grading is testing for this exact phenomenon.

2.1.2 `fork-wait` and `fork-wait-exit` assignments

That assignment is later built on by the `fork-wait` and `fork-wait-exit` assignments. These now require the implementation of the `fork()` and `wait()` system calls and the extension of the `exit()` system call. This now allows for processes to be created and started during code execution (instead of only at the beginning) and comes with a strict parent-child process structure where the exit code of the child can be retrieved by the parent. Additionally, different processes can now follow different code paths (eg. a condition being fulfilled for the parent but not for the child process resulting in different code being executed).

2.1.3 `threads` assignment

Finally, as a last prerequesite assignment, the `threads` assignment requires the existing process structure to be extended to support threads. Threads are different because they share all heap memory (heap, code, and data segment to be precise). The previously mentioned system calls, `fork()`, `wait()`,

`lr.d`	`rd = memory[rs1];`
	`reserve(pid, rs1);`
	`pc = pc + 4;`

`sc.d`	`IF is_reserved(pid, rs1)`
	`  THEN memory[rs1] = rs2, rd = 0;`
	`  ELSE rd = 1;`
	`pc = pc + 4;`

Figure 6: The semantics of the LR and SC instructions.

31 27	26	25	24 20	19 15	14 12	11 7	6 0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
LR	ordering		0	addr	width	dest	AMO
SC	ordering		src	addr	width	dest	AMO

Figure 7: The encoding of the AMO instructions LR and SC [2].

and `exit()` also need to be "mirrored" for threads in `pthread_create()`, `pthread_join()`, and `pthread_exit()`.

2.2 `threadsafe-malloc` assignment

The `threadsafe-malloc` assignment requires you to make any `malloc()` call thread-safe, by replacing the `LD` and `SD` instructions with the atomic `LR` and `SC` instructions which are to be implemented.

2.2.1 `LR` and `SC` instructions

As demonstrated in Figure 6, the *load-reserved* (`LR`) instruction loads a value from a given address and registers a registration on said address with the thread that executed the instruction. If another thread executes `LR` on the same address, this reservation is overridden. The encoding of these instructions is shown in Figure 7.

This reservation is looked up when *store-conditional* (`SC`) is called. If and only if the thread calling this instruction was also the one that last reserved the address, then the result is not flagged as failure and the value is stored at said address. Otherwise, the value is not stored and the result flagged as failure.

These instructions now allow secure loading and storing of values in a multi-threaded environment. First the value is loaded (`LR`), then it is edited, and finally it is stored (`SC`). If the store-conditional fails, load the value again and repeat everything until it does not fail. `malloc()` must be changed to adhere to this structure.

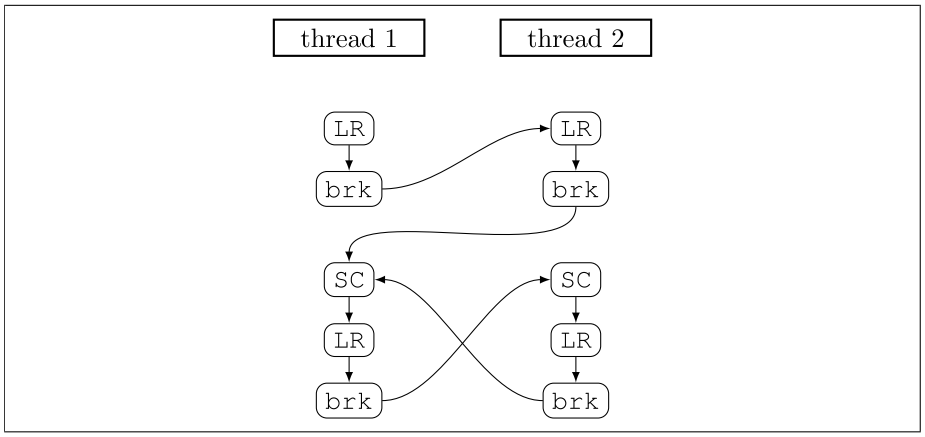

Figure 8: `malloc()` livelock caused by context switches.

