CSCE 313
Unix Threads MP7

Wei Zhang

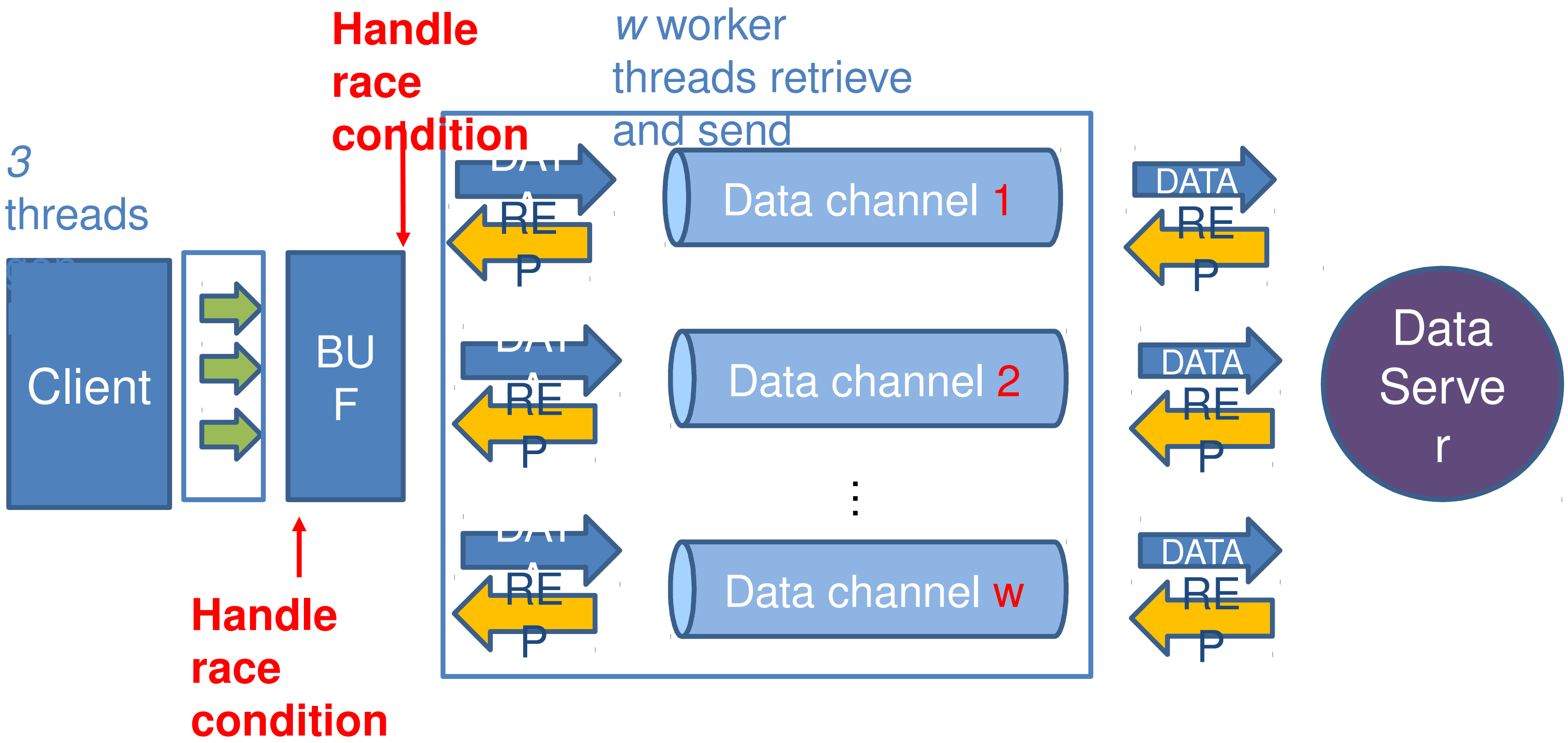
DUE: FRI APR 21, 2017

Acknowledgment: Yi Cui, Wei Zhang, Prof. Tanzir Ahmed, CSCE-3

MP 6: Recap

- Client creates 3 threads
  - Generate *3n* DATA in the buffer, each thread generate one type of data.
  - Run concurrently
    - Feed the data to the buffer, handle race condition!
- Then creates *w* worker threads
  - Each thread creates its own data channel
  - Run concurrently
    - Retrieve data form the buffer
    - Send the data to the server, handle race condition!

