# Chapter 6: Process Synchronization

# Chapter 6: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Alternative Approaches

# 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 */
    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

- 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
- **`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 `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} |

# Critical Section Problem

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

# Critical-Section Handling in OS

Two approaches are used to handle critical sections in OS

- **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 $P_i$

$P_i$

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

$P_j$

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

# Peterson's Solution (Cont.)

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

     $\mathbf{P_i}$ enters CS only if:

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

- Since $\mathbf{P_i}$ does not change the value of the variable turn while executing the while statement, $\mathbf{P_i}$ will enter the critical section (progress) after at most one entry by $\mathbf{P_j}$ (bounded waiting).

# 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 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);
```

# The definition of the Swap () instruction.

```
void Swap(boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
    }
```

1. It is executed atomically

# Mutual-exclusion implementation with the Swap() instruction.

```
do {
 key = TRUE;
 while (key == TRUE)
     Swap(&lock, &key);
     // critical section
 lock = FALSE;
     // remainder section
 } while (TRUE);
```

# Bounded-waiting Mutual Exclusion with test_and_set

- The data structures **`boolean waiting[n]; boolean lock;`** are initialized to **`false`**

```
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);
```

| | P0 | P1 | P2 | P3 |
|---|---|---|---|---|
| Waiting(i) | F | F | F | F |
| Key | T | | F | |

n=4
lock=T
i=2, j=0

# Cont'd

- 1: **Mutual Exclusion**
- 2: **Progress:** Since a process i exiting the critical section either sets lock to false or sets waiting[j] to false.
- 3: **Bounded waiting:** it scans the array waiting in the cyclic ordering (i + 1, i + 2, ... , n- 1, 0, ... , i -1)

# Semaphore

- Synchronization tool that provides more sophisticated ways for process 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
  - use:
    - Control access to a given resource consisting of a finite number of instances.
    - The semaphore is initialized to the number of resources available.
    - Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count).
    - When a process releases a resource, it performs a signal() operation (incrementing the count).
    - When the count for the semaphore goes to 0, all resources are being used.
    - After that, processes that wish to use a resource will block until the count becomes greater than 0

- **Binary semaphore** – integer value can range only between 0 and 1
  - Deal with the critical-section problem for multiple processes
  - 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

# Mutual-exclusion implementation with semaphores.

- Binary semaphore is implemented through mutex.
- The *n* processes share a semaphore, mutex, initialized to 1.
- Each process *Pi* is organized as

```
do {
    wait (mutex) ;
        // critical section
    signal(mutex);
        // remainder section
    } while (TRUE);
```

# 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

$P_0$ $P_1$

```
P0                      P1
wait(S);                wait(Q);
wait(Q);                wait(S);
...                     ...
signal(S);              signal(Q);
signal(Q);              signal(S);
```

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

# 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 in the buffer pool, each can hold one item
- Semaphore (Binary) **`mutex`** initialized to the value 1
- Semaphore (counting) **`full`** initialized to the value 0
- Semaphore (counting) **`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 until empty>0 and then decrement empty
    wait(mutex); //acquire lock
       ...
      /* add next produced to the buffer */
       ...
    signal(mutex); //release lock
    signal(full); // increment full
 } while (true);
```

# Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
Do {
    wait(full); // wait until full>0, then decrement full
    wait(mutex); // acquire lock
        ...
        /* remove an item from buffer to next_consumed */
        ...
    signal(mutex); // release lock
    signal(empty);// increment 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
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** 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(rw_mutex);
              ...
            /* writing is performed */
              ...
        signal(rw_mutex);
  } while (true);
```

# Readers-Writers Problem (Cont.)

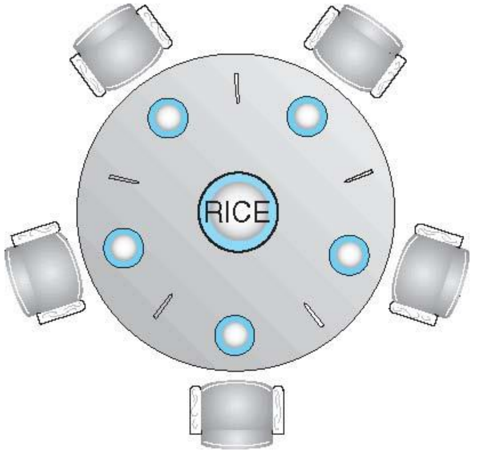
- The structure of a reader process

```
do {
        wait(mutex);
        read_count++;//no of readers has increase by 1
        if (read_count == 1)
        wait(rw_mutex);//no writer can enter even if 1 reader
    signal(mutex);// allows other reader to enter CS

         ...
        /* reading is performed */

         ...
    wait(mutex);// acquire lock to update shared variable
         read_count
        read_count--;
        if (read_count == 0) //no reader in CS
    signal(rw_mutex); //writers can enter in CS
    signal(mutex); // reader leaves
} while (true);
```

# Readers-Writers Problem Variations

- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

# 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.

# Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.

# Synchronization Examples

- Solaris
- Windows
- Linux
- Pthreads

# Windows Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted
- Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
  - **Events**
    - An event acts much like a condition variable
  - Timers notify one or more thread when time expired
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)

# Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - Semaphores
  - Atomic integers
  - spinlocks
  - reader-writer versions of both
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

# End of Chapter 6