CS3211 Brain Picks

Notes

- Conceptually, shared memory maintains invariants on fully replicated state, whereas message passing maintains invariants on disjoint / partially replicated state.
  - In shared memory, count is shared, so we must protect it by granting exclusive access with a mutex.
  - In message passing, count can be passed between goroutines, so we can guarantee no data races via exclusive ownership.
- Go runtime which includes the garbage collector and Go's scheduler.
- Choosing the right memory order may result in far greater performance! By correctly specifying weaker memory orders, the compiler may be allowed to generate more optimized code depending on the situation.
- *std::jthread* works exactly the same as *std::thread*, except that it automatically joins.
- Rust's ownership/borrowing model imposes more ceremony but gives compile-time freedom fromdata races and dangling references; C++ offers more flexibility but the programmer must manually prevent races and lifetime errors, so concurrency bugs are easier to introduce.
  - Default passing
    - Rust: Move semantics for non-Copy types; ownership transferred.
    - C++: Copy value parameters, alias references/pointers.
    - Impact: a value cannot be accessed from two threads simultaneously unless it is wrapped in a Send + Sync type (e.g. $Arc < Mutex < T >>$); compiler stops data races at compile time.
  - Aliasing rules
    - Rust: At most one mutable reference or many immutable ones (enforced by borrow checker).
    - C++: Multiple mutable references allowed; UB if missynchronised.
    - Impact: Rust forces explicit synchronization before sharing mutability, preventing many classic C++ race bugs.
  - Lifetime checking
    - Rust: Explicit lifetimes ensure borrowed data outlives its use.
    - C++: No lifetime tracking beyond scope rules.
    - Impact: Rust's lifetimes make dangling-reference and use-after-free errors compile-time failures, whereas C++ relies on programmer discipline.
- Use-After-Move is unspecified behaviour, not undefined behaviour.
- Mutexes are used to implement critical section, whereas semaphores are used as a signalling mechanism. You can acquire a semaphore on one thread, and release on another, but you cannot do this using mutexes.
- Atomics are processor-level constructs i.e. different (special) instructions are generated. Atomics work faster than mutexes and semaphores because these are implemented using atomics under the hood.
- Channels are not lock-free. chan's internal implementation uses a lock. So, if a goroutine holds the channel's mutex and gets suspended, nobody else can use the channel!

Important Patterns

MPSC in Rust

```
let (s, r) = mpsc::channel(2);
let r = Arc::new(Mutex::new(r)); // cannot clone
chan.send(id).await.unwrap(); // send & recv
let h1 = chan_guard.recv().await.unwrap();
tokio::spawn(async move { // 1 read at a time
    while let Some(op) = sq_rx.recv().await {
      tokio::spawn(async move {
        let res = process(op.data).await;
        cq_tx.send(IoOperation::Write(op.stream,
            ↪ res)).await;
      });
    }
});
```

Hand-over-hand locking

```
// Acquire lock on root
std::unique_lock<std::mutex> prevLock(root->m);
:
// Lock coupling: lock the next node,
// then let prevLock go out of scope (unlocks)
Node* next = stk.top();
std::unique_lock<std::mutex> nextLock(next->m);
prevLock = std::move(nextLock);
```

Fan-out Fan-In in Go

```
fanOut := make([]<- chan int, N)
for i:=0; i< N; i++ {
    fanOut[i] = doWork(done, ...)
}
func fanIn(done <-chan struct{}, channels... <-
   ↪ chan struct{}) <-chan interface {
    multiplexedStream := make(chan interface{})
    multiplex := func(c chan struct{}) {
        defer wg.Done()
        for i := c:
            select {
                case <-done:
                    return
                case multiplexedStream <- i:
            }
    }
    var wg sync.WaitGroup
    wg.Add(len(channels))
    for c := channels:
        go multiplex(c)
    go func() {
        wg.Wait()
        close(multiplexedStream)
    }
    return multiplexedStream
}
for _ := range fanIn(done, fanOut) {
    // ... do work
}
```

