Remaining Contemplation Questions

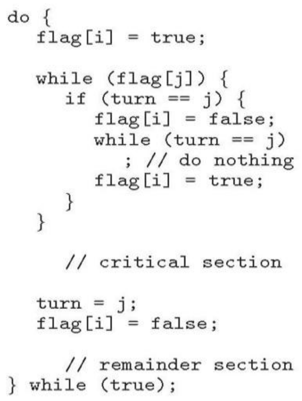
Process Synchronisation

1. The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P0 and P1, share the following variables:

```
boolean flag[2];/*initiallyfalse*/

int turn;
```

The structure of process Pi (i ==0or1) is shown in the figure below; the other process is Pj (j ==1or0). 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)
            ; // do nothing
         flag[i] = true;
      }
   }

      // critical section

   turn = j;
   flag[i] = false;

      // remainder section
} while (true);
```

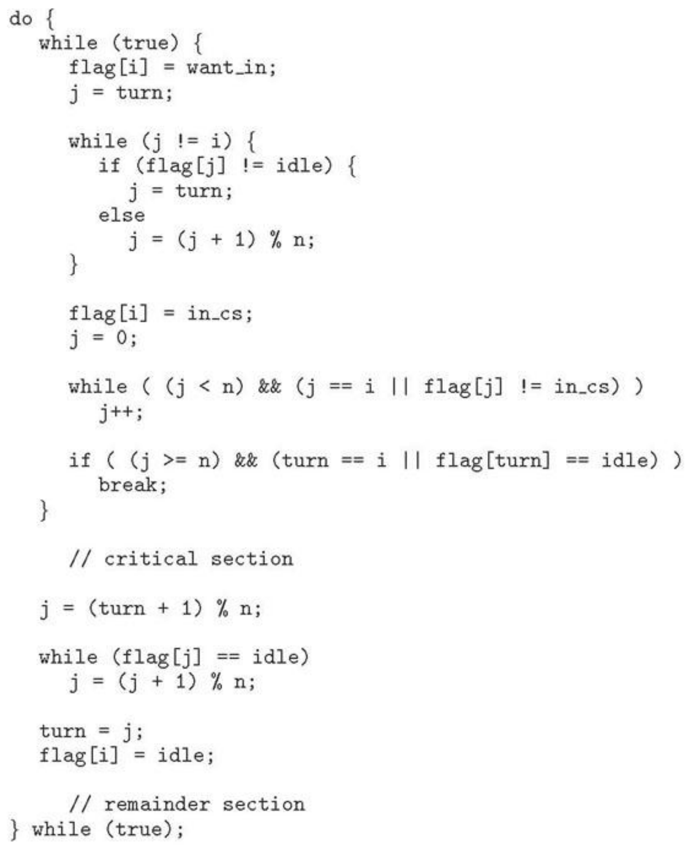
Figure 6.42 The structure of process P_i in Dekker's algorithm.

This algorithm satisfies the three conditions: (1) Mutual exclusion is ensured through the use of the flag and turn variables. If both processes set their flag to true, only one will succeed, namely, the process whose turn it is. The waiting process can only enter its critical section when the other process updates the value of turn. (2) Progress is provided, again through the flag and turn variables. This algorithm does not provide strict alternation. Rather, if a process wishes to access their critical section, it can set their flag variable to true and enter their critical section. It sets turn to the value of the other process only upon exiting its critical section. If this process wishes to enter its critical section again – before the other process – it repeats the process of entering its critical section and setting turn to the other process upon exiting. (3)Bounded waiting is preserved through the use of the turn variable. Assume two processes wish to enter their respective critical sections. They both set their value of flag to true; however, only the thread whose turn it is can proceed; the other thread waits. If bounded waiting were not preserve, it would therefore be possible that the waiting process would have to wait indefinitely while the first process repeatedly entered – and exited – its critical section. However, Dekker's algorithm has a process set the value of turn to the other process, thereby ensuring that the other process will enter its critical section next.

2. The first known correct software solution to the critical-section problem for *n* processes with a lower bound on waiting of *n* − 1 turns was presented by Eisenberg and McGuire. The processes share the following variables:

```
enum pstate {idle, want in, in cs};
pstate flag[n];
int turn;
```

All the elements of flag are initially idle; the initial value of turn is immaterial (between 0 and n-1). The structure of process *Pi* is shown in the figure below. Prove that the algorithm satisfies all three requirements for the critical-section problem.



