Chapter 6: Synchronization

Module 6: Synchronization

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Atomic Transactions
Objectives

- 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 introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

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 count that keeps track of the number of full buffers.
- Initially, count 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**

```c
while (true) {
    /* produce an item and put in nextProduced */
    while (count == BUFFER_SIZE)
        ; // do nothing

    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}
```

**Consumer**

```c
while (true) {
    while (count == 0)
        ; // do nothing

    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;

    // consume the item in nextConsumed
}
```
Race Condition

- `count++` could be implemented as

  ```
  register1 = count
  register1 = register1 + 1
  count = register1
  ```

- `count--` could be implemented as

  ```
  register2 = count
  register2 = register2 - 1
  count = register2
  ```

  Consider this execution interleaving with “count = 5” initially:

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

A General Framework for Synchronization

the Critical-Section Problem

```c
do {
  permission request
  entry section;
  critical section;
  exit notification
  exit section;
  remainder section;
} while (1);
```

Assumptions:
- Atomic execution of each statement line
- Interleaving execution among processes
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

- Two process solution
- Assume that the LOAD and STORE 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 $P_i$

```c
do {
    1. flag[i] = TRUE;
    2. turn = 1-i;
    3. while (flag[1-i] && turn == 1-i);
    4. critical section
    5. flag[i] = FALSE;
    6. remainder section
} while (TRUE);
```

Peterson’s Solution

Proof for the mutual exclusion (1/2)

**Lemma:** When a Pi is in the entry and the critical sections, flag[i] = true.

For convenience, a state is denoted as [h,k,x,y,z] where h, k,x,y,z are respectively the statement index of P0, statement index of P1, value of turn, value of flag[0], and value of flag[1].

According to the lemma, we assume that [4,4,0,1,1] happens.

This implies that P0 enters the critical section last from [3,4,0,1,1].
Peterson’s Solution

Proof for the mutual exclusion (2/2)

There are two possibilities of the predecessor to [3,4,0,1,1].

- One possible predecessor of [3,4,0,1,1] is [3,3,0,1,1] which is impossible.
  - From [3,3,0,1,1], the while loop condition for P1 is false.

- The other possible predecessor of [3,4,0,1,1] is [2,3,?,1,1] which is also impossible.
  - From [2,3,?,1,1], statement 2 for P0 changes turn to 1 instead of 0.

Since both possibilities are contradictions, the assumption of violation of mutual exclusion is a contradiction.

Thus the lemma is proven. ■

Peterson’s Solution

Properties

- Mutual Exclusion
  - The eventual value of turn determines which process enters the critical section.

- Progress
  - A process can only be stuck in the while loop, and the process which can keep it waiting must be in its critical sections.

- Bounded Waiting
  - Each process wait at most one entry by the other process.
The critical-section problem
A solution for n processes

Bakery Algorithm
- Originally designed for distributed systems
- Token-based
  - Processes which are ready to enter their critical section must take a number and wait till the number becomes the lowest.
- Two arrays of local variables
  - int number[i]:
    - Pi’s token number if it is nonzero.
  - boolean choosing[i]:
    - Pi is taking a number.

```
do {
  choosing[i]=true;
  number[i]=max(number[0], ...number[n-1])+1;
  choosing[i]=false;
  for (j=0; j < n; j++) {
    while choosing[j] ;
    while (number[j] != 0 && (number[j],j)<(number[i],i)) ;
  }
  critical section
  number[i]=0;
  remainder section
} while (1);
```

An observation: If
- Pi is in its critical section, and
- Pk (k != i) has already chosen its number[k],
then (number[i],i) < (number[k],k).
Synchronization Hardware

- Many systems provide hardware support for critical section code
- 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-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

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

- Definition:

```c
bool TestAndSet (bool *target) {
    bool rv = *target;
    *target = TRUE;
    return rv;
}
```

Solution using TestAndSet

- Shared boolean variable lock, initialized to false.
- Solution:

```c
do {
    while (TestAndSet (&lock))
        ; // do nothing

    // critical section

    lock = FALSE;

    // remainder section
}
```
Swap Instruction

Definition:

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

Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key
- Solution:
  ```c
do {
    key = TRUE;
    while ( key == TRUE)
        Swap (&lock, &key );

    // critical section

    lock = FALSE;

    // remainder section
}
```
Bounded-waiting Mutual Exclusion with TestAndSet()

do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&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);
Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore $S$ as a binary semaphore
- Provides mutual exclusion
  
  ```
  Semaphore mutex;    // initialized to 1
  do {
    wait (mutex);
    // Critical Section
    signal (mutex);
    // remainder section
  } while (TRUE);
  ```

Semaphore Implementation

- Must guarantee that no two processes can execute `wait()` and `signal()` on the same semaphore at the same time
- Thus, 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.

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:
  ```c
  wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
      add this process to S->list;
      block();
    }
  }
  ```

- Implementation of signal:
  ```c
  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.

\[
\begin{align*}
&P_0 \quad P_1 \\
&\text{wait (S); wait (Q);}
&\text{wait (Q); wait (S);}
&\cdot 
&\cdot 
&\cdot 
&\cdot 
&\text{signal (S); signal (Q);}
&\text{signal (Q); signal (S);}
\end{align*}
\]

- **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.

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem
Bounded-Buffer Problem

- \( N \) buffers, each can hold one item
- Semaphore \texttt{mutex} initialized to the value 1
- Semaphore \texttt{full} initialized to the value 0
- Semaphore \texttt{empty} initialized to the value \( N \).

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```c
  do {
    // produce an item in nextp
    wait (empty);
    wait (mutex);
    // add the item 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 nextc
      signal (mutex);
      signal (empty);
      // consume the item in nextc
  } 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

- Shared Data
  - Data set
  - Semaphore `mutex` initialized to 1
  - Semaphore `wrt` initialized to 1
  - Integer `readcount` initialized to 0
Readers-Writers Problem (Cont.)

- The structure of a writer process

