Thread Pools and Synchronisation with Condition Variables

Troels Henriksen

18th of November, 2019

But first! An aside on defensive programming.

The task

We are asked by assignment/boss/divine inspiration to implement this prototype.

```
int transducers_link_1(stream **out,
                       transducers_1 t, const void *arg,
                       stream* in);
```

How much error checking do we do?

. . .

"An attempt to connect a stream that has already been used as the input to another transducer should cause the API function to return an error."

Bad

```
int transducers_link_1(stream **out,
                       transducers_1 t, const void *arg,
                       stream* in) {
  if (out == NULL || in == NULL || in->linked) {
    return 1;
  }
}
```

Why bad?

. . .

- Inefficient.

- Turns *bugs* into *errors*.

Good interfaces have clear responsibilities

```
int transducers_link_1(stream **out,
                       transducers_1 t, const void *arg,
                       stream* in);
```

- *Precondition*: The *caller* must provide non-NULL `out` and `in`.
- *Postcondition*: A `stream` is put in `*out` (and various other things).

Violations of pre- and post-conditions are *bugs*, not *errors*.

Errors are things that can happen to correct programs, e.g. missing files, out of memory, or invalid user commands.

Use return codes to detect errors and `assert()` to detect bugs

```
int transducers_link_1(stream **out,
                       transducers_1 t, const void *arg,
                       stream* in) {
  assert(out != NULL);
  assert(in != NULL);
  if (in->linked) {
    return 1;
  }
}
```

- `assert()`s can have false positives - not all bugs can be detected (sadly).
- Errors are always checked fully (barring bugs...).

Let's thread a program!

The Fibonacci Function

```
int fib (int n) {
  if (n < 2) {
    return 1;
  } else {
    return fib(n-1) + fib(n-2);
  }
}
```

- This is the slow formulation, but it will give us something to work with.
- Goal: apply fib() to all lines of a file

The getline() function

```
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

- Allocates memory for us (`lineptr`), but it is our responsibility to `free()` it when we are *completely* done.
- Returns the size of the line, and stores the size of the memory underlying it in `n`.
- *We* must free the line when we are done with `getline()`.

Using getline()

```
int main() {
  char *line = NULL;
  ssize_t line_len;
  size_t buf_len = 0;

  while ((line_len = getline(&line, &buf_len, stdin)) != -1) {
    int n = atoi(line);
    printf("fib(%d) = %d\n", n, fib(n));
  }

  free(line);
}
```

The atoi() function

```
int atoi(const char *nptr);
```

- Returns integer represented by a string.

Is it fast?

```
$ ./fibs < fibs-huge.input
fib(40) = 165580141
fib(41) = 267914296
fib(42) = 433494437
fib(43) = 701408733
fib(45) = 1836311903
```

```
real    0m5,902s
user    0m5,886s
sys     0m0,000s
```

- Depends.

Could it be faster?

- Yes - this program uses only a single thread, and my machine has eight cores.

One thread per line

The thread function

```
void* fib_thread(void* arg) {
  char *line = arg;
  int n = atoi(line);
  printf("fib(%d) = %d\n", n, fib(n));
  free(arg);
  return NULL;
}
```

Changes to main()

```
int i = 0;
pthread_t threads[200000]; // arbitrary

while ((line_len = getline(&line, &buf_len, stdin)) != -1) {
  pthread_create(&threads[i], NULL, fib_thread, strdup(line));
  i++;
}

for (int j = 0; j < i; j++) {
  pthread_join(threads[j], NULL);
}
```

- Note the `strdup()` - this copies the line to avoid a race condition.

Is it faster?

```
$ time ./fibs-mt > /dev/null < fibs-huge.input
```

```
real    0m3,956s
user    0m8,354s
sys     0m0,004s
```

Looks good, but…

```
$ time ./fibs > /dev/null < fibs-verytiny.input

real    0m0,007s
user    0m0,007s
sys     0m0,001s

$ time ./fibs-mt > /dev/null < fibs-verytiny.input

real    0m0,189s
user    0m0,045s
sys     0m0,222s
```

- Spawning a thread is *expensive* (relatively).

Thread Pools

Amortising thread startup cost

- It is often too slow to start a new thread for every piece of work.
- For compute-bound work, we only need one thread per CPU core.

Solution: thread pools

- A *thread pool* is a collection of *worker threads* that wait for tasks.
- When a task is submitted, a thread is awoken, performs the task, then goes back to waiting for more.

Complex topic

- How big is the pool? How flexible? Do we use thread affinity?
- *We will only lightly touch on these concerns in the following.*

Creating threads for the pool is easy

```
// The number of processors.
int num_threads = sysconf(_SC_NPROCESSORS_ONLN);

