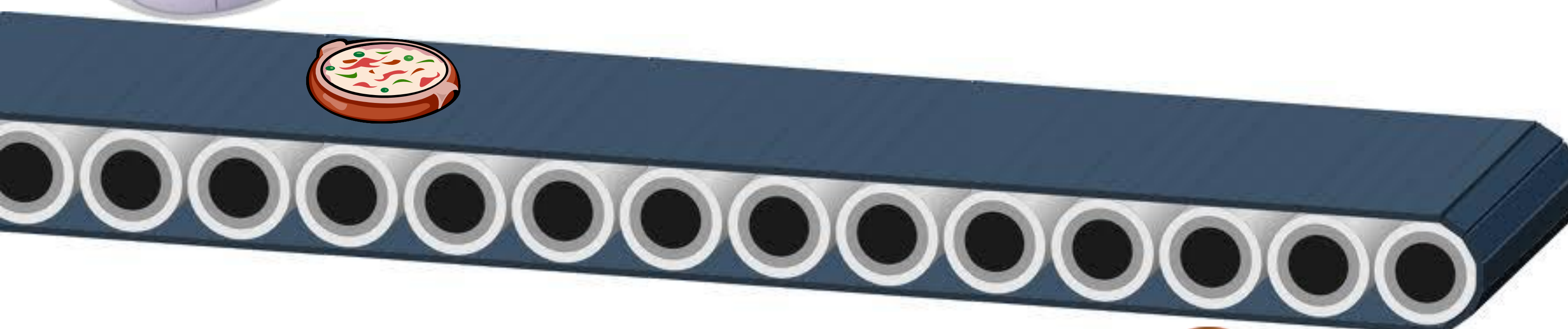
Classical Synchronization Problems

This lecture

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

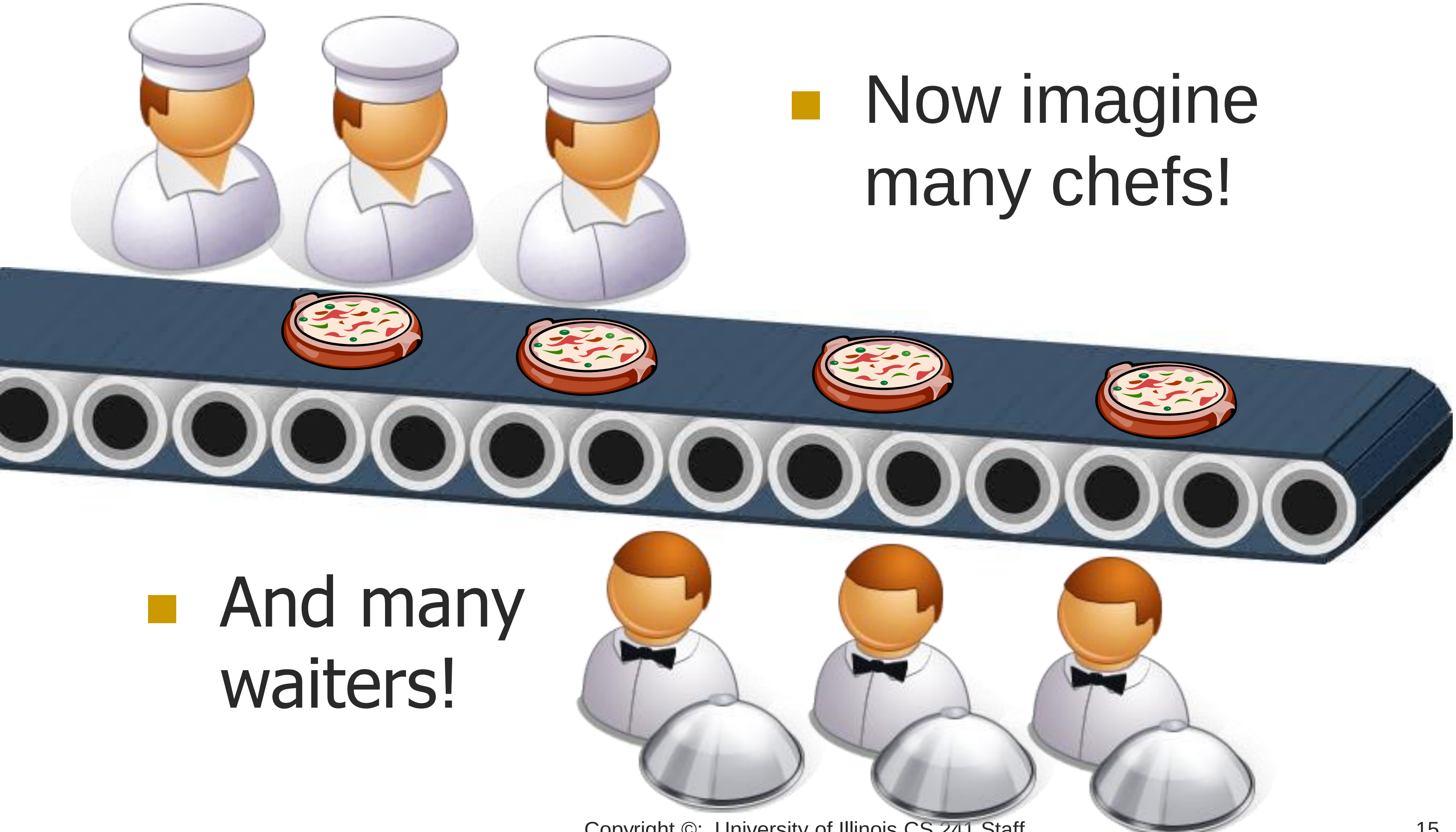
Producer-consumer problem

- Chefs cook items and put them on a conveyer belt



- Waiters pick items off the belt

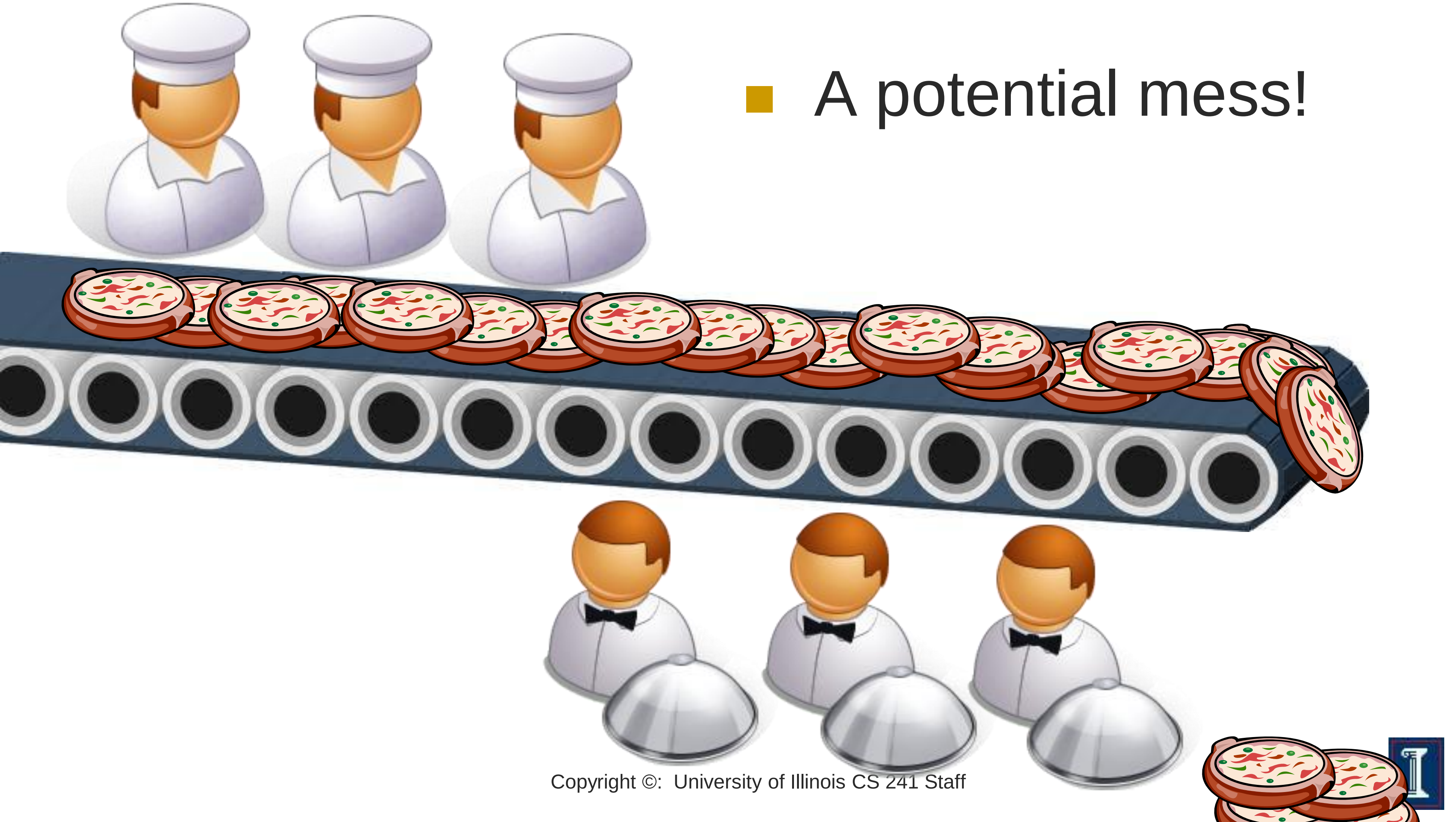
Producer-consumer problem



- Now imagine many chefs!
- And many waiters!

Producer-consumer problem

- A potential mess!



Producer-Consumer Problem

Chef (Producer)

inserts items

Waiter (Consumer)