Streaming + Cleanup + Pipeline in Go

```
func doWork(done chan<- struct{},stream chan int)
    ↪ <-chan interface {
    outCh := make(chan struct{})
    defer close(outCh)
    go func() {
        for d := range stream {
            select {
            case <-done:
                return
            default:
               outCh <- func(d) { ... }
            }
        }
    }
    return outCh
}
done := make(chan struct{})
defer close(done)
outCh := doWork(done)
for _, val := range outCh:
    ... // after work completed, close(done)
```

Load Balancer in Go

```
type Request struct {
    fn func() int // operation to perform
    c  chan int // channel for output
}
type Worker struct {
    requests chan Request
    pending int
    index int
}
type Pool = []*Worker
type Balancer {
    done chan *Worker
    pool Pool
}
func (w *Worker) work(done chan *Worker) {
    for {
        req := <-w.requests
        req.c := req.fn()
        done <- w
    }
}
func requester(work chan<- Request) {
    c := make(chan int)
    for {
        work <- Request{ workFn, c }
        result := <- c
        furtherProcess(result)
    }
}
func (b *Balancer) balance(work <-chan Request) {
    for {
        select {
            case req := <-work:
                b.dispatch(req)
            case w := <- b.done:
                b.completed(w)
        }
    }
}
func(b *Balancer) dispatch(req Request) {
    w := heap.Pop(&b.pool).(*Worker)
    w.requests <-req
    w.pending ++
    heap.Push(&b.pool, w)
}
func(b *Balancer) completed(w *Worker) {
    w.pending--
    heap.Remove(&b.pool, w.index)
    heap.Push(&b.pool, w)
}
work := make(chan Request, BUFFER_SIZE)
b := Balancer {
    done: make(chan *Worker, NUM_WORKERS),
    pool: make([]*Worker, NUM_WORKERS)
}
go b.balance(work)
for j := RANGE NUM_WORKERS {
    go worker.work()
}
for i := range NUM_REQUESTS {
    go requester(work)
}
```

Tutorial 1

Queue with non-blocking/blocking dequeue

```
struct Job {
  int id;
  int count;
};
class JobQueue4 {
  std::queue<Job> jobs;
  std::mutex mut;
  std::condition_variable cond;

 public:
  JobQueue4() : jobs{}, mut{}, cond{} {}
  void enqueue(Job job) {
    {
      std::unique_lock lock{mut};
      jobs.push(job);
    }

    cond.notify_one();
  }
  std::optional<Job> try_dequeue() {
    std::unique_lock lock{mut};

    if (jobs.empty()) {
      return std::nullopt;
    } else {
      Job job = jobs.front();
      jobs.pop();
      return job;
    }
  }
  Job dequeue() {
    std::unique_lock lock{mut};

    while (jobs.empty()) {
        cond.wait(lock);
    }
    // Alternatively, this is exactly the same
       ↪ code
    // cond.wait(lock, [this]() { return !jobs.
       ↪ empty(); });
    Job job = jobs.front();
    jobs.pop();
    return job;
  }
};
```

Tutorial 3

Shared Pointers

```
template <typename T>
struct SharedPtr
{
private:
    std::atomic<size_t>* m_count;
    T* m_ptr;
public:
    SharedPtr(T* ptr) : m_count(new std::atomic<
        ↪ size_t>(1)), m_ptr(ptr) { }

    SharedPtr(const SharedPtr& other) : m_count(
        ↪ other.m_count), m_ptr(other.m_ptr)
    {
        m_count->fetch_add(1, std::memory_order::
            ↪ relaxed);
    }
    ~SharedPtr()
    {
        size_t old_count = m_count->fetch_sub(1,
            ↪ std::memory_order::acq_rel);
        if(old_count == 1)
        {
            delete m_ptr;
            delete m_count;
        }
    }
    // shouldn't need to worry about this
    T* get() { return m_ptr; }
    const T* get() const { return m_ptr; }
};
```

Tutorial 4

Compare and Swap in C++

```
GenNodePtr old_front = m_queue_front.load(stdmo::
    ↪ relaxed);
while(true)
{
    Node* old_front_next = old_front.node->next.
        ↪ load(stdmo::acquire);
    if(old_front_next == QUEUE_END)
        return std::nullopt;

    // generation is strictly increasing
    GenNodePtr new_front { old_front_next,
        ↪ old_front.gen + 1 };
    if(m_queue_front.compare_exchange_weak(
        ↪ old_front, new_front, stdmo::relaxed))
    {
        break;
    } // retry, with our refreshed ‘old_value‘.
    }
}
```