2.2.2 `malloc()` livelocks

Additionally, the `malloc()` call which includes the `brk` system call must be changed in such a way that the fetch-decode-execute loop is not broken anymore. If this is not done, multiple threads calling the new `LR`/`SC`-based `malloc()` can cause a livelock as shown in Figure 8.

Let us assume the `main()` method only consists of a `pthread_create()` call, followed by a `malloc()` call, so basically just 2 threads both calling `malloc()`. Thread 1 as such executes `LR`, followed by the system call `brk`, followed by `SC`. However, before the store-conditional is actually executed there is a context switch happening because of the system call. Now Thread 2 calls `LR`, followed by the `brk` system call. Back to thread 1 which now gets to execute `SC`. This store-conditional fails, however, as thread 2 currently has a reservation on the address of the bump-pointer. So it repeats `LR` and then calls `brk`. Now thread 2 attempts `SC` but also fails, so it repeats as well.

Finally, this assignment also requires you to implement the methods `lr()` and `sc()` with the former using the `LR` instruction and returning the value at the given address, and the later using `SC` and storing the given value at the given address and returning the failure-flag. These methods are to be implemented as macros, not system calls, meaning that on compilation no `ECALL` instruction is to be emitted.

2.3 `treiber-stack` assignment

The `treiber-stack` assignment requires you to implement the native methods `init_stack()`, `push()`, and `pop()`. Again, these are to be implemented as macros, not system calls. The `push()` method pushes a given value on to the stack and `pop()` returns them again following the *LIFO* principle.

`init_stack()` is called once first, before the other two, to do any required initialization. These methods must be implemented thread-safe so that detached heads are fully avoided.

Formerly, this assignment also required you to implement all tasks of the `threadsafe-malloc` assignment (the later did not exist) with the exception of specifically having to make `malloc()` not force context switches. It is also worth mentioning that the livelock problem has more solutions but the one explained is deemed to be the most solid one.

3 Implementation

This chapter will dive into more specifics about implementations used to accomplish the tasks of all assignments. There will be no practical code shown. However, all the principles used will be explained in depth.

3.1 `threads` assignment implementation

The most difficult part of this assignment is the shared heap-memory of threads. The creation of new contexts and the new methods are trivial and will not be explained.

One approach to implement the heap-memory sharing of threads is by mapping their virtual pages to the same physical pages. Assume there is a thread `A` and a thread `B` both having shared heap-memory. The value of heap-memory address x would be stored on a single physical page with two different virtual pages (one of `A` and one of `B`) being mapped on to it.

The easiest way to accomplish this is by immediately mapping a physical page to all threads instead of only one as shown in Figure 9:

- Assume thread `A` stores a value at a heap-address that has not been mapped before. A page fault happens, so thread `A` now creates a new virtual page and maps it to a new physical page which contains the value. All other threads now also create a new virtual page and map it onto the same physical page and therefore have the same value at the same virtual address. Overriding the value modifies the physical page which reflects the change for all threads since they all translate to said page.
- Assume thread `A` stores a value at a heap-address that has already been mapped. The address is translated to the physical page and stored/overridden there. Once again, since the value is modified in the physical page it is reflected to all other threads.

Of course, the program break must also be changed for all threads not only a single one, and whenever a thread creates a new child thread all existing virtual pages must be created and mapped for the child.

```c
void do_map_page(uint64_t* thrd, uint64_t vpage, uint64_t
↪ ppage) {
  // map the page and do everything required
  // ...

  // now "trickle down" to all child threads
  if (is_heap_page(vpage)) {
    thrd = get_thread_child(thrd);

    while (thrd != (uint64_t*) 0) {
      do_map_page(thrd, vpage, ppage);
      thrd = get_thread_sibling(thrd);
    }
  }
}

void map_page(uint64_t* thrd, uint64_t vpage, uint64_t
↪ ppage) {
  if (is_heap_page(vpage))
    while (get_thread_parent(thrd) != (uint64_t*) 0)
      thrd = get_thread_parent(thrd);

  do_map_page(thrd, vpage, ppage);
}
```

Figure 9: Example algorithm to keep mapped pages synchronized across threads.