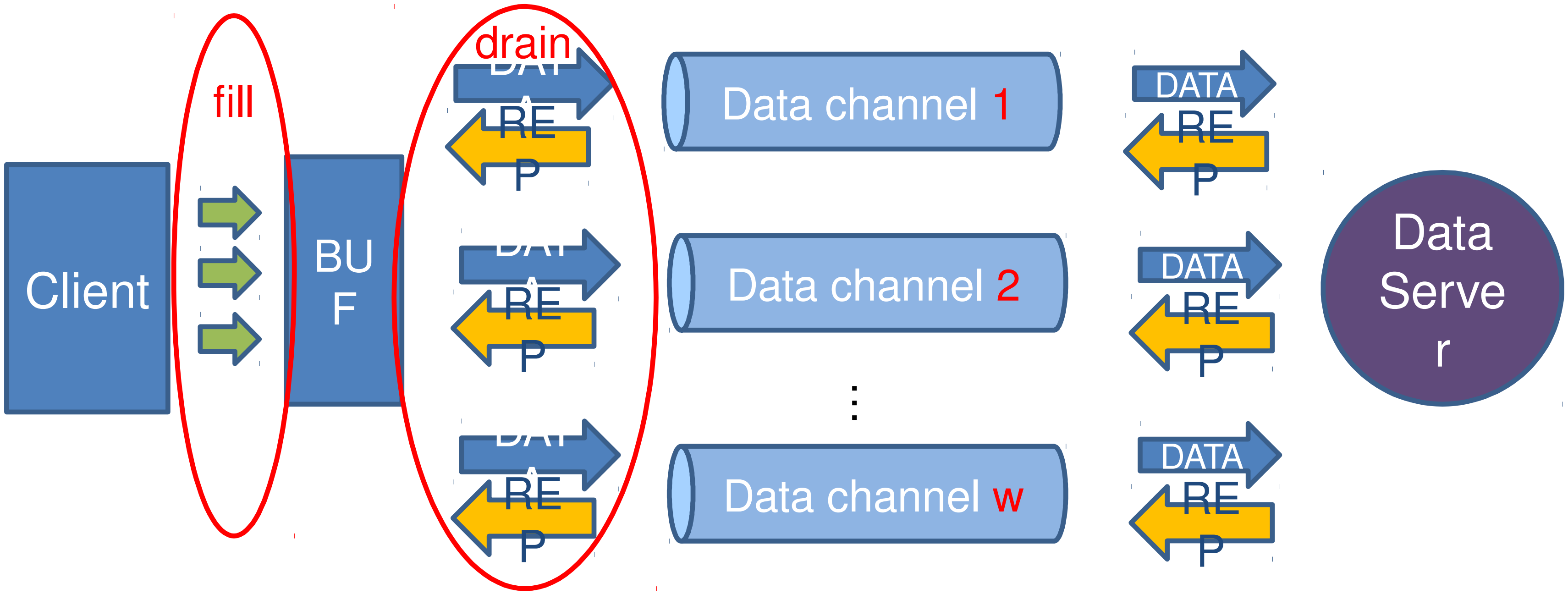
MP6: recap



Handle race condition
w worker threads retrieve and send
3 threads
Client
BUF
DATA
REP
Data channel 1
Data channel 2
⋮
Data channel w
Data Server
Handle race condition

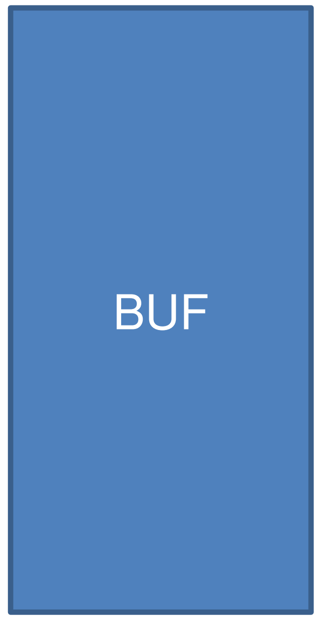
MP 6 Limitations

- Limitation #1: Fill the buffer first (with 3 threads), then drain the buffer (using w threads)
  - –FIX: Can fill and drain happen simultaneously?



