SNS COLLEGE OF TECHNOLOGY

Coimbatore-35.

An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A+' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

COURSE NAME : 19CSB201 – OPERATING SYSTEMS

II YEAR/ IV SEMESTER

UNIT – II Process Scheduling And Synchronization

Topic: Process Synchronization: The Critical-Section Problem

Mr.N.Selvakumar
Assistant Professor
Department of Computer Science and Engineering

Process Synchronization

Process Synchronization is the coordination of execution of multiple processes in a multi-process system to ensure that they **access shared resources in a controlled and predictable manner**.

It aims to **resolve the problem of race conditions** and other synchronization issues in a concurrent system.

Race Condition

- When **more than one process** is executing the same code or **accessing the same memory or any shared variable** in that condition there is a possibility that the **output** or the value of the shared variable is **wrong**.
- Several processes access and process the manipulations over the same data concurrently, then the **outcome depends on the particular order in which the access takes place**.
- A race condition is a situation that may occur inside a critical section.

The Critical-Section Problem

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.

Consider a system consisting of n processes {P0 , P1 , ..., Pn−1 }. The important feature of the system is that, 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**.

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

Figure 5.1 General structure of a typical process P_i.

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**.

The general structure of a typical process Pi is shown in Figure 5.1. The entry section and exit section are enclosed in boxes to highlight these 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.

Approaches to handle critical sections

Two general approaches are used to handle critical sections in operating systems: **preemptive kernels** and **non preemptive kernels**.

A preemptive kernel allows a process to be preempted while it is running in kernel mode.

A non preemptive kernel does not allow a process running in kernel mode to be preempted.

- Obviously, a **non preemptive kernel is essentially free from race conditions** on kernel data structures, as only one process is active in the kernel at a time.

- **Preemptive kernels** are especially **difficult to design** for SMP architectures, since in these environments it is possible for two kernel-mode processes to run simultaneously on different processors.

1.Peterson's Solution

- We illustrate a classic software-based solution to the critical-section problem known as **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 . For convenience, when presenting Pi , we use P j to denote the other process; that is, j equals 1 – i

```
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.

Peterson's solution requires the two processes to share two data items:

```
int turn;
boolean flag[2];
```

We now prove that this solution is correct. We need to show that:

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

2. Synchronization Hardware

- Hardware instructions that are available on many systems and showing how they can be used effectively in solving the critical-section problem
- All these solutions are based on the premise of **locking** —that is, protecting critical regions through the use of locks.

Multi-processor environment

- Hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words **atomically**
- we abstract the main concepts behind these types of instructions by describing the **test and set()** and **compare and swap()** instructiaons.

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

Figure 5.3 The definition of the `test_and_set()` instruction.

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

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

Figure 5.4 Mutual-exclusion implementation with `test_and_set()`.

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

Figure 5.5 The definition of the `compare_and_swap()` instruction.

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

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

Figure 5.6 Mutual-exclusion implementation with the `compare_and_swap()` instruction.

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;

      /* critical section */

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = false;
   else
      waiting[j] = false;

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

Figure 5.7 Bounded-waiting mutual exclusion with `test_and_set()`.

3.Mutex Locks

- A process must **acquire the lock before entering a critical section**; it releases the lock when it exits the critical section.
- We use the mutex lock to protect critical regions and thus prevent race conditions.

- This type of mutex lock is also called a **spinlock ,** because the process "spins" while waiting for the lock to become available.

```
do {
        acquire lock

            critical section

        release lock

            remainder section

} while (true);
```

Figure 5.8 Solution to the critical-section problem using mutex locks.

The definition of `acquire()` is as follows:

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;;
}
```

The definition of `release()` is as follows:

```
release() {
   available = true;
}
```

REFERENCES

TEXT BOOKS:

T1 Silberschatz, Galvin, and Gagne, “Operating System Concepts”, Ninth Edition, Wiley India Pvt Ltd, 2009.)