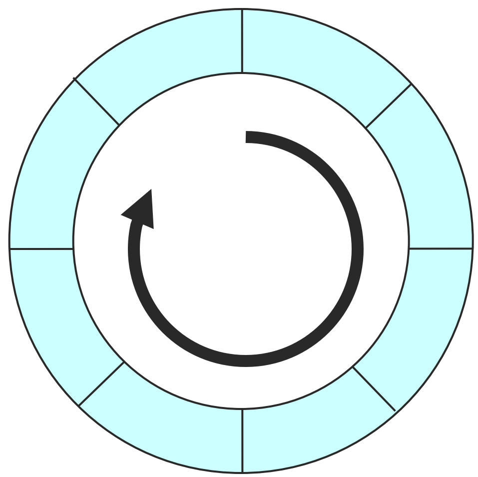
removes items

Shared resource:
bounded buffer

Efficient implementation:
circular fixed-size buffer

Producer-Consumer

Chef = Producer
Waiter = Consumer

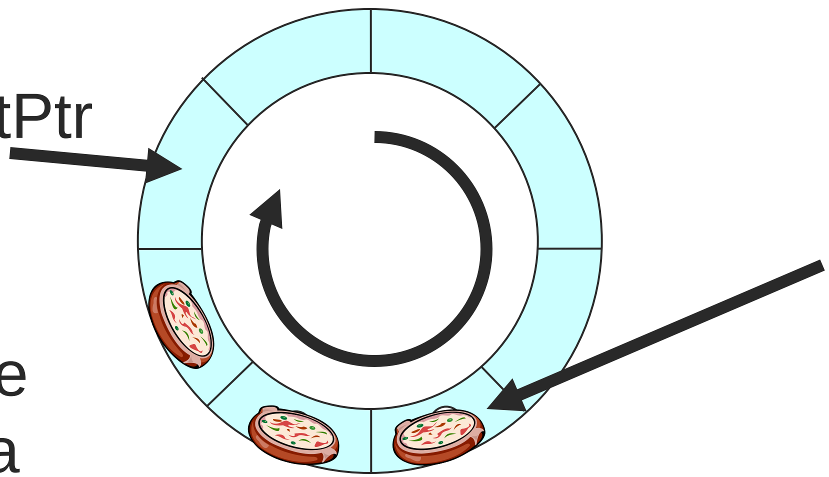


Producer-Consumer

Chef = Producer

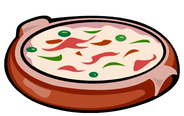
Waiter = Consumer

insertPtr



removePtr

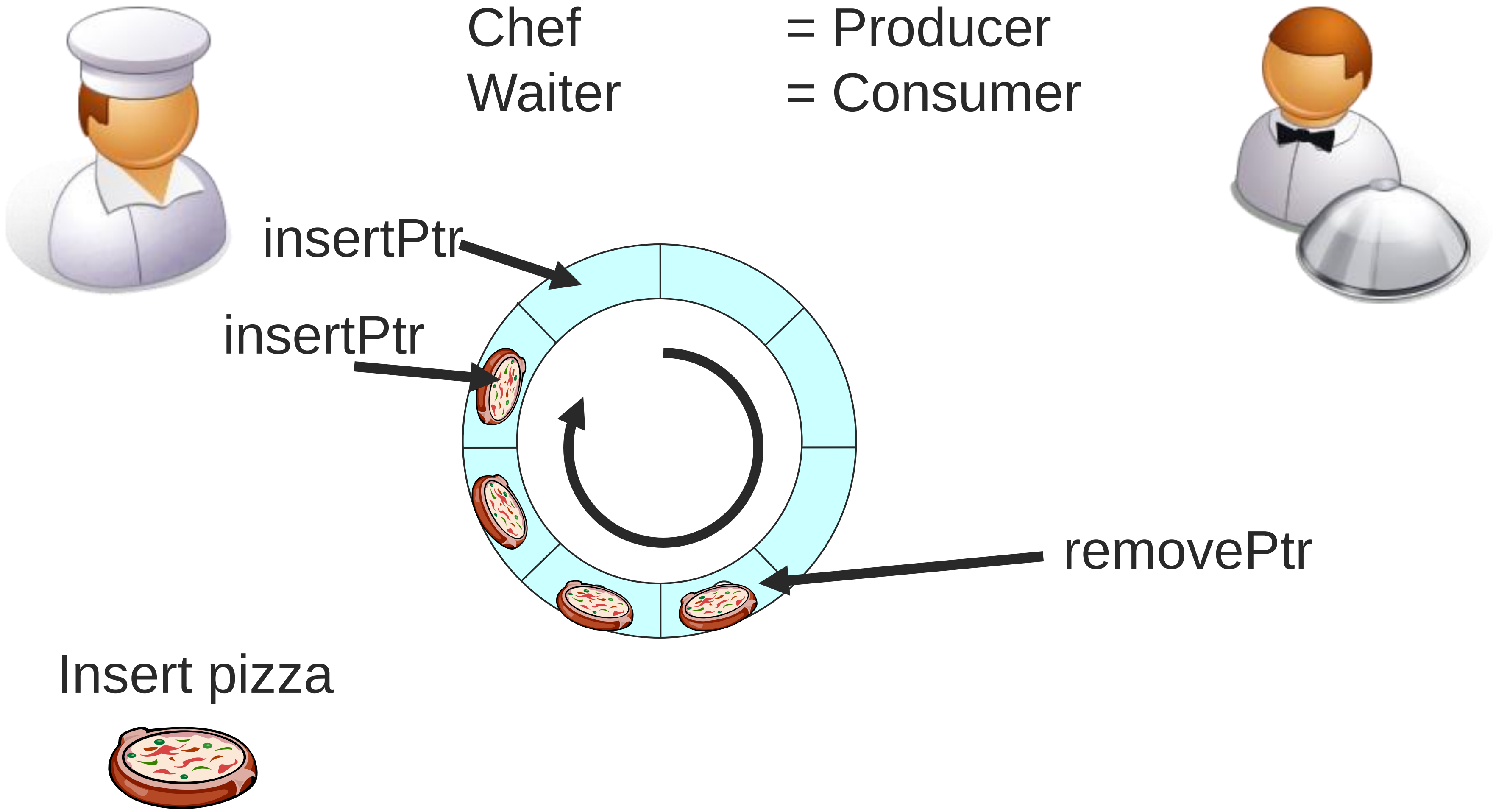
What does the chef do with a new pizza?



Where does the waiter take a pizza from?

