CHANDIGARH UNIVERSITY

Discover. Learn. Empower.

UNIVERSITY INSTITUTE OF ENGINEERING

Bachelor of Engineering (Computer Science & Engineering)

Operating System (CST-328)

Subject Coordinator: Er. Puneet kaur(E6913)

Lecture 8
Process Synchronization

- Process Synchronization
- Race Conditions
- Critical Section Problem
- Exercise
- Petersons solution
- Hardware implementation
- Mutex

Process Synchronization

- On the basis of synchronization, processes are categorized as one of the following two types:

- **Independent Process** : Execution of one process does not affects the execution of other processes.
- **Cooperative Process** : Execution of one process affects the execution of other processes.
- Reason is Modularity, Computation speed, information sharing

- Process synchronization problem arises in the case of Cooperative process also because resources are shared in Cooperative processes.

Race Condition

- When more than one processes are 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 so for that all the processes doing race to say that my output is correct
- This condition known as **race condition**.
- 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 typically occurs when two or more threads try to read, write and possibly make the decisions based on the memory that they are accessing concurrently.

Race Condition-Example

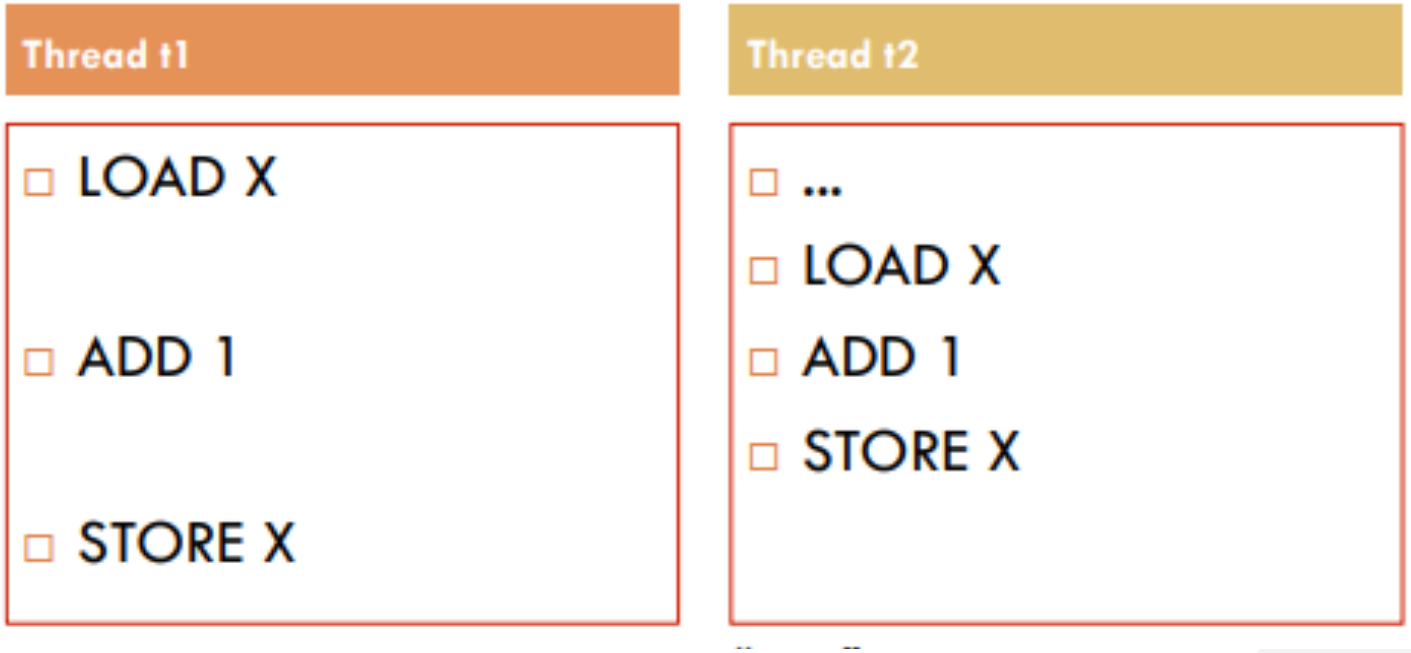
- Example:
- Suppose threads t1 and t2 simultaneously execute the statement x = x + 1;

  for some static global variable x.

- Internally, this involves loading x, adding 1, storing x .
- If t1 and t2 do this concurrently, we execute the statement twice, but x may only be incremented once.
- Here t1 and t2 “race” to do the update

Race Condition-Example

- Suppose X is initially 5



- After finishing, X=6! We “lost” an update
- t1 and t2 “race” to do the update

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.

- 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.

Critical Section Problem

- 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**.

