# A Scheme concurrency library

Takashi Kato
ktakashi@ymail.com

## Abstract

Concurrent programming is one of the most important techniques to use single or multi core CPUs efficiently. SRFI-18 defines primitive APIs for multithreading programming. However, using raw thread, mutex, or condition variables often causes unexpected behaviours. It is beneficial to construct an abstract layer on top of the SRFI to write more robust program. This paper describes a concurrency library built upon the multithreading SRFI and shows what would be the benefit to use it.

***Keywords*** Scheme, Concurrent

## 1. Introduction

The Scheme standards do not have neither multithreading nor concurrent functionality even the current latest standard, R7RS-small[1]. The only capability of writing such a program portably is using SRFI-18[2]. The SRFI defines APIs and types for multithreading primitives; thread, mutex, and condition variables. The following code is an example of using some of the APIs defined in the SRFI:

Listing 1: Example of SRFI-18

```
(import (rnrs) (srfi :18))

(define (fib n)
  (if (< n 2)
    n
    (+ (fib (- n 1))
       (fib (- n 2)))))

(define (async-fib n)
  (thread-start!
    (make-thread (lambda () (fib n)))))

(map thread-join!
     (list (async-fib 25)
           (async-fib 20)))
;; -> (75025 6765)
```

The example computes 2 Fibonacci numbers of 25 and 20 asynchronously by using 2 threads and retrieves the results by using `thread-join!` procedure.

Even in this small example, there are 2 possible problems. One is the sequential retrieval of the results of the thread executions. If the first thread has a heavier process than the other threads, then the retrieval would block the rest of process. In the example, the first `async-fib` procedure would take more time than the second one. The other problem is the possibility of thread count explosion. The `async-fib` procedure creates a thread when it is called. Thus, if it is called a large number of times, then the same number of threads are created. This contains 2 issues. The number of threads may reach the limit, and creation cost of thread may be expensive.

The first problem can be resolved by queue like inter-thread communications. If results of thread executions are pushed into a queue, then retrievers can take the results if it is already pushed or wait until one of the threads pushes its result into the queue. In this manner, blocking time would be less than retrieving all results at the same time.

The second problem is related to resource management. Each operating system has own limit of number of maximum threads, which might be per process or per operating system. For example, the maximum number of threads on Windows may depend on process' stack size. It would be around 2000 with combination of default stack size and thread creation option[1)]. If users can reuse created threads, then number of threads and its creation cost can be reduced.

To prevent Scheme users having these problems, we have created a library called `(util concurrent)`[2)] which is built on top of SRFI-18. It provides abstraction layers of concurrent programming such as shared queue, thread pool, and future and executor. The following sections describe how we implemented the library, then compares performance between using bare threads and our models.

## 2. Shared queue

There are variety of definition for shared queue[3)]. Here, shared queue is a queue like data structure whose enqueue and dequeue operations are done atomically. In our library, shared queue is a record which has usual queue structure fields[4)] and fields for synchronisation; mutex and condition variable. The `make-shared-queue`



[1)] https://blogs.msdn.microsoft.com/oldnewthing/20050729-14/?p=34773/

[2)] It can work on both R6RS and R7RS implementations, if they support SRFI-18 and some others.
Repository: https://github.com/ktakashi/scheme-concurrent

[3)] For example, IBM MQ has shared queue but not used for its local queue.
https://www.ibm.com/support/knowledgecenter/SSFKSJ_9.0.0/com.ibm.mq.pro.doc/q003640_.htm

[4)] c.f. *head*, *tail* and *size*

procedure creates a shared queue, and the `shared-queue-put!` and `shared-queue-get!` procedures are the enqueue and dequeue operations of shared queue. The following example shows how it works:

Listing 2: Shared queue

```
(import (rnrs) (util concurrent) (srfi :18))

(define shared-queue (make-shared-queue))

(define thread1
  (make-thread
    (lambda ()
      (list
        (shared-queue-get! shared-queue)
        (shared-queue-get! shared-queue)))))

(thread-start! thread1)

(define (push)
  (shared-queue-put! shared-queue 'wakeup!))

(for-each thread-start!
  (list (make-thread push)
        (make-thread push)))

(thread-join! thread1)
;; -> (wakeup! wakeup!)
```

The `shared-queue-put!` and `shared-queue-get!` ensure that queue mutations are done by atomically, thus it blocks all other operations. On the example, 2 threads are enqueueing their results in one shared queue. If the queue operations is not done by atomically, then one of the results may be lost.

