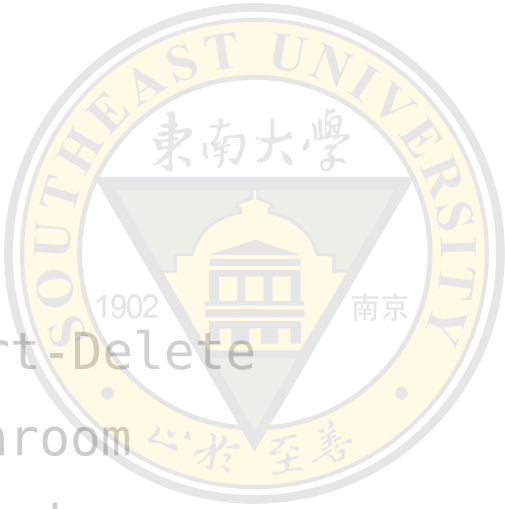
東南大學
SOUTHEAST UNIVERSITY

OPERATING SYSTEM CONCEPTS

Chapter 12. File-System Implementation

A/Prof. Kai Dong

Contents



No-Starve Mutex

Puzzle

Write a solution to the mutual exclusion problem using weak semaphores. Your solution should provide the following guarantee: once a thread arrives and attempts to enter the mutex, there is a bound on the number of threads that can proceed ahead of it. You can assume that the total number of threads is finite.

No-Starve Mutex

Key

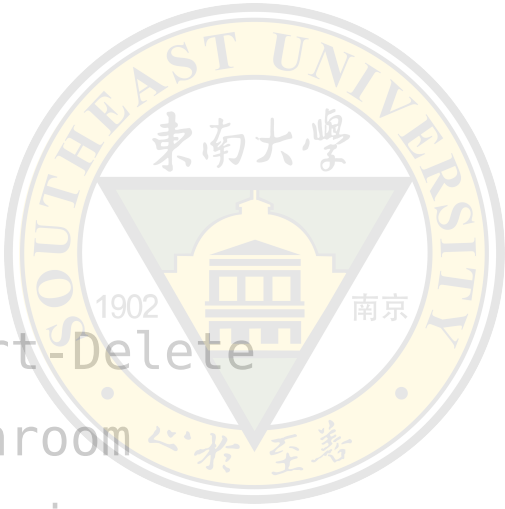
Initialization

```
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
5  t1.wait()
6      room2 += 1
7      mutex.wait()
8      room1 -= 1
9
10     if room1 == 0:
11         mutex.signal()
12         t2.signal()
13     else:
14         mutex.signal()
15         t1.signal()
```

```
21
22 t2.wait()
23     room2 -= 1
24
25     # critical section
26
27     if room2 == 0:
28         t1.signal()
29     else:
30         t2.signal()
```

Contents



Cigarette Smokers Problem

Puzzle

Four threads are involved: an agent and three smokers. The smokers loop forever, first waiting for ingredients, then making and smoking cigarettes. The ingredients are tobacco, paper, and matches.

We assume that the agent has an infinite supply of all three ingredients, and each smoker has an infinite supply of one of the ingredients; that is, one smoker has matches, another has paper, and the third has tobacco.

The agent repeatedly chooses two different ingredients at random and makes them available to the smokers. Depending on which ingredients are chosen, the smoker with the complementary ingredient should pick up both resources and proceed.

For example, if the agent puts out tobacco and paper, the smoker with the matches should pick up both ingredients, make a cigarette, and then signal the agent.

Cigarette Smokers Problem

Key

Initialization

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

Agent

```
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()
```

Smoker with tobacco

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

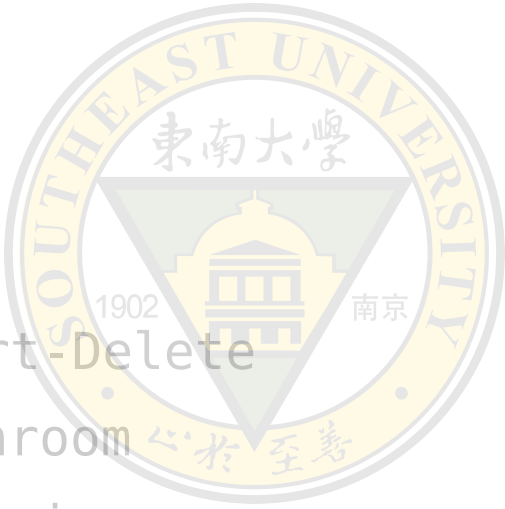
Smoker with paper

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

Smoker with match

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

Contents



The Dining Savages Problem

Puzzle

A tribe of savages eats communal dinners from a large pot that can hold M servings of stewed missionary1. When a savage wants to eat, he helps himself from the pot, unless it is empty. If the pot is empty, the savage wakes up the cook and then waits until the cook has refilled the pot.

