CMPUT 379 Lab

ETLC E1003: Tuesday, 5:00 – 7:50 PM.

Tianyu Zhang, Peiran Yao

CAB 311: Thursday, 2:00 – 4:50 PM.

Max Ellis, Aidan Bush

Assignments

- Don’t put your code in a public repo (GitHub has free private repos)
- Make sure to test it on the lab machines
- Remember to name the zip or tar.gz file correctly
  - i.e. Assignment 1: dragonshell.zip or dragonshell.tar.gz
- Remember to name the executable correctly
  - i.e. Assignment 1: dragonshell

Last Week...

- Introduction to networking
- Creating and using sockets
- Getting familiar with TCP and UDP
- A brief look at pthreads
- Race conditions and deadlocks

Today’s Lab

- Review on Threads and Thread safe data structures
- Assignment 2: Threadpool and MapReduce

Threads

- Threads allow concurrent processing with shared memory
- In the next assignment you will have to use pthreads

See also: ***Operating Systems: Three Easy Pieces (Chap. 25-32)****. V1.0.1. Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau.*

Pthreads

- The POSIX thread library
- Compile and link with **-pthread**

Creating Pthreads

*int pthread_create(pthread_t *, pthread_attr_t *, void * (*)(void*), void *)*

- Starts the given function in a thread
- For default attributes attr = NULL

Defining thread start routines

void *(*start_routine) (void *)

- start_routine is a function pointer
- Return value of type **void *** (pointer to any type)
- Single parameter of type **void ***

```
void* thread_function(void* args_p /* the input */) {
    int* num = (int *) args_p; /* cast the input to the type you want */
    /* do something */
    return (void *) ret; /* return the value as void * pointer */
}
```

Defining thread start routines

- Pass the function name when calling **pthread_create**

```
pthread_create(&thread_handle,  NULL, thread_function, NULL);
```

Passing parameters

*int pthread_create(pthread_t *, pthread_attr_t *, void * (*func)(void*), void *)*

- The last parameter of **pthread_create()** is a pointer to the argument
- For example, to pass an integer, take its address as the last argument

```
int num_to_pass = 42;
pthread_create(&thread_handle, NULL, thread_function, &num_to_pass);
```

- To receive it, cast the last argument back to **int *** in the thread routine

```
void* thread_function(void* args_p /* the input */) {
    int* num = (int *) args_p; /* cast the input to the type you want */
    /* do something */
```

Joining Pthreads

*int pthread_join(pthread_t thread, void **value_ptr);*

- Wait until the target thread terminates
- Store the return value to where *value_ptr* points to

Receiving return value

- Values defined in thread routines are stored in the stack, and are destroyed after the thread terminates.
- To persist it, use **malloc / new** to store it in the heap. ⚠️ Remember to free

```
void* thread_function(void* args_p /* the input */) {
    int* num = malloc(sizeof(int));  *num = 42;
    return num;
```

- To receive it

```
    int* p_ret_value;
    pthread_join(thread, &p_ret_value);  /* now p_ret_value points to 42 */
```

Receiving return value

- For small types (size <= sizeof(void *)), you can return it directly

```
void* thread_function(void* args_p /* the input */) {
    int num = 42;
    return (void*) num;
```

- To receive it

```
    int ret;
    pthread_join(thread_handle, (void**)&ret);
    printf("ret: %d\n", ret);
```

Pthread synchronization

- A resource (i.e. variable) shouldn’t be accessed by multiple threads at the same time
- Mutex
- Conditional Variable

Too much milk revisited

Consider that Peter and Greg are roommates

Time	Peter	Greg
3:00	Look in fridge, no milk	
3:05	Leave for store	
3:10	Arrive at store	Look in fridge, no milk
3:15	Buy milk	Leave for store
3:20	Arrive at home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive at home, put milk away (*oof*)

Too much milk revisited

- How to solve this problem talk with your neighbour?
- Share your solutions!

Too much milk revisited

Adding locks!

Time	Peter	Greg
3:00	🔒 Lock the kitchen. Look in fridge, no milk	
3:05	Leave for store	
3:10	Arrive at store	Try to look in fridge, but the kitchen is locked!
3:15	Buy milk	Waiting for Peter 🕰️
3:20	Arrive at home, put milk away 🔓 Unlock the kitchen.	Look in fridge after Peter unlocks the door
3:25		Enjoy milk 🥛!

Pthread mutex

pthread_mutex_t

- Implements a mutually exclusive object and allows locking access to a single thread

- Initialization:
  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER

Pthread mutex locking and unlocking

- Locking
  *int pthread_mutex_lock(pthread_mutex_t *)*

- Unlocking
  *int pthread_mutex_unlock(pthread_mutex_t *)*

- Both return non zero on an error

Exhausted assistant

This time, Peter has an assistant Arthur to help refill milk

Time	Peter	Arthur
3:00		Check the fridge: enough milk
3:05		Check the fridge: enough milk
3:10		Check the fridge: enough milk
3:15	Drink milk 🥛	Check the fridge: no milk
3:20		Buy milk
3:25		Arrive at home, put milk away
3:30		Check the fridge: enough milk

Exhausted assistant

Arthur has a very inefficient way of working. Why?

~~Exhausted~~ Relaxed assistant

Arthur can wait for the fridge to be empty, and Peter can notify Arthur.

