Lab 5. Simple KVS

SNU System Programming Assignment

Junghan Yoon
SNU TNET Lab.

What You Should Do

1. Implement basic server/client
    - Use socket APIs
    - Use multiple threads
    - Use skvs helper function

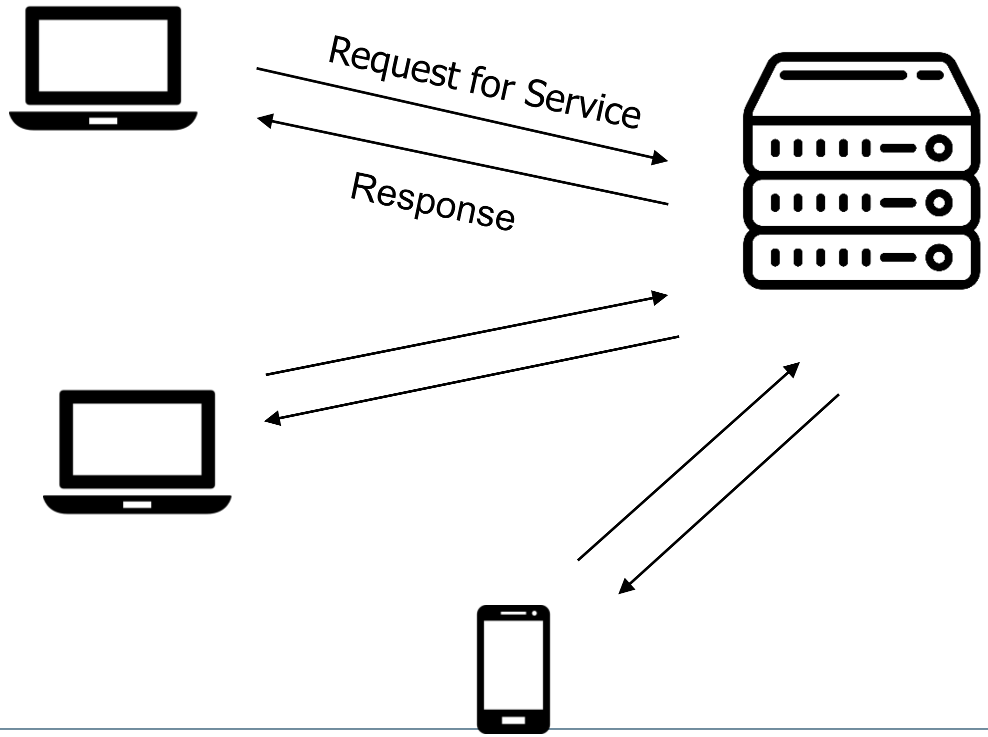
2. Implement a global hash table and rwlock
    - Multiple threads may access at the same time
    - Global hash table uses rwlock library
    - rwlock should support multiple concurrent readers when there is no writer

- Reference for socket programming in C:
    - https://beej.us/guide/bgnet/
- Reference for rwlock:
    - https://docs.oracle.com/cd/E37838_01/html/E61057/sync-124.html
    - https://www.ibm.com/docs/en/aix/7.3?topic=p-pthread-rwlock-rdlock-pthread-rwlock-tryrdlock-subroutines

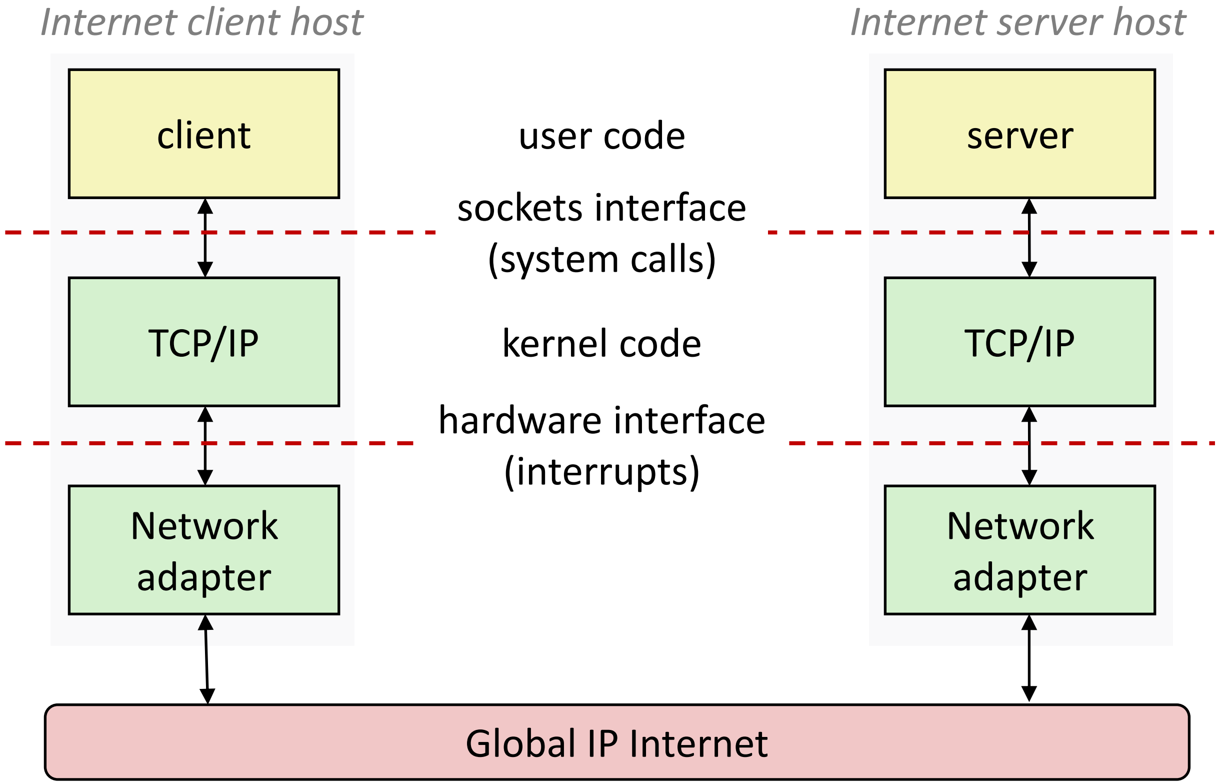
Background 1:
Basic Server/Client

What is the Client-Server Model?

- **Definition**: A distributed computing architecture where a **client** requests services, and a **server** provides them.
- **Structure**: Typically involves multiple clients connecting to a centralized server.