The Dining Savages Problem

Key

Initialization

```
1  servings = 0
2  mutex = Semaphore(1)
3  emptyPot = Semaphore(0)
4  fullPot = Semaphore(0)
```

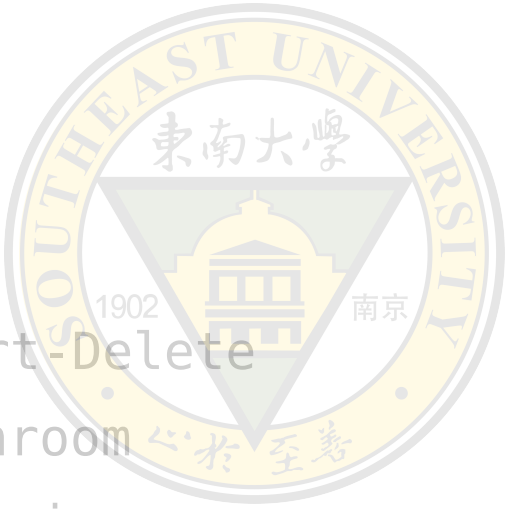
Cook

```
1  while True:
2      emptyPot.wait()
3      putServingsInPot(M)
4      fullPot.signal()
```

Savage

```
1  while True:
2      mutex.wait()
3          if servings == 0:
4              emptyPot.signal()
5              fullPot.wait()
6              servings = M
7          servings -= 1
8          getServingFromPot()
9      mutex.signal()
10
11     eat()
```

Contents



The Barbershop Problem

Puzzle

A barbershop consists of a waiting room with *n* chairs, and the barber room containing the barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. 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. Write a program to coordinate the barber and the customers.

The Barbershop Problem

Key

Initialization

```
1  n = 4
2  customers = 0
3  mutex = Semaphore(1)
4  customer = Semaphore(0)
5  barber = Semaphore(0)
6  customerDone = Semaphore(0)
7  barberDone = Semaphore(0)
```

Barber

```
1  customer.wait()
2  barber.signal()
3
4  # cutHair()
5
6  customerDone.wait()
7  barberDone.signal()
```

Customer

```
1  mutex.wait()
2      if customers == n:
3          mutex.signal()
4          balk()
5      customers += 1
6  mutex.signal()
7
8  customer.signal()
9  barber.wait()
10
11 # getHairCut()
12
13 customerDone.signal()
14 barberDone.wait()
15
16 mutex.wait()
17     customers -= 1
18 mutex.signal()
```

The FIFO Barbershop Problem

Puzzle

In the previous solution there is no guarantee that customers are served in the order they arrive. Up to `n` customers can pass the turnstile, signal `customer` and wait on `barber`. When the barber signal `barber`, any of the customers might proceed. Modify this solution so that customers are served in the order they pass the turnstile.

Hint: you can refer to the current thread as `self`, so if you write

```
self.sem = Semaphore(0),
```

each thread gets its own semaphore.

The FIFO Barbershop Problem

Key

Initialization

```
1  n = 4
2  customers = 0
3  mutex = Semaphore(1)
4  customer = Semaphore(0)
5  customerDone = Semaphore(0)
6  barberDone = Semaphore(0)
7  queue = []
```

Barber

```
1  customer.wait()
2  mutex.wait()
3      sem = queue.pop(0)
4  mutex.signal()
5
6  sem.signal()
7
8  # cutHair()
9
10 customerDone.wait()
11 barberDone.signal()
```

Customer

```
1  self.sem = Semaphore(0)
2  mutex.wait()
3      if customers == n:
4          mutex.signal()
5          balk()
6      customers += 1
7      queue.append(self.sem)
8  mutex.signal()
9
10 customer.signal()
11 self.sem.wait()
12
13 # getHairCut()
14
15 customerDone.signal()
16 barberDone.wait()
17
18 mutex.wait()
19     customers -= 1
20 mutex.signal()
```

More Complicated Barbershop Problem

Puzzle

Our barbershop has three chairs, three barbers, and a waiting area that can accommodate four customers on a sofa and that has standing room for additional customers. Fire codes limit the total number of customers in the shop to 20.
A customer will not enter the shop if it is filled to capacity with other customers. Once inside, the customer takes a seat on the sofa or stands if the sofa is filled. When a barber is free, the customer that has been on the sofa the longest is served and, if there are any standing customers, the one that has been in the shop the longest takes a seat on the sofa. When a customer's haircut is finished, any barber can accept payment, but because there is only one cash register, payment is accepted for one customer at a time. The barbers divide their time among cutting hair, accepting payment, and sleeping in their chair waiting for a customer.

