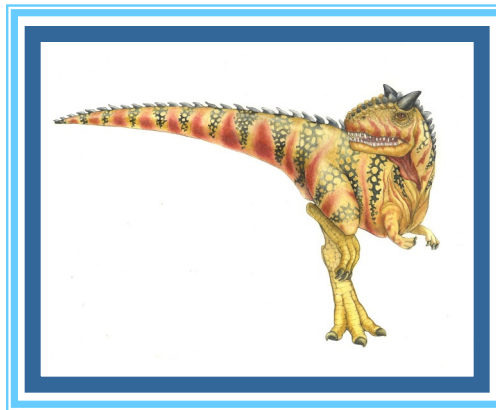


# Chapter 5: Process Synchronization

---





# 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 */  
    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
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 | <b>register1 = counter</b>       | {register1 = 5} |
| S1: producer execute | <b>register1 = register1 + 1</b> | {register1 = 6} |
| S2: consumer execute | <b>register2 = counter</b>       | {register2 = 5} |
| S3: consumer execute | <b>register2 = register2 - 1</b> | {register2 = 4} |
| S4: producer execute | <b>counter = register1</b>       | {counter = 6}   |
| S5: consumer execute | <b>counter = register2</b>       | {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 $P_i$

---

```
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 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 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





# 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);
```





# Peterson's Solution (Cont.)

---

- Provable that the three Critical Section(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





# 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
    - 4 Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
  - 4 **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 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);
```







# 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.





# 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);
```





# 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
- Protect a critical section by first **acquire()** a lock then **release()** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **acquire()** and **release()** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **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);
```





# Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations

- **wait()** and **signal()**

4 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 :

$S_1$ ;

**signal**(**synch**) ;

P2 :

**wait**(**synch**);

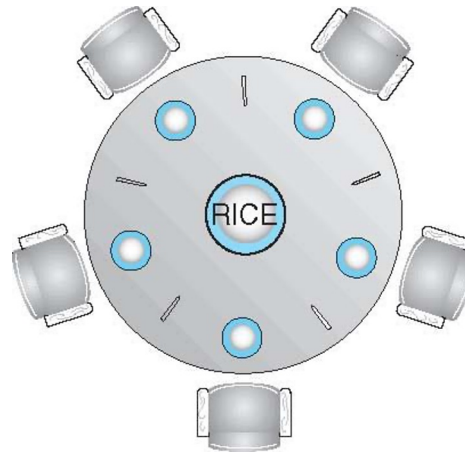
$S_2$ ;

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





# 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
    - 4 Bowl of rice (data set)
    - 4 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.





# Problem with dining philosopher algorithm

---

The problem arises when all five philosophers are hungry and try to pick up their left and right chopsticks at the same time. If this happens, each philosopher will be holding one chopstick and waiting for the other chopstick from their neighbor. This creates a deadlock situation, where no philosopher can eat and the system will eventually crash.



# End of Chapter 5

---

