

# The Little Book of Semaphores

Allen B. Downey

# Par grāmatu

- Tā apkopo daudzas teorētiskas sinhronizēšanas problēmas paralēlajā skaitļošanā un to risinājumus...
- ..., lai labāk saprastu un apzinātu sinhronizēšanu, kurai pieskaras vairākosursos, bet parasti nepietiekami (autora prāt)
- Lieto Python-am līdzīgu sintaksi
- Pamata saturs ir problēmu formulējumi, papildus informācija un risinājumi stāstījuma veidā

# Grāmatas saturs

- 1. Ievads
- 2. Semafori
- 3. Parastie sinhronizēšanas lietojumi
- 4. Klasiskās sinhronizēšanas problēmas
- 5. Mazāk klasiskas sinhronizēšanas problēmas
- 6. Ne visai klasiskas sinhronizēšanas problēmas
- ...

# Grāmatas saturs

- 7. Ne tuvu klasiskas sinhronizēšanas problēmas
- 8. Sinhronizēšana Python valodā
- 9. Sinhronizēšana C valodā
- A. Python pavedienu "tīrīšana" (*cleaning up*)
- A. POSIX pavedienu "tīrīšana"

# Introduction

- In common use, “synchronization” means making two things happen at the same time.
- In computer systems, synchronization is a little more general; it refers to relationships among events—any number of events, and any kind of relationship (before, during, after).
- Computer programmers are often concerned with **synchronization constraints**, which are requirements pertaining to the order of events.

# Introduction

- **Serialization:** Event A must happen before Event B.
- **Mutual exclusion:** Events A and B must not happen at the same time.
- That's what this book is about: software techniques for enforcing synchronization constraints.

# Execution model

- For purposes of synchronization, there is no difference between the parallel model and the multithread model. The issue is the same—within one processor (or one thread) we know the order of execution, but between processors (or threads) it is impossible to tell.

# Puzzle

- Puzzle: Assuming that Bob is willing to follow simple instructions, is there any way you can guarantee that tomorrow you will eat lunch before Bob?

# Serialization with messages

You

```
a1  Eat breakfast  
a2  Work  
a3  Eat lunch  
a4  Call Bob
```

Bob

```
b1  Eat breakfast  
b2  Wait for a call  
b3  Eat lunch
```

- $a1 < a2 < a3 < a4$
- $b1 < b2 < b3$
- Combine:  $b3 > b2 > a4 > a3$

# Serialization with messages

- In this case, we would say that you and Bob ate lunch **sequentially**, because we know the order of events, and you ate breakfast **concurrently**, because we don't.
- Definition: Two events are concurrent if we cannot tell by looking at the program which will happen first.

# Non-determinism

Thread A

```
a1 print "yes"
```

Thread B

```
b1 print "no"
```

- Because the two threads run concurrently, the order of execution depends on the scheduler. During any given run of this program, the output might be “yes no” or “no yes”.

# Shared variables

- Most of the time, most variables in most threads are **local**, meaning that they belong to a single thread and no other threads can access them.
- But usually some variables are **shared** among two or more threads; this is one of the ways threads interact with each other.

# Concurrent writes

Thread A

```
a1  x = 5  
a2  print x
```

Thread B

```
b1  x = 7
```

- Puzzle: What path yields output 5 and final value 5?
- Puzzle: What path yields output 7 and final value 7?
- Puzzle: Is there a path that yields output 7 and final value 5? Can you prove it?

# Concurrent updates

Thread A

```
a1  count = count + 1
```

Thread B

```
b1  count = count + 1
```

- In machine language:

Thread A

```
a1  temp = count
a2  count = temp + 1
```

Thread B

```
b1  temp = count
b2  count = temp + 1
```

- Consider:  $a1 < b1 < b2 < a2$  (a problem)
- An operation that cannot be interrupted is said to be **atomic**.

# Semaphores

- Definition: A semaphore is like an integer, with three differences:
  - 1. When you create the semaphore, you can initialize its value to any integer, but after that the only operations you are allowed to perform are increment (increase by one) and decrement (decrease by one). You cannot read the current value of the semaphore.
  - 2. When a thread decrements the semaphore, if the result is negative, the thread blocks itself and cannot continue until another thread increments the semaphore.
  - 3. When a thread increments the semaphore, if there are other threads waiting, one of the waiting threads gets unblocked.

