CMPS 405
Exam 2
Fall 2016
November 30th, 2016

Name:

Student ID Number:

Section:

Instructor:

Instructions:

- **Read carefully** through the entire exam first, and plan your time accordingly. Note the relative weights of each segment.
- This exam is closed book, closed notes. You may not refer to any other books or materials during the exam. No electronic devices are allowed.
- Write your answers on this exam. **Do not write on the back of any pages.** If you need additional space, use the extra pages at the end of the exam. Anything written on the backside of a page will not be graded.
- When answering questions that request an explanation, **keep your explanations short and correct**. Explanations containing incorrect information will be marked wrong, even if correct information is also included.
- When solving programming questions, **do not write down `imports` or `#includes`**.
- When you are done, present your completed exam to the instructor at the head table. If leaving before the exam period is concluded, please leave as quietly as possible as a courtesy to your neighbors.

Grading:

Page	Points	Score
1	15	
3	10	
4	25	
6	25	
8	25	
Total:	100	

1. Short answer

   (a) (5 points) What does it mean for an operation to be atomic?

   Solution: The entire operation executes without being interrupted.

   (b) (5 points) What is busy-waiting and why is it generally considered bad?

   Solution: Busy waiting is when a process waits using a loop that constantly checks for a state change in a variable before proceeding. It is bad because that process is taking CPU time just to wait.

   (c) (5 points) What is one advantage of using semaphores instead of Java's `wait/notify/notifyAll` primitives?

   Solution: This is a subjective question with many possible answers. One possible answer is that semaphores are a simpler mechanism and the programmer is less likely to make a mistake.

2. Dining Philosophers

Consider the solution below for the dining philosophers problem:

```java
public class Philosopher extends Thread {
  private int id;
  private Semaphore[] chopstick;
  private static final int NAP_TIME = 5;
  private static final int ROUNDS = 3;

  public Philosopher(int id, Semaphore[] chopstick) {
    this.id = id;
    this.chopstick = chopstick;
  }

  public void run() {
    for (int i = 0; i < ROUNDS; i++) {
      try {
        chopstick[id].acquire();
        chopstick[(id + 1) % 5].acquire();
        eat();
      } catch (InterruptedException e) {
        e.printStackTrace();
      } finally {
        chopstick[id].release();
        chopstick[(id + 1) % 5].release();
      }
      think();
    }
  }

  private void eat() {
    System.out.println("Philosopher " + id + " eating");
    int sleeptime = (int) (NAP_TIME * Math.random());
    try {
      Thread.sleep(sleeptime * 1000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }

  private void think() {
    System.out.println("Philosopher " + id + " thinking");
    int sleeptime = (int) (NAP_TIME * Math.random());
    try {
      Thread.sleep(sleeptime * 1000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

```java
public class Dining {
  Semaphore chopstick[] = new Semaphore[5];

  public Dining(){
    int i;
    for(i=0; i<5; i++)
      chopstick[i] = new Semaphore(1);
    for(i=0; i<5; i++)
      (new Philosopher(i,chopstick)).start();
  }

  public static void main(String[] args) {
    new Dining();
  }
}
```

(a) (5 points) This solution has a synchronization problem. Identify the problem and describe a specific scenario when the problem will occur.

> **Solution:** The program could get stuck (deadlock). As an example, imagine philosopher 1 wakes up, acquires one chopstick, then context switch. Philosopher 2 acquires one chopstick, then context switch. Continue this for all philosophers. Once each philosopher has one chopstick, then philosopher 1 wakes up again and tryies to acquire the other chopstick, which isn't available so he is forced to wait. The other philosophers do the same, and now everyone is waiting and no one can proceed.

(b) (5 points) Briefly describe how you could fix this problem. (No need to write code, an explanation is sufficient.)

> **Solution:** There are a lot of ways. The lazy way is to simply add a lock around the entire process of acquiring chopsticks and eating. (Basically, ensure only one philosopher can eat at a time.) This is sloppy, but works. A better solution is to add a state variable that keeps track of what each philosopher is doing, that way a philosopher can check the status of his neighbors before trying to acquire the chopsticks. If his neighbors are eating, then he needs to wait.

3. (25 points) Multithreaded Server

Consider the following source code for a simple time of day server:

```
public class DaytimeServer {
  public static void main(String[] args) throws Exception {
    ServerSocket serverSocket = new ServerSocket(13000);

    while (true) {
      Socket clientSocket = serverSocket.accept();
      PrintWriter writer = new PrintWriter(clientSocket.getOutputStream(), true);

      while(true) {
        String timeStamp = new SimpleDateFormat("yyyy.MM.dd.HH.mm.ss")
                           .format(new Date());
        writer.println(timeStamp);
        if (writer.checkError()) {
          // We couldn't write, meaning the socket is disconnected.
          break;
        }
        Thread.sleep(5000);
      }
      writer.close();
      clientSocket.close();
    }
  }
}
```

When a client connects to this server, the server sends the current date and time to the client every 5 seconds. The server, as given, can only handle one client at a time. Write a multithreaded version of this server that can handle multiple clients at the same time. You may ignore exception handling in your solution.

Solution:

