Communicating Sequential Processes

CSCI 5828: Foundations of Software Engineering
Lecture 28 — 12/04/2014

Goals

- Cover the material in Chapter 6 of the Concurrency Textbook
  - Communicating Sequential Processes
    - channels, go routines
    - lots of concurrency, no parallelism (useful?)

Communicating Sequential Processes (I)

- An approach to concurrency that consists of
  - concurrently executing entities
  - communicating by sending “messages” to each other
- This is similar to the actor model. In that model, you had
  - concurrently executing entities (actors)
  - communicating by sending “messages” to each other
- So, what’s the difference?
  - With the actor model, the focus was on actors and their associated messages
  - With CSP, the focus is on the communication channels themselves

Communicating Sequential Processes (II)

- We spend time designing what channels are present in our system and what will be passed over each one
    - Any entity that has access to the channel can write to it and read from it
- In contract, with actors, we spent time designing the types of actors present and what messages they processed
    - In order to send a message, you needed to have a particular destination in mind: “I’m sending THIS message to THAT actor”
- With CSP, you decouple the sender and the receiver. When you add a message to a channel, you don’t necessarily know (or care) where it’s going

Communicating Sequential Processes (III)

- Programs that use CSPs enable concurrency
  - any “go routine” that is not blocked on a channel will be executed and allowed to make progress
    - as soon as it becomes blocked, the thread that was executing it can “drop it” and go execute a “go routine” that isn’t blocked
- You might get “some” parallelism with this approach but not a lot. Instead, a single thread can “multiplex” thousands of go routines and instead ensure that “logically concurrent entities” are making progress via time slicing

Basics: Channels (I)

- We'll be looking at a CSP implementation in Clojure via core.async
  - CSPs came back in vogue after they were used as the basis of concurrency in Google's systems programming language know as Go
- The core element of the CSP approach is a channel
  - A channel is conceptually a thread-safe queue
    - Any entity can add messages or retrieve messages from a channel without worrying about interference
  - However, by default, a channel is synchronous
    - writing to a channel blocks until something reads from it

Basics: Channels (II)

- Creating a synchronous channel
  - `(def c (chan))`
- Reading from a channel
  - `(<!! c)`
    - => returns the next element of the queue or blocks
- Writing to a channel
  - `(>!! c value)`
    - => writes an element to the queue if a reader is waiting or blocks

Basics: Channels (III)

- Example: “reads” block until a “write” occurs
  - `(def c (chan))`
  - `(thread (println "Read:" (<!! c) "from c”))`
  - `(>!! c "Hello thread")`
- Prints:
  - `Read: Hello thread from c`
  - `nil`
- The former is printed by a background thread that was blocked on the queue
  - The latter is the return value of the `>!!` function.

Basics: Channels (IV)

- Example: “writes” block until a “read” occurs
  - `(def c (chan))`
  - `(thread (>!! c "Hello") (println "Write completed"))`
  - `(<!! c)`

- Prints:
  - `Write completed`
  - `“Hello”`

- The former is printed by a background thread that was blocked on the queue
  - The latter is the return value of the `<!!` function.

Buffered Channels (I)

- If we want to avoid the blocking behavior of synchronous queues, we can create a buffered queue (of fixed size)
    - (def bc (chan 5))
- As long as there are slots available, writes to the channel are non-blocking
    - As soon as the slots fill up, then a write will block until a slot is available
- Reads from the channel still block until an element is available
    - UNLESS the channel has been closed; then a read will return `nil`
    - If a read was blocked on a channel that was closed, the read will immediately “unblock” and return nil

Buffered Channels (II)

- Example
  - `(def bc (chan 5))`
  - `(>!! bc 0)`
  - `(>!! bc 1)`
  - `(close! bc)`
  - `(<!! bc) => 0`
  - `(<!! bc) => 1`
  - `(<!! bc) => nil`

Dealing with Full Buffers (I)

- Even with buffered channels, you can encounter blocking reads/writes
    - You can create large buffers, but you still have a chance of blocking on the channel
- Why not allow arbitrarily large buffers?
    - You're just trading one class of "errors" for another
        - In this case, "out of memory" errors if you have a bug in your system that fills up channels without something consuming those messages
- Instead, CSPs would rather you "deal with the problem now"
    - what should your program do when the buffer is full?
        - is blocking okay? is it not? Make this explicit in your design

Dealing with Full Buffers (II)