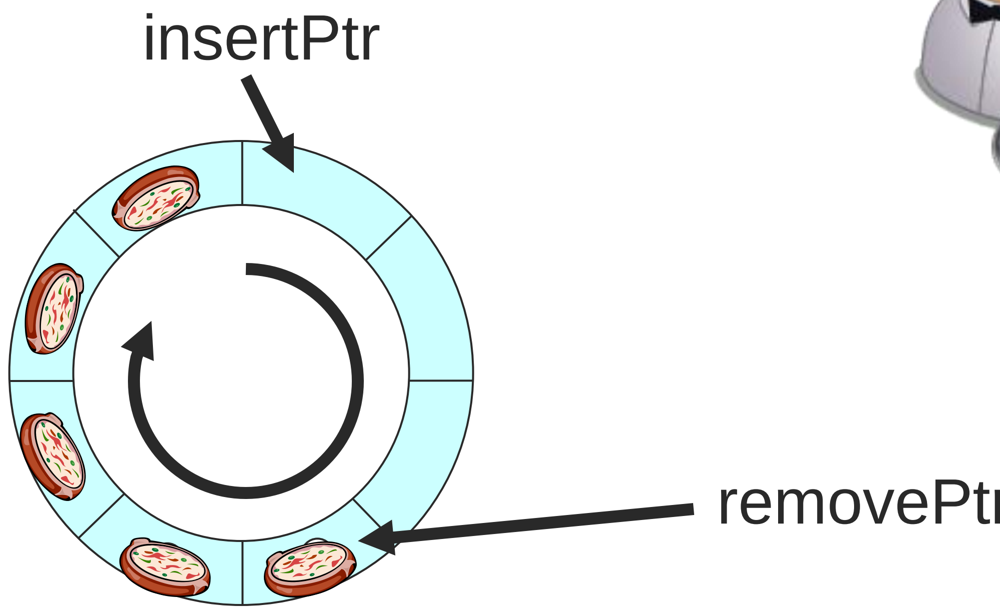
Producer-Consumer

Chef = Producer
Waiter = Consumer

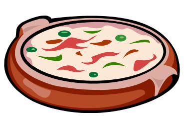


Producer-Consumer

Chef = Producer
Waiter = Consumer

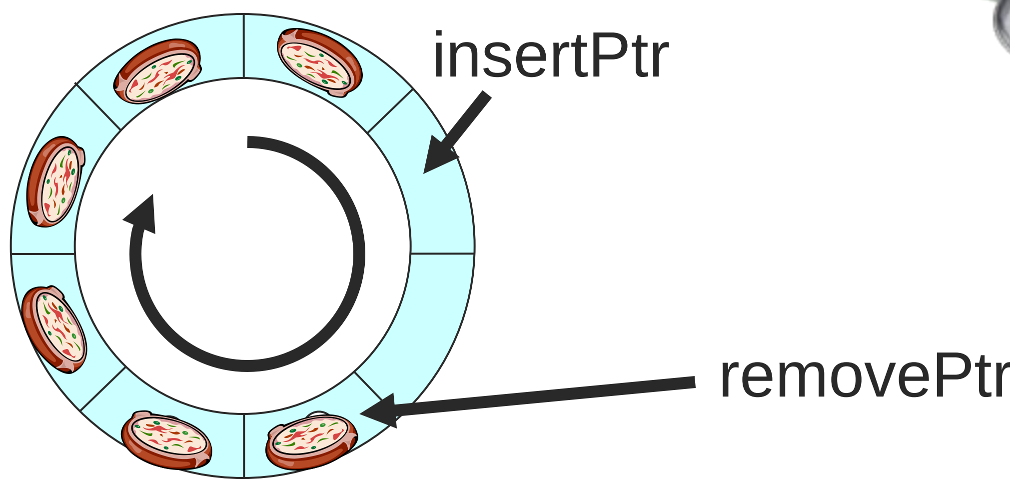


Insert pizza

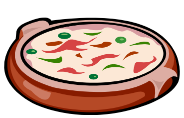


Producer-Consumer

Chef	= Producer
Waiter	= Consumer

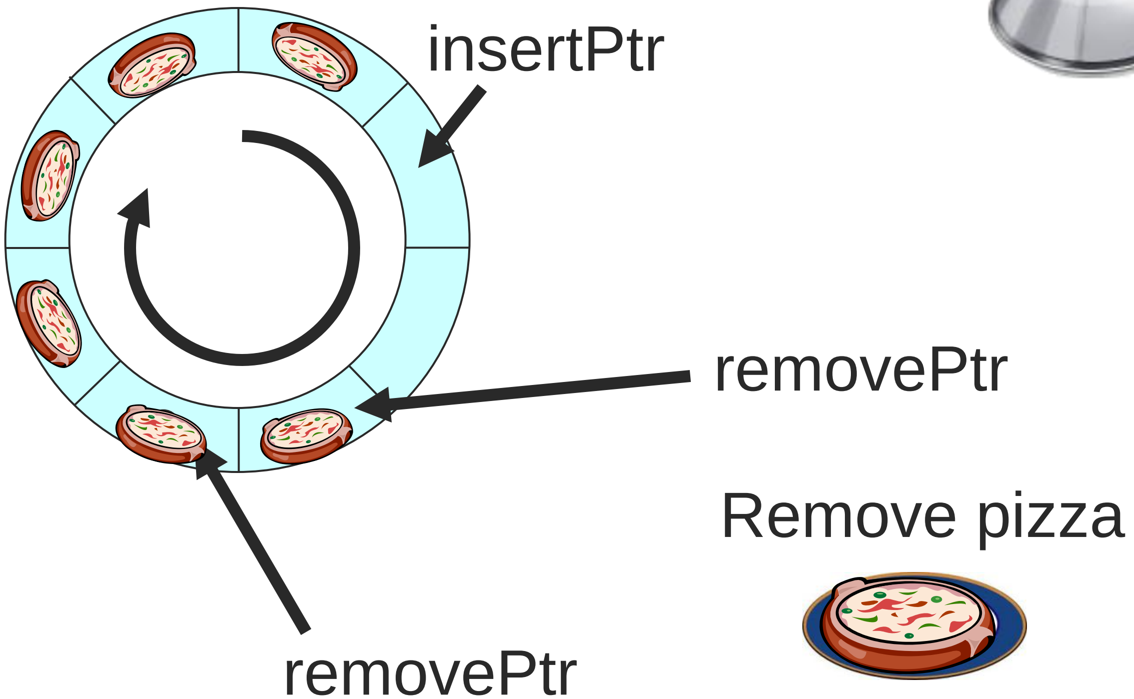


Insert pizza



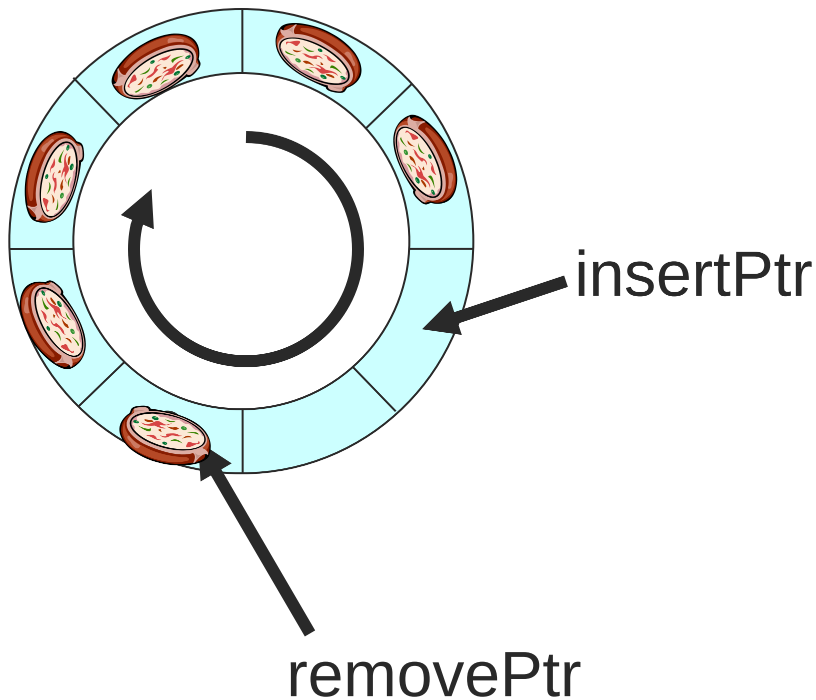
Producer-Consumer

Chef	= Producer
