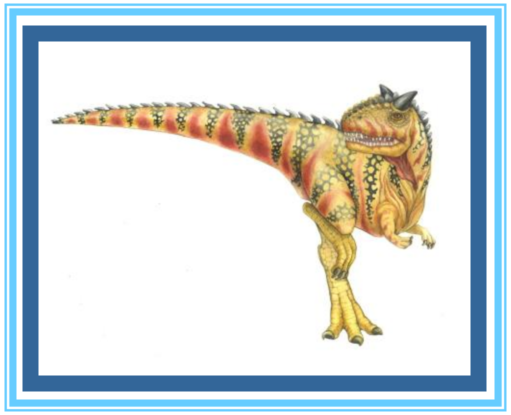
Chapter 5: Process Synchronization

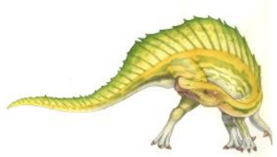


Background

- Processes can execute concurrently
  - May be interrupted at any time due to context switch, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

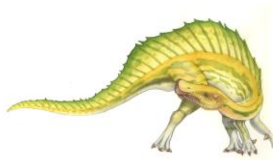
Illustrating the problem

- Revisit the Producer-Consumer problem
- Suppose we want to provide a solution that fills ***all*** the buffers.
- We can do so by having 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 and is decremented by the consumer after it consumes a 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 */
}
```

Race Condition

- **counter++** could be implemented as

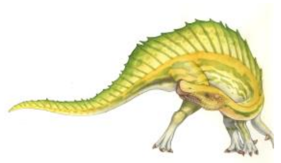
```
register1 = counter              LOAD R1,[count]
register1 = register1 + 1        ADDI R1, R1, 1
counter = register1              STORE R1, [count]
```

- **counter--** could be implemented as

```
register2 = counter              LOAD R2, [count]
register2 = register2 - 1        SUBI R2, R2, 1
counter = register2              STORE R2, [count]
```

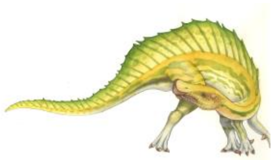
- Consider this execution interleaving with "count = 5" initially:

```
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}
```



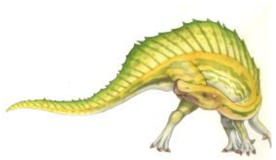
Race Condition

- Concurrent execution of parent process and child process may result in race condition
- Race condition: outcome of execution depends on the exact order in which the instructions (spread over multiple processes) get executed
- Need process synchronization to avoid race conditions



Critical Section Problem

- Consider system of ***n* cooperating** processes {$\boldsymbol{p_0, p_1, \ldots p_{n-1}}$} that access shared data
- Each process has a **critical section** segment of code
  - Process may be modifying shared data (e.g., update common variables, updating table, writing file, etc.)
- Basic requirement:
  - While one process is executing in its critical section, no other process is allowed to execute in its critical section
- ***Critical section problem*** is to design a protocol to solve this



Critical Section

- General structure of process ***P**$_i$*

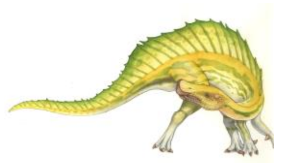
```
do {

        entry section

            critical section

        exit section

            remainder section

} while (true);
```

Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Solution to Critical-Section Problem

A correct to the critical section problem must satisfy these conditions:

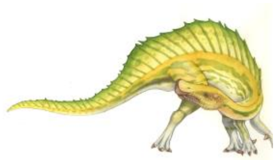
1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections (even if P_i is not assigned the CPU at this point of time)
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process P_i 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



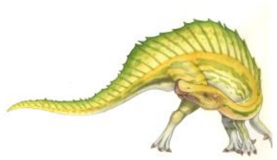
Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows a process to be preempted while it is running in kernel mode
- **Non-preemptive** – does not allow a process running in kernel mode to be preempted
  - Process runs until it exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode
- Preemptive kernels require very careful design; more suitable for real-time systems



Solutions to the Critical Section Problem

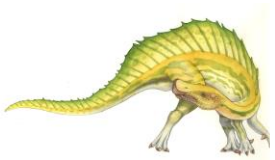


Peterson's Solution

- An algorithmic (software-based) solution to the control section problem
- Restricted to two processes that alternate execution between their critical section and remainder section
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted

- The two processes share two variables:
  - **`int turn             // usually initialized to 0`**
  - **`Boolean flag[2]   // initialized to both false`**

- 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. **`flag[i] = true`** implies that process P_i is ready



Algorithm for Process P_i

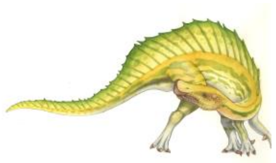
```
do {
     flag[i] = true;
     turn = j;
     while (flag[j] && turn == j);
               critical section
     flag[i] = false;
               remainder section
 } while (true);
```

entry section

exit section

•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. **`flag[i] = `*`true`*** implies that process P_i is ready



Algorithms for individual processes

P_0