- Having said that, CSPs offer one additional approach you can use
  - ***but only if you don't care about processing all elements on the queue***
- A dropping buffer will "drop" (i.e. delete) any items added to the channel once it is full
  - `(def dc (chan (dropping-buffer 5)))`
  - `(onto-chan dc (range 0 10))`
  - `(<!! (async/into [] dc)) => [0 1 2 3 4]`
- A sliding buffer will replace existing items on the queue with new ones
  - `(def sc (chan (sliding-buffer 5)))`
  - `(onto-chan sc (range 0 10))`
  - `(<!! (async/into [] sc)) => [5 6 7 8 9]`
- I'll describe `onto-chan` and `async/into` in a moment

Dealing with Full Buffers (III)

- With dropping buffers and sliding buffers, your “reads” and “writes” will never block but you may lose messages on the queues
    - For some applications, this may be acceptable
        - Think: running averages on streaming data
    - If not, then you’ll have to deal with blocking (which is fine, as we will see)

- Having said THAT, you still have to watch our for deadlock
    - If you have all of your “go routines” blocked on reads, that’s DEADLOCK
        - likewise if you have all of your “go routines” blocked on writes

Working with Buffers (I)

- If you want to take a sequence and write all of it to a channel, you can do something like this

```
(defn writeall!! [ch coll] (doseq [x coll]
    (>!! ch x))
  (close! ch))
```

- or use core.async's native function: `onto-chan`
- If you want to read everything off of a channel into a collection, you can do

```
(defn readall!! [ch]
  (loop [coll []]
    (if-let [x (<!! ch)]
      (recur (conj coll x))
      coll)))
```

- or use core.async's native function: `async/into`

Working with Buffers (II)

- Examples
  - `(def ch (chan 10))`
  - `(writeall!! ch (range 0 10))`
  - `(readall!! ch)`
    - `=> [0 1 2 3 4 5 6 7 8 9]`
- Using onto-chan and async/into
  - `(def ch (chan 10))`
  - `(onto-chan ch (range 0 10))`
  - `(<!! (async/into [] ch))`
    - `=> [0 1 2 3 4 5 6 7 8 9]`
- Note: async/into returns a **channel**, not a collection
  - which is why we had to call `<!!` on it to see the final result

Go Blocks: Background & Motivation

- When discussing java.util.concurrent, we advocated the use of thread pools
  - over the direct use of threading primitives
- The thread pool can deal with the creation of threads and assigning tasks to threads
- The use of thread pools is great but not a panacea
  - As we saw with the “counting files” example, you can deadlock a thread pool if your tasks perform operations which block
    - As each task blocks its thread is no longer available to the pool
      - If enough tasks block, then you have deadlocked the pool
- You can get around this by redesigning your tasks or by using a “fork-join” pool that can perform “work stealing”: blocked tasks can lose their threads, allowing the thread to work on other tasks

Go Blocks: Background & Motivation

- CSPs use a similar technique to “work stealing”
  - Tasks are specified using “go blocks” that
    - specify the sequential logic that needs to be performed for a task
  - These blocks are then compiled into a state machine that allows that sequential logic to be implemented in an event-driven manner
    - if a statement in the state machine blocks, the state machine is “parked” and its thread is freed up to execute “go routines” that are not blocked

(Very Simple) Example

```
(def ch (chan)
(go
  (let [x (<! ch)
        [y (<! ch)]
    (println “Sum:” (+x y))))
```

- Here, <! is the “parking version” of <!!
  - Read from the channel but if I block, then park the state machine
- >! is the “parking version” of >!!
  - Write to the channel but park the state machine if I block on the write
- core.async will not allow you to use >! and <! outside of a go block

The point of this design? Efficiency!

- Go blocks are very efficient
  - A single thread can execute 10K go blocks at once
    - time slicing between them; if one blocks, it switches to another
  - Parallelism can be achieved with this approach
    - You have a thread pool of a fixed size and you assign an “unblocked” go routine to the next available thread
    - This means a go block’s state machine can park off of one thread and resume on another thread

How efficient? (I)

- The book demonstrates using this function
  - `(defn go-add [x y]`
    - `(<!! (nth (iterate #(go (inc (<! %))) (go x)) y)))`
- go-add takes advantage of the fact that a “go block” returns a channel with the result of the go block written to it
  - this allows Clojure to compose series of go blocks in an efficient manner
- Let’s break this down
  - `#(go (inc (<! %)))` — an anonymous function: channel -> channel
  - `(go x)` — creates a channel with the value x written to it
  - `(iterate <function> <value>)` — creates a lazy sequence `(x f(x) f(f(x)) …)`
  - `(nth <sequence> y)` — retrieve the yth element of the sequence
  - `(<!! <chan>)` — retrieve the result of the channel returned by `nth`

How efficient? (II)

- `(time (go-add 10 100000))`
- `Elapsed time: 791.9 msecs`
- `100010`

- 100,000 channels created and executed in less than a second
- Need to watch memory if you need to go higher than this