Tutorial 5

MPMC with 1 channel for all consumers

```
func producer(done chan struct{}, q chan int) {
    for {
        select {
        case q <- 1:
        case <-done:
            return
        }
    }
}
func consumer(done chan struct{}, q queue, sumCh
    ↪ chan<- int) {
    sum := 0
    for {
        select {
        case <-done:
            sumCh <- sum
            return
        case num := <-q:
            sum += num
        }
    }
}
func main() {
    start, done := make(chan struct{}), make(chan
        ↪ struct{})
    q := make(chan int, 10)
    sumCh := make(chan int, NumConsumer)

    for i := 0; i < NumProducer; i++ {
        go func() {
            <-start
            producer(done, q)
        }()
    }
    for j := 0; j < NumConsumer; j++ {
        go func() {
            <-start
            consumer(done, q, sumCh)
        }()
    }
    close(start)
    close(done)
    sum := 0
    for i := 0; i < NumConsumer; i++ {
        sum += <-sumCh
    }
    close(sumCh)
}
```

MPMC with 1 channel for each consumer

```go
func consumer(done chan struct{}, q chan int,
   ↪ sumCh chan<- int) {
    sum := 0
    for {
        select {
        case <-done:
            sumCh <- sum
            close(sumCh)
            return
        case num := <-q:
            sum += num
        }
    }
}
func main() {
    start, done := make(chan struct{}), make(chan
        ↪  struct{})
    q := make(chan int)

    sumChs := make([]chan int, 0, NumConsumer)
    for i := 0; i < NumConsumer; i++ {
        sumCh := make(chan int, 1)
        sumChs = append(sumChs, sumCh)
    }

    for i := 0; i < NumProducer; i++ {
        go func() {
            <-start
            producer(done, q)
        }()
    }
    for j := 0; j < NumConsumer; j++ {
        j := j // capture j in the scope
        go func() {
            <-start
            consumer(done, q, sumChs[j])
        }()
    }
    close(start) // start all goroutines
    close(done)
    sum := 0
    for _, ch := range sumChs { // NOTE: range
        ↪ over slice, not combined channel
        sum += <-ch
    }
}
```

Tutorial 6

Fan-In, Fan-out + Serializer

```go
type EventFunc func(Event) Event
type Event struct {
  id       int64
  procTime time.Duration
}
type worker struct {
  inputCh  <-chan Event
  outputCh chan<- Event
}
func newWorker(inputCh, outputCh chan Event) *
   ↪ worker {
  return &worker{
    inputCh:  inputCh,
    outputCh: outputCh,
  }
}
func (w *worker) start(
  ctx context.Context,
  fn EventFunc, wg *sync.WaitGroup,
) {
  go func() {
    defer wg.Done()
    for {
        select {
        case e, more := <-w.inputCh:
            if !more {
                return
            }
            select {
            case w.outputCh <- fn(e):
            case <-ctx.Done():
                return
            }
        case <-ctx.Done():
            return
        }
    }
  }()
}
func genEventsCh() chan Event {
  dataCh := make(chan Event)
  go func() {
      counter := int64(1)
      rand.Seed(time.Now().Unix())
      for i := 0; i < 30; i++ {
          dataCh <- Event{
              id: counter,
              procTime: time.Duration(rand.Intn
                 ↪ (100)) *     time.
                 ↪ Millisecond,
          }
          counter++
      }
      close(dataCh)
  }()
  return dataCh
  }
func main() {
  ctx, cancel := context.WithCancel(context.
     ↪ Background())
  processedCh := make(chan Event, 1)
  unprocessedCh := genEventsCh()
  finalOutputCh := make(chan Event, 1)
  // Fan-out the stream to multiple workers
  var wg sync.WaitGroup
  for i := 0; i < 10; i++ {
      wg.Add(1)
      newWorker(unprocessedCh, processedCh)
      .start(ctx, func(e Event) Event {
          time.Sleep(e.procTime)
          return e }, &wg)
  }
  go func() {   // Serializer goroutine
      eventMap := make(map[int64]Event) // cache
      var curEventId int64 = 1
      for output := range processedCh {
          if output.id == curEventId {
              finalOutputCh <- output
              curEventId += 1
              // may set off cascade of events
              for {
                  if event, present := eventMap[
                     ↪ curEventId]; present {
                      finalOutputCh <- event
                      curEventId = curEventId + 1
                  } else {
                      break
                  }
              }
          } else {
              eventMap[output.id] = output
          }
      }
      close(finalOutputCh)
```

```
 }()
 // Final reader goroutine
 readerDone := make(chan struct{})
 go func() {
   for output := range finalOutputCh {
     fmt.Printf("Event:␣%d\n", output.id)
   }
   close(readerDone)
 }()
 wg.Wait()
 close(processedCh)
 <-readerDone
 cancel()
}
```

