Little Book of Semaphores, Chapter 4

Geoffrey Matthews
Western Washington University

January 11, 2016

Producers and Consumers

Producer i

```
1  event[i] = waitForEvent()
2  buffer.add(event[i])
```

Consumer j

```
1  event[j] = buffer.get()
2  event[j].process()
```

- Threads must have exclusive access to the buffer. No adding and getting at the same time.
- If a consumer thread arrives when the buffer is empty, it blocks until a producer adds an item.
- `waitForEvent` and `process` can happen simultaneously, but not buffer access.
- `event` is a local variable for each thread—not shared.
- We could use an array, as above, with all producers and consumers having different indices.

Producers and Consumers Hint

```
1  mutex = Semaphore(1)
2  items = Semaphore(0)
3  local event
```

- Local events can be handled several ways:
  - Each thread has its own run-time stack. (We use this in scheme and python, where threads are functions.)
  - Threads could be objects, with local private variables.
  - Threads can use unique IDs as indices into an array.

Producer-consumer solution

Producer

```
1  event = waitForEvent()
2  mutex.wait()
3    buffer.add(event)
4    items.signal()
5  mutex.signal()
```

Consumer

```
1  items.wait()
2  mutex.wait()
3    event = buffer.get()
4  mutex.signal()
5  event.process()
```

Producer-consumer solution

Producer

```
1  event = waitForEvent()
2  mutex.wait()
3    buffer.add(event)
4    items.signal()
5  mutex.signal()
```

Consumer

```
1  items.wait()
2  mutex.wait()
3    event = buffer.get()
4  mutex.signal()
5  event.process()
```

- Could the `items.signal()` be taken out of the mutex?
- What would be the advantage?

Producer-consumer solution (slight improvement)

Producer

```
1  event = waitForEvent()
2  mutex.wait()
3    buffer.add(event)
4  mutex.signal()
5  items.signal()
```

Consumer

```
1  items.wait()
2  mutex.wait()
3    event = buffer.get()
4  mutex.signal()
5  event.process()
```

Producer-consumer solution (slight improvement)

Producer

```
1  event = waitForEvent()
2  mutex.wait()
3    buffer.add(event)
4  mutex.signal()
5  items.signal()
```

Consumer

```
1  items.wait()
2  mutex.wait()
3    event = buffer.get()
4  mutex.signal()
5  event.process()
```

- `items` could at times not accurately reflect the actual number of waiting consumers.

Producer-consumer solution (broken)

Producer

```
1  event = waitForEvent()
2  mutex.wait()
3    buffer.add(event)
4  mutex.signal()
5  items.signal()
```

Consumer

```
1  mutex.wait()
2    items.wait()
3    event = buffer.get()
4  mutex.signal()
5  event.process()
```

- Why is this broken?

Producer-consumer solution (broken)

Producer

```
1  event = waitForEvent()
2  mutex.wait()
3    buffer.add(event)
4  mutex.signal()
5  items.signal()
```

Consumer

```
1  mutex.wait()
2    items.wait()
3    event = buffer.get()
4  mutex.signal()
5  event.process()
```

- Why is this broken?
- Don't wait for a semaphore after grabbing a mutex!

Producer-consumer with finite buffer

Broken finite buffer solution

```
1 if items >= bufferSize:
2   block()
```

- `items` is a semaphore, we can't check its size
- Even if we could, we could be interrupted between checking and blocking.
- We *don't* want to block inside a mutex!

Producer-consumer with finite buffer

Broken finite buffer solution

```
1 if items >= bufferSize:
2   block()
```

- `items` is a semaphore, we can't check its size
- Even if we could, we could be interrupted between checking and blocking.
- We *don't* want to block inside a mutex!
- What to do?

