# Project 4: “Goroutines” User-Space Threads
15-410 Operating Systems
April 23, 2018

## Contents

1 Introduction ................................................. 2  
  1.1 Concurrency and Parallelism ......................... 2  
  1.2 The CSP Lineage ....................................... 3  
  1.3 Project 4 .............................................. 4  
  1.4 Document Roadmap .................................... 4  

2 Grading Guidance ........................................ 5  

3 Specifications ............................................ 5  
  3.1 Virtual Timer Interrupts ............................... 6  
  3.2 Thread Library ....................................... 6  
  3.3 Hand-in .............................................. 7  

4 Provided Tests .......................................... 7  
  4.1 C tests ............................................... 8  
  4.2 P tests ............................................... 8  
  4.3 Actual Parallelism .................................... 9  

5 Plan of Attack .......................................... 10  

A Test Feature Table ................................... 11
1 Introduction

Back in January, before you embarked on the journey of implementing a thread library, we discussed reasons why people use threads and different kinds of threads: “1:1,” “M:1,” and “M:N.” Although it was never explicitly specified, all groups implemented a 1:1 thread library for P2. In this project you will explore the other kinds of threading—and, of course, write an extension to your kernel.

1.1 Concurrency and Parallelism

This project should shed some light on two related terms which are often confused, namely concurrency and parallelism. Two piece of code can be considered concurrent if their executions do not logically depend on each other. For example, adding 7 to one variable and subtracting 19 from another don’t interact at all, so there is no need to impose a temporal ordering between them. If the two activities are carried out on an old-fashioned processor with a single ALU, one must be done first. The traditional linear/sequential encoding of instructions forces a compiler to place one instruction before the other in memory, and the old-fashioned processor will execute that one before the other—the executions will be strictly ordered by accident, even though the operations are concurrent.

Two activities can be executed in parallel if they can be carried out during overlapping time ranges because there is enough hardware to support that. A modern processor core has enough hardware to perform multiple addition/subtraction operations in parallel, so it could actually do the addition and the subtraction at the same time. There is a style of processors, “VLIW,” in which every instruction specifies multiple operations to be carried out in parallel, but VLIW processors are not common. Instead, many modern processors execute instructions “out of order” (with respect to the serial order in which the compiler placed them in memory). To do this, they analyze serial code, prove that parts of it are concurrent, and then execute the concurrent parts in parallel. There is a higher-level version of the approach of finding concurrency and using the concurrency to enable parallel execution. Programmers can explicitly provide a system with multiple independent “sequential” instruction streams (functions) within a single program and then we explicitly declare them to be concurrent (by running those functions as thread bodies). This may enable a system to execute these multiple functions in parallel.

But concurrency can be used for reasons other than parallelism, and attempting excessive parallelism can hurt performance instead of improving it. While a 1:1 thread library has a 1:1 relationship between “logical threads,” created by thr_create(), and “kernel threads,” created by thread_fork, sometimes other mappings work better. In some situations it makes sense to create 1,000 logical threads which will be executed by four kernel threads, or even one kernel thread.
1.2 The CSP Lineage

Some of you are familiar with the “Go” language.\(^1\) Go inherits a programming-language lineage which begins in 1978, when Sir Charles Antony Richard Hoare proposed decomposing complicated programs into “communicating sequential processes” (CSP). In the CSP world view, the processes that make up a computation (which we would probably call “threads” today) do not share memory; all interactions occur via passing messages through communication “channels.” This approach greatly simplifies reasoning about the correctness of concurrent programs and also generally eliminates the need for using basic synchronization primitives — all locking takes place inside the code which implements channels. The CSP lineage includes CSP, Squeak and Newsqueak, Alef, Limbo, and Go.\(^2\)

Go calls its logical threads “goroutines”.\(^3\) They are “light-weight” in the sense that creating a goroutine doesn’t require the kernel to allocate a thread control block and a kernel stack. In fact, creating a goroutine may not involve making a system call at all. For reasons that are out of scope for this project, goroutine user-space stacks can often be smaller than user-space stacks allocated by traditional thread systems. In terms of time, switching from one goroutine to another can be faster than a traditional thread switch because it may not be necessary to dive into the kernel and come back out. Together, these space and time efficiencies make it feasible for a programmer to divide a problem into many logical threads in accordance with the fundamental structure of the problem, and for these logical threads to be executed on top of enough kernel threads to keep the physical processor cores busy. In practice you may need some more kernel threads to keep all of the processors busy if a kernel thread makes a blocking system call on behalf of a logical thread. One elegant approach to the blocking-system-call problem is called “scheduler activations;” two papers about this are on the book-report list. A less-elegant approach involves adding preemption to user space, so that a kernel thread executing on behalf of one logical thread can receive a “timer interrupt” and switch to running a different logical thread.