# Semaphores

- To say that a thread blocks itself (or simply “blocks”) is to say that it notifies the scheduler that it cannot proceed.
- The scheduler will prevent the thread from running until an event occurs that causes the thread to become unblocked.
- In the tradition of mixed metaphors in computer science, unblocking is often called “waking”.

# Semaphores

- Consequences of the definition:
  - In general, there is no way to know before a thread decrements a semaphore whether it will block or not (in specific cases you might be able to prove that it will or will not).
  - After a thread increments a semaphore and another thread gets woken up, both threads continue running concurrently. There is no way to know which thread, if either, will continue immediately.
  - When you signal a semaphore, you don't necessarily know whether another thread is waiting, so the number of unblocked threads may be zero or one.

# Semaphores

- What the value of the semaphore means:
  - If the value is positive, then it represents the number of threads that can decrement without blocking.
  - If it is negative, then it represents the number of threads that have blocked and are waiting.
  - If the value is zero, it means there are no threads waiting, but if a thread tries to decrement, it will block.

# Semaphore syntax

```
1    fred = Semaphore(1)
```

- The function Semaphore is a constructor; it creates and returns a new Semaphore.
- The initial value of the semaphore is passed as a parameter to the constructor.

# Semaphore operations

```
1    fred.increment()  
2    fred.decrement()
```

- or

```
1    fred.signal()  
2    fred.wait()
```

- `increment` and `decrement` describe what the operations do.
- `signal` and `wait` describe what they are often used for.

# Why semaphores?

- Semaphores impose deliberate constraints that help programmers avoid errors.
- Solutions using semaphores are often clean and organized, making it easy to demonstrate their correctness.
- Semaphores can be implemented efficiently on many systems, so solutions that use semaphores are portable and usually efficient.

# Signaling

- Possibly the simplest use for a semaphore is **signaling**, which means that one thread sends a signal to another thread to indicate that something has happened.

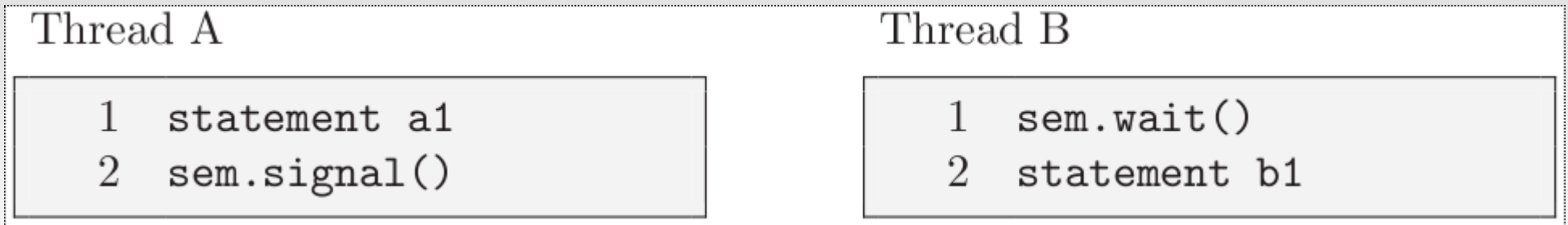
Thread A

```
1 statement a1  
2 sem.signal()
```

Thread B

```
1 sem.wait()  
2 statement b1
```

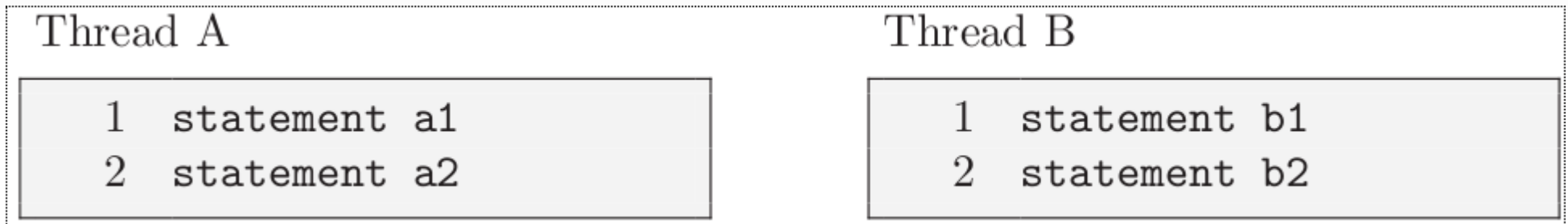
# Signaling



- The word `statement` represents an arbitrary program statement.
- To make the example concrete, imagine that `a1` reads a line from a file, and `b1` displays the line on the screen.
- The semaphore in this program guarantees that Thread A has completed `a1` before Thread B begins `b1`.