```
do {
   while (true) {
      flag[i] = want_in;
      j = turn;

      while (j != i) {
         if (flag[j] != idle) {
            j = turn;
         else
            j = (j + 1) % n;
      }

      flag[i] = in_cs;
      j = 0;

      while ( (j < n) && (j == i || flag[j] != in_cs) )
         j++;

      if ( (j >= n) && (turn == i || flag[turn] == idle) )
         break;
   }

      // critical section

   j = (turn + 1) % n;

   while (flag[j] == idle)
      j = (j + 1) % n;

   turn = j;
   flag[i] = idle;

      // remainder section
} while (true);
```

Figure 6.43 The structure of process P_i in Eisenberg and McGuire's algorithm.

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 set sits 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:
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.
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.
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.

3. Discuss the tradeoff between fairness and throughput of operations in the readers-writers problem. Propose a method for solving the readers-writers problem without causing starvation.

   Throughput in the readers-writers problem is increased by favouring multiple readers as opposed to allowing a single writer to exclusively access the shared values. On the other hand, favouring readers could result in starvation for writers. The starvation in the readers-writers problem could be avoided by keeping timestamps associated with waiting processes. When a writer is finished with its task, it would wakeup the process that has been waiting for the longest duration. When a reader arrives and notices that another reader is accessing the database, then it would enter the critical section only if there are no waiting writers. These restrictions would guarantee fairness.

4. How does the signal() operation associated with monitors differ from the corresponding operation defined for semaphores?

   The signal() operation associated with monitors is not persistent in the following sense: if a signal is performed and if there are no waiting threads, then the signal is simply ignored and the system does not remember that the signal took place. If a subsequent wait operation is performed, then the corresponding thread simply blocks. In semaphores, on the other hand, every signal results in a corresponding increment of the semaphore value even if there are no waiting threads future wait operation would immediately succeed because of the earlier increment.

5. What is the meaning of the term busy waiting? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.

   Busy waiting means that a process is waiting for a condition to be satisfied in a tight loop without relinquishing the processor. Alternatively, a process could wait by relinquishing the processor, and block on a condition and wait to be awakened at some appropriate time in the future. Busy waiting can be avoided but incurs the overhead associated with putting a process to sleep and having to wake it up when the appropriate program state is reached.

6. Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user level programs.

   If a user-level program is given the ability to disable interrupts, then it can disable the timer interrupt and prevent context switching from taking place, thereby allowing it to use the processor without letting other processes execute.

7. Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.

   Interrupts are not sufficient in multiprocessor systems since disabling interrupts only prevents other processes from executing on the processor in which interrupts were disabled; there are no limitations on what processes could be executing on other processors and therefore the process disabling interrupts cannot guarantee mutually exclusive access to program state.

8. Demonstrate that monitors and semaphores are equivalent in so far as they can be used to implement the same types of synchronization problems.

   A semaphore can be implemented using the following monitor code:

```
monitor semaphore {
    int value = 0;
    condition c;