If you’ve taken 15-312, you’ll recognize this general approach as similar to the the “concurrent Algol” paradigm. If you’re in ECE, you may see some similarity to systolic array architectures, the Inmos Transputer architecture, or various modern architectures that include explicit inter-processor communication channels. If this style of programming appeals to you, you might consider taking 15-440/640 (Distributed Systems) in a Fall semester, as it is traditionally taught in Go. To demonstrate the effectiveness of this programming style, we have provided a simple channel library in `410user/libchan` and an implementation of the classical “Sieve of Eratosthenes” in `410user/progs/sieve.c`. We hope you find it fairly elegant!

\(^1\)https://golang.org/
\(^2\)https://swtch.com/~rsc/thread/
\(^3\)https://golangbot.com/goroutines/
1.3 Project 4

The goal of this project is to implement something similar to Go’s goroutines. You will end up implementing one of the four following types of thread library.

Cooperative N:1 The most basic version of this project is a cooperative N:1 thread library. This means that all user-space threads will run in a single kernel thread (thread_fork is not invoked), and threads can be switched out only via explicit calls to thr_yield().

You will note that implementing this should draw upon a lot of code that you’ve already written, namely, a context switch, a scheduler, and thread library functions based on robust synchronization primitives.

Preemptible N:1 Programming using a purely cooperative thread library can be annoying - few people like to explicitly yield all the time, and a single thread refusing to yield can completely block all the others. Wouldn’t it be nice if we could use timer interrupts to run different threads, as we do in the kernel? For this flavor of the project, your thread library will receive virtual timer interrupts in user space and will use them to switch between logical threads.

Cooperative M:N The real fun of Goroutines is that they can run in multiple kernel threads. That being said, multiple kernel threads brings the risk of a myriad of race conditions :-) For this more challenging version of the project, you will allow the user to specify the number of kernel threads in which user-space threads are run. Ideally, all of the user-space threads owned by the thread library should have approximately even loads. Furthermore, the multiple kernel threads should be able to run in parallel.

The synchronization design for this type of thread library can be rather tricky, and it is important to keep in mind that all of the kernel threads should be making progress if possible. Some synchronization approaches may not satisfy this requirement, so design carefully.

Preemptible M:N The most challenging version of this project features the M:N thread library but with preemptible threads. This means leveraging the kernel changes to deliver timer interrupts in user space and using them to context switch. However, because multiple kernel threads will be switching among logical threads, this will undoubtedly introduce/reveal a myriad of bugs. Make sure that your synchronization design is robust before enabling context switching on timer ticks.

1.4 Document Roadmap

This document consists of the following parts:

1. Introductory material about Goroutines and types of thread libraries (Section 1)
2. Grading guidance (Section 2)

3. Specifications for changes required to P3 (Section 3.1)

4. Specifications for new thread library (Section 3.2)

5. Description of provided tests (Section 4)

6. Attack plan, including suggestions (Section 5)

2 Grading Guidance

This section is intended to provide you with some guidance on our thinking. Actual grading criteria will be finalized later.

In addition to shared/baseline items, we will be accepting four different finished products, targeting achievable points based on difficulty.

<table>
<thead>
<tr>
<th>10</th>
<th>README etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td><code>swexn()</code> timer</td>
</tr>
<tr>
<td>45</td>
<td>N:1 cooperative thread library</td>
</tr>
<tr>
<td>55</td>
<td>N:1 preemptible thread library</td>
</tr>
<tr>
<td>65</td>
<td>M:N cooperative thread library</td>
</tr>
<tr>
<td>75</td>
<td>M:N preemptible thread library</td>
</tr>
</tbody>
</table>

Discuss how much time you plan to devote to this project with your partner before starting, and try to choose a target. Keep in mind that, depending on design decisions you take, it may be difficult to change a completed N:1 implementation to a working M:N implementation.

