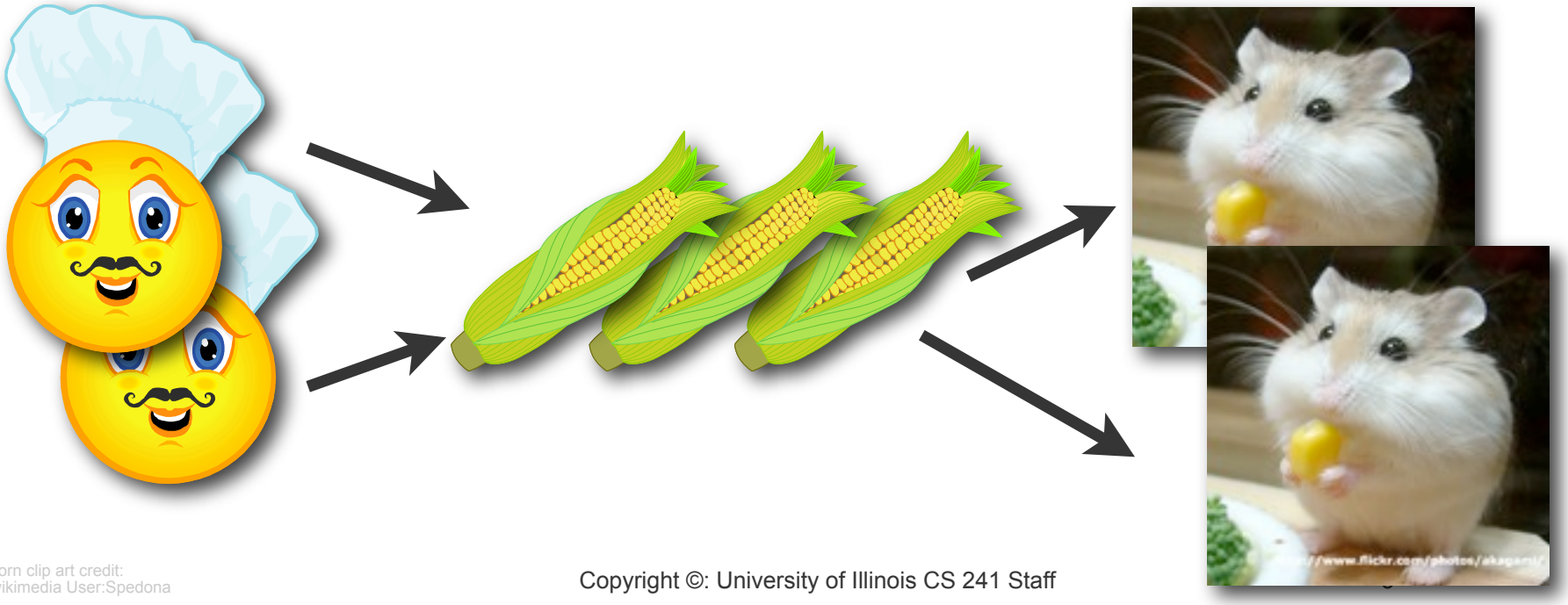
Classical Synchronization Problems

This lecture

- Goals:
  - Introduce classical synchronization problems
- Topics
  - Producer-Consumer Problem
  - Reader-Writer Problem
  - Dining Philosophers Problem
  - Sleeping Barber's Problem

1. Producer-Consumer

- Chefs cook items and put them on a conveyer belt
- Customers pick items off the belt