Initialization

```
n = 20
customers = 0
mutex = Semaphore(1)
sofa = Semaphore(4)
customer1 = customer2 = barber =
    payment = receipt =
    Semaphore(0)
queue1 = queue2 = []
```

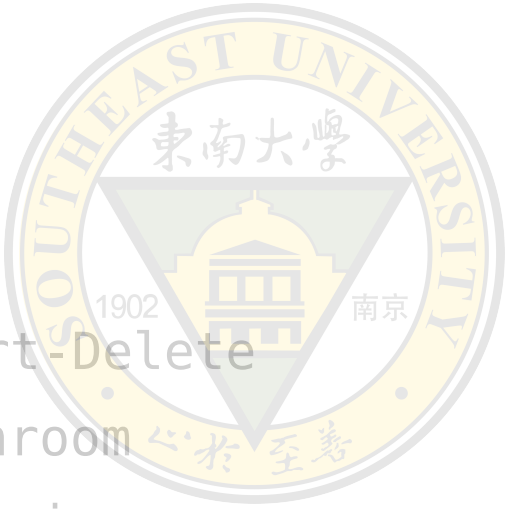
Barber

```
customer1.wait()
mutex.wait()
    sem = queue1.pop(0)
    sem.signal()
    sem.wait()
mutex.signal()
sem.signal()
customer2.wait()
mutex.wait()
    sem = queue2.pop(0)
mutex.signal()
sem.signal()
barber.signal()
# cutHair()
payment.wait()
# acceptPayment()
receipt.signal()
```

Customer

```
self.sem1 = Semaphore(0)
self.sem2 = Semaphore(0)
mutex.wait()
    if customers == n:
        mutex.signal()
        balk()
    customers += 1
    queue1.append(self.sem1)
mutex.signal()
# enterShop()
customer1.signal()
self.sem1.wait()
sofa.wait()
    # sitOnSofa()
    self.sem1.signal()
    mutex.wait()
        queue2.append(self.sem2)
    mutex.signal()
    customer2.signal()
    self.sem2.wait()
sofa.signal()
# sitInBarberChair()
# pay()
payment.signal()
receipt.wait()
mutex.wait()
    customers -= 1
mutex.signal()
```

Contents



The Santa Claus Problem

Puzzle

Santa Claus sleeps in his shop at the North Pole and can only be awakened by either (1) all nine reindeer being back from their vacation in the South Pacific, or (2) some of the elves having difficulty making toys; to allow Santa to get some sleep, the elves can only wake him when three of them have problems. When three elves are having their problems solved, any other elves wishing to visit Santa must wait for those elves to return. If Santa wakes up to find three elves waiting at his shop's door, along with the last reindeer having come back from the tropics, Santa has decided that the elves can wait until after Christmas, because it is more important to get his sleigh ready. (It is assumed that the reindeer do not want to leave the tropics, and therefore they stay there until the last possible moment.) The last reindeer to arrive must get Santa while the others wait in a warming hut before being harnessed to the sleigh.

The Santa Claus Problem

Key

Initialization

```
1  elves = reindeer = 0
2  santaSem = reindeerSem =
       Semaphore(0)
3  elfTex = mutex = Semaphore(1)
```

Elves

```
1  elfTex.wait()
2  mutex.wait()
3      elves += 1
4      if elves == 3:
5          santaSem.signal()
6      else
7          elfTex.signal()
8  mutex.signal()
9  getHelp()
10 mutex.wait()
11     elves -= 1
12     if elves == 0:
13         elfTex.signal()
14 mutex.signal()
```

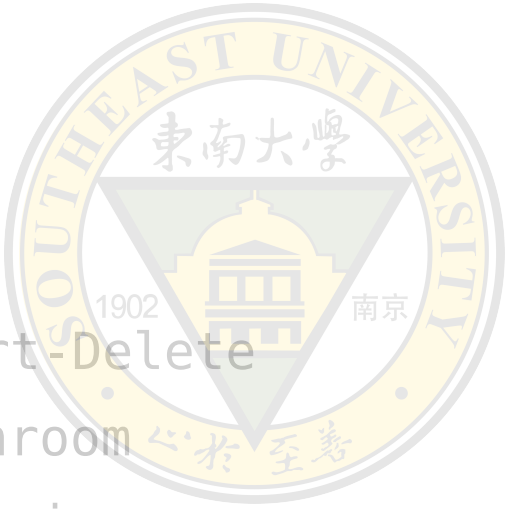
Santa

```
1  santaSem.wait()
2  mutex.wait()
3      if reindeer >= 9:
4          prepareSleigh()
5          reindeerSem.signal(9)
6          reindeer -= 9
7      else if elves == 3:
8          helpElves()
9  mutex.signal()
```

