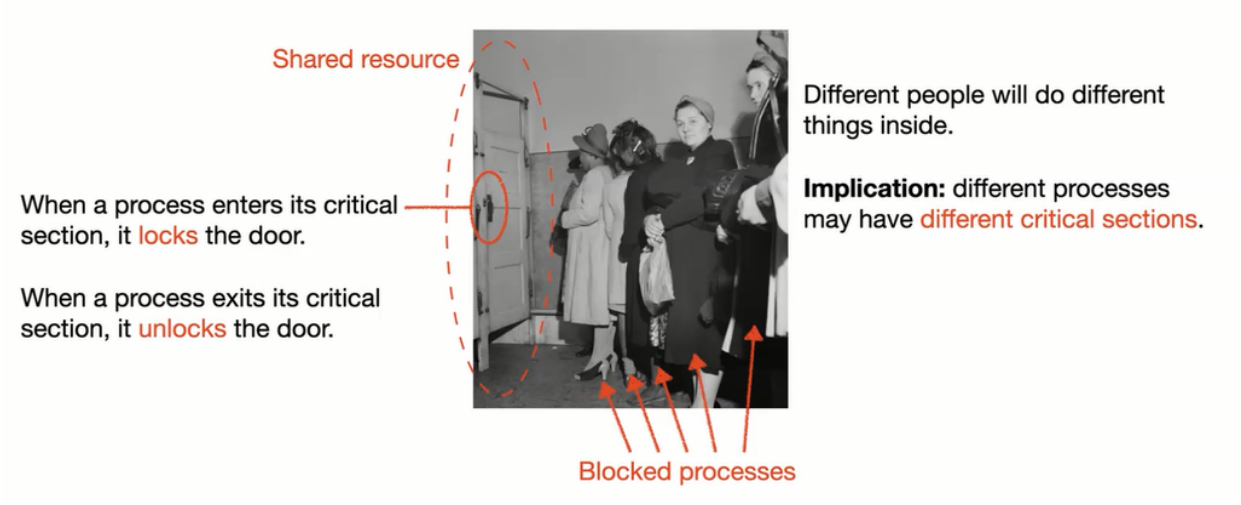
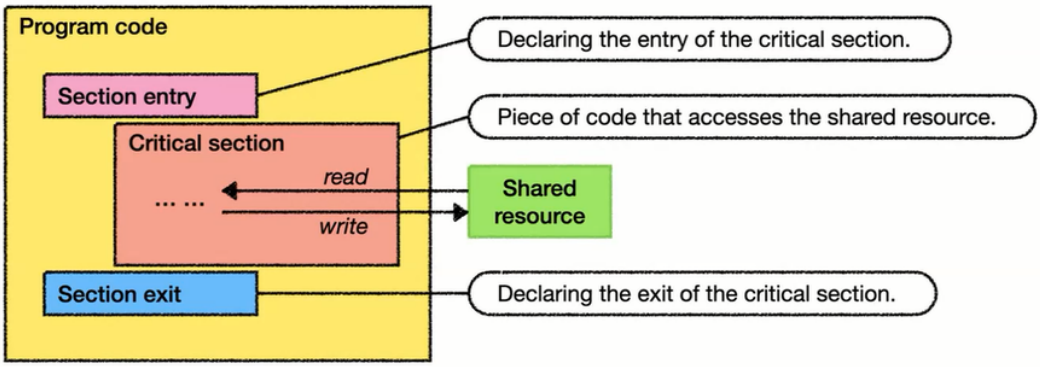
OS: Lecture 11

Review: Critical Sections



A **critical section** (*a.k.a.* critical region) is a piece of code that accesses a shared resource.



- Requirements

Requirements

No two processes may be simultaneously inside their critical sections.

- This is the mutual exclusion requirement: when one process is inside its critical section, any attempts to go inside the critical sections by other processes are not allowed.

No assumptions may be made about the speeds or the number of CPUs.

- The solution cannot depend on the time spent inside the critical section or assume the number of CPUs in the system.

No process running outside its critical section may block other processes.

- This ensures all processes can make progress. Otherwise, it may end up with a scenario where all processes are blocked but no process is inside its critical section.

No process should have to wait forever to enter its critical section.

- This guarantees bounded waiting, *i.e.*, no process will starve to death.

- Attempts

Attempt #1: disabling interrupts.

Attempt #2: using a “lock” variable.

Attempt #3: strict alternation.

Attempt #4: Peterson’s algorithm.

Attempt #5: spinlocks.

Attempt #6: semaphores.

- Attempt 5

```
int test_and_set(int *ptr, int new) {
  // this code executes atomically
  int old = *ptr;
  *ptr = new;
  return old;
}
```

How to implement `section_entry()` and `section_exit()` using the atomic `test_and_set` instruction?

```
int lock = 0;  // 0: available, 1: held
```

```
void section_entry(int *lock) {

  while (test_and_set(lock, 1) == 1)
    ;  // busy waiting

}
```

```
void section_exit(int *lock) {

  *lock = 0;

}
```

- It's a good solution with some limitations
  - The critical section has to be small
  - The number of processes should not be more than the number of cores in your system

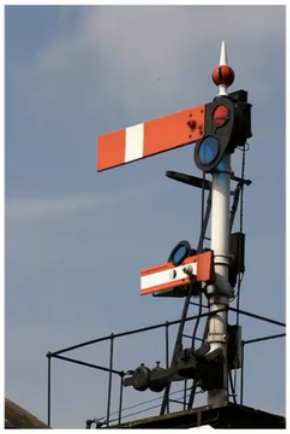
- Attempt 6

A **semaphore** is an object with a non-negative integer value.

- The value must be initialized before being used.

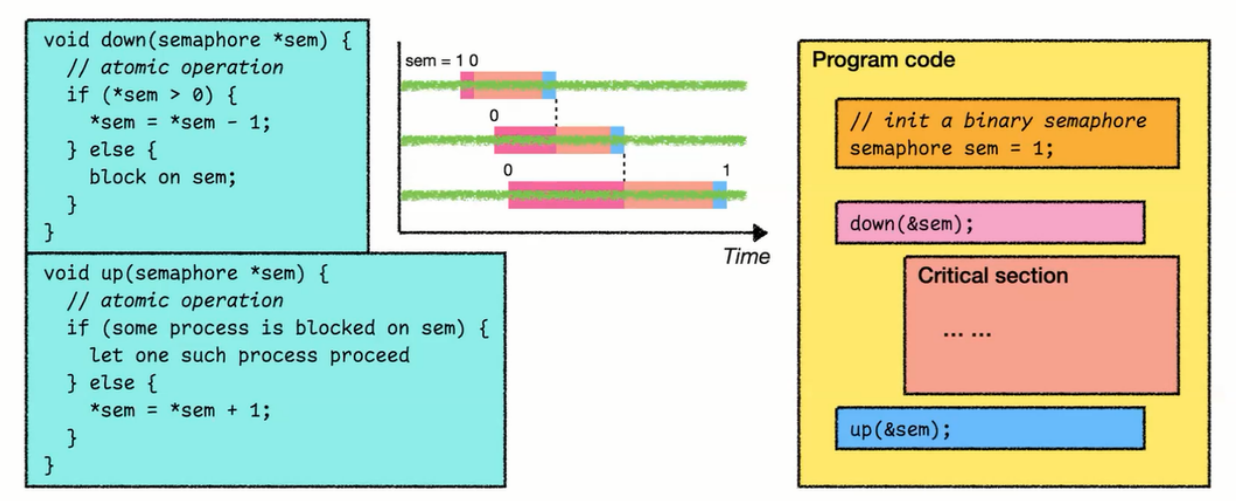
There are two operations: `down()` and `up()`.