Channels as Sequences

- Channels represent an ordered set of values
  - And thus are similar to sequences
- Like sequences, we can use them in higher-order functions like map and reduce
- Example
  - `(def ch (to-chan (range 0 1000)))`
  - `(<!! (async/into [] (map< (partial * 2) (filter< even? ch))))`
  - `[0 4 8 12 16 20 24 28 32 … 1988 1992 1996]`
- map< and filter< are versions of these functions that operate on channels rather than sequences. They return channels so they can be composed

Finding Primes

- The book ends Day 1 by shown a CSP version of Prime Finder
- It depends on these two helper functions
  - `(defn factor? [x y] (zero? (mod y x)))`
  - `(defn get-primes [limit]`
    - `(let [primes (chan)`
    - `numbers (to-chan (range 2 limit))]`
    - `(go-loop [ch numbers]`
      - `(when-let [prime (<! ch)]`
        - `(>! primes prime)`
        - `(recur (remove< (partial factor? prime) ch)))`
      - `(close! primes))`
    - `primes))`
- You pass in a limit and create a channel of numbers from 2 to that limit; you then store the first number of that channel as a prime and remove all of its factors; loop until input channel is empty and return primes

Timeouts

- Recall that if you are blocked on a read of a synchronous channel and that channel is closed, the thread that was blocked is unblocked and nil is returned
  - async.core can use that feature to provide channels that timeout
    - `(timeout 1000)` <= returns a channel that will close in 1 sec
    - Example
      - `(time (<!! (timeout 1000))) => "Elapsed time: 1002.925 msecs"`
      - `(time (<!! (timeout 10000))) => "Elapsed time: 10005.497 msecs”`
- This is a surprisingly useful feature when combined with the ability to read multiple channels at once (as discussed next)

Handling Multiple Channels

- The `alt!` function lets code handle more than one channel at a time
  - It can only be used inside of a go block
  - Its structure is to be handed a list of channels and code that should be executed whenever one of them has a message to process

```
(def ch1 (chan))
(def ch2 (chan))
(go-loop []
  (alt!
    ch1 ([x] (println “Read” x “from channel 1”))
    ch2 ([x] (println “Twice” x “is” (* x 2))))
  (recur))
```

- This go block enters an infinite loop waiting for input from one of two channels

Combining alt! with timeout

- Given the alt! function, you can use the channels returned by timeout as one of the channels that is supplied to the alt! function.
    - This can then allow your code to trigger behavior when a timeout has been reached; it can also be used to schedule behavior that should run periodically
- To demonstrate this, the book presents a version of “Finding Primes” that generates as many prime numbers as it can until a timeout has been reached
- It takes the same code as what appears on slide 24 and changes it such that
    - (range 2 limit) becomes (iterate inc 2)
- This transforms the code from operating on a specific sequence of length (limit - 2) to a lazy sequence that is infinite. It then uses alt! to pull numbers from the primes channel until a timeout channel fires. **DEMO**

Other uses for timeout: polling

- You can generate a macro that will generate a function that will execute a task on a periodic basis
  - It does this by creating an infinite loop that performs the action and then reads from a timeout channel
    - The read from the channel blocks until the timeout expires
    - You then loop around and perform the function again
    - and then you block once again reading from the timeout channel
- Explaining Clojure macros is outside of the scope of this class
  - See the book for details

Putting It All Together: An RSS Reader

- We can use all of the techniques described so far to explore an application that involves asynchronous IO: polling RSS feeds for new articles
  - The book presents an example that
    - creates a go routine for each RSS feed being monitored; this routine polls the feed for new articles every minute; it looks for links in the RSS feed and writes them to a channel
    - another go routine maintains a list of already seen articles from a particular feed. Anytime a new article appears, it writes its URL to a channel
    - a third routine retrieves articles and counts the number of words in them and writes those counts to a (third) channel
    - a fourth routine merges all of the counts from multiple RSS feeds into a single channel
    - a fifth routine monitors the merged channel and prints out new counts

Five types of go routines: Lots of Parallelism?

