Chapter 30 Condition Variables

Locks are not always enough

Sometimes a thread may wish to

- Check whether a condition is true before continuing its execution
  - Ex: parent thread wants to wait for a child thread before continuing
  - (Without spinning as in the case of a lock)

Def. Condition Variable: *an explicit queue that threads can put themselves on when some **condition** is not satisfied, to **wait** for that condition to become true. Another thread can change said condition, and then **signal** (to wake up one or more) waiting threads allowing them to continue.*

Go doc: sync.Cond implements a condition variable, a rendezvous point for goroutines waiting for or announcing the occurrence of an event.

Go doc: sync.Cond implements a condition variable, a rendezvous point for goroutines waiting for or announcing the occurrence of an event.

```
func NewCond(mutex Locker) *Cond
func (*Cond) Wait()
func (*Cond) Signal()
```

Wait():

- Wait() assumes that the associated mutex is locked when Wait() is called
- Wait() will atomically:
  - Release the mutex lock, then
  - Put calling goroutine to sleep
- When goroutine wakes up after being signaled by some other goroutine
  - Must re-acquire mutex lock before resuming after the line of Wait()

Signal(): Always hold the lock when calling

Rule: Hold the lock when calling Signal() and Wait().

Example: Implement thread_join using a Condition Variable

Recall C examples

- Chapter 5: Wait for child process to finish
- Chapter 26: Thread_join to wait for a thread to finish

Two cases:

1. Parent creates child
    1. Call thread_join()
        1. Get lock
        2. Check condition
        3. Wait for child; parent is put to sleep
    2. Child runs
        1. Print child
        2. Get locks
        3. Set done = 1
        4. Signal to parent (waking it up)
        5. Unlock

2. Parent creates child; child runs immediately
    1. Call child
    2. Get Lock
    3. Set done = 1
    4. Signal to wake sleeping threads (waiting on condition variable); but there is none
    5. Return from the child()
    6. Parent calls thread_join()
    7. See that done = 1; (in the for loop) — child has already finished
    8. Does not wait; returns after unlock

30.2 The Producer/Consumer Problem (Bounded Buffer)

Imagine:

- One or more producer threads
  - Producers generate data items
  - Places them in a buffer
- One or more consumer threads
  - Consumers grab items from the buffer
  - Consume them in some way

Example Web server

- Producer
  - Puts HTTP requests into work queue
- Consumer
  - Threads / workers that take requests out of the queue and process them

Example: grep foo file.txt | wc -l

- Two processes run concurrently
  - Recall: the difference between concurrency and parallelism
- Connected over a Unix pipe (the | symbol, the unix pipe system call)
- grep is the producer
- wc is the consumer

First attempt: Implement using only a single int as buffer.

This buffer is shared between a producer and a consumer

- Use get() and put() functions
- Count = 1 (mean buffer is full)
- Count = 0 (mean buffer is empty)

Conditions:

- Only put data into buffer when count == 0 (buffer is empty)
- Only get data from the buffer when count == 1 (buffer is full)

A Broken Solution

(*) Single producer and single consumer

- First step:
  - Add lock around critical sections (in producer and consumer)
- Not enough ; we need to add
  - Condition variables
- Second step:
  - Add a single condition variable:
    - cv and associated mutex lock
- This works for single P and single C
  - When P wants to fill buffer: waits for it become empty: lines p1-p3
  - When C wants empty the buffer: waits for it to become full: lines c1-c3
  - (These are different conditions)

What happens if we add another consumer?

Explaining the steps in table on next page:

- Turns out that Tc2 gets scheduled before Tc1 even though Tc1 was waiting to consume item
- Causes Tc1 to panic because the condition is no longer what is expected after the call to wait.

Tc1	State	Tc2	State	Tp	State	Count	Comment
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Running	0	
	Sleep		Ready	p2	Running	0	
	Sleep		Ready	p4	Running	1	Buffer now full
	Ready		Ready	P5	Running	1	Tc1 woken
	Ready		Ready	p6	Running	1	
	Ready		Ready	p1	Running	1	
	Ready		Ready	p2	Running	1	
	Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	c1	Running		Sleep	1	Tc2 sneaks in…
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	… and grabs data
	Ready	c5	Running		Ready	0	Tp awoken
	Ready	c6	Running		Ready	0	
c4	Running		Ready		Ready	0	Oh no! No data to read!

What we saw: if we add more than one consumer, things break…

Problem illustrated in table above:

- After Tp woke Tc1 but before Tc1 ran,
- The state of the bounded buffer changed
- Due to Tc2

This interpretation of what a signal means is referred to as:

Mesa Semantics

- Signaling a thread only wakes it up
- Only a hint that the state has changed
- No guarantee that when woken thread runs, the state will be as desired

Hoare Semantics

- Stronger guarantees; the woken thread will run immediately
- More difficult to implement this semantics
- But we can use *if-condition*
- All systems: use Mesa semantics

Always re-check condition when woken up (Mesa semantics)

- Always use while loop
- or in Go a for loop
  - for condition { Wait }

Problem is that we only have one condition variable

- Shared between consumer and producer

Q: How to fix this?

- Signaling must be more directed:
  - A consumer should not wake other consumers, only producers.
  - And vice-versa!

The Single Buffer Producer/Consumer Solution

Fig 30.12 (see also Go code) shows the solution

- P waits on condition **empty**, and signals **fill.**
- C waits on **fill** and signal **empty.**

A Consumer can never accidentally wake a Consumer

A Producer can never accidentally wake a Producer

The Correct Producer/Consumer Solution with Real Buffer

Add buffer slots

- Allow multiple values to be “produced” before sleeping
- Similarly, multiple values can be “consumed” before sleeping

More efficient

- Reduces context switches
- Allows concurrent producing/consuming

Signaling logic in Fig 30.14

- P only sleep if all buffer slots are filled (p2)
- C only sleep if all buffer slots are empty (c2)
-

Summary

- We have introduced **condition variables**
  - Wait()
  - Signal()
    - Signal to waiting goroutines that the condition **may** have changed
    - Important: waiting goroutine must check condition before resuming: for/while loop
  - Broadcast()
    - Works similar to signal
    - But wakes up all waiting goroutines