- See “`man 7 sem_overview`” for more details.



```
void down(semaphore *sem) {
  // atomic operation
  if (*sem > 0) {
    *sem = *sem - 1;
  } else {
    block on sem;
  }
}
```

```
void up(semaphore *sem) {
  // atomic operation
  if (some process is blocked on sem) {
    let one such process proceed
  } else {
    *sem = *sem + 1;
  }
}
```



Classical IPC problems

The producer-consumer problem

- It models access to a bounded buffer.

The dining philosopher problem

- It models processes competing for exclusive access to a limited number of resources (*e.g.*, I/O devices).

The readers and writers problem

- It models access to a database.

The sleeping barber problem

- It models a queueing situation (♫*please stay on the line…*♪♪).

The producer-consumer problem

The producer-consumer problem

- It models access to a bounded buffer.

The dining philosopher problem

- It models processes competing for exclusive access to a limited number of resources (*e.g.*, I/O devices).

The readers and writers problem

- It models access to a database.

The sleeping barber problem

- It models a queueing situation (♫*please stay on the line…*♪♪).

a.k.a. the bounded-buffer problem

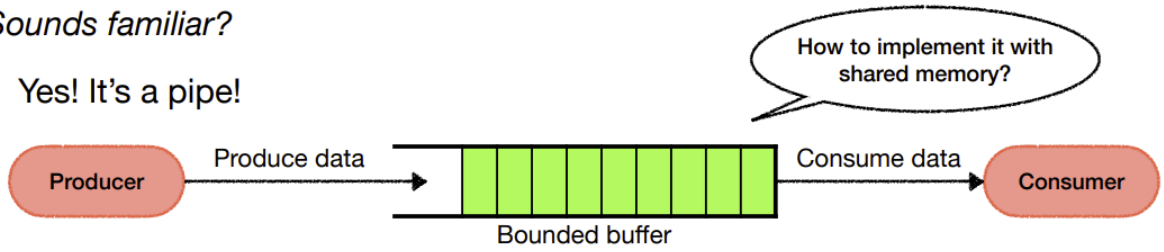
Two processes share a fixed-size buffer.

The **producer** inserts data to the tail of the buffer.

The **consumer** removes data from the head of the buffer.

Sounds familiar?

- Yes! It's a pipe!



- Requirements

Synchronization requirement #1

When the **producer** wants to insert an item into the buffer, but the buffer is already full…

What should the producer and the consumer do?

- The **producer** should block.
- The **consumer** should wake up the producer after it has consumed an item.

Synchronization requirement #2

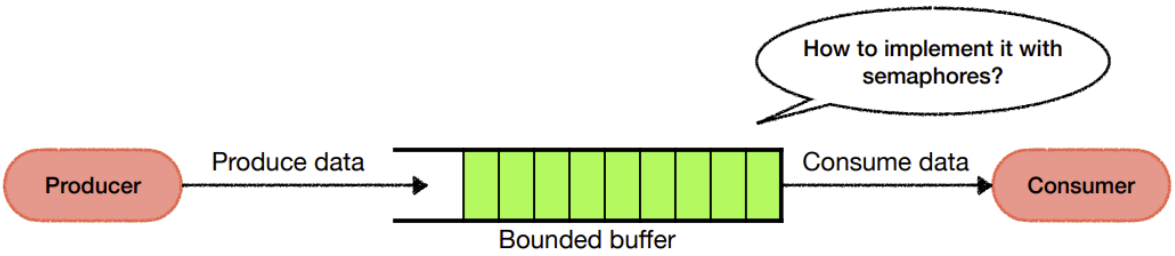
When the **consumer** wants to remove an item from the buffer, but the buffer is empty…

What should the producer and the consumer do?

- The **consumer** should block.
- The **producer** should wake up the consumer after it has produced an item.

Of course, we also need **mutual exclusion**…

- No two processes can access the shared buffer at the same time.



- Implementation using semaphores

```
semaphore mutex = 1;           // controls access to critical section
semaphore empty = BUFFER_SIZE; // counts empty buffer slots
semaphore filled = 0;          // counts filled buffer slots
```

Why do we need three semaphores?

```
1  void producer() {
2    int item;
3
4    for (;;) {
5      item = produce_item();
6      down(&empty);  // decrement empty count
7      down(&mutex);  // enter critical section
8      insert(item);  // put new item in buffer
9      up(&mutex);    // exit critical section
10     up(&filled);   // increment filled count
11   }
12 }
```

```
1  void consumer() {
2    int item;
3
4    for (;;) {
5      down(&filled);    // decrement filled count
6      down(&mutex);     // enter critical section
7      item = remove();  // take item from buffer
8      up(&mutex);       // exit critical section
9      up(&empty);       // increment empty count
10     consume_item(item);
11   }
12 }
```

- Only `insert()` and `remove()` are between the `down(mutex)` and `up(mutex)`,
  - as we want our critical sections to be as short as possible
- `empty` counts the number of empty buffer slots
  - `down(&empty)`, if the buffer is already full and *empty* = 0, the **producer** is blocked
  - `up(&empty)` means that the **consumer** has removed something from the buffer and there's one more empty slot
    - If the **producer** was blocked, the consumer will wake up the producer by adding the *empty*
    - If it's not blocked, it means we still have plenty of empty space here
- `filled` counts the number of filled buffer slot, initial = 0
-

`mutex` guarantees mutual exclusion.

`empty` and `filled` are for synchronization.

- `empty` represents the number of empty slots.
  - The **producer** blocks when this value is 0, *i.e.*, there is no empty slot, *i.e.*, the buffer is already full.
- `filled` represents the number of filled slots.
  - The **consumer** blocks when this value is 0, *i.e.*, there is no filled slot, *i.e.*, the buffer is empty.