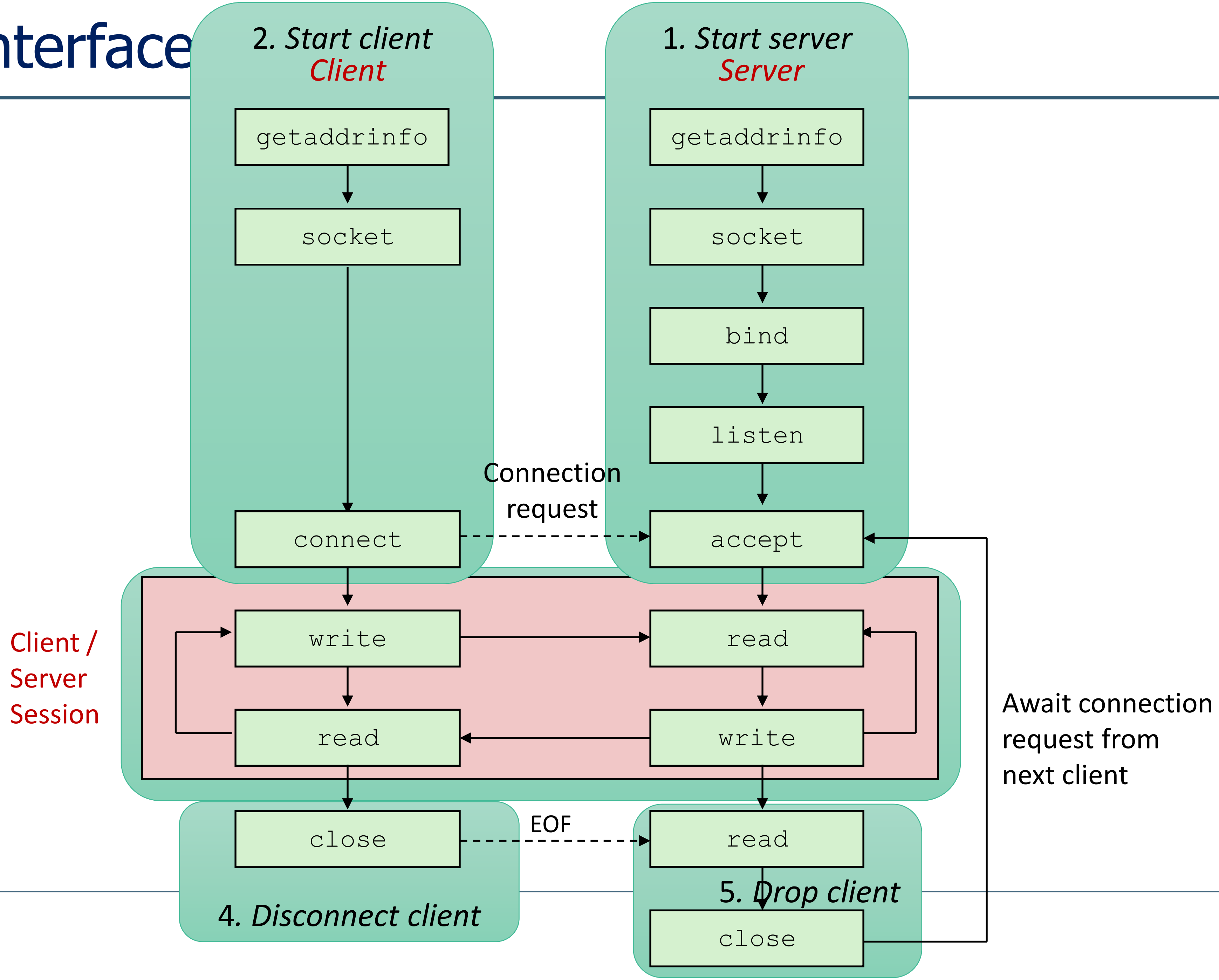
Programmer's View



- Both client and server use socket interface!
- Not only for TCP, but also for UDP

TCP? UDP?

TCP Socket Interface



System Calls for Basic Server

- `getaddrinfo() & freeaddrinfo()`
  - Helper function for IP lookup
  - You may use `gethostbyname()` or `gethostbyname_r()`

For this assignment, use 0.0.0.0 for binding server IP address

- `socket()`
  - Makes a socket and allocates system resources for the socket

Socket descriptor,
Socket structure for metadata,
sk_buff, …

- `bind()`
  - Binds IP address and port to the socket
- `listen()`
  - Creates a connection queue to allow connections from clients

Listen, accept: for TCP
TCP server has 2 sockets

- `accept()`
  - Retrieves a connection from the connection queue and **returns a socket** for an established connection
- `read() & write()`
  - Next slide
- `close()`
  - Close the socket and cleans up resources

System Calls for Basic Client

- `getaddrinfo() & freeaddrinfo()`
  - Helper function for IP lookup
  - You may use `gethostbyname()` or `gethostbyname_r()`
- `socket()`
  - Makes a socket and allocates system resources for the socket
- `bind()`
  - Binds IP address and port to the socket
- `connect()`
  - Creates a connection to the specified address:port (starts TCP handshake)
- `read() & write()`
  - Next slide
- `close()`
  - Close the socket and cleans up resources

Socket descriptor,
Socket structure for metadata,
sk_buff, …

connect: for TCP
TCP client has 1 socket

read()

- `ssize_t read(int fd, void *buf, size_t nbytes)`

- `Returns`
  - # of bytes read (> 0)
  - -1 (error, you should check `errno`)
    - No bytes read
  - 0 (closed by the peer)
- `errno == EAGAIN, EWOULDBLOCK`
  - Nothing to read from the TCP socket recv buffer, try later
- `errno == EINTR`
  - Failed due to interrupt, try again right now
- `errno == ECONNRESET`
  - The peer abruptly reset the connection

write()

- ssize_t write(int fd, void *buf, size_t nbytes)

- Returns
  - # of bytes wrote (> 0)
  - -1 (error, you should check errno)
    - No bytes wrote

- errno == EAGAIN, EWOULDBLOCK
  - TCP socket send buffer is full, try later
- errno == EINTR
  - Failed due to interrupt, try again right now
- errno == ECONNRESET
  - The peer abruptly reset the connection
- errno == EPIPE
  - The peer closed the connection

Background 2:
Lock

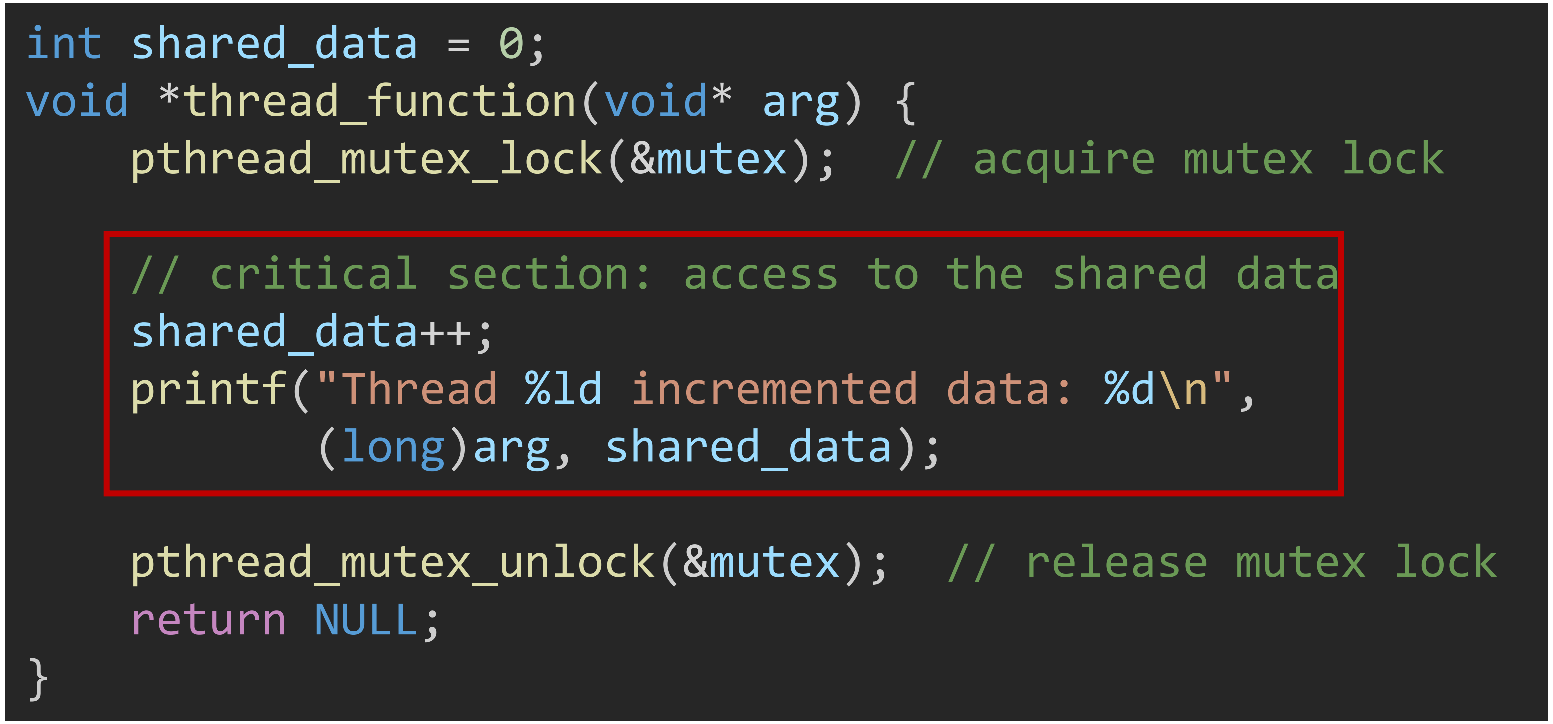
Spin Lock vs. Mutex Lock

- Spin Lock
  - Continuously checks if the lock is available while consuming CPU cycles (busy-waiting)
  - Optimized for short wait times and multi-core environments
- Mutex Lock
  - Puts the waiting thread to sleep, releasing the CPU until the lock becomes available
  - Suitable for longer wait times or resource-intensive applications

