Classical Synchronization Problems in C++ and Go

CS3211 Parallel and Concurrent Programming

Outline

- Why study classical synchronization problems?
  - Recap
- Problem solutions in C++ and Go
  - Barrier
  - Dining philosophers
  - Barber shop

Motivation

- Classical synchronization problems model problems that we have nowadays in our computer systems

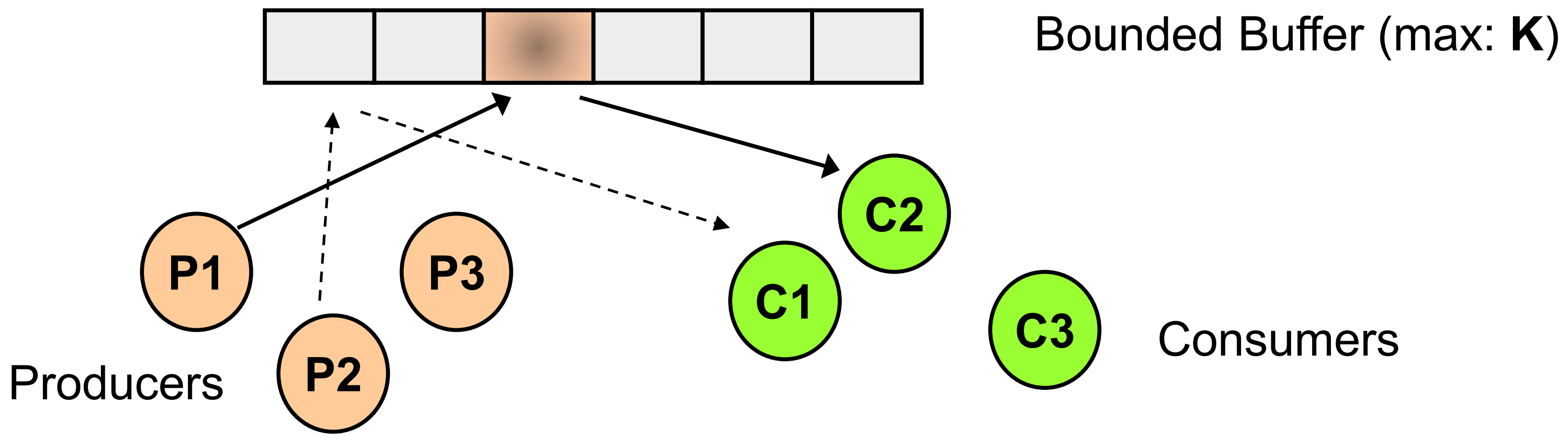
Problem	CS problem
Barrier	Wait until threads/processes reach a specific point in the execution
Producer-consumer	Model interactions between a processor and devices that interact through FIFO channels.
Readers-writers	Model access to shared memory
Dining philosophers	Allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.
Barbershop	Coordinating the execution of a processor.
FIFO Semaphore	Needed to avoid starvation and increase fairness in the system.
H20	Allocation of specific resource to a process.
Cigarette smokers	The agent represents an operating system that allocates resources, and the smokers represent applications that need resources.

Recap from your OS class

- Producer consumer problem
- Readers writers problem

Producer consumer: specification

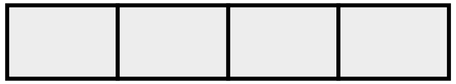
- Processes share a buffer (bounded or unbounded)
  - Producers produce items to insert in buffer
    - Only when the buffer is not full (< K items)
  - Consumers remove items from buffer
    - Only when the buffer is not empty (> 0 items)



Producer consumer in tutorial 1

- https://godbolt.org/z/66zsjWrhd

Producer consumer: buffered version



Example Buffer (**K = 4**)

```
while (TRUE) {
        Produce Item;

        wait(notFull);
        wait(mutex);
        buffer[in] = item;
        in = (in+1) % K;
        count++;
        signal(mutex);
        signal(notEmpty);
}
```

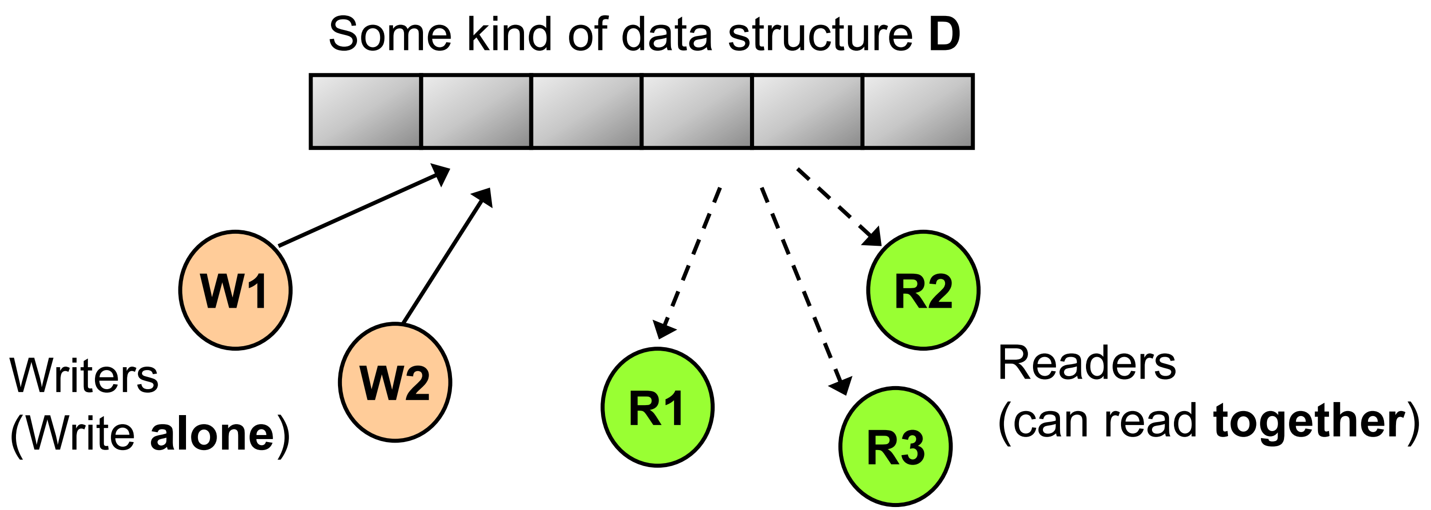
Producer

```
while (TRUE) {

        wait(notEmpty);
        wait(mutex);
        item = buffer[out];
        out = (out+1) % K;
        count--;
        signal(mutex);
        signal(notFull);

        Consume Item;
}
```

Consumer

Readers writers: specification

- Processes share a data structure D:
  - Reader: retrieves information from D
  - Writer: modifies information in D
- Writer must have exclusive access to D
- Reader can access with other readers



Readers writers

Writers

```
roomEmpty.wait ()
    #critical section for writers
roomEmpty.signal ()
```

Starvation of writers is possible

Readers

```
mutex.wait ()
readers += 1
if readers == 1:
    roomEmpty.wait ()  # first in locks
mutex.signal ()
# critical section for readers
mutex.wait ()
readers -= 1
if readers == 0:
    roomEmpty.signal ()  # last out unlocks
mutex.signal ()
```