Invariant: `empty` + `filled` = buffer size.

```
semaphore mutex = 1;           // controls access to critical section
semaphore empty = BUFFER_SIZE; // counts empty buffer slots
semaphore filled = 0;          // counts filled buffer slots
```

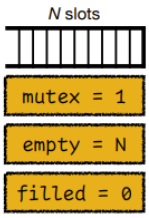
Can we swap lines 6 & 7 in the producer?

```
1  void producer() {
2    int item;
3
4    for (;;) {
5      item = produce_item();
6      down(&empty);  // decrement empty count
7      down(&mutex);  // enter critical section
8      insert(item);  // put new item in buffer
9      up(&mutex);    // exit critical section
10     up(&filled);   // increment filled count
11   }
12 }
```

```
1  void consumer() {
2    int item;
3
4    for (;;) {
5      down(&filled);    // decrement filled count
6      down(&mutex);     // enter critical section
7      item = remove();  // take item from buffer
8      up(&mutex);       // exit critical section
9      up(&empty);       // increment empty count
10     consume_item(item);
11   }
12 }
```

- If we swapped lines 6 & 7 in the producer…

If we swapped lines 6 & 7 in the producer…

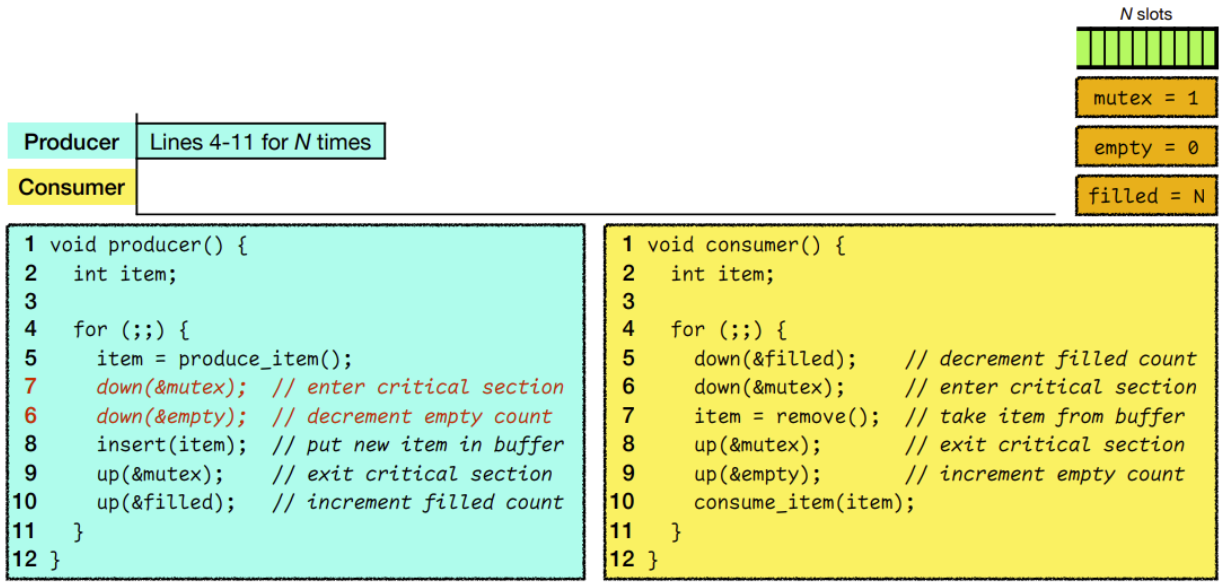


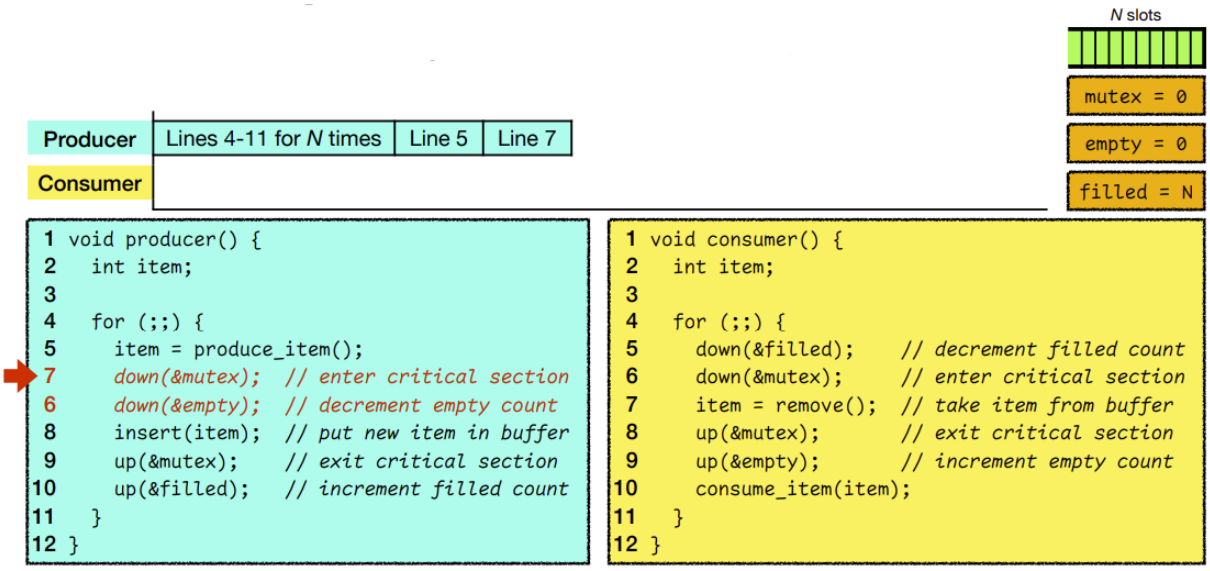
Producer

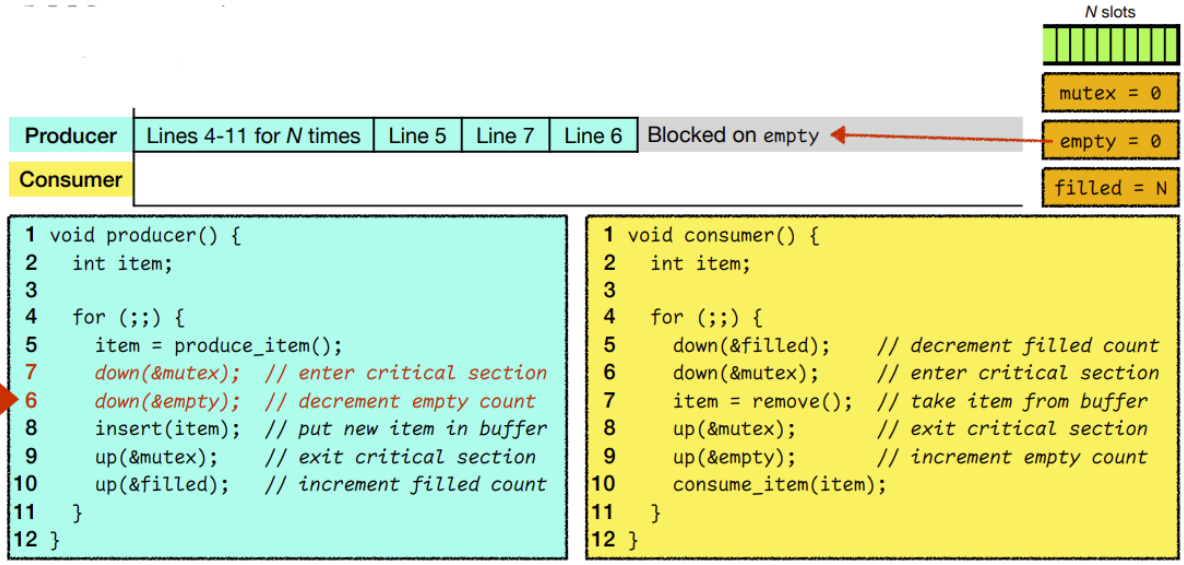
Consumer

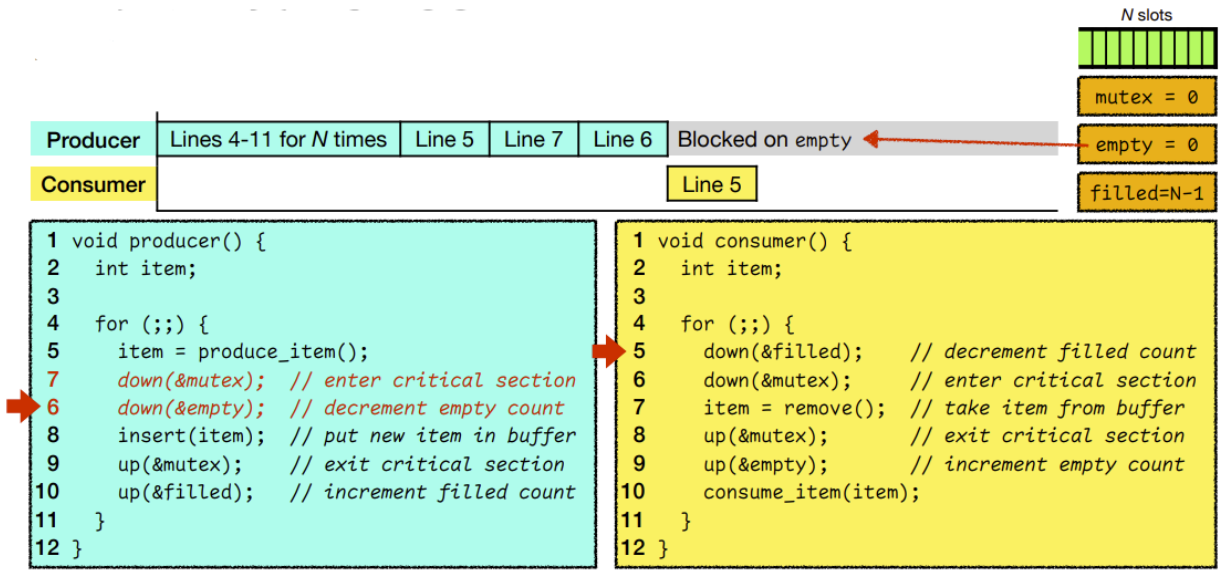
```
1  void producer() {
2    int item;
3
4    for (;;) {
5      item = produce_item();
7      down(&mutex);  // enter critical section
6      down(&empty);  // decrement empty count
8      insert(item);  // put new item in buffer
9      up(&mutex);    // exit critical section
10     up(&filled);   // increment filled count
11   }
12 }
```

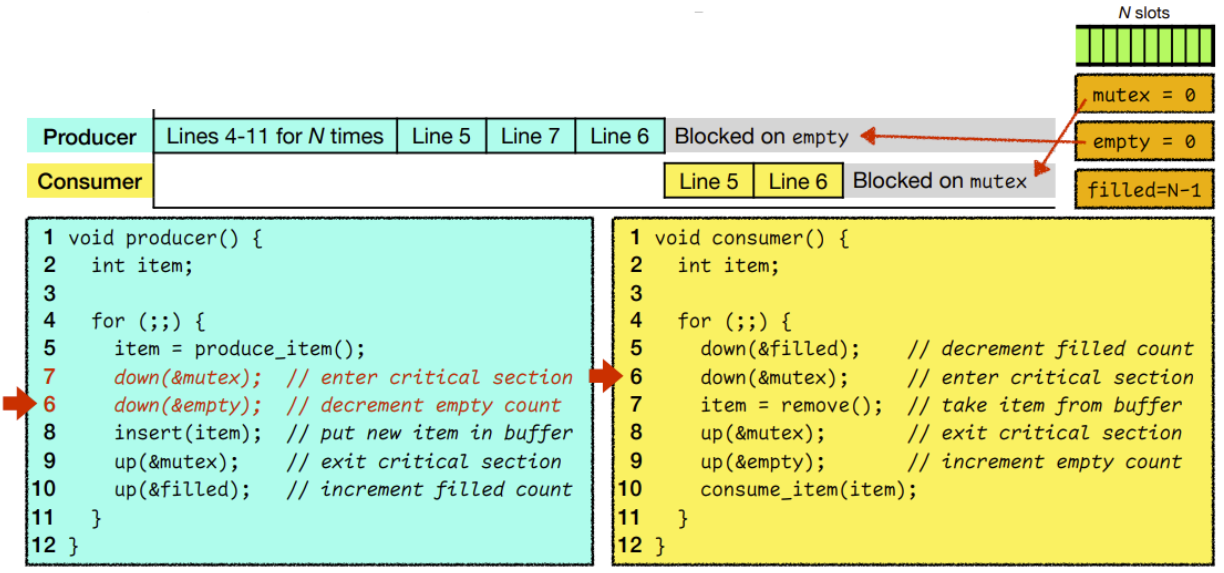