3.2 `threadsafe-malloc` assignment implementation

First the new `LR` and `SC` instructions must be implemented. These mirror the existing `LD` and `SD` instructions to a large part. Apart from the difference in semantics, the primary addition to be implemented here are the new semantics involving reservations. Threads all have a unique id which can be used for marking reservations. These reservations must be set when emulating `LR` and checked when emulating `SC`. The ideal data structure for this seems to be a linked list.

The `LD` and `SD` instructions of `malloc()` must be replaced by the newly implemented `LR` and `SC` instructions. The structure must be extended so that whenever `SC` failes the program counter jumps back to `LR`. One should also adapt the `gc_brk` system call which replaces `brk` whenever the garbage collector is enabled. This call skips a few instructions of `malloc()` and the amount of instructions skipped must be increased to account for the thread-safe extension.

Additionally, `malloc()` may not force context switches anymore. By default, every system call returns a `EXIT`/`DONOTEXIT` flag and the emulator either exits or switches context depending on this flag. Ideally, this concept is changed so that every system call makes the emulator either `EXIT`, `SWITCH` (context switch), or `STAY` (do not switch context). Of course, the scheduler must also be adapted for this change.

Finally, one must also implement the native `lr` and `sc` methods. These will not be explained as they are trivial.

3.3 `treiber-stack` assignment implementation

For this assignment a new variable must be introduced which always points at the head of the stack. The whole concept of this native variable is already implemented for the bump-pointer allocator (variable "`_bump`") and must essentially be copied. This is the only way to do it because the new variable must be in the heap memory as all Treiber stack methods are supposed to be macros and not system calls. Assume the name of this new variable is `_head`.

To avoid any confusion in the following paragraphs, "stack" refers the actual stack-memory of the thread accessed with `REG_SP`, and "Treiber stack" refers to the stack accessed by the three methods which must be implemented for this assignment.

For the `init_stack()` method all that needs to be done is resetting `_head` to 0. This can be done with a single `SD` instruction followed by `JALR`, of course.

The `push()` method is the most difficult to do. It requires the parameter (the value pushed) to be gathered from the stack and temporarily stored in some register. Next `malloc()` must be called (jumped to) with 16 as parameter (`ADDI` 16 into a register and then move it onto the stack). This is done to allocate a new Treiber stack entry with two values: A pointer to the previous entry (_head points to it) and the pushed value itself. After the return from malloc() the pointer must be gathered from the stack and those two values must be stored at the returned address. Finally, _head must point to the address of the new Treiber stack and the whole macro must obviously end with JALR.

To implement pop(), you first load the value of _head to get the address of the top Treiber stack entry. Move the value of this entry into REG_A0 for the return and then set _head to the Treiber stack entry this entry points to. End with JALR again.

To mention this specifically, push() and pop() require loading and storing of _head. These methods must also be threadsafe, however. And as such, this must be done using the LR and SC instructions. A critical section lies between these two instructions and is to be repeated on failure. The critical section of push() is setting the pointer of the newly allocated entry to the loaded value. The critical section of pop() is gathering the the address of the entry the top entry (loaded value) points to.

4 Grading

This section will go through the various tests created to automatically test students' assignment submissions. Some of these tests are very simple and straight forward and some are more complicated. The code of all test files can be found in the appendix.

The grader is a program written in Python and run with an assignment as parameter. To test the assignment implementation of a student it compiles the student's modified selfie system and then compiles or emulates certain pre-written C* files on it usually checking the output or return code for a specific pattern or value. If this is not found or if this compilation or emulation fails or throws any errors the test failed.

4.1 threadsafe-malloc assignment grading

The load-reserved.c and store-conditional.c tests already existed as part of the old threadsafe-malloc assignment but will be explained nevertheless. This assignment expects all systems calls with the exception of malloc() to still force context switches.

To pass this assignment malloc() must be made fully thread-safe: If two different threads call malloc() at the same time there are no issues as if the calls happen sequentally and with no effects depending on the interleaving of their actual machine instructions.

