CS 110 Final Exam Fall 2019 Key

Problem 1: Short Answer

12 points

This question has five parts. Answer the questions *succinctly as possible*. Answers that are too verbose will lose points. Use your judgement, but be as brief as you need to be to answer each question.

a) Do modern high-concurrency servers generally use event-driven asynchronous I/O or threads with blocking I/O? (Copy one of the statements into your answer). State, in one sentence, one reason why they do. [4 points]

event-driven asynchronous I/O

Threads have too much memory overhead, and event-driven I/O allows the kernel to run a particular function when something happens on a file descriptor, e.g., for a network socket.

b) What is a TPU? What application is it designed for? [2 points]

A TPU is a *Tensor Processing Unit*, and it is designed to efficiently perform AI neural network training.

c) RAID 0 splits (or "stripes") data evenly across two or more disks, without parity information or redundancy. Some have referred to the "0" meaning "how much data you can recover if there is a disk failure." So, this is a large downside to RAID 0. Describe one significant performance benefit of using RAID 0 [2 points]

RAID 0 is useful in situations where better performance is desired, because a computer can access both drives simultaneously. (not part of the answer: you would only use this on disks that don't have volatile data, e.g., an operating system or application disk that can be re-loaded in case of failure).

d) RAID 5 stripes the data across disks and sets aside one disk's worth of storage as parity. The parity for a sector is the XOR of all of each sector on all of the other drives. In other words, the parity for any particular byte address on a sector is the XOR of the bytes on all the other drives at that address. Assume there is a RAID 5 system with 6 disks. At the first byte at a particular sector, assume that the first five disks have actual data, and the 6th holds the parity, as follows:

```
disk:  D1   D2   D3   D4   D5   D6
value: 0x23 0xFE 0x03 0x55 0xB4 0x3F
```

If disk D3 fails, what calculation is performed to recover the 0x03 value at that particular byte on the sector? [2 points]

All bytes other than D3 need to be XOR'd together:

```
0x23 ^ 0xFE ^ 0x55 ^ 0xB4 ^ 0x3F == 0x03
```

e) Because of your mastery of CS110, you were hired at Microgoogbookazonle to work on developing thread-safe containers for their new programming language, C+=2. You create a Vector class that you claim is 100% thread safe for all operations, including **`contains`** and **`add`**. After the language is released, you receive a bug report that your Vector code can't be thread-safe because a user created the following function and claims that their database was corrupted because of a race condition that must be present in the Vector class. Explain to them how to make their access pattern thread-safe.

```
void insertIfAbsent(Vector<user_object> &vector, user_object &object) {
    // Put object into vector if not already in vector
    if (vector.contains(object)) {
        return;
    }
    vector.add(object);
}
```

[2 points]

There is a race condition in the *user's* code — two threads could both check that an object is not already in the Vector and both get to `vector.add(object)` at the same time. Another lock is needed around the entire function.

Problem 2: fork / dup2 / pipe

5 points

After the program on the next page is run:

a) What is output to the screen?

```
starting program
```

b) What is the contents of test.txt?

```
in parent
in child
hello
goodbye
```

Hint: You may want to write down the file descriptor table to track what is happening in the program.

```
int main() {
    int fds[2];
    pipe(fds);

    printf("starting program\n");

    int file_fd = open("test.txt", O_WRONLY | O_TRUNC | O_CREAT, 0666);
    dup2(file_fd, STDOUT_FILENO);
    close(file_fd);
    pid_t pid = fork();

    if (pid == 0) { // child
        char buffer[6];
        close(fds[1]);
        read(fds[0], buffer, sizeof(buffer) - 1);
        buffer[sizeof(buffer) - 1] = '\0';
        printf("in child\n");
        printf("%s\n", buffer);
        close(fds[0]);
        exit(0);
    }

    printf("in parent\n");
    close(fds[0]);
    dprintf(fds[1], "hello");
    waitpid(pid, NULL, 0);

    close(fds[1]);
    printf("goodbye\n");
}
```

