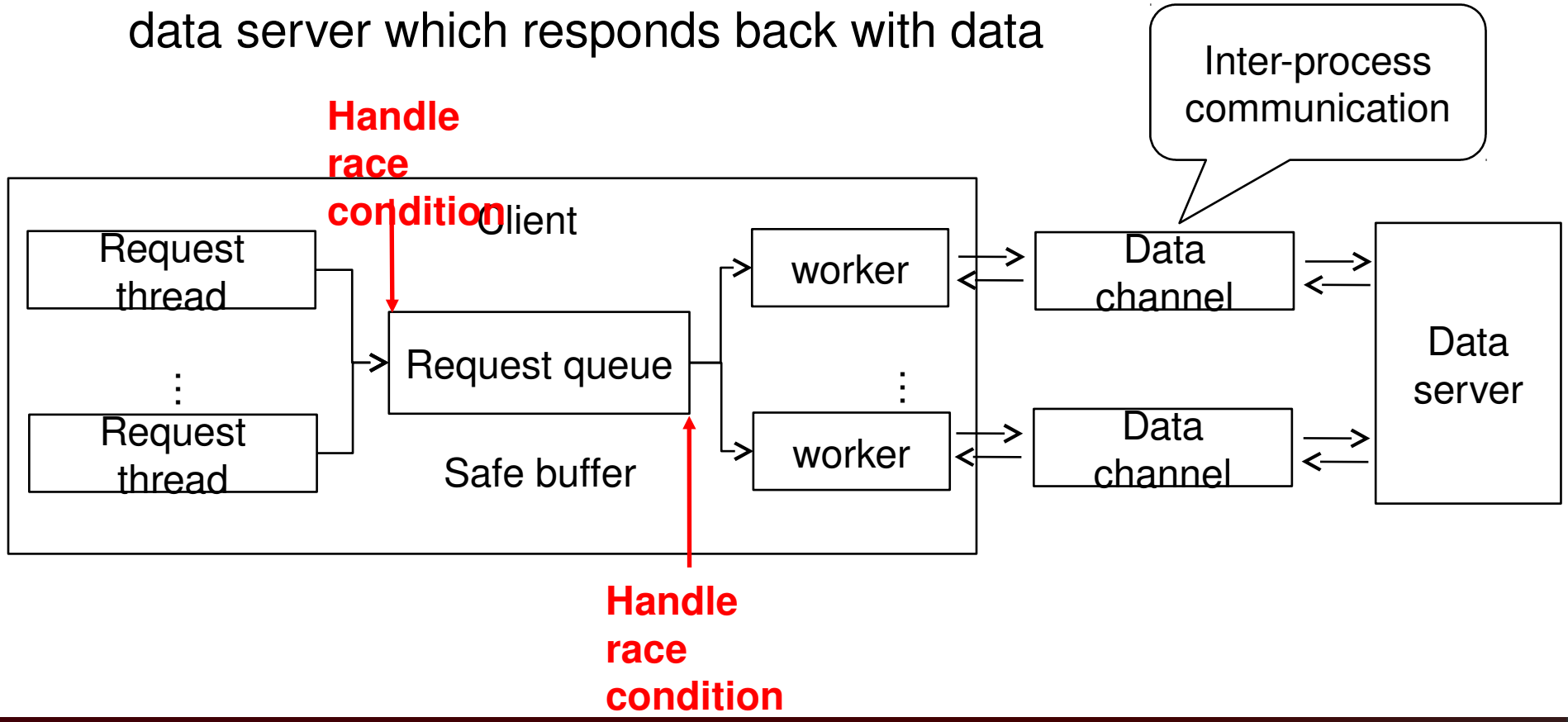
CSCE 313 MP6
Unix Threads

Wei Zhang

Acknowledgment: Yi Cui, GSTA, CSCE-313, Prof. Tanzir Ahmed, CSCE-

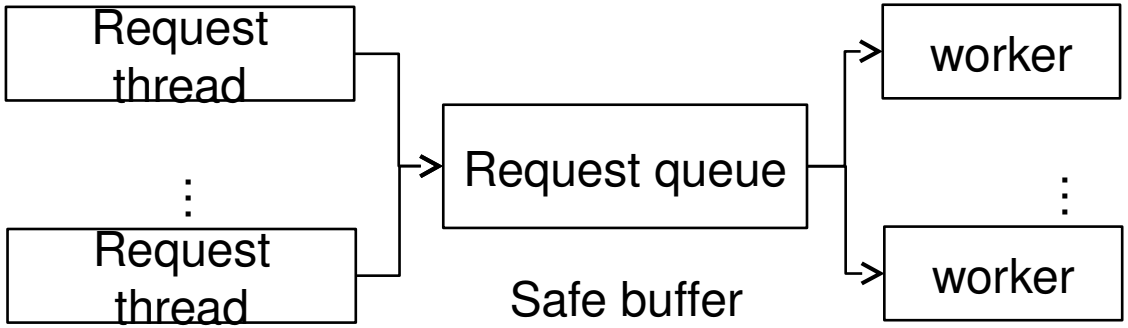
MP6 – Big Picture

- System structure of MP6
  - Two processes running: client and data server
  - Client registers requests in a buffer, then sends them to a data server which responds back with data