# Rendezvous

- Puzzle: Generalize the signal pattern so that it works both ways. Thread A has to wait for Thread B and vice versa. Given this code:



- ...we want to guarantee that a1 happens before b2 and b1 happens before a2.

# Rendezvous hint

- Create two semaphores, named `aArrived` and `bArrived`, and initialize them both to zero.
- As the names suggest, `aArrived` indicates whether Thread A has arrived at the rendezvous, and `bArrived` likewise.

# Rendezvous solution

Thread A

```
1 statement a1
2 aArrived.signal()
3 bArrived.wait()
4 statement a2
```

Thread B

```
1 statement b1
2 bArrived.signal()
3 aArrived.wait()
4 statement b2
```

- or (less efficient):

Thread A

```
1 statement a1
2 bArrived.wait()
3 aArrived.signal()
4 statement a2
```

Thread B

```
1 statement b1
2 bArrived.signal()
3 aArrived.wait()
4 statement b2
```

# Deadlock #1

Thread A

```
1 statement a1
2 bArrived.wait()
3 aArrived.signal()
4 statement a2
```

Thread B

```
1 statement b1
2 aArrived.wait()
3 bArrived.signal()
4 statement b2
```

- Assuming that A arrives first, it will block at its wait. When B arrives, it will also block, since A wasn't able to signal aArrived. At this point, neither thread can proceed, and never will.

# Mutex

- The mutex guarantees that only one thread accesses the shared variable at a time.
- A mutex is like a token that passes from one thread to another, allowing one thread at a time to proceed.
- Puzzle: Add semaphores to the following example to enforce mutual exclusion to the shared variable `count`.

Thread A

```
count = count + 1
```

Thread B

```
count = count + 1
```

# Mutex hint

- Create a semaphore named mutex that is initialized to 1.
- A value of one means that a thread may proceed and access the shared variable; a value of zero means that it has to wait for another thread to release the mutex.

# Mutex solution

Thread A

```
mutex.wait()  
  # critical section  
  count = count + 1  
mutex.signal()
```

Thread B

```
mutex.wait()  
  # critical section  
  count = count + 1  
mutex.signal()
```

- Since `mutex` is initially 1, whichever thread gets to the `wait` first will be able to proceed immediately.
- The second thread to arrive will have to wait until the first signals.

# Mutex

Thread A

```
mutex.wait()  
  # critical section  
  count = count + 1  
mutex.signal()
```

Thread B

```
mutex.wait()  
  # critical section  
  count = count + 1  
mutex.signal()
```

- In this example, both threads are running the same code. This is sometimes called a **symmetric** solution. If the threads have to run different code, the solution is **asymmetric**.

# Mutex

- In the metaphor we have been using so far, the mutex is a token that is passed from one thread to another.
- In an alternative metaphor, we think of the critical section as a room, and only one thread is allowed to be in the room at a time. In this metaphor, mutexes are called locks, and a thread is said to lock the mutex before entering and unlock it while exiting.

# Multiplex

- Puzzle: Generalize the previous solution so that it allows multiple threads to run in the critical section at the same time, but it enforces an upper limit on the number of concurrent threads. In other words, no more than  $n$  threads can run in the critical section at the same time.
- This pattern is called a **multiplex**.

# Multiplex solution

- To allow multiple threads to run in the critical section, just initialize the semaphore to  $n$ , which is the maximum number of threads that should be allowed.
- At any time, the value of the semaphore represents the number of additional threads that may enter.

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

# Barrier

- Puzzle: Generalize the rendezvous solution. Every thread should run the following code:

```
1 rendezvous  
2 critical point
```

# Barrier

```
1 rendezvous  
2 critical point
```

- The synchronization requirement is that no thread executes `critical point` until after all threads have executed `rendezvous`.
- You can assume that there are  $n$  threads and that this value is stored in a variable,  $n$ , that is accessible from all threads.
- When the first  $n - 1$  threads arrive they should block until the  $n$ th thread arrives, at which point all the threads may proceed.