Finite buffer producer-consumer hint

```
1  mutex = Semaphore(1)
2  items = Semaphore(0)
3  spaces = Semaphore(bufferSize)
```

- `items + spaces = bufferSize`

Finite buffer producer-consumer solution

```
1  mutex = Semaphore(1)
2  items = Semaphore(0)
3  spaces = Semaphore(bufferSize)
```

Producer

```
1  event = waitForEvent()
2
3  spaces.wait()
4  mutex.wait()
5    buffer.add(event)
6  mutex.signal()
7  items.signal()
```

Consumer

```
1  items.wait()
2  mutex.wait()
3    event = buffer.get()
4  mutex.signal()
5  spaces.signal()
6
7  event.process()
```

Finite buffer producer-consumer solution

```
1  mutex = Semaphore(1)
2  items = Semaphore(0)
3  spaces = Semaphore(bufferSize)
```

Producer

```
1  event = waitForEvent()
2
3  spaces.wait()
4  mutex.wait()
5    buffer.add(event)
6  mutex.signal()
7  items.signal()
```

Consumer

```
1  items.wait()
2  mutex.wait()
3    event = buffer.get()
4  mutex.signal()
5  spaces.signal()
6
7  event.process()
```

- Note that `items + spaces` is a constant.
- This is an *invariant*
- Using invariants is a good way to design programs and help prove properties.

Readers-writers problem

- Suppose a number of processes all access the same data.
- Any number of readers can be in the critical section simultaneously.
- Writers must have exclusive access to the critical section.
- Ideas?

Readers-writers hint

```
1  readers = 0
2  mutex = Semaphore(1)
3  roomEmpty = Semaphore(1)
```

- "wait" means "wait for the condition to be true"
- "signal" means "signal that the condition is true"

Readers-writers solution

Writers

```
1  roomEmpty.wait()
2  # critical section for writer
3  roomEmpty.signal()
```

Readers

```
1  mutex.wait()
2    readers += 1
3    if readers == 1:
4      # first in locks:
5      roomEmpty.wait()
6  mutex.signal()
7
8  # critical section for reader
9
10 mutex.wait()
11   readers -= 1
12   if readers == 0:
13     # last out unlocks
14     roomEmpty.signal()
15 mutex.signal()
```

- A **Lightswitch**

Claims useful in a proof of correctness

- Only one reader can queue on `roomEmpty`
- Several writers might be queued on `roomEmpty`
- When a reader signals `roomEmpty` the room is empty

A lightswitch object

```
1  class Lightswitch:
2    def __init__(self):
3      self.counter = 0
4      self.mutex = Semaphore(1)
5
6    def lock(self, semaphore):
7      self.mutex.wait()
8        self.counter += 1
9        if self.counter == 1:
10         semaphore.wait()
11     self.mutex.signal()
12
13   def unlock(self, semaphore):
14     self.mutex.wait()
15       self.counter -= 1
16       if self.counter == 0:
17         semaphore.signal()
18         self.mutex.signal()
```

Initialization

```
1  readswitch = Lightswitch()
2  roomEmpty = Semaphore(1)
```

Readers

```
1  readswitch.lock(roomEmpty)
2  # critical section
3  readwitch.unlock(roomEmpty)
```

Writers

```
1  roomEmpty.wait()
2  # critical section for writer
3  roomEmpty.signal()
```

Starvation

- No deadlock in the above readers-writers solution.
- However, it is possible for a writer to **starve**.
- While a writer is blocked, readers can come and go, and the writer never progresses.
- (In the buffer problem, readers eventually empty the buffer, but we can imagine readers who simply examine the buffer without removing an item.)

Starvation

- No deadlock in the above readers-writers solution.
- However, it is possible for a writer to **starve**.
- While a writer is blocked, readers can come and go, and the writer never progresses.
- (In the buffer problem, readers eventually empty the buffer, but we can imagine readers who simply examine the buffer without removing an item.)
- Puzzle: extend the solution so that when a writer arrives, the existing readers can finish, but no additional readers may enter.