Reindeer

```
1  mutex.wait()
2      reindeer += 1
3      if reindeer == 9:
4          santaSem.signal()
5  mutex.signal()
6  reindeerSem.wait()
7  getHitched()
```

Contents



Building H_2O

Puzzle

There are two kinds of threads, oxygen and hydrogen. In order to assemble these threads into water molecules, we have to create a barrier that makes each thread wait until a complete molecule is ready to proceed.

- If an oxygen thread arrives at the barrier when no hydrogen threads are present, it has to wait for two hydrogen threads.
- If a hydrogen thread arrives at the barrier when no other threads are present, it has to wait for an oxygen thread and another hydrogen thread.

Building H_2O

Key

Initialization

```
1  mutex = Semaphore(1)
2  oxygen = hydrogen = 0
3  barrier = Barrier(3)
4  oxyQueue = hydroQueue = Semaphore(0)
```

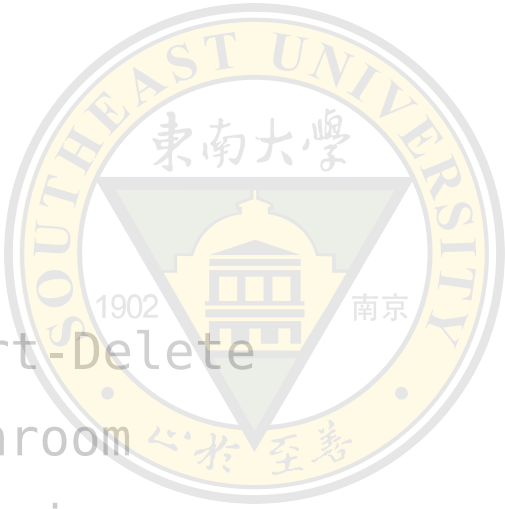
Oxygen

```
1  mutex.wait()
2  oxygen += 1
3  if hydrogen >= 2:
4      hydroQueue.signal(2)
5      hydrogen -= 2
6      oxyQueue.signal()
7      oxygen -= 1
8  else:
9      mutex.signal()
10 oxyQueue.wait()
11 bond()
12 barrier.wait()
13 mutex.signal()
```

Hydrogen

```
1  mutex.wait()
2  hydrogen += 1
3  if hydrogen >= 2 and oxygen >= 1:
4      hydroQueue.signal(2)
5      hydrogen -= 2
6      oxyQueue.signal()
7      oxygen -= 1
8  else:
9      mutex.signal()
10 hydroQueue.wait()
11 bond()
12 barrier.wait()
```

Contents



River Crossing Problem

Puzzle

Somewhere near Redmond, Washington there is a rowboat that is used by both Linux hackers and Microsoft employees (serfs) to cross a river. The ferry can hold exactly four people; it won't leave the shore with more or fewer. To guarantee the safety of the passengers, it is not permissible to put one hacker in the boat with three serfs, or to put one serf with three hackers. Any other combination is safe.

River Crossing Problem

Key

Initialization

```
1  barrier = Barrier(4)
2  mutex = Semaphore(1)
3  hackers = 0
4  serfs = 0
5  hackerQueue = Semaphore(0)
6  serfQueue = Semaphore(0)
7  local isCaptain = False
```

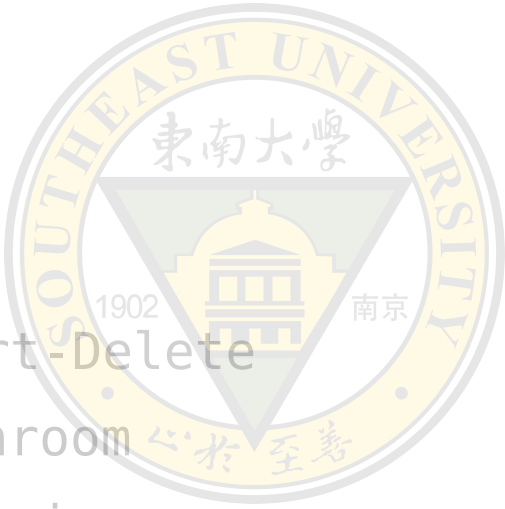
Serfs

```
1  # The serf code is symmetric
2  # Except, of course, that it is
       1000 times bigger, full of
       bugs, and it contains an
       embedded webbrowser
```