If `shared-queue-get!` is called on empty queue, then the procedure waits until the queue gets an element. To avoid unwanted blocking, the procedures also accept optional arguments; *timeout* and *timeout value*. If the *timeout* argument is passed, then the procedures only wait for specified period. And the *timeout value* is returned, when the *timeout* period has passed, but the queue is not available yet.

Both `shared-queue-put!` and `shared-queue-get!` have similar sequence to achieve its atomicity. As a precondition, shared queue has a mutex and condition variable. When the procedures are called, it locks the mutex of the queue then operates enqueue or dequeue. After the operation is done, it calls `condition-variable-broadcast!` on the condition variable and releases the mutex. If the `shared-queue-get!` is called for an empty queue, then the procedure waits on the condition variable either until the queue gets a new element or specified *timeout value* is passed.

## 3. Thread pool

Thread pool is a concept which makes threads reusable. In our library, it is implemented as a container which holds specified number of threads which never stop until stop operation is explicitly invoked. One of the benefits of reusing threads is reducing overhead of thread creation. Depending on implementations, creating a thread is usually not a cheap operation and may consume a lot of resources especially when the number of threads are huge. This section describes the design of the thread pool implemented on the library.

A thread pool initialises the given number of threads during creation. Each thread has a shared queue as a channel between the pool and the thread[5)]. The threads wait on their queue until a task, which is a procedure with 0 argument, is enqueued. When a task is pushed to the pool, it chooses one of the threads and enqueues the task into the thread's shared queue. The chosen thread dequeues the task from its shared queue and executes it. If there are more tasks enqueued, then the thread repeats the execution process until the queue is empty. The following example shows creating a thread pool which holds 10 threads:

Listing 3: Thread pool

```
(import (rnrs) (util concurrent))

(define thread-pool (make-thread-pool 10))

(thread-pool-push-task! thread-pool
  (lambda ()
    (display (fib 30)) (newline)))

(thread-pool-wait-all! thread-pool)
;; -> returns unspecified value
;;    and prints 832040
```

Thread pool reuses created threads, so threads cannot return values. If results of pushed tasks are required, then the task itself needs to handle it by using, for example, shared queue.

### 3.1 Choosing thread

Since we decided to have shared queue as a channel per thread, choosing a proper thread from a thread pool is important to make the thread pool work efficiently. If a thread pool chooses the same thread for each task, then it is the same as using a single thread. The easiest way to distribute to tasks evenly is checking the number of queued tasks per queue and taking the least queued thread. This takes $\mathrm{O}(n\,\mathrm{O}(m))$ where $n$ is the number of threads and $\mathrm{O}(m)$ denotes the cost of counting the length of shared queue. Our shared queue implementation takes $\mathrm{O}(1)$ to return its length, so the choosing process would take $\mathrm{O}(n)$.

The checking the number of enqueued tasks though out the threads would not take the worst case most of the time. However, if the first $n-1$ threads have heavy tasks and $n$th thread is always free, then it would always take the worst case. To avoid such situations, we added an extra shared queue, called idling queue, to thread pool. This shared queue holds idling threads, which does not have any task in its queue, of the thread pool, so dequeuing the queue is sufficient if there is any thread idling. This makes the best case complexity $\mathrm{O}(1)$.

### 3.2 Error handling

Handling an error at thread pool level is simple. Any managed thread should not propagate it to thread pool itself, otherwise error signalling thread would terminate and cannot be reused. Thus, whenever tasks signal errors, threads should catch it and discard it. So the basic strategy of error handling is making tasks responsible for it.

Even though it is tasks' responsibility to handle errors, the `make-thread-pool` procedure accepts optional argument error

[5)] The reason why we choose to let threads have own queue is to avoid `abandoned-mutex-exception`. If all threads share one queue and when one of the threads is terminated, `abandoned-mutex-exception` might occur.

handler, which should accept 1 argument, for convenience. This error handler can be used for generic logging or diagnosis[6].

### 3.3 Thread termination

Terminating a thread is not an ideal operation in any situation. However, if managed threads got into dead lock, then the only possibility to recover from the situation without stopping the whole process is terminating the locked threads. The following example shows how to do it:

Listing 4: Terminating thread

```
(import (rnrs) (util concurrent))

(define thread-pool (make-thread-pool 10))

(define id
  (thread-pool-push-task! thread-pool
    (lambda ()
      (shared-queue-get!
        (make-shared-queue)))))

;; Terminate it!
(thread-pool-thread-terminate!
  thread-pool id)
```

Each thread gets its own id which is an exact non-negative integer[7]. The `thread-pool-push-task!` procedure returns the id of the thread that the task is pushed. Using the returned id and the `thread-pool-terminate-thread!` procedure makes thread termination of managed thread possible.