Time	Peter	Arthur
3:00		Sleeping 🛌😴
3:05		Sleeping 🛌😴
3:10		Sleeping 🛌😴
3:15	Drink milk 🥛, notify Arthur ⏰	Buy milk
3:20		Arrive at home, put milk away
3:25		Sleeping 🛌😴
3:30		Sleeping 🛌😴

Pthread condition

pthread_cond_t

- Representing a condition to wait for
- Initialization:
  pthread_cond_t cond = PTHREAD_COND_INITIALIZER

Pthread condition - wait

*int pthread_cond_wait(pthread_cond_t *, pthread_mutex_t *)*

- Allows pthreads to wait on for a condition to be set before continuing
- The mutex must be locked before calling
- When it returns the mutex will be locked and owned by that thread

Pthread condition - waiting

```
    pthread_mutex_lock(&kitchen_mutex);
    if (has_milk) {
        pthread_cond_wait(&no_milk_cond, &kitchen_mutex); // automatically
release the lock
    } else {
        buy_milk();
    }
    pthread_mutex_unlock(&kitchen_mutex);
```

Pthread condition - signalling and broadcasting

*int pthread_cond_signal(pthread_cond_t *)*

*int pthread_cond_broadcast(pthread_cond_t *)*

- Signal one or all threads waiting for the condition
- The mutex must be locked before calling

Pthread cheatsheet

	Thread	**Process**
Starting point	Another function	Current line of code
Creation	`pthread_create()`	`fork()`
Get identifier	`pthread_self()`	`getpid()`
Waiting / Joining	`pthread_wait()`	`wait() / waitpid()`
Terminating	`pthread_exit()`	`exit()`

Thread Safe Data Structures

- Avoid writing data at the same time
- Avoid reading data while data is being written
- Lock data when it is being written to
- A straightforward solution: add a global mutex for each data structure
  - Usually, the more you lock, the worse performance you will have

See also: ***Operating Systems: Three Easy Pieces (Chap. 29, 30)****. V1.0.1. Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau.*

Assignment 2

Thread pool

- Solution for managing multiple threads with multiple jobs
- Thread pool organizes its work queue and threads
  - A mechanism is involved to organize the work queue when adding new works
- Each thread in the threadpool continually checks for work
- Concurrency control for accessing the work queue

Thread pool

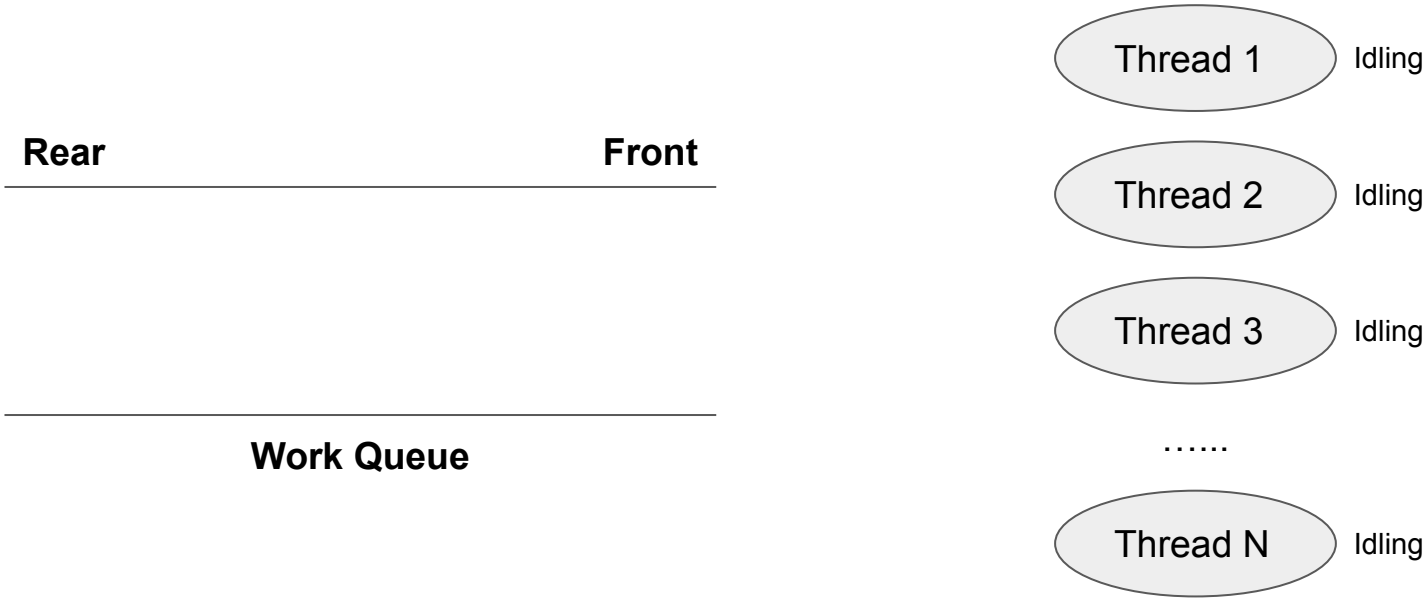
You are implementing a thread pool library, make sure it is general

