Module – 3

PROCESS SYNCHRONIZTION

- A **cooperating process** is one that can affect or be affected by other processes executing in the system.
- Cooperating processes can either directly share the address space (that is, both code and data) or be allowed to share data through files or messages.
- One process may only partially complete execution before another process is scheduled.
- A process may be interrupted at any point in its instruction stream, and the processor may be assigned to execute instructions of another process.
- As different processes update the same data concurrently, we would arrive at an incorrect state because we allow many processes to manipulate the same variable concurrently.
- A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**.
- Eg: Consider a scenario in which husband deposits and wife withdraws from the same account concurrently

P1:	P2:
(Husband deposits 1000/-)	(Wife withdraws 400/-)
1. Read balance;	A. Read balance;
2. balance=balance + 1000;	B. balance = balance – 400;

- Assume that the initial balance is 5000.
- Interleaved execution may result either new balance as 4600/- (1, A, 2, B) or 6000/- (A, 1, B, 2). But sequential execution, in P1, P2 order or P2, P1 order gives the correct answer.
- To guard against the race condition, we need to ensure that only one process at a time can be manipulating the shared variable.
- To make such a guarantee, we require that the processes be synchronized in some way.

THE CRITICAL SECTION PROBLEM

- Consider a system consisting of n processes. Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on.
- When one process is executing in its critical section, no other process is allowed to execute in its critical section.
- That is, no two processes are executing in their critical sections at the same time.

- The **critical-section problem** is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section.
- The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**.

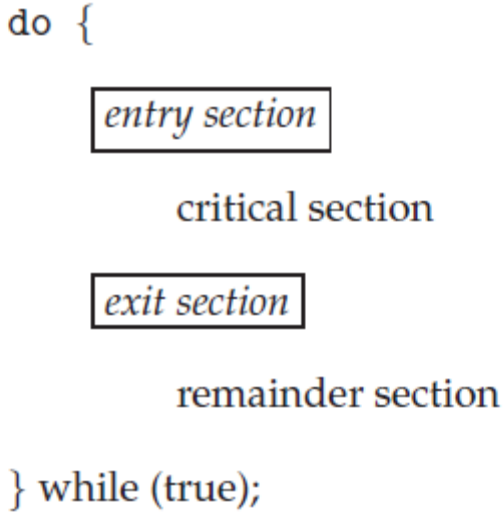

Figure 5.1 General structure of a typical process P_i.

- The entry section and exit section are enclosed in boxes to highlight the important segments of code.
- A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual exclusion**. If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress**. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will

enter its critical section next, and this selection cannot be postponed indefinitely.

3. **Bounded waiting**. There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Race conditions in Kernel processes

- Consider as an example a kernel data structure that maintains a list of all open files in the system.
- This list must be modified when a new file is opened or closed. If two processes were to open files simultaneously, the separate updates to this list could result in a race condition.
- Other kernel data structures that are prone to possible race conditions include structures for maintaining memory allocation, for maintaining process lists, and for interrupt handling.
- Two general approaches are used to handle critical sections in OS: (1) **preemptive kernels** and (2) **nonpreemptive kernels**.
- A preemptive kernel allows a process to be preempted while it is running in kernel mode.
- A nonpreemptive kernel does not allow a process running in kernel mode to be pre-empted

- A nonpreemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time.
- But we cannot say the same about preemptive kernels, so they must be carefully designed to ensure that shared kernel data are free from race conditions.
- However, preemptive kernel is more suitable for real-time programming, as it will allow a real-time process to preempt a process currently running in the kernel.

PETERSON'S SOLUTION

- It is a classic software-based solution to the critical-section problem
- Because of the way modern computer architectures perform basic machine-language instructions, there are no guarantees that Peterson's solution will work correctly on such architectures.
- But it provides a good algorithmic description of solving the critical-section problem
- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
- The processes are numbered Pi and Pj
- Peterson's solution requires the two processes to share two data items:

int turn;

boolean flag[2];

- The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process Pi is allowed to execute in its critical section. If turn == j, then process Pi is allowed to execute in its critical section
- The flag array is used to indicate if a process is ready to enter its critical section.
- For example, if flag[i] is true, this value indicates that Pi is ready to enter its critical section. If flag[i]==false, Pi is not ready.
- Its algorithm is described as follows

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

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

Figure 5.2 The structure of process P_i in Peterson's solution.

- To enter the critical section, process Pi first sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so.

- If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time.
- Only one of these assignments will last; the other will occur but will be overwritten immediately.
- The eventual value of turn determines which of the two processes is allowed to enter its critical section first.
- We now prove that this solution is correct. We need to show that all the 3 requirements have been satisfied

1. Mutual exclusion is preserved.
    - Pi enters in critical section if either flag[j]==false or turn ==i.
    - If both processes are ready, then flag[i] and flag[j] will be true.
    - So value of the shared variable 'turn' will decide
    - Since it is a single variable, it can have only one value at a time. Either i or j
    - If turn==i, Pi will enter into critical section and Pj has to wait
    - Otherwise, if turn==j, Pj will enter into critical section and Pi has to wait
    - So only one process is executing critical section at a time. Hence mutual exclusion is ensured
2. The progress requirement is satisfied.
    - If Pj is not ready to enter the critical section, then flag[j] == false, and Pi can enter its critical section.
    - If Pi is not ready to enter the critical section, then flag[i] == false, and Pj can enter its critical section.

- If both processes are ready, then flag[i] and flag[j] will be true.
- So value of the shared variable ‘turn’ will decide
- Since it is a single variable, it can have only one value at a time. Either i or j
- If turn==i, Pi will continue, otherwise Pj
- So anyone process will enter into critical section. Hence progress is ensured

3. The bounded-waiting requirement is met.
   - If the turn is for Pi and Pj is waiting, after executing the critical section, Pi sets flag[i]=false. At that time Pj can exit from the while loop and enter into critical section.
   - Similar in case of Pj also. So a ready process enter in critical section after at most one execution by the another process
   - Hence bounded waiting is ensured

- Since all the three requirements have been satisfied, it is proved that Peterson’s solution is a valid solution for the critical section problem.