corn clip art credit:
wikimedia User:Spedona

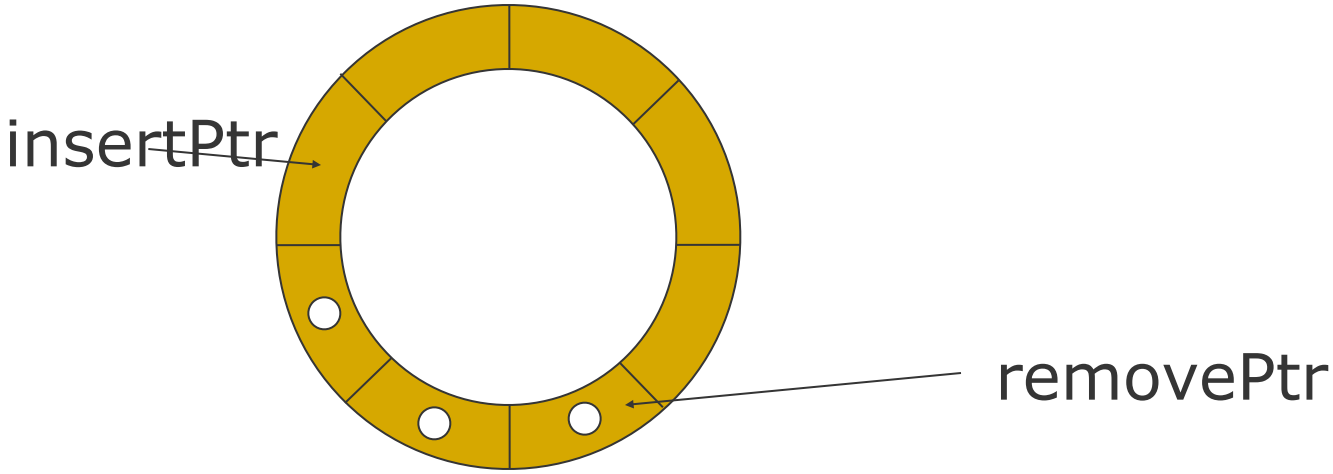
Producer-Consumer Problem

Chef = Producer
Customer = Consumer

- Producers insert items
- Consumers remove items
- Shared bounded buffer
  - Efficient implementation: circular buffer with an insert and a removal pointer.