Your implementation will include following

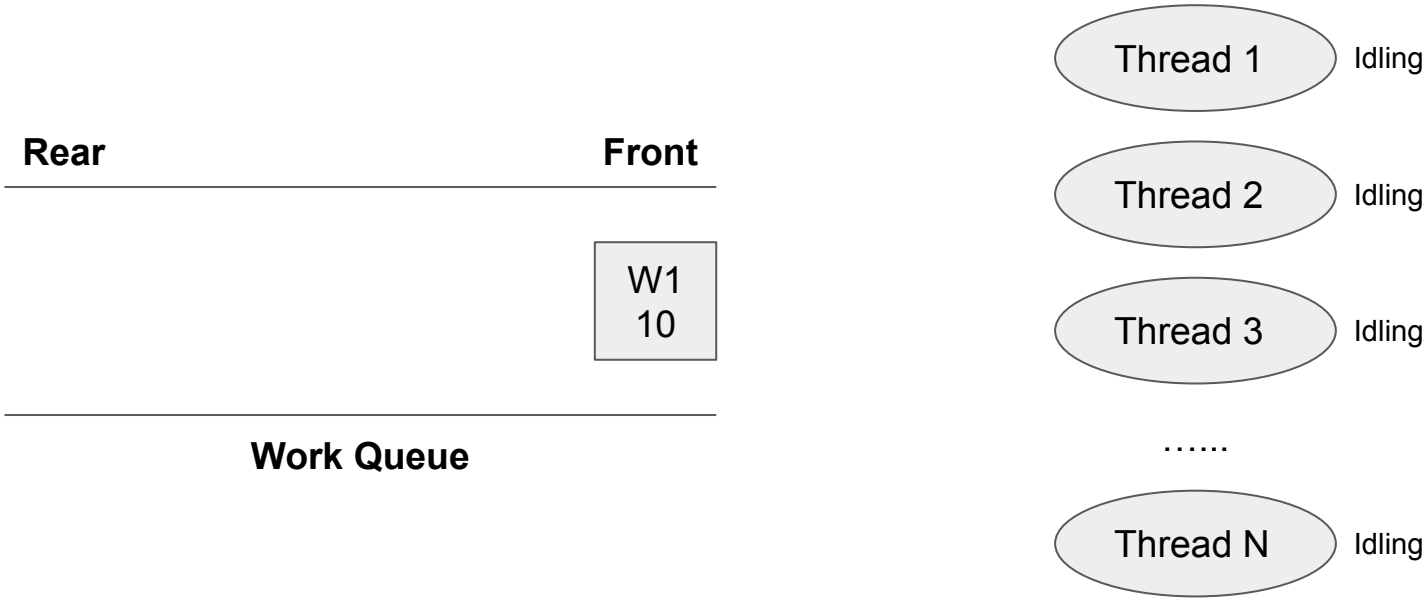
- Data structures for your **thread pool** and your **shared data structure**

- Your thread pool library must include
  - Create / destroy pool
  - In creation take callback functions (function pointers) to process work
  - Have access to the shared data structure