Hackers

```
1  mutex.wait()
2      hackers += 1
3      if hackers == 4:
4          hackerQueue.signal(4)
5          hackers = 0
6          isCaptain = True
7      elif hackers == 2 and serfs
           >= 2:
8          hackerQueue.signal(2)
9          serfQueue.signal(2)
10         serfs -= 2
11         hackers = 0
12         isCaptain = True
13     else:
14         mutex.signal()
15 hackerQueue.wait()
16 board()
17 barrier.wait()
18 if isCaptain:
19     rowBoat()
20     mutex.signal()
```

Contents



The Roller Coaster Problem

Puzzle

Suppose there are n passenger threads and a car thread. The passengers repeatedly wait to take rides in the car, which can hold C passengers, where $C < n$. The car can go around the tracks only when it is full.

The Roller Coaster Problem

Key

Initialization

```
1  mutex = Semaphore(1)
2  mutex2 = Semaphore(1)
3  boarders = 0
4  unboarders = 0
5  boardQueue = Semaphore(0)
6  unboardQueue = Semaphore(0)
7  allAboard = Semaphore(0)
8  allAshore = Semaphore(0)
```

Car

```
1  load()
2  boardQueue.signal(C)
3  allAboard.wait()
4
5  run()
6
7  unload()
8  unboardQueue.signal(C)
9  allAshore.wait()
```

Passenger

```
1  boardQueue.wait()
2  board()
3
4  mutex.wait()
5      boarders += 1
6      if boarders == C:
7          allAboard.signal()
8          boarders = 0
9  mutex.signal()
10
11 unboardQueue.wait()
12 unboard()
13
14 mutex2.wait()
15     unboarders += 1
16     if unboarders == C:
17         allAshore.signal()
18         unboarders = 0
19 mutex2.signal()
```

Multi-Car Roller Coaster Problem

Puzzle

Suppose there are n passenger threads and m car thread. Each car can hold C passengers, where $C < n$. The passengers repeatedly wait to take rides. Each car can go around the tracks only when it is full.

Initialization

```
mutex = Semaphore(1)
mutex2 = Semaphore(1)
boarders = 0
unboarders = 0
boardQueue = Semaphore(0)
unboardQueue = Semaphore(0)
allAboard = Semaphore(0)
allAshore = Semaphore(0)
```

Initialization

```
loadingArea = [Semaphore(0) for i
    in range(m)]
loadingArea[1].signal()
unloadingArea = [Semaphore(0) for
    i in range(m)]
unloadingArea[1].signal()
def next(i):
    return (i + 1) % m
```

Passenger

```
boardQueue.wait()
board()
mutex.wait()
    boarders += 1
    if boarders == C:
        allAboard.signal()
        boarders = 0
mutex.signal()
unboardQueue.wait()
unboard()
mutex2.wait()
    unboarders += 1
    if unboarders == C:
        allAshore.signal()
        unboarders = 0
mutex2.signal()
```

Car

```
loadingArea[i].wait()
load()
boardQueue.signal(C)
allAboard.wait()
loadingArea[next(i)].signal()

run()

unloadingArea[i].wait()
unload()
unboardQueue.signal(C)
allAshore.wait()
unloadingArea[next(i)].signal()
```

Contents

The Search-Insert-Delete Problem

Puzzle

Three kinds of threads share access to a singly-linked list: searchers, inserters and deleters. Searchers merely examine the list; hence they can execute concurrently with each other. Inserters add new items to the end of the list; insertions must be mutually exclusive to preclude two inserters from inserting new items at about the same time. However, one insert can proceed in parallel with any number of searches. Finally, deleters remove items from anywhere in the list. At most one deleter process can access the list at a time, and deletion must also be mutually exclusive with searches and insertions.

The Search-Insert-Delete Problem

Key

Initialization

```
1  insertMutex = Semaphore(1)
2  noSearcher = Semaphore(1)
3  noInserter = Semaphore(1)
4  searchSwitch = Lightswitch()
5  insertSwitch = Lightswitch()
```

Inserter

```
1  insertSwitch.wait(noInserter)
2  insertMutex.wait()
3  # critical section
4  insertMutex.signal()
5  insertSwitch.signal(noInserter)
```

Searcher

```
1  searchSwitch.wait(noSearcher)
2  # critical section
3  searchSwitch.signal(noSearcher)
```

Deleter

```
1  noSearcher.wait()
2  noInserter.wait()
3  # critical section
4  noInserter.signal()
5  noSearcher.signal()
```

Contents

The Unisex Bathroom Problem

Puzzle

- There cannot be men and women in the bathroom at the same time.
- There should never be more than three employees squandering company time in the bathroom.