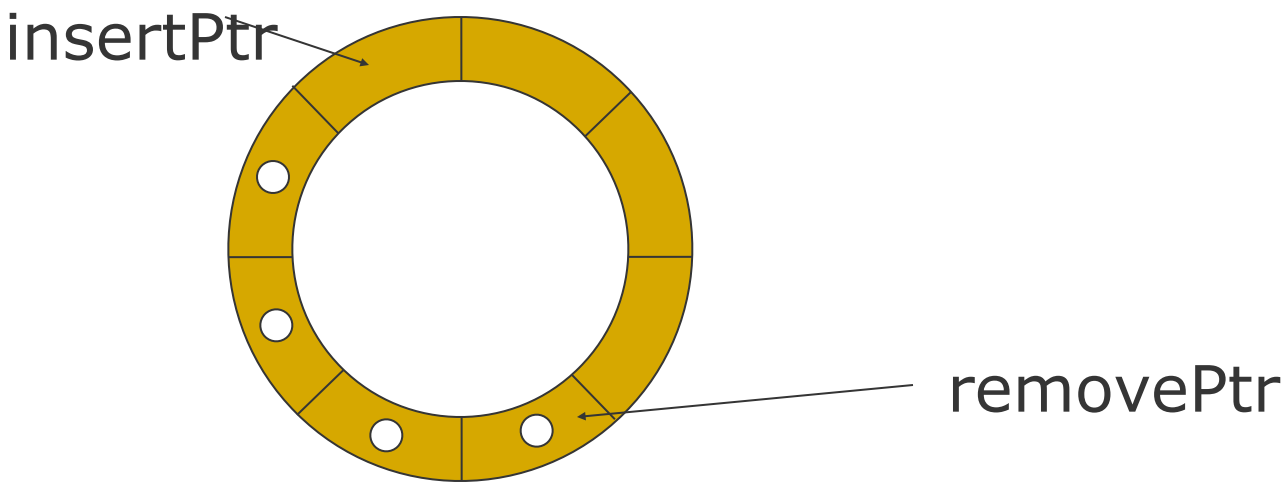
Producer-Consumer

Chef = Producer
Customer = Consumer



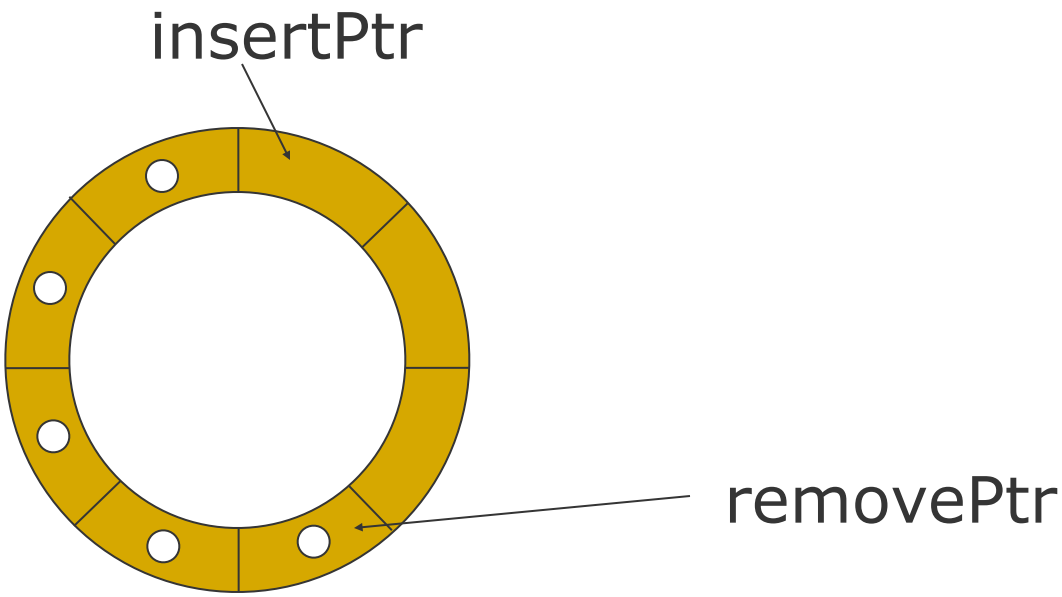
Producer-Consumer

Chef = Producer
Customer = Consumer



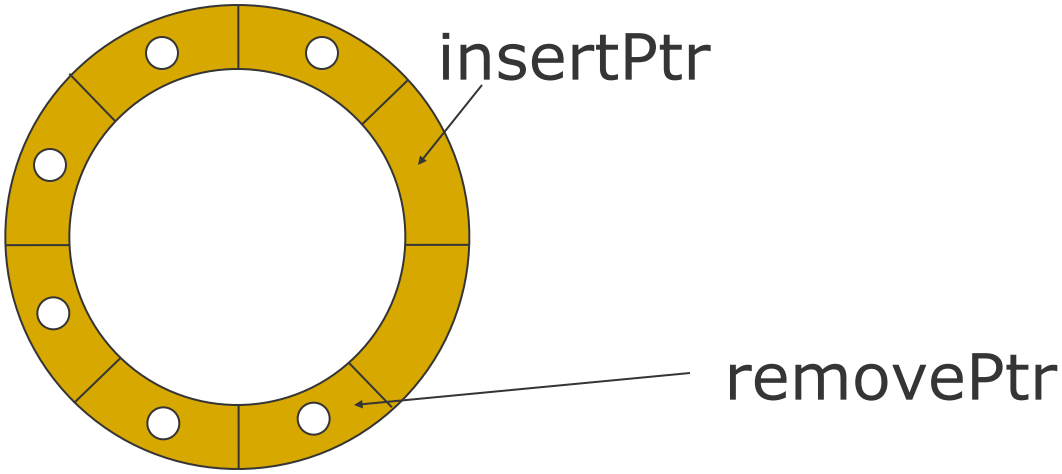
Producer-Consumer

Chef = Producer
