15-440 Homework #1 (Spring 2011)
Due: Thursday, February 110, 11:59 pm

Communication

1. Consider a sliding window protocol, such as the one we discussed in class. What effect does the window size have upon throughput? Please derive a formula that expresses the maximum *throughput* (bits/second) of the connection between two systems as a function of the *bit rate* of the channel, the one-way *latency* of the channel (seconds), the *window size* (number of frames), and the *frame size* (bits).

2. Again consider the window size of a sliding window protocol. What factor(s) influence the appropriate window size? What happens if the window is too small? Too large?

3. I suggested in class that TCP will generate a duplicate ACK if a segment being lost or reordered. Please draw a picture that demonstrates each of these situations.

4. Does it make sense to use slow start when initiating communication with another host on the same network (LAN)? Why or why not?

5. Given the existing fabric of the standard internet protocols, how do routers communicate to senders that they have reached, or will soon reach, overburdened?

6. Java's RMI considers all parameters, except remote object references to be input parameters. They are passed by copy and not returned. In the context of the Java environment, why is this a necessity? In other words, why can't Java emulate a pass-by-reference as a pass by in-out, as was done in C with RPC?

7. In Java objects are very straight-forward to decompose and reconstruct. .class files provide blue-prints for each type of object, and Java has rich functionality for extracting the properties of an object, including all of its state.

   As a result, it would seem that the serialization of an object for use as a parameter or return value would require nothing from the object, itself. But, it doesn't work this way. Java's RMI can only send *Serializable* objects as parameters or return values. These classes of objects implement their own serialization methods.
   Assume that .class files could be sent to a client, as needed, via HTTP, as is done with stubs. Why doesn't the Java RMI just parse the objects, flatten the fields, and serialize the objects automatically? In your discussion, you should include at least one example of a problematic type of object.

Mutual Exclusion

8. The following is a potential solution to the challenge of properly ensuring mutually exclusive access to a critical section using only the atomicity of fundamental memory operations. By fundamental memory operations we mean loads or stores. Operations such as += are not fundamental and therefore are not considered to be atomic. Please argue for its correctness or provide an explained execution sequence that demonstrates a failure.

```
Process (int pid)
{
  boolean request_set[MAX_PIDS];
  boolean request_lock = FALSE;
  boolean in_cs = FALSE;

  while (FOREVER){

    /* Make request */

    while (request_lock)
    ;

    request_lock = TRUE; // Acquire lock on request list
    request_set[pid] = WANT_CS; // Add request to list
    request_lock = false; // Release list lock

    /* Wait for our turn */
    for (int index=(pid+1); index < MAX_PIDS; index++){
      while (request_set[index]);
    }

    in_cs = TRUE; // Note that we're in the CS

    // Enter critical section
    << Critical Section >>

    // Get out of the CS
    in_cs = FALSE;

    << Non-Critical Section >>

   request_set[pid] = DONT_WANT_CS;
  }
}
```

9. Fix the code below so that the numbers print in order, using mutexes and conditional variables.

```
#include <pthread.h>
#include <stdio.h>

void* thread_one (void* args){
  printf("1");
  printf("5");
  return NULL;
}

void* thread_two (void* args){
  printf("2");
  printf("4");
  return NULL;
}

int main(){
  pthread_t tid_one;
  pthread_t tid_two;

  pthread_create(&tid_one, NULL, thread_one, NULL);
  pthread_create(&tid_two, NULL, thread_two, NULL);

  printf("3");

  pthread_join(tid_one, NULL);
  pthread_join(tid_two, NULL);

  printf("6\n");
}
```

Time

10. Assume that the real-time clocks within some population of *N* workstations can drift, at most, 10 seconds per day. Consider the task of synchronizing these clocks, to within 0.01 seconds of each other, using each of *Cristian's Algorithm* and *The Berkeley Algorithm*.

    How many messages are required for each approach?

11. In class we discussed using a host id, such as IP address, to break ties and impose a total ordering on Lamport time stamps. This approach, on its face, is unfair. Some hosts are more likely to win than others. Why don't we just randomly break ties when they occur? Would this approach provide for a total ordering? Why or why not?

12. In class we discussed vector timestamps as a tool for detecting causality violations, but we didn't discuss certain trade-offs present in the design of the approach. The algorithm that we discussed in class sends the full vector of timestamps with each message. This can be wasteful if the sending processor has received very few messages since the last time it sent a message to the same host. In this case, most (if not all) of the entries remain unchanged.

    A protocol can be designed that reduces the size of the vector timestamp by sending fewer of these uninteresting entries, but this savings comes at the expense of overhead, including storage and processing, on each host.

    Please assume that the hosts have plenty of processing power and O(P^2) space available to store state information related to the timestamp protocol. Operating under these assumptions, design an algorithm for compressed vector timestamps that reduces the size of the vector timestamps as much as possible, without affecting the utility of the timestamping. Please describe the resulting approach.