Fan-In, Fan-out + Higher Order Channels

```
func (w *worker) start(
 ctx context.Context,
 fn EventFunc, wQueue chan *worker, wg *sync.
   ↪ WaitGroup,
) {
 go func() {
   defer func() {
     close(w.outputCh)
     wg.Done()
   }()
   for {
     select {
    case wQueue <- w: // register worker
       e := <-w.inputCh
       w.outputCh <- fn(e)
     case <-ctx.Done():
       return
     }
   }
 }()
}

func orderMux(
 ctx context.Context, cancel context.CancelFunc,
 inputCh chan Event, wQueue chan *worker,
   ↪ wOutputCh chan chan Event,
) {
 go func() {
  for {
   select {
   case e, more := <-inputCh:
    if !more { // genEventsCh is finished
     cancel() // signal to terminate
     return
    }
    select {
    case w := <-wQueue:
     wOutputCh <- w.outputCh
     w.inputCh <- e
    case <-ctx.Done():
     return
    }
   case <-ctx.Done():
    return
   }
  }
 }()
}

func main() {
 ctx, cancel := context.WithCancel(context.
   ↪ Background())
 wQueue := make(chan *worker)
 wOutputCh := make(chan chan Event, numWorkers)
 inputCh := genEventsCh()
 orderMux(ctx, cancel, inputCh, wQueue,
   ↪ wOutputCh)
 var wg sync.WaitGroup
 for i := 0; i < numWorkers; i++ {
  wg.Add(1)
  newWorker().start(ctx, func(e Event) Event {
   time.Sleep(e.procTime)
   return e
  }, wQueue, &wg)
 }
 readerDone := make(chan struct{})
 go func() {
  for outputCh := range wOutputCh {
   output, more := <-outputCh
   if !more {
    break // <-- a worker breaks its promise;
   }
   fmt.Printf("Event␣id:␣%d\n", output.id)
  }
  close(readerDone)
 }()
 wg.Wait() // wait for them to exit
 close(wOutputCh)
 <-readerDone
 cancel()
}
```

Pipelining

```go
func toPStage(ctx context.Context, reqCh <-chan
    ↪ Request, instances int, fn func(*Request)
    ↪ *Request ) chan Request {
  outCh := make(chan Request)
  for i := 0; i < instances; i++ {
    go func() {
      for req := range reqCh {
        select {
        case outCh <- *fn(&req):
        case <-ctx.Done():
          return
        }
      }
    }()
  }
  return outCh
}
func client(ctx context.Context, reqCh chan<-
    ↪ Request) {
  for {
    select {
    case reqCh <- *NewReq():
    case <-ctx.Done():
      return
    }
  }
}
func server(ctx context.Context, outCh <-chan
    ↪ Request) {
  for {
    select {
    case completed := <-outCh:
      completed.Done()
    case <-ctx.Done():
      return
    }
  }
}
func main() {
  ctx, cancel := context.WithCancel(context.
      ↪ Background())
  reqCh := make(chan Request, 100)
  outCh := toPStage(ctx, reqCh, 2, request1)
  outCh = toPStage(ctx, outCh, 5, request2)
  outCh = toPStage(ctx, outCh, 10, request3)
  outCh = toPStage(ctx, outCh, 5, request4)
  // Spawn a clients and servers
  start := make(chan struct{})
  for i := 0; i < numClients; i++ {
    go func() {
      <-start
      client(ctx, reqCh)
    }()
  }
  for i := 0; i < numServers; i++ {
    go func() {
      <-start
      server(ctx, outCh)
    }()
  }
  close(start)
  cancel()
}
```