# Barrier hint

```
1 n = the number of threads
2 count = 0
3 mutex = Semaphore(1)
4 barrier = Semaphore(0)
```

- `count` keeps track of how many threads have arrived.
- `mutex` provides exclusive access to `count` so that threads can increment it safely.
- `barrier` is locked (zero or negative) until all threads arrive; then it should be unlocked (1 or more).

# Barrier non-solution

```
1 rendezvous
2
3 mutex.wait()
4     count = count + 1
5 mutex.signal()
6
7 if count == n: barrier.signal()
8
9 barrier.wait()
10
11 critical point
```

- Puzzle: What is wrong with this solution?

# Deadlock #2

- Imagine that  $n = 5$  and that 4 threads are waiting at the barrier.
- When the 5th thread signals the barrier, one of the waiting threads is allowed to proceed, and the semaphore is incremented to -3.
- But then no one signals the semaphore again and none of the other threads can pass the barrier.
- Puzzle: Fix the problem.

# Barrier solution

```
1 rendezvous
2
3 mutex.wait()
4     count = count + 1
5 mutex.signal()
6
7 if count == n: barrier.signal()
8
9 barrier.wait()
10 barrier.signal()
11
12 critical point
```

- The only change is another `signal` after waiting at the barrier. Now as each thread passes, it signals the semaphore so that the next thread can pass.

# Barrier solution

- This pattern, a `wait` and a `signal` in rapid succession, occurs often enough that it has a name; it's called a **turnstile**, because it allows one thread to pass at a time, and it can be locked to bar all threads.
- In its initial state (zero), the turnstile is locked. The `n`th thread unlocks it and then all `n` threads go through.

# Deadlock #3

```
1 rendezvous
2
3 mutex.wait()
4     count = count + 1
5     if count == n: barrier.signal()
6
7     barrier.wait()
8     barrier.signal()
9 mutex.signal()
10
11 critical point
```

- A common source of deadlocks: blocking on a semaphore while holding a mutex.

# Reusable barrier

- Puzzle: Rewrite the barrier solution so that after all the threads have passed through, the turnstile is locked again.

# Reusable barrier non-solution #1

```
1 rendezvous
2
3 mutex.wait()
4     count += 1
5 mutex.signal()
6
7 if count == n: turnstile.signal()
8
9 turnstile.wait()
10 turnstile.signal()
11
12 critical point
13
14 mutex.wait()
15     count -= 1
16 mutex.signal()
17
18 if count == 0: turnstile.wait()
```

- Puzzle: What is the problem?

# Reusable barrier problem #1

- There is a problem spot at Line 7.
- If the  $n - 1$ th thread is interrupted at this point, and then the  $n$ th thread comes through the mutex, both threads will find that `count==n` and both threads will signal the turnstile. In fact, it is even possible that all the threads will signal the turnstile.
- Similarly, at Line 18 it is possible for multiple threads to `wait`, which will cause a deadlock.
- Puzzle: Fix the problem.

# Reusable barrier non-solution #2

```
1 rendezvous
2
3 mutex.wait()
4     count += 1
5     if count == n: turnstile.signal()
6 mutex.signal()
7
8 turnstile.wait()
9 turnstile.signal()
10
11 critical point
12
13 mutex.wait()
14     count -= 1
15     if count == 0: turnstile.wait()
16 mutex.signal()
```

- Puzzle: Identify and fix the problem.

# Reusable barrier hint

```
1  turnstile = Semaphore(0)
2  turnstile2 = Semaphore(1)
3  mutex = Semaphore(1)
```

- Initially the first is locked and the second is open. When all the threads arrive at the first, we lock the second and unlock the first. When all the threads arrive at the second we relock the first.

# Reusable barrier solution

```
1 # rendezvous
2
3 mutex.wait()
4     count += 1
5     if count == n:
6         turnstile2.wait()      # lock the second
7         turnstile.signal()    # unlock the first
8 mutex.signal()
9
10 turnstile.wait()              # first turnstile
11 turnstile.signal()
12
13 # critical point
14
15 mutex.wait()
16     count -= 1
17     if count == 0:
18         turnstile.wait()      # lock the first
19         turnstile2.signal()   # unlock the second
20 mutex.signal()
21
22 turnstile2.wait()             # second turnstile
23 turnstile2.signal()
```

# Reusable barrier solution

- This solution is sometimes called a **two-phase barrier** because it forces all the threads to wait twice: once for all the threads to arrive and again for all the threads to execute the critical section.
- A "proof":
  - 1. Only the nth thread can lock or unlock the turnstiles.
  - 2. Before a thread can unlock the first turnstile, it has to close the second, and vice versa;

# Preloaded turnstile

- One nice thing about a turnstile is that it is a versatile component you can use in a variety of solutions. But one drawback is that it forces threads to go through sequentially, which may cause more context switching than necessary.
- In the reusable barrier solution, we can simplify the solution if the thread that unlocks the turnstile preloads the turnstile with enough signals to let the right number of threads through.