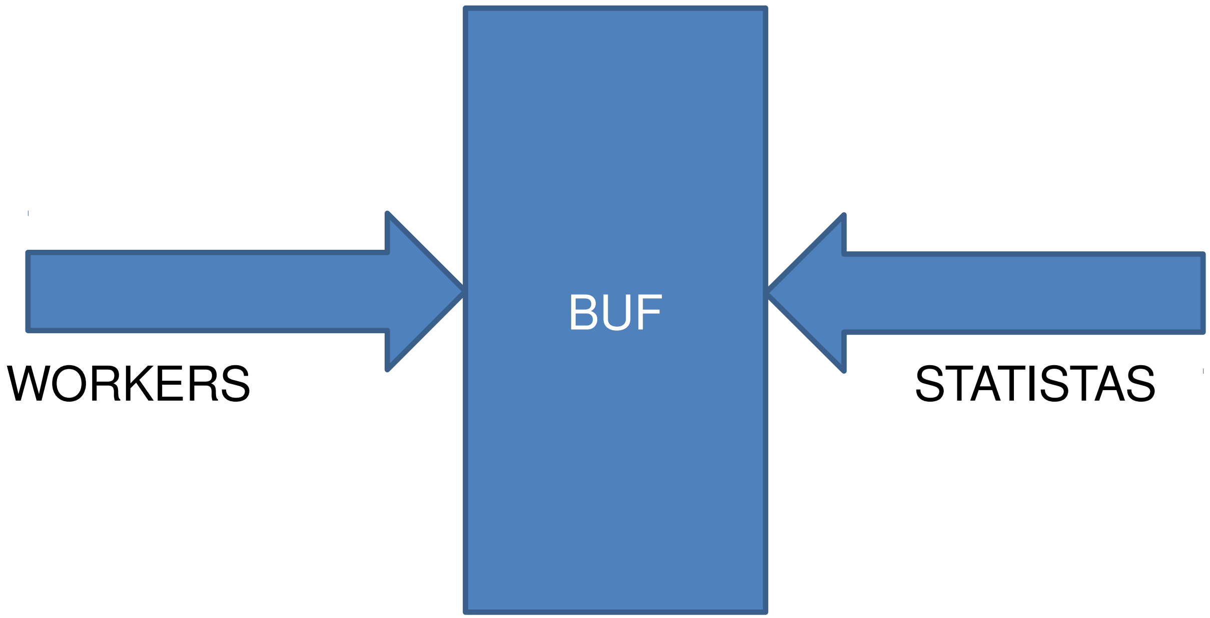
MP 6 Limitations

- Limitation #2: The buffer can be infinite in MP5
    - We have no idea how many data a user can generate
  - FIX: Can we limit the size of the buffer?



MP 6 Limitations

- Limitation #3: Poor modularity for worker threads
  - Two distinct roles – (a) interact with server and (b) create stats.
  - FIX: Can we devise a scheme to separate these two roles?