When thread termination is occurred on a thread pool, the thread pool first locks idling and channel queue and cleans up the channel queue to avoid abandoned mutex exception. Then, it creates a new thread and puts the thread into its pool with the same condition as terminated thread except the tasks. This avoids creating an empty thread pool and makes it possible to continue working.

## 4. Future and executor

Future and executor are higher level concepts inspired by JSR166[5]. A future is an object, returned by an executor[8], which contains a task and a channel. Its task is executed on the executor asynchronously and the result value is stored into its channel. An executor is a concurrent resource manager. The library provides 2 types of executors, `<thread-pool-executor>` and `<fork-join-executor>`. The `<thread-pool-executor>` uses thread pool as underlying resource management. The `<fork-join-executor>` simply creates a thread per task. When a task is submitted to an executor by `executor-submit!`, then a future, which is executed on the executor, is returned. When the `future-get` procedure is called with the future, then it waits until the task is finished and returns the result of the task, or it returns the result immediately if the task is already finished.

Listing 5: Executor

```
(import (rnrs) (util concurrent))

(define thread-pool-executor
  (make-thread-pool-executor 10))

(define future
 (executor-submit! thread-pool-executor
   (lambda () 'executed)))

(future-get future) ;; -> executed
```

The `<thread-pool-executor>` is implemented on top of the thread pool described previous section. Using the executor makes retrieving results of tasks easier and gives users more flexible resource management. The `make-thread-pool-executor` accepts optional argument `reject-handler` which controls rejection strategy. Rejection strategies are control the behaviour of executors when tasks are submitted, and there is no thread available[9] at the moment.

The library provides the following 4 predefined reject handlers; `abort-rejected-handler`, `terminate-oldest-handler`, `wait-finishing-handler`, and `push-future-handler`. The `abort-rejected-handler` signals an error if there is no thread available. If `reject-handler` is not specified, then this handler is used as the default value. The `terminate-oldest-handler` terminates the thread, which is executing the oldest task, using the `thread-pool-thread-terminate!` procedure. The `wait-finishing-handler` waits until one of the threads is available or raises an error when specified retry count is exceeded[10]. The `push-future-handler` pushes given task to underlying thread pool by using `thread-pool-push-task!` procedure which chooses the least engaged thread.

The `executor-available?` procedure can be used to check executors' availability. Using this procedure gives users capability of writing scripts: It submits tasks while an executor is available, then retrieves the results of the tasks and processes the results. After it finishes the process of the result values, then it starts submitting tasks again until it submits all the tasks.

## 5. Benchmarks

This section shows one of the benefits of using this concurrent library. We benchmarked 4 concurrency models; no management, manual thread management, using thread pool, and future and executor. Each script calculates Fibonacci numbers of in between 20 and 25, and all scripts are executed 3 times with different numbers of invocation of Fibonacci number calculation. The numbers of invocation are 100, 1000, and 10000.

Each benchmark script has `run` procedure which takes an argument, *n**. *n** is a list whose length is the same as invocation count, and elements are arguments to calculate Fibonacci number. All benchmark script consumes at least $\mathrm{O}(n)$ memory space where *n* is invocation count to emulate retrieving result values of threads.

Listing 6: No management

```
(define (run n*)
  (for-each thread-join!
   (map (lambda (n)
          (thread-start!
           (make-thread (heavy-task n))))
        n*)))
```

[6] Of course, this can also be used more specific error handling such as releasing resource. For such cases, it's better to use condition system defined in R6RS[3] or SRFI-35[4]

[7] So that users can terminate a thread from other thread, process, or using remote REPL.

[8] It can also be created calling its constructors, but basic usage would be combination of executor.

[9] This means all managed threads on the thread pool are running.

[10] Each retry waits 0.5 second.

The no management script simply creates thread for each Fibonacci calculation and joins it. For this implementation, the invocation number and thread count are the same.