```
class ClientHandler extends Thread {            +3 Defines a class that extends Thread
  Socket clientSocket;                          +1 Way to store client socket

  public ClientHandler(Socket clientSocket) {   +1 Constructor
    this.clientSocket = clientSocket;           +1 Handling argumnet
  }

  public void run() {                           +2 run method with correct prototype
    PrintWriter writer =                        +1 This line
          new PrintWriter(clientSocket.getOutputStream(), true);
    while (true) {                              +1 Infinite loop
      String timeStamp =
          new SimpleDateFormat("yyyy.MM.dd.HH.mm.ss").format(new Date());
      writer.println(timeStamp);                +1 Forms data string and sends it
      if (writer.checkError()) {                +1 Proper error checking with break
        // We couldn't write, meaning the socket is disconnected.
        break;
      }
      Thread.sleep(5000);                       +1 Sleeps properly
    }
    writer.close();                             +1 Closes writer
    clientSocket.close();                       +1 Closes socket
  }
}

public class DaytimeServerThreaded {            +2 Defines a class that has the main

  public static void main(String[] args) throws Exception {   +1 Defines main
    ServerSocket serverSocket = new ServerSocket(13000);
                                                +1 Creates correct server socket

    while (true) {                              +1 Loop accepting clients
      Socket clientSocket = serverSocket.accept();   +1 Accepts client
      ClientHandler ch = new ClientHandler(clientSocket);
                                                +2 Creates an instance of the thread enabled object
      ch.start();                               +2 Starts the thread
    }
  }
}
```

4. (25 points) PThreads

   Write a C program that uses threads to sum two different arrays, prints the sum of each array, and then prints the total sum of all the numbers in both arrays. The arrays contain only positive numbers, except the last entry in the array, which is 0. You must use a separate thread to sum each array. For example, if the program is compiled with these two arrays:

```
int arr1[] = { 1, 54, 76, 53, 24, 0 };
int arr2[] = { 4, 24, 32, 43, 25, 0 };
```

   Then the output is...

```
Sum is 208
Sum is 128
The total sum is 336
```

   Use the skeleton code below to get started:

```
int main(void)
{
  int arr1[] = { 1, 54, 76, 53, 24, 0 };
  int arr2[] = { 4, 24, 32, 43, 25, 0 };
  long sum = 0;

  // Your code here.
  // You will also need to write a helper function for the threading.
```

Solution:

```
void *calc_sum(void *arg)                     +2 Has a separate function to calculate the sum
                                              +1 correct prototype for that function
{
  long sum = 0;
  int i;
  int *arr = (int *) arg;     +2 Has a correct method of getting access to the array to be summed

  for (i = 0; arr[i] > 0; i++) {              +3 Has a loop to sum the array
    sum += arr[i];              +1 The loop stops properly and doesn't assume a fixed size
  }

  printf("Sum is %ld\n", sum);                +2 individual array sum is printed
  pthread_exit((void *) sum);                 +2 Valid method of returing the sum
}

int main(int argc, char *argv[])
{
  int arr1[] = { 1, 54, 76, 53, 24, 0 };
  int arr2[] = { 4, 24, 32, 43, 25, 0 };

  long retval;
  long sum = 0;

  pthread_t t1;                               +2 Allocates thread structures
  pthread_t t2;

  pthread_create(&t1, NULL, calc_sum, (void *) arr1);   +1 Recognizes need for pthread_create
  pthread_create(&t2, NULL, calc_sum, (void *) arr2);   +1 Calls pthread_create twice
                                              +1 Arguments to pthread_create are correct.
                        +1 Somehow properly assigns individual arrays to the correct threads

  pthread_join(t1, (void **) &retval);        +2 Correct joins both threads
  sum += retval;
  pthread_join(t2, (void **) &retval);
  sum += retval;                              +2 Sums the results from both threads

  printf("The total sum is %ld\n", sum);      +2 Prints the total sum

  pthread_exit(NULL);
}
```

5. (25 points) Synchronization with Barriers

A *barrier* is a synchronization primitive that is used to allow threads to synchronize their progress by waiting for each other to all finish certain parts of the work. When a thread finishes a milestone, it waits at the barrier until all other threads also finish that same milestone. For example, a worker thread finishing milestone k will not proceed to start working on milestone $k + 1$ until all the other threads have also finished milestone k. In practice, this means that threads which finish the milestone early are put to sleep and get woken up later when the last thread finishes the milestone.

Consider the following simple example of three threads using a barrier:

```java
class WorkerThread extends Thread {
  Barrier b;

  public WorkerThread(Barrier b) {
    this.b = b;
  }

  private void doWork( int k ) {
    /*
    * Do work unit k, then return
    */
    return;
  }

  public void run() {
    int k=0;
    while (true) {
      // Do a set of work
      doWork(k++);

      /* Enter the barrier to wait for other threads
       * to finish their part of the work
       * When all threads have finished, this returns.
       */
      b.enterBarrier();
    }
  }
}

public class BarrierExample {
  public static void main(String args[]) {

    Barrier b = new Barrier(3);

    for (int i = 0; i < 3; i++) {
      Thread t = new WorkerThread(b);
      t.start();
    }

  }
}
```

Code the Java class Barrier such that one object of this class, once shared among a group of worker threads, can be reused to provide this kind of synchronization for each milestone.

Hints: Put threads to sleep if they need to wait and wake them up later when it is time for them to proceed.

```
public class Barrier {
```

Solution:

```
public class Barrier {
  private int total_threads;          +4 Recognizes need to store total number of threads

  public Barrier(int total) {                                      +2 Has constructor
    this.total_threads = total;                      +4 Constructor does what it should
  }

  public synchronized void enterBarrier() {
    total_threads--;                      +5 Uses a counter to track how many threads are waiting

    if (total_threads != 0) {              +2 Identifies that a thread is not the last one
      wait();                                                  +2 Calls wait properly
    } else {                                      +2 Identifies thread is the last one
      notifyAll();                                       +2 Uses notifyAll properly
    }
    total_threads++;                              +2 Properly maintains counter value
  }
}
```

Extra Page

End of Exam.