T2. Andrew S. Tanenbaum, “Modern Operating Systems”, Fourth Edition, Pearson Education, 2010

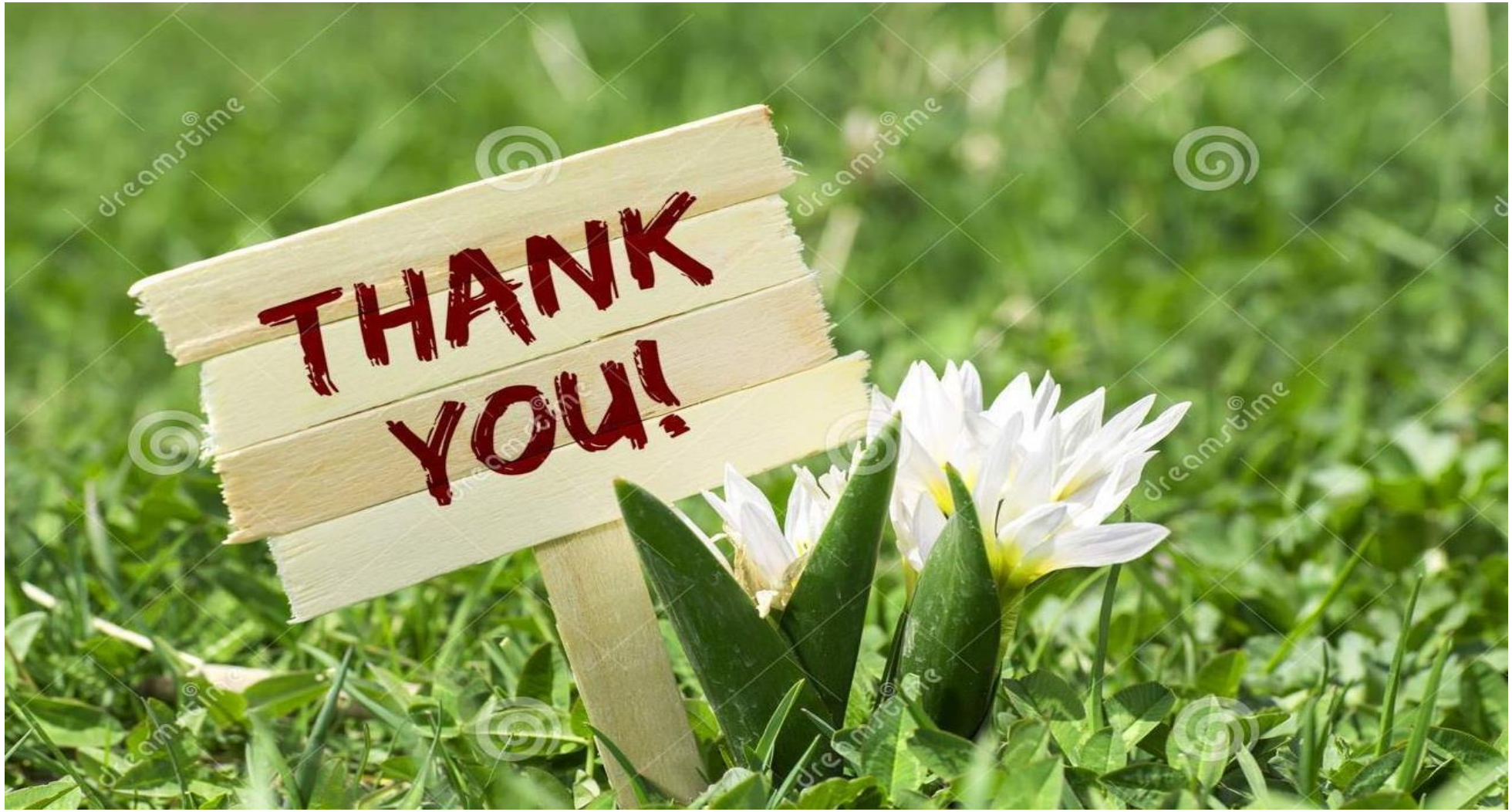
REFERENCES:

R1 Gary Nutt, “Operating Systems”, Third Edition, Pearson Education, 2004.

R2 Harvey M. Deitel, “Operating Systems”, Third Edition, Pearson Education, 2004.

R3 Abraham Silberschatz, Peter Baer Galvin and Greg Gagne, “Operating System Concepts”, 9th Edition, John Wiley and Sons Inc., 2012.

R4. William Stallings, “Operating Systems – Internals and Design Principles”, 7th Edition, Prentice Hall, 2011


THANK
YOU!