Waiter	= Consumer

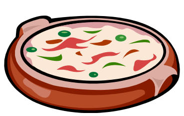


Producer-Consumer

Chef = Producer
Waiter = Consumer

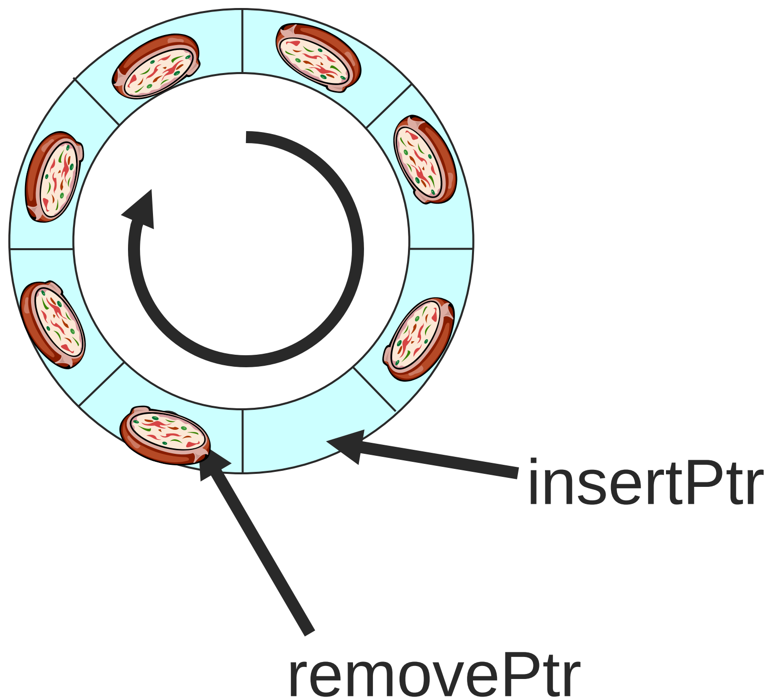


Insert pizza

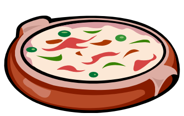


Producer-Consumer

Chef = Producer
Waiter = Consumer



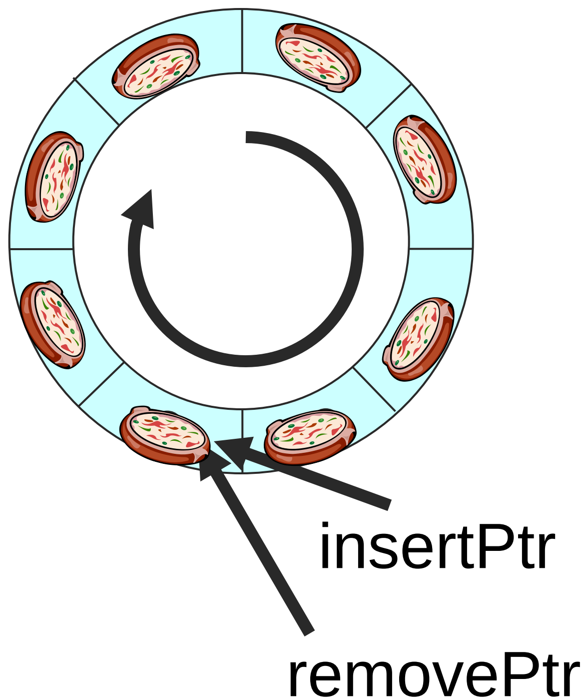
Insert pizza



Producer-Consumer

Chef = Producer
Waiter = Consumer

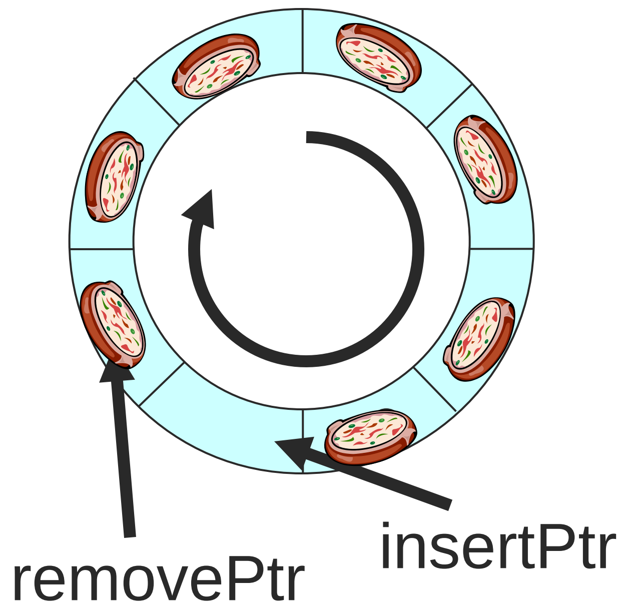
BUFFER FULL: Producer must be blocked!