4.1.1 load-reserved.c and store-conditional.c

First, we must make sure that the new LR and SC instructions are emitted properly with the right format. The new native lr() and sc() methods must also compile and emit said instructions. This is accomplished by the `load-reserved.c` and `store-conditional.c` tests.

These each only contain one instruction which is a call of the native `lr()` or `sc()` method. The semantics are not tested but rather the point of these tests is to force the selfie system to emit the new atomic instruction when compiling these tests and creating the RISC-U binaries containing them. The grader then searches these binaries for the correct format (AMO) of these instructions. If they are not found the tests fail. If the native methods do not compile the grader, of course, also fails.

4.1.2 `lr-sc-interleaved.c`

Next, we must test the actual semantics of the `LR` and `SC` instructions. The `SC` instruction must fail if the thread calling it was not the last one to call `LR` on the same address. This means that the failure-flag is set to cod1 and that the value at said address is not overridden. Otherwise, the opposite must happen, of course. This is done by the `lr-sc-interleaved.c` test.

Initially, a value is stored at an address and then two threads are created (parent and child). The child thread is running first and calls `lr()` on said address followed by a context switch. The parent thread now also calls `lr()` on the same address and checks the value. If the value is incorrect the test fails. Otherwise the parent calls `sc()` and stores a new value (still the same address). A context switch is forced and the child now also calls `sc()` overriding the value the parent just stored. If the semantics are correct this override should fail. The flag whether or not it did fail is returned by the child (it terminates). Finally, he parent fetches this flag and validates that the `sc()` call of the child has failed by both checking this flag and checking the value at the used address. The test returns the value 42 only if the test did not fail and the grader checks this value.

4.1.3 `no-switch-malloc.c`

The `no-switch-malloc.c` test simply validates if a `malloc()` call still forces a context switch. This may not happen for the test to pass.

Again, two threads are created (parent and child). The child runs first and calls `malloc()` followed by printing the first half of a `"Hello World!  "` string and terminating. The parent thread then prints the second half of this string and also terminates. If `malloc()` would still force context switches then (depending on the student's implementation) the string halfs are either printed the wrong way around or the child does not print the first half at all. The grader validates if the output is correct.

It is important to note that this assignment expects only `malloc()` to not force context switches anymore. It does still expect the other system calls to follow this behaviour (to force context switches).

4.1.4 **`threadsafe-malloc.c`**

The goal of the `threadsafe-malloc.c` test is to force a context switch between the `LR` and `SC` instructions of the `malloc()` call. Since `malloc()` does not force context switches anymore this can only be done by timeout. So the grader attempts to run a lot of instructions almost reaching the instructions limit followed by `malloc()` in order to have that context switch by timeout happen at the right moment. Again, there are two threads (parent and child) and the child goes first. This is the timeline of the actual test:

1. The child thread initializes a bunch of variables and then forces a context switch to have its next section run on a fresh timer.
2. The parent thread immediately switches back to child.
3. The child now has an endless loop running with a counter in its body. A context switch happens due to timeout.
4. Next, the parent thread terminates the endless loop of the child and switches.
5. The child thread finishes the last run of the loop and now has a counter which holds the amount of times such a loop can be repeated before a context switch happens by timeout. This value is slightly decreased to later repeat the loop right before the instructions limit to then call `malloc()` and have the context switch happen during said call (between `LR` and `SC`). However, since it finished the last run of the endless loop the timer must be reset before this is done so the child switches.
6. The parent immediately forces a context switch back to the child.
7. The child runs a loop almost reaching the limit of instructions before a context switch happens and calls `malloc()`. Before said call finishes this context switch happens by timeout, of course.
8. The parent now calls `malloc()` itself twice and joins the child (waits for the child to terminate and switches). These `malloc()` calls should conflict with the one done by the child since they all increase the bump-pointer variable and try to store them. The child will have `SC` fail and jump back to `LR` if correctly implemented.
9. The child finishes the `malloc()` call. `SC` fails and is repeated. This malloc call actually allocates enough space for two variables, obviously adjacent to each other. Then different values are stored there and the address returned by the `malloc()` is returned to the parent.
10. Finally, the parent fetches this address and stores new values at the two addresses it allocated memory at before. To clarify, all stored values and their addresses are distinct. The test passes only if this holds. Otherwise, some values or addresses are duplicate and the test fails

4.2 `treiber-stack` assignment grading

The `stack-push.c` and `stack-pop.c` tests already existed as part of the old `threadsafe-malloc` assignment but will be explained nevertheless.