Aspect	Spin Lock	Mutex Lock
Waiting	Busy-waiting (CPU is fully utilized)	Sleep-waiting (CPU is released)
Overhead	Low (no context switching)	High (context switching involved)
Use case	Short wait times	Long wait times
Multi-core	Effective in multi-core systems	Works well in both single/multi-core
Complexity	Simple	Relies on OS support

Mutex Lock

- Protects shared data
- Allows only one thread to access the critical section

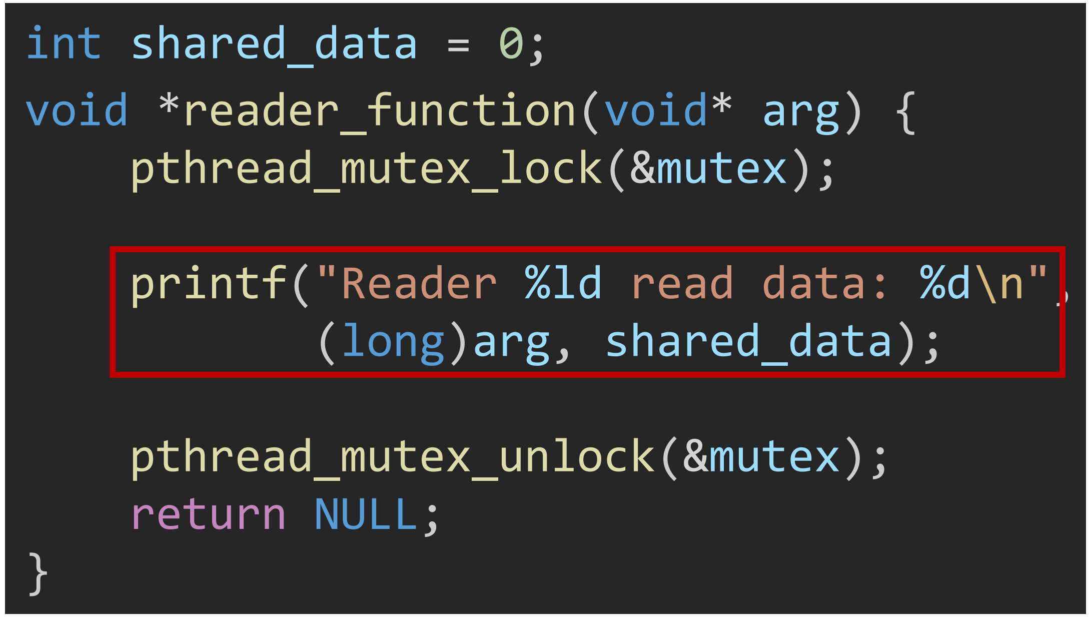


```
int shared_data = 0;
void *thread_function(void* arg) {
    pthread_mutex_lock(&mutex);  // acquire mutex lock

    // critical section: access to the shared data
    shared_data++;
    printf("Thread %ld incremented data: %d\n",
           (long)arg, shared_data);

    pthread_mutex_unlock(&mutex);  // release mutex lock
    return NULL;
}
```

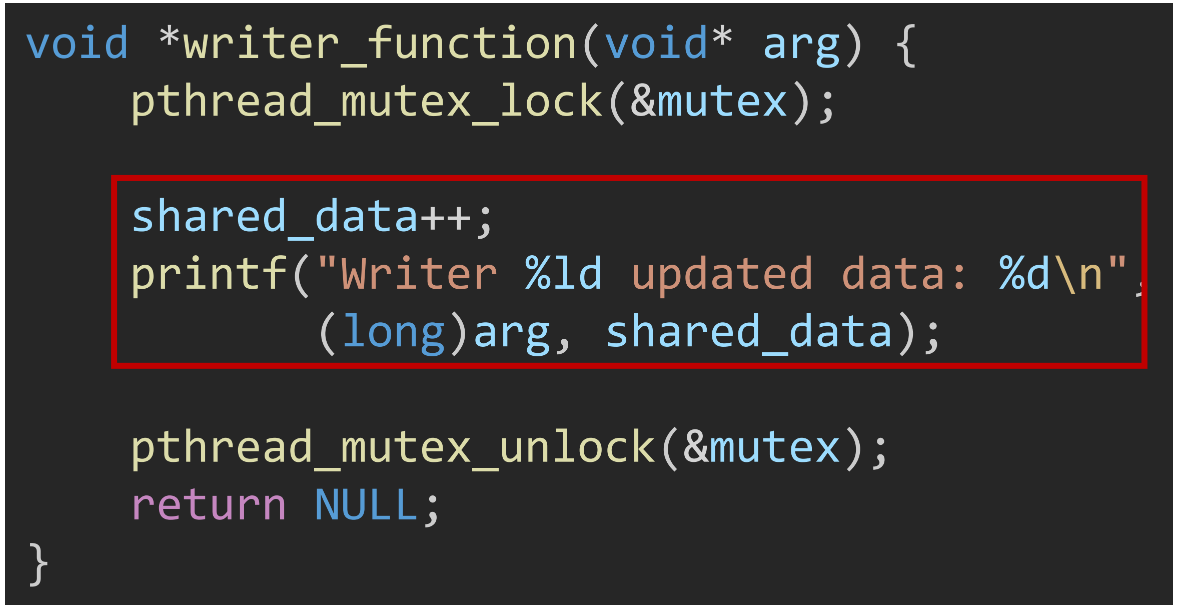
- What if most of the accesses are reads?