Please identify which P4 you are implementing at the beginning of your README!

3 Specifications

This section describes `swexn()`-based “user-space timer interrupts” and the various thread-library flavors.

Note that you are expected to provide a complete implementation of one flavor of thread library. All tests should run on your 1:1 P2 thread library, and should also run on an M:N preemptible thread library, but some tests may inherently fail to complete if scheduling is impeded by either a lack of sufficient kernel threads or a problem with cooperation.

Also note that thread libraries with better modularity (including layering instead of copy/paste re-implementation) can be more easily shifted from one flavor to another.
3.1 Virtual Timer Interrupts

Regardless of whether you deliver a thread library with user-space preemption, we expect you to implement “user-space timer interrupts” via an extension to the `swexn()` system call. Virtual timer interrupts are delivered to user threads via their `swexn()` handler with `ureg->cause` set to the newly-added `SWEXN_CAUSE_TIMER`. You should install a handler for a new system call, system call, `set_timer()`, at IDT slot `SET_TIMER_PERIOD_INT`.

Your virtual timer interrupts should have the following behavior:

1. When a thread is created via `thread_fork`, its timer is not running.
2. `exec()`, if successful, stops a thread’s timer.
3. The new system call, `void set_timer(int period)`, indicates that when the invoking thread is interrupted by the hardware timer `period` times a virtual timer interrupt becomes pending. Any non-positive value of `period` stops the thread’s timer.
4. Whenever a virtual timer interrupt is pending and a software exception handler is installed, a timer exception should be delivered “immediately” via the exception handler.
5. As is true of other exception causes, delivery of a virtual timer interrupt unregisters the handler.

Something to consider: should it be possible for a thread to “simultaneously” receive two software exceptions?

`timer_test1.c` and `station.c` have been provided for you to test user-space timer interrupt delivery.

3.2 Thread Library

- `int thr_init_multi(unsigned int size, unsigned int maxpar, int timer_period )` - This function initializes the thread library. The argument `size` specifies the amount of stack space which will be available to each thread using the thread library.

The argument `maxpar` indicates the maximum parallelism that this application expects from the thread library, which is often related to the number of kernel threads that the library uses. This value is a hint rather than a precise mandate, e.g., N:1 thread libraries inherently cannot comply.

The argument `timer_period` suggests the number of hardware timer interrupts that should elapse between preemptive context switches. Thread libraries that do not implement user-space preemption will not have much use for this parameter.

This function returns zero on success, and a negative number on error. For more information about assumptions and illegal conditions, see `thr_lib.pdf`. 
- **int thr_init( unsigned int size )** - For compatibility purposes, this function should behave substantially as it did in P2, but it should have results equivalent to a call to `thr_init_multi(size, m, t)`. We believe that `m = 8` and `t = 1` are plausible default values.

  For more information on desired behavior, see thr_lib.pdf.

- **int thr_create( void *(*func)(void *), void *arg )** - see thr_lib.pdf.

- **int thr_join( int tid, void **statusp )** - see thr_lib.pdf.

- **int thr_exit( void *status )** - see thr_lib.pdf.

- **thr_getid( void )** - see thr_lib.pdf.

- **thr_yield( int tid )** - see thr_lib.pdf.

- **All functions specified in cond.h** - see thr_lib.pdf.

- **All functions specified in mutex.h** - see thr_lib.pdf.

- **All functions specified in sem.h** - see thr_lib.pdf.

- **All functions specified in rwlock.h** - see thr_lib.pdf.

  For those sections which refer you to see the P2 handout, the specification is unchanged. However, the functions should be implemented with user-space threads instead of P2’s kernel threads.

### 3.3 Hand-in

For this project, your deliverables will be as follows:

- The new system call `set.timer()` will need to be added to your kernel, and its associated wrapper should be added to `libsyscall`.

- A thread library containing all of the specified functions (required)

- `README` etc.