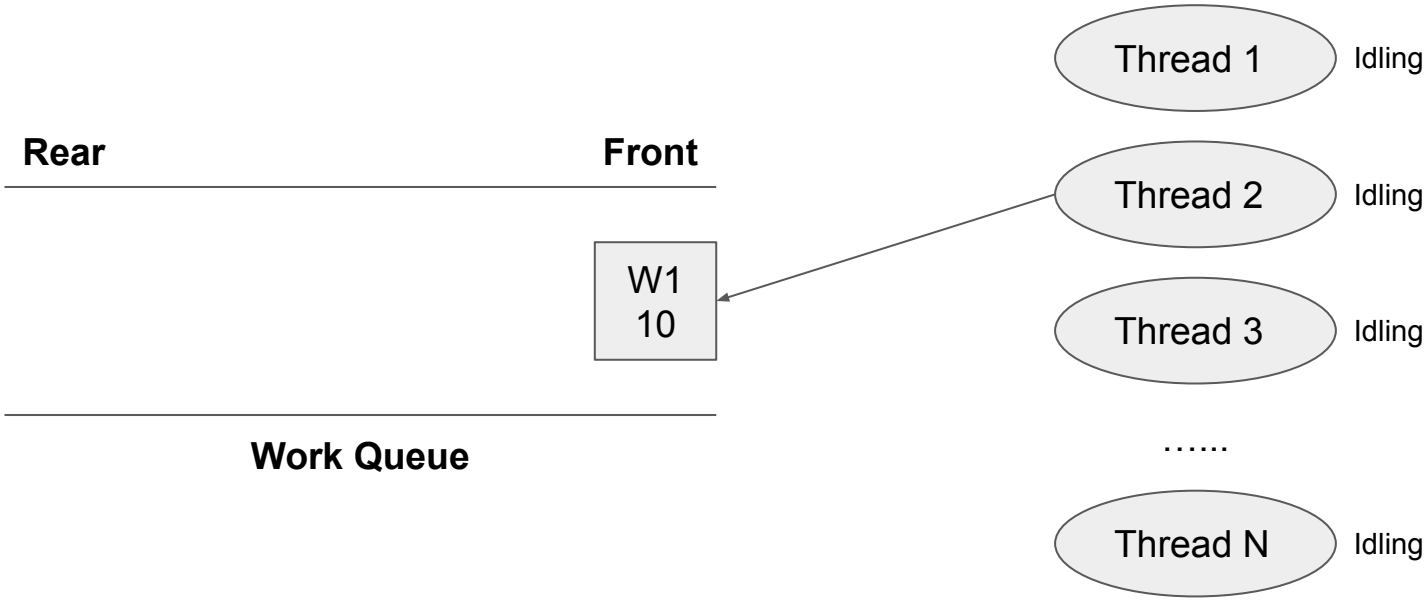
Thread pool - Create



Thread pool - Add works



Thread pool - Add works

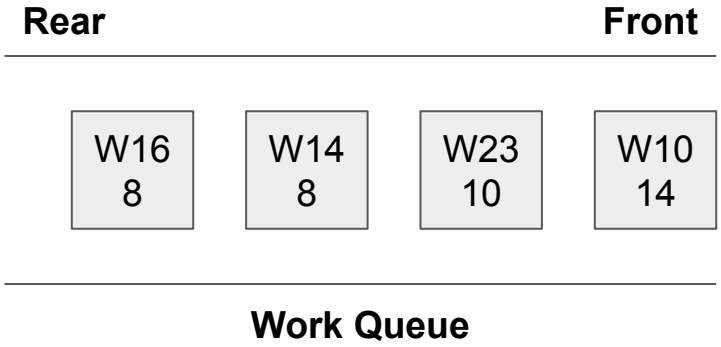


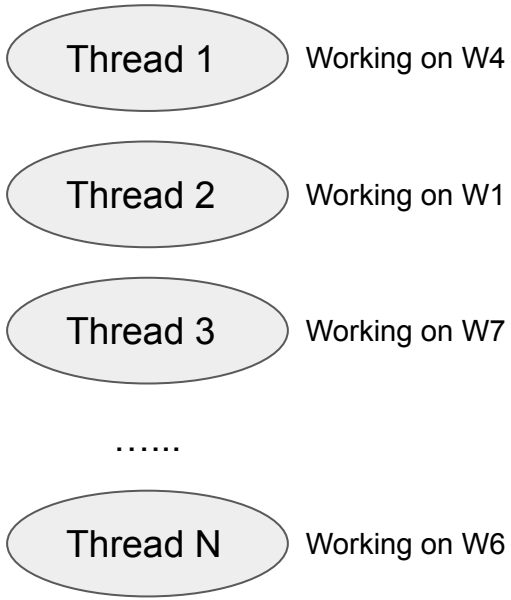
Immediately picked by an idle thread
(Not necessarily the first thread, all threads are running parallel, so you need concurrency control)

Thread pool - Add works

Immediately picked by an idle thread
(Not necessarily the first thread, all threads are running parallel, so you need concurrency control)

