1. (25%) Explain why spinlocks are not appropriate for uniprocessor systems yet may be suitable for multiprocessor systems.

**Answer:** Textbook p.202

2. (25%) Prove that, in the bakery algorithm, the following property holds: If \( P_i \) is in its critical section and \( P_k \) (\( k \neq i \)) has already chosen its number \( k \neq 0 \), then \( (\text{number}[i], i) < (\text{number}[k], k) \).

**Answer:** Suppose that \( P_i \) is in its critical section, and \( P_k (k \neq i) \) has already chosen its number \( k \), there are two cases:

1. \( P_k \) has already chosen its number when \( P_i \) does the last test before entering its critical section: In this case, if \( (\text{number}[i], i) < (\text{number}[k], k) \) does not hold, since they can not be equal, so \( (\text{number}[i], i) > (\text{number}[k], k) \). Suppose this is true, then \( P_i \) can not get into the critical section before \( P_k \) does, and \( P_i \) will looping at the last while statement until the condition does not hold, which is modified by \( P_k \) when it exits from its critical section. Note that if \( P_k \) get a new number again, it must be larger than of \( P_i \)’s.

2. \( P_k \) did not chose its number when \( P_i \) does the last test before entering its critical section: In this case, since \( P_k \) has chosen its number when \( P_i \) is in its critical section, \( P_k \) must chose its number later than \( P_i \). According to the algorithm, \( P_k \) can only get a bigger number than \( P_i \), so \( (\text{number}[i], i) < (\text{number}[k], k) \) holds.

3. (25%) The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, \( P_0 \) and \( P_1 \), share the following variables:

   boolean flag [2]; /* initially false */
   int turn;

   The structure of process \( P_i \) (\( i == 0 \) or 1), with \( P_j \) (\( j == 1 \) or 0) being the other process, is shown in the following figure. Prove that the algorithm satisfies all three requirements for the critical-section problem.

   ```
   do {
       flag [i] = true;
       while (flag [j]) {
           if (turn == j) {
               flag [i] = false;
               while (turn == j);
               flag [i] = true;
           }
       }
   }
   critical section
   turn = j;
   flag [i] = false;
   remainder section
   ```
1. Mutual exclusion: turn = i or j, not both.
2. Progress: “while (flag [j])”.
3. Bounded waiting: after exiting the critical section, Pi makes “turn = j” and “flag [i] = false”.

4. (0%) The Sleeping-Barber Problem. A barbershop consists of a waiting room with \( n \) chairs and the barber room containing the barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber. Write a program to coordinate the barber and the customers.

**Answer:**

```c
int CustomerWait=0;
semaphore mutex=1;
semaphore CustomerCome=0;
semaphore BeginBarber=0;
semaphore EndBarber=0;

Barber process:
while(1)
{
    wait(CustomerCome);//barber goes to sleep
    wait(mutex);
    call next waiting customer to the barber room;
    CustomerWait--;
    signal(mutex);
    signal(BeginBarber);
    barbering;
    signal(EndBarber);
}

Customer process:
wait(mutex);
if(CustomerWait<n)
{
    CustomerWait++;
    signal(mutex);
    signal(CustomerCome);//wakes the barber up
}
```
wait(BeginBarber);
    bartering;
    wait(EndBarber);

    leave barber shop;
} else {
    signal(mutex);
    leave barber shop;
}

5. (25%) Write a bounded-buffer monitor in which the buffers (portions) are embedded within the monitor itself.

**Answer:**

```c
monitor bb {
    char buffer[BUFFER_SIZE];
    int in, out, count;
    condition not_full, not_empty;

    void produce(char x) {
        if(count==BUFFER_SIZE) {
            not_full.wait();
        }
        buffer[in]=x;
        in=(in+1)%BUFFER_SIZE;
        count++;
        not_empty.signal();
    }

    void consume(char x) {
        if(count==0) {
            not_empty.wait();
        }
        x=buffer[out];
        out=(out+1)%BUFFER_SIZE;
        count--;
        not_full.signal();
    }

    void init()
```
```plaintext
{
  in=0;
  out=0;
  count=0;
}
```