```
1  void consumer() {
2    int item;
3
4    for (;;) {
5      down(&filled);    // decrement filled count
6      down(&mutex);     // enter critical section
7      item = remove();  // take item from buffer
8      up(&mutex);       // exit critical section
9      up(&empty);       // increment empty count
10     consume_item(item);
11   }
12 }
```


N slots
mutex = 1
empty = 0
filled = N
Producer
Lines 4-11 for N times
Consumer
1 void producer() {
2 int item;
3
4 for (;;) {
5 item = produce_item();
7 down(&mutex); // enter critical section
6 down(&empty); // decrement empty count
8 insert(item); // put new item in buffer
9 up(&mutex); // exit critical section
10 up(&filled); // increment filled count
11 }
12 }
1 void consumer() {
2 int item;
3
4 for (;;) {
5 down(&filled); // decrement filled count
6 down(&mutex); // enter critical section
7 item = remove(); // take item from buffer
8 up(&mutex); // exit critical section
9 up(&empty); // increment empty count
10 consume_item(item);
11 }
12 }


N slots
mutex = 0
empty = 0
filled = N
Producer
Lines 4-11 for N times
Line 5
Line 7
Consumer
1 void producer() {
2 int item;
3
4 for (;;) {
5 item = produce_item();
7 down(&mutex); // enter critical section
6 down(&empty); // decrement empty count
8 insert(item); // put new item in buffer
9 up(&mutex); // exit critical section
10 up(&filled); // increment filled count
11 }
12 }
1 void consumer() {
2 int item;
3
4 for (;;) {
5 down(&filled); // decrement filled count
6 down(&mutex); // enter critical section
7 item = remove(); // take item from buffer
8 up(&mutex); // exit critical section
9 up(&empty); // increment empty count
10 consume_item(item);
11 }
12 }