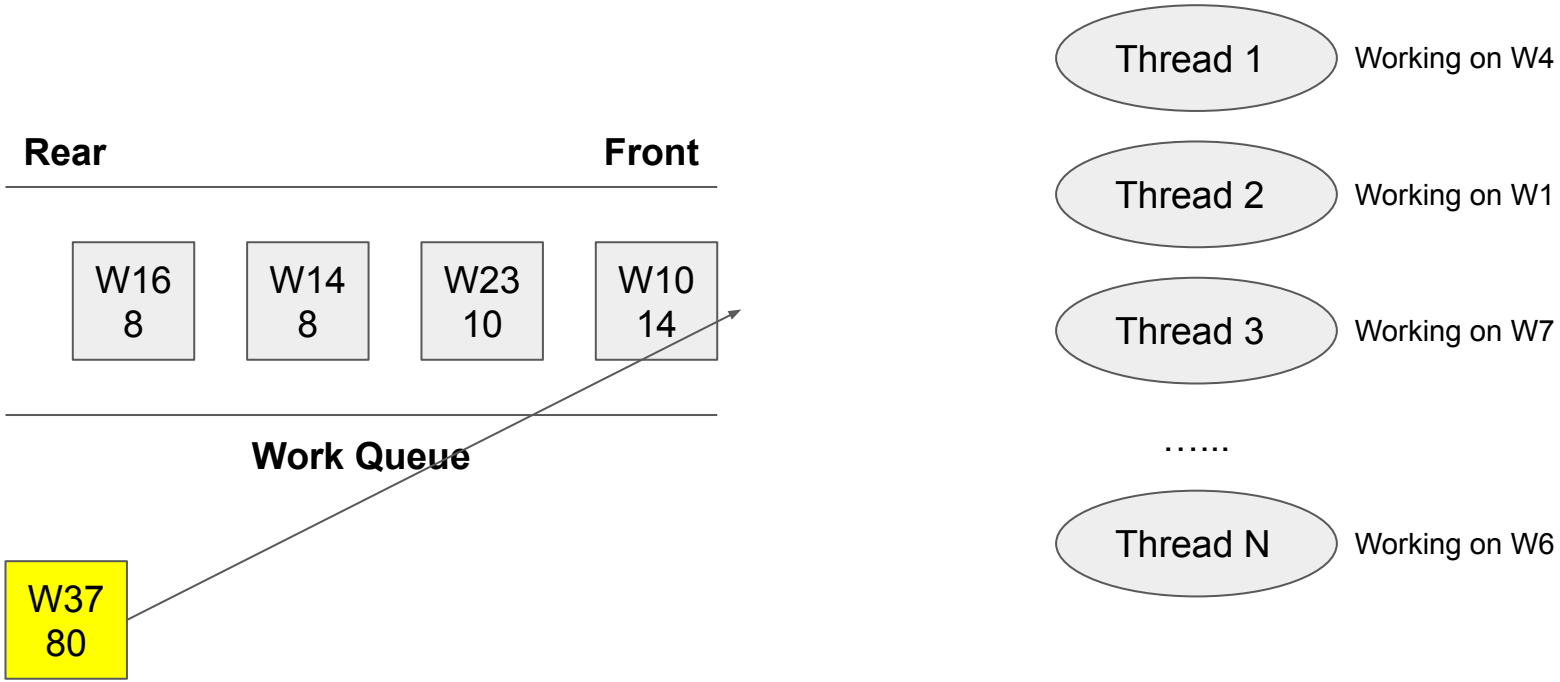
Thread pool - Add works





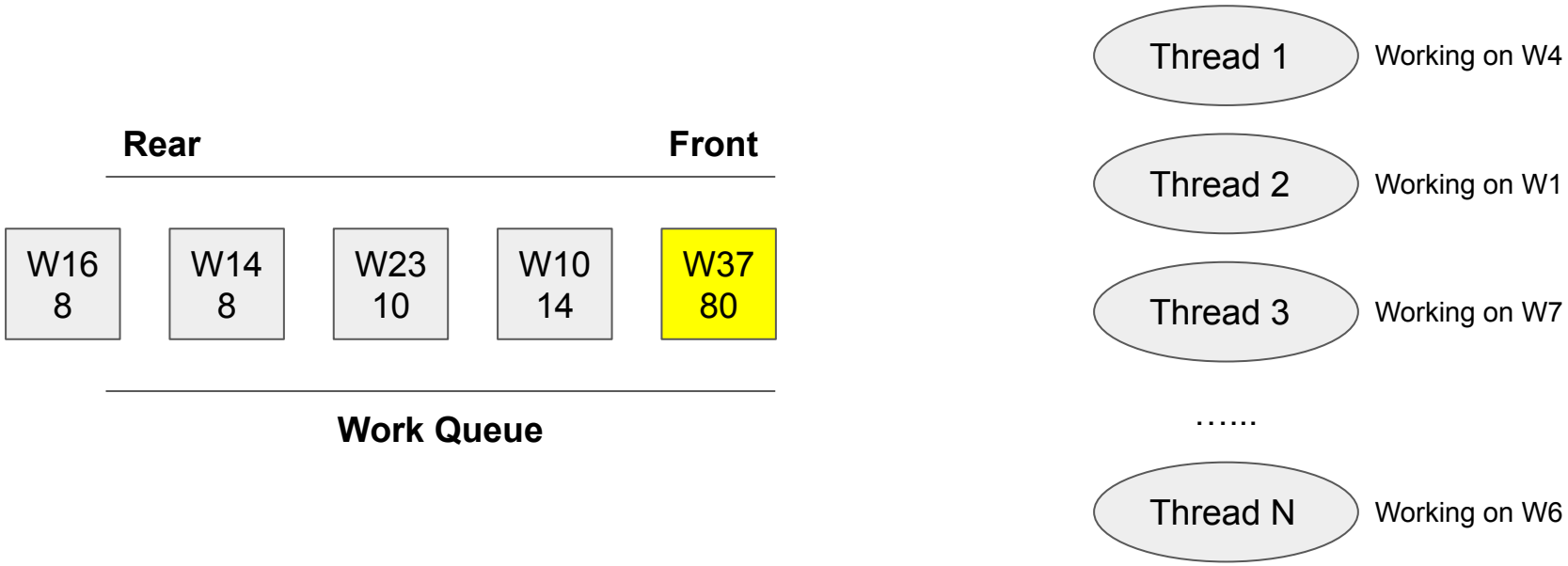
After all threads are working on some jobs, new jobs wait in the queue in an order (LJF)
Break tie by FIFO

Thread pool - Add works



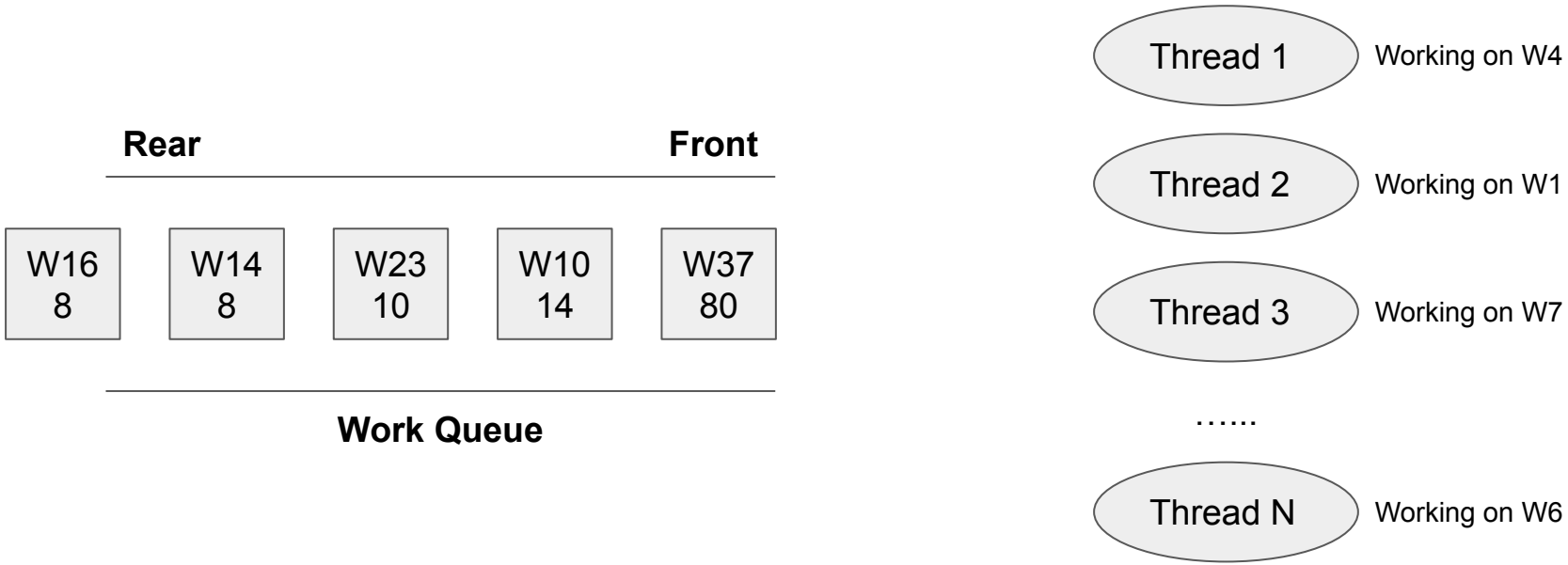
After all threads are working on some jobs, new jobs wait in the queue in an order (LJF)
Break tie by FIFO