Tutorial 7

H20 Bonding in C++ (Barrier + Semaphore)

```cpp
struct WaterFactory3 {
  std::counting_semaphore<> oxygenSem;
  std::counting_semaphore<> hydrogenSem;
  std::barrier<> barrier;
  WaterFactory3() : oxygenSem{1},
    hydrogenSem{2}, barrier{3} {}
  void oxygen(void (*bond)()) {
    oxygenSem.acquire();
    barrier.arrive_and_wait();
    bond();
    oxygenSem.release();
  }
  void hydrogen(void (*bond)()) {
    hydrogenSem.acquire();
    barrier.arrive_and_wait();
    bond();
    hydrogenSem.release();
  }
};
```

H20 Bonding in Go (Daemon)

```go
type WaterFactoryWithDaemon struct {
    // Channels for atoms to send
    their arrival requests
    precomH chan chan struct{}
    precomO chan chan struct{}
}
func NewFactoryWithDaemon()
    ↪ WaterFactoryWithDaemon {
    wfd := WaterFactoryWithDaemon{
        precomH: make(chan chan struct{}),
        precomO: make(chan chan struct{}),
    }
    go func() { // Daemon goroutine
        for {
            // Step 1: (Precommit)
            h1 := <-wfd.precomH
            h2 := <-wfd.precomH
            o := <-wfd.precomO
            // Step 2: (Commit)
            h1 <- struct{}{} // unbuffered
            h2 <- struct{}{}
            o <- struct{}{}
            // Step 3: (Postcommit)
            <-h1
            <-h2
            <-o
        }
    }()
    return wfd
}
func (wfd *WaterFactoryWithDaemon) hydrogen(bond
    ↪ func()) {
    commit := make(chan struct{})
    wfd.precomH <- commit
    <-commit
    bond()
    commit <- struct{}{}
}
func (wfd *WaterFactoryWithDaemon) oxygen(bond
    ↪ func()) {
    // Step 1: Create private comms channel
    commit := make(chan struct{})
    // Step 2: (Precommit)
    wfd.precomO <- commit
    // Step 3: (Commit)
    <-commit
    // Step 4: Bond
    bond()
```

Since daemon goroutine holds a reference to the water factory, the factory cannot be garbage collected even if all other references are dropped, causing a memory leak. One way is to add a Destroy method to manually shut down the daemon, but this introduces manual memory management. Additionally, the daemon is a **bottleneck**.

H20 Bonding with Go (Oxygen Leader)

```go
type WaterFactoryWithLeader struct {
    oxygenMutex chan struct{}
    precomH     chan chan struct{}
}
func NewFactoryWithLeader()
    ↪ WaterFactoryWithLeader {
    wf := WaterFactoryWithLeader{
        oxygenMutex: make(chan struct{}, 1),
        precomH:     make(chan chan struct{}),
    }
    wf.oxygenMutex <- struct{}{}
    return wf
}
func (wf *WaterFactoryWithLeader) hydrogen(bond
    ↪ func()) {
    commit := make(chan struct{})
    wf.precomH <- commit
    <-commit
    bond()
    commit <- struct{}{}
}
func (wf *WaterFactoryWithLeader) oxygen(bond
    ↪ func()) {
    <-wf.oxygenMutex  // Become leader
    h1 := <-wf.precomH  // Precommit
    h2 := <-wf.precomH
    h1 <- struct{}{} // Commit
    h2 <- struct{}{}
    bond() // Bond
    <-h1 // Postcommit
    <-h2
    wf.oxygenMutex <- struct{}{}  // Step down
```