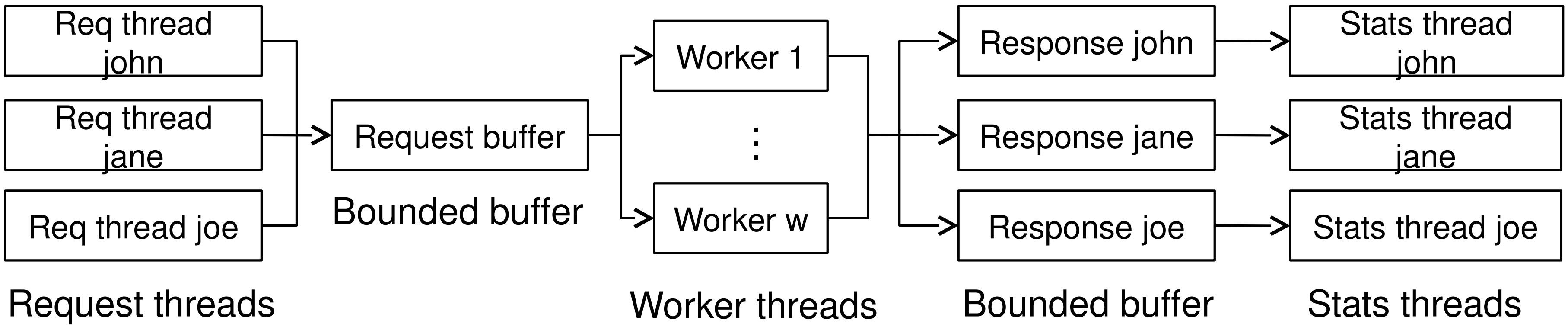


NET: What's new in MP7 relative to MP6?

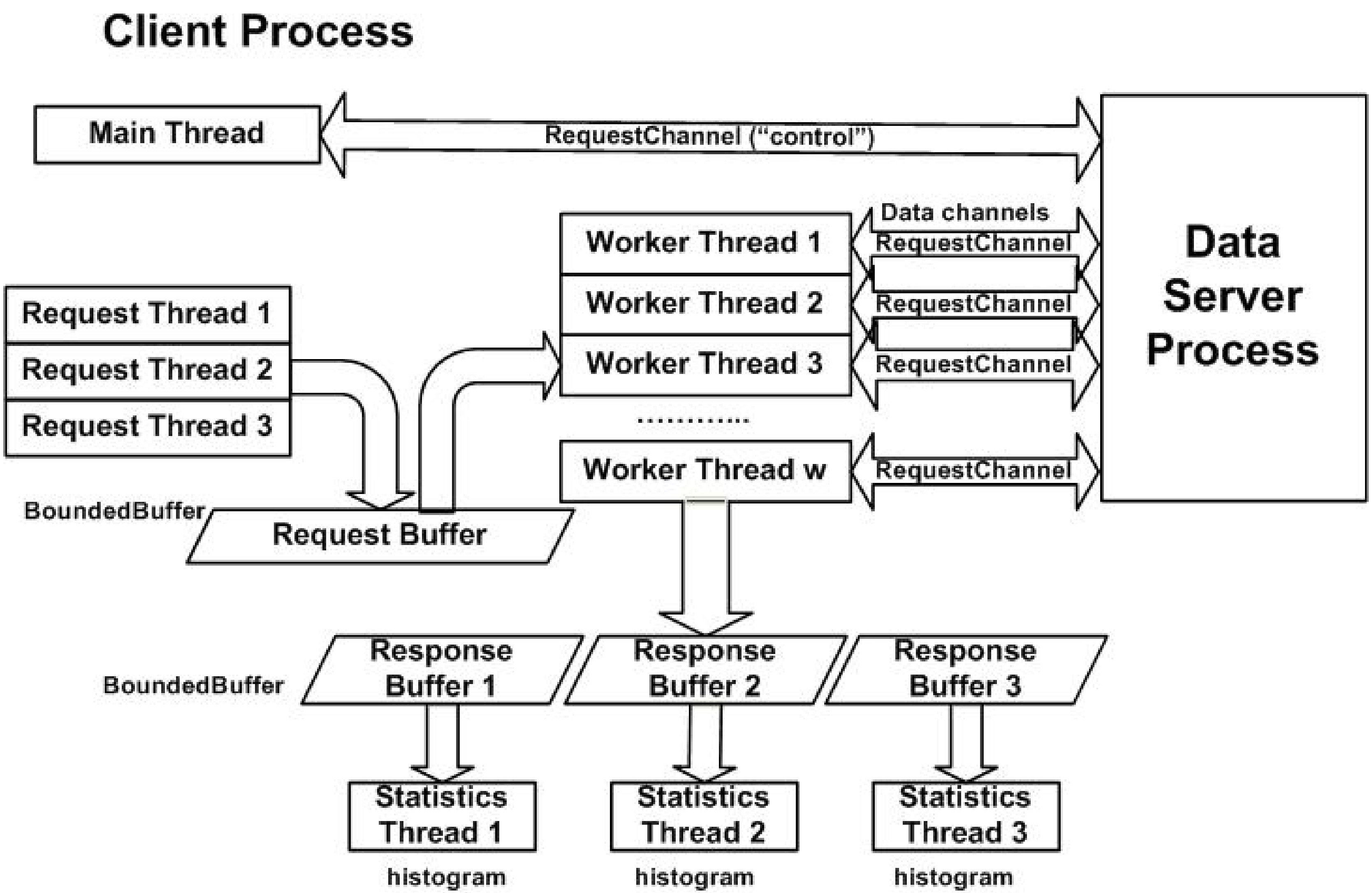
- Request threads can now run concurrently with worker threads
- Request Buffer is BOUNDED
- Worker Threads record server responses in a BOUNDED Response Buffer
- Statistics Threads run concurrently with Worker threads and compile frequency counts from entries in the Response Buffer