Customer = Consumer



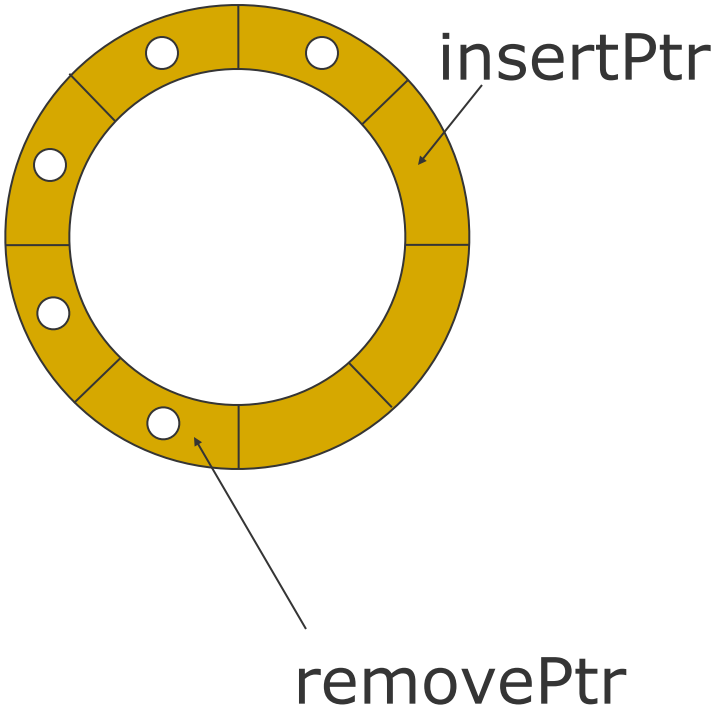
Producer-Consumer

Chef = Producer
Customer = Consumer



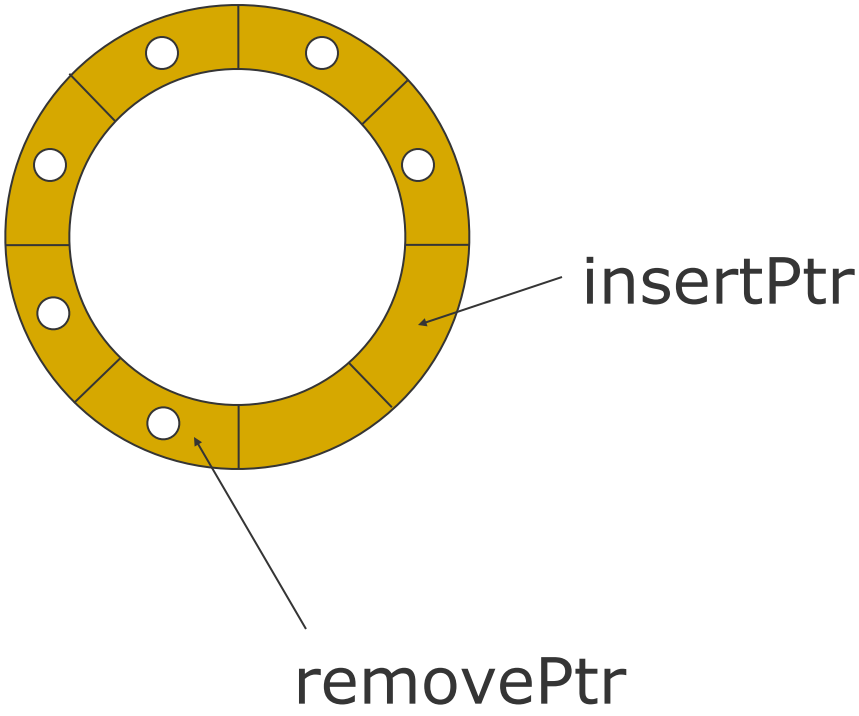
Producer-Consumer

Chef = Producer
Customer = Consumer



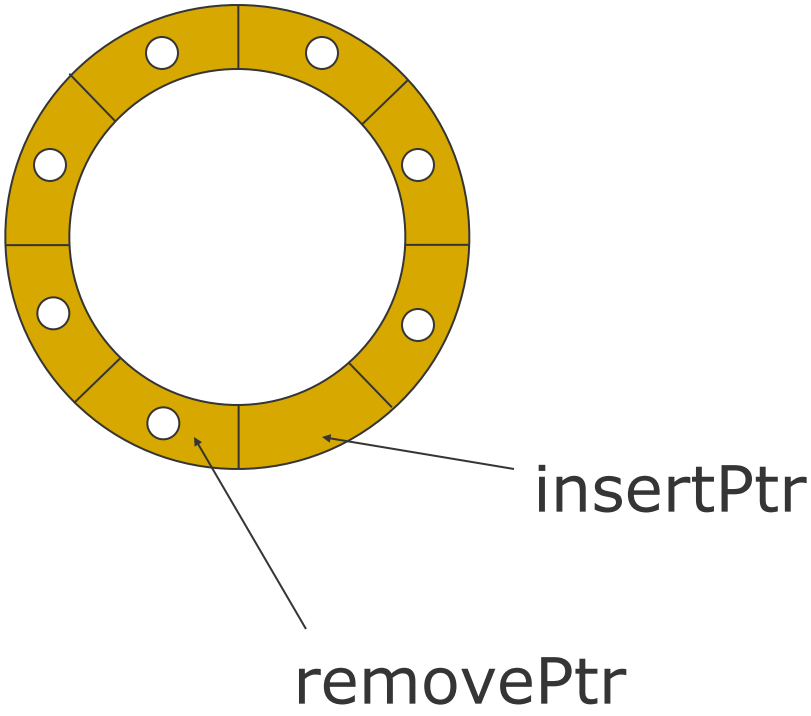
Producer-Consumer