Lightswitch definition

```
Lightswitch :
    counter = 0
    mutex = Semaphore (1)
    lock (semaphore):
        mutex.wait ()
        counter += 1
        if counter == 1:
            semaphore.wait ()
        mutex.signal ()
    unlock (semaphore):
        mutex.wait ()
        counter -= 1
        if counter == 0:
            semaphore.signal ()
        mutex.signal ()
```

Readers writers with light switch

Writers

```
roomEmpty.wait ()
    #critical section for writers
roomEmpty.signal ()
```

Readers

```
readLightswitch.lock(roomEmpty)
    # critical section
readLightswitch.unlock(roomEmpty)
```

#starving writers

Use a

    turnstile = Semaphore (1)

No-starve readers writers

Writers

turnstile.wait ()

roomEmpty.wait ()

    # critical section for writers

turnstile.signal ()

roomEmpty.signal ()

Readers

turnstile.wait ()

turnstile.signal ()

readSwitch.lock (roomEmpty)

    # critical section for readers

readSwitch.unlock (roomEmpty)

Readers writers locks

- Programming languages have rwlocks
  - C++17 onwards: use shared_mutex
  - Go has RWLock

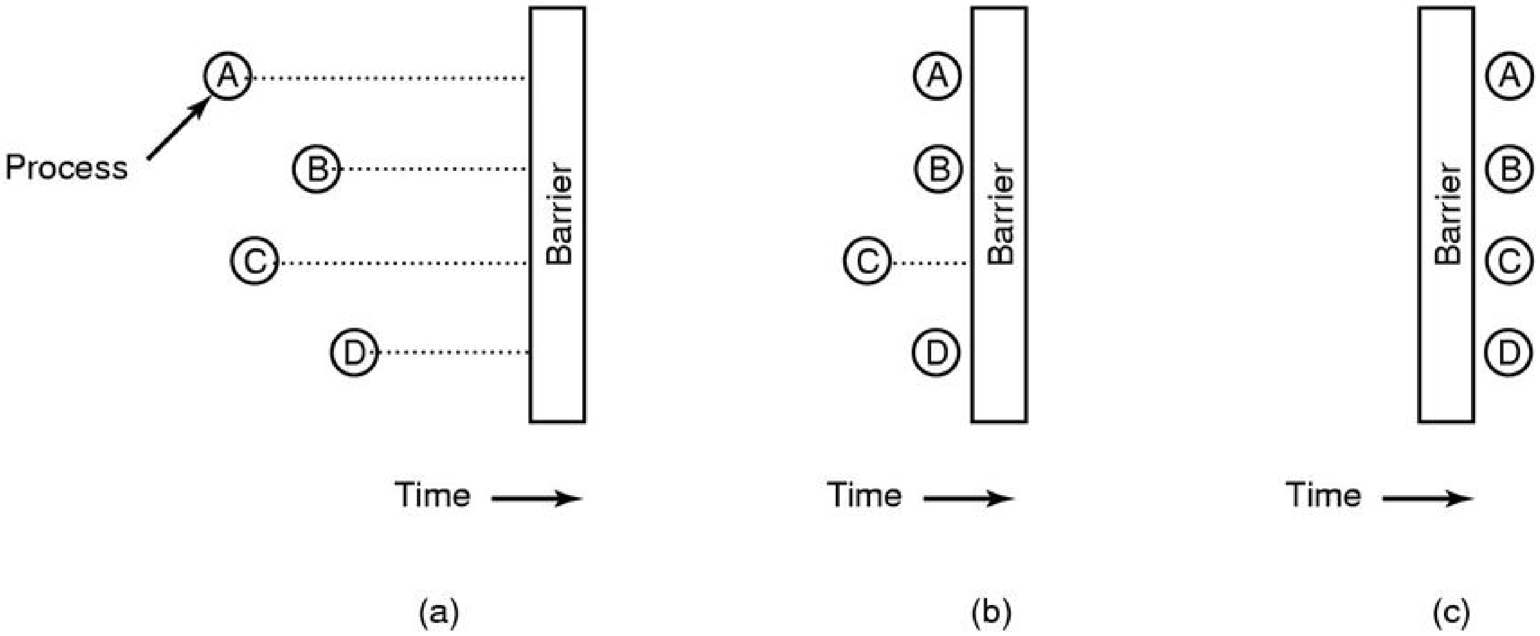
```cpp
class ThreadSafeCounter {
 public:
  ThreadSafeCounter() = default;

  // Multiple threads/readers can read
  //the counter's value at the same time.
  unsigned int get() const {
    std::shared_lock lock(mutex_);
    return value_;
  }

  // Only one thread/writer can
  // increment/write the counter's value.
  unsigned int increment() {
    std::unique_lock lock(mutex_);
    return ++value_;
  }

  // Only one thread/writer can reset/write
  //the counter's value.
  void reset() {
    std::unique_lock lock(mutex_);
    value_ = 0;
  }

 private:
  mutable std::shared_mutex mutex_;
  unsigned int value_ = 0;
};
```

Outline

- Why study classical synchronization problems?
- Problem solutions in C++ and Go
  - Barrier
  - Dining philosophers
  - Barber shop

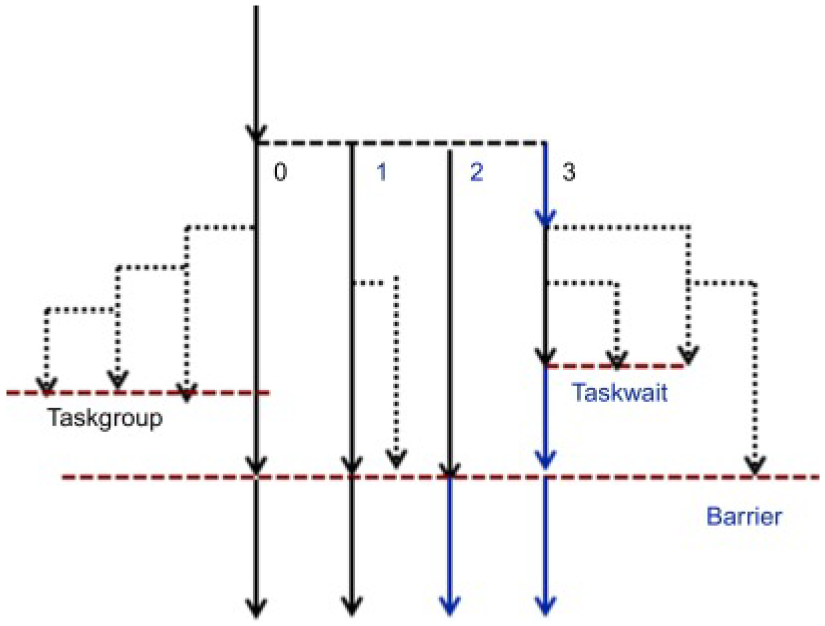
Barrier: specification

- Any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier



Barrier usage

- Appears in many collective routines as part of directive-based parallel languages
  - Parallel `for` loop in OpenMP
  - Collective communication in MPI
- Part of the programming language
  - `std::barrier` in C++20
  - `WaitGroup` `wait` in Go