Critical Section Problem

- The general structure of a typical process Pi is shown as:

```
do
{
        entry section
                critical section
        exit section
                remainder section

}while (true);
```

Critical Section Problem

- 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.

Critical Section Problem

- 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 pre-empted while it is running in kernel mode.
- A **non preemptive** 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.

- Obviously,a **non preemptive** kernel is essentially free from race conditions, as only one process is active in the kernel at a time.
- **Preemptive** kernel must be carefully designed to ensure that shared kernel data are free from race conditions.

Critical Section Problem

- Why would any one favour a preemptive kernel over a non preemptive one?
- A preemptive kernel may be more responsive, since there is less risk that a kernel-mode process will run for an arbitrarily long period before relinquishing the processor to waiting processes.
- Furthermore, a preemptive kernel is more suitable for real-time programming, as it will allow a real-time process to pre-empt a process currently running in the kernel.

Petersons 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 Pj to denote the other process; that is, j equals 1−i.

- 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.

Petersons Solution

- To enter the critical section, process Pi first sets flag[i] to be true and then sets turn to the value j, there by 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.

Petersons Solution

- Structure of Pi in Petersons Solution:

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

}while (true);
```

Hardware Implementaion

- Hardware features can make any programming task easier and improve system efficiency.
- The critical-section problem could be solved simply in a single-processor environment if we could prevent interrupts from occurring while a shared variable was being modified.
- In this way, we could be sure that the current sequence of instructions would be allowed to execute in order without pre-emption. No other instructions would be run, so no unexpected modifications could be made to the shared variable.
- This is often the approach taken by non pre-emptive kernels.
- But it is not good for multi-processing systems.

Hardware Implementaion

- test and set() and compare and swap() instructions are used to solve critical section problem.

- The test and set() instruction which executed atomically can be defined as :

```
boolean test and set(boolean *target) {
        boolean rv = *target;
        *target = true;
        return rv;
 }
```

Hardware Implementaion

- Mutual Exclusion Implementation with test and set()

```
do {
        while (test and set(&lock) ;
                /* do nothing */

                /* critical section */

        lock = false;

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

Hardware Implementaion

- The compare and swap() instruction (executes atomically), in contrast to the test and set() instruction, operates on three operands; it is defined as:

```
int compare and swap(int *value, int expected, int new value) {
        int temp = *value;
        if (*value == expected)
                *value = new value;
        return temp;
}
```

- The operand value is set to new value only if the expression (*value == expected) is true. Regardless, compare and swap() always returns the original value of the variable value.

Hardware Implementaion

- Mutual Exclusion Implementation with compare and swap()

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

                /* critical section */

        lock = 0;

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

Hardware Implementaion

- The above implementation is not valid for bounded waiting. So for it new data structures are used as:

- boolean waiting[n];
- boolean lock;

- These data structures are initialized to false.

Hardware Implementaion

- Mutual Exclusion Implementation and bounded waiting with test and set()

```
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);
```

Mutex Locks

- The hardware-based solutions to the critical-section problem complicated as well as generally inaccessible to application programmers.
- Instead, operating-systems designers build software tools to solve the critical-section problem.
- The simplest of these tools is the mutex lock. (short for mutual exclusion.)
- We use the mutex lock to protect critical regions and thus prevent race conditions. That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section.
- The acquire() function acquires the lock, and the release() function releases the lock.
- A mutex lock has a boolean variable available whose value indicates if the lock is available or not. If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable.
- A process that attempts to acquire an unavailable lock is blocked until the lock is released.

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;
  }
```

Solution with mutex locks

```
do {
        acquire lock
        critical section
        release lock
        remainder section

   }while (true);
```

Spin Locks

- The main disadvantage is that it requires busy waiting.
- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire().
- In fact, this type of mutex lock is also called a spinlock because the process “spins” while waiting for the lock to become available.
- This continual looping is clearly a problem in a real multi-programming system, where a single CPU is shared among many processes.
- Busy waiting wastes CPU cycles that some other process might be able to use productively.

Spin Locks

- Spinlocks do have an advantage, however, in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time.
- Thus, when locks are expected to be held for short times, spinlocks are useful.
- They are often employed on multi-processor systems where one thread can “spin” on one processor while another thread performs its critical section on another processor.

Conclusion

This Lecture enables you to understand Process synchronization, race condition, solution to critical section problem, use of mutex locks and spin locks.

References

https://www.includehelp.com/c-programming-questions/

https://www.studytonight.com/operating-system/

https://computing.llnl.gov/tutorials/

https://www.tutorialspoint.com/operating_system/index.htm#:~:text=An%20operating%20system%20(OS)%20is,software%20in%20a%20computer%20system.

https://www.javatpoint.com/os-tutorial

https://www.guru99.com/operating-system-tutorial.html

https://www.geeksforgeeks.org/operating-systems/