Starvation

- No deadlock in the above readers-writers solution.
- However, it is possible for a writer to **starve**.
- While a writer is blocked, readers can come and go, and the writer never progresses.
- (In the buffer problem, readers eventually empty the buffer, but we can imagine readers who simply examine the buffer without removing an item.)
- Puzzle: extend the solution so that when a writer arrives, the existing readers can finish, but no additional readers may enter.
- Hint: Add a turnstyle and allow the writers to lock it.

No-starve readers-writers hint

```
1  readSwitch = Lightswitch()
2  roomEmpty = Semaphore(1)
3  turnstile = Semaphore(1)
```

No-starve readers-writers hint

```
1  readSwitch = Lightswitch()
2  roomEmpty = Semaphore(1)
3  turnstile = Semaphore(1)
```

- `turnstile` is a turnstile for readers and a mutex for writers

No-starve readers-writers solution

Writers

```
1  turnstile.wait()
2    roomEmpty.wait()
3    # critical section
4  turnstile.signal()
5
6  roomEmpty.signal()
```

Readers

```
1  turnstile.wait()
2  turnstile.signal()
3
4  readSwitch.lock(roomEmpty)
5    # critical section
6  readSwitch.unlock(roomEmpty)
```

- `turnstile` is a turnstile for readers and a mutex for writers

No-starve readers-writers solution

Writers

```
1  turnstile.wait()
2    roomEmpty.wait()
3    # critical section
4  turnstile.signal()
5
6  roomEmpty.signal()
```

Readers

```
1  turnstile.wait()
2  turnstile.signal()
3
4  readSwitch.lock(roomEmpty)
5    # critical section
6  readSwitch.unlock(roomEmpty)
```

- `turnstile` is a turnstile for readers and a mutex for writers
- It is now possible for *readers* to starve!

Priority Scheduling

- Some schedulers allow priority scheduling.
- Puzzle: Write a solution to readers-writers that gives priority to writers. In other words, once a writer arrives, no readers are allowed in the critical section until *all* writers have left the system.

Priority Scheduling

- Some schedulers allow priority scheduling.
- Puzzle: Write a solution to readers-writers that gives priority to writers. In other words, once a writer arrives, no readers are allowed in the critical section until *all* writers have left the system.
- Hint: use two lightswitches

Writer-priority readers-writers hint

```
1  readSwitch = Lightswitch()
2  writeSwitch = Lightswitch()
3  mutex = Semaphore(1)
4  noReaders = Semaphore(1)
5  noWriters = Semaphore(1)
```

Writer-priority readers-writers solution

Writers

```
1  writeSwith.lock(noReaders)
2    noWriters.wait()
3
4      # critical section
5
6    noWriters.signal()
7  writeSwitch.unlock(noReaders)
```

Readers

```
1  noReaders.wait()
2    readSwitch.lock(noWriters)
3  noReaders.signal()
4
5  # critical section
6
7  readSwitch.unlock(noWriters)
```

- Writers in critical section hold *both* `noReaders` and `noWriters`.
- `writeSwitch` allows writers to queue on `noWriters`, but keeps `noReaders` locked
- The last writer signals `noReaders`

Writer-priority readers-writers solution

Writers

```
1  writeSwith.lock(noReaders)
2    noWriters.wait()
3
4      # critical section
5
6    noWriters.signal()
7  writeSwitch.unlock(noReaders)
```

Readers

```
1  noReaders.wait()
2    readSwitch.lock(noWriters)
3  noReaders.signal()
4
5  # critical section
6
7  readSwitch.unlock(noWriters)
```

- Writers in critical section hold *both* `noReaders` and `noWriters`.
- `writeSwitch` allows writers to queue on `noWriters`, but keeps `noReaders` locked
- The last writer signals `noReaders`
- Readers in critical section hold `noWriters` but don't hold `noReaders`, so a writer can lock `noReaders`
- The last reader signals `noWriters` so writers can go

Thread starvation

- We just addressed **categorical starvation**: one category of threads makes another category starve.
- **Thread starvation** is the more general possibility of a thread waiting indefinitely while other threads proceed.