# Preloaded turnstile

```
1 # rendezvous
2
3 mutex.wait()
4     count += 1
5     if count == n:
6         turnstile.signal(n)    # unlock the first
7 mutex.signal()
8
9 turnstile.wait()                # first turnstile
10
11 # critical point
12
13 mutex.wait()
14     count -= 1
15     if count == 0:
16         turnstile2.signal(n)  # unlock the second
17 mutex.signal()
18
19 turnstile2.wait()              # second turnstile
```

# Barrier objects

```
1 class Barrier:
2     def __init__(self, n):
3         self.n = n
4         self.count = 0
5         self.mutex = Semaphore(1)
6         self.turnstile = Semaphore(0)
7         self.turnstile2 = Semaphore(0)
8
9     def phase1(self):
10        self.mutex.wait()
11            self.count += 1
12            if self.count == self.n:
13                self.turnstile.signal(self.n)
14        self.mutex.signal()
15        self.turnstile.wait()
16
17    def phase2(self):
18        self.mutex.wait()
19            self.count -= 1
20            if self.count == 0:
21                self.turnstile2.signal(self.n)
22        self.mutex.signal()
23        self.turnstile2.wait()
24
25    def wait(self):
26        self.phase1()
27        self.phase2()
```

# Queue

- In this case, the initial semaphore value is 0, and usually the code is written so that it is not possible to signal unless there is a thread waiting, so the value of the semaphore is never positive.
- Puzzle: write code for leaders and followers that enforces these constraints.

# Queue hint

```
1 leaderQueue = Semaphore(0)
2 followerQueue = Semaphore(0)
```

- leaderQueue is the queue where leaders wait and followerQueue is the queue where followers wait.

# Queue solution

- Leaders:

```
1 followerQueue.signal()  
2 leaderQueue.wait()  
3 dance()
```

- Followers:

```
1 leaderQueue.signal()  
2 followerQueue.wait()  
3 dance()
```

- Whether they actually proceed in pairs is not clear.

# Queue solution

- Let's add the additional constraint that each leader can invoke dance concurrently with only one follower, and vice versa.
- Puzzle: write a solution to this “exclusive queue” problem.

# Exclusive queue hint

```
1 leaders = followers = 0
2 mutex = Semaphore(1)
3 leaderQueue = Semaphore(0)
4 followerQueue = Semaphore(0)
```

- `leaders` and `followers` are counters that keep track of the number of dancers of each kind that are waiting.
- `mutex` guarantees exclusive access to the counters.
- `leaderQueue` and `followerQueue` are the queues where dancers wait.

# Exclusive queue solution

- Leaders:

```
1  mutex.wait()
2  if followers > 0:
3      followers--
4      followerQueue.signal()
5  else:
6      leaders++
7      mutex.signal()
8      leaderQueue.wait()
9
10 dance()
11 mutex.signal()
```

# Exclusive queue solution

- Followers:

```
1  mutex.wait()
2  if leaders > 0:
3      leaders--
4      leaderQueue.signal()
5  else:
6      followers++
7      mutex.signal()
8      followerQueue.wait()
9
10 dance()
```

# Exclusive queue solution

- When a leader arrives, it gets the mutex. If there is a follower waiting, the leader decrements followers, signals a follower, and then invokes dance, all before releasing mutex. That guarantees that there can be only one follower thread running dance concurrently.
- When a follower arrives, it checks for a waiting leader. If there is one, the follower decrements leaders, signals a leader, and executes dance, all without releasing mutex.
- When a semaphore is used as a queue, I find it useful to read “wait” as “wait for this queue” and signal as “let someone from this queue go.”