// Make space for that many threads.
pthread_t *threads = malloc(num_threads*sizeof(pthread_t));

// Then launch them.
for (int i = 0; i < num_threads; i++) {
  pthread_create(&threads[i], NULL, worker, NULL);
}
```

But how do we submit work?

- Pipes would not work here, because multiple threads would read from the same pipe.
- A line of input is bigger than one byte.

Global shared variables

```
// If not NULL, a line is ready to be processed.
char *volatile line = NULL;

// Lock before accessing 'line'.
pthread_mutex_t line_mutex = PTHREAD_MUTEX_INITIALIZER;

// If 1, threads should shut down.
volatile int die = 0;
```

The thread function

```
void* worker(void* arg) {
  arg=arg;
  int done = 0;

  while (!done) {
    char *my_line = NULL;
    assert(pthread_mutex_lock(&line_mutex) == 0);

    if (line == NULL && die) {
      done = 1;
    }
```

```
    if (line != NULL) {
      my_line = line;
      line = NULL;
    }

    assert(pthread_mutex_unlock(&line_mutex) == 0);

    if (my_line != NULL) {
      int n = atoi(my_line);
      printf("fib(%d) = %d\n", n, fib(n));
      free(my_line);
    }
  }

  return NULL;
}
```

The line-reading loop

```
while ((line_len = getline(&my_line, &buf_len, stdin)) != -1) {
  int done = 0;
  while (!done) {
    assert(pthread_mutex_lock(&line_mutex) == 0);
    if (line == NULL) {
      line = strdup(my_line);
      done = 1;
    }
    assert(pthread_mutex_unlock(&line_mutex) == 0);
  }
}

die = 1;
```

Synchronisation by busy-waiting

- Worker threads spin in a lock/unlock-loop waiting for `line` to be non-NULL.
- The `main()` function spins in a lock/unlock-loop waiting for `line` to be NULL.

This is wasteful!

- When a worker thread sets `line` to NULL, it should *signal* the main thread that it can now store a new line.
- Similarly, the main thread should signal the worker threads when a line becomes available.

This is where we use condition variables

Condition variables

Initialisation

```
pthread_cond_t line_cond = PTHREAD_COND_INITIALIZER;
```

Signaling

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Waiting

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
```

- Blocks until another thread calls `pthread_cond_signal()`.
- The `mutex` *must* be locked when we call `pthread_cond_wait()`.
- Will be unlocked while the thread sleeps, and locked again when `pthread_cond_wait()` returns.
- *Spurious wakeups* may occur. ("MESA semantics".)

Using condition variables in the worker threads

```
void* fib_thread(void* arg) {
  arg=arg;
  int done = 0;

  while (!done) {
    char *my_line = NULL;
    assert(pthread_mutex_lock(&line_mutex) == 0);
```

```
    if (line == NULL && !die) {
      pthread_cond_wait(&line_cond, &line_mutex);
    } else if (line == NULL && die) {
      done = 1;
    } else if (line != NULL) {
      my_line = line;
      line = NULL;
      pthread_cond_broadcast(&line_cond);
    }

    assert(pthread_mutex_unlock(&line_mutex) == 0);

    if (my_line != NULL) {
      int n = atoi(my_line);
      printf("fib(%d) = %d\n", n, fib(n));
      free(my_line);
    }
  }

  return NULL;
}
```

And in the main thread

```
while ((line_len = getline(&my_line, &buf_len, stdin)) != -1) {
  int done = 0;
  while (!done) {
    assert(pthread_mutex_lock(&line_mutex) == 0);
    if (line == NULL) {
      line = strdup(my_line);
      pthread_cond_signal(&line_cond);
      done = 1;
    } else {
      pthread_cond_wait(&line_cond, &line_mutex);
    }
    assert(pthread_mutex_unlock(&line_mutex) == 0);
  }
}
```

- We still have the while-loop, but now it likely runs for much fewer iterations.

Another alternative: futures

A *future* (sometimes *promise*) is a value that is being computed asynchronously. We may ask for the *value* of the future, which will block until it is ready.

- Not supported directly by POSIX threads.
- ... but `pthread_join()` is almost this model if you squint a bit.

Pseudocode for Fibonacci with futures:

```
def fib(n):
  if n < 2:
    return 1
  x = future fib(n-1)
  y = future fib(n-2)
  return x.get() + y.get()
```

Why futures?

- A future may be evaluated in parallel, thus speeding up our program.
- They may also do other blocking non-CPU tasks, like network requests.
- **Most importantly:** Futures, if used correctly, are **deterministic**.
- (And they are not that hard to use correctly.)

Futures are probably the simplest way to get a bit of parallelism or concurrency in your programs.