The Unisex Bathroom Problem

Initialization

```
1  empty = Semaphore(1)
2  maleSwitch = Lightswitch()
3  femaleSwitch = Lightswitch()
4  maleMultiplex = Semaphore(3)
5  femaleMultiplex = Semaphore(3)
```

Female

```
1  femaleSwitch.lock(empty)
2      femaleMultiplex.wait()
3          # bathroom code here
4      femaleMultiplex.signal()
5  femaleSwitch.unlock(empty)
```

Male

```
1  maleSwitch.lock(empty)
2      maleMultiplex.wait()
3          # bathroom code here
4      maleMultiplex.signal()
5  maleSwitch.unlock(empty)
```

Lightswitch definition

```
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()
```

No-Starve Unisex Bathroom Problem

Puzzle

The problem with the previous solution is that it allows starvation. A long line of women can arrive and enter while there is a man waiting, and vice versa.
Fix the problem.

No-Starve Unisex Bathroom Problem

Key

Initialization

```
empty = Semaphore(1)
maleSwitch = Lightswitch()
femaleSwitch = Lightswitch()
maleMultiplex = Semaphore(3)
femaleMultiplex = Semaphore(3)
```

Female

```
turnstile.wait()
    maleSwitch.lock(empty)
turnstile.signal()
    maleMultiplex.wait()
        # bathroom code here
    maleMultiplex.signal()
maleSwitch.unlock(empty)
```

Male

```
turnstile.wait()
    femaleSwitch.lock(empty)
turnstile.signal()
    femaleMultiplex.wait()
        # bathroom code here
    femaleMultiplex.signal()
femaleSwitch.unlock(empty)
```

Contents

Baboon Crossing Problem

Puzzle

There is a deep canyon somewhere in Kruger National Park, South Africa, and a single rope that spans the canyon. Baboons can cross the canyon by swinging hand-over-hand on the rope, but if two baboons going in opposite directions meet in the middle, they will fight and drop to their deaths. Furthermore, the rope is only strong enough to hold 5 baboons. If there are more baboons on the rope at the same time, it will break.

Contents

The Modus Hall Problem

Puzzle

After a particularly heavy snowfall this winter, the denizens of Modus Hall created a trench-like path between their cardboard shantytown and the rest of campus. Every day some of the residents walk to and from class, food and civilization via the path; we will ignore the indolent students who chose daily to drive to Tier 3. We will also ignore the direction in which pedestrians are traveling. For some unknown reason, students living in West Hall would occasionally find it necessary to venture to the Mods.

Unfortunately, the path is not wide enough to allow two people to walk side-by-side. If two Mods persons meet at some point on the path, one will gladly step aside into the neck high drift to accommodate the other. A similar situation will occur if two ResHall inhabitants cross paths. If a Mods heathen and a ResHall prude meet, however, a violent skirmish will ensue with the victors determined solely by strength of numbers; that is, the faction with the larger population will force the other to wait.

Initialization

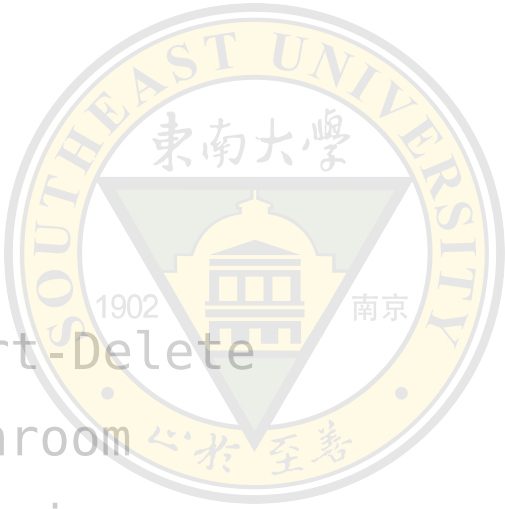
```
heathens = 0
prudes = 0
status = 'neutral'
mutex = Semaphore(1)
heathenTurn = Semaphore(1)
prudeTurn = Semaphore(1)
heathenQueue = Semaphore(0)
prudeQueue = Semaphore(0)
```

```
heathenTurn.wait()
heathenTurn.signal()

mutex.wait()
heathens++

if status == 'neutral':
    status = 'heathens rule'
    mutex.signal()
elif status == 'prudes rule':
    if heathens > prudes:
        status = 'transition to
            heathens'
        prudeTurn.wait()
    mutex.signal()
    heathenQueue.wait()
```

(Contd.)

```
elif status == 'transition to
    heathens':
    mutex.signal()
    heathenQueue.wait()
else
    mutex.signal()

# cross the field

mutex.wait()
heathens--

if heathens == 0:
    if status == 'transition to
        prudes':
        prudeTurn.signal()
    if prudes:
        prudeQueue.signal(prudes)
        status = 'prudes rule'
    else:
        status = 'neutral'

if status == 'heathens rule':
    if prudes > heathens:
        status = 'transition to
            prudes'
        heathenTurn.wait()

mutex.signal()
```

Contents



The Sushi Bar Problem

Puzzle

Imagine a sushi bar with 5 seats. If you arrive while there is an empty seat, you can take a seat immediately. But if you arrive when all 5 seats are full, that means that all of them are dining together, and you will have to wait for the entire party to leave before you sit down.