H2O2 with Go (Leader Election)

```go
type H2O2FactoryWithLeader struct {
    oxygenMutex chan struct{}
    precomH     chan chan struct{}
    precomO     chan chan struct{}
}
func NewFactoryWithLeader() H2O2FactoryWithLeader
    ↪ {
    h2o2f := H2O2FactoryWithLeader{
        oxygenMutex: make(chan struct{}, 1),
        precomH:     make(chan chan struct{}),
        precomO:     make(chan chan struct{}),
    }
    h2o2f.oxygenMutex <- struct{}{}
    return h2o2f
}
func (h2o2f *H2O2FactoryWithLeader) hydrogen(bond
    ↪ func()) {
    commit := make(chan struct{})
    h2o2f.precomH <- commit
    <-commit
    bond()
    commit <- struct{}{}
}
func (h2o2f *H2O2FactoryWithLeader) oxygen(bond
    ↪ func()) {
    // Step 1: Create channel for oxygen follower
    commit := make(chan struct{})
    // Step 2: Elect a leader, or become a
        ↪ follower
    select {
    case h2o2f.precomO <- commit: // Follower
        <-commit
        bond()
        commit <- struct{}{}
    case <-h2o2f.oxygenMutex: // Leader
        // Step 3: (Precommit)
        h1 := <-h2o2f.precomH
        h2 := <-h2o2f.precomH
        o2 := <-h2o2f.precomO
        // Step 4: (Commit)
        h1 <- struct{}{}
        h2 <- struct{}{}
        o2 <- struct{}{}
        // Step 5: Bond
        bond()
        // Step 6: (Postcommit)
        <-h1
        <-h2
        <-o2
        h2o2f.oxygenMutex <- struct{}{}
    }
}
```

FIFO Semaphore with C++ (Atomics)

```cpp
struct FIFOSemaphore2 {
  alignas(128) std::atomic<std::ptrdiff_t>
      ↪ next_ticket;

  alignas(128) std::atomic<std::ptrdiff_t>
      ↪ now_serving; // 2

  FIFOSemaphore2(std::ptrdiff_t count)
      : next_ticket{1}, now_serving{count} {}

  void acquire() {
    // When a thread arrives, get a ticket
    std::ptrdiff_t my_ticket =
        next_ticket.fetch_add(1, std::
            ↪ memory_order_relaxed);
    // Wait until the number reaches or passes us
    while (now_serving.load(std::
        ↪ memory_order_acquire) < my_ticket) {
    }
  }

  void release() {
    now_serving.fetch_add(1, std::
        ↪ memory_order_release);
  }
};
```

Using relaxed memory order for all atomic operations would allow the semaphore's ticket queue to function correctly, preventing internal data races within the semaphore. However, relaxed ordering does not guarantee proper synchronization for the critical section protected by the semaphore, potentially leading to data races there. In *release()*, using release memory order (instead of acquire-release) is sufficient because each *fetch_add* extends the release sequence. Thus, the acquire load in *acquire()* will synchronize not only with the immediately preceding release, but also transitively with all prior releases through the chain of RMW operations on *now_serving*.

FIFO Semaphore with C++ (Conditional Variable)

Rather than spin (which is costly when there is high contention), use a condition variable based monitor. Requires that mutexes are fair, OR a bound on the number of consecutive acquire/releases there are, and the number of spurious wakeups low enough that the acquire at the front of the queue can progress.

```cpp
struct FIFOSemaphore3 {
  std::mutex mut;
  std::condition_variable cond;
  std::atomic<std::ptrdiff_t> next_ticket;
  std::ptrdiff_t now_serving;

  FIFOSemaphore3(std::ptrdiff_t initial_count)
      : mut{}, cond{}, next_ticket{1},
        ↪ now_serving{initial_count} {}
  void acquire() {
    std::ptrdiff_t my_ticket =
        next_ticket.fetch_add(1, std::
          ↪ memory_order_relaxed);
    std::unique_lock lock{mut};
    cond.wait(lock, [=]() { return now_serving >=
        ↪  my_ticket; });
  }
  void release() {
    {
      std::scoped_lock lock{mut};
      now_serving++;
    }
    cond.notify_all();
  }
};
```