The goal is to validate that `push()` and `pop()` methods work correctly in a multi-threaded environment. Multiple threads are created and forced to run in a specific order (using `pthread_wait()`) while calling said methods and the result gets validated by the grader.

4.2.1 `stack-push.c`

The `stack-push.c` test first creates a total of 8 different threads. All of these threads call `push()` once each with distinct numbers $e \in [0, 7]$. Next, threads wait or terminate so that only the main thread is left. This main thread now calls `pop()` 8 times and writes all returned values to console. The test is valid if all distinct values are found.

4.2.2 `stack-pop.c`

The `stack-push.c` test works similarly. This time the main thread calls `push()` with all distinct numbers $e \in [0, 7]$ and then disperses into 8 different threads. Each of these threads calls `pop()` and then waits for its children (if there are any). Again, the tast passes if all distinct values are found.

5 Changes to selfie

There have been numerous other changes and additions to the selfie system which this section is about.

5.1 `logical-and-or-not` assignment

Two additional assignments were added to the compiler construction class with the first one being the `logical-and-or-not` assignment [3]. The student must implement the logical (boolean) AND (`&&`), OR (`||`), and NOT (`!`) operators including their correct precedence. Whenever an operation is done using any of these operators the `uint64_t` result must always be either 1 (`true`) or 0 (`false`). Any `uint64_t` operant $n = 0$ is seen as `false`, otherwise if $n \neq 0$ it is seen as `true`.

To grade this assignment different test code files are run and their results checked. A lot of different cases (eg. with precedence) are checked and if anything fails the result will be wrong resulting in a failed grade.

5.2 `lazy-evaluation` assignment

The second assignment added is the `lazy-evaluation` assignment. This assignment requires the student to change the compiling of the logical AND

```
uint64_t endless_loop(uint64_t v) {
  while(1) {}
  return v;
}

void main() {
  uint64_t f;
  uint64_t t;

  f = 0; // false
  t = 1; // true

  if (t || endless_loop(f))
    if (f && endless_loop(t))
      return 10; // incorrect branch
    else
      return 42; // correct branch
  else
    return 11; // incorrect branch
}
```

Figure 10: Example using endless loops to test for lazy evaluation.

and OR operators in such a way that the principle of lazy evaluation is applied. The tests are done using endless loops as shown in figure reffig10. Whenever lazy evaluation should have skipped any left over operants (in this case always the operants after the only operator) these operants are calls to an endless loop. If they are indeed properly skipped, the endless loop does not get executed, otherwise it does. The grader simply executes these tests and checks for the correct return value while also having the tests fail if they timeout which happens if the endless loop gets executed.

5.3 Changes to the grader

New flags were added to the grader which can be invoked on command line [4]:

- –dependency-tree: This flag prints out all assignments showing their dependencies as shown in figures 4 and 5. Without this flag the assignments are shown as list.
- `-a`: Whenever an assignment is graded and this flag is active all assignments it is dependent on are also tested.
- `-s`: This flag disables any animations as they can cause problems in certain consoles on certain operating systems.

5.4 Changes to GitHub actions

There were two issues involving the selfie workflows of GitHub actions, one being severe:

- For the major issue, student repositories are to be kept private, of course, to obstruct cheating attempts. These repositories, however, would execute different workflows on every git push and since the repositories are private the amount of time these workflows are running is using up minutes from a certain limit that you can only increase by paying. But for a proper grade these workflows should work for the final submission. To combat the unnecessary using up of minutes-quote two things were changed [5]:

  1. The amount of quota used up is dependent on the OS the workflow is running on. The selfie system used to run on Windows, MacOS, and Linux for its workflows. However, MacOS uses up ten times of the amount that Linux does, and Windows uses twice the amount of Linux. As such, the workflows were changed in such a way that, for private repositories only, Linux is the only OS the workflows are run on leaving out Windows and MacOS. Public repositories stay unchanged. This way the quota used up by students is severely lowered making it a lot harder to reach the monthly free limit.