# Fifo queue

- Puzzle: use semaphores to build a first-in-first-out queue. Each time the Fifo is signaled, the thread at the head of the queue should proceed. If more than one thread is waiting on a semaphore, you should not make any assumptions about which thread will proceed when the semaphore is signaled.

# Fifo queue hint

- Allocate one semaphore to each thread by having each thread run the following initialization:

```
1 local mySem = Semaphore(0)
```

- Fifo class definition might look like:

```
1 class Fifo:  
2     def __init__(self):  
3         self.queue = Queue()  
4         self.mutex = Semaphore(1)
```

# Fifo queue solution

```
1 class Fifo:
2     def __init__(self):
3         self.queue = Queue()
4         self.mutex = Semaphore(1)
5
6     def wait():
7         self.mutex.wait()
8         self.queue.add(mySem)
9         self.mutex.signal()
10        mySem.wait()
11
12    def signal():
13        self.mutex.wait()
14        sem = self.queue.remove()
15        self.mutex.signal()
16        sem.signal()
```

# Producer-consumer problem

- Event-driven programs are a good example.
  - An “event” is something that happens that requires the program to respond: the user presses a key or moves the mouse, a block of data arrives from the disk, a packet arrives from the network, a pending operation completes.
  - Whenever an event occurs, a producer thread creates an event object and adds it to the event buffer. Concurrently, consumer threads take events out of the buffer and process them. In this case, the consumers are called “event handlers.”

# Producer-consumer problem

- There are several synchronization constraints that we need to enforce to make this system work correctly:
  - While an item is being added to or removed from the buffer, the buffer is in an inconsistent state. Therefore, threads must have exclusive access to the buffer.
  - If a consumer thread arrives while the buffer is empty, it blocks until a producer adds a new item.

# Producer-consumer problem

- Assume producers:

```
1 event = waitForEvent()  
2 buffer.add(event)
```

- ... and consumers:

```
1 event = buffer.get()  
2 event.process()
```

- Puzzle: Add synchronization statements to the producer and consumer code to enforce the synchronization constraints.

# Producer-consumer hint

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

- `mutex` provides exclusive access to the buffer. When `items` is positive, it indicates the number of items in the buffer. When it is negative, it indicates the number of consumer threads in queue.
- `event` is a **local variable**, which in this context means that each thread has its own version.

# Producer-consumer hint

- Sometimes it is useful to attach a variable to each thread.
- This can be implemented in different environments:
  - If each thread has its own run-time stack, then any variables allocated on the stack are thread-specific.
  - If threads are represented as objects, we can add an attribute to each thread object.
  - If threads have unique IDs, we can use the IDs as an index into an array or hash table, and store per-thread data there.

# Producer-consumer solution

- Producer:

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

- Several threads can run `waitForEvent` concurrently.
- The `items` semaphore keeps track of the number of items in the buffer. Each time the producer adds an item, it signals `items`, incrementing it by one.

# Producer-consumer solution

- Consumer:

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

- The buffer operation is protected by a mutex, but before the consumer gets to it, it has to decrement `items`. If `items` is zero or negative, the consumer blocks until a producer signals.

# Readers-writers problem

- Readers and writers execute different code before entering the critical section. The synchronization constraints are:
  - Any number of readers can be in the critical section simultaneously.
  - Writers must have exclusive access to the critical section.
- So, a writer cannot enter the critical section while any other thread (reader or writer) is there, and while the writer is there, no other thread may enter.
- The exclusion pattern here might be called **categorical mutual exclusion**.

# Readers-writers solution

- ...
- Patterns similar to this reader code are common: the first thread into a section locks a semaphore (or queues) and the last one out unlocks it.
- The name of the pattern is **Lightswitch**, by analogy with the pattern where the first person into a room turns on the light (locks the mutex) and the last one out turns it off (unlocks the mutex).

# Starvation

- If a writer arrives while there are readers in the critical section, it might wait in queue forever while readers come and go. As long as a new reader arrives before the last of the current readers departs, there will always be at least one reader in the room.
- This situation is not a deadlock, because some threads are making progress, but it is not exactly desirable...

# No-starve mutex

- At a more basic level, we have to address the issue of **thread starvation**, which is the possibility that one thread might wait indefinitely while others proceed.
- For most concurrent applications, starvation is unacceptable, so we enforce the requirement of **bounded waiting**, which means that the time a thread waits on a semaphore (or anywhere else, for that matter) has to be provably finite.