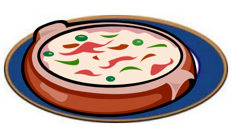
Insert pizza

Producer-Consumer

Chef = Producer
Waiter = Consumer

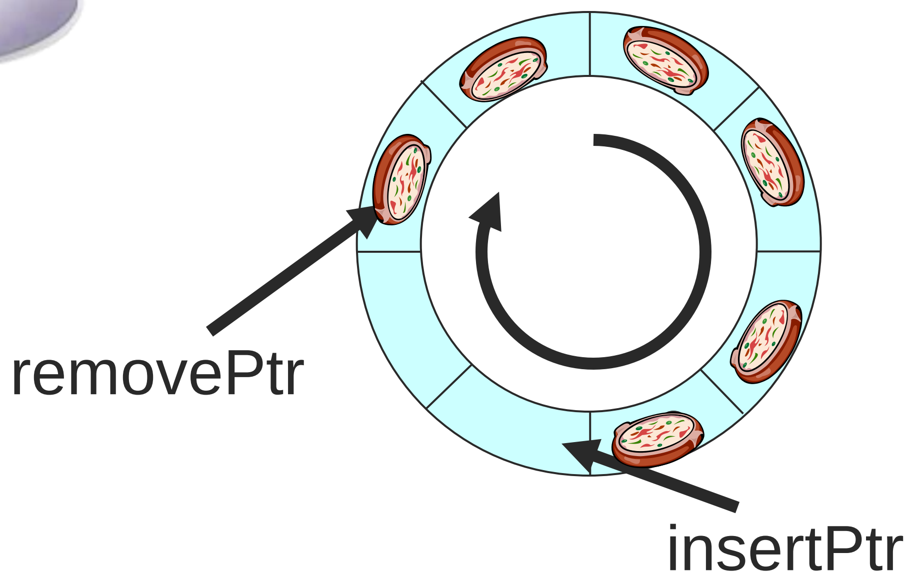


Remove pizza

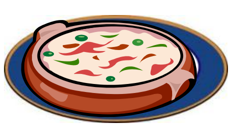


Producer-Consumer

Chef = Producer
Waiter = Consumer

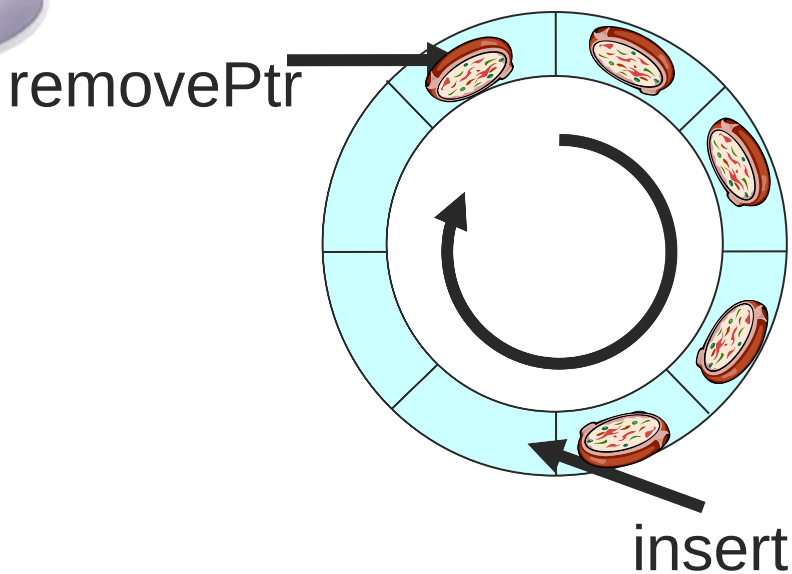


Remove pizza

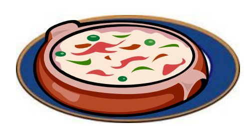


Producer-Consumer

Chef = Producer
Waiter = Consumer

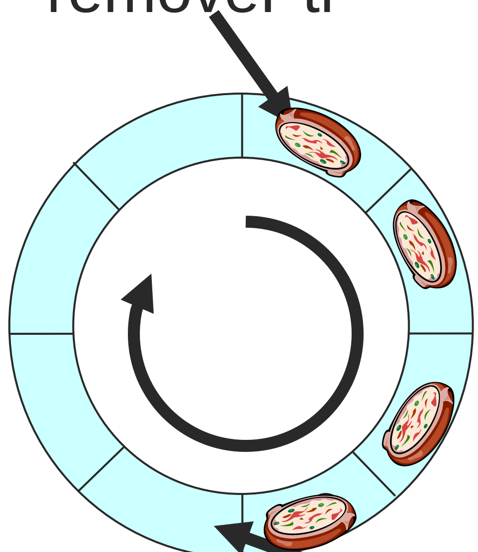


Remove pizza

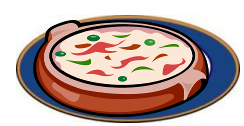


Producer-Consumer

Chef = Producer
Waiter = Consumer

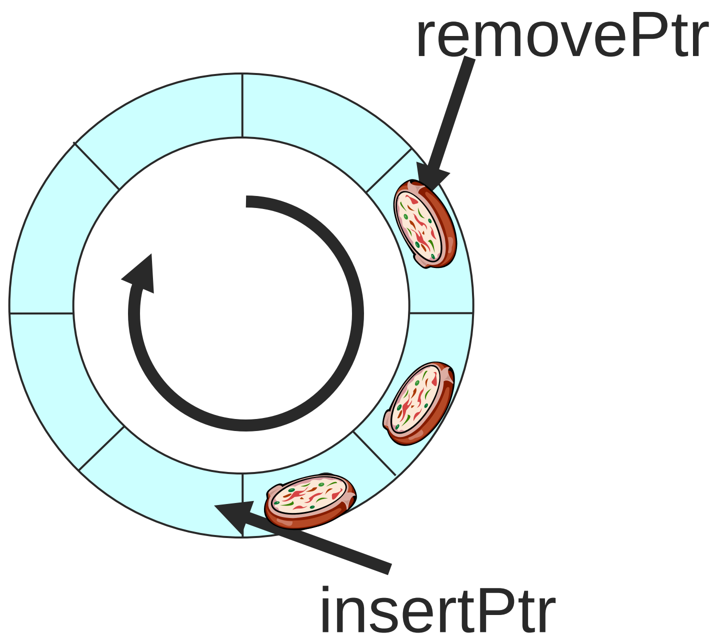


Remove pizza

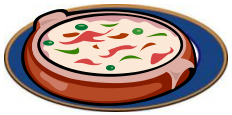


Producer-Consumer

Chef = Producer
Waiter = Consumer

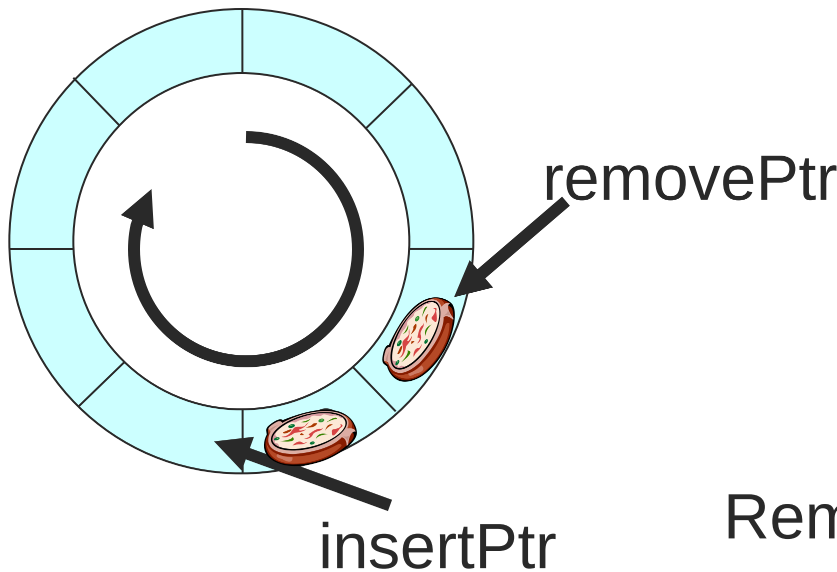


Remove pizza

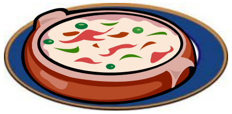


Producer-Consumer

Chef = Producer
Waiter = Consumer



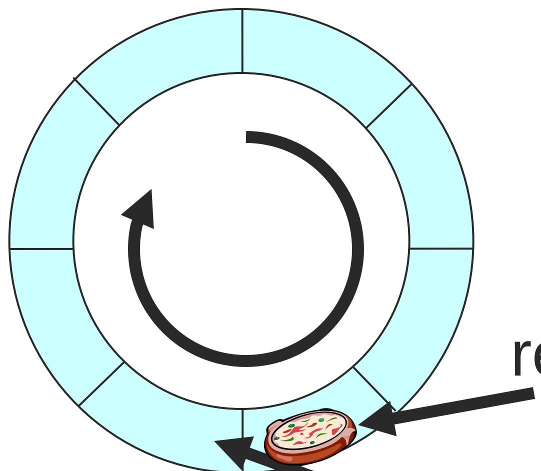
Remove pizza



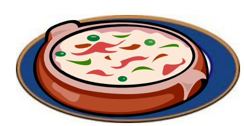
Producer-Consumer

Chef = Producer

Waiter = Consumer



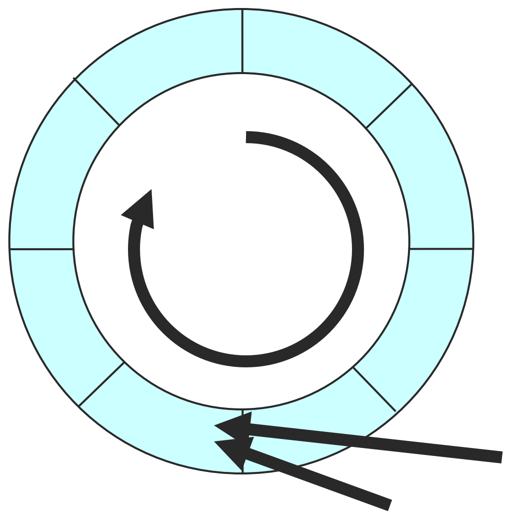
Remove pizza



Producer-Consumer

Chef = Producer
Waiter = Consumer

BUFFER EMPTY:
Consumer must be blocked!



Remove pizza

Producer-Consumer Summary

- Producer
  - Insert items
  - Update insertion pointer

- Consumer
  - Execute destructive read on the buffer
  - Update removal pointer

- Both
  - Update information about how full/empty the buffer is
- Solution
  - Must allow multiple producers and consumers

Designing a solution

Chef (Producer)

Waiter (Consumer)

Wait for empty slot
Insert item
Signal item arrival

Wait for item arrival
Remove item
Signal empty slot available

What synchronization do we need?