Types of barriers

- Single use barrier or latch (`std::latch` in C++20)
  - Starts in the raised state and cannot be re-raised once it is in the lowered state
- Reusable barriers (`std::barrier` in C++20)
  - Once the arriving threads are unblocked from a barrier phase's synchronization point, the same barrier can be reused
  - Combining tree barrier - a hierarchical way of implementing barrier
    - Resolve the scalability by avoiding the case that all threads are spinning at the same location

C++: `std::barrier`

- In C++, create a synchronization point
  - Lines 21 & 25: wait for all threads to arrive at the barrier

```cpp
7  int main() {
8    const auto tas = { "hw", "g", "zr", "c" };
9
10   auto on_completion = []() noexcept {
11     // locking not needed here
12     static auto phase = "... done\n" "Cleaning up...\n";
13     std::cout << phase;
14     phase = "... done\n";
15   };
16   std::barrier sync_point(std::ssize(tas), on_completion);
17
18   auto work = [&](std::string name) {
19     std::string product = "  " + name + " worked\n";
20     std::cout << product;
21     sync_point.arrive_and_wait();
22
23     product = "  " + name + " cleaned\n";
24     std::cout << product;
25     sync_point.arrive_and_wait();
26   };
27
28   std::cout << "Starting...\n";
29   std::vector<std::thread> threads;
30   for (auto const& worker : tas) {
31     threads.emplace_back(work, worker);
32   }
33   for (auto& thread : threads) {
34     thread.join();
35   }
36 }
```

C++: Barrier implementation

- Attempt 1:
  - Threads can go ahead other threads by one lap

```
struct BarrierAttempt1 {
  std::ptrdiff_t expected;
  std::ptrdiff_t count;
  std::mutex mut;
  std::counting_semaphore<> turnstile;

  BarrierAttempt1(std::ptrdiff_t expected)
    : expected{expected}, count{0}, mut{}, turnstile{0} {

  void arrive_and_wait() {
    {
      std::scoped_lock lock{mut};
      count++;
      if (count == expected) {
        // Open turnstile
        turnstile.release();
      }
    }

    turnstile.acquire();
    turnstile.release();

    {
      std::scoped_lock lock{mut};
      count--;
      if (count == 0) {
        // Close turnstile to reset barrier
        turnstile.acquire();
      }
    }
  }
};
```

C++: Barrier implementation (2)

```cpp
struct Barrier2 {
  std::ptrdiff_t expected;
  std::ptrdiff_t count;
  std::mutex mut;
  std::counting_semaphore<> turnstile;
  std::counting_semaphore<> turnstile2;

  Barrier2(std::ptrdiff_t expected)
    : expected{expected}, count{0}, mut{},
      turnstile{0}, turnstile2{1} {}

  void arrive_and_wait() {
    {
      std::scoped_lock lock{mut};
      count++;
      if (count == expected) {
        // Close waiter turnstile
        turnstile2.acquire();
        // Open turnstile into the critical section
        turnstile.release();
      }
    }

    turnstile.acquire();
    turnstile.release();
```

- Use 2 turnstiles: raise and lower the barrier

```cpp
struct Barrier2 {
  std::ptrdiff_t expected;
  std::ptrdiff_t count;
  std::mutex mut;
  std::counting_semaphore<> turnstile;
  std::counting_semaphore<> turnstile2;

  Barrier2(std::ptrdiff_t expected)
    : expected{expected}, count{0}, mut{},
      turnstile{0}, turnstile2{1} {}

  void arrive_and_wait() {
    {
      std::scoped_lock lock{mut};
      count++;
      if (count == expected) {
        // Close waiter turnstile
        turnstile2.acquire();
        // Open turnstile into the critical section
        turnstile.release();
      }
    }

    turnstile.acquire();
    turnstile.release();

    {
      std::scoped_lock lock{mut};
      count--;
      if (count == 0) {
        // Close turnstile to reset barrier
        turnstile.acquire();
        // Open second turnstile to let waiters through
        turnstile2.release();
      }
    }

    turnstile2.acquire();
    turnstile2.release();
  }
};
```

C++: Barrier implementation (3)

```
82   // Use a preloaded turnstile to let threads through faster
83   struct Barrier3 {
84     std::ptrdiff_t expected;
85     std::ptrdiff_t count;
86     std::mutex mut;
87     std::counting_semaphore<> turnstile;
88     std::counting_semaphore<> turnstile2;
89
90     Barrier3(std::ptrdiff_t expected)
91         : expected{expected}, count{0}, mut{},
92           turnstile{0}, turnstile2{1} {}
93
94     void arrive_and_wait() {
95       {
96         std::scoped_lock lock{mut};
97         count++;
98         if (count == expected) {
99           // Close waiter turnstile
100          turnstile2.acquire();
101          // Open turnstile into the critical section
102          turnstile.release(expected);
103        }
104      }
105
106      turnstile.acquire();
107
```

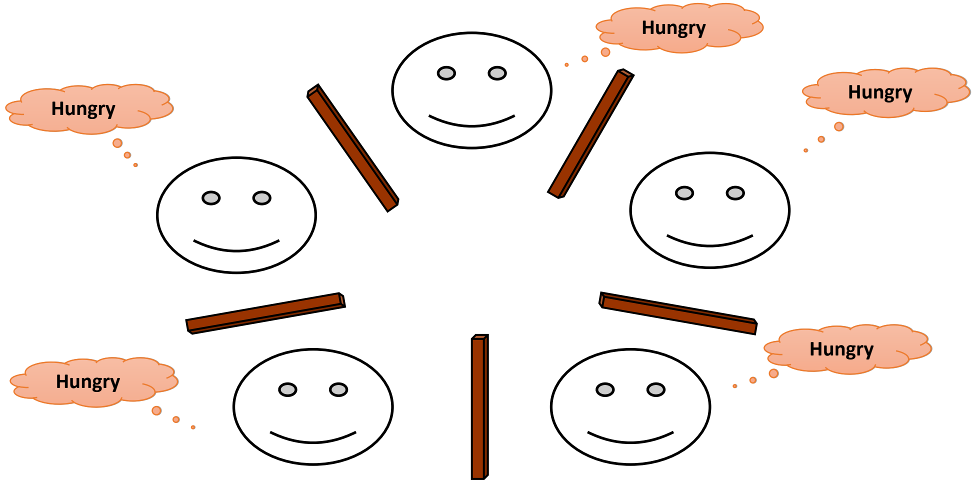
- Lines 102 and 115: counting semaphore can be increased by `expected` to allow threads to pass

```
108      {
109        std::scoped_lock lock{mut};
110        count--;
111        if (count == 0) {
112          // Close turnstile to reset barrier
113          turnstile.acquire();
114          // Open second turnstile to let waiters through
115          turnstile2.release(expected);
116        }
117      }
118
119      turnstile2.acquire();
120    }
121  };
```

Go: Barrier implementation

- Use 2 WaitGroups for a reusable barrier for a group of goroutines

```go
type Barrier1 struct {
    wg  sync.WaitGroup
    wg2 sync.WaitGroup
}

func (b *Barrier1) Init(expected int) {
    b.wg.Add(expected)
    b.wg2.Add(expected)
}

func (b *Barrier1) Wait() {
    b.wg.Done()
    b.wg.Wait()
    // This line only reached when expected
    //threads have called Wait
    // Reset the barrier now
    b.wg.Add(1)
    b.wg2.Done()
    // Wait because the barrier might not
    //be fully reset yet
    b.wg2.Wait()
    // Now barrier is fully reset
    b.wg2.Add(1)
}
```