# Assumptions about the scheduler

- Property 1: if there is only one thread that is ready to run, the scheduler has to let it run.
  - If we can assume Property 1, then we can build a system that is provably free of starvation.
- Property 2: if a thread is ready to run, then the time it waits until it runs is bounded.

# Assumptions about the scheduler

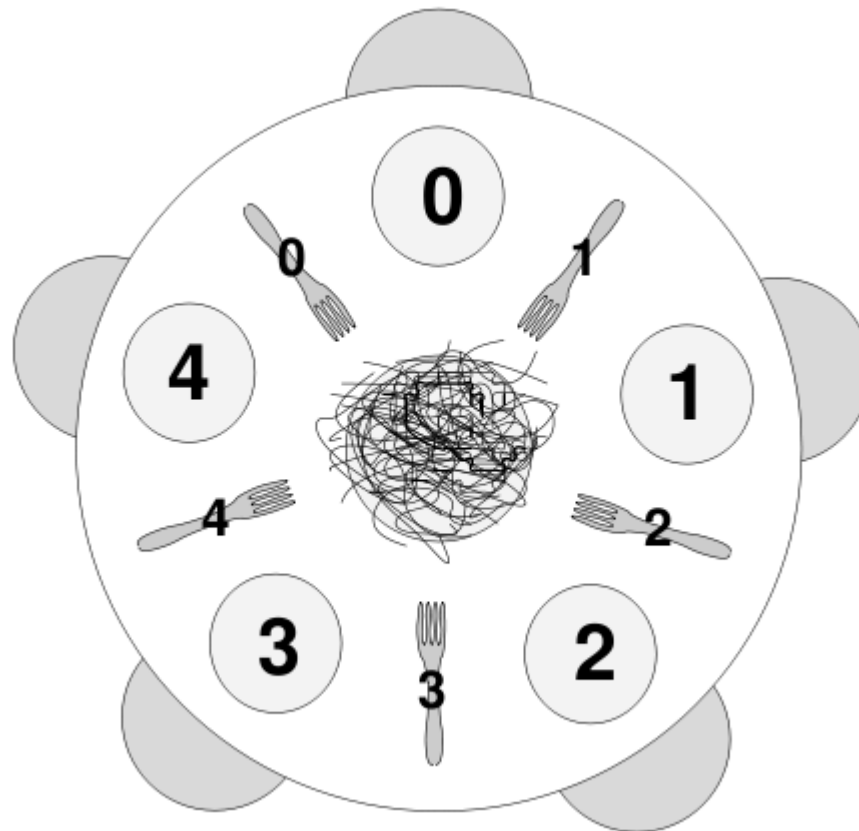
- Property 3: if there are threads waiting on a semaphore when a thread executes `signal`, then one of the waiting threads has to be woken.
  - With Property 3, it becomes possible to avoid starvation.

# Assumptions about the scheduler

- Property 4: if a thread is waiting at a semaphore, then the number of threads that will be woken before it is bounded.
  - A semaphore that has Property 4 is sometimes called a **strong semaphore**; one that has only Property 3 is called a **weak semaphore**.

# Dining philosophers

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



# Cigarette smokers problem

- 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.

# Cigarette smokers problem

- 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.

# Less classical synchronization problems

- The dining savages problem
- The barbershop problem
- Hilzer's Barbershop problem
- The Santa Claus problem
- Building H<sub>2</sub>O
- River crossing problem
- The roller coaster problem

# Not-so-classical problems

- The search-insert-delete problem
- The unisex bathroom problem
- Baboon crossing problem
- The Modus Hall Problem

# Not remotely classical problems

- The sushi bar problem
- The child care problem
- The room party problem
- The Senate Bus problem
- The Faneuil Hall problem
- Dining Hall problem

# Synchronization in Python

```
1 from threading_cleanup import *
2
3 class Shared:
4     def __init__(self):
5         self.counter = 0
6
7 def child_code(shared):
8     while True:
9         shared.counter += 1
10        print shared.counter
11        time.sleep(0.5)
12
13 shared = Shared()
14 children = [Thread(child_code, shared) for i in range(2)]
15 for child in children: child.join()
```

# Mutex checker solution

```
1 def child_code(shared):
2     while True:
3         shared.mutex.wait()
4         if shared.counter < shared.end:
5             shared.array[shared.counter] += 1
6             shared.counter += 1
7             shared.mutex.signal()
8         else:
9             shared.mutex.signal()
10            break
```

