Table 5.1 Some Key Terms Related to Concurrency

atomic operation	A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.
critical section	A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.
deadlock	A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
livelock	A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.
mutual exclusion	The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
race condition	A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.
starvation	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

```
boolean flag [2];
int turn;
void P0()
{
     while (true) {
          flag [0] = true;
     A:   while (flag [1]) {
               if (turn == 1) {
                    flag [0] = false;
                    while (turn == 1);

                    flag [0] = true;
               }
          }
          /* critical section  */
          turn = 1;
          flag [0] = false;
          /* remainder   */
     }
}
void P1( )
{
     while (true) {
          flag [1] = true;
     B:   while (flag [0]) {
               if (turn == 0) {
                    flag [1] = false;
                    while (turn == 0);

                    flag [1] = true;
               }
          }
          /* critical section   */
          turn = 0;
          flag [1] = false;
          /* remainder   */
     }
}
void main ()
{
     flag [0] = false;
     flag [1] = false;
     turn = 1;
     parbegin (P0, P1);
}
```

Dekker's Algorithm

Combine aspects of previous attempts (`flag, turn`)

Now it works!

But it's complicated.

And you need to prove that it works.

Dekker's Algo : ME Proof

Proof : Suppose P0 and P1 are both in the CS. the proof is by contradiction.

P0 : Initially, flag[0] = true and then P0 executes a loop at A: that checks flag[1]. this loop does not terminate unless flag[1] = false. It's also clear that flag[0] remains true when the A: loop exits. Furthermore, only process P0 can change flag[0]. thus flag[0] = true and flag[1] = false.

P1: Initially flag[1] = true at the P1 executes a loop at B: that checks flag[0]. this loop does not terminate unless flag[0] = false. It's also clear that flag[1] remains true when the B: loop exits. Only process P1 can change flag[1]. therefore flag[0] = false at flag[1] = true.

Clearly both sets of underlined conditions cannot be simultaneously true, therefore P0 at P1 cannot be in the CS at the same time, as first supposed.

□.

Note that this is only a safety proof—— fairness, liveness, et al are separate proofs.