Chef = Producer
Customer = Consumer



Producer-Consumer

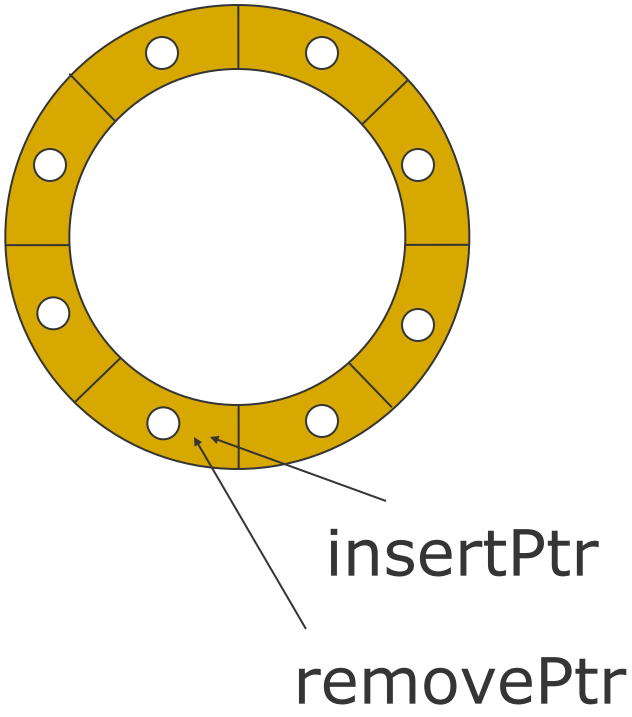
Chef = Producer
Customer = Consumer



Producer-Consumer

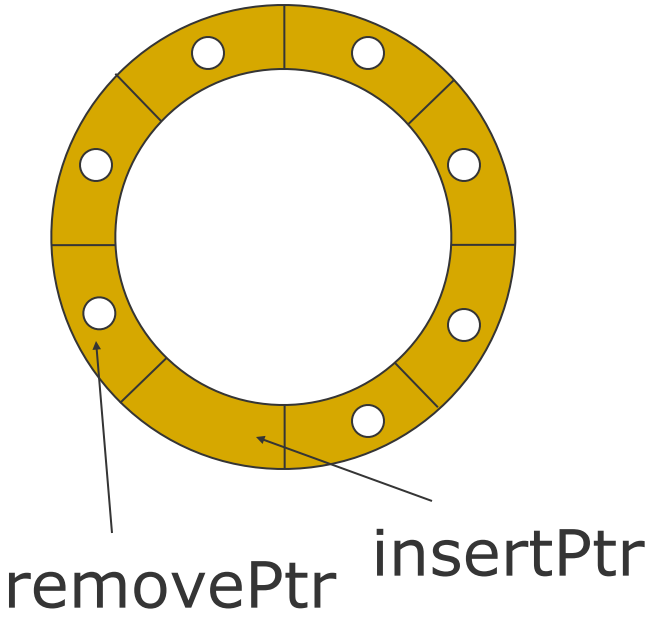
Chef = Producer
Customer = Consumer

BUFFER FULL: Producer must be blocked!