N slots
mutex = 0
empty = 0
filled = N
Producer
Lines 4-11 for N times
Line 5
Line 7
Line 6
Blocked on empty
Consumer
1 void producer() {
2 int item;
3
4 for (;;) {
5 item = produce_item();
7 down(&mutex); // enter critical section
6 down(&empty); // decrement empty count
8 insert(item); // put new item in buffer
9 up(&mutex); // exit critical section
10 up(&filled); // increment filled count
11 }
12 }
1 void consumer() {
2 int item;
3
4 for (;;) {
5 down(&filled); // decrement filled count
6 down(&mutex); // enter critical section
7 item = remove(); // take item from buffer
8 up(&mutex); // exit critical section
9 up(&empty); // increment empty count
10 consume_item(item);
11 }
12 }

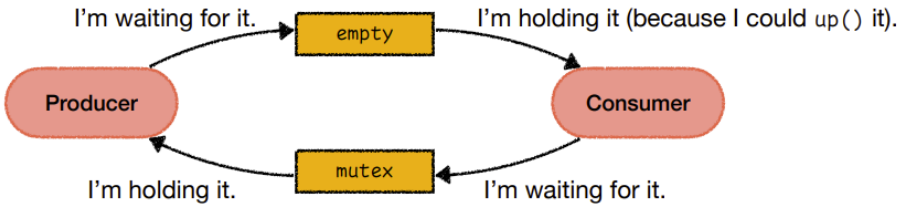




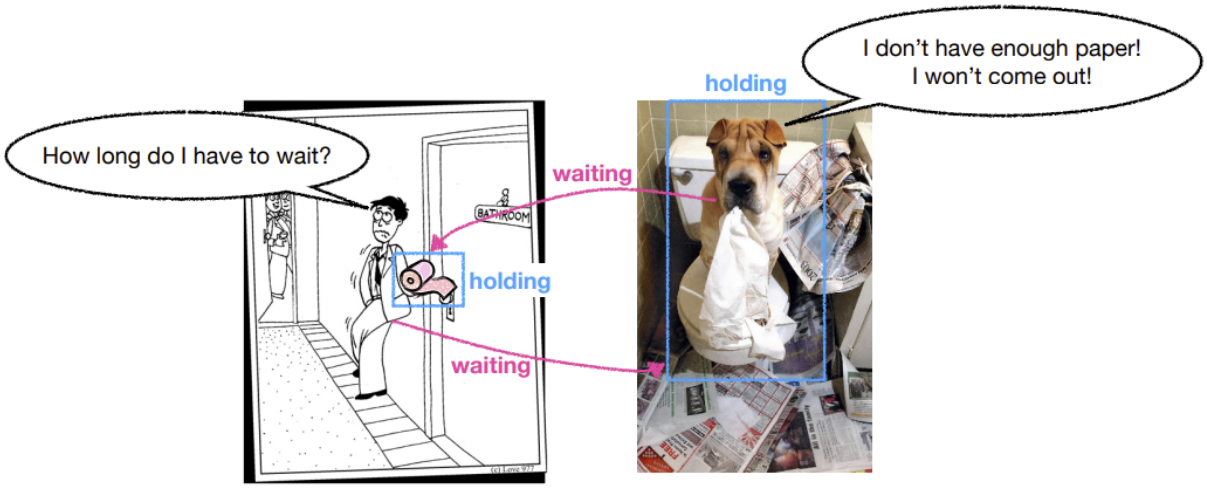
This scenario is called a **deadlock**.

It happens when there is a circular wait…

- The **producer** is waiting for the consumer to `up()` the `empty` semaphore.
- The **consumer** is waiting for the producer to `up()` the `mutex` semaphore.



- deadlock



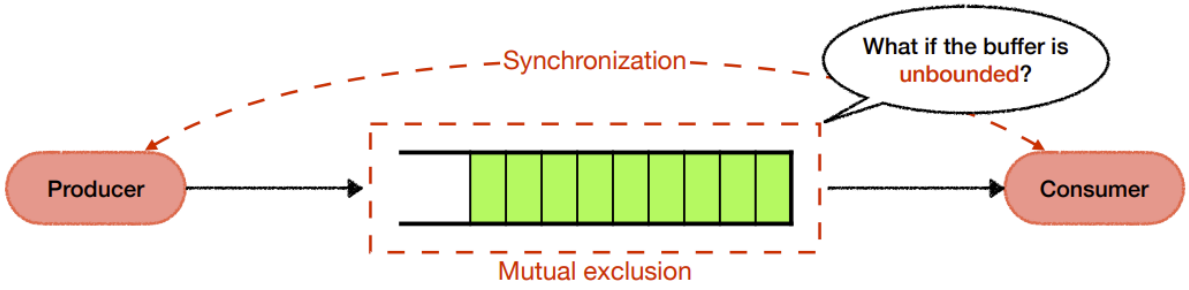
- Summary

Mutual exclusion

A binary semaphore is used as a lock to protect the critical sections.

Synchronization

Two are used to monitor the status of the buffer.counting semaphoresoblem



The dining philosopher problem

The producer-consumer problem

- It models access to a bounded buffer.

The dining philosopher problem

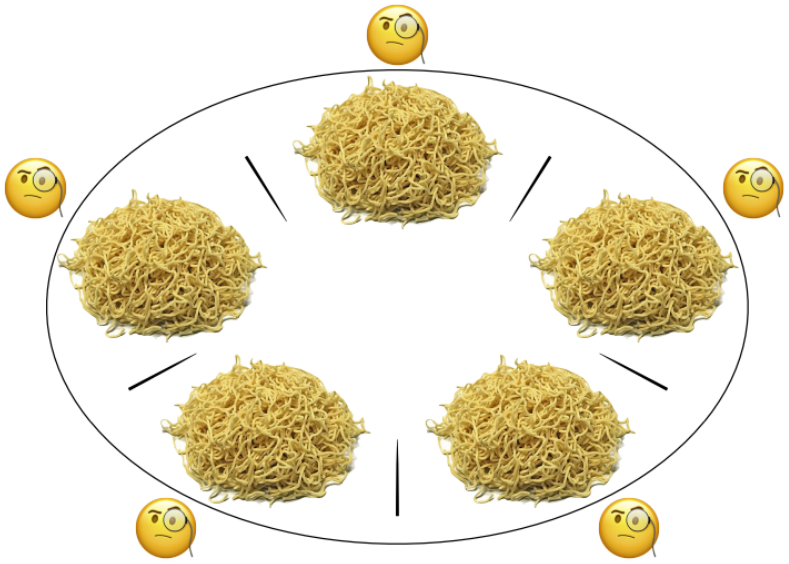
- It models processes competing for exclusive access to a limited number of resources (*e.g.*, I/O devices).

The readers and writers problem

- It models access to a database.

The sleeping barber problem

- It models a queueing situation (♫*please stay on the line…*♪♪).