FIFO Semaphore with C++ (Queue of Semaphores)

```cpp
struct FIFOSemaphore5 {
  struct Waiter {
    std::binary_semaphore sem{0};
  };
  std::mutex mut;
  std::queue<std::shared_ptr<Waiter>> waiters;
  std::ptrdiff_t count;

  FIFOSemaphore5(std::ptrdiff_t initial_count)
      : mut{}, waiters{}, count{initial_count} {}
  void acquire() {
    auto waiter = std::make_shared<Waiter>();
    {
      std::scoped_lock lock{mut};
      if (count > 0) {
        count--;  // Positive count,
        return;   // simply decrement without
          ↪ blocking
      }
      waiters.push(waiter);  // Zero count, add
        ↪ to waiters
    }
    waiter->sem.acquire();  // and block on the
      ↪ semaphore
  }
  void release() {
    std::shared_ptr<Waiter> waiter;
    {
      std::scoped_lock lock{mut};
      if (waiters.empty()) {
        count++;  // No waiters, simply increment
          ↪  count
        return;
      }
      waiter = waiters.front();  // Pop a waiter
      waiters.pop();
    }
    waiter->sem.release();  // and signal it
  }
};
```

FIFO Semaphore with Go (Channels)

```go
type Semaphore1 struct {
    sem chan struct{}
}
func NewSemaphore1(capacity int, initial_count
    ↪ int) *Semaphore1 {
    sem := Semaphore1{
        sem: make(chan struct{}, capacity),
    }
    for i := 0; i < initial_count; i++ {
        sem.Release()
    }
    return &sem
}
func (s *Semaphore1) Acquire() {
    <-s.sem
    // Blocked goroutines will be unblocked in
        ↪ FIFO order as of Go 1.24
}
func (s *Semaphore1) Release() {
    s.sem <- struct{}{}
}
```

FIFO Semaphore with Go (Daemon)

```go
type Semaphore2 struct {
    acquireCh chan chan struct{}
    releaseCh chan struct{}
}
func NewSemaphore2(initial_count int) *Semaphore2
    ↪ {
    sem := new(Semaphore2)
    sem.acquireCh = make(chan chan struct{}, 100)
    sem.releaseCh = make(chan struct{}, 100)
    go func() {
        count := initial_count
        // The FIFO queue that stores the
            ↪ channels used to unblock waiters
        waiters := NewChanQueue()
        for {
            select {
            case <-sem.releaseCh:
                if waiters.Len() > 0 {
                    // Unblocks the oldest waiter
                    ch := waiters.Pop()
                    ch <- struct{}{}
                } else {
                    count++
                }
            case ch := <-sem.acquireCh:
                if count > 0 {
                    count--
                    ch <- struct{}{} // Don't
                        ↪ keep waiter blocked
                } else {
                    // Add waiter to queue
                    waiters.PushBack(ch)
                }
            }
        }
    }()
    return sem
}
func (s *Semaphore2) Acquire() {
    ch := make(chan struct{})
    s.acquireCh <- ch // daemon
    // Block until daemon decides to unblock us
    <-ch
}
func (s *Semaphore2) Release() {
    s.releaseCh <- struct{}{}
}
```

Not FIFO. No enforced order unblocking.

Tutorial 8

Parallelizing with Rayon

```
use rayon::prelude::*;
fn magic_sum(from: u128, to: u128) -> u128 {
    (from..to).into_par_iter().filter(|i| i % 7
        ↪ == i % 5).sum()
}
fn main() {
    let (from, to) = {
        let mut args = std::env::args();
        args.next(); // skip argv[0]
        (args.next().unwrap(), args.next().unwrap
            ↪ ())
    };
    println!("{}", magic_sum(from.parse().unwrap
        ↪ (), to.parse().unwrap()));
}
```