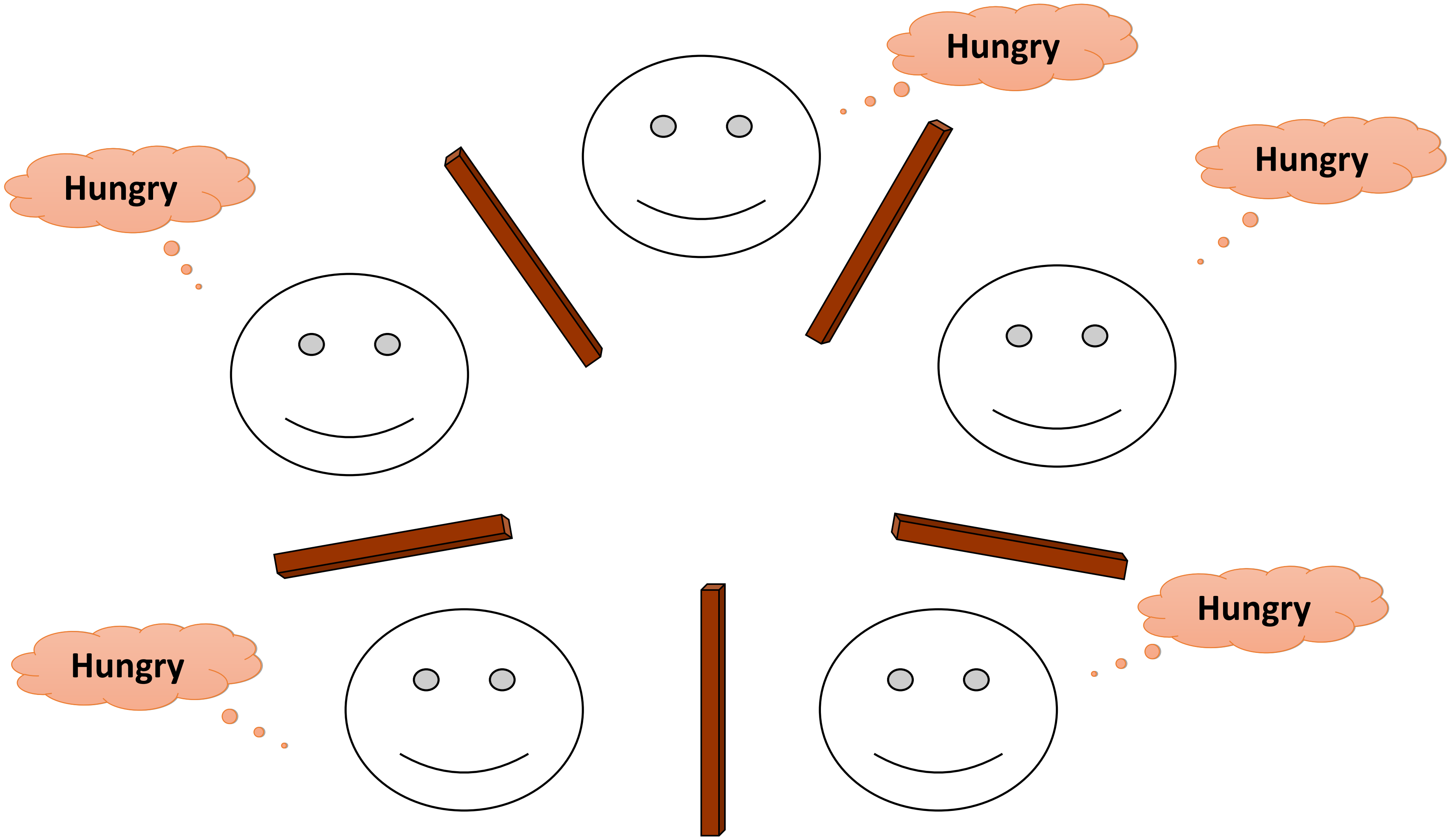


Dining Philosophers

Dining philosophers: specification

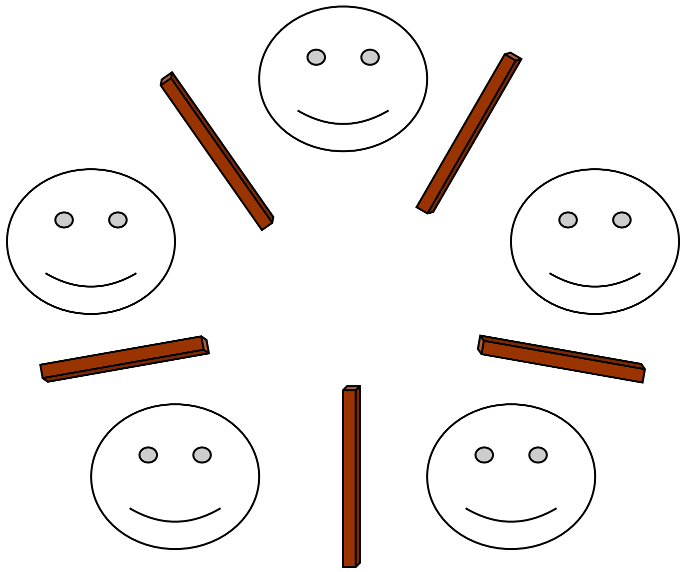
- Appeared when computers were competing for access to tape drive peripherals
- Models the problem of allocating limited resources to a group of processes in a deadlock-free and starvation-free manner
- An algorithm that solves this problem without deadlock
  - Low contention: performs wonderfully when the philosophers spend any appreciable amount of time thinking, compared to eating
  - High contention: philosophers are hungry

Dining philosophers: specification



Dining philosophers: attempt 1

```
#define N 5
#define LEFT i
#define RIGHT ((i+1) % N)

//For philosopher i
while (TRUE){

      Think();
      //hungry, need food!
      takeChpStick(LEFT);
      takeChpStick(RIGHT);

      Eat();

      putChpStick(LEFT);
      putChpStick(RIGHT);
}
```

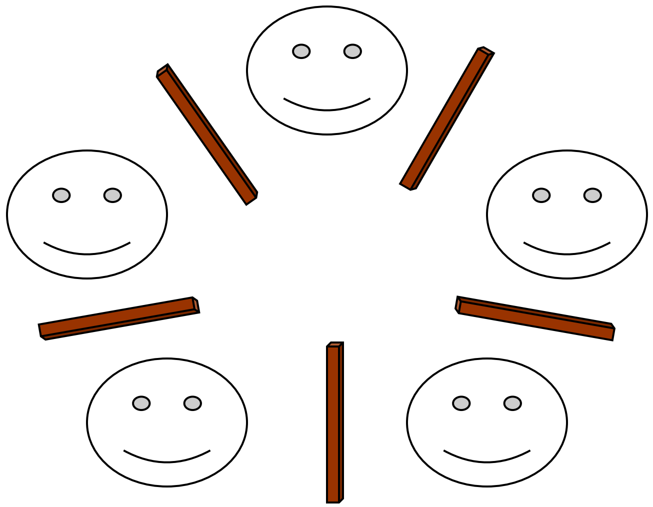


Dining philosophers: attempt 1

- Deadlock:
  - All philosopher simultaneously takes up the left chopstick, and none can proceed
- Fix the attempt:
  - Make the philosopher to put down the left chopstick if right chopstick cannot be acquired
    - Try again later
  - No deadlock:
    - Livelock: All philosopher take up left chopstick, put it down, take it up, put it down,…