    semaphore_increment() {
      value++;
      c.signal();
    }

    semaphore_decrement() {
      while (value == 0)
      c.wait();
      value--;
    }
}
```

   A monitor could be implemented using a semaphore in the following manner. Each condition variable is represented by a queue of threads waiting for the condition. Each thread has a semaphore associated with its queue entry. When a thread performs await operation, it creates a new semaphore (initialized to zero), appends the semaphore to the queue associated with the condition variable, and performs a blocking semaphore decrement operation on the newly created semaphore. When a thread performs a signal on a condition variable, the first process in the queue is awakened by performing an increment on the corresponding semaphore.

9. Show that, if the acquire() and release() semaphore operations are not executed atomically, then mutual exclusion may be violated.

   An acquire operation atomically decrements the value associated with a semaphore. If two acquire operations are executed on a semaphore when its value is 1, if the two operations are not performed atomically, then it is possible that both operations might proceed to decrement the semaphore value, thereby violating mutual exclusion.

10. Suppose we replace the wait() and signal() operations of monitors with a single construct await(B), where B is a general Boolean expression that causes the process executing it to wait until B becomes true.
    a. Write a monitor using this scheme to implement the readers–writers problem.
    b. Explain why, in general, this construct cannot be implemented efficiently.
    c. What restrictions need to be put on the await statement so that it can be implemented efficiently? (Hint: Restrict the generality of B; see Kessels[1977].)

    a. The readers–writers problem could be modified with the following more general await statements: A reader can perform "await(active_writers==0 && waiting_writers==0)" to check that there are no active writers and there are no waiting writers before it enters the critical section. The writer can perform a "await(active_writers==0 && active_readers==0)" check to ensure mutually exclusive access.
    b. The system would have to check which one of the waiting threads have to be awakened by checking which one of their waiting conditions are satisfied after a signal. This requires considerable complexity and might require some interaction with the compiler to evaluate the conditions at different points in time.
    c. One could restrict the Boolean condition to be a disjunction of conjunctions with each component being a simple check (equality or inequality with respect to a static value) on a program variable. In that case, the Boolean condition could be communicated to the run-time system, which could perform the check every time it needs to determine which thread to be awakened.

Deadlocks

1. Consider the following snapshot of a system:

Allocation	Max	Available
A B C D	A B C D	A B C D

P0 0 0 1 2	0 0 1 2	1 5 2 0
P1 1 0 0 0	1 7 5 0	
P2 1 3 5 4	2 3 5 6	
P3 0 6 3 2	0 6 5 2	
P4 0 0 1 4	0 6 5 6	

Answer the following questions using the banker's algorithm:

a) What is the content of the matrix Need?
b) Is the system in a safe state?
c) If a request from process P1 arrives for (0,4,2,0), can the request be granted immediately?

What is the content of the matrix Need? The values of Need for processes P0 through P4 respectively are (0,0,0,0), (0,7,5,0), (1,0,0,2), (0,0,2,0), and (0,6,4,2).

Is the system in a safe state? Yes. With Available being equal to (1,5,2,0), either process P0 or P3 could run. Once process P3 runs, it releases its resources, which allow all other existing processes to run.

If a request from process P1 arrives for (0,4,2,0), can the request be granted immediately? Yes, it can. This results in the value of Available being (1,1,0,0). One ordering of processes that can finish is P0, P2, P3, P1, and P4.

2. Consider a system consisting of four resources of the same type that are shared by three processes, each of which needs at most two resources. Show that the system is deadlock-free.

   Suppose the system is deadlocked. This implies that each process is holding one resource and is waiting for one more (hold and wait). Since there are three processes and four resources, one process must be able to obtain two resources. This process requires no more resources and, therefore it will return its resources when done.

3. Consider a system consisting of m resources of the same type, being shared by n processes. Resources can be requested and released by processes only one at a time. Show that the system is deadlock free if the following two conditions hold:
   - The maximum need of each process is between 1 and m resources
   - The sum of all maximum needs is less than m + n

   Using the terminology of Section7.6.2, we have:

a. $\sum_{i=1}^{n} Max_i < m + n$

b. $Max_i \geq 1$ for all i
Proof: $Need_i = Max_i - Allocation_i$
If there exists a deadlock state then:

c. $\sum_{i=1}^{n} Allocation_i = m$

Use a. to get: $\sum Need_i + \sum Allocation_i = \sum Max_i < m + n$
Use c. to get: $\sum Need_i + m < m + n$
Rewrite to get: $\sum_{i=1}^{n} Need_i < n$

This implies that there exists a process P_i such that $Need_i = 0$. Since $Max_i \geq 1$ it follows that P_i has at least one resource that it can release. Hence the system cannot be in a deadlock state.

4. Compare the circular-wait scheme with the deadlock-avoidance schemes (like the banker's algorithm) with respect to the following issues:
   - Runtime overheads
   - System throughput

A deadlock-avoidance scheme tends to increase the runtime overheads due to the cost of keep track of the current resource allocation. However, a deadlock-avoidance scheme allows for more concurrent use of resources than schemes that statically prevent the formation of deadlock. In that sense, a deadlock-avoidance scheme could increase system throughput.

CPU Scheduling

1. Consider a system running ten I/O-bound tasks and one CPU-bound task. Assume that the I/O-bound tasks issue an I/O operation once for every millisecond of CPU computing and that each I/O operation takes 10 milliseconds to complete. Also assume that the context switching overhead is 0.1millisecond and that all processes are long-running tasks. What is the CPU utilization for a round-robin scheduler when:
   - The time quantum is 1 millisecond
   - The time quantum is 10 milliseconds

The time quantum is 1 millisecond: Irrespective of which process is scheduled, the scheduler incurs a 0.1 millisecond context-switching cost for every context-switch. This results in a CPU utilization of 1/1.1*100=91%.

The time quantum is 10 milliseconds: The I/O-bound tasks incur a context switch after using up only 1 millisecond of the time quantum. The time required to cycle through all the processes is therefore 10*1.1+10.1 (as each I/O-bound task executes for 1 millisecond and then incurs the context switch, whereas the CPU-bound task executes for 10 milliseconds before incurring a context switch). The CPU utilization is therefore 20/21.1*100 = 94%.

2. Consider a pre-emptive priority scheduling algorithm based on dynamically changing priorities. Larger priority numbers imply higher priority. When a process is waiting for the CPU (in the ready queue, but not running), its priority changes at a rate w; when it is running, its priority changes at a rate r. All processes are given a priority of 0 when they enter the ready queue. The parameters w and r can be set to give many different scheduling algorithms.
   - What is the algorithm that results from $r > w > 0$?
   - What is the algorithm that results from $w < r < 0$?

   FCFS

   LIFO

3. The traditional UNIX scheduler enforces an inverse relationship between priority numbers and priorities: The higher the number, the lower the priority. The scheduler recalculates process priorities once per second using the following function:

   Priority = (recent CPU usage / 2) + base

   where base = 60 and recent CPU usage refers to a value indicating how often a process has used the CPU since priorities were last recalculated. Assume that recent CPU usage for process P1 is 40, process P2 is 18, and process P3 is 10. What will be the new priorities for these three processes when priorities are recalculated? Based on this information, does the traditional UNIX scheduler raise or lower the relative priority of a CPU-bound process?

   The priorities assigned to the processes are 80, 69, and 65 respectively. The scheduler lowers the relative priority of CPU-bound processes.