### 4 Provided Tests

The tests provided in your handout can be divided into two categories: those that require preemptible threads and those that don’t. If you wish to turn in a P4 without preemptible threads, you should aim to pass all C tests. A P4 with preemptible threads should aim to pass both C and P tests.
4.1 C tests

- ack
- agility_drill
- beady_test
- cvar_test
- cyclone
- fib
- join_specific_test
- juggle
- mandelbrot
- multitest
- mutex_destroy_test
- racer
- rwlock_downgrade_read_test
- sieve
- switzerland
- thr_exit_join

4.2 P tests

In order to test involuntary thread switching, some of these tests rely heavily on needless busy waiting - don’t replicate this pattern in your own code!

- p4_startle
- roundest_robin
- relay

As usual, you are encouraged to write more of your own tests!
4.3 Actual Parallelism

Let’s say you implement an M:N thread library and run it on your kernel. Your kernel most likely is a single-processor implementation, so its maximum parallelism is 1. Concurrent activities such as kernel threads will be executed in a pseudo-parallel fashion due to timer interrupts, and your thread library may in turn switch one kernel thread between concurrent logical threads. But something will be missing: actual parallelism.

One way to achieve actual parallelism would be to upgrade your kernel to symmetric multiprocessing (“SMP”), but this takes a while (roughly one P4 time). Luckily, we have provided you with an alternative. As you may recall, during P2 we made available a multi-processor version of the reference kernel, called smpathos. Your tireless\(^4\) course staff has added `set_timer()` to `smpathos`. This means that you can run your thread library on a multi-processor machine, either in Simics or on the 410 crash box.\(^5\)

Due to reasons, the procedure for building your thread library on top of your kernel \textit{and} on top of `smpathos` is a little inelegant (perhaps if this P4 is used in a later semester it will be smoother, perhaps because you will be a TA and will sand off some of the rough edges).

1. Make sure that a relevant thread library builds and runs on top of your kernel. Call the root of that build directory `$P4$`.

2. Download a copy of the P2 tarball, expand it, and build it. Call the root of \textit{that} build directory `$P2$`.

3. In `$P2$`:
   \begin{enumerate}
   \item \textbf{make}
   \item \textbf{mv user XuserX}
   \item \textbf{mv 410user X410userX}
   \item \textbf{mv config.mk config.mkX}
   \item \textbf{ln -s $P4$/user user}
   \item \textbf{ln -s $P4$/410user 410user}
   \item \textbf{ln -s $P4$/config.mk config.mk}
   \item \textbf{echo `REFK=smpathos\_timer’ >> config.mk}
   \item \textbf{make} # ’‘make clean’’ would not be a good idea
   \item \textbf{simics46smpathos}
   \item \textbf{In addition to Simics, there’s also a crash box!}
   \end{enumerate}

\(^4\)Pretty tired, actually, but our hearts are in the work.
\(^5\)\url{http://www.cs.cmu.edu/~410/doc/crashbox.html}
5 Plan of Attack

1. Review P2 ink - try to understand the mistakes you made and sketch out an improved implementation. A solid P2 will provide a good foundation for P4.

2. Think quickly through the design of an N:1 thread library. In particular, consider thr_create(): it wouldn’t call thread_fork, of course. What would it need to do instead?

3. Take a moment to consider the nature of thr_yield() in this new kind of thread library and appreciate what makes it different from what you did in P2. Does this remind you of any code you have written?

4. How would cond_wait() work in an N:1 thread library?

5. Together with your partner, make a preliminary decision on the kind of P4 you wish to implement.

6. Design and plan when to implement the set_timer() system call.

7. Design and implement synchronization primitives. For condition variables, think about how the deschedule() and make_runnable() system calls might be “remade” within your user-space scheduler.

8. Design thr_create(), thr_exit(), and thr_join(). How do these functions interact or interfere with one another in your P4?

9. Take note of how much time is left until the deadline. Together with your partner, finalize your decision on the kind of P4 you wish to implement.

10. Inventory your kernel code looking for parts you could use in your thread library. Implement your ideas. For M:N, special attention should be paid to synchronization between kernel threads!