Thread pool - Add works



After all threads are working on some jobs, new jobs wait in the queue in an order (LJF)
Break tie by FIFO

Thread pool - Add works



After all threads are working on some jobs, new jobs wait in the queue in an order (LJF)
Break tie by FIFO

MapReduce

- Method for solving problems in a distributed setting

- Components
  - Map
  - Reduce

- You will implement a simplified version of MapReduce library for your next assignment
  - It is a library make sure it will work when we test with different Map and Reduce functions

See also: *Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. Communications of the ACM, 51(1), 107-113.*

MapReduce - Programming model

- Map
  - Transform elements of an array, creating any number of K-V pairs for each element.
  - Applied on different elements in parallel
  - Input type: `list(data)`
  - Output type: `list(k, v)`

- Reduce
  - Take all values for a given key then generate one or more values
  - Applied on different keys in parallel
  - Input type: `(k, list(v1))`
  - Output type: `list(v2)`

MapReduce - Programming model

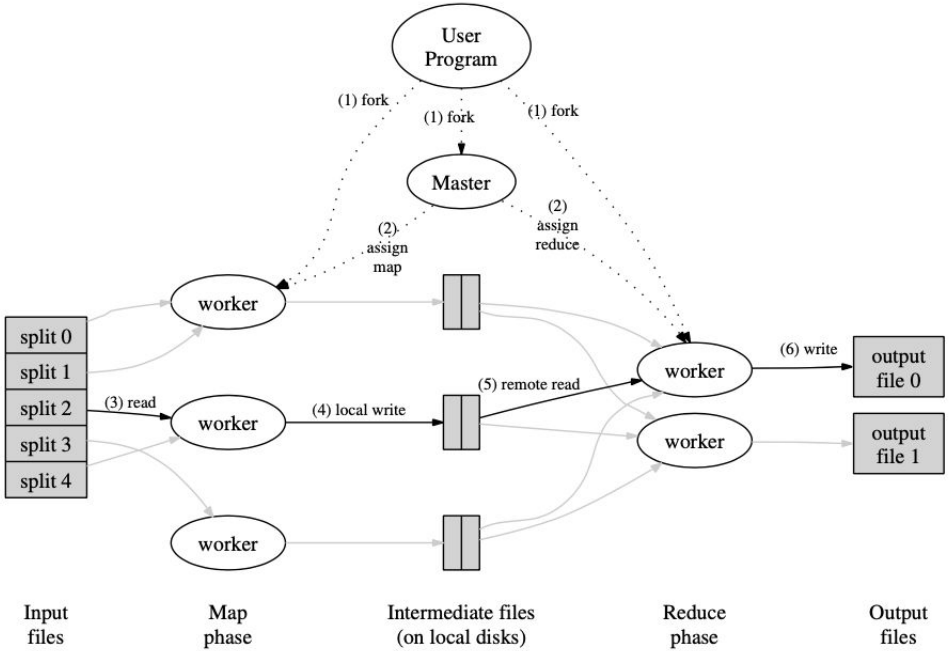
- Example: counting words

```
map(document):
    // document: list of words
    for each word w in value:
        EmitIntermediate(w, 1);

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += v;
    Emit(result);
```

MapReduce - Distributed implementation

- You'll need to implement a slightly different version in Assignment 2



Simplified MapReduce Flow

Master thread manages the solution

1. Creates a thread pool contains M threads
2. Wait for all Map threads to complete
3. Creates R Reduce threads
    - Each thread processes a given partition
4. Wait for all Reduce threads to complete

Map Function

- We provide you an implementation of a map function to convert input file into multiple key/value pairs

- Calls ***MR_Emit*** (you implement) whenever a new key/value pair is generated

- ***MR_Emit(key, value)*** inserts the given key/value pair into the correct partition (using ***MR_Partition***) in your intermediate data structure