  2. The workflows are always running on every push but only need to for the final commit. Since students typically work on a side branch (other than `main`) to do their assigment and then must do a squashed merge into `main` the commits/pushes to `main` are almost exclusively what they are going to submit as their solution (the finished assignment). And as such the workflows were changed to do the workflows only whenever there is a push to the `main` branch, for private repositories only (public repositories stay unchanged again).

- The minor issue is about the usage of grading withing the GitHub actions interface. The grader is using some smaller animations when evaluating by changing what was already written to the console. Within said interface this would not work correctly, however, and result in every single frame of the animation being printed (and staying) in the console. The `-s` option was added to the invocation of the grader in all workflows to omit any animations and prevent this issue.

Additionally, you can now chose the OS to run the workflows on if they are triggered manually. And as a final quality of life addition, the assignment chosen for grading is now chosen via drop down menu instead of a string input field.

5.5 Code restructuring

As a final addition major code restructuring took place in the compiler [6]. The code part defining how statements are compiled is now shorter and more clearly structured making one of the hardest assignments of the compiler construction class, the `array` assignment, a lot easier now. Additionally, the `grammar.md` file (it mirrors the components and structure of the entire compiler and acts as an overview of how the compiler works) has been more closely aligned to the actual compiler code.

6 References

[1] Computational Systems Group of the Department of Computer Sciences at the University of Salzburg in Austria. selfie.
`https://github.com/cksystemsteaching/selfie`, 2022.

[2] Andrew Waterman, Krste Asanovi‘c, SiFive Inc. The RISC-V Instruction Set Manual.
`https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf`, 2022.

[3] Luis Thiele. New grading tasks: logical-and-or-not and lazy-evaluation by CAS-ual-TY - Pull Request #312 - cksystemsteaching/selfie.
`https://github.com/cksystemsteaching/selfie/pull/312`, 2022.

[4] Luis Thiele. lazy-eval description / Show grading dependencies by CAS-ual-TY - Pull Request #320 - cksystemsteaching/selfie.
`https://github.com/cksystemsteaching/selfie/pull/320`, 2022.

[5] Luis Thiele. Workflow improvements by CAS-ual-TY - Pull Request #313 - cksystemsteaching/selfie.
`https://github.com/cksystemsteaching/selfie/pull/313`, 2022.

[6] Luis Thiele. compile_statement restructure (remake of #293) by CAS-ual-TY - Pull Request #322 - cksystemsteaching/selfie.
`https://github.com/cksystemsteaching/selfie/pull/322`, 2022.

[7] Luis Thiele. Thread-safe malloc grading task by CAS-ual-TY - Pull Request #319 - cksystemsteaching/selfie.
`https://github.com/cksystemsteaching/selfie/pull/319`, 2022.

[8] Daniel Kocher. “Systems Engineering” Lecture at the University of Austria.
`https://online.uni-salzburg.at/plus_online/ee/ui/ca2/app/desktop/#/slc.tm.cp/student/courses/575978`, 2021/2022.

[9] Christoph Kirsch. “Introduction to Compiler Systems” Lecture at the University of Austria.
`https://online.uni-salzburg.at/plus_online/ee/ui/ca2/app/desktop/#/slc.tm.cp/student/courses/523123`, 2021.

A Grading tests (C* Code)

A.1 `threadsafe-malloc` assignment tests

A.1.1 `load-reserved.c`

```c
uint64_t lr(uint64_t address);

int main(int argc, char** argv) {
  return (int) lr(0);
}
```

A.1.2 `store-conditional.c`

```c
uint64_t sc(uint64_t address, uint64_t value);

int main(int argc, char** argv) {
  return (int) sc(0, 1);
}
```