Listing 7: Manual management

```
(define (run n*)
  (let loop ((n* n*)
             (n 0)
             (t* '())
             (r '()))
    (cond ((null? n*)
           (apply append
            (cons (map thread-join! t*) r)))
          ((= n 10)
           (loop (cdr n*)
                 0
                 '()
                 (cons (map thread-join! t*)
                       r)))
          (else
           (loop (cdr n*)
                 (+ n 1)
                 (cons (thread-start!
                        (make-thread
                         (heavy-task
                          (car n*))))
                       t*)
                 r)))))
```

The manual management script keeps executing thread count to $10^{11)}$. When the created thread count reaches 10, then it retrieves the result from the threads.

Listing 8: Thread pool

```
(define (run n*)
  (let ((tp (make-thread-pool 10))
        (sq (make-shared-queue)))
    (let loop ((n* n*))
      (if (null? n*)
          (thread-pool-release! tp)
          (let ((p (heavy-task (car n*))))
            (thread-pool-push-task! tp
             (lambda ()
               (shared-queue-put! sq (p))))
            (loop (cdr n*)))))))
```

The thread pool script uses a thread pool which has $10^{11)}$ worker threads. It first creates the thread pool and a shared queue which is used to retrieve the results of calculation, then pushes tasks to the thread pool. After all tasks are pushed, then it waits and releases the thread pool.

[11)] 10 is an arbitrary number, but it should be smaller than the number of invocations. Otherwise, the thread creation would be bigger than the no management.

Listing 9: Future and executor

```
(define (run n*)
  (let ((e (make-thread-pool-executor 10)))
    (let loop ((n* n*) (f* '()))
      (cond ((null? n*)
             (for-each future-get f*)
             (shutdown-executor! e))
            ((executor-available? e)
             (let ((f (executor-submit! e
                        (heavy-task
                          (car n*)))))
               (loop (cdr n*) (cons f f*))))
            (else
             (for-each future-get f*)
             (loop n* '()))))))
```

The future and executor script uses thread pool executor whose underlying thread pool contains $10^{11)}$ worker threads. The script submits tasks while the executor is available. When executor is not available, then it retrieves the result from the futures returned by the executor. After the retrieval, it starts submitting until it consumes all the arguments for the calculation.

This benchmark was executed on Ubuntu 14.04, 16GB RAM, Intel(R) Core(TM) i5-2520M @ 2.50GHz x 4 and 32 bit Cygwin (on Windows 7 Home Premium, 8GB RAM, Intel(R) Core(TM) i5-3317U @ 1.70GHz 1.70GHz). Executed implementations are Sagittarius 0.7.1, Guile 2.0.11, Gauche 0.9.4 and Chibi Scheme 0.7.3. The first 2 were run as R6RS and the other 2 were run as R7RS.

Table 1: Sagittarius 0.7.1

Ubuntu 14.04

| # of invocations | 100 | 1000 | 10000 |
|---|---|---|---|
| no management | 0.179s | 1.814s | 19.771s |
| manual management | 0.217s | 2.313s | 25.991s |
| thread pool | 0.296s | 1.908s | 18.062s |
| future and executor | 0.34s | 2.25s | 21.74s |

Cygwin

| # of invocations | 100 | 1000 | 10000 |
|---|---|---|---|
| no management | 0.268s | 2.739s | Out of Memory |
| manual management | 0.257s | 2.734s | 27.492s |
| thread pool | 0.332s | 2.310s | 21.346s |
| future and executor | 0.34s | 2.53s | 25.02s |

Table 2: Guile 2.0.11

Ubuntu 14.04

| # of invocations | 100 | 1000 | 10000 |
|---|---|---|---|
| no management | 0.29s | 3.22s | 28.38s |
| manual management | 0.27s | 2.70s | 24.82s |
| thread pool | 0.24s | 2.56s | 21.33s |
| future and executor | 0.26s | 2.58s | 26.30s |

Cygwin

| # of invocations | 100 | 1000 | 10000 |
|---|---|---|---|
| no management | 0.42s | Out of Memory | Out of Memory |
| manual management | 0.43s | Out of Memory | Out of Memory |
| thread pool | 0.33s | 3.06s | 30.91s |
| future and executor | 0.37s | 3.80s | Out of Memory |

Table 3: Gauche 0.9.4

Ubuntu 14.04

| # of invocations | 100 | 1000 | 10000 |
|---|---|---|---|
| no management | 0.190s | 1.780s | 18.129s |
| manual management | 0.231s | 2.286s | 22.487s |
| thread pool | 0.192s | 1.794s | 17.867s |
| future and executor | 0.23s | 2.39s | 22.56s |