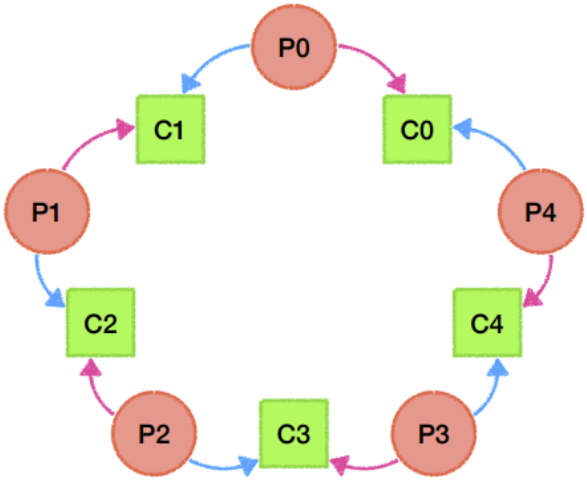
5 philosophers around a table;
5 instant ramen noodles;
5 chopsticks in between.

A philosopher does only two things in their entire life: eating and thinking.

In order to eat, a philosopher needs exactly two chopsticks.

How to design a protocol so that they can enjoy their life?

- Modeling



For each philosopher i,

- $left\ chopstick = i$
- $right\ chopstick = (i+1)\,\%\,N$
- $LEFT\ neighbor = (i+N-1)\,\%\,N$
- $RIGHT\ neighbor = (i+1)\,\%\,N$

States:

- `THINKING`
- `HUNGRY` ← Trying to get chopsticks.
- `EATING` ← Critical section.

- Requirement #1: mutual exclusion

Requirement #1: mutual exclusion

What if there is no mutual exclusion?

— Then, while you're eating, the two philosophers sitting next to you will steal your chopsticks!

- Attempt #1: mutex for each chopstick

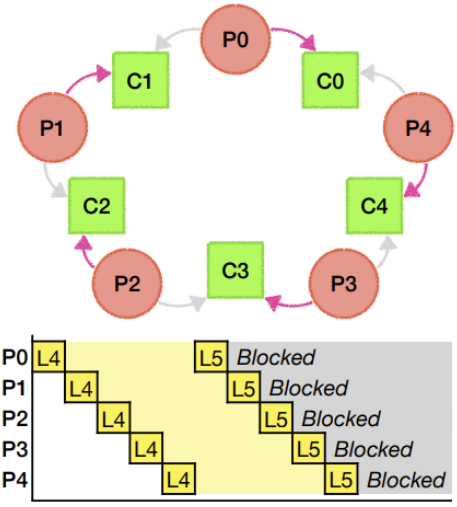
Attempt #1: mutex for each chopstick

```
semaphore chopsticks[N];  // what are the initial values?
```

```
void take_chopstick(int i) {
  down(&chopsticks[i]);
}
```

```
void put_chopstick(int i) {
  up(&chopsticks[i]);
}
```

```
1  void philosopher(int i) {
2    for (;;) {
3      think();
4      take_chopstick(i);
5      take_chopstick((i + 1) % N);
6      eat();                          // critical section
7      put_chopstick((i + 1) % N);
8      put_chopstick(i);
9    }
10 }
```



- Requirement #2: synchronization

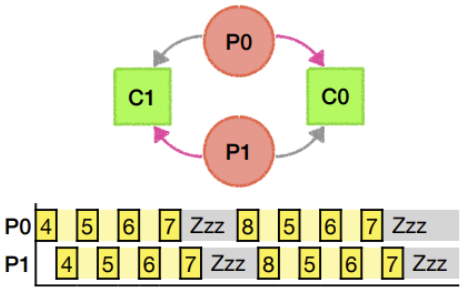
Requirement #2: synchronization

The solution should avoid any potential deadlock or starvation scenarios.

- Attempt #2: mutex for each chopstick with backoff

Attempt #2: mutex for each chopstick with backoff

```
1  void philosopher(int i) {
2    for (;;) {
3      think();
4      take_chopstick(i);
5      while (try_to_take_chopstick((i + 1) % N) == FAILED) {
             /* it either succeeds and takes the chopstick,
                or fails and does not take the chopstick. */
6        put_chopstick(i);
7        sleep(1);              // wait for 1s and try again
8        take_chopstick(i);
9      }
10     eat();                   // critical section
11     put_chopstick((i + 1) % N);
12     put_chopstick(i);
13   }
14 }
```



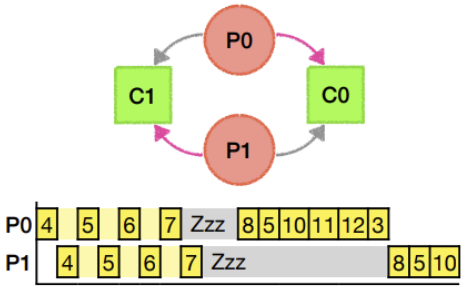
This scenario is called a **livelock**.

No process is blocked, but they don't make any progress.

It's a kind of starvation, literally.

- Attempt #3: mutex for each chopstick with random backoff

```
1  void philosopher(int i) {
2    for (;;) {
3      think();
4      take_chopstick(i);
5      while (try_to_take_chopstick((i + 1) % N) == FAILED) {
               /* it either succeeds and takes the chopstick,
                  or fails and does not take the chopstick. */
6        put_chopstick(i);
7        sleep(random());    // wait a random time
8        take_chopstick(i);
9      }
10     eat();                // critical section
11     put_chopstick((i + 1) % N);
12     put_chopstick(i);
13   }
14 }
```



This is good in most applications.

In fact, many network protocols are designed this way.

Is there a non-randomized solution?

- Attempt #4: global mutex

```
semaphore mutex = 1;
semaphore chopsticks[N];  // what are the initial values?
```

```
1  void philosopher(int i) {
2    for (;;) {
3      think();
4      down(&mutex);
5      take_chopstick(i);
6      take_chopstick((i + 1) % N);
7      eat();                        // critical section
8      put_chopstick((i + 1) % N);
9      put_chopstick(i);
10     up(&mutex);
11   }
12 }
```

Is it correct?

- Any deadlock? — No.
- Any starvation? — No.

This solution is theoretically correct.

Is it a good solution?

With the global mutex, there is no parallelization.

Only one philosopher can be eating at any time.

The performance is very bad.

- Attempt #5: one semaphore per philosopher

Before we propose the final solution, let's see what issues we've been facing…

- Modeling each chopstick as a semaphore is intuitive, but it's not working.
- We are afraid to `down()` a chopstick, as that may lead to a deadlock.

However, our real issue is that philosophers need to eat, not chopsticks.

So, instead of worrying about chopsticks, we should guarantee…

- When a philosopher is eating, their left and right neighbors cannot eat.