Many Readers & Few Writers

- Contention necessary?



```
int shared_data = 0;
void *reader_function(void* arg) {
    pthread_mutex_lock(&mutex);

    printf("Reader %ld read data: %d\n",
           (long)arg, shared_data);

    pthread_mutex_unlock(&mutex);
    return NULL;
}
```



```
void *writer_function(void* arg) {
    pthread_mutex_lock(&mutex);

    shared_data++;
    printf("Writer %ld updated data: %d\n",
           (long)arg, shared_data);

    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

- If there is no concurrent writer, multiple threads can concurrently read

Reader-Writer Lock

- A specialized lock that distinguishes b/w read and write access

1. Allows concurrent multiple readers
   - Blocks writer when any readers exist
   - When there is no more readers, wake up one pending writer (Which one?)
2. Allows single writer only
   - Blocks other threads when writing
   - When there is no more writers, wake up all pending readers, then wake up one pending writer

Aspect	Mutex Lock	Reader-Writer Lock
Concurrency	Only one thread (reader or writer) at a time	Multiple readers or a single writer
Performance	Suitable for low read-to-write ratio	Optimized for high read-to-write ratio
Complexity	Simpler	More complex
Starvation Risk	No starvation (single queue)	Writer starvation
Use Case	Any critical section	Scenarios with frequent reads and rare writes

- By default, reader-writer lock is usually **reader-priority**

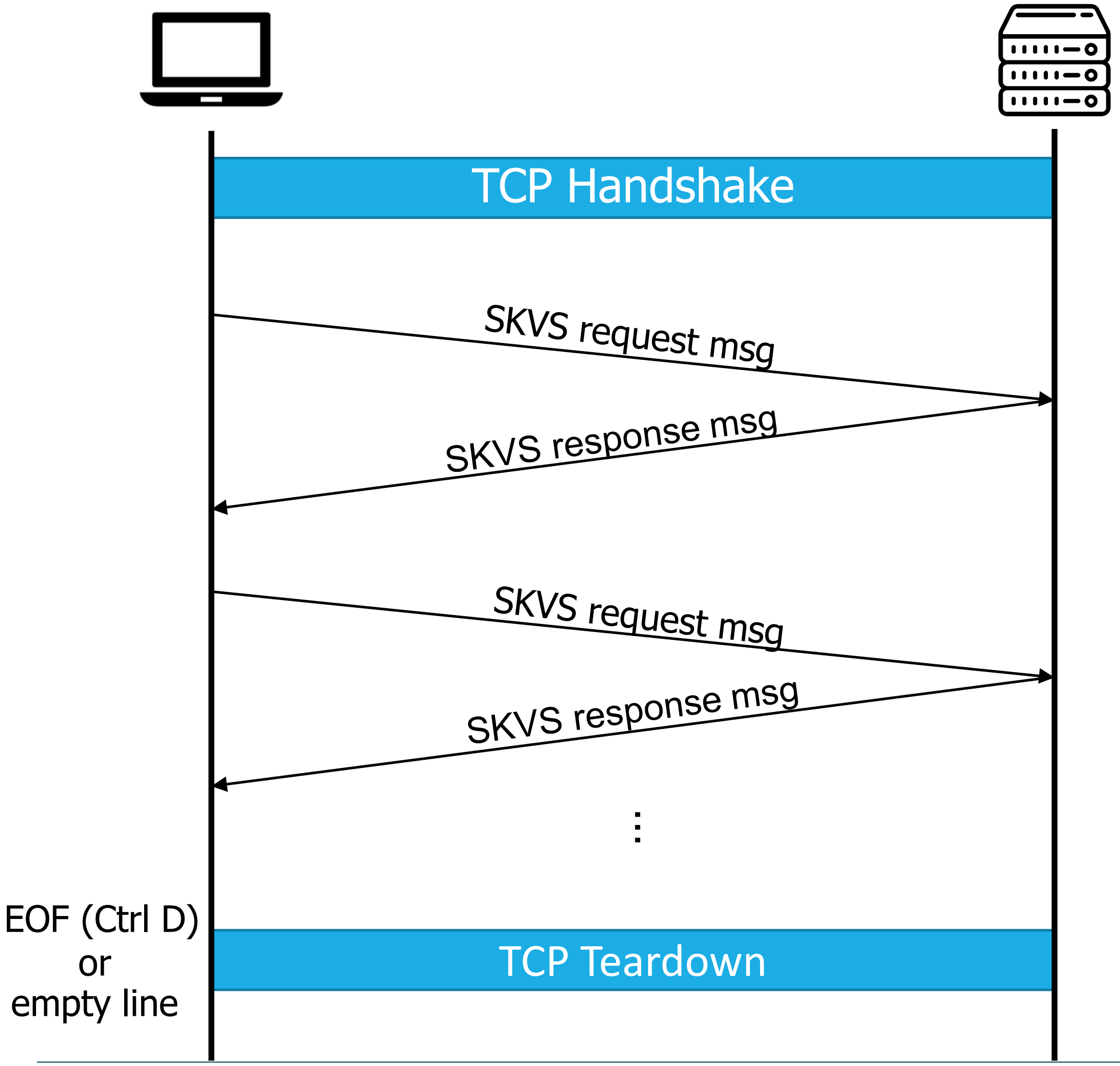
Reader-Priority vs. Writer-Priority (Out of Scope)

- Reader-Priority RW Lock
  - A read request is granted immediately
  - A write request waits until all ongoing read operations complete
  - If new requests keep arriving, the write request can be indefinitely delayed (writer starvation)
- Writer-Priority RW Lock
  - If a write request is waiting, any new read requests are delayed
  - Once ongoing read operations finish, the write request is executed immediately
  - After the write completes, pending read requests are processed
  - Subsequent write requests continue to take precedence over new reads

Aspect	Reader-Priority	Writer-Priority
Priority	Readers are prioritized	Writers are prioritized
Starvation Risk	Writers may starve	Readers may starve
Performance	Optimized for read-heavy workloads	Balances performance for both reads and writes
Complexity	Simpler	More complex
Use Case	Best for frequent reads	Suitable for fair scheduling b/w reads and writes

Part 1:
Simple Key-Value Store Protocol

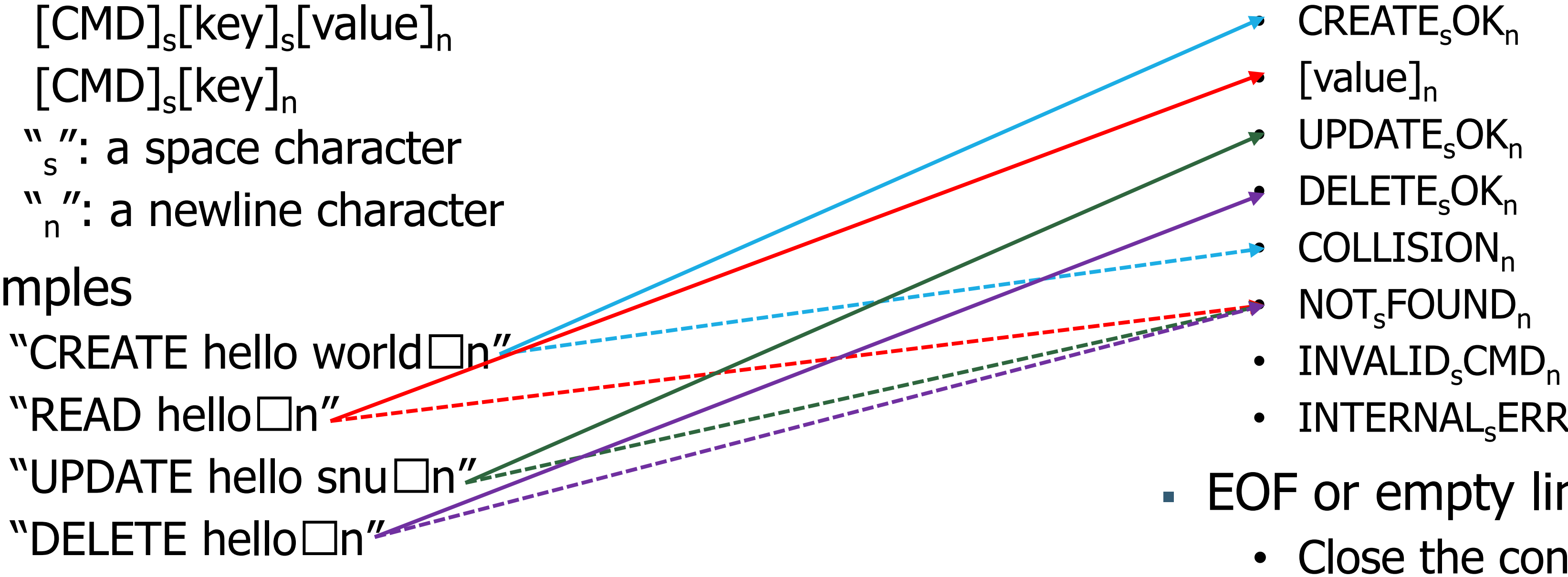
SKVS Protocol



- One connection per one client
- **Keep alive** until typing empty line (□n) or EOF (Ctrl D)
- **Half-duplex**
  - After sending request, wait for the response