Challenges

- Prevent buffer <u>over</u>flow
- Prevent buffer <u>under</u>flow
- Mutual exclusion when modifying the buffer data structure

Assembling the solution

- Producer
  - `sem_wait(slots)`, `sem_signal(slots)`
  - Initialize semaphore `slots` to `N`
- Consumer
  - `sem_wait(items)`, `sem_signal(items)`
  - Initialize semaphore `items` to `0`
- Synchronization
  - `mutex_lock(m)`, `mutex_unlock(m)`
- Buffer management
  - `insertptr =  insertptr+1`
  - `removalptr =  removalptr+1`

Assembling the solution

- Producer
  - `sem_wait(slots)`, `sem_signal(slots)`
  - Initialize semaphore `slots` to `N`
- Consumer
  - `sem_wait(items)`, `sem_signal(items)`
  - Initialize semaphore `items` to `0`
- Synchronization
  - `mutex_lock(m)`, `mutex_unlock(m)`
- Buffer management
  - `insertptr = (insertptr+1) % N`
  - `removalptr = (removalptr+1) % N`

Producer-Consumer Code

Critical Section: move insert pointer

```
buffer[ insertPtr ] =
   data;
insertPtr = (insertPtr
   + 1) % N;
```

Critical Section: move remove pointer

```
result =
   buffer[removePtr];
removePtr = (removePtr
   +1) % N;
```

Producer-Consumer Code

Counting semaphore – check and decrement the number of free slots

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

Block if there are no free slots

Done – increment the number of available items

Counting semaphore – check and decrement the number of available items

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

Block if there are no items to take

Done – increment the number of free slots

Consumer Pseudocode: getItem()

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

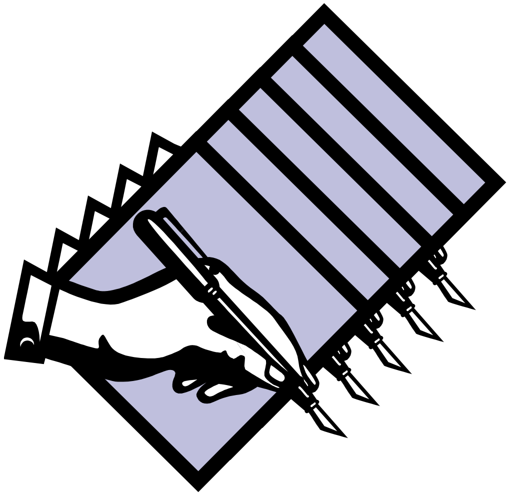
Error checking/EINTR handling not shown

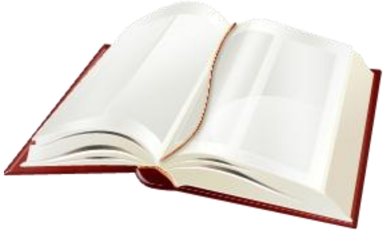
Producer Pseudocode: putItem(data)