Dining philosophers: attempt 2

```
#define N 5
#define LEFT i
#define RIGHT ((i+1) % N)

//For philosopher i
while (TRUE){
      Think( );

      wait(mutex);

      takeChpStick(LEFT);
      takeChpStick(RIGHT);
      Eat( );
      putChpStick(LEFT);
      putChpStick(RIGHT);

      signal(mutex);
}
```



C++: Dining philosophers’ implementation (1)

- Uses *some* deadlock avoidance algorithm
  - The objects are locked by an unspecified series of calls to lock, try_lock, and unlock. If a call to lock or unlock results in an exception, unlock is called for any locked objects before rethrowing
- https://howardhinnant.github.io/dining_philosophers.html

```cpp
// Basic solution to the dining philosopher's problem
template <size_t NumP>
struct DiningTable1 {
  using ChpStick = std::mutex;

  ChpStick chpSticks[NumP];

  // pid = philosopher id
  ChpStick& get_left_chpStick(size_t pid) { return chpSticks[pid]; }
  ChpStick& get_right_chpStick(size_t pid) { return chpSticks[(pid + 1) % NumP]; }

  void eat(size_t pid, void (*eat_callback)(size_t pid)) {
    std::scoped_lock lock{get_left_chpStick(pid), get_right_chpStick(pid)};
    eat_callback(pid);
  }
};
```

Go: Dining philosophers' implementation

- Use odd-even ring communication to avoid the deadlock

```go
type ChpStick struct{}

type DiningTable1 struct {
    numPhilosophers int
    chpStickChs         []chan ChpStick
}

func (t *DiningTable1) Init(numPhilosophers int) {
    t.numPhilosophers = numPhilosophers

    t.chpStickChs = make([]chan ChpStick, 0, numPhilosophers)
    for i := 0; i < numPhilosophers; i++ {
        chpStick := make(chan ChpStick, 1)
        chpStick <- ChpStick{}
        t.chpStickChs = append(t.chpStickChs, chpStick)
    }
}
```

```go
func (t *DiningTable1) Eat(pid int, eat_callback func(pid int)) {
    evenChpStickCh := t.evenChpStickCh(pid)
    oddChpStickCh  := t.oddChpStickCh(pid)

    // Use even / odd chpSticks so the resulting
    // chpStick locking order is acyclic
    <-evenChpStickCh
    <-oddChpStickCh

    eat_callback(pid)

    evenChpStickCh <- ChpStick{}
    oddChpStickCh <- ChpStick{}
```

Go: Dining philosophers' implementation

- Use odd-even ring communication to avoid the deadlock

```
func (t *DiningTable1) leftChpStickCh(pid int) chan ChpStick {
    return t.chpStickChs[pid]
}
func (t *DiningTable1) rightChpStickCh(pid int) chan ChpStick {
    return t.chpStickChs[(pid+1)%t.numPhilosophers]
}
func (t *DiningTable1) evenChpStickCh(pid int) chan ChpStick {
    if pid%2 == 0 {
        return t.leftChpStickCh(pid)
    } else {
        return t.rightChpStickCh(pid)
    }
}
func (t *DiningTable1) oddChpStickCh(pid int) chan ChpStick {
    if pid%2 == 1 {
        return t.leftChpStickCh(pid)
    } else {
        return t.rightChpStickCh(pid)
    }
}
```

```
func (t *DiningTable1) Eat(pid int, eat_callback func(pid int
    evenChpStickCh := t.evenChpStickCh(pid)
    oddChpStickCh := t.oddChpStickCh(pid)

    // Use even / odd chpSticks so the resulting
    // chpStick locking order is acyclic
    <-evenChpStickCh
    <-oddChpStickCh

    eat_callback(pid)

    evenChpStickCh <- ChpStick{}
    oddChpStickCh <- ChpStick{}
```

Go: Dining philosophers' implementation (2)

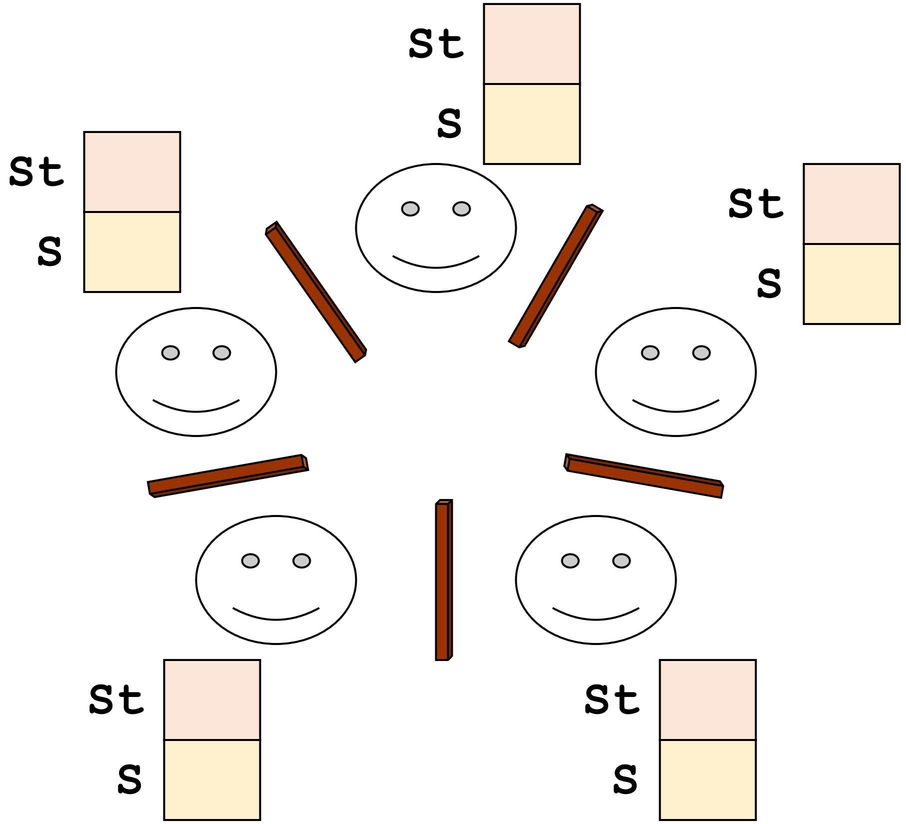
- Swap order of chopsticks to avoid deadlock

```go
func (t *DiningTable2) Init(numPhilosophers int) {
    t.numPhilosophers = numPhilosophers

    t.chpStickChs = make([]chan ChpStick, 0, numPhilosophers)
    for i := 0; i < numPhilosophers; i++ {
        chpStick := make(chan ChpStick, 1)
        chpStick <- ChpStick{}
        t.chpStickChs = append(t.chpStickChs, chpStick)
    }
}
```

```go
func (t *DiningTable2) Eat(pid int, eat_callback func(pid int)) {
    evenChpStickCh := t.evenChpStickCh(pid)
    oddChpStickCh := t.oddChpStickCh(pid)


breakLock:
    for {
        <-evenChpStickCh
        select {
        case <-oddChpStickCh:
            break breakLock
        default:
            // Couldn't get oddChpStickCh, swap order of chpSticks
        }
        evenChpStickCh <- ChpStick{}

        <-oddChpStickCh
        select {
        case <-evenChpStickCh:
            break breakLock
        default:
            // Couldn't get evenChpStickCh, swap order of chpSticks
        }
        oddChpStickCh <- ChpStick{}
    }

    eat_callback(pid)

    evenChpStickCh <- ChpStick{}
    oddChpStickCh <- ChpStick{}
}
```

