6.7
We have starvation as follows: Let one of the processes, say process \( P_0 \), wish to enter its CS. It will set flag[0] to true, turn to 1, and enter the while loop. It will iterate indefinitely in its while loop because its governing condition is true. If \( P_1 \) also wishes to enter its CS, it will set flag[1] to true and turn to 0. The change in the value of turn does not affect \( P_0 \), because flag[1] has been changed to true. Hence \( P_0 \) continues to loop. Process \( P_1 \) will also loop indefinitely because flag[0] is true.

6.10
Following the hint given in the problem itself, define a binary semaphore \( rw \) permission that is initialized to 1. A reader process checks \( runread \) to find whether reading is already in progress. If so, it goes on to read straightaway; otherwise, it performs a \( wait(rw \) permission). When it finishes reading, it checks whether any other readers are engaged in reading. If not, it performs a \( signal(rw \) permission) which would wake up a writer process waiting to write, if any, or enable reading and writing in future. A writer process uses semaphore \( rw \) permission instead of the semaphore \( writing \). It performs a \( wait on \) \( rw \) permission, writes, and performs a \( signal on \) \( rw \) permission.

```plaintext
var
  runread : integer;
  rw_permission : semaphore := 1;
  sem_CS : semaphore := 1;

Initialization
  runread := 0;

Reader(s):
repeat
  wait(sem_CS);
  if runread = 0 then
    wait(rw_permission);
    runread := runread + 1;
    signal(sem_CS); // wait(rw_permission) and signal(sem_CS) should be done atomically
  else
    runread := runread + 1;
    signal(sem_CS);

{Read}
  wait(sem_CS);
  runread := runread – 1;
  if runread = 0 then
    signal(rw_permission);
  signal(sem_CS);
forever;
```
Writer(s):

repeat
    wait(rw_permission);
    {Write}
    signal(rw_permission);

forever;

6.17
The monitor should have an appropriate number of condition variables as explained in the following, and a data structure to hold pending debits. It has two procedures named debit and credit. Every banking transaction is assumed to be a process—we call it a transaction process. A process invokes the procedures credit or debit with its own id and the amount to be credited or debited.

Procedure debit carries out a feasible debit and returns. It enters an infeasible debit in the pending debits structure and executes a wait statement on an appropriate condition variable. Procedure credit ignores a credit if balance > n. Otherwise, it performs the credit operation.

Now, as many pending debits should be performed as possible. One approach to implementing it is to have a condition variable for each of the transaction processes and maintain a list of condition variables of those processes whose debits were delayed. After a credit operation, each of the blocked processes would be activated through a signal operation on its condition variable. It would check whether its debit can be performed and remove its condition variable from the linked list if it is the case. However, this approach needs a condition variable for each transaction, which is infeasible if the number of banking transactions is not known. To avoid this problem, the monitor could provide a finite set of condition variables, associate a condition variable dynamically with a transaction process if its debit is delayed, and store this association in its own data structures. An alternative approach is to use a single condition variable on which all transaction processes would wait, maintain a count of the number of processes blocked on it, and arrange to execute those many signal statements following a credit operation. In both approaches, a process that is activated would execute a wait statement once again if its debit still cannot be performed. It amounts to a busy wait situation. It can be avoided in the first approach by activating a process only when its debit operation can be performed; however, it cannot be avoided in the second approach. The following monitor design uses the second approach.

type Bank_account_type = monitor
    const
        n = ...;
    var
        balance : integer;
        feasible: condition;
        blocked : integer; {number of blocked processes}
procedure credit(pid : integer, amount : integer);
begin
    var i : integer;
    if balance <= n then
        balance := balance + amount;
        for (i = 0; i < blocked; i++)
            feasible.signal;
end;
procedure debit(pid : integer, amount : integer);
begin
    repeat
        if balance >= amount then
            balance := balance - amount;
            exit;
        else
            blocked := blocked + 1;
            feasible.wait;
            blocked := blocked - 1;
        forever;
    end;
begin {initialization}
    balance := 0;
    blocked := 0;
end;

6.18
We have one barber process and several customer processes. A deadlock arises if the barber goes
to sleep and customers expect him to wake up and admit a customer, whereas the barber expects
a customer to wake him. The following solution ensures that deadlock cannot arise.

Shared variables:
semaphore customers := 0 //number of customers waiting for service
semaphore barber := 0
semaphore mutex := 1 //for mutual exclusive access to variable ‘waiting’
int waiting := 0 //number of customers waiting for service

Barber:
while (1) {
    wait(customers); //go to sleep if no customers are waiting
    wait(mutex);
    waiting := waiting - 1;
    signal(mutex);
    signal(barber); //barber is ready to cut hair
    cut_hair();
}
Customer(s):
wait(mutex);
If (waiting < n ) {  
          waiting := waiting + 1;
          signal(mutex);
          signal(customers); //wake up barber if necessary
          wait(barber);
          get_haircut();
    }  
else signal(mutex);