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

Error checking/EINTR handling not shown

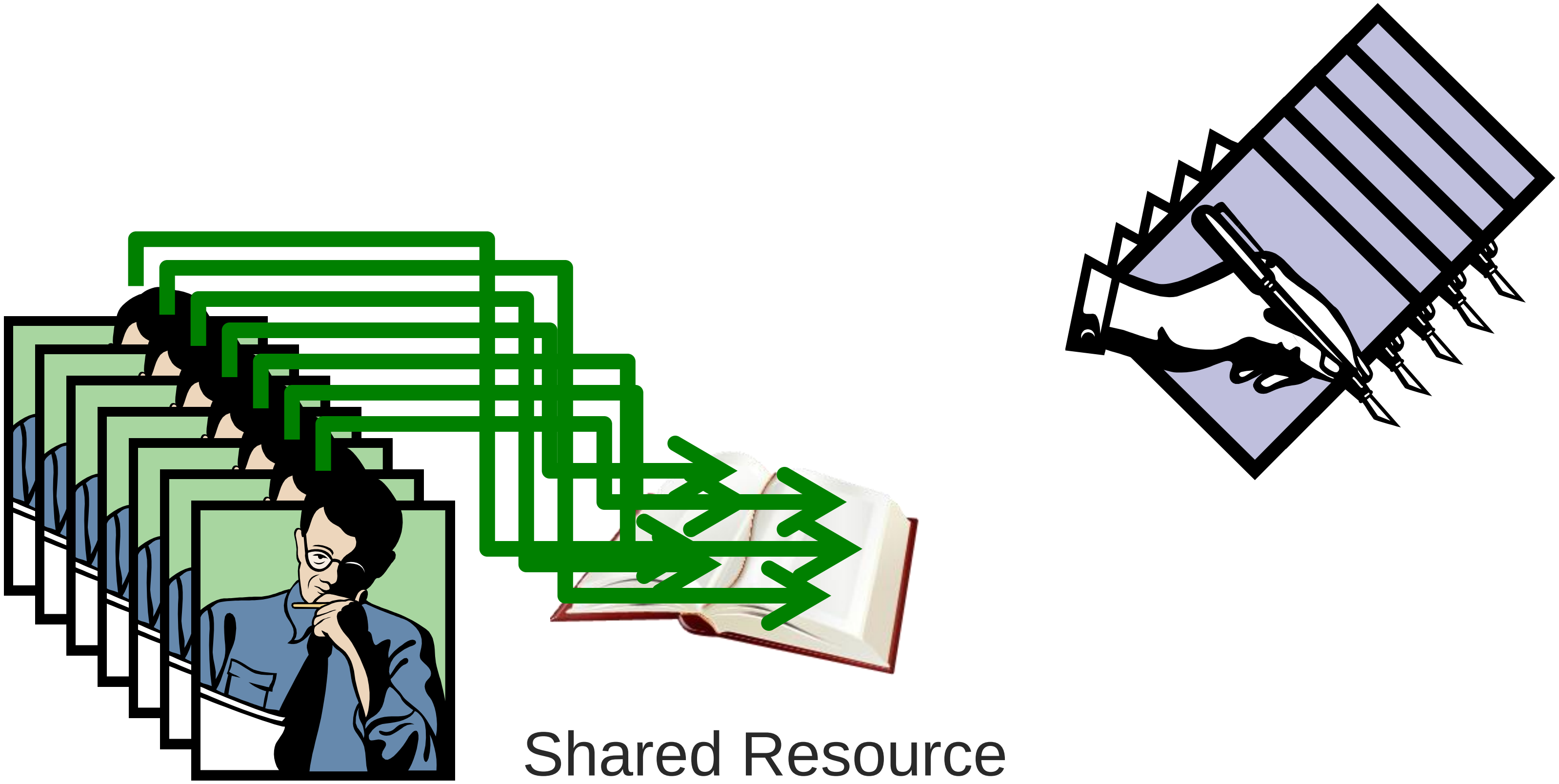
Readers-Writers Problem





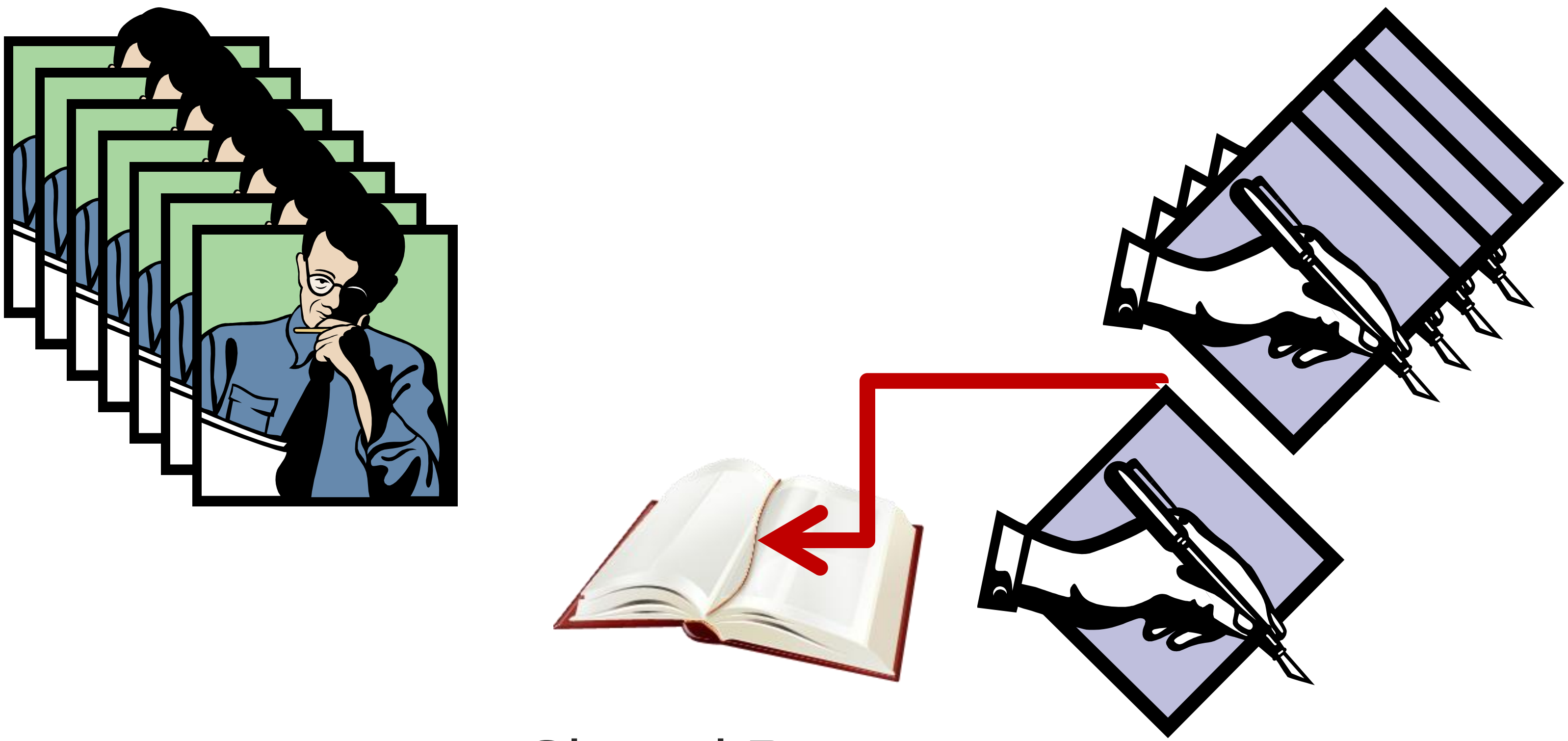
Shared Resource

Readers-Writers Problem



Shared Resource

Readers-Writers Problem



Shared Resource

Reader-Writer Problem

- Readers read data
- Writers 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: First Solution

```
int readCount = 0;
semaphore mutex = 1;
semaphore writeBlock = 1;
```

```
reader() {
  while(TRUE) {
    <other stuff>;
    sem_wait(mutex);
    readCount++;

    if(readCount == 1)
      sem_wait(writeBlock);
    sem_signal(mutex);

    /* Critical section */
       access(resource);

    sem_wait(mutex);
    readCount--;
    if(readCount == 0)
      sem_signal(writeBlock);
    sem_post(mutex);
  }
}
```

```
writer() {
  while(TRUE) {
    <other computing>;
    sem_wait(writeBlock);
    /* Critical section */
    access(resource);
    sem_signal(writeBlock);
  }
}
```

Reader-Writer: Second Solution

```
int readCount=0, writeCount=0;
semaphore mutex1=1, mutex2=1;
Semaphore readBlock=1,writeBlock=1
```

```
reader() {
  while(TRUE) {
    <other computing>;
    sem_wait(readBlock);
    sem_wait(mutex1);
    readCount++;
    if(readCount == 1)
      sem_wait(writeBlock);
    sem_signal(mutex1);
    sem_signal(readBlock);

    access(resource);
    sem_wait(mutex1);
    readCount--;
    if(readCount == 0)
      sem_signal(writeBlock)
    sem_signal(mutex1);
  }
}
```

```
writer() {
  while(TRUE) {
    <other computing>;