Dining philosophers: Tanenbaum solution

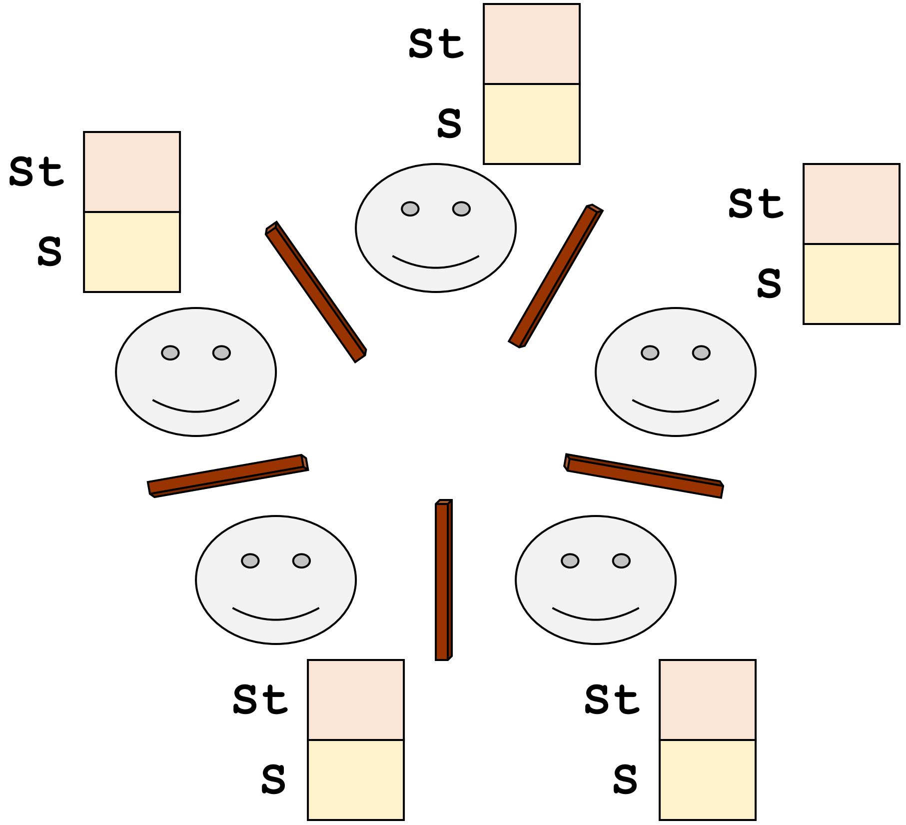
```
#define N 5
#define LEFT ((i+N-1)% N)
#define RIGHT ((i+1) % N)

#define THINKING 0
#define HUNGRY 1
#define EATING 2

int state[N];
Semaphore mutex = 1;
Semaphore s[N];

void philosopher(int i){
    while (TRUE){
        Think();
        takeChpSticks(i);
        Eat();
        putChpSticks(i);
    }
}
```



Dining philosophers: Tanenbaum solution

```
void takeChpSticks( i )
{
        wait(mutex);
        state[i] = HUNGRY;
        safeToEat(i);
        signal(mutex);
        wait(s[i]);
}
```

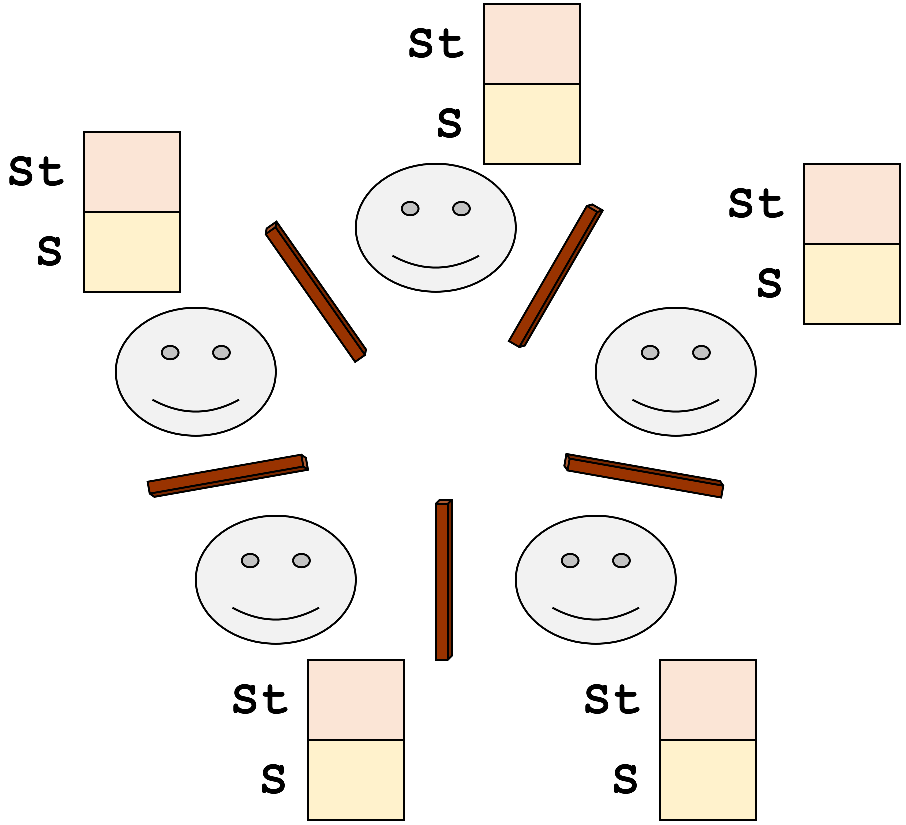
```
void safeToEat( i )
{
    if( (state[i] == HUNGRY) &&
        (state[LEFT] != EATING) &&
        (state[RIGHT] != EATING) ) {

         state[i] = EATING;
         signal(s[i]);
    }
}
```