MP7 Structure

- Requests from John, Jane, and Joe are mixed into one request buffer
- Workers obtain responses from data server and send them to three separate buffers
- Each person has a dedicated stats thread



Another look at MP7 Structure



MP7 – Tangibles of the Assignment

- dataserver.cpp, reqchannel.h, reqchannel.cpp, and makefile are already done for you
- Your task
  - Implement bounded buffer, semaphore classes and client functions in accordance with the requirements published in the "Assignment" section of MP7_handout.pdf available here.
  - In addition, write a report with three key sections:
    - Performance Evaluation (especially relative to MP6)
    - Graph the runtime of your client program
      - execution time versus the number of worker threads (i.e. w)
      - execution time versus the size of request buffer (i.e. b)
    - Commentary on your client program performance in context of the system you ran it on

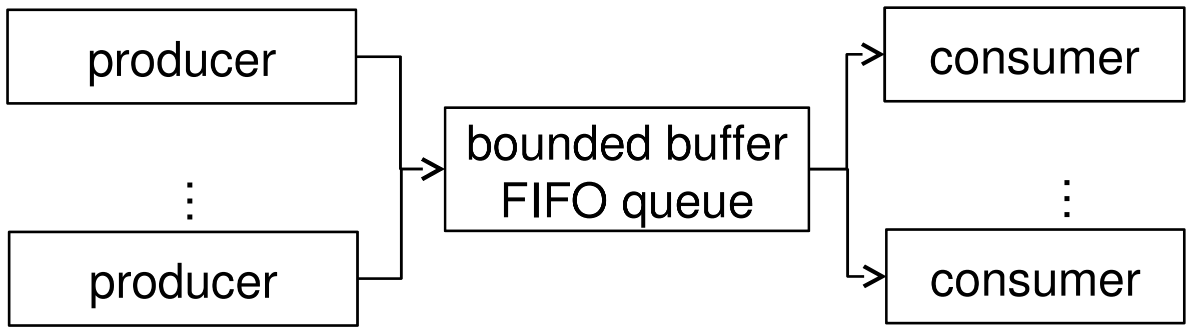
Key Reminders

- Make sure to take note of the specific DON'T(s) mentioned on pp. 4 of MP7_handout.pdf available here.
- Take note of the operations of Semaphore and Bounded Buffer classes mentioned on pp. 4-5 of MP7_handout.pdf available here.
- Consider taking advantage of the BONUS points opportunity mentioned on pp. 5-6 of MP7_handout.pdf available here.
- Read the rubrics carefully! They are listed on the last page pp. 7 of MP7_handout.pdf available here.