Thread starvation

- We just addressed **categorical starvation**: one category of threads makes another category starve.
- **Thread starvation** is the more general possibility of a thread waiting indefinitely while other threads proceed.
- Part of the problem is the responsibility of the scheduler. If a thread is never scheduled, it is starved.

Thread starvation

- We just addressed **categorical starvation**: one category of threads makes another category starve.
- **Thread starvation** is the more general possibility of a thread waiting indefinitely while other threads proceed.
- Part of the problem is the responsibility of the scheduler. If a thread is never scheduled, it is starved.
- Some schedulers use algorithms that guarantee bounded waiting.

Thread starvation

- If we don't want to assume too much about the scheduler, can we assume:
- **Property 1:** if there is only one thread that is ready to run, the scheduler has to let it run.
- This would be sufficient for the boundary problem.
- In general we need a stronger assumption.

Thread starvation

- **Property 2:** if a thread is ready to run, then the time it waits until it runs is bounded.
- We use this assumption in all our work.
- Some schedulers in the real world do not guarantee this strictly.
- Property 2 is not strong enough if we use semaphores. Why?

Semaphore starvation

- The weakest assumption about semaphores that makes it possible to avoid starvation is:
- **Property 3:** if there are threads waiting on a semaphore when a thread executes `signal`, then one of the waiting threads has to be woken.

Semaphore starvation

- The weakest assumption about semaphores that makes it possible to avoid starvation is:
- **Property 3:** if there are threads waiting on a semaphore when a thread executes `signal`, then one of the waiting threads has to be woken.

- Prevents a thread from signalling a semaphore, racing around a loop and catching its own signal!

Thread i

```
1  while True:
2    mutex.wait()
3    # critical section
4    mutex.signal()
```

Semaphore starvation

- The weakest assumption about semaphores that makes it possible to avoid starvation is:
- **Property 3:** if there are threads waiting on a semaphore when a thread executes `signal`, then one of the waiting threads has to be woken.

- Prevents a thread from signalling a semaphore, racing around a loop and catching its own signal!

Thread i

```
1  while True:
2    mutex.wait()
3    # critical section
4    mutex.signal()
```

- However, if A, B, and C are using a mutex in a loop, A and B could race around and around, starving C.

Semaphore starvation

- The weakest assumption about semaphores that makes it possible to avoid starvation is:
- **Property 3:** if there are threads waiting on a semaphore when a thread executes `signal`, then one of the waiting threads has to be woken.

- Prevents a thread from signalling a semaphore, racing around a loop and catching its own signal!

Thread i

```
1  while True:
2    mutex.wait()
3    # critical section
4    mutex.signal()
```

- However, if A, B, and C are using a mutex in a loop, A and B could race around and around, starving C.
- A semaphore with Property 3 is called a **weak semaphore**.

Semaphore starvation

- **Property 4:** if a thread is waiting at a semaphore, then the number of threads that will be woken before it is bounded.

Semaphore starvation

- **Property 4:** if a thread is waiting at a semaphore, then the number of threads that will be woken before it is bounded.
- FIFO queues satisfy this property.

Semaphore starvation

- **Property 4:** if a thread is waiting at a semaphore, then the number of threads that will be woken before it is bounded.
- FIFO queues satisfy this property.
- A semaphore with Property 4 is called a **strong semaphore**.

Semaphore starvation

- **Property 4:** if a thread is waiting at a semaphore, then the number of threads that will be woken before it is bounded.
- FIFO queues satisfy this property.
- A semaphore with Property 4 is called a **strong semaphore**.
- Dijkstra (inventor of semaphores) conjectured in 1965 that it was impossible to solve the mutex problem without starvation with weak semaphores.

Semaphore starvation

- **Property 4:** if a thread is waiting at a semaphore, then the number of threads that will be woken before it is bounded.
- FIFO queues satisfy this property.
- A semaphore with Property 4 is called a **strong semaphore**.
- Dijkstra (inventor of semaphores) conjectured in 1965 that it was impossible to solve the mutex problem without starvation with weak semaphores.
- Morris showed you could do it in 1979.