- **Text** based protocol
  - Key and value should be also text
  - Refer to the nest slide
- Server should be **stateful**
  - Key-value pairs should be accessible by other clients
- Default service port: 8080

SKVS Protocol (cont.)

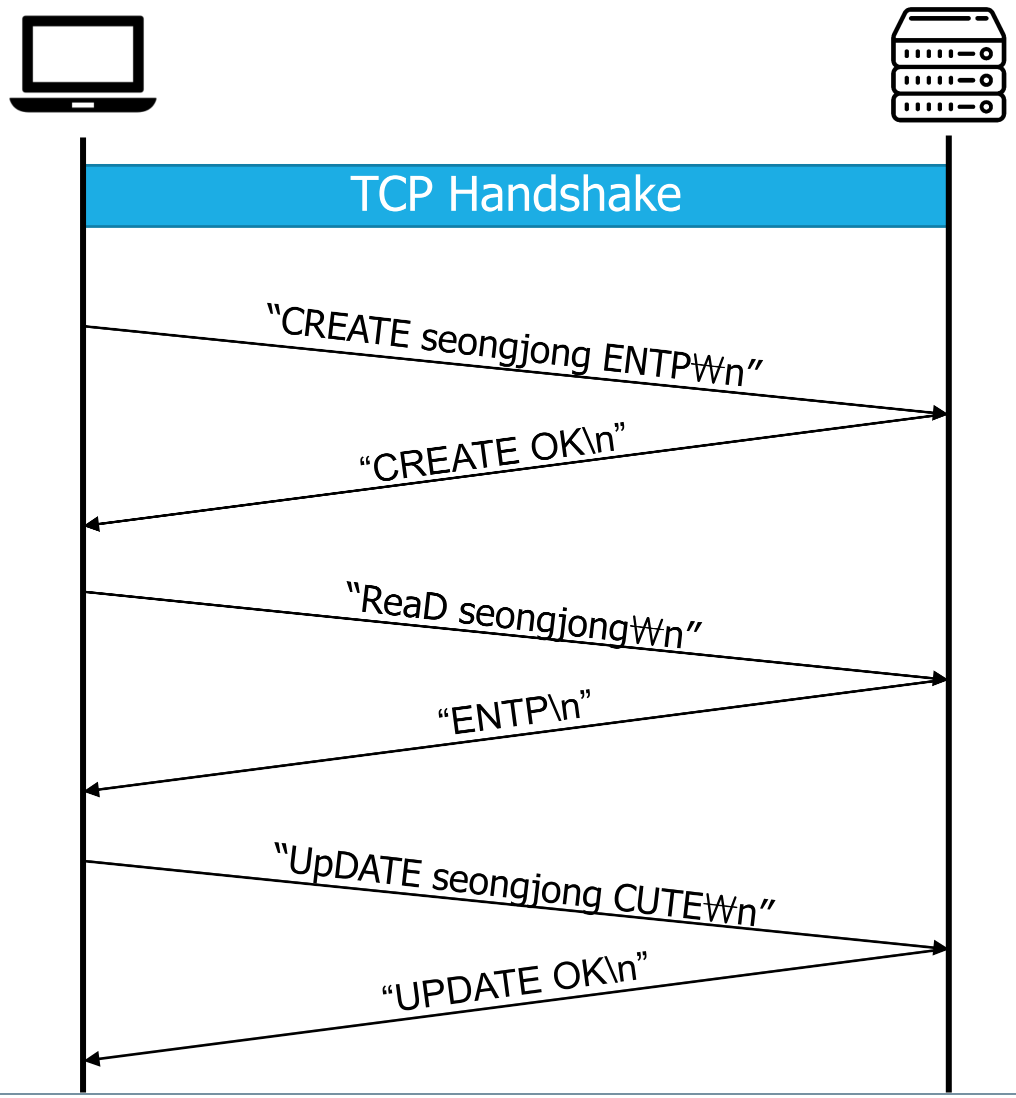
- Request Msg Format:
  1. [CMD]$_s$[key]$_s$[value]$_n$
  2. [CMD]$_s$[key]$_n$
  - "$_s$": a space character
  - "$_n$": a newline character



- Examples
  - "CREATE hello world□n"
  - "READ hello□n"
  - "UPDATE hello snu□n"
  - "DELETE hello□n"
- CMD: one of CREATE, READ, UPDATE, and DELETE
  - case-insensitive (e.g., rEAd, cReAtE are okay)
- Key, value: string without $_s$ nor $_n$
  - No binary, only text, case-sensitive
  - len(key) <= 32B, len(SKVS msg) <= 4096B
    - Including $_n$

- Response Msg Format:
  - CREATE$_s$OK$_n$
  - [value]$_n$
  - UPDATE$_s$OK$_n$
  - DELETE$_s$OK$_n$
  - COLLISION$_n$
  - NOT$_s$FOUND$_n$
  - INVALID$_s$CMD$_n$
  - INTERNAL$_s$ERR$_n$
- EOF or empty line on client
  - Close the connection
  - Exit client program
- Support for file option
  - `./client -f input.txt`
  - Read the input file as line by line
  - Close and exit after receiving last response
  - Again, EOF should close and exit

Example



TCP Handshake

"CREATE seongjong ENTP₩n"

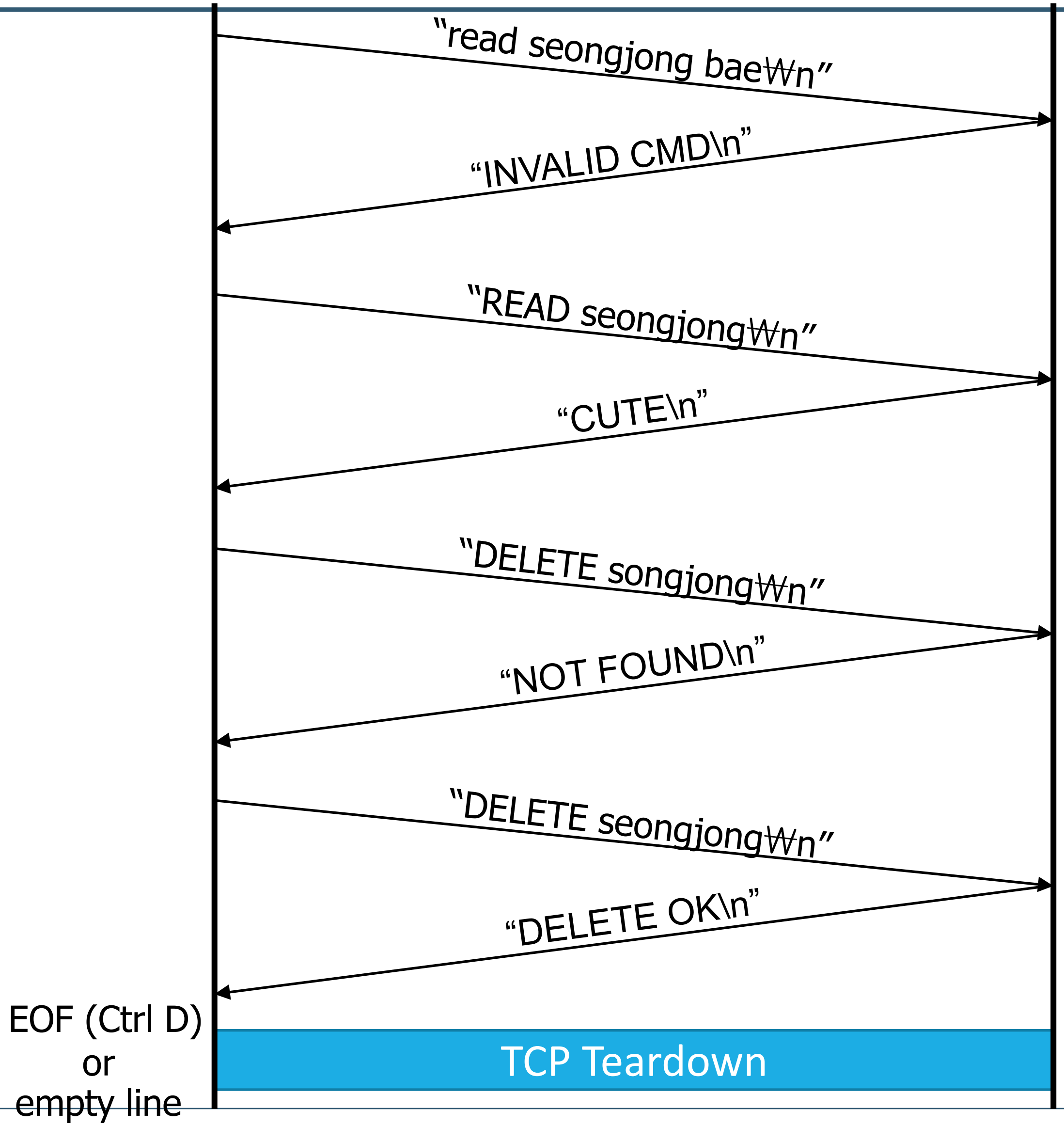
"CREATE OK\n"

"ReaD seongjong₩n"

"ENTP\n"

"UpDATE seongjong CUTE₩n"

"UPDATE OK\n"



"read seongjong bae₩n"

"INVALID CMD\n"

"READ seongjong₩n"

"CUTE\n"

"DELETE songjong₩n"

"NOT FOUND\n"

"DELETE seongjong₩n"

"DELETE OK\n"

EOF (Ctrl D)
or
empty line

TCP Teardown

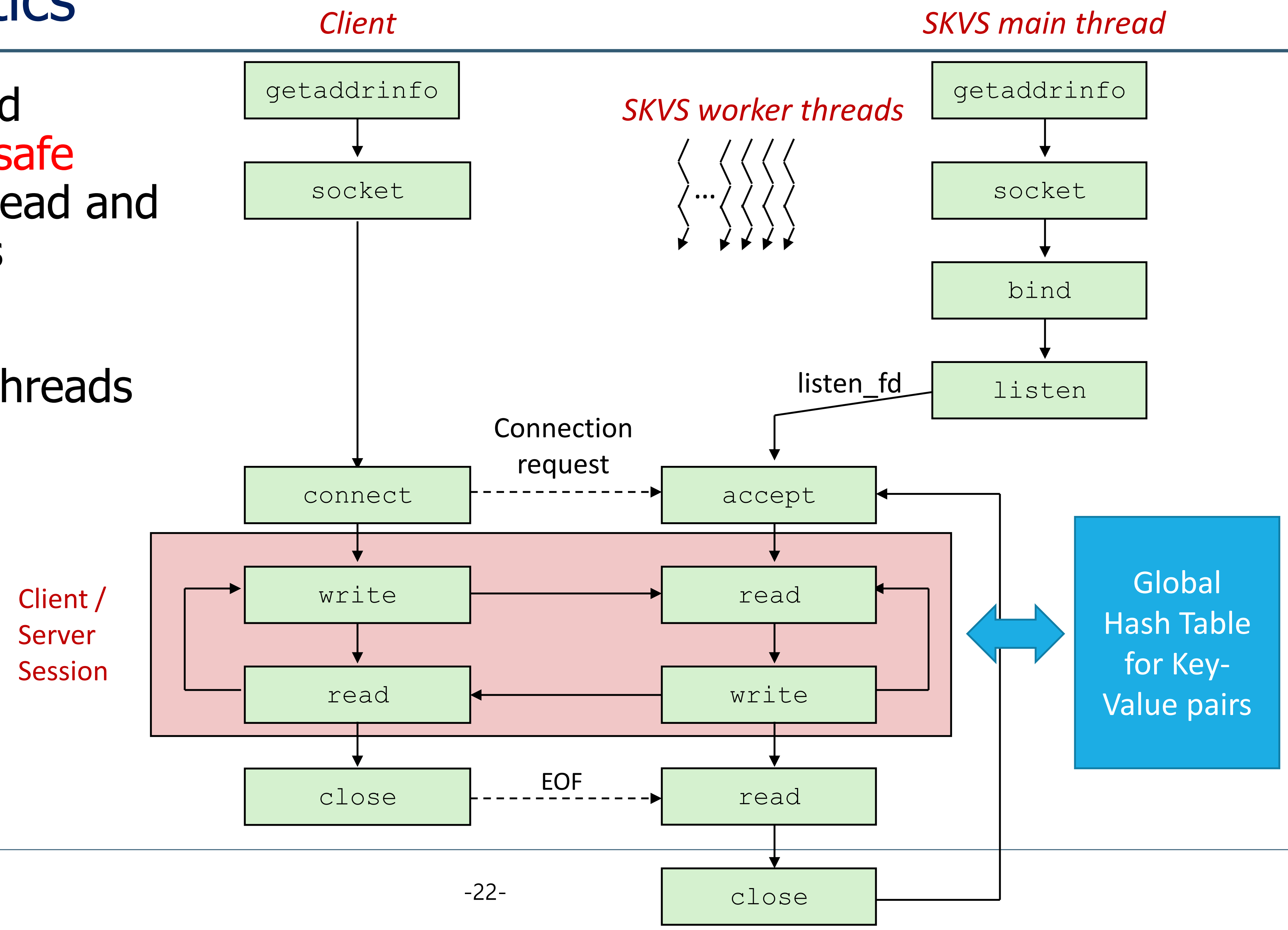
Handling Errors

- If SKVS request is in invalid format?
  - Responds “INVALID CMD□n”
- If the length of SKVS msg > 4096
  - Responds “INVALID CMD□n”
- If the length of [key] > 32
  - Responds “INVALID CMD□n”
- Received CREATE msg, but key-value pair already exists?
  - Responds “COLLISION□n”
- Received READ/UPDATE/DELETE msg, but no such key exists?
  - Responds “NOT FOUND□n”
- If any internal error occurs, so cannot serve the request?
  - Responds “INTERNAL ERR□n”

SKVS Semantics

- Global HT should support thread-safe access to both read and write operations

- Use 10 worker threads



Why Static 10 Threads? Any Other Ways for Concurrency?

- What you have learned
  - `accept(), read(), and write()` are blocking..
  - What if dynamically `fork() / pthread_create()` for every clients?

  (+) Easy to implement

  (+) Easy to leverage multiple cores

- Reality..

  (--) Wastes system resources

  (--) Poor performance due to creating processes or threads

  (--) Poor performance due to context switching

