There are 100 possible points. Write all answers on the answer sheets provided (unless otherwise specified).

1. (3 pts each) Briefly define each of the following terms in the context of synchronization and deadlock issues within operating systems.
   (a) critical section
   (b) monitor
   (c) ostrich algorithm
   (d) race condition

2. Consider this proposed software method for enforcing mutual exclusion.

   -process 0-    -process 1-
    -----------    -----------
    flag[0] = false; flag[1] = false;
    while (true) {
      NonCriticalSection();
      flag[0] = true;
      while (flag[1] == true)
        do_nothing();
      CriticalSection();
      flag[0] = false;
    }
    -----------    -----------
    flag[1] = false;
    while (true) {
      NonCriticalSection();
      flag[1] = true;
      while (flag[0] == true)
        do_nothing();
      CriticalSection();
      flag[1] = false;
    }

   (a) (4 pts) Does this method enforce mutual exclusion? Explain.
   (b) (2 pts) Besides the question of enforcing mutual exclusion, are there other concerns with this solution? Explain.

3. (4 pts) Why would semaphores be preferred to the use of spin locks for enforcing mutual exclusion in a single-processor system?

4. For semaphores to work correctly, the $P(S)$ and $V(S)$ operations must be atomic.

   (a) (2 pts) What does “atomic” mean in this context?
   (b) (4 pts) Describe in detail (provide an example) what can happen if these operations are not atomic.
   (c) (2 pts) Describe one possible way that these operations could be made to be atomic.

5. (4 pts) List the four necessary conditions for deadlock.

6. (4 pts) If you were designing an operating system, what mechanism for handling deadlock would you choose? Why?
7. (4 pts) Does this instance of Dijkstra’s Bankers Algorithm represent a safe state or an unsafe state? Explain.
   - Bank still has $100 to loan
   - A has $50 and won’t need more than $200
   - B has $100 and won’t need more than $210
   - C has $5 and won’t need more than $10

8. Imagine a system with three types of resources (2 of $R_1$, 2 of $R_2$, and 2 of $R_3$) and three active processes ($P_1$, $P_2$, and $P_3$). Suppose that $P_1$ has 1 of $R_1$, 2 of $R_2$ and wants 1 of $R_3$. Furthermore, $P_2$ has 1 of $R_3$ and wants 1 of $R_1$. Finally, $P_3$ has 1 of $R_1$ and wants 1 of $R_2$.
   (a) (4 pts) Represent this system as a resource graph.
   (b) (2 pts) Reduce the resulting graph.
   (c) (2 pts) Interpret the reduced graph.
   (d) (2 pts) Would this graph be most useful for deadlock prevention, deadlock detection, or deadlock avoidance?

9. (4 pts) Briefly describe the Dining Philosopher’s problem. Don’t forget to specify the goal.

10. (4 pts) Name an advantage and disadvantage of using monitors rather than semaphores.

11. (16 pts) The Readers-Writers Problem is a classic synchronization problem. There are variations of this problem. For now, consider the “strong reader preference” variation which was described in class. In this version, incoming readers are allowed to start reading as long as no writer is currently writing to the shared resource.
    Suppose that a multithreaded Java application makes use of the attached ServerThread class to handle requests from concurrently running clients to read from and write to a single instance of a SharedResource object.
    Write the necessary changes to the provided ServerThread class that will use Java semaphores to enforce the concurrency rules specified by the readers/writers problem. NOTE: One approach is to create a separate class that is shared by each server thread. If you take this approach you will write the new class and share it with each instance of ServerThread by providing it to the constructor.
    Feel free to write in modifications on the provided printout.
class ServerThread implements Runnable
{
    private Socket con;
    private SharedResource sr;
    // sr.write(str) writes string str to the shared resource
    // ans= sr.read() reads string from the shared resource

    public ServerThread(Socket con, SharedResource sr)
    {
        this.con= con;
        this.sr= sr;
    }

    public void run()
    {
        String command, ans;
        Scanner in= new Scanner(con.getInputStream());
        PrintStream out= new PrintStream(con.getOutputStream());

        command= in.nextLine(); // read client's command
        while (!command.equals("quit")) {
            if (command.equals("read")) {
                ans= sr.read(); // read data from shared resource
                out.println(ans); // send data to client
            }

            if (command.equals("write")) {
                ans= in.nextLine(); // get value to write from client
                sr.write(ans); // then write to shared resource

            }

            command= in.nextLine(); // get next command
        }
    }
}