Morris's algorithm

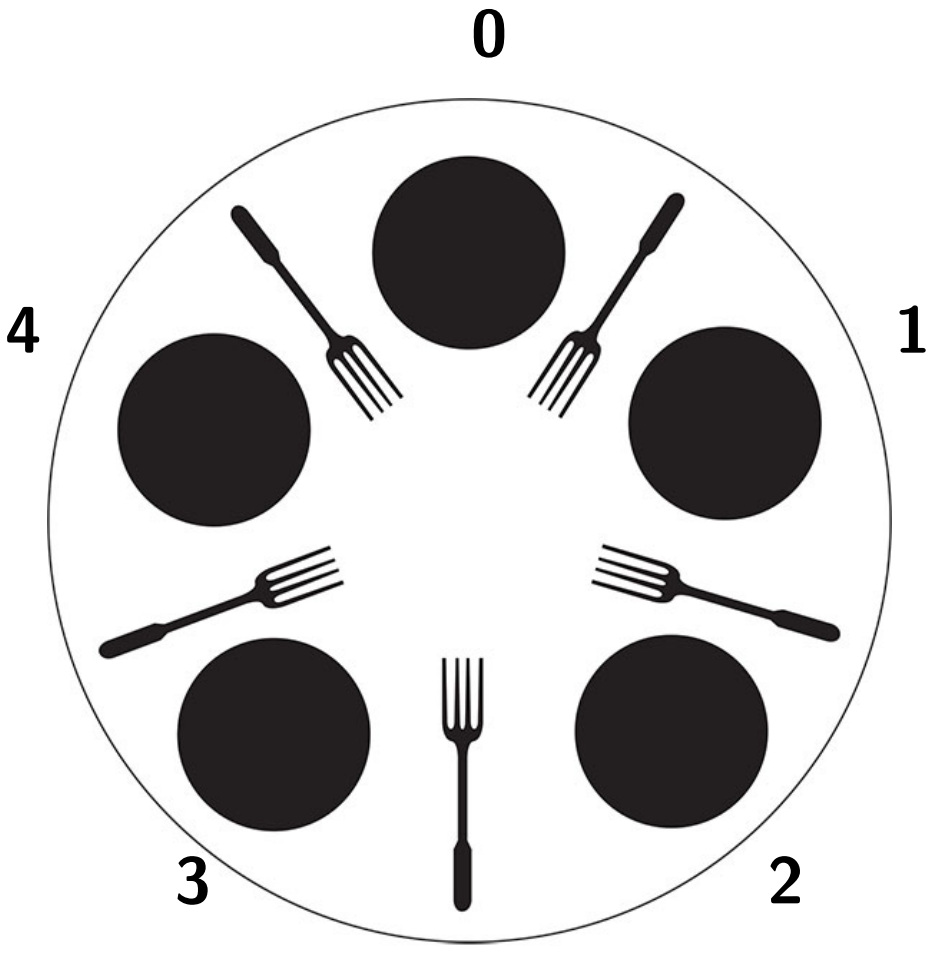
Initialization

```
1  room1 = room2 = 0
2  mutex = Semaphore(1)
3  t1 = Semaphore(1)
4  t2 = Semaphore(0)
```

```
1  mutex.wait()
2    room1 += 1
3  mutex.signal()
4  t1.wait()
5    room2 += 1
6    mutex.wait()
7    room1 -= 1
8    if room1 == 0:
9      mutex.signal()
10     t2.signal()
11 t2.wait()
12   room2 -= 1
13   # critical section
14   if room2 == 0:
15     t1.signal()
16   else:
17     t2.signal()
```

The Dining Philosophers

- Five philosophers are eating spaghetti.
- There are five forks.
- Eating spaghetti requires two forks.
- More than one philosopher can eat at a time.



