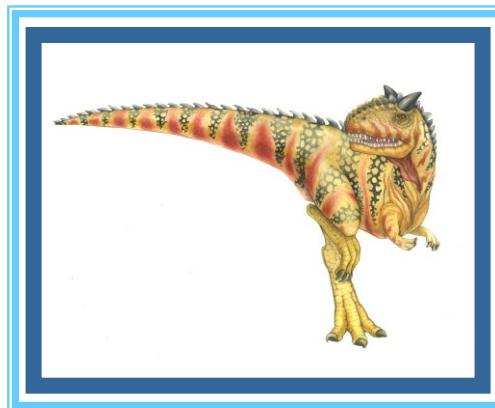


# Chapter 5: Process Synchronization





# Objectives

---

- To present the concept of process synchronization.
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems





# Background

---

- Processes can execute concurrently
  - May be interrupted at any time, 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
- Illustration of the problem:  
Suppose that we wanted to provide a solution to the consumer-producer problem 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 */ if Buffer Full--> wait until for Empty slot  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++; // 1 item increased in Buffer  
}
```

Shared resources: Counter--> no of Item in Buffer , Buffer Local variable:

IN --> Next Empty Slot in buffer (0 to n-1)

OUT --> Next Item in bufer to be Consumed (0 to n-1)

Item insert=> from 0 to n-1 => in= (in+1)%n

item remove=> from 0 to n-1 => out= (out+1)%n

Load Rp, m[counter] -->3  
INCR Rp-->4  
--- context swich (preEmptive) --

Store m[counter], Rp -->couldn't execute counter-->still 3

no of items in buffer-->4 but will show 3--> race condition





# Consumer

```
while (true) {      if Buffer Empty =>wait for Item to be produced in Producer  
                    program  
    while (counter == 0)  
        ; /* do nothing */  
  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--; // 1 item decreased from buffer.  
    /* consume the item in next consumed */  
}  
Load Rc, m[counter]-->3  
DECR Rc-->2  
Store m[counter], Rc-->2
```

now no of items in Buffer  
actually 3 , but showing  
2---> race condition occur





# Race Condition

- A situation 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 called a race condition.
- Cooperating processes need to be synchronized to avoid race condition.
- Producer and Consumer are cooperating processes.
- Concurrent execution of
  - “counter++” in producer
  - “counter--” in consumercan result in race condition.





# 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 “count = 5” initially:

S0: producer execute <code>register1 = counter</code>	{register1 = 5}
S1: producer execute <code>register1 = register1 + 1</code>	{register1 = 6}
S2: consumer execute <code>register2 = counter</code>	{register2 = 5}
S3: consumer execute <code>register2 = register2 - 1</code>	{register2 = 4}
S4: producer execute <code>counter = register1</code>	{counter = 6 }
S5: consumer execute <code>counter = register2</code>	{counter = 4}





# Critical Section Problem

---

- Consider system of  $n$  processes  $\{p_0, p_1, \dots p_{n-1}\}$
- 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**





# Critical Section

- General structure of process  $P_i$

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





# Algorithm for Process P<sub>i</sub>

```
do {  
  
    while (turn == j);  
  
        critical section  
  
    turn = j;  
  
        remainder section  
  
} while (true);
```





# Solution to Critical-Section Problem

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** - 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





# Critical-Section Handling in OS

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

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - ▶ Essentially free of race conditions in kernel mode





# Peterson's Solution

---

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- 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. **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);
```





# Algorithm for Process P<sub>j</sub>

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





# Correctness of Peterson's Solution

- 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





# Peterson's Solution and Modern Architecture

---

- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures.
  - To improve performance, processors and/or compilers may reorder operations that have no dependencies
- Understanding why it will not work is useful for better understanding race conditions.
- For single-threaded this is ok as the result will always be the same.
- For multithreaded the reordering may produce inconsistent or unexpected results!





# Modern Architecture Example

- Two threads share the data:

```
boolean flag = false;  
int x = 0;
```

- Thread 1 performs

```
while (!flag)  
;  
print x
```

- Thread 2 performs

```
x = 100;  
flag = true
```

- What is the expected output?

100





# Modern Architecture Example (Cont.)

- However, since the variables `flag` and `x` are independent of each other, the instructions:

```
flag = true;  
x = 100;
```

for Thread 2 may be reordered

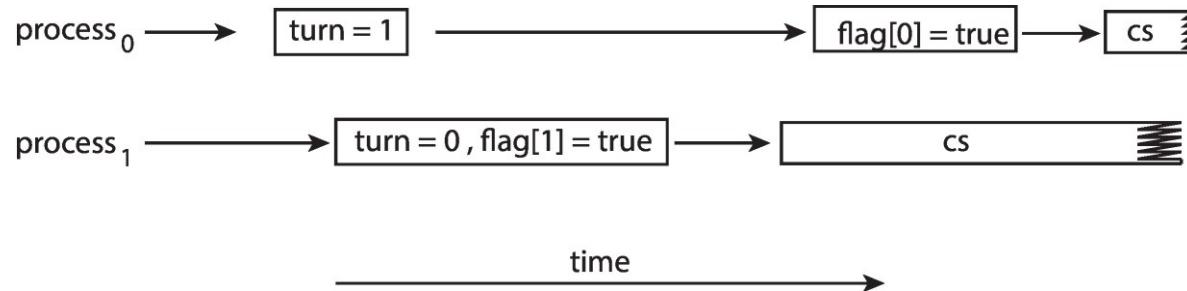
- If this occurs, the output may be 0!