The Sushi Bar Problem

Key #1

Initialization

```
1  eating = waiting = 0
2  mutex = Semaphore(1)
3  block = Semaphore(0)
4  must_wait = False
```

Key #1

```
1  mutex.wait()
2  if must_wait:
3      waiting += 1
4      mutex.signal()
5      block.wait()
6  else:
7      eating += 1
8      must_wait = (eating == 5)
9      mutex.signal()
10
11 # eat sushi
12
13 mutex.wait()
14 eating -= 1
15 if eating == 0:
16     n = min(5, waiting)
17     waiting -= n
18     eating += n
19     must_wait = (eating == 5)
20     block.signal(n)
21 mutex.signal()
```

Initialization

```
1  eating = waiting = 0
2  mutex = Semaphore(1)
3  block = Semaphore(0)
4  must_wait = False
```

Key #2

```
1  mutex.wait()
2  if must_wait:
3      waiting += 1
4      mutex.signal()
5      block.wait()
6      waiting -= 1
7
8  eating += 1
9  must_wait = (eating == 5)
10 if waiting and not must_wait:
11     block.signal()
12 else:
13     mutex.signal()
14
15 # eat sushi
16
17 mutex.wait()
18 eating -= 1
19 if eating == 0: must_wait = False
20
21 if waiting and not must_wait:
22     block.signal()
23 else:
24     mutex.signal()
25 mutex.signal()
```

Contents

The Child Care Problem

Puzzle

At a child care center, state regulations require that there is always one adult present for every three children.

The Child Care Problem

Key

Initialization

```
1  mutex = semaphore(1)
2  multiplex = Semaphore(0)
```

Child

```
1  multiplex.wait()
2
3  # critical section
4
5  multiplex.signal()
```

Adult

```
1  multiplex.signal(3)
2
3  # critical section
4
5  mutex.wait()
6      multiplex.wait()
7      multiplex.wait()
8      multiplex.wait()
9  mutex.signal()
```

Extended Child Care Problem

Puzzle

Imagine that there are 4 children and two adults, so the value of the multiplex is 2. If one of the adults tries to leave, she will take two tokens and then block waiting for the third. If a child thread arrives, it will wait even though it would be legal to enter. From the point of view of the adult trying to leave, that might be just fine, but if you are trying to maximize the utilization of the child care center, it's not.

Initialization

```
children = adults = waiting = leaving = 0
mutex = Semaphore(1)
childQueue = Semaphore(0)
adultQueue = Semaphore(0)
```

Child

```
mutex.wait()
    if children < 3 * adults:
        children++
        mutex.signal()
    else:
        waiting++
        mutex.signal()
        childQueue.wait()

# critical section

mutex.wait()
    children--
    if leaving and children <= 3
          * (adults-1):
        leaving--
        adults--
        adultQueue.signal()
mutex.signal()
```

Adult

```
mutex.wait()
    adults++
    if waiting:
        n = min(3, waiting)
        childQueue.signal(n)
        waiting -= n
        children += n
mutex.signal()

# critical section

mutex.wait()
    if children <= 3 * (adults-1)
          :
        adults--
        mutex.signal()
    else:
        leaving++
        mutex.signal()
        adultQueue.wait()
```

Contents

The Senate Bus Problem

Puzzle

Riders come to a bus stop and wait for a bus. When the bus arrives, all the waiting riders invoke `boardBus`, but anyone who arrives while the bus is boarding has to wait for the next bus. The capacity of the bus is 50 people; if there are more than 50 people waiting, some will have to wait for the next bus.

When all the waiting riders have boarded, the bus can invoke `depart`. If the bus arrives when there are no riders, it should depart immediately.