11. Write and test thr_create(). You should also implement thr_yield(), otherwise it may be hard to test your threads.

12. Once you’ve extensively tested your synchronization primitives, write thr_join() and thr_exit(). Ideally, they should depend on existing primitives. Take hints from your P2 ink!

13. What should happen if the user-space scheduler is empty? Update your implementation accordingly.

14. If you are implementing a preemptible thread library, test your user-space timer interrupts. Make sure they work independent of your thread library.

15. Put it all together, and debug.

16. Congratulations, you’re done!
# A Test Feature Table

Here is a table briefly summarizing the features of various tests we have provided.

<table>
<thead>
<tr>
<th>Test</th>
<th>N:1 co-op</th>
<th>N:1 preempt</th>
<th>M:N co-op</th>
<th>M:N preempt</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>p4_startle</td>
<td>vital</td>
<td></td>
<td>vital</td>
<td></td>
<td>Preemption to run threads once</td>
</tr>
<tr>
<td>roundest_robin</td>
<td>vital</td>
<td></td>
<td>vital</td>
<td></td>
<td>Preemption to run threads several times</td>
</tr>
<tr>
<td>relay</td>
<td>vital</td>
<td></td>
<td>vital</td>
<td></td>
<td>Preemption between creator and created</td>
</tr>
<tr>
<td>unfair</td>
<td></td>
<td>vital</td>
<td></td>
<td></td>
<td>User threads deschedule kthreads</td>
</tr>
<tr>
<td>switzerland</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
<td>Root thread stack size</td>
</tr>
<tr>
<td>agility_drill</td>
<td>vital</td>
<td>vital</td>
<td>vital</td>
<td></td>
<td>Mutexes</td>
</tr>
<tr>
<td>cyclone</td>
<td>vital</td>
<td>vital</td>
<td>vital</td>
<td></td>
<td>One parent &amp; one child &amp; repeat</td>
</tr>
<tr>
<td>join_specific_test</td>
<td>vital</td>
<td>vital</td>
<td>vital</td>
<td></td>
<td>One parent &amp; many children</td>
</tr>
<tr>
<td>thr_exit_join</td>
<td>vital</td>
<td>vital</td>
<td>vital</td>
<td></td>
<td>Child exits &amp; then is joined</td>
</tr>
<tr>
<td>cvar_test</td>
<td>vital</td>
<td>vital</td>
<td>vital</td>
<td></td>
<td>Condition variables</td>
</tr>
<tr>
<td>multittest</td>
<td>stress</td>
<td>stress</td>
<td>stress</td>
<td>stress</td>
<td>Runs the 5 tests above</td>
</tr>
<tr>
<td>mutex_destroy_test</td>
<td>vital</td>
<td>vital</td>
<td>vital</td>
<td></td>
<td>Illegal</td>
</tr>
<tr>
<td>rwlock downgrade_read_test</td>
<td>vital</td>
<td>vital</td>
<td>vital</td>
<td>vital</td>
<td>Downgrade &amp; then read</td>
</tr>
<tr>
<td>jugglee</td>
<td>stress</td>
<td>stress</td>
<td>stress</td>
<td>stress</td>
<td>Create/join + sleep; mutexes</td>
</tr>
<tr>
<td>ack</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
<td>Recursive computation &amp; multiple threads</td>
</tr>
<tr>
<td>jlb</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
<td>Recursive computation &amp; multiple threads</td>
</tr>
<tr>
<td>chow</td>
<td>stress</td>
<td>stress</td>
<td>stress</td>
<td>stress</td>
<td>ack + jlb + others &amp; repeated</td>
</tr>
<tr>
<td>sieve</td>
<td>stress</td>
<td>stress</td>
<td>stress</td>
<td>stress</td>
<td>Synch primitives; high thread count</td>
</tr>
<tr>
<td>racer</td>
<td>stress</td>
<td>stress</td>
<td>stress</td>
<td>stress</td>
<td>Synch primitives; scheduling visualization</td>
</tr>
<tr>
<td>beady_test</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
<td>Basic synch; some threads</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>stress</td>
<td>stress</td>
<td>stress</td>
<td>stress</td>
<td>Synch primitives; high thread count</td>
</tr>
<tr>
<td>bistromath</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
<td>Computation &amp; minimal threading</td>
</tr>
</tbody>
</table>