# Peterson's Solution Revisited

- The effects of instruction reordering in Peterson's Solution



- This allows both processes to be in their critical section at the same time!
- To ensure that Peterson's solution will work correctly on modern computer architecture we must use **Memory Barrier**.





# Synchronization Hardware

---

- Many systems provide hardware support for implementing the critical section code.
- All solutions below 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);
```





# Test and Set Lock

---

- A hardware solution to the synchronization problem.
- There is a shared lock variable which can take either of the two values, 0 (FALSE) or 1(TRUE).
- Before entering into the critical section, a process inquires about the lock.
- If it is locked, it keeps on waiting till it becomes free.
- lock = 0/FALSE -> Unlocked
- lock = 1/TRUE -> Locked





# 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”.





# Solution using test\_and\_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

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





# Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
  - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First **acquire()** a lock
  - Then **release()** the lock
- Calls to **acquire()** and **release()** must be **atomic**
  - Usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**





# Solution to CS Problem Using Mutex Locks

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





# Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **wait()** and **signal()**
    - ▶ Originally called **P()** and **V()**

- Definition of the **wait()** operation

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}
```

- Definition of the **signal()** operation

```
signal(S) {  
    S++;  
}
```





# Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider  $P_1$ , and  $P_2$  that require  $S_1$  to happen before  $S_2$   
Create a semaphore “**synch**” initialized to 0

**P1:**

```
S1;  
    signal(synch);
```

**P2:**

```
wait(synch);  
S2;
```

- Can implement a counting semaphore **S** as a binary semaphore





# Semaphore Implementation

- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the `wait` and `signal` code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - ▶ But implementation code is short
    - ▶ Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution





# Semaphore Implementation with no Busy waiting

---

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue
- ```
typedef struct{
    int value;
    struct process *list;
} semaphore;
```





## Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {  
    S->value--;  
    if (S->value < 0) {  
        add this process to S->list;  
        block();  
    }  
}  
  
signal(semaphore *S) {  
    S->value++;  
    if (S->value <= 0) {  
        remove a process P from S->list;  
        wakeup(P);  
    }  
}
```





# Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let  $S$  and  $Q$  be two semaphores initialized to 1

|                         |                         |
|-------------------------|-------------------------|
| $P_0$                   | $P_1$                   |
| <code>wait(S);</code>   | <code>wait(Q);</code>   |
| <code>wait(Q);</code>   | <code>wait(S);</code>   |
| ...                     | ...                     |
| <code>signal(S);</code> | <code>signal(Q);</code> |
| <code>signal(Q);</code> | <code>signal(S);</code> |

- **Starvation – indefinite blocking**

- A process may never be removed from the semaphore queue in which it is suspended





# Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem





# Bounded-Buffer Problem

---

- $n$  buffers, each can hold one item
- Binary Semaphore **mutex** initialized to the value 1
- Counting Semaphore **full** initialized to the value 0
- Counting Semaphore **empty** initialized to the value  $n$





# Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {  
    ...  
    /* produce an item in next_produced */  
    ...  
    wait(empty);  
    wait(mutex);  
    ...  
    /* add next produced to the buffer */  
    ...  
    signal(mutex);  
    signal(full);  
} while (true);
```





# Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
Do {  
    wait(full);  
    wait(mutex);  
    ...  
    /* remove an item from buffer to next_consumed */  
    ...  
    signal(mutex);  
    signal(empty);  
    ...  
    /* consume the item in next consumed */  
    ...  
} while (true);
```





# Readers-Writers Problem

---

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do *not* perform any updates
  - Writers – can both read and write
- If two readers access the shared data simultaneously, no adverse affects will result.
- If a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.
- Writers must have exclusive access to the shared database while writing to the database.
- Shared Data
  - Data set
  - Semaphore `wrt` initialized to 1
  - Semaphore `mutex` initialized to 1
  - Integer `read_count` initialized to 0





# Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {  
    wait(wrt);  
  
    ...  
    /* writing is performed */  
  
    ...  
    signal(wrt);  
} while (true);
```





# Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {  
    wait(mutex);  
    read_count++;  
    if (read_count == 1)  
        wait(wrt);  
    signal(mutex);  
  
    ...  
    /* reading is performed */  
  
    ...  
    wait(mutex);  
    read_count--;  
    if (read_count == 0)  
        signal(wrt);  
    signal(mutex);  
} while (true);
```





# Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - ▶ Bowl of rice (data set)
    - ▶ Semaphore **chopstick [5]** initialized to 1





# Dining-Philosophers Problem Algorithm

- The structure of Philosopher  $i$ :

```
do {  
    wait (chopstick[i] ) ;  
    wait (chopStick[ (i + 1) % 5] ) ;  
  
        // eat  
  
    signal (chopstick[i] ) ;  
    signal (chopstick[ (i + 1) % 5] ) ;  
  
        // think  
  
} while (TRUE) ;
```

- What is the problem with this algorithm?





# Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section).
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