Dining philosophers: Tanenbaum solution

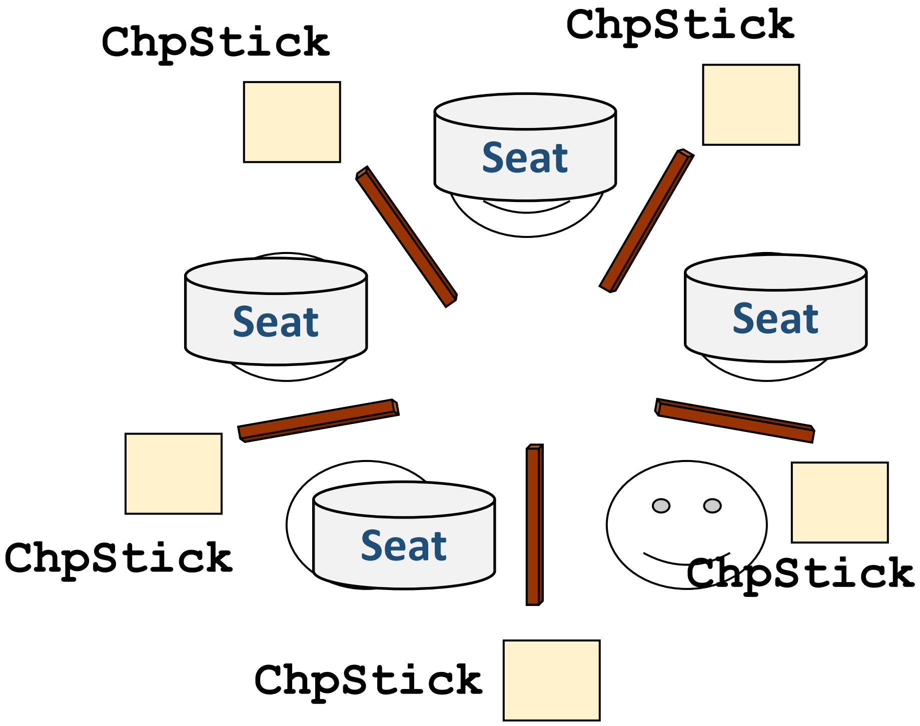
```
void safeToEat(i)
{
    if( (state[i] == HUNGRY) &&
        (state[LEFT] != EATING) &&
        (state[RIGHT] != EATING) ) {

         state[i] = EATING;
         signal(s[i]);
    }
}
```

```
void putChpSticks(i)
{
        wait(mutex);

        state[i] = THINKING;
        safeToEat(LEFT);
        safeToEat(RIGHT);

        signal(mutex);
}
```



Dining philosophers: limited eater

```
void philosopher(int i){

    while (TRUE){
        Think( );
        wait(seats);
        wait(chpStick[LEFT]);
        wait(chpStick[RIGHT]);
        Eat( );
        signal(chpStick[LEFT]);
        signal(chpStick[RIGHT]);
        signal(seats);
    }

}
```



C++: Dining philosophers’ implementation (2)

- `footman_sem` used to limit the number of eaters
- Starvation is possible because mutexes and semaphores are not fair in C++

```
template <size_t NumP>
struct DiningTable2 {
  using ChpStick = std::mutex;

  ChpStick chpSticks[NumP];
  std::counting_semaphore<> footman_sem{NumP - 1};

  // pid = philosopher id
  ChpStick& get_left_chpStick(size_t pid) { return chpSticks[pid];
  ChpStick& get_right_chpStick(size_t pid) { return chpSticks[(pic

  void eat(size_t pid, void (*eat_callback)(size_t pid)) {
    footman_sem.acquire();
    std::scoped_lock left_lock{get_left_chpStick(pid)};
    std::scoped_lock right_lock{get_right_chpStick(pid)};
    eat_callback(pid);
    footman_sem.release();
  }
};
```

C++: Dining philosophers' implementation (3)

- Use a fair semaphore
  - FIFO semaphore will be discussed in tutorial 7

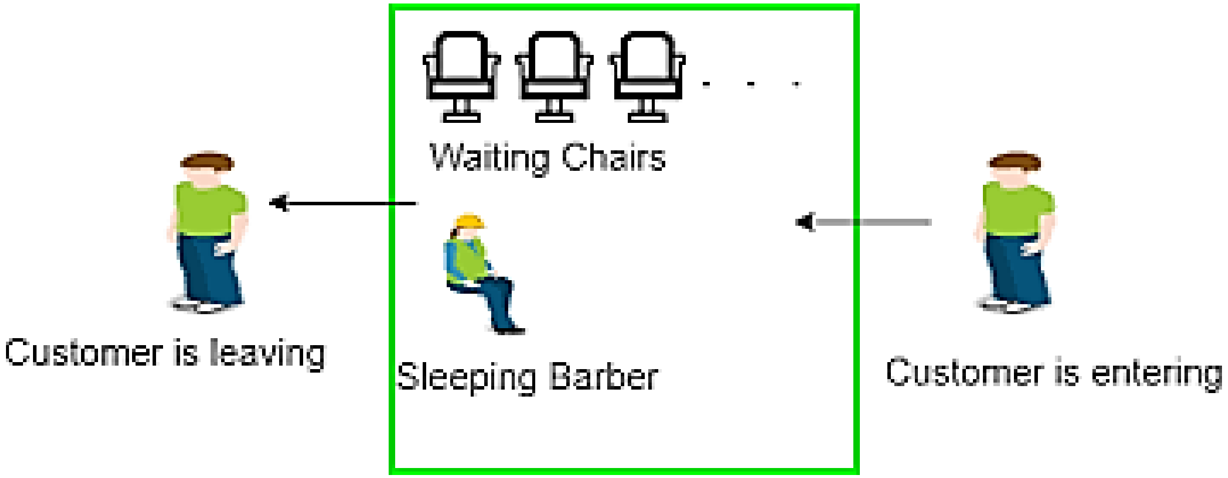
```
template <size_t NumP>
struct DiningTable3 {
  using ChpStick = FairMutex;

  ChpStick chpSticks[NumP];
  FairSemaphore footman_sem{NumP - 1};

  // pid = philosopher id
  ChpStick& get_left_chpStick(size_t pid) { return chpSticks[p
  ChpStick& get_right_chpStick(size_t pid) { return chpSticks[

  void eat(size_t pid, void (*eat_callback)(size_t pid)) {
    footman_sem.acquire();
    std::scoped_lock left_lock{get_left_chpStick(pid)};
    std::scoped_lock right_lock{get_right_chpStick(pid)};
    eat_callback(pid);
    footman_sem.release();
  }
};
```

Go: Dining philosophers' implementation (3)

- This time it doesn't matter what order each philosopher tries to obtain each chopsticks

```go
type DiningTable3 struct {
    numPhilosophers int
    chpStickChs         []chan ChpStick
    footman         chan struct{}
}

func (t *DiningTable3) Init(numPhilosophers int) {
    t.numPhilosophers = numPhilosophers

    t.chpStickChs = make([]chan ChpStick, 0, numPhilosophers)
    for i := 0; i < numPhilosophers; i++ {
        chpStick := make(chan ChpStick, 1)
        chpStick <- ChpStick{}
        t.chpStickChs = append(t.chpStickChs, chpStick)
    }

    t.footman = make(chan struct{}, numPhilosophers-1)
    for i := 0; i < numPhilosophers-1; i++ {
        t.footman <- struct{}{}
    }
}
```