  - Using `fork() / pthread_create()` for each request is inefficient (fork()/pthread_create() is heavyweight)

Event-Driven Socket Programming (Out of Scope)

- Set the sockets non-blocking
- Use event-driven API (e.g., epoll)
- Static threads to leverage multiple cores
- Then,
  - (+) Best performance
    - Supports concurrency without context switching
    - Small number of threads
  - (--) Hard to implement using event-driven APIs

➔Workaround: Simply use static threads for this assignment! ☺

SKVS API

- Helper API for parsing and processing the SKVS requests
- `struct skvs_ctx *skvs_init(size_t hash_size, int delay)`
  - Initiates SKVS context
- `void skvs_destroy(struct skvs_ctx *ctx)`
  - Destroys SKVS context
- `const char *skvs_serve(struct skvs_ctx *ctx, char *buffer, size_t len)`
  - Serves an appropriate command for the given SKVS request
  - e.g., [value], “INVALID CMD”, “CREATE OK”, “UPDATE OK”, “COLLISION”, …
  - Notice: You should add a line feed at the end

Client & Server Requirement

- Your client should connect remote server using domain name with arbitrary port
  - e.g., client on sp03.snucse.org → server on sp04.snucse.org:8080

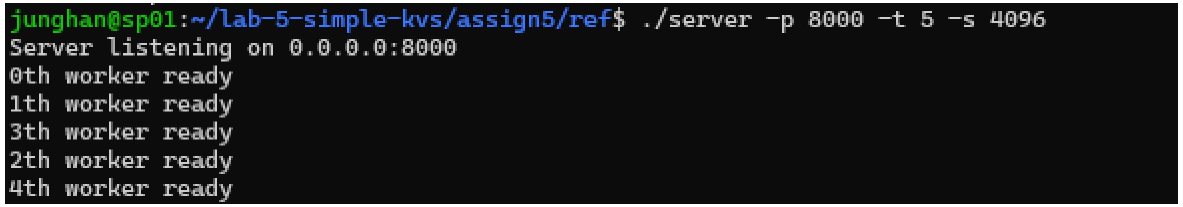
```
./server [-p port (8080)] [-t num_threads (10)] [-d rwlock_delay (0)]
[-s hash_size (1024)]
./client [-i server_ip_or_domain (127.0.0.1)] [-p port (8080)] [-t]
```

  - Your server should serve ~10 concurrent clients
  - Your server and client should interoperate with reference client and server, respectively
- Your server should serve large key-value pair (~32B key, ~4096B SKVS msg)

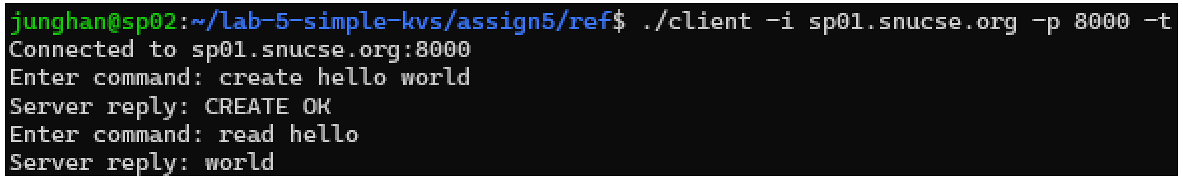
- Reference server/client binaries will be provided
  - No support for Mac, windows
- For usage, refer to the reference server/client

Usage Example for Interactive Mode (client w/ -t)

- Interactive mode for your better understanding and testing
  - Try it on using our reference client
- We will not use this mode for grading
  - No deduction even if your client does not support -t



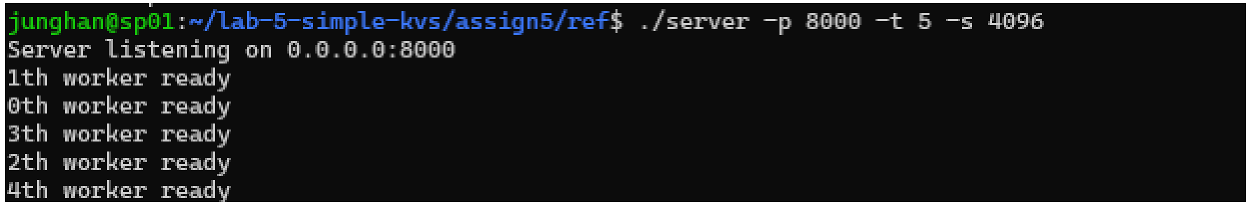
```
junghan@sp01:~/lab-5-simple-kvs/assign5/ref$ ./server -p 8000 -t 5 -s 4096
Server listening on 0.0.0.0:8000
0th worker ready
1th worker ready
3th worker ready
2th worker ready
4th worker ready
```



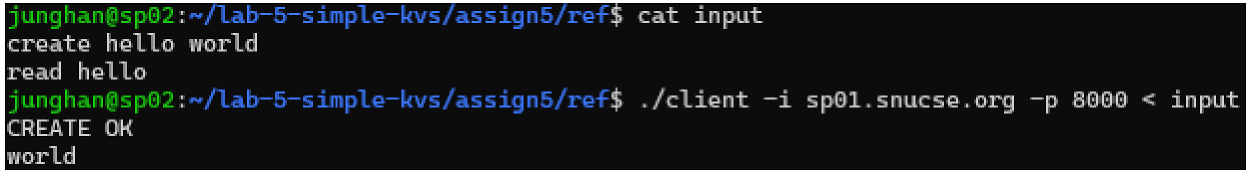
```
junghan@sp02:~/lab-5-simple-kvs/assign5/ref$ ./client -i sp01.snucse.org -p 8000 -t
Connected to sp01.snucse.org:8000
Enter command: create hello world
Server reply: CREATE OK
Enter command: read hello
Server reply: world
```

Usage Example for Silent Mode (client w/o -t)

- We will use this mode for grading
  - On the client, SKVS responses should be printed out to stdout
  - No any other messages allowed on the client