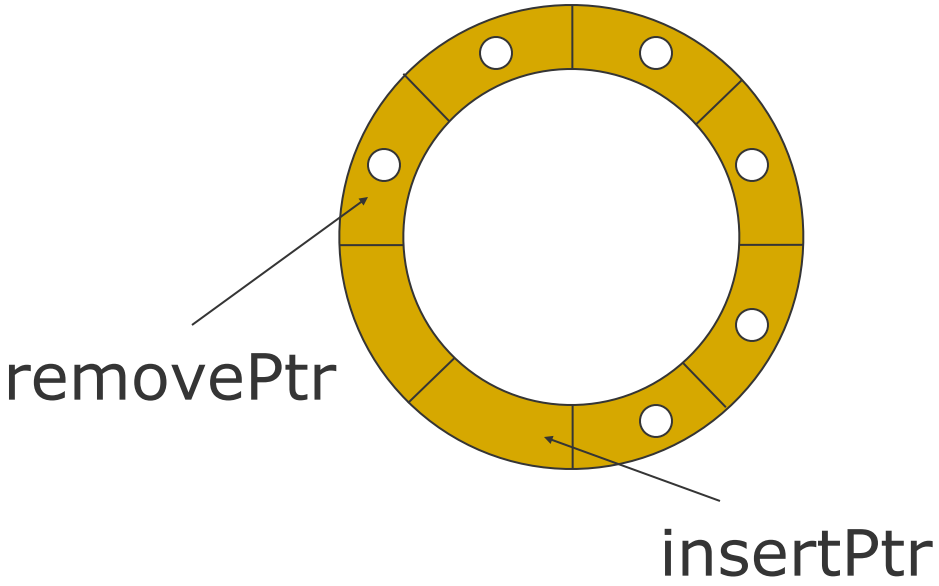
Producer-Consumer

Chef = Producer
Customer = Consumer



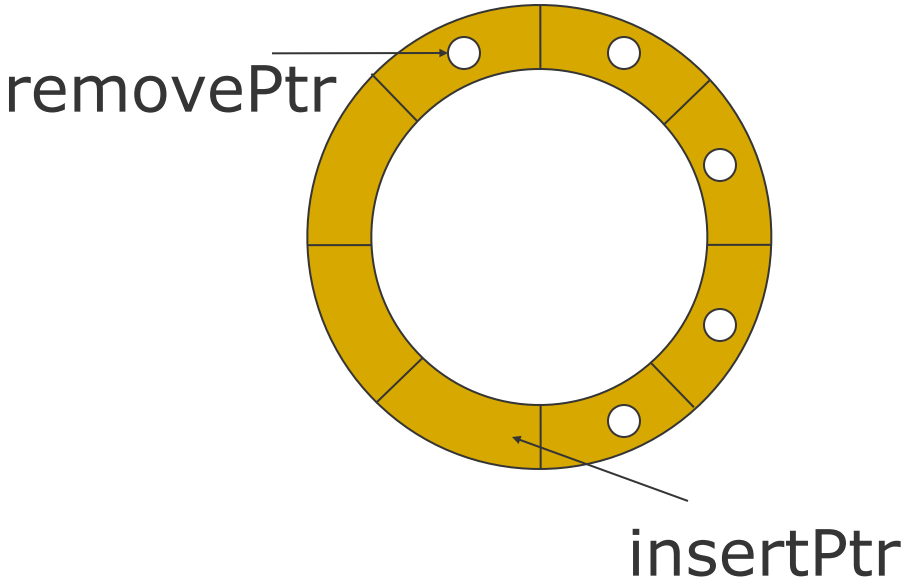
Producer-Consumer

Chef = Producer
Customer = Consumer



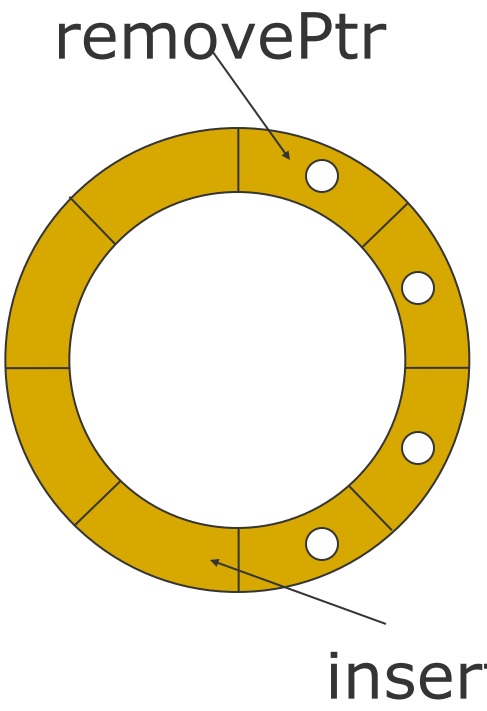
Producer-Consumer

Chef = Producer
Customer = Consumer



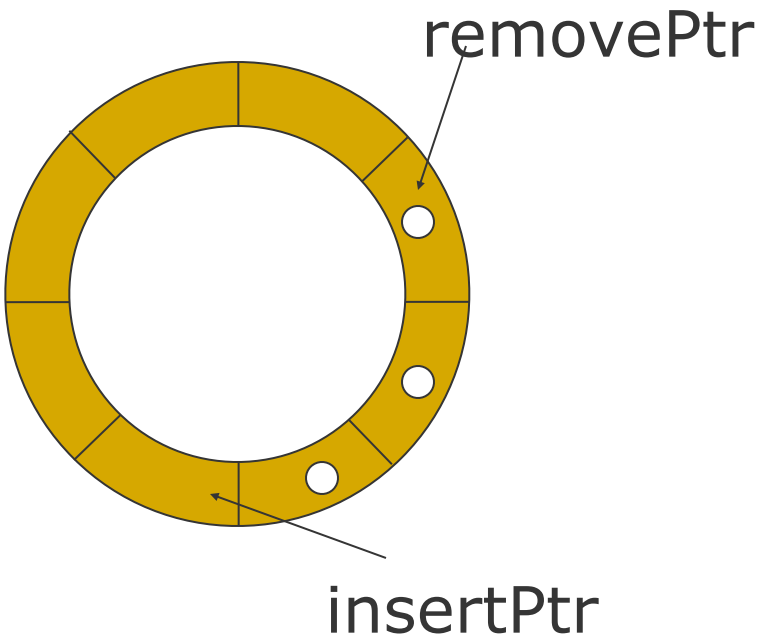
Producer-Consumer

Chef = Producer
Customer = Consumer



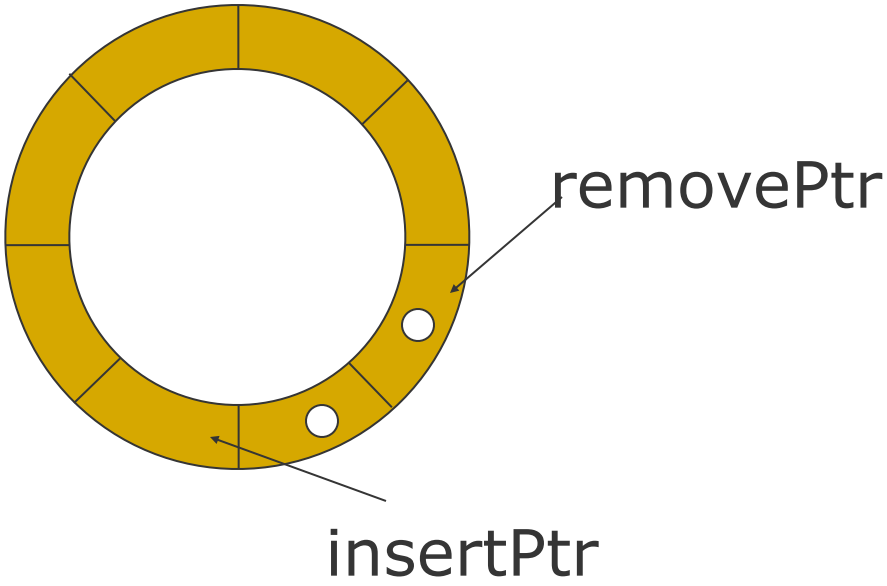
Producer-Consumer

Chef = Producer
Customer = Consumer



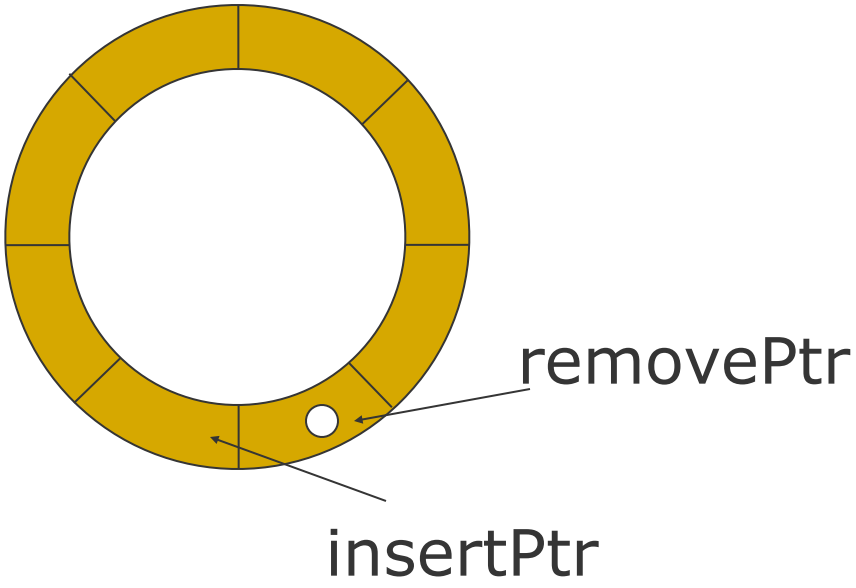
Producer-Consumer

Chef = Producer
Customer = Consumer



Producer-Consumer

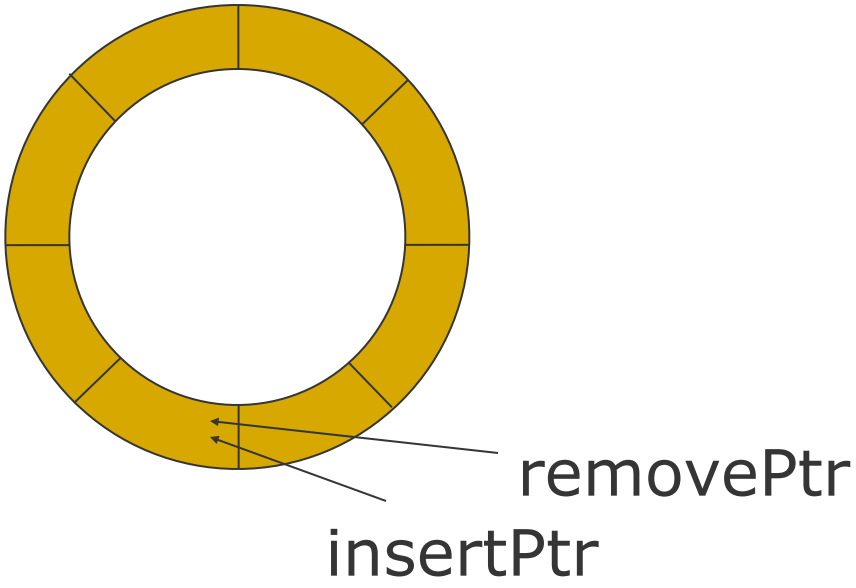
Chef = Producer
Customer = Consumer



Producer-Consumer

Chef = Producer
Customer = Consumer

BUFFER EMPTY: Consumer must be blocked!



Producer-Consumer Problem

- Producer inserts items. Updates insertion pointer.
- Consumer executes destructive reads on the buffer. Updates removal pointer.
- Both update information about how full/ empty the buffer is.
- Solution should allow multiple producers and consumers

Challenges

- Prevent buffer overflow
- Prevent buffer underflow
- Mutual exclusion when modifying the buffer data structure

Solution

- Prevent overflow: block producer when full! Counting semaphore to count #free slots
  - 0 → block producer