- No. Lots of CONCURRENCY but (in this case) no need for parallelism. Why?
- We have five different types of go routines
  - Type 1: feed monitor => one per feed; polls the feed every 60 seconds
  - Type 2: uniqueness filter => one per feed; only active when a new URL appears
  - Type 3: word counter => one per feed; active once per unique URL
  - Type 4: merge counts => a single routine that adds counts to a single merged channel; not much logic to it; doesn't need a lot of horsepower
  - Type 5: print counts => a final routine that pulls counts off the merged channel and prints them to standard out
- There's not a lot of computation going on. Most of the routines are blocked waiting for new URLs from the feed monitors.
  - As a result, a single thread can handle ALL of these go routines

The Code: Step 1

- We need a routine to perform a single HTTP GET request on an RSS feed
  - `(defn http-get [url] (let [ch (chan)]`
    - `(http/get url (fn [response]`
      - `(if (= 200 (:status response))`
        - `(put! ch response)`
        - `(do (report-error response) (close! ch)))))`
    - `ch))`

The Code: Step 2

- We need a function that uses http-get and parses a feed every minute

```
(defn poll-feed [url]
  (let [ch (chan)]
    (poll 60
      (when-let [response (<! (http-get url))]
        (let [feed (parse-feed (:body response))]
          (onto-chan ch (get-links feed) false))))
    ch))
```

- This function makes use of the poll macro (slide 28). The “go routine” is hiding inside of this call to poll. This code thus represents our “Type 1” go routine
  - The functions parse-feed and get-links can be found in the books source code. They just involve parsing the XML contained in the feed

The Code: Step 3

- To create our “Type 2” go routine (uniqueness filter), we need this code
- ```
(defn new-links [url]
  (let [in (poll-feed url) out (chan)]
    (go-loop [links #{}]
      (let [link (<! in)]
        (if (contains? links link)
          (recur links)
          (do
            (>! out link)
            (recur (conj links link))))))
    out))
```
- This routine looks complex but is actually straightforward. It creates a set called links. Each time it gets a URL, it checks to see if it already has it. If it does it loops back and waits for the next URL. Otherwise, it writes the link to the output channel and adds the link to the set.
```

The Code: Step 4

- Now, we need the code for our “Type 3” word counter go routine.

```
(defn get-counts [urls]
  (let [counts (chan)]
    (go (while true
      (let [url (<! urls)]
        (when-let [response (<! (http-get url))]
          (let [c (count (get-words (:body response)))]
            (>! counts [url c])))))))
  counts))
```

- This function returns a channel and sets up a go routine that loops forever. When a URL is available from the urls channel, it retrieves the contents of the URL and uses get-words to parse the response and count the words. It writes the URL and the word count to its output channel

The Code: Step 5

- The final main function plays the roll of our “type 4” and “type 5” go routines

```
- (defn -main [feeds-file]
  - (with-open [rdr (io/reader feeds-file)]
    - (let
      - [feed-urls (line-seq rdr)
      - articles (doall (map new-links feed-urls))
      - article-counts (doall (map get-counts articles))
      - counts (async/merge article-counts)]

      - (while true
        - (println (<!! counts))))))
```

- That is one powerful “let” statement! Let’s take a look in detail

Step 5 Breakdown

- `feed-urls (line-seq rdr)`
  - Create a sequence that contains the contents of the input file (feed URLs)
- `articles (doall (map new-links feed-urls))`
  - map is used to pass each URL to new-links (type 2 go routine)
    - the result? A sequence of channels
  - each type 2 routine in turn creates its associated type 1 routine
  - doall is used to ensure that map does not return a lazy sequence
- `article-counts (doall (map get-counts articles))`
  - map is used to pass each channel to get-counts (type 3 go routine)
    - Producing ANOTHER sequence of channels
- `counts (async/merge article-counts)`
  - Creates our type 4 go routine: merge all counts into a single channel

DEMO

- With these pieces in place, we can invoke the program on a file of RSS feeds provided by our textbook's author
    - While it is running, we can see that CPU usage is relatively low
    - This is because most of the type 1 routines are blocked waiting for their timeout to expire
        - When they wake up, they make an asynchronous request to get the URL feed (blocking again) and then pass the links in the feed to their uniqueness filter
- In all cases, the ability to "park" the state machines of all these go routines ensures that our thread pool is only executing those routines that have actual work to do

What's Left? Clojurescript!

- On Day 3, the book introduces Clojurescript
  - a version of Clojure that compiles to Javascript and runs in the browser!
- I'm not going to cover these examples but it is useful to look at them yourself
  - BTW, I could only get these examples to work in Chrome
- One cool thing is that if you look at the generated Javascript, you can see the code that is generated for the state machines
- The point of these examples is that
  - Javascript is single threaded but CSPs can be used to enable concurrent behavior via time slicing within the browser

Summary

- Communicating Sequential Processes is an approach to concurrency that
  - focuses on the channels that are used to send messages
  - between program entities that are executing concurrently
- These channels can be composed in interesting ways to specify concurrent algorithms that (via parking) can be executed efficiently by just a few threads
  - If each entity is mostly blocked, then a single thread can often be used to execute 100K CSPs simultaneously
- This approach has been used to provide concurrent behavior of code executing in a web browser, even though the underlying language (Javascript) is single threaded.

Coming Up Next

- Lecture 29: The Lambda Architecture
- Lecture 30: TBD