```
junghan@sp01:~/lab-5-simple-kvs/assign5/ref$ ./server -p 8000 -t 5 -s 4096
Server listening on 0.0.0.0:8000
1th worker ready
0th worker ready
3th worker ready
2th worker ready
4th worker ready
```



```
junghan@sp02:~/lab-5-simple-kvs/assign5/ref$ cat input
create hello world
read hello
junghan@sp02:~/lab-5-simple-kvs/assign5/ref$ ./client -i sp01.snucse.org -p 8000 < input
CREATE OK
world
```

Part 2:
Global Hash Table
and
rwlock

hashtable.c

- SKVS library depends on hash table functions
- Hash table functions depend on rwlock functions
- Currently `hashtable.c, rwlock.c` are empty, but you should fill them

- `int hash_insert(hashtable_t *table, char *key, char *value)`
  - Needs a **write lock**
  - Duplicates key and value
  - Fills node structure and inserts node to the table
  - Returns 0 if collision, 1 if inserted, -1 if any internal error
- `int hash_search(hashtable_t *table, char *key, char **value)`
  - Needs a **read lock**
  - Returns 0 if not found, 1 (and outputs to `value`) if found, -1 if any internal error

hashtable.c

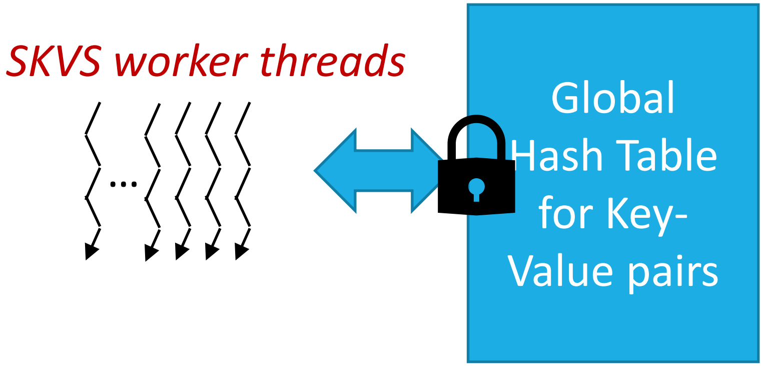
- `int hash_update(hashtable_t *table, char *key, char *value)`
  - Needs a **write lock**
  - Duplicates value
  - Updates node->value to new one
  - Free old value
  - Returns 0 if not found, 1 if updated, -1 if any internal error
- `int hash_delete(hashtable_t *table, char *key)`
  - Needs a **write lock**
  - Frees key and value
  - Evicts from the table
  - Returns 0 if not found, 1 if deleted, -1 if any internal error

Dumping Global Hash Table

- Your SKVS server should dump the hash table using `hash_dump()` before exit on SIGINT
- Register SIGINT handler for dump

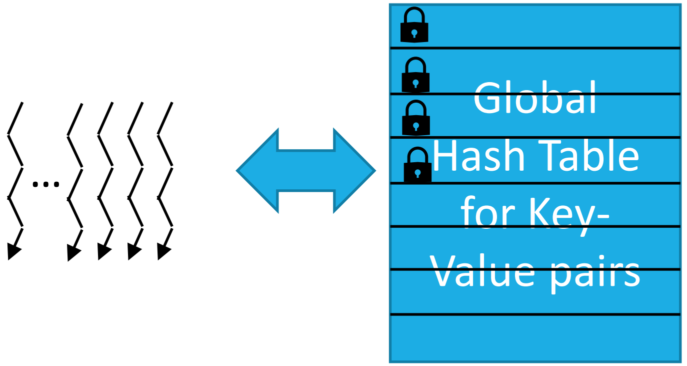
Lock?

- SKVS uses global hashtable
  - Any threads consistently access the key-value pair
  - Accesses to the hashtable needs mutex lock

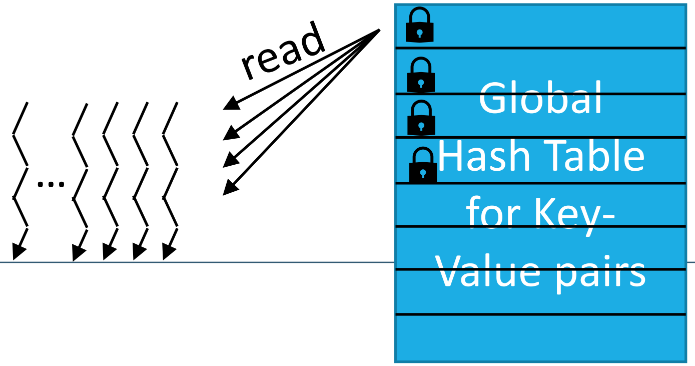


- But, lock contention leads to poor performance..

1. Fine-grained lock
   - **For each bucket**
   - Less contention compared to course-grained lock



2. Reader-writer lock (reader-priority)
   - Assumes many readers, few writers
   - **Multiple readers are allowed at the same time**
   - Wakes up readers first, then the **oldest writer**



rwlock.c

- Reader-priority reader-writer lock
- **Not** allowed to use `pthread_rwlock` API
- Implement your **own** one in `rwlock.c` using `pthread_mutex` and `pthread_cond` API

- `int rwlock_init(rwlock_t *rw, int delay)`
- `int rwlock_read_lock(rwlock_t *rw)`
- `int rwlock_read_unlock(rwlock_t *rw)`
- `int rwlock_write_lock(rwlock_t *rw)`
- `int rwlock_write_unlock(rwlock_t *rw)`
- `int rwlock_destroy(rwlock_t *rw)`

Wake Up using Condition Variables