    sem_wait(mutex2);
    writeCount++;
    if(writeCount == 1)
      sem_wait(readBlock);
    sem_signal(mutex2);
    sem_wait(writeBlock);
    access(resource);
    sem_signal(writeBlock);
    sem_wait(mutex2);
    writeCount--;
    if(writeCount == 0)
      sem_signal(readBlock);
    sem_signal(mutex2);
  }
}
```

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?

Reader-Writer: Fairer Solution?

```
int readCount = 0, writeCount = 0;
semaphore mutex1 = 1, mutex2 = 1;
semaphore readBlock = 1, writeBlock = 1, writePending = 1;
```

```
reader() {
  while(TRUE) {
    <other computing>;
    sem_wait(writePending);
    sem_wait(readBlock);
    sem_wait(mutex1);
    readCount++;
    if(readCount == 1)
      sem_wait(writeBlock);
    sem_signal(mutex1);
    sem_signal(readBlock);
    sem_signal(writePending);
    access(resource);
    sem_wait(mutex1);
    readCount--;
    if(readCount == 0)
      sem_signal(writeBlock);
    sem_signal(mutex1);
  }
}
```

```
writer() {
  while(TRUE) {
    <other computing>;
    sem_wait(writePending);
    sem_wait(mutex2);
    writeCount++;
    if(writeCount == 1)
      sem_wait(readBlock);
    sem_signal(mutex2);
    sem_wait(writeBlock);
    access(resource);
    sem_signal(writeBlock);
    sem_signal(writePending);
    sem_wait(mutex2);
    writeCount--;
    if(writeCount == 0)
      sem_signal(readBlock);
    sem_signal(mutex2);
  }
}
```

Summary

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

Dining Philosophers

- N philosophers and N forks
  - Philosophers eat/think
  - Eating needs 2 forks
  - Pick one fork at a time