Chap 4: Process Synchronization

BY PRATYUSA MUKHERJEE, ASSISTANT PROFESSOR (I)

KIIT DEEMED TO BE UNIVERSITY

Background

- Case 1: Processes can execute concurretly or in parallel. The CPU scheduler switches rapidly between processes to enable concurrent environment. Thus, while a process is only partially complete, another process can be scheduled. Hence a process may be interrupted at any point of time.
- Case 2: During parallel execution, two different processes can run simltaneously on different processors.
- Thus, a process can be affected by other processes. They are known as cooperating processes. They can either share both code and data or be allowed to share data through files or messages
- Such concurrentaccess to shared data may lead to data inconsistency and loss of data integrity.

Producer - Consumer Problem

- Consider a producer problem that fills all the buffers.
- We can have an integer counter that keeps track of the number of full buffers.
- Initially, counter is set to 0.
- It is incremented by the producer after it produces a new buffer.
- It is decremented by the consumer after it consumes a buffer.

Producer Side

```
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 Side

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

Race Condition

counter++ could be implemented as

```
    register1 = counter
    register1 = register1 + 1
    counter = register1
```

counter-- could be implemented as

```
    register2 = counter
    register2 = register2 - 1
    counter = register2
```

Consider this execution interleaving with "counter = 5" initially:

S0: producer execute register1 = counter	{register1 = 5}
S1: producer execute register1 = register1 + 1	{register1 = 6}
S2: consumer execute register2 = counter	{register2 = 5}
S3: consumer execute register2 = register2 – 1	{register2 = 4}
S4: producer execute counter = register1	{counter = 6 }
S5: consumer execute counter = register2	{counter = 4}

Thus your final counter value is 4!

Do you think it is correct ??

What will be value of counter if order of S4 and S5 are interchanged ??

Race Condition contd.

- We arrived at incorrect value of counter as we allowed both processes to manipulate it simultaneously.
- A situation like this, 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 known as **race condition**

Need of Synchronization??

- To guard against the race condition, we must ensure that only one process at a time can manipulate the particular shared variable.
- To make such a guarantee, we require that the processes are synchronized in some way.
- Thus whenever processes run concurretly, we must make sure that any changes on shared data / variable by one process does not interfere with the working of other processes.

Critical Section Problem

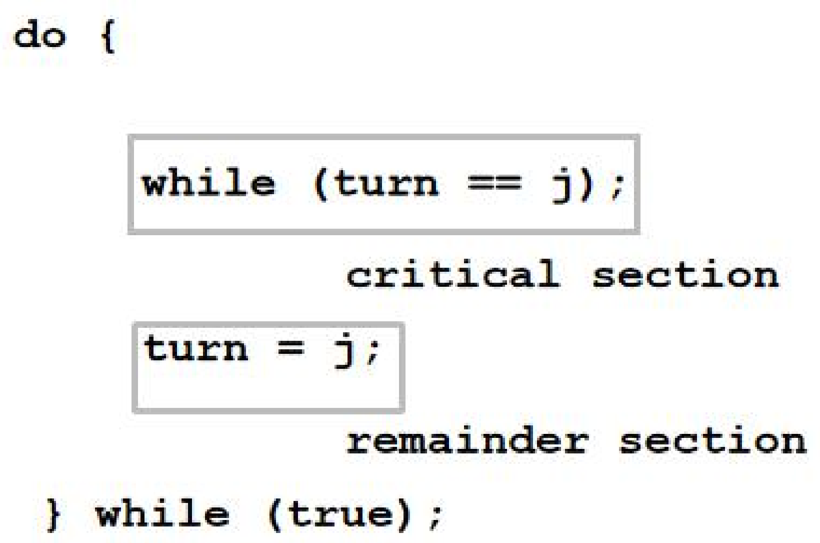
- Consider system of n processes {P_0, P_1, … ,P_{n-1}}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- **Critical section problem** is to design protocol to solve this
- Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section

Structure of Critical Section

General structure of process P_i

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

Algorithm for Process P_i



```
do {

     while (turn == j);

              critical section

     turn = j;