USEFUL BACKGROUND MATERIAL

Producer-Consumer

- A classical yet powerful multi-thread programming framework
  - Three parts: producers, consumers, and a bounded buffer
  - Simple and efficient: producers and consumers don't need to talk to each other, they both interact with the queue



Bounded Buffer

- In MP4, we do not set any bounds on queue size, which may lead to two problems
  - Overflow: the queue size can grow to infinity (limited by RAM size)
  - Underflow: consumers may attempt to pop an empty queue
- To solve the problems, we introduce bounded buffers
  - The queue size S is bounded by K. When S = K, producers must wait for consumers; when S = 0, consumers must wait for producers
  - Use semaphores to control the queue size

Semaphore

- You can simply view a semaphore as a counter
- When a thread calls sem.p()
  - Decrease the counter value C
  - If C >= 0, it can pass; otherwise, it is blocked, i.e., the maximum allowed number has been reached
- When a thread calls sem.v()
  - Increase the counter value C
  - If C <= 0, release a blocked thread

Semaphore - Example

- NOT standardized, you need to implement it.
- Let sem be a semaphore
  - Two operations
    - $sem.p()$
    - $sem.v()$
- A class called semaphore (or other names)
  - A int variable $count$
  - A mutex variable mu
  - A conditional variable q

$p()$ implementation - Example

Implementation - Example

- $sem.p()$
  - $-mu.lock()$
  - $-count--$
  - $-mutex.unlock()$

```
sem.p()
{
   mu.lock();
   count--;
   if (count<0) {
      cond_wait(&q,&mu)
   }
   mu.unlock();
}
```

v() implementation - Example

implementation - Example

- $sem.p()$
  - $-mu.lock()$
  - $-count--$
  - $-mutex.unlock()$

```
sem_v()
{
   mu.lock();
   count++;
   if (count<=0) {//if Q is empty
      cond_signal(&q);
   }
   mu.unlock();
}
```

Use semaphore

- Retrieving data (standardized routine)

- Retrieving data (standardized routine)

  - $full.p()$
  - Remove the data from the buffer
  - $empty.v()$

- Inserting data (standardized routine)

- Inserting data (standardized routine)

  - $empty.p()$
  - Add the data to the buffer

  - Add the data to the buffer

  - $full.v()$

Producer-Consumer Implementation

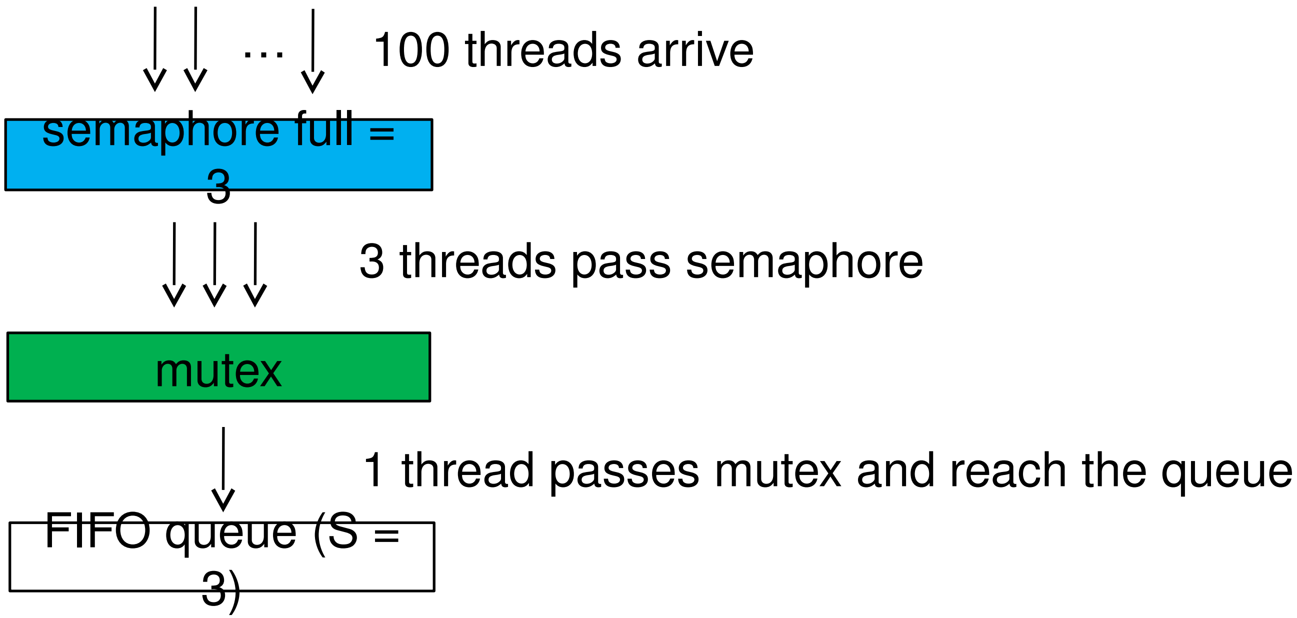
- A standard approach
  - Two semaphores full and empty, which count the number of items and empty slots in the queue
  - empty initialized to buffer capacity, full initialized to 0
  - A mutex m to control concurrency on queue operations

```
Producer() {
    while (true) {
        // decrease the counter of empty slots
        empty.p();
        m.lock();
        add one item to the queue
        m.unlock();
        // increase the counter of items
        full.v();
    }
}
```

```
Consumer() {
    while (true) {
        // decrease the counter of items
        full.p();
        m.lock();
        pop one item from the queue
        m.unlock();
        // increase the counter of empty slots
        empty.v();
    }
}
```