Intermediate Data Structure

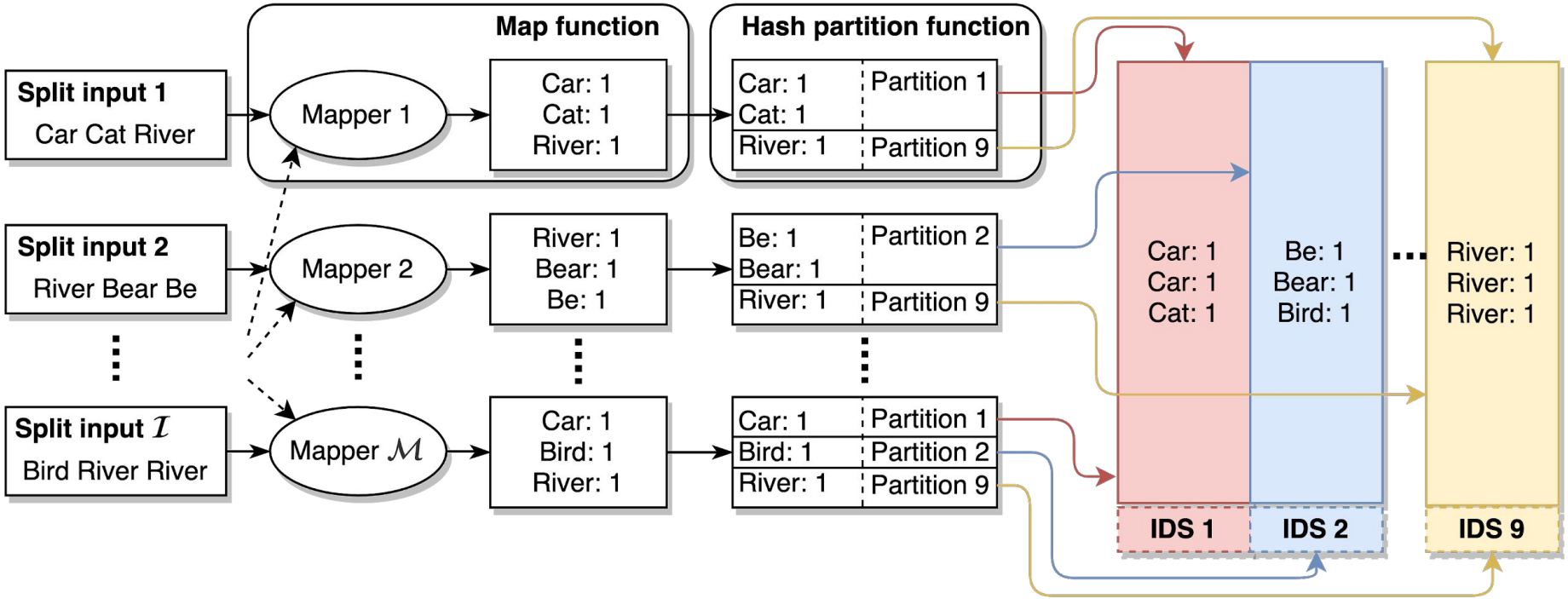
- You need to implement your own intermediate data structure, as long as it can store key/value pairs, it must be your own

- Your intermediate data structure must support R partitions (R is the number of reducers)

Map Threads

- Your thread pool library will allow threads to run jobs concurrently
- The Map function will be continually called in each thread until all files have been processed
- You need to manage access to the list of jobs in your work queue

Map


Map function
Hash partition function
Split input 1
Car Cat River
Mapper 1
Car: 1
Cat: 1
River: 1
Car: 1
Cat: 1
Partition 1
River: 1
Partition 9
Split input 2
River Bear Be
Mapper 2
River: 1
Bear: 1
Be: 1
Be: 1
Bear: 1
Partition 2
River: 1
Partition 9
Split input $\mathcal{I}$
Bird River River
Mapper $\mathcal{M}$
Car: 1
Bird: 1
River: 1
Car: 1
Partition 1
Bird: 1
Partition 2
River: 1
Partition 9
Car: 1
Car: 1
Cat: 1
Be: 1
Bear: 1
Bird: 1
River: 1
River: 1
River: 1
IDS 1
IDS 2
IDS 9

Reduce

- Reducer threads must reduce all keys in their assigned partition
  - You must implement ***MR_ProcessPartition*** (Master Reduce function) to handle the whole partition

- We provide you an implementation of a reduce function that only reduces for a single key
  - It will use ***MR_GetNext*** to retrieve the next value for the given key

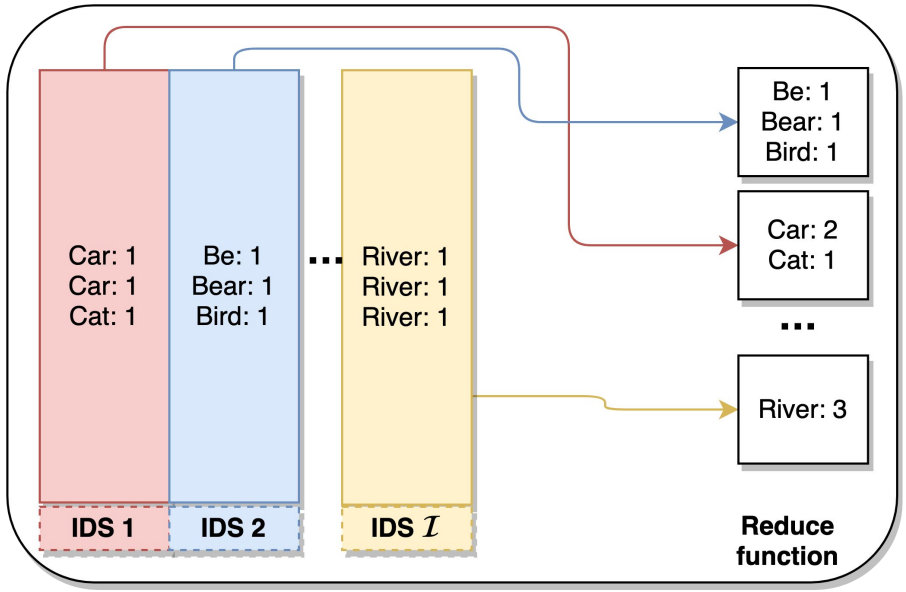
Reduce Threads

- R reduce threads are created, one for each partition
  - Hence there must be R partitions
  - The thread function is *MR_ProcessPartition*

