Process Synchronization

- One process may only partially complete execution before another process is scheduled.
- In fact, a process may be interrupted at any point in its instruction stream, and the processing core may be assigned to execute instructions of another process.
- **Parallel execution**, in which two instruction streams (representing different processes) execute simultaneously on separate processing cores.

Producer-Consumer problem

- Producer produces items in the buffer and consumer consumes them from the buffer
- Both are executing simultaneously
- They share the buffer

- One possibility is to add an integer variable counter , initialized to 0.
- counter is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer.
- **Producer**

```
while (true) {
    /* produce an item in next produced */
    while (counter == BUFFER SIZE ); /* do nothing */
    buffer[in] = next produced;
    in = (in + 1) % BUFFER SIZE ;
    counter++;
    }
```

Consumer

```
while (true) {
    while (counter == 0); /* do nothing */
    next consumed = buffer[out];
    out = (out + 1) % BUFFER SIZE ;
    counter--;
    /* consume the item in next consumed */
}
```

- Suppose that the value of the variable counter is currently 5 and that the producer and consumer processes concurrently execute the statements “ counter++ ” and “ counter-- ”.

- The statement “ counter++ ” may be implemented in machine language as follows:

  register1 = counter

  register1 = register1 + 1

  counter = register1

  where register1 is one of the local CPU registers. Similarly, the statement “ counter-- ” is implemented as follows:

  register2 = counter

  register2 = register2 − 1

  counter = register2

  where again register2 is one of the local CPU registers.

- The concurrent execution of “ counter++ ” and “ counter-- ” is equivalent to a sequential execution. One possible interleaving is the following:

- T 0 : producer execute register1 = counter {register1 = 5}
- T 1 : producer execute register1 = register1 + 1 {register1 = 6}
- T 2 : consumer execute register2 = counter {register2 = 5}
- T 3 : consumer execute register2 = register2 − 1 {register2 = 4}
- T 4 : producer execute counter = register1 {counter = 6}
- T 5 : consumer execute counter = register2 {counter = 4}
- We have arrived at the incorrect state “ counter == 4”, indicating that four buffers are full, when, in fact, five buffers are full.
- If we reversed the order of the statements at T 4 and T 5 , we would arrive at the incorrect state “ counter == 6”.
- This reaches incorrect state because we allowed both processes to manipulate the variable counter concurrently.
- When 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**.
- To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter .

The Critical-Section Problem

- Consider a system consisting of n processes {P0 , P1 , ..., Pn−1 }. 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.
- 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**.

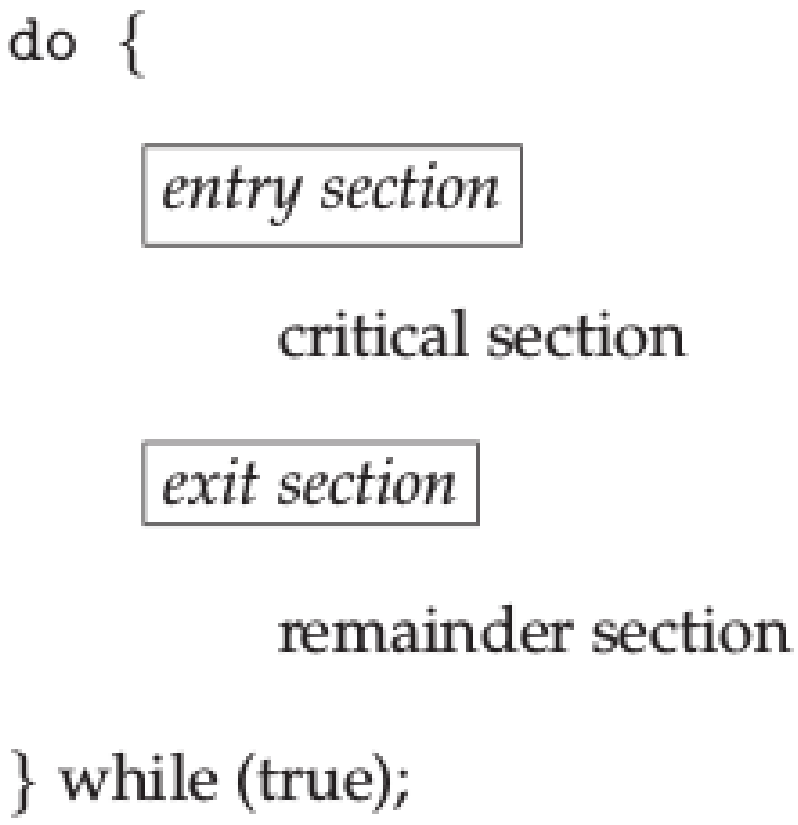

Figure 5.1 General structure of a typical process P_i.

- A solution to the critical-section problem must satisfy the following three requirements:

  1. **Mutual exclusion**. If process P i 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.

- Two general approaches are used to handle critical sections in operating systems: **preemptive kernels and 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 preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

Peterson’s Solution

- Peterson’s solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered P0 and P1 .
- 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.
- 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.

```
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 need to show that:

  1. Mutual exclusion is preserved.

- Each Pi enters its critical section only if either flag[j] == false or turn == i. flag[0] == flag[1] == true is possible at the same time but the value of turn can be either 0 or 1.
- Hence, one of the processes —say, Pj —must have successfully executed the while statement, whereas Pi had to execute at least one additional statement (“ turn == j ”).
- However, at that time, flag[j] == true and turn == j , and this condition will persist as long as Pj is in its critical section;

  2. The progress requirement is satisfied.

  3. The bounded-waiting requirement is met.

- A process Pi can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] == true and turn == j.
- If Pj is not ready to enter the critical section, then flag[j] == false , and Pi can enter its critical section.
- If Pj has set flag[j] to true and is also executing in its while statement, then either turn == i or turn == j . If turn == i , then Pi will enter the critical section. If turn == j , then Pj will enter the critical section.
- Let Pj enters. However, once Pj exits its critical section, it will reset flag[j] to false , allowing Pi to enter its critical section.
- If Pj resets flag[j] to true , it must also set turn to i . Thus, since Pi does not change the value of the variable turn while executing the while statement, Pi will enter the critical section (**progress**) after at most one entry by Pj (**bounded waiting**).

Semaphores

- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal() .
- wait(S) {

  while (S <= 0); // busy wait

  S--;

  }
- signal(S) {

  S++;

  }
- All modifications to the integer value of the semaphore in the wait() and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
- In addition, in the case of wait(S) , the testing of the integer value of S ($S \leq 0$), as well as its possible modification (S--), must be executed without interruption.

- The value of a **counting semaphore** can range over an unrestricted domain.
- The value of a **binary semaphore** can range only between 0 and 1.
- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.
- The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count).
- When a process releases a resource, it performs a signal() operation (incrementing the count).
- When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

- Consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2 .
- Suppose we require that S2 be executed only after S1 has completed. We can implement this scheme readily by letting P1 and P2 share a common semaphore synch , initialized to 0.
- In process P1 , we insert the statements

  S1 ;

  signal(synch);
- In process P 2 , we insert the statements

  wait(synch);

  S2 ;
- Because synch is initialized to 0, P 2 will execute S 2 only after P 1 has invoked signal(synch) , which is after statement S 1 has been executed.