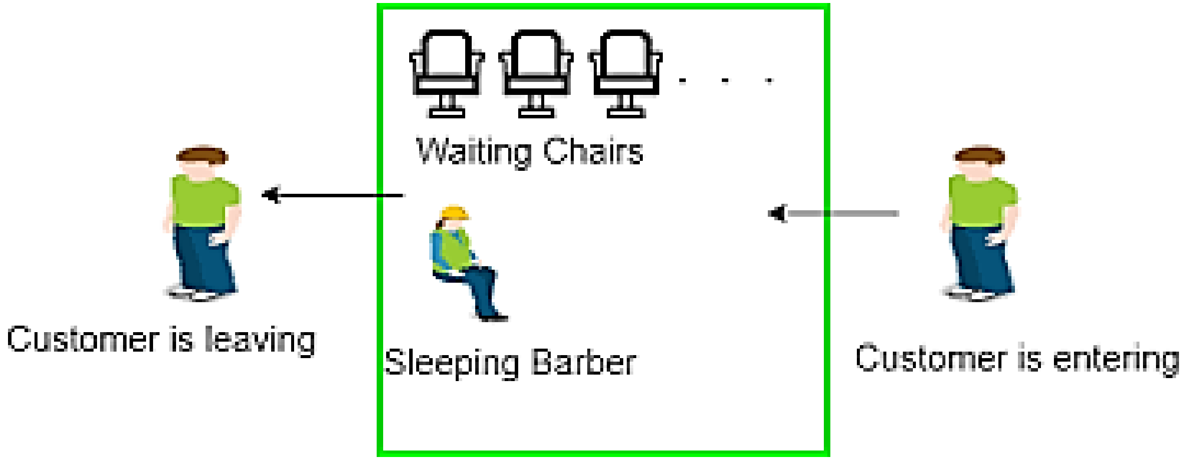
```go
func (t *DiningTable3) Eat(pid int,
        eat_callback func(pid int)) {
    leftChpStickCh := t.leftChpStickCh(pid)
    rightChpStickCh := t.rightChpStickCh(pid)

    <-t.footman

    select {
    case <-leftChpStickCh:
        <-rightChpStickCh
    case <-rightChpStickCh:
        <-leftChpStickCh
    }

    eat_callback(pid)

    leftChpStickCh <- ChpStick{}
    rightChpStickCh <- ChpStick{}
    t.footman <- struct{}{}
}
```



Barbershop

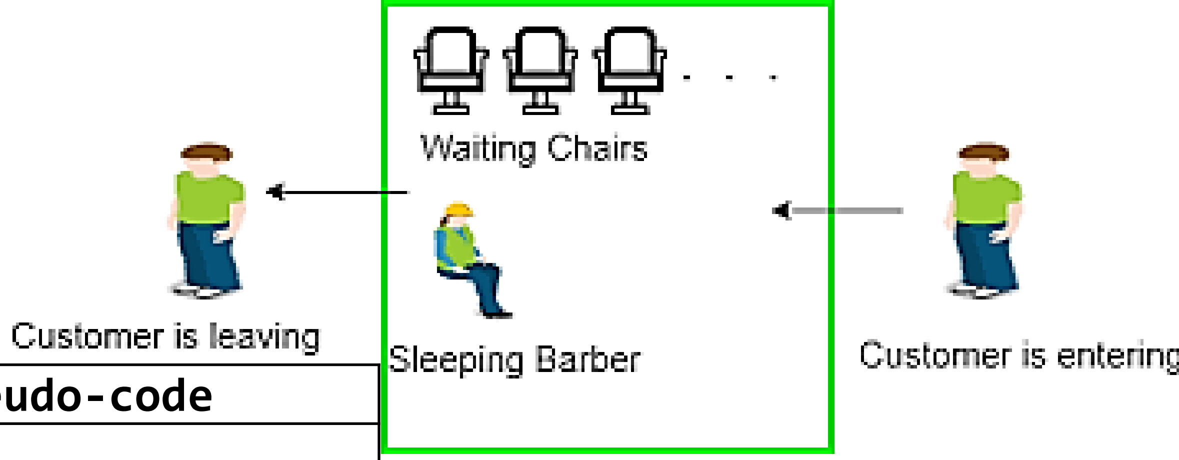
(Sleeping Barber)

Problem description



- A barbershop consists of a waiting room with n chairs and the barber chair
- If there are no customers to be served, the barber goes to sleep
- If the barber is busy, but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber.
- If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop.

Solution



Customer Pseudo-code

```
wait(mutex);
if (customers == n) {
      signal(mutex);
      exit();
}
customers += 1;
signal(mutex);
signal(customer);
wait(barber);
getHairCut ();
signal(customerDone);
wait (barberDone);
wait(mutex);
customers -= 1;
signal(mutex);
```

Barber Pseudo-code

```
while (TRUE) {
  wait(customer);
  signal(barber);
  cutHair();
  wait(customerDone);
  signal(barberDone);
}
```

Line#	Initialization
1	customers = 0
2	mutex = Semaphore (1)
3	customer = Semaphore (0)
4	barber = Semaphore (0)
5	customerDone = Semaphore (0)
6	barberDone = Semaphore (0)

C++: Barbershop implementation

```cpp
template <size_t MaxCustomers>
struct Barbershop1 {
  size_t customers;
  std::mutex mut;
  std::counting_semaphore<> customer_sem;
  std::counting_semaphore<> barber_sem;
  std::counting_semaphore<> done_customer_sem;
  std::counting_semaphore<> done_barber_sem;

  void barber(void (*cutHair)()) {
    while (true) {
      customer_sem.acquire();
      barber_sem.release();
      cutHair();
      done_customer_sem.acquire();
      done_barber_sem.release();
    }
  }
```

```cpp
  void customer(void (*balk)(), void (*getHairCut)()) {
    {
      std::scoped_lock lock{mut};
      if (customers == MaxCustomers + 1) {
        balk();
        return;
      }
      customers++;
    }

    customer_sem.release();
    barber_sem.acquire();
    getHairCut();
    done_customer_sem.release();
    done_barber_sem.acquire();

    {
      std::scoped_lock lock{mut};
      customers--;
    }
  }
};
```

Go: Barbershop implementation

- Using channels
  - Customers might not get served in the FIFO order

```go
type Barbershop1 struct {
    chairs          chan int
    barberAvailable chan struct{}
}

func (bs *Barbershop1) Init(numChairs int) {
    bs.chairs = make(chan int, numChairs)
    bs.barberAvailable = make(chan struct{})
}

func (bs *Barbershop1) Barber(cutHair func()) {
    for {
        bs.barberAvailable <- struct{}{}
        // Someone is sitting in the chair, free them
        <-bs.chairs
        // Cut their hair
        cutHair()
    }
}
```

```go
func (bs *Barbershop1) Customer(balk func(), getHairCut func()) {
    select {
    case bs.chairs <- 1:
        // Sat in chair, ask for barber
        <-bs.barberAvailable
        // Get hair cut
        getHairCut()
    default:
        // Chairs are full
        balk()
    }
}
```

Variations

- FIFO barbershop
- Multiple barbers

Summary

- Take advantage of the specification of the programming language when possible
- No fair semaphore
- Deadlock avoidance in
  - Acquiring multiple locks in C++
  - Odd-even ring communication

- References
- The little book of semaphores: https://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf
- Dining philosophers: https://howardhinnant.github.io/dining_philosophers.html