Problem 3: One-way Traffic

10 points

In Hawaii, there are a number of roads with one-way bridges. On such bridges, traffic can back up in both directions (say, *eastward*, and *westward*). If there are cars going in a particular direction on the bridge, all other cars that arrive at the bridge going in that direction tailgate behind the car already on the bridge, forcing all cars going in the opposite direction to wait until there are 0 cars on the bridge. Indeed, this could last forever, if cars kept coming in the same direction (this is called *starvation*). In other words, as long as a car is on the bridge going in the same direction, an arriving car in that direction will be the next to go. The moment that the bridge is empty and there are no more cars going in the opposite direction, the cars that have been waiting can finally go. Then, any car coming in the opposite direction has to wait for all the cars that were waiting (and any that arrive) leave the bridge.
In this problem, we will model this one-way-bridge situation using threads, a single mutex, and a single condition variable. Here are the global variables (yes, we're using a bunch of global variables) for the problem:

```
const int EASTBOUND = 0;
const int WESTBOUND = 1;

bool emptyRoad = true;
int waiters[2] = {0, 0}; // eastbound, westbound
int dir = 0;
int cars = 0;
mutex roadLock;
condition_variable_any cv;
```

Every car is in its own thread, and cars simply **`arrive`** at and **`depart`** from the bridge:

```
void car(int direction, int id) {
    arrive(direction, id);
    depart(direction, id);
}
```

There is a **`populateQueuesWithCars()`** function that randomly sends cars in both directions to the bridge by creating a **`car`** thread (code not shown, and you do not need to use this function at all).

For this problem, complete the **`arrive`** and **`depart`** functions so that the simulation works.
Notes:

- Only one car can actually be on the bridge at a given time, so it must start on the bridge and finish on the bridge before another car can come onto the bridge
- But, if a car is on the bridge, and if cars are waiting in both directions, the next car must be from the same direction.
- There is no other ordering -- you do not need to worry about queueing particular cars before the bridge -- you can think of it as a free-for-all while waiting on a particular side.
- You do not need to worry about thread mechanics (e.g., no need to **`.join`** any threads) -- just get the details for **`arrive`** and **`depart`** correct.

CS 110 Final Exam Fall 2019 Key

(Note: there are multiple different, correct solutions to this problem)

```
void arrive(int myway, int id) {
    int otherway;
    if (myway == EASTBOUND) {
        otherway = WESTBOUND;
    } else {
        otherway = EASTBOUND;
    }
    // TODO: Finish arrive function
```

```
lock_guard<mutex> lg(roadLock);
waiters[myway]++;
cv.wait(roadLock, [this]() {
  return emptyRoad && (waiters[otherWay] == 0 || dir == myway);
});
emptyRoad = false;
dir = myway;
waiters[myWay]--;
```

```
}
```

```
void depart(int myway, int id) {
    int otherway;
    if (myway == EASTBOUND) {
        otherway = WESTBOUND;
    } else {
        otherway = EASTBOUND;
    }
    // TODO: Finish depart function
```

```
lock_guard<mutex> lg(roadLock);
emptyRoad = true;
cv.notify_all();
```

```
}
```

Problem 4: Lock Free Data Structures

12 points

This question has four parts.

a) In one succinct sentence, define atomicity. [2 points]

Atomicity means that a series of operations will appear to have not yet executed or have executed completely — other computations or threads cannot see intermediate state during execution.

b) A mailbox is a data sharing abstraction used in many low-level systems. A mailbox has one slot. A writer can put data into the mailbox and a reader can read data out of the mailbox. In our mailbox implementation, if a writer tries to write to a full mailbox, it will enter a spin loop until the mailbox is empty. Similarly, if a reader tries to read from an empty mailbox, it will enter a spin loop until the mailbox is full.
In most modern processors, each instruction is atomic. This is true for memory loads and stores. Race conditions can occur due to the timing of sequences of instructions, but a write or read of memory is an atomic operation.

Please look at the following implementations of **`mailbox_put`** and **`mailbox_get`**. You can assume there is a single reader (calling **`mailbox_get`**) and a single writer (calling **`mailbox_put`**).

```
typedef struct mailbox {
  int value;
} mailbox_t;

void mailbox_put(mailbox_t* mailbox, int value) {
    while (mailbox->value) {} // A memory load
    mailbox->value = value;   // A memory store
}
```

```
int mailbox_get(mailbox_t* mailbox) {
    while (mailbox->value == 0) {} // A memory load
    int val = mailbox->value;      // A memory load
    mailbox->value = 0;            // A memory store
    return val;
}
```

Are there any race conditions between **`mailbox_get`** and **`mailbox_put`**? (Put the correct answer into your response) Yes, there are potential race conditions when **`mailbox_get`** and **`mailbox_put`** run concurrently. No, there are no race conditions when **`mailbox_get`** and **`mailbox_put`** run concurrently. In 1-2 sentences, explain your answer. If your answer is yes, write an interleaving of the two that causes the race condition. [4 points]

There are no race conditions. There is a single writer at any time. Whether value stores something controls which function can write to it, and once a function writes it can't write again until the other function writes.