Philosopher i

```
1  while True:
2    think()
3    get_forks()
4    eat()
5    put_forks()
```

The Dining Philosophers: a Non-solution

Which fork?

```
1  def left(i) = return i
2  def right(i) = return (i+1)%5
```

Initialization

```
1  forks =
2    [Semaphore(1)
3      for i in range(5)]
```

Non-solution

```
1  def get_forks(i):
2    fork[right(i)].wait()
3    fork[left(i)].wait()
4
5  def put_forks(i):
6    fork[right(i)].signal()
7    fork[left(i)].signal()
```

- Why does this not work?

The Dining Philosophers: a Non-solution

Which fork?

```
1  def left(i) = return i
2  def right(i) = return (i+1)%5
```

Initialization

```
1  forks =
2    [Semaphore(1)
3      for i in range(5)]
```

Non-solution

```
1  def get_forks(i):
2    fork[right(i)].wait()
3    fork[left(i)].wait()
4
5  def put_forks(i):
6    fork[right(i)].signal()
7    fork[left(i)].signal()
```

- Why does this not work?
- Deadlock is possible. How?

The Dining Philosophers: Solution #1

Initialization

```
1  footman = Semaphore(4)
```

Solution #1

```
1  def get_forks(i):
2    footman.wait()
3    fork[right(i)].wait()
4    fork[left(i)].wait()
5
6  def put_forks(i):
7    fork[right(i)].signal()
8    fork[left(i)].signal()
9    footman.signal()
```

- Only allow 4 philosophers at a time.
- Can you prove deadlock is impossible?

The Dining Philosophers: Solution #2

- Have at least one leftie and at least one rightie.
- How can you prove deadlock is impossible?

The Dining Philosophers: Tanenbaum's solution

Initialization

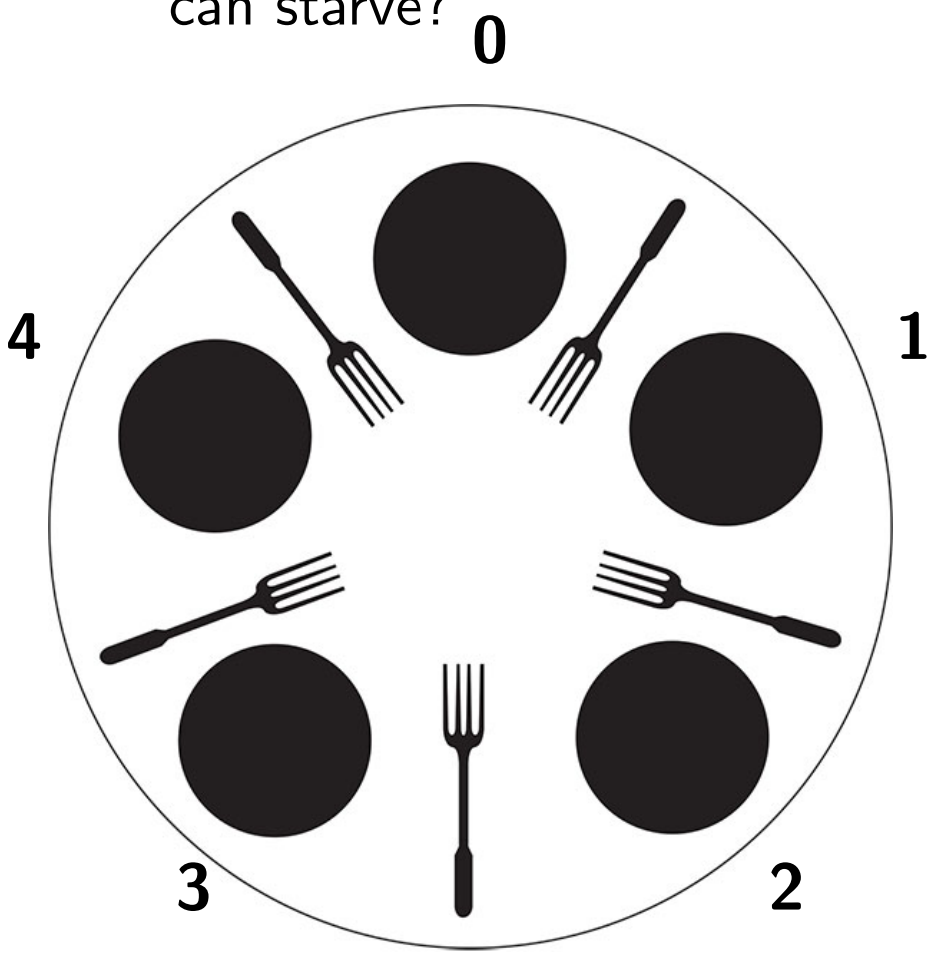
```
1  state = ['thinking'] * 5
2  sem = [Semaphore(0) for i in range(5)]
3  mutex = Semaphore(1)
```

```
1  def get_fork(i):
2    mutex.wait()
3    state[i] = 'hungry'
4    test(i)
5    mutex.signal()
6    sem[i].wait()
7
8  def put_fork(i):
9    mutex.wait()
10   state[i] = 'thinking'
11   test(right(i))
12   test(left(i))
13   mutex.signal()
```

```
1  def test(i):
2  if (
3    state[i] == 'hungry'
4    and
5    state(left(i)) != 'eating'
6    and
7    state(right(i)) != 'eating'
8    ) :
9      state[i] = 'eating'
10     sem[i].signal()
```