Client

- Request threads generate requests to the queue
- Worker threads pop requests, deliver them to the data server, collect histogram of response for each request type which happens to be a number between 0 - 99 (see typical output in backup slide)



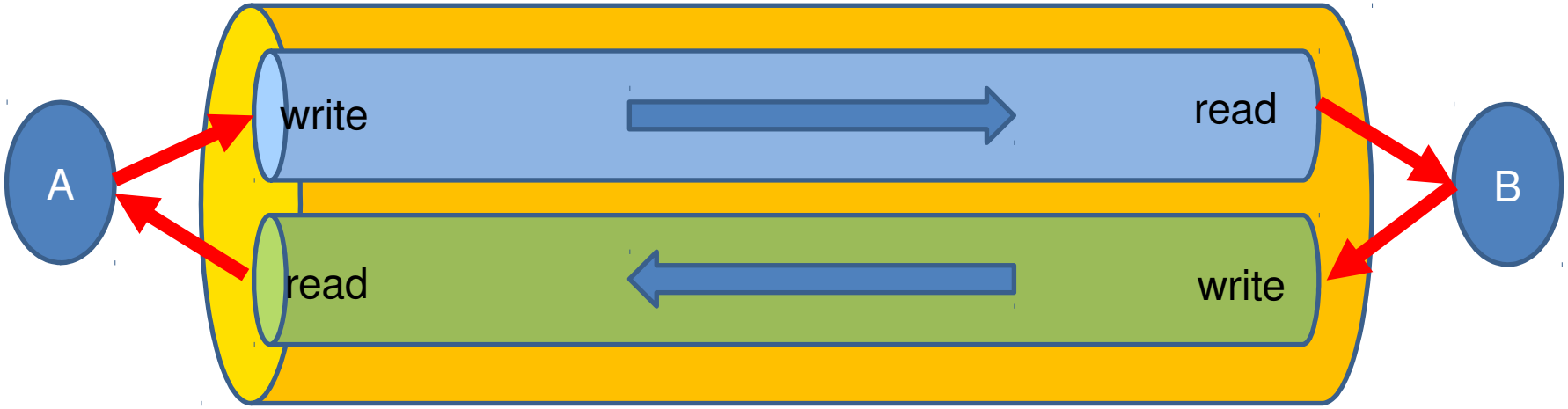
- Requests are sent from 3 requestors
  - Use 3 request threads, each sends n requests and quits; handle race condition!
- Create w workers to process the 3n requests
    - Retrieve data from the buffer
    - Send the data to the server and receive response from the server, handle race condition!
  - When do workers quit? Additionally add w “quit” requests in the client once all requests are entered; a worker thread quits when seeing “quit”

MP6 – Tangibles of the Assignment

- dataserver.cpp, reqchannel.h, and reqchannel.cpp are already done for you
- Your task
  - Implement safe buffer and client functions in accordance with the requirements published in MP6_handout.pdf available here.
  - Specifically, Modify SafeBuffer.h, SafeBuffer.cpp, and client_MP6.cpp
  - In addition, write a report with three key sections:
    - Describe your implementation
    - Graph the runtime of your client program (execution time versus the number of worker threads)
    - Commentary on your client program performance in context of the system you ran it on

Channel

- Bidirectional inter-process communication
- Not a standard concept, implemented with two unidirectional pipes



- See reqchannel.h and reqchannel.cpp for detail

Channel

- We first need a single control channel

```
RequestChannel *chan = new RequestChannel("control", RequestChannel::CLIENT_SIDE
```

- Then, we can create multiple data channels by sending new thread commands to control channel
  - One data channel per worker

```
td::string s = chan->send_request("newthread");
equestChannel *workerChannel = new RequestChannel(s, RequestChannel::CLIENT_SID
```

- Exchange message through data channel

```
std::string response = workerChannel->send_request(request);
```

Response from data server

Request from client

- See client.cpp for detail

Safe Buffer

- Thread safe
  - Multiple threads can work on the buffer
  - Use a lock to control concurrency
  - Only one thread can work on the queue at a time

```
SafeBuffer::push_back(dat
a) {
    lock
    Q.push_back(data);
    unlock
}
```

```
SafeBuffer::retrieve_front
() {
    lock
    data = Q.front();
    Q.pop_front();
    unlock
    return data;
}
```

Safe Buffer

- Protect the resource shared by multiple threads with a mutex lock.
  - E.g. if x++ is shared by two threads, then

    pthread_mutex_lock()

    x++

    pthread_mutex_unlock()

  - pthread_mutex_init() and pthread_mutex_destroy() is also necessary.
- For more details and examples: http://www.thegeekstuff.com/2012/05/c-mutex-examples/

BACKUP

Typical output from running client

```
Results for n == 100, w == 1
              John Smith     Jane Smith      Joe Smith
0-9                    7             10             11
10-19                  4              9             11
20-29                 15             13             14
30-39                 10              8             11
40-49                  7             10              6
50-59                 17             10              8
60-69                  8             15              8
70-79                 10              6              5
80-89                 10              8             17
90-99                 12             11              9
Total                100            100            100
```

HINT: You'd be well advised to write a sanity checker code inside the client that sums up the values across the entire range of histogram for any given requestor and verifies it to be equal to the # of requests from that requestor. This will ensure that your worker threads all completed correctly.