Attempt #5: one semaphore per philosopher

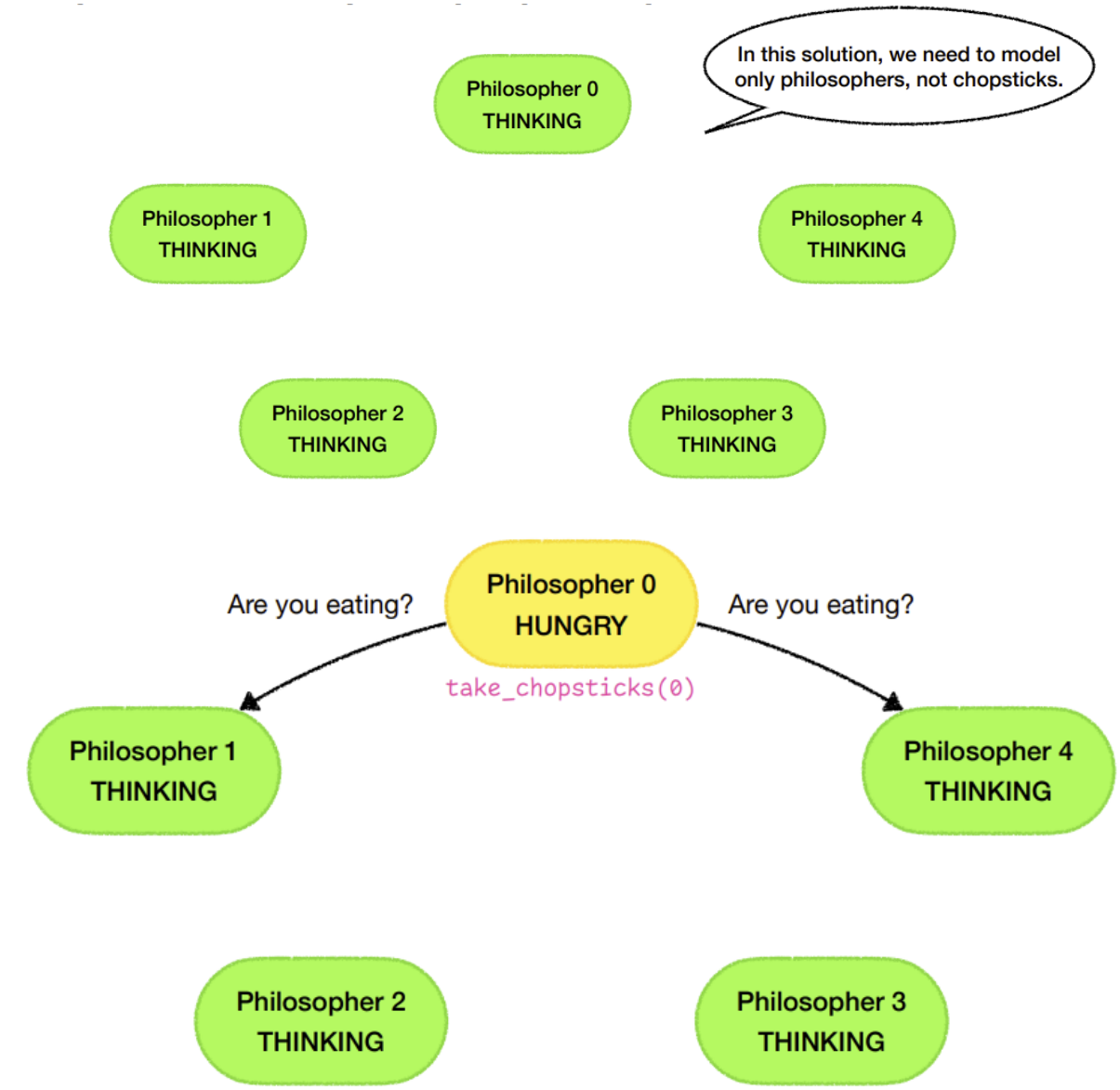
```
int state[N];          // THINKING, HUNGRY, or EATING
semaphore mutex = 1;
semaphore sem[N];      // what are the initial values?
```

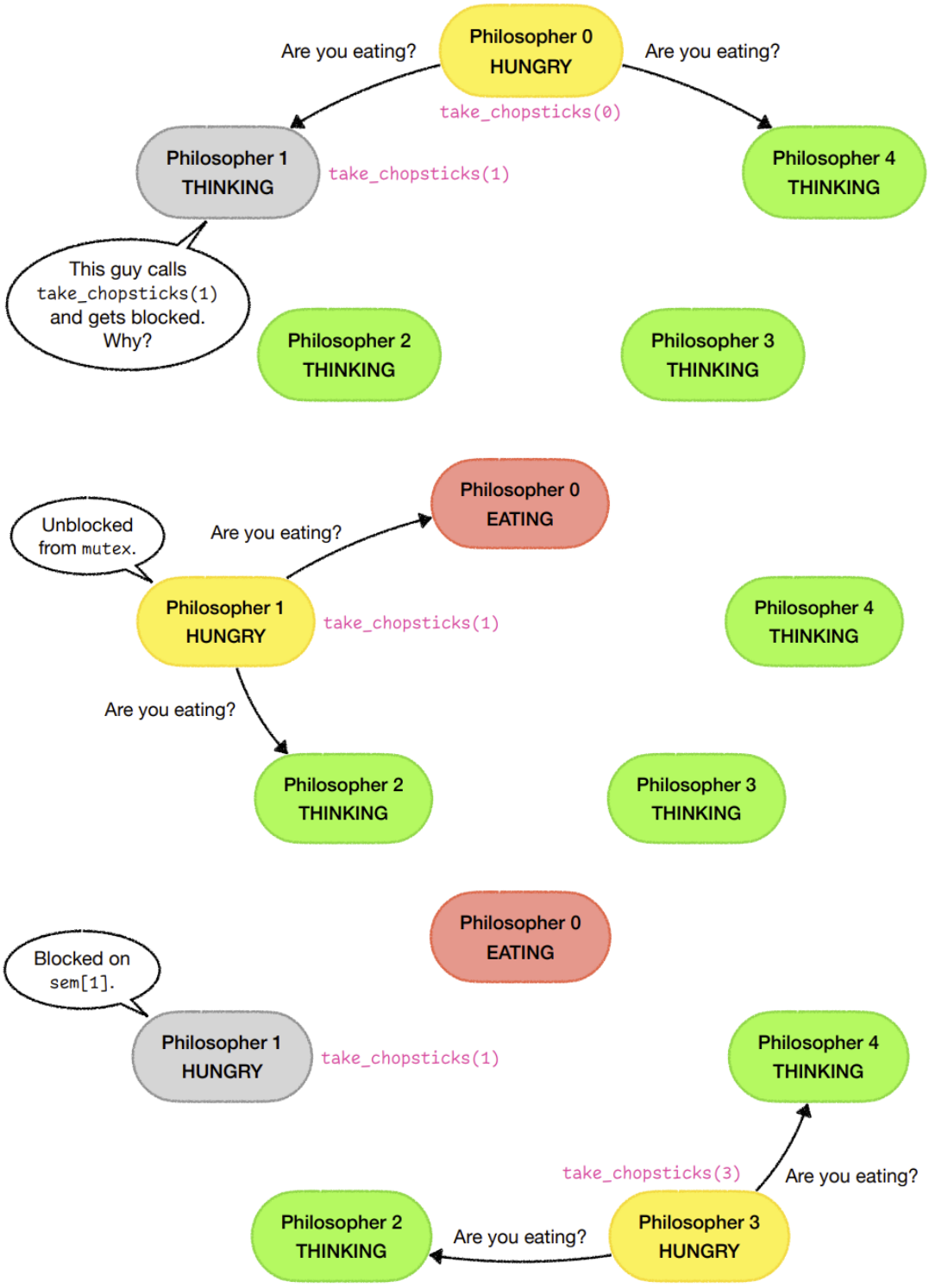
```
void take_chopsticks(int i) {
  down(&mutex);
  state[i] = HUNGRY;
  test(i);         // try to take both chopsticks
  up(&mutex);
  down(&sem[i]);  // block if cannot take chopsticks
}
```