```
typedef struct {
        int read_count;          // number of current readers
        int write_count;         // number of current writers
        pthread_mutex_t lock;    // mutex lock for protecting other fields
        pthread_cond_t readers;  // condvar for threads waiting read
        pthread_cond_t writers;  // condvar for threads waiting write
        …
} rwlock_t
```

- Example for waking up other threads using condition variables

Thread 1: `pthread_cond_wait(&condvar, &mutex);`

Thread 2: `pthread_cond_signal(&condvar);`

- You should distinguish reader threads and writer threads

`rwlock_read_lock`

```
int rwlock_read_lock(rwlock_t *rw)
```

1. Acquire mutex `lock`
2. Wait if any threads are writing
   - By waiting for reader condvar `readers`

- Wake up!

3. Increment `read_count`
4. Release mutex `lock`

```
typedef struct {
        int read_count;
        int write_count;
        pthread_mutex_t lock;
        pthread_cond_t readers;
        pthread_cond_t writers;
        …
} rwlock_t
```

rwlock_write_lock

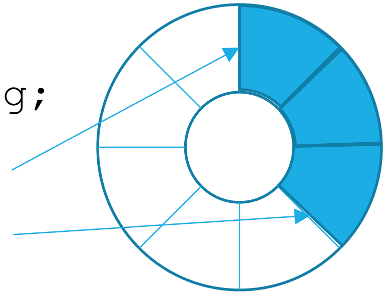
```
int rwlock_write_lock(rwlock_t *rw)
```

1. Acquire mutex `lock`
2. Insert this thread to `writer_ring`
3. Wait if any threads are writing
   - By waiting for writer condvar `writers`

- Wake up!

4. Check if this thread is the oldest
   - If not, wait again
5. Increment `write_count`
6. Evict this thread from `writer_ring`
7. Release mutex `lock`

```
typedef struct {
        int read_count;
        int write_count;
        pthread_mutex_t lock;
        pthread_cond_t readers;
        pthread_cond_t writers;

        // writer ring
        pthread_t *writer_ring;
        int writer_ring_head;
        int writer_ring_tail;
} rwlock_t
```



How to Test Write Lock?

- Before testing, check below
  1. Implement server/client first with **robust read/write**
  2. Your server **must** be ready for concurrency first
  3. `./server -d 5`
     - Add `sleep(5)` in `rwlock_write_unlock()`
     - Add `sleep(5)` in `rwlock_read_unlock()`

- Example test scenario

1. Send "CREATE hello 1□n" on client 1
   - It will hold a write lock for 10 seconds
2. Send "CREATE hello 2□n" on client 2
   - Check whether client 2 gets stuck
3. Send "CREATE hello 3□n" on client 3
   - Check whether client 3 gets stuck
4. Send "CREATE hello 4□n" on client 4
   - Check whether client 4 gets stuck

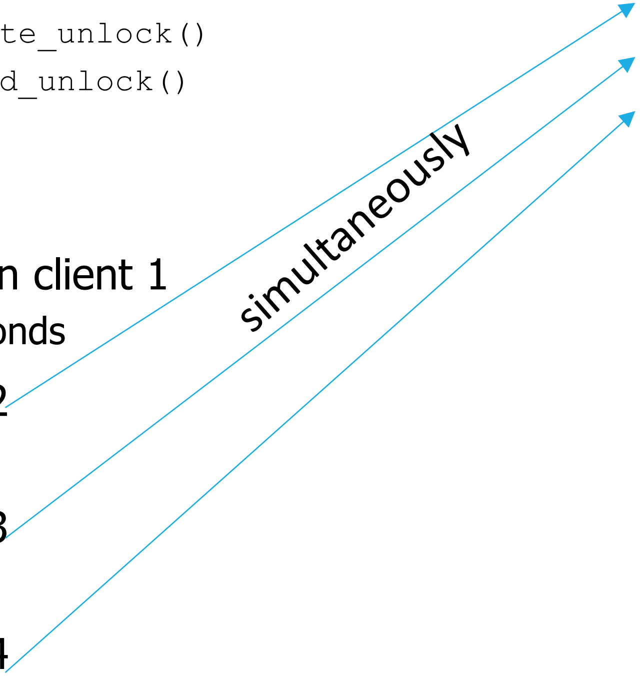
7. After 5 seconds…
   - Check whether client 1 receives "CREATE OK□n"
8. After 5 seconds…
   - Check whether client 2 receives "COLLISION□n"
9. After 5 seconds…
   - Check whether client 3 receives "COLLISION□n"
10. After 5 seconds…
    - Check whether client 4 receives "COLLISION□n"

How to Test Read Lock?

- Before testing, check below
  1. Implement server/client first with **robust read/write**
  2. Your server **must** be ready for concurrency first
  3. `./server -d 5`
     - Add `sleep(5)` in `rwlock_write_unlock()`
     - Add `sleep(5)` in `rwlock_read_unlock()`

- Example test scenario

1. Send “CREATE hello world□n” on client 1
   - It will hold a write lock for 10 seconds
2. Send “READ hello□n” on client 2
   - Check whether client 2 gets stuck
3. Send “READ hello□n” on client 3
   - Check whether client 3 gets stuck
4. Send “READ hello□n” on client 4
   - Check whether client 4 gets stuck



7. After 5 seconds…
   - Check whether client 1 receives “CREATE OK□n”
8. After 5 seconds…
   - Check whether client 2 receives “world□n”
   - Check whether client 3 receives “world□n”
   - Check whether client 4 receives “world□n”

Complex Scenario

- Before testing, check below
  1. Implement server/client first with **robust read/write**
  2. Your server **must** be ready for concurrency first
  3. `./server -d 5`
     - Add `sleep(5)` in `rwlock_write_unlock()`
     - Add `sleep(5)` in `rwlock_read_unlock()`
- Example test scenario

1. Send “CREATE hello world□n” on client 1
   - It will hold a write lock for 10 seconds
2. Send “DELETE bye□n” on client 2
   - Check whether client 2 receives “NOT FOUND□n” immediately (why?)
3. Send “READ hello□n” on client 2
   - Check whether client 2 gets stuck
4. Send “UPDATE hello snu□n” on client 3
   - Check whether client 3 gets stuck
5. Send “DELETE hello□n” on client 4
   - Check whether client 4 gets stuck
6. Send “READ hello□n” on client 5
   - Check whether client 5 gets stuck
7. After 5 seconds…
   - Check whether client 1 receives “CREATE OK□n”
   - Check whether client 2 receives “world□n”
   - Check whether client 5 receives “world□n”
8. After 5 seconds…
   - Check whether client 3 receives “UPDATE OK□n”
9. After 5 seconds…
   - Check whether client 4 receives “DELETE OK□n”

Requirements Summary

- Your client should connect remote server using domain name with arbitrary port
  - e.g., client on sp03.snucse.org → server on sp04.snucse.org:8080

```
./server [-p port (8080)] [-t num_threads (10)] [-d rwlock_delay (0)]
[-s hash_size (1024)]
./client [-i server_ip_or_domain (127.0.0.1)] [-p port (8080)] [-t]
```

  - Your server should serve ~10 concurrent clients
  - Your server and client should interoperate with reference client and server, respectively
- Your server should serve large key-value pair (~32B key, ~4096B SKVS msg)
- Print the entries in the global hash table using `hash_dump()`
  - Do not modify this function
- Concurrent multiple readers' access to the global hash table
- Correct reader-priority RW lock semantic
- Do not modify other files than `client.c, server.c, hashtable.c, and rwlock.c`

Guidelines

Notice

- Do not change the name and the prototype of the skeleton code

- What to submit
  - A tarball named `202412345_assign5` including `server.c client.c hashtable.c rwlock.c.`

```
cd assign5/src
make submit ID=202412345
```

- Replace `202412345` to your student ID without dash

Deadline

- Deadline: ~ 2024. 12. 20 21:00
  - 0 Points if deadline is missed
  - 0 Points for copying
- Contact
  - Lab 5 TA e-mail: cerotyki@snu.ac.kr
  - TA mailing list: snu-sysp@googlegroups.com

Reference Binaries for self-checking

- Reference server/client binaries will be provided
  - No support for Mac, windows
- For any ambiguities, refer to the reference server/client

- You may use `TRACE_PRINT()` and `DEBUG_PRINT()` for debugging

```
CFLAGS += -DTRACE
CFLAGS += -DDEBUG
```

Q&A