A.1.3 lr-sc-interleaved.c

```c
uint64_t lr(uint64_t address);
uint64_t sc(uint64_t address, uint64_t value);

uint64_t pthread_create();
uint64_t pthread_join(uint64_t* wstatus);

// used to force context switches
// there are no process children, so this does nothing
uint64_t wait(uint64_t* wstatus);

uint64_t child = 0;

int main(int argc, char** argv) {
  uint64_t* address;
  uint64_t* status;
  uint64_t pid;
  uint64_t x;

  address = malloc(8);
  *address = 10;

  status = malloc(8);

  pid = pthread_create();

  if (pid == 0) {
    // child
    child = 1;

    x = lr(address);

    wait((uint64_t*) 0);

    x = sc(address, x);

    return x; // must be 1 (failed)
  } else {
    // parent

    // make sure child is running first
    while (child == 0)
      wait((uint64_t*) 0);

    x = lr(address);

    if (x == 10) {
      if (sc(address, 42))
        return 7; // parent sc may not fail but it did
    } else
      return 8; // wrong x value, something went entirely wrong

    // switch to child
    x = pthread_join(status);

    if (x == 0)
      return 9; // child sc must fail but it did not
    else if (*address == 10)
      return 6; // child sc stored a new value which it may not do

    // should be 1 * 42 = 42
    return *status * *address;
  }
}
```

A.1.4 no-switch-malloc.c

```c
uint64_t write(uint64_t fd, uint64_t* buffer, uint64_t bytes_to_write);

uint64_t pthread_create();
uint64_t pthread_join(uint64_t* wstatus);

uint64_t* foo;
uint64_t child;

int main(int argc, char** argv) {
  uint64_t pid;

  foo = "Hello World!    ";

  child = 0;

  pid = pthread_create();

  if (pid) {
    // main thread

    // make sure child is executed first
    // variable is shared
    while (child == 0) {}

    write(1, foo + 1, 8);

    return 0;

  } else {
    // child thread
    child = 1;

    // see if we switch back to main (parent)
    malloc(8);

    write(1, foo, 8);

    return 0;
  }
}
```

A.1.5 threadsafe-malloc.c

```c
uint64_t pthread_create();
uint64_t pthread_join(uint64_t* wstatus);
void pthread_exit(uint64_t code);

uint64_t wait(uint64_t* wstatus);

uint64_t thrd;
uint64_t loop;

int main(int argc, char** argv) {
  uint64_t pid;
  uint64_t counter;
  uint64_t i;
  uint64_t* p1;
  uint64_t* p2;
  uint64_t counter;
  uint64_t zero;

  thrd = -1;

  pid = pthread_create();

  if (thrd == -1)
    thrd = pid;

  // make sure the child thread is running first
  if (thrd != 0)
    wait((uint64_t*) 0);

  // assert: thrd == 1

  //counter = 1111108;

  if (pid == 0) {
    // child

    // ------------------------------ 1st --

    zero = 0;

    counter = 0;

    loop = 1;

    i = 0;

    // force a context switch, reset the timer
    wait((uint64_t*) 0);

    // ------------------------------ 3rd --

    while(zero < loop)
      counter = counter + 1;

    // endless loop forces context switch by timeout

    // ------------------------------ 5th --

    counter = counter - 2;

    // force a context switch, reset the timer
    wait((uint64_t*) 0);

    // ------------------------------ 7th --

    // 3 jalr instruction leftover from wait()

    // 9 instructions per loop
    // 1111109 * 9 = 9999981
    while (i < counter)
      i = i + 1;

    // 4 instructions on loop end

    // so far: 3 + 9999981 + 4 = 9999988

    // 12 instructions left

    // lr is 10th instruction of malloc() call

    // context switch should happen in here by timeout
    // after lr but before sc
    p2 = malloc(16);

    // ------------------------------ 9th --

    *p2 = 18;
    *(p2 + 1) = 19;

    return (uint64_t) p2;

  } else {
    // parent

    // ------------------------------ 2nd --

    // force a context switch, reset the timer
    wait((uint64_t*) 0);

    // ------------------------------ 4th --

    loop = 0;

    // force a context switch, reset the timer
    wait((uint64_t*) 0);

    // ------------------------------ 6th --

    // force a context switch, reset the timer
    wait((uint64_t*) 0);

    // ------------------------------ 8th --

    p1 = malloc(16);
    p2 = malloc(8);

    pthread_join(p2); // switch

    // ------------------------------ 10th --

    // p2 points to other p2 which points to 2 and 3

    p2 = (uint64_t*) *p2;

    // now p2 directly points to 2 and 3

    *p1 = 2;
    *(p1 + 1) = 3;

    // 10 = 2 + 3 + 2 + 3
    // 42 = 2 + 3 + 18 + 19 (thread-safe)
    return *p1 + *(p1 + 1) + *p2 + *(p2 + 1);
  }
}
```