- Reducer will call user-defined reduce function iteratively until all key/value pair in the partition is processed

- The partitions is sorted so that all of the same keys are in order

Reduce

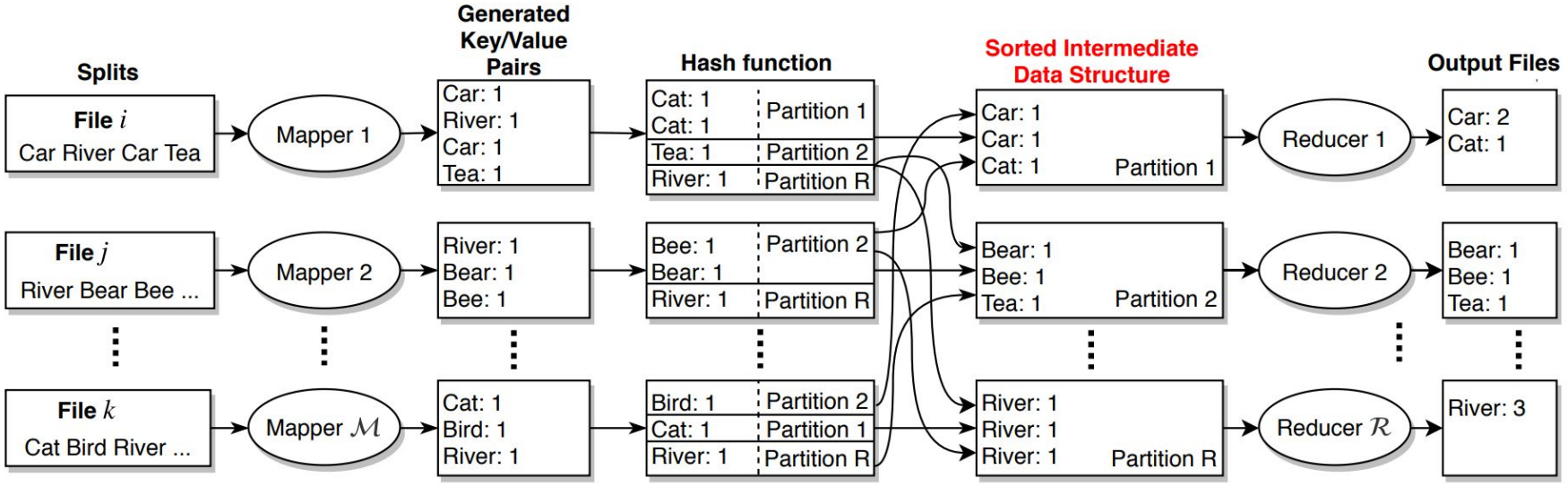

Car: 1
Car: 1
Cat: 1
Be: 1
Bear: 1
Bird: 1
River: 1
River: 1
River: 1
IDS 1
IDS 2
IDS $\mathcal{I}$
Be: 1
Bear: 1
Bird: 1
Car: 2
Cat: 1
River: 3
Reduce function

Intermediate Data Structure

- Must use *pthread_mutex* for locking access
- Must be thread safe
- Must be designed and documented individually

MapReduce Summary

- Given list of files
- Call Map on each file using M threads
- Use a hash function to partition keys into R partitions
- Call Reduce for each resulting key using R threads



MapReduce - examples

- Distributed Grep
  - Find all lines matching the given pattern
  - Map: emits a line if it matches a supplied pattern
  - Reduce: return the same line unchanged

- Inverted Index
  - For all words, list all documents in which a word appeared
  - Map: for each word in a document, emit `<word, document ID>`
  - Reduce: for each word, combine all IDs into a list. Emit `<word, list(Document ID)>`

MapReduce - more examples

- Spark and Hadoop MapReduce are distributed data-processing libraries inspired by MapReduce.
- Used widely, by Amazon, eBay, Facebook, Hulu, Alibaba Taobao, Yahoo, Yandex …..

Practice problem 1

(1) Design a concurrent array data structure that allows multiple threads to access its content without conflicts. Your implementation can

   (a) Have a global mutex (lock) for the entire array
   (b) Have one separate mutex for each element of the array

(2) Create multiple threads that accesses your array concurrently. Measure the performance difference across different designs

Practice problem 2

In Facebook, each user has a list of friends (note that in Facebook, friend is a symmetric relationship). Friends are stored as User -> [List of friends]. For example, user A has friends B C and D, then it will be stored as A -> B C D. Design a pair of map and reduce function that calculates the number of friends in common based on friend lists.

Sample input:

A -> B C D
B -> A C D E
C -> A B D E
D -> A B C E
E -> B C D

Sample output:

(A B) -> (C D)
(A C) -> (B D)
(A D) -> (B C)
(B C) -> (A D E)
…….

Key and credit:
http://steverekrenzel.com/finding-friends-with-mapreduce

Practice problem 3

Create a program that using threads accepts concurrent TCP connections which provide integers separated by new lines. From the connections calculate the average of all numbers across all connections.