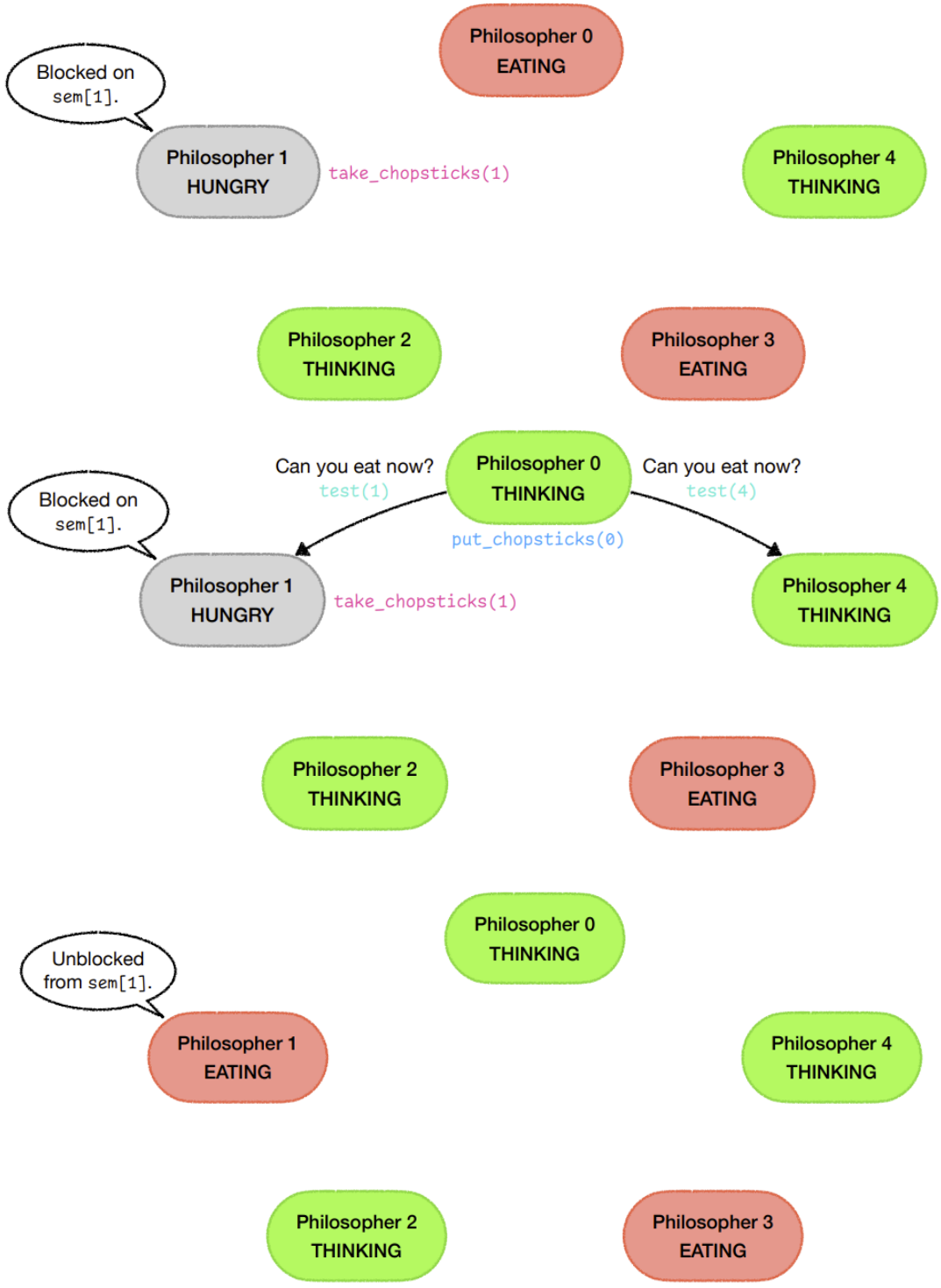
```
void put_chopsticks(int i) {
  down(&mutex);
  state[i] = THINKING;
  test(LEFT);   // see if left neighbor can now eat
  test(RIGHT);  // see if right neighbor can now eat
  up(&mutex);
}
```

```
void test(i) {
  if (state[i] == HUNGRY &&
      state[LEFT] ≠ EATING &&
      state[RIGHT] ≠ EATING) {
    state[i] = EATING;
    up(&sem[i]);  // take both chopsticks
  }
}
```

```
void philosopher(int i) {
  for (;;) {
    think();
    take_chopsticks(i);  // section entry
      // block until I take both chopsticks
    eat();               // critical section
    put_chopsticks(i);   // section exit
  }
}
```




Are you eating?
Philosopher 0
HUNGRY
Are you eating?
take_chopsticks(0)
Philosopher 1
THINKING
take_chopsticks(1)
Philosopher 4
THINKING
This guy calls take_chopsticks(1) and gets blocked. Why?
Philosopher 2
THINKING
Philosopher 3
THINKING
Unblocked from mutex.
Are you eating?
Philosopher 0
EATING
Philosopher 1
HUNGRY
take_chopsticks(1)
Philosopher 4
THINKING
Are you eating?
Philosopher 2
THINKING
Philosopher 3
THINKING
Philosopher 0
EATING
Blocked on sem[1].
Philosopher 1
HUNGRY
take_chopsticks(1)
Philosopher 4
THINKING
take_chopsticks(3)
Are you eating?
Philosopher 2
THINKING
Are you eating?
Philosopher 3
HUNGRY


Philosopher 0
EATING
Blocked on
sem[1].
Philosopher 1
HUNGRY
take_chopsticks(1)
Philosopher 4
THINKING
Philosopher 2
THINKING
Philosopher 3
EATING
Can you eat now?
test(1)
Philosopher 0
THINKING
Can you eat now?
test(4)
put_chopsticks(0)
Blocked on
sem[1].
Philosopher 1
HUNGRY
take_chopsticks(1)
Philosopher 4
THINKING
Philosopher 2
THINKING
Philosopher 3
EATING
Philosopher 0
THINKING
Unblocked
from sem[1].
Philosopher 1
EATING
Philosopher 4
THINKING
Philosopher 2
THINKING
Philosopher 3
EATING

- Summary

This problem is once again about mutual exclusion and synchronization.

Multiple processes compete for multiple resources.

Each process needs two distinct resources to enter the critical section.

The random backoff approach is adopted in many network protocols.

The final solution is deadlock-free, starvation-free, and allows the maximum parallelism for any number of processes.