- Prevent underflow: block consumer when empty! Counting semaphore to count #items in buffer
  - 0 → block consumer
- Mutex to protect accesses to shared buffer & pointers.

Solution

- Prevent overflow: block producer when full! Counting semaphore to count #free slots
  - 0 → block producer
- Prevent underflow: block consumer when empty! Counting semaphore to count #items in buffer
  - 0 → block consumer
- Mutex to protect accesses to shared buffer & pointers.

Assembling the solution

- sem_wait(slots), sem_signal(slots)
- sem_wait(items), sem_signal(items)
- mutex_lock(m), mutex_unlock(m)

- insertptr = (insertptr+1) % N
- removalptr = (removalptr+1) % N

- Initialize semaphore slots to size of buffer
- Initialize semaphore items to zero.

Pseudocode getItem()

- **For consumer**
- *Error checking/EINTR handling not shown*

```
sem_wait(items);
mutex_lock(mutex);
result = buffer[ removePtr ];
removePtr = (removePtr +1) % N;
mutex_unlock(mutex);
sem_signal(slots);
```


Pseudocode putItem(*data*)

- **For producer**
- *Error checking/EINTR handling not shown*

```
sem_wait(slots);
mutex_lock(mutex);
buffer[ insertPtr ] = data;
insertPtr = (insertPtr + 1) % N;
mutex_unlock(mutex);
sem_signal(items);
```


II. Reader-Writer Problem

- A reader: read data
- A writer: write data
- Rules:
  - Multiple readers may read the data simultaneously