# Synchronization in C

```
1 typedef struct {
2     int counter;
3     int end;
4     int *array;
5 } Shared;
6
7 Shared *make_shared (int end)
8 {
9     int i;
10    Shared *shared = check_malloc (sizeof (Shared));
11
12    shared->counter = 0;
13    shared->end = end;
14
15    shared->array = check_malloc (shared->end * sizeof(int));
16    for (i=0; i<shared->end; i++) {
17        shared->array[i] = 0;
18    }
19    return shared;
20 }
```

# Parent code

```
1  int main ()
2  {
3      int i;
4      pthread_t child[NUM_CHILDREN];
5
6      Shared *shared = make_shared (100000);
7
8      for (i=0; i<NUM_CHILDREN; i++) {
9          child[i] = make_thread (entry, shared);
10     }
11
12     for (i=0; i<NUM_CHILDREN; i++) {
13         join_thread (child[i]);
14     }
15
16     check_array (shared);
17     return 0;
18 }
```

# Child code

```
1 void child_code (Shared *shared)
2 {
3     while (1) {
4         shared->array[shared->counter]++;
5
6         shared->counter++;
7         if (shared->counter >= shared->end) {
8             return;
9         }
10    }
11 }
```

- If everything works correctly, each element of the array should be incremented once.

# Synchronization errors

- <http://greenteapress.com/semaphores/counter.c>
- `gcc -lpthread counter.c`
- `./a.out`

```
99970000  
99980000  
99990000  
Child done.  
Child done.  
Checking...  
2440124 errors.
```

- Puzzle: use semaphores to enforce exclusive access to the shared variables and run the program again to confirm that there are no errors.

# Mutual exclusion hint

```
1 typedef struct {
2     int counter;
3     int end;
4     int *array;
5     Semaphore *mutex;
6 } Shared;
7
8 Shared *make_shared (int end)
9 {
10     int i;
11     Shared *shared = check_malloc (sizeof (Shared));
12
13     shared->counter = 0;
14     shared->end = end;
15
16     shared->array = check_malloc (shared->end * sizeof(int));
17     for (i=0; i<shared->end; i++) {
18         shared->array[i] = 0;
19     }
20     shared->mutex = make_semaphore(1);
21     return shared;
22 }
```

# Mutual exclusion solution

- [http://greenteapress.com/semaphores/counter\\_mutex.c](http://greenteapress.com/semaphores/counter_mutex.c)

```
1 void child_code (Shared *shared)
2 {
3     while (1) {
4         sem_wait(shared->mutex);
5         shared->array[shared->counter]++;
6
7         shared->counter++;
8         if (shared->counter >= shared->end) {
9             sem_signal(shared->mutex);
10            return;
11        }
12        sem_signal(shared->mutex);
13    }
14 }
```

# Make your own semaphores

- The most commonly used synchronization tools for programs that use Pthreads are mutexes and condition variables, not semaphores.

# Semaphore implementation hint

```
1 typedef struct {
2     int value, wakeups;
3     Mutex *mutex;
4     Cond *cond;
5 } Semaphore;
```

```
1 Semaphore *make_semaphore (int value)
2 {
3     Semaphore *semaphore = check_malloc (sizeof(Semaphore));
4     semaphore->value = value;
5     semaphore->wakeups = 0;
6     semaphore->mutex = make_mutex ();
7     semaphore->cond = make_cond ();
8     return semaphore;
9 }
```

# Semaphore implementation

```
1 void sem_wait (Semaphore *semaphore)
2 {
3     mutex_lock (semaphore->mutex);
4     semaphore->value--;
5
6     if (semaphore->value < 0) {
7         do {
8             cond_wait (semaphore->cond, semaphore->mutex);
9         } while (semaphore->wakeups < 1);
10        semaphore->wakeups--;
11    }
12    mutex_unlock (semaphore->mutex);
13 }
14
15 void sem_signal (Semaphore *semaphore)
16 {
17     mutex_lock (semaphore->mutex);
18     semaphore->value++;
19
20     if (semaphore->value <= 0) {
21         semaphore->wakeups++;
22         cond_signal (semaphore->cond);
23     }
24     mutex_unlock (semaphore->mutex);
25 }
```

**Paldies par uzmanību!**

Jautājumi un atbildes