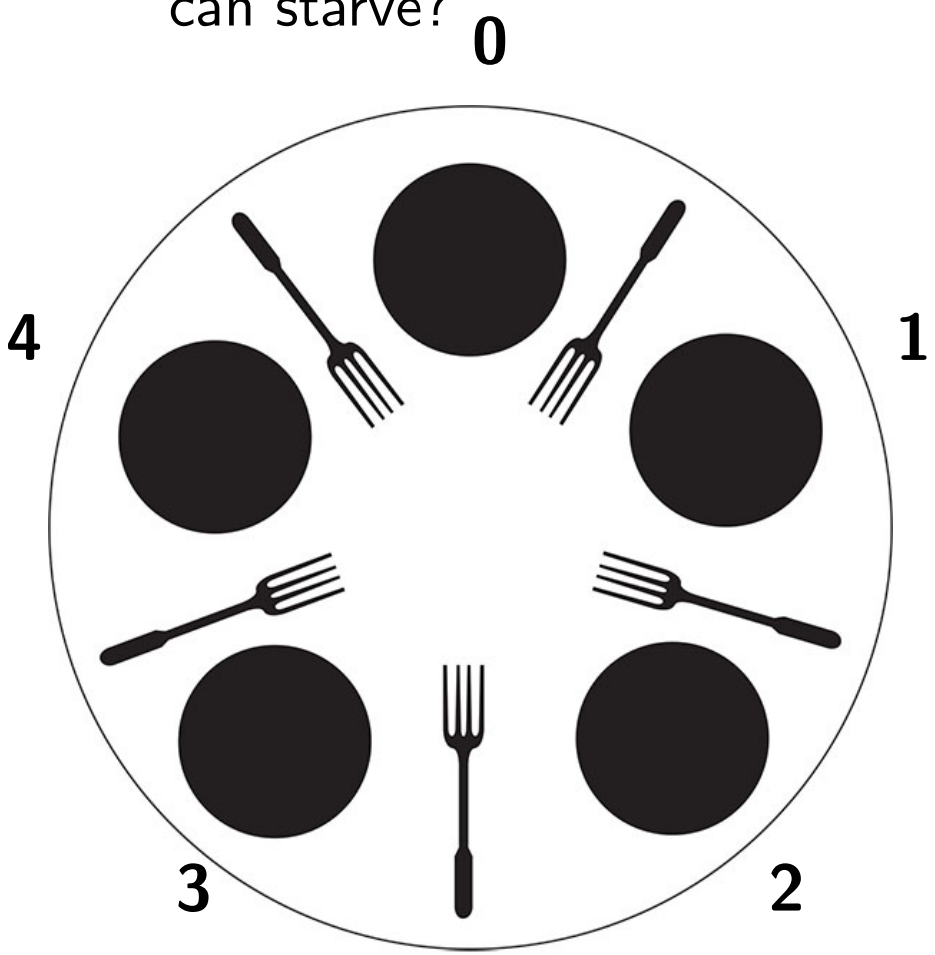
Tanenbaum's solution not starvation-free

