This algorithm satisfies the three conditions. Before we show that the three conditions are satisfied, we give a brief explanation of what the algorithm does to ensure mutual exclusion. When a process $i$ requires access to critical section, it first sets its flag variable to want in to indicate its desire. It then performs the following steps: (1) It ensures that all processes whose index lies between turn and $i$ are idle. (2) If so, it updates its flag to in cs and checks whether there is already some other process that has updated its flag to in cs. (3) If not and if it is this process’s turn to enter the critical section or if the process indicated by the turn variable is idle, it enters the critical section. Given the above description, we can reason about how the algorithm satisfies the requirements in the following manner:

a. Mutual exclusion is ensured: Notice that a process enters the critical section only if the following requirements are satisfied: no other process has its flag variable set to in cs. Since the process sets its own flag variable set to in cs before checking the status of other processes, we are guaranteed that no two processes will enter the critical section simultaneously.

b. Progress requirement is satisfied: Consider the situation where multiple processes simultaneously set their flag variables to in cs and then check whether there is any other process has the flag variable set to in cs. When this happens, all processes realize that there are competing processes, enter the next iteration of the outer while(1) loop and reset their flag variables to want in. Now the only process that will set its turn variable to in cs is the process whose index is closest to turn. It is however possible that new processes whose index values are even closer to turn might decide to enter the critical section at this point and therefore might be
able to simultaneously set its flag to in cs. These processes would then realize there are competing processes and might restart the process of entering the critical section. However, at each iteration, the index values of processes that set their flag variables to in cs become closer to turn and eventually we reach the following condition: only one process (say $k$) sets its flag to in cs and no other process whose index lies between turn and $k$ has set its flag to in cs. This process then gets to enter the critical section.

c. Bounded-waiting requirement is met: The bounded waiting requirement is satisfied by the fact that when a process $k$ desires to enter the critical section, its flag is no longer set to idle. Therefore, any process whose index does not lie between turn and $k$ cannot enter the critical section. In the meantime, all processes whose index falls between turn and $k$ and desire to enter the critical section would indeed enter the critical section (due to the fact that the system always makes progress) and the turn value monotonically becomes closer to $k$. Eventually, either turn becomes $k$ or there are no processes whose index values lie between turn and $k$, and therefore process $k$ gets to enter the critical section.

6.5

The pseudocode is as follows:

```c
monitor file access {
    int curr sum = 0;
    int n;
    condition c;

    void access file(int my num) {
        while (curr sum + my num >= n)
            ; // Do nothing
        c.wait();
        curr sum += my num;
    }

    void finish access(int my num) {
        curr sum -= my num;
        c.broadcast(); // 或是寫c.signal();
    }
}
```
6.10

Solaris, Linux, and Windows 2000 use spinlocks as a synchronization mechanism only on multiprocessor systems. Spinlocks are not appropriate for single-processor systems because the condition that would break a process out of the spinlock could be obtained only by executing a different process. If the process is not relinquishing the processor, other processes do not get the opportunity to set the program condition required for the first process to make progress. In a multiprocessor system, other processes execute on other processors and thereby modify the program state in order to release the first process from the spinlock.

6.14

A schedule refers to the execution sequence of the operations for one or more transactions. A serial schedule is the situation where each transaction of a schedule is performed atomically. If a schedule consists of two different transactions where consecutive operations from the different transactions access the same data and at least one of the operations is a write, then we have what is known as a conflict. If a schedule can be transformed into a serial schedule by a series of swaps on nonconflicting operations, we say that such a schedule is conflict serializable. The two-phase locking protocol ensures conflict serializability because exclusive locks (which are used for write operations) must be acquired serially, without releasing any locks during the acquire (growing) phase. Other transactions that wish to acquire the same locks must wait for the first transaction to begin releasing locks. By requiring that all locks must first be acquired before releasing any locks, we are ensuring that potential conflicts are avoided.
6.19

There are many answers to this question. Some kernel data structures include a process id (pid) management system, kernel process table, and scheduling queues. With a pid management system, it is possible two processes may be created at the same time and there is a race condition assigning each process a unique pid. The same type of race condition can occur in the kernel process table: two processes are created at the same time and there is a race assigning them a location in the kernel process table. With scheduling queues, it is possible one process has been waiting for IO which is now available. Another process is being context-switched out. These two processes are being moved to the Runnable queue at the same time. Hence there is a race condition in the Runnable queue.

6.24

Simplicity. If RWlocks provide fairness or favor readers or writers, there is more overhead to the lock. By providing such a simple synchronization mechanism, access to the lock is fast. Usage of this lock may be most appropriate for situations where reader−locks are needed, but quickly acquiring and releasing the lock is similarly important.