Also correct: There isn't a race condition, but there is a data consistency problem if put is called with value = 0; this put will be ignored.

c) A generalization of a mailbox is a *circular buffer*. In this abstraction, there can be more than one data item in the mailbox. The buffer is a fixed size. As before, if a writer tries to write to a full mailbox, it will enter a spin loop until there is space. Similarly, if a reader tries to read from an empty mailbox, it will enter a spin loop until the mailbox has data.
Such a queue maintains two variables: the index of the head of the queue (the next element to get) and the index of the tail of the queue (the place to put the next item). When head == tail, the queue is empty. When the tail is immediately behind the head, the queue is full. The code looks something like this:

```
#define CQ_LEN 64
typedef struct cqueue {
    char buffer[CQ_LEN];
    size_t head;
    size_t tail;
}

void cqueue_init(cqueue* q) {
    q->head = 0;
    q->tail = 0;
}

bool cqueue_full(cqueue* q) {
    int next = (q->tail + 1) % CQ_LEN;
    return next == c->head;
}

bool queue_empty(cqueue* q) {
    return (c->tail == c->head);
}
```

It turns out it is possible to implement this queue such that it is like a correctly implemented one element mailbox above: it allows one reader and one writer to run concurrently with no race conditions. Fill in the following two functions so they do not have any race conditions. Be sure to use `%` to wrap around when the indexes go past **`CQ_LEN`**.

[4 points]

```
void cqueue_put(cqueue* q, char c) {
```

```
while (cqueue_full(q)) {}
q->buffer[q->tail] = c;
q->tail = (q->tail + 1) % CQ_LEN
```

```
}
```

```
char cqueue_get(cqueue* q) {
```

```
while (cqueue_empty(q)) {}
char c = q->buffer[q->head];
q->head = (q->head + 1) % CQ_LEN;
return c;
```

```
}
```

d) What about this abstraction allows it to be safely implemented without any synchronization mechanisms like mutexes? Write 1-2 succinct sentences explaining why. [2 points]

Each index always has a single writer, head for get and tail for put, and. These indexes control where in the array can be written, so at any moment any given array cell has at most one writer.

Problem 5: Parallel Search

10 points

Below is the implementation for a simple linear search that finds all indexes where **`strToFind`** can be found in **`fullStr`**.

```
bool strAtOffset(size_t offset, const string &strToFind,
                 const string &fullStr) {
    // Returns true if strToFind is found at fullStr[offset],
    // false otherwise
    if (offset + strToFind.size() > fullStr.size()) return false;
    for (size_t j = 0; j < strToFind.size(); j++) {
        if (fullStr[offset + j] != strToFind[j]) {
            return false;
        }
    }
    return true;
}

void singleThreadSearch(const string &strToFind, const string
                        &fullStr, vector<int> &indexes) {
    /**
     * Searches for strToFind inside fullStr, and populates the
     * indexes vector with the resulting indexes where strToFind
     * can be found in fullStr.
     */
    for (size_t i = 0; i < fullStr.size(); i++) {
        if (strAtOffset(i, strToFind, fullStr)) {
            indexes.push_back(i);
        }
    }
}
```

For large enough strings, and with a processor with many cores, the search can potentially be sped up by parallelizing the problem with threads. Using the C++ **`thread`** library, write the **`parallelSearch()`** function that can directly replace the **`singleThreadSearch()`** function. Also write the helper function, **`parallelSearchThread()`** (with your defined parameters) to serve as the function that will run in each thread. You should use **`strAtOffset()`** in your implementation.

Your vector should be sorted after the function, and we have provided the **`sort`** function for you at the end of the **`parallelSearch()`** function.

Ensure that your code does not have any race conditions.

~~void parallelSearchThread(~~
~~{~~
~~// TODO: Your code here (and add parameters above)~~

```
void parallelSearchThread(const string &strToFind, const
                          string &fullStr, vector<int>
                          &indexes, size_t start, size_t end,
                          mutex &indexesMutex)

{
    // TODO: Your code here (and add parameters above)
    for (size_t i = start; i < end; i++) {
        if (strAtOffset(i, strToFind, fullStr)) {
            indexesMutex.lock();
            indexes.push_back(i);
            indexesMutex.unlock();
        }
    }

}
```