Concurrency Control

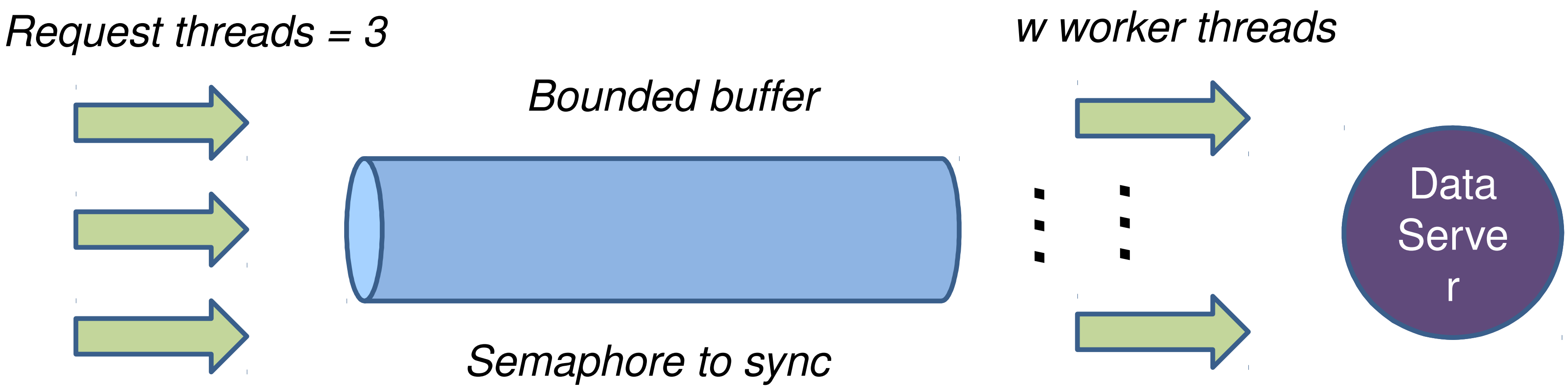
- In the consumers perspective, assume there are currently 3 items in the queue (semaphore full = 3)



- Each consumer that reaches the queue must be able to find one item to consume

Parting Reminder

- 3 request threads
  - 1 bounded buffer for requests
    - The w worker threads convey requests to the dataserver



Parting Reminder

- 3 statistics threads
  - Draw histogram
  - 3 bounded buffers for response reception
    - The w worker threads put the corresponding responses to the right buffer