              remainder section

 } while (true);
```

Requirements for Solution to Critical-Section Problem

- **Mutual Exclusion** - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections
- **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
- **Bounded Waiting** - A bound must exist 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
- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the n processes

Approaches to handle Critical Section

We have 2 approaches depending on whether the kernel is preemptive or non-preemptive.

- **Preemptive Kernel**: It allows a process to be preempted while it is running in kernel mode.
- **Non-Preemptive Kernel**: It does not allow a process to be preempted while it is running in kernel mode. Thus in this case, a kernel mode process will run until it exceeds the kernel mode, gets blocked or voluntarily hands over the control of CPU.

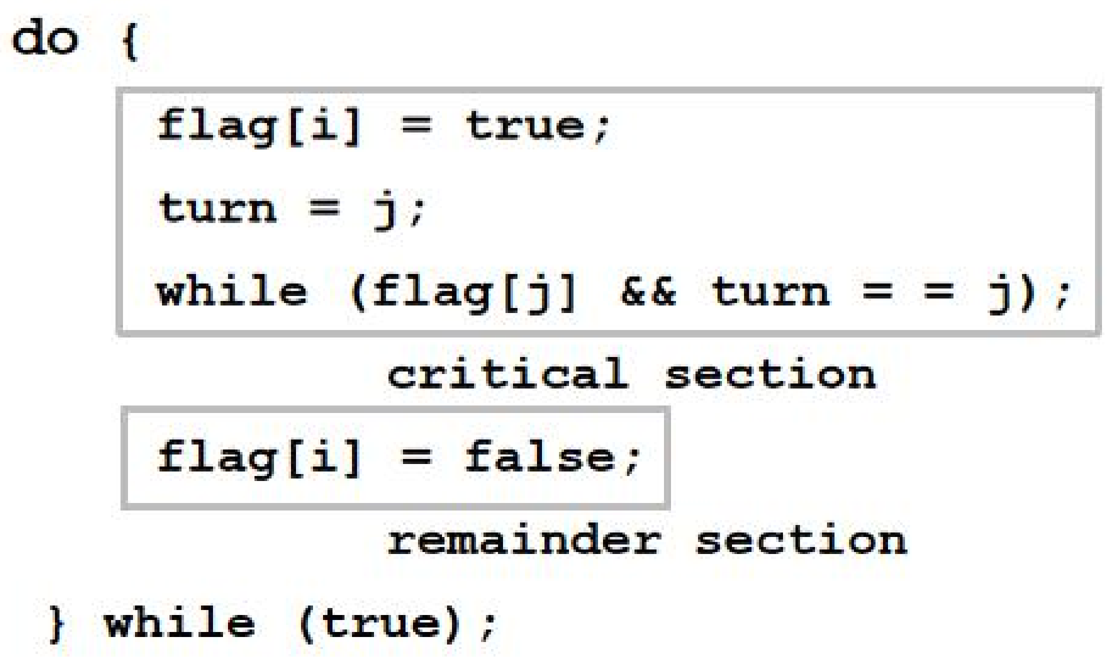
Amongst these which one do you think is prone to race condition and why??

Which mode will you prefer??

Peterson’s Solution

- This is a **classic software based solution** to critical section problem.
- Peterson’s Solution is restricted to **2 processes** that alternate execution between their critical section and remainder section. Let these processes be P_0 and P_1
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable **turn** indicates whose turn it is to enter the critical section
- The **flag** array is used to indicate if a process is ready to enter the critical section.

Algorithm for Peterson’s Solution



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

Can you justify if Peterson’s Solution is fullfilling all the requirements for a solution to Critical Section Problem??

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- Here solutions to protect Critical Section is based on idea of locking
- Uniprocessors systems could disable interrupts and ensure that only currently running code would execute without preemption
- However, this method is too inefficient on multiprocessor systems as an operating systems using this is not broadly scalable
- Modern machines provide special atomic hardware instructions. Atomic means non-interruptible.
- Thus Either test memory word and set value are utilised Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

```
do {
      acquire lock
               critical section
      release lock
               remainder section
} while (TRUE);
```

Does it satisfy all the requirements of a Solution to Critical Section??

test_and_set () Instruction

```
boolean test_and_set (boolean *target)
  {
       boolean rv = *target;
       *target = TRUE;
       return rv:
  }
```

Thus it is an atomic instruction. It returns the original value of passed parameter as well as sets the new value of passed parameter to “TRUE”.

Solution to Critical-section Problem Using test_and_set ()

Initially, shared Boolean variable lock, initialized to FALSE

```
do {
    while (test_and_set(&lock))
        ; /* do nothing */
            /* critical section */
    lock = false;
            /* remainder section */
 } while (true);
```

further explained on white board

HW: Discuss how is this solution fullfilling all the requirements of the Solution to CSP

compare_and_swap() Instruction

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

    if (*value == expected)
       *value = new_value;
 return temp;
}
```

It is an atomic instruction working on 3 operands. Returns the original value of passed parameter “value”. It sets the variable “value” the value of the passed parameter “new_value” but only if “value” ==“expected”. That is, the swap takes place only under this condition.

Solution to Critical-section Problem Using compare_and_swap ()

Initially, shared integer “lock” initialized to 0.

```
do {
     while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */
   /* critical section */
lock = 0;
   /* remainder section */
} while (true);
```

further explained on white board

Mutex

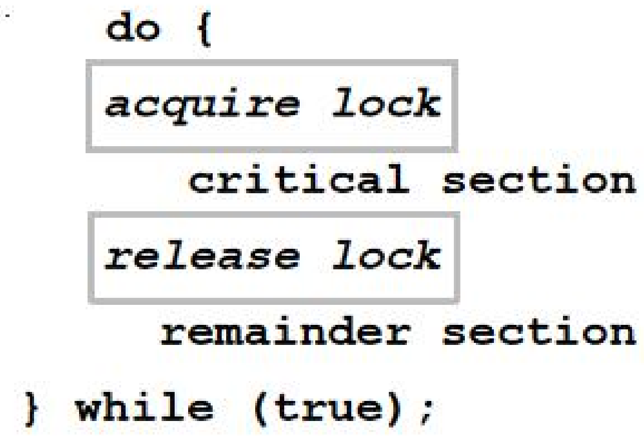
- Previous hardware based solutions are complicated and generally inaccessible to application programmers
- OS designers therefore built software tools to solve critical section problem
- Simplest is **mutex lock. (**derived from **mut**ual **ex**clusion**)**
- Protect a critical section by first **acquire()** a lock then **release()** the lock. Basically, it has a Boolean variable **available** which indicates if lock is available or not
- Calls to **acquire()** and **release()** must be atomic. Usually implemented via hardware atomic instructions discussed before.
- But this solution requires **busy waiting.** This lock therefore called a **spinlock**

Advantage and disadvantage of Spinlock?

acquire() and release()

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

   available = false;

}
```

```
release() {

   available = true;

}
```



```
    do {
      acquire lock
          critical section
      release lock
         remainder section
} while (true);
```