  - Only one writer can write the data at any time
  - A reader and a writer cannot access data simultaneously
- Locking table: whether any two can be in the critical section simultaneously

	Reader	Writer
Reader	OK	No
Writer	No	No

Reader-writer solution

```
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

void reader(void)
{
  while (TRUE) {
        down(&mutex);
        rc = rc + 1;
        if (rc == 1) down(&db);
        up(&mutex);
        read_data_base();
        down(&mutex);
        rc = rc - 1;
        if (rc == 0) up(&db);
        up(&mutex);
        use_data_read();
  }
}
```

```
/* controls access to 'rc' */
/* controls access to the data base */
/* # of processes reading or wanting to */

void writer(void)
{
  while (TRUE) {
        think_up_data();
        down(&db);
        write_data_base();
        up(&db);
  }
}
```

This solution can starve the writer!

Note:
`down` means `semWait`
`up` means `semSignal`

[From Tanenbaum, Modern Operating Systems]

Better R-W solution idea

- Idea: serve requests in order
  - once a writer requests access, any entering readers have to block until the writer is done
- Advantage?
- Disadvantage?

Summary

- Classic synchronization problems
  - Producer-Consumer Problem
  - Reader-Writer Problem
- Saved for next time:
  - Sleeping Barber's Problem
  - Dining Philosophers Problem