~~}~~

CS 110 Final Exam Fall 2019 Key

```
const int kMaxThreads = 24;
void parallelSearch(const string &strToFind, const string &fullStr,
                    vector &indexes) {
    /**
     * Searches for strToFind inside fullStr, and populates the
     * indexes vector with the resulting indexes where strToFind can
     * be found in fullStr.
     * Uses up to kMaxThreads to complete the search.
     */

     // TODO: Your code here
```

```
    // using c++ threads (ThreadPool next page)
    mutex indexesMutex;
    vector<thread> threads;

    size_t numPerSearch = fullStr.size() / kMaxThreads;
    size_t extra = fullStr.size() % kMaxThreads;
    if (extra != 0) {
        numPerSearch++;
    }

    for (size_t i = 0; i < kMaxThreads; i++) {
        size_t start = i * numPerSearch;
        size_t end = (i + 1) * numPerSearch;
        if (end > fullStr.size() - 1) {
            end = fullStr.size() - 1;
        }
       threads.push_back(thread(parallelSearchThread,
                           ref(strToFind), ref(fullStr),
                           ref(indexes), start, end,
                           ref(indexesMutex)));
    }
    for (thread &t : threads) {
        t.join();
    }
```

```
    sort(indexes.begin(), indexes.end());
}
```

CS 110 Final Exam Fall 2019 Key

```
const int kMaxThreads = 24;
void parallelSearch(const string &strToFind, const string &fullStr,
                    vector &indexes) {
    /**
     * Searches for strToFind inside fullStr, and populates the
     * indexes vector with the resulting indexes where strToFind can
     * be found in fullStr.
     * Uses up to kMaxThreads to complete the search.
     */

     // TODO: Your code here
```

```
    // using ThreadPool
    ThreadPool pool(kMaxThreads);
    mutex indexesMutex;
    size_t numPerSearch = fullStr.size() / kMaxThreads;
    size_t extra = fullStr.size() % kMaxThreads;
    if (extra != 0) {
        numPerSearch++;
    }
    for (size_t i = 0; i < kMaxThreads; i++) {
        size_t start = i * numPerSearch;
        size_t end = (i + 1) * numPerSearch;
        if (end > fullStr.size() - 1) {
            end = fullStr.size() - 1;
        }
        pool.schedule([&strToFind, &fullStr, &indexes,
start, end, &indexesMutex]() {
            parallelSearchThread(strToFind, fullStr,
indexes, start, end, indexesMutex);
        });
    }
    pool.wait();
```

```
    sort(indexes.begin(), indexes.end());
}
```

CS 110 Final Exam Fall 2019 Key

Relevant Prototypes

```
// filesystem access
int close(int fd); // ignore retval
int dup(int fd);   // ignore retval
int dup2(int oldfd, int newfd); // ignore retval
int pipe(int fds[]); // ignore retval
int pipe2(int fds[], int flags); // ignore retval, flags typically O_CLOEXEC
#define STDIN_FILENO 0
#define STDOUT_FILENO 1

// exceptional control flow and multiprocessing
pid_t fork();
pid_t waitpid(pid_t pid, int *status, int flags);
int execvp(const char *path, char *argv[]); // ignore retval
int kill(pid_t pid, int signal); // ignore retval
typedef void (*sighandler_t)(int sig);
sighandler_t signal(int signum, sighandler_t handler); // ignore retval
int sigemptyset(sigset_t *set); // ignore retval
int sigaddset(sigset_t *set, int sig); // ignore retval
int sigprocmask(int how, const sigset_t *set, sigset_t *old);

#define WIFEXITED(status)   // macro
#define WIFSTOPPED(status)  // macro
```

```
class mutex {
public:
   void lock();
   void unlock();
};

class semaphore {
public:
   semaphore(int count = 0);
   void wait();
   void signal();
};
```

```
class condition_variable_any {
public:
   void wait(mutex& m);
   template <typename Pred>
   void wait(mutex& m, Pred p);
   void notify_one();
   void notify_all();
};
```

```
class ThreadPool {
public:
   ThreadPool(size_t numThreads);
   void schedule(Thunk t);
   void wait();
};
```

CS 110 Final Exam Fall 2019 Key

```
struct subprocess_t {
 pid_t pid;
 int supplyfd;
 int ingestfd;
};

subprocess_t subprocess(char *argv[],
      bool supplyChildInput,
      bool ingestChildOutput);

template <typename T>
class vector {
public:
   size_t size() const;
      void push_back(const T& elem);
      T& operator[](size_t i);
      const T& operator[](size_t i) const;
};

template <typename T>
class list {
public:
bool empty() const;
size_t size() const;
void push_back(const T& elem);
T& front();
      void pop_front();
};

template <typename U, typename V>
struct pair {
   U first;
   V second;
};

Template <typename Key, typename Value>
class map {
public:
   // iter points to pair<Key, Value>
   size_t size() const;
   iter find(const Key& k);
   Value& operator[](const Key& k);
};
```