Tutorial 9

Async I/O

```
use std::net::SocketAddr;
use tokio::io::{AsyncBufReadExt,AsyncWriteExt,
    ↪ BufReader};

use tokio::net::{TcpListener, TcpStream};

async fn handle_client(stream: TcpStream) -> std
    ↪ ::io::Result<()> {
    let mut reader = BufReader::new(stream);
    let mut buf: Vec<u8> = Vec::new();
    loop {
        let size = reader.read_until(b'\n', &mut
            ↪ buf).await?;
        if size == 0 || buf[size - 1] != b'\n' {
            break;
        }
        reader.get_mut().write_all(&buf[..size]).
            ↪ await?;
        buf.clear();
    }
    Ok(())
}

#[tokio::main]
async fn main() -> std::io::Result<()> {
    let port = std::env::args()
        .nth(1)
        .map(|s| s.parse().unwrap())
        .unwrap_or(50000u16);

    let listener = TcpListener::bind(SocketAddr::
        ↪ from(([127, 0, 0, 1], port))).await?;

    loop {
        let (socket, _) = listener.accept().await
            ↪ ?;
        tokio::spawn(async move {
            eprintln!("Accepted␣connection");
            std::mem::drop(handle_client(socket).
                ↪ await);
            eprintln!("Connection␣ended");
        });
    }
}
```

H20 Bonding in Rust

```
#![allow(dead_code)]
use futures::future::join_all;
use std::sync::Arc;
use tokio::sync::{Barrier, Semaphore};

fn bond(s: &str) {
    println!("bond␣{s}!");
}
#[tokio::main]
async fn main() {
    let n = 100;
    let f = Arc::new(WaterFactory::new());

    let hs = (0..n * 2).map(|i| {
        let f = f.clone();
        tokio::spawn((|| async move {
            f.hydrogen(|| bond(&format!("h{i}")))
                ↪ .await;
        })())
    });
    let os = (0..n).map(|i| {
        let f = f.clone();
        tokio::spawn((|| async move {
            f.oxygen(|| bond(&format!("o{i}"))).
                ↪ await;
        })())
    });

    join_all(Iterator::chain(hs, os)).await;
}
struct WaterFactory {
    o_sem: Semaphore,
    h_sem: Semaphore,
    barrier: Barrier,
}
impl WaterFactory {
    fn new() -> Self {
        Self {
            o_sem: Semaphore::new(1),
            h_sem: Semaphore::new(2),
            barrier: Barrier::new(3),
        }
    }

    async fn oxygen(&self, bond: impl FnOnce()) {
        let _permit = self.o_sem.acquire().await.
            ↪ unwrap();
        self.barrier.wait().await;
        bond();
    }

    async fn hydrogen(&self, bond: impl FnOnce())
        ↪ {
        let _permit = self.h_sem.acquire().await.
            ↪ unwrap();
        self.barrier.wait().await;
        bond();
    }
}
```

H20 Bonding Variant (Print IDs)

```
use std::sync::Arc;
use futures::future::join_all;
use tokio::sync::{mpsc, Barrier, Mutex, Semaphore
    ↪ };

async fn hydrogen(
    id: usize,
    barrier: Arc<Barrier>,
    sem: Arc<Semaphore>,
    chan: mpsc::Sender<usize>,
) {
    let _permit = sem.acquire().await.unwrap();
    barrier.wait().await;

    chan.send(id).await.unwrap();
}
async fn oxygen(id: usize, barrier: Arc<Barrier>,
    ↪  chan: Arc<Mutex<mpsc::Receiver<usize>>>)
    ↪ {
    let mut chan_guard = chan.lock().await;
    barrier.wait().await;

    let h1 = chan_guard.recv().await.unwrap();
    let h2 = chan_guard.recv().await.unwrap();
    println!("H␣{}␣-␣O␣{}␣-␣H␣{}", h1, id, h2);
}
```

```
#[tokio::main]
async fn main() {
    let barrier = Arc::new(Barrier::new(3));
    let h_sem = Arc::new(Semaphore::new(2));

    let (s, r) = mpsc::channel(2);
    // wrap single consumer with Arc<Mutex<R>> so
    // it can be cloned
    let r = Arc::new(Mutex::new(r));

    let hydrogens =
        (0..200).map(|i| tokio::spawn(hydrogen(i,
            ↪  barrier.clone(), h_sem.clone(), s
            ↪ .clone())));
    let oxygens = (0..100).map(|i| tokio::spawn(
        ↪ oxygen(i, barrier.clone(), r.clone()))
        ↪ );

    let join_handles = Iterator::chain(hydrogens,
        ↪  oxygens).collect::<Vec<_>>();

    std::mem::drop(s);
    std::mem::drop(r);

    join_all(join_handles).await;
}
```

–End of CS3211–