- Can you think of a situation where a Tanenbaum philosopher can starve?



Tanenbaum's solution not starvation-free

- Can you think of a situation where a Tanenbaum philosopher can starve?



- Imagine trying to starve 0.
- Initially 1 and 3 are eating
- 3 swaps with 4
- 1 swaps with 2
- Now 2 and 4 are eating
- Repeat

Cigarette smokers problem

- To smoke you need: paper, tobacco, and a match.
- Four threads: an agent and three smokers.
- One smoker already has lots of paper.
- One smoker already has lots of tobacco.
- One smoker already has lots of matches.
- The agent repeatedly obtains two ingredients at random.
- The agent should wake up the smoker who needs those ingredients.

Cigarette smokers problem: agent code

- Interesting problem assumes we cannot change agent code.

Initialization

```
1  agentSem = Semaphore(1)
2  tabacco = Semaphore(0)
3  paper = Semaphore(0)
4  match = Semaphore(0)
```

Agent A

```
1  agentSem.wait()
2  tabacco.signal()
3  paper.signal()
```

Agent B

```
1  agentSem.wait()
2  paper.signal()
3  match.signal()
```

Agent C

```
1  agentSem.wait()
2  tabacco.signal()
3  match.signal()
```

Cigarette smokers problem: Non-solution

Agent A

```
1  agentSem.wait()
2  tabacco.signal()
3  paper.signal()
```

Agent B

```
1  agentSem.wait()
2  paper.signal()
3  match.signal()
```

Agent C

```
1  agentSem.wait()
2  tabacco.signal()
3  match.signal()
```

Smoker with matches

```
1  tabacco.wait()
2  paper.signal()
3  agentSem.signal()
```

Smoker with tobacco

```
1  paper.wait()
2  match.wait()
3  agentSem.signal()
```

Smoker with paper

```
1  tabacco.wait()
2  match.wait()
3  agentSem.signal()
```

- Why doesn't this work?

Smokers problem: Parnas solution

Initialization

```
1  isTobacco = isPaper =
2      isMatch = False
3  tobaccoSem = Semaphore(0)
4  paperSem = Semaphore(0)
5  matchSem = Semaphore(0)
```

Smoker with tobacco

```
1  tobaccoSem.wait()
2  makeCigarette()
3  agentSem.signal()
4  smoke()
```

Pusher A

```
1  tobacco.wait()
2  mutex.wait()
3    if isPaper:
4      isPaper = False
5      matchSem.signal()
6    elif isMatch:
7      isMatch = False
8      paperSem.signal()
9    else:
10     isTobacco = True
11 mutex.signal()
```

Generalized smokers problem

Initialization

```
numTobacco = numPaper =
    numMatch = 1
tobaccoSem = Semaphore(0)
paperSem = Semaphore(0)
matchSem = Semaphore(0)
```

Smoker with tobacco

```
tobaccoSem.wait()
makeCigarette()
agentSem.signal()
smoke()
```

Pusher A

```
tobacco.wait()
mutex.wait()
  if numPaper:
    isPaper -= 1
    matchSem.signal()
  elif numMatch:
    isMatch -= 1
    paperSem.signal()
  else:
    isTobacco += 1
mutex.signal()
```

- Keep a **scoreboard**:
  variables describing the state of the system.
- Each agent checks the scoreboard to see if it can proceed.