Cygwin

| # of invocations | 100 | 1000 | 10000 |
|---|---|---|---|
| no management | 0.269s | 2.735s | Out of Memory |
| manual management | 0.298s | 2.665s | 26.692s |
| thread pool | 0.299s | 2.116s | 20.269s |
| future and executor | 0.32s | 2.47s | 24.04s |

Table 4: Chibi Scheme 0.7.3

Ubuntu 14.04

| # of invocations | 100 | 1000 | 10000 |
|---|---|---|---|
| no management | 0.545s | 5.820s | 64.945s |
| manual management | 0.438s | 4.487s | 44.951s |
| thread pool | 0.610s | 5.307s | 52.893s |
| future and executor | N/A | N/A | N/A |

Cygwin

| # of invocations | 100 | 1000 | 10000 |
|---|---|---|---|
| no management | 0.695s | 7.083s | 73.904s |
| manual management | 0.618s | 6.134s | 61.386s |
| thread pool | 0.658s | 6.178s | 60.489s |
| future and executor | N/A | N/A | N/A |

Except Chibi Scheme on Ubuntu, the result shows that when number of threads is small, there are not much difference among these benchmarks. When it gets bigger, the thread pool is the fastest. So at some point, the overhead of thread creation got bigger than initialising thread pool. This, of course, depends on implementation and environment but the benchmark shows the threshold is around 1000 threads.

Using thread pool and future and executor also shows less memory consumption. On Cygwin, 3 out of 4 implementations signalled out of memory error on no management benchmark with 10000 threads. Guile even signalled 1000 threads benchmark of both no management and manual management. 384 Megabyte[12] is the default amount of memory that processes can use on 32 bit Cygwin environment. So creating 10000 threads would consume as close as or more than the default amount of memory and cause out of memory error.

Future and executor scored slightly worse performance than thread pool on 3 out of 4 implementations[13]. This would be overhead of future creation and explicit blocking. Future and executor blocks main thread up to 1/10 of invocation count when the executor is not available while thread pool model blocks only once. And it also impacted GC on Guile since it returned out of memory error on 10000 invocations.

Considering this result, if scripts need to handle indefinite number of tasks simultaneously, then thread pool would provide better performance and stability.

[12] Cygwin User's Guide.
Chapter 2. Setting Up Cygwin - Changing Cygwin's Maximum Memory
URL https://www.cygwin.com/cygwin-ug-net/setup-maxmem.html

[13] Chibi Scheme went either infinite waiting or segmentation fault. We are not sure which side of issue it is.

## 6. Conclusion

We described the basic implementation strategies of our concurrency library and showed the performance impact of using it. Concurrent programming is not easy. Users often need to consider a lot of things such as resource management and synchronisation. However, these are often not crux of what programmers want to do. We believe using the library reduces these trivial problems and make programmers concentrate to resolve main problems.

## A. Benchmark script

The `counts` file defines the invocation numbers as a list. For our benchmarks, it has `(100 1000 10000)`

Listing 10: Benchmark procedures

```
(import (rnrs)
        (only (srfi :1) iota)
        (except (srfi :18)
                raise with-exception-handler)
        (util concurrent)
        (time))

(define (fib n)
  (if (< n 2)
    n
    (+ (fib (- n 1))
       (fib (- n 2)))))

(define (heavy-task n) (lambda () (fib n)))

(define counts
  (call-with-input-file "counts" read))

(define n**
  (map (lambda (count)
         (fold-left
          (lambda (acc o)
            (cons (+ (mod (length acc) 6) 20)
                  acc))
          '() (iota count)))
       counts))

(for-each (lambda (count n*)
            (time (run count n*)))
          counts n**)
```

## References


[1] Alex Shinn, John Cowan, and Arthur A. Gleckler, editors. Revised[7] report on the algorithmic language Scheme. July 2013.
URL http://scheme-reports.org/

[2] Marc Feeley. SRFI-18: Multithreading support. March 2003
URL http://srfi.schemers.org/srfi-18/

[3] Michael Sperber, R. Kent Dybvig, Mathew Flatt, and Anton van Straaten, editors. Revised[6] report on the algorithmic language Scheme. September 2007.
URL http://www.r6rs.org/

[4] Richard Kelsey and Michael Sperber. SRFI-35: Conditions. December 2002
URL http://srfi.schemers.org/srfi-35/

[5] Doug Lea. Concurrency Utilities. September 2004
URL https://www.jcp.org/en/jsr/detail?id=166