A.2 `treiber-stack` assignment tests

A.2.1 `stack-push.c`

```c
uint64_t write(uint64_t fd, uint64_t* buffer, uint64_t
 ↪  bytes_to_write);

uint64_t pthread_create();
uint64_t pthread_join(uint64_t* wstatus);

uint64_t lr(uint64_t address);
uint64_t sc(uint64_t address, uint64_t value);

uint64_t* id;
uint64_t* c;

uint64_t allocate_id() {
  uint64_t value;

  value = lr((uint64_t) id);

  while (sc((uint64_t) id, value + 1)) {
    value = lr((uint64_t) id);
  }

  return value;
}

void print_integer(uint64_t i) {
  // single character number
  *c = 32 * 256 + 48 + i;
  write(1, c, 2);
}

int main(int argc, char** argv) {
  uint64_t pid1;
  uint64_t pid2;
  uint64_t pid3;
  uint64_t* s;
  uint64_t* results;
  uint64_t i;

  id = malloc(8);

  *id = 0;

  s = malloc(8);

  c = malloc(8);

  init_stack();

  push(allocate_id());
  push(allocate_id());
  push(allocate_id());
  push(allocate_id());
  push(allocate_id());
  push(allocate_id());
  push(allocate_id());
  push(allocate_id());

  // 2^3 processes
  pid1 = pthread_create();
  pid2 = pthread_create();
  pid3 = pthread_create();

  print_integer(pop());

  // do not wait for child-threads of the parent-process
  if (pid3 == 0)
    pid2 = 0;
  if (pid2 == 0)
    pid1 = 0;

  if (pid1)
    pthread_join(s);
  if (pid2)
    pthread_join(s);
  if (pid3)
    pthread_join(s);
}
```

A.2.2 stack-pop.c

```c
uint64_t write(uint64_t fd, uint64_t* buffer, uint64_t
  ↪  bytes_to_write);

uint64_t pthread_create();
uint64_t pthread_join(uint64_t* wstatus);

uint64_t lr(uint64_t address);
uint64_t sc(uint64_t address, uint64_t value);

uint64_t* id;
uint64_t* c;

uint64_t allocate_id() {
  uint64_t value;

  value = lr((uint64_t) id);

  while (sc((uint64_t) id, value + 1)) {
    value = lr((uint64_t) id);
  }

  return value;
}

void print_integer(uint64_t i) {
  // single character number
  *c = 32 * 256 + 48 + i;
  write(1, c, 2);
}

int main(int argc, char** argv) {
  uint64_t pid1;
  uint64_t pid2;
  uint64_t pid3;
  uint64_t* s;
  uint64_t* results;
  uint64_t i;

  id = malloc(8);

  *id = 0;

  s = malloc(8);

  c = malloc(8);

  init_stack();

  // 2^3 processes
  pid1 = pthread_create();
  pid2 = pthread_create();
  pid3 = pthread_create();

  push(allocate_id());

  // do not wait for child-threads of the parent-process
  if (pid3 == 0)
    pid2 = 0;
  if (pid2 == 0)
    pid1 = 0;

  if (pid1)
    pthread_join(s);
  if (pid2)
    pthread_join(s);
  if (pid3)
    pthread_join(s);

  if (pid1 != 0)
    if (pid2 != 0)
      if (pid3 != 0) {
        // main thread
        print_integer(pop());
        print_integer(pop());
        print_integer(pop());
        print_integer(pop());
        print_integer(pop());
        print_integer(pop());
        print_integer(pop());
        print_integer(pop());
      }
}
```