The Senate Bus Problem

Key #1

Initialization

```
1  riders = 0
2  mutex = Semaphore(1)
3  multiplex = Semaphore(50)
4  bus = Semaphore(0)
5  allAboard = Semaphore(0)
```

Bus

```
1  mutex.wait()
2  if riders > 0:
3      bus.signal()
4      allAboard.wait()
5  mutex.signal()
6
7  depart()
```

Rider

```
1  multiplex.wait()
2      mutex.wait()
3          riders += 1
4      mutex.signal()
5
6      bus.wait()
7  multiplex.signal()
8
9  boardBus()
10
11 riders -= 1
12 if riders == 0:
13     allAboard.signal()
14 else:
15     bus.signal()
```

The Senate Bus Problem

Key #2

Initialization

```
1  waiting = 0
2  mutex = new Semaphore(1)
3  bus = new Semaphore(0)
4  boarded = new Semaphore(0)
```

Bus

```
1  mutex.wait()
2  n = min(waiting, 50)
3  for i in range(n):
4      bus.signal()
5      boarded.wait()
6
7  waiting = max(waiting-50, 0)
8  mutex.signal()
9
10 depart()
```

Rider

```
1  mutex.wait()
2      waiting += 1
3  mutex.signal()
4
5  bus.wait()
6  board()
7  boarded.signal()
```

Contents

The Faneuil Hall Problem

Puzzle

There are three kinds of threads: immigrants, spectators, and one judge. Immigrants must wait in line, check in, and then sit down. At some point, the judge enters the building. When the judge is in the building, no one may enter, and the immigrants may not leave. Spectators may leave. Once all immigrants check in, the judge can confirm the naturalization. After the confirmation, the immigrants pick up their certificates of U.S. Citizenship. The judge leaves at some point after the confirmation. Spectators may now enter as before. After immigrants get their certificates, they may leave.

Initialization

```
noJudge = Semaphore(1)
entered = 0
checked = 0
mutex = Semaphore(1)
confirmed = Semaphore(0)
```

Immigrant

```
noJudge.wait()
enter()
entered++
noJudge.signal()
mutex.wait()
checkIn()
checked++
if judge = 1 and entered ==
      checked:
    allSignedIn.signal()
else:
    mutex.signal()
sitDown()
confirmed.wait()
swear()
getCertificate()
noJudge.wait()
leave()
noJudge.signal()
```

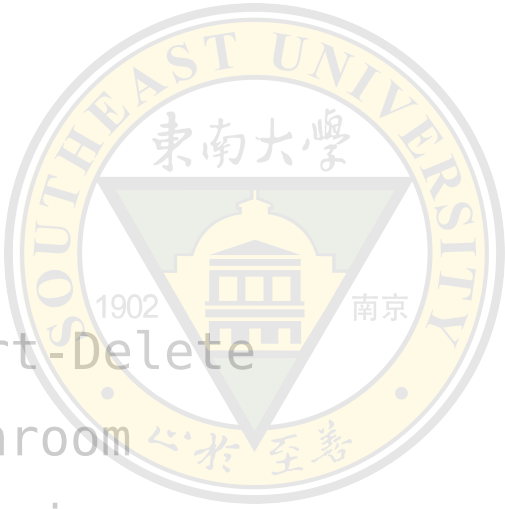
Judge

```
noJudge.wait()
mutex.wait()
enter()
judge = 1
if entered > checked:
    mutex.signal()
    allSignedIn.wait()
confirm()
confirmed.signal(checked)
entered = checked = 0
leave()
judge = 0
mutex.signal()
noJudge.signal()
```

Spectator

```
noJudge.wait()
enter()
noJudge.signal()
spectate()
leave()
```

Contents



Dining Hall Problem

Puzzle

Students in the dining hall invoke `dine` and then `leave`. After invoking `dine` and before invoking `leave` a student is considered "ready to leave".
The synchronization constraint that applies to students is that, in order to maintain the illusion of social suave, a student may never sit at a table alone. A student is considered to be sitting alone if everyone else who has invoked `dine` invokes `leave` before she has finished `dine`.

Dining Hall Problem

Key

Initialization

```
1  eating = 0
2  readyToLeave = 0
3  mutex = Semaphore(1)
4  okToLeave = Semaphore(0)
```

```
1  getFood()
2
3  mutex.wait()
4  eating++
5  if eating == 2 and readyToLeave
       == 1:
6      okToLeave.signal()
7      readyToLeave--
8  mutex.signal()
9
10 dine()
```

Contd.

```
11
12 mutex.wait()
13 eating--
14 readyToLeave++
15
16 if eating == 1 and readyToLeave
       == 1:
17     mutex.signal()
18     okToLeave.wait()
19 elif eating == 0 and readyToLeave
       == 2:
20     okToLeave.signal()
21     readyToLeave -= 2
22     mutex.signal()
23 else:
24     readyToLeave--
25     mutex.signal()
26
27 leave()
```