```
do{
 flag[0] = true;
 turn = 1;
 while(flag[1] && turn == 1);
     critical section
 flag[0] = false;
     remainder section
 } while (true);
```

P_1

```
do {
 flag[1] = true;
 turn = 0;
 while(flag[0] && turn == 0);
     critical section
 flag[1] = false;
     remainder section
 } while (true);
```

•The variable **`turn`** indicates whose turn it is to enter the critical section

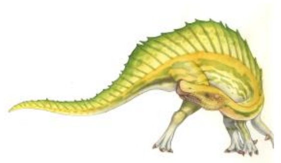
•The **`flag`** array is used to indicate if a process is ready to enter the critical section. **`flag[i] = `*`true`*** implies that process ***P_i*** is ready!

Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

     P$_i$ enters CS only if:

     either **`flag[j] = false`** or **`turn = i`**
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met



Problem with Peterson's Solution

- Modern computer architectures allow out-of-order execution
- There is no guarantee that Peterson's solution will work correctly on such architectures
- E.g., assume the first two statements before the inner while loop (marked in red below) execute out of order

P_0

```
do{
   flag[0] = true;
   turn = 1;
   while (flag[1] && turn == 1);
   [critical section]
```

P_1

```
do{
   flag[1] = true;
   turn = 0;
   while (flag[0] && turn == 0);
   [critical section]
```

Problem with Peterson's Solution (cont.)

P_0

```
do{
   flag[0] = true;
   turn = 1;
   while (flag[1] && turn == 1);
   [critical section]
```

P_1

```
do{
   flag[1] = true;
   turn = 0;
   while (flag[0] && turn == 0);
   [critical section]
```

- For the following order of execution, both P0 and P1 may enter their critical section simultaneously

P1: turn = 0

P0: turn = 1

P0: flag[0] = true

P0: while (…); Comes out of while loop, since flag[1] is false

P0 enters critical section

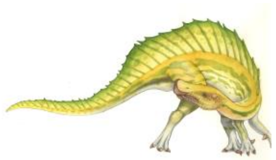
P1: flag[1] = true

P1: while (…); Comes out of while loop, since turn is 1

P1 enters critical section

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions to be discussed next based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value
  - 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);
```

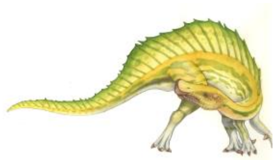
General structure of the solution

test_and_set Instruction

Definition:

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

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.



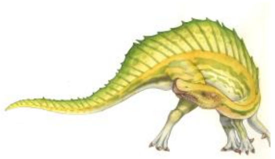
test_and_set Instruction

Definition:

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

1. For ease of explanation, we have abstracted the functionality of the TAS instruction as a C-like function
2. In practice, there can be a single low-level instruction
3. E.g., in MIPS, we can have a machine instruction such as

TAS R1, 0(1000) $\Longrightarrow$ $\boxed{\begin{array}{l}\text{R1 = M[1000]}\\ \text{M[1000] = 1}\end{array}}$



Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE (indicates unlocked)
- Solution:

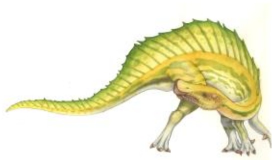
```
do {
    while (test_and_set(&lock))
        ; /* do nothing */
            /* critical section */
    lock = false;
            /* remainder section */
 } while (true);
```

compare_and_swap Instruction

Definition:

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

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

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set 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.



compare_and_swap Instruction

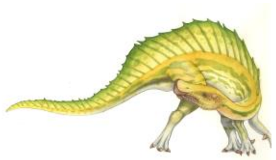
Definition:

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

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

1. For ease of explanation, we have abstracted the functionality as a C-like function
2. In practice, there can be a single low-level instruction
3. E.g., in MIPS, we can have a machine instruction such as

CAS R1, R2, 0(1000) $\Longrightarrow$

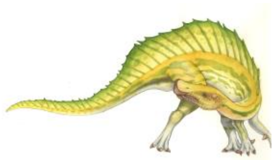
```
if (R1 ==  M[1000])
   M[1000] = R2
```



Solution using compare_and_swap

- Shared integer “lock” initialized to 0
- Solution:

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

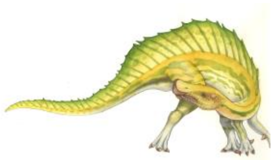


Bounded waiting requirement

- We discussed multiple requirements for any solution to the Critical Section Problem
  - Mutual Exclusion
  - Progress
  - Bounded Waiting

- The hardware solutions discussed till now satisfy Mutual Exclusion, but may not satisfy Bounded Waiting

- Next slide shows another algorithm using test_and_set() that satisfies all critical section requirements



Bounded-waiting Mutual Exclusion 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);
```

How to show this algorithm satisfies all Critical Section requirements?

See details in book