```c
    do {
        wait (wrt) ;
        // writing is performed
        signal (wrt) ;
    } while (TRUE);
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```c
    do {
        wait (mutex) ;
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
            wait (wrt) ;
        signal (mutex)
        // reading is performed
        wait (mutex) ;
        readcount -- ;
        if (readcount == 0)
            signal (wrt) ;
            signal (mutex) ;
    } while (TRUE);
```
Dining-Philosophers Problem

- Shared data
  - Bowl of rice (data set)
  - Semaphore \texttt{chopstick [5]} initialized to 1

Dining-Philosophers Problem (Cont.)

- The structure of Philosopher \( i \):

  \[
  \text{do } \{
  \text{wait (chopstick}[i] );}
  \text{wait (chopStick[ (i + 1) \% 5 ] );}

  \text{// eat}

  \text{signal (chopstick}[i] );}
  \text{signal (chopstick[ (i + 1) \% 5 ] );}

  \text{// think}

  \} \text{ while (TRUE);}
\]
Problems with Semaphores

- Correct use of semaphore operations:
  - `signal (mutex) .... wait (mutex)`
  - `wait (mutex) ... wait (mutex)`
  - Omitting of `wait (mutex)` or `signal (mutex)` (or both)

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```plaintext
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (...) { .... }
  ...

  procedure Pn (...) {.....}

  Initialization code { ....} { ... }
  ...
}
```
Condition Variables

- condition x, y;

- Two operations on a condition variable:
  - `x.wait()` – a process that invokes the operation is suspended.
  - `x.signal()` – resumes one of processes (if any) that invoked `x.wait()`
Monitor with Condition Variables

```c
monitor DP
{
    enum { THINKING, HUNGRY, EATING } state [5];
    condition self [5];

    void pickup (int i) {
        state[i] = HUNGRY;
        test();
        if (state[i] != EATING) self[i].wait;
    }

    void putdown (int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }
}
```

Solution to Dining Philosophers
void test (int i) {
    if ( (state[(i + 4) % 5] != EATING) &&
         (state[i] == HUNGRY) &&
         (state[(i + 1) % 5] != EATING) ) {
        state[i] = EATING ;
        self[i].signal () ;
    }
}

initialization_code() {
    for (int i = 0; i < 5; i++)
        state[i] = THINKING;
}
Monitor Implementation Using Semaphores

- Variables
  
  semaphore mutex;  // (initially = 1)
  semaphore next;   // (initially = 0)
  int next-count = 0;

- Each procedure \( F \) will be replaced by
  
  wait(mutex);
  ...
  body of \( F \);
  ...
  if (next_count > 0)
    signal(next)
  else
    signal(mutex);

- Mutual exclusion within a monitor is ensured.

Monitor Implementation

- For each condition variable \( x \), we have:
  
  semaphore x_sem; // (initially = 0)
  int x-count = 0;

- The operation \( x.wait \) can be implemented as:
  
  x-count++;
  if (next_count > 0)
    signal(next);
  else
    signal(mutex);
  wait(x_sem);
  x-count--;
Monitor Implementation

- The operation `x.signal` can be implemented as:

```
if (x-count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

A Monitor to Allocate Single Resource

```java
monitor ResourceAllocator {
    boolean busy;
    condition x:
    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = TRUE;
    }
    void release() {
        busy = FALSE;
        x.signal();
    }
    initialization code() {
        busy = FALSE;
    }
}
```
Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutexes for efficiency when protecting data from short code segments
- Uses condition variables and readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events
  - An event acts much like a condition variable

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
  - spin locks
Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spin locks

Atomic Transactions

- System Model
- Log-based Recovery
- Checkpoints
- Concurrent Atomic Transactions
System Model

- Assures that operations happen as a single logical unit of work, in its entirety, or not at all
- Related to field of database systems
- Challenge is assuring atomicity despite computer system failures
- **Transaction** - collection of instructions or operations that performs single logical function
  - Here we are concerned with changes to stable storage – disk
  - Transaction is series of **read** and **write** operations
  - Terminated by **commit** (transaction successful) or **abort** (transaction failed) operation
  - Aborted transaction must be **rolled back** to undo any changes it performed

Types of Storage Media

- **Volatile storage** – information stored here does not survive system crashes
  - Example: main memory, cache
- **Nonvolatile storage** – information usually survives crashes
  - Example: disk and tape
- **Stable storage** – Information never lost
  - Not actually possible, so approximated via replication or RAID to devices with independent failure modes

Goal is to assure transaction atomicity where failures cause loss of information on volatile storage
Log-Based Recovery

- Record to stable storage information about all modifications by a transaction
- Most common is write-ahead logging
  - Log on stable storage, each log record describes single transaction write operation, including
    - Transaction name
    - Data item name
    - Old value
    - New value
  - \(<T_i \text{ starts}>\) written to log when transaction \(T_i\) starts
  - \(<T_i \text{ commits}>\) written when \(T_i\) commits
- Log entry must reach stable storage before operation on data occurs

Log-Based Recovery Algorithm

- Using the log, system can handle any volatile memory errors
  - \(\text{Undo}(T_i)\) restores value of all data updated by \(T_i\)
  - \(\text{Redo}(T_i)\) sets values of all data in transaction \(T_i\) to new values
- \(\text{Undo}(T_i)\) and \(\text{redo}(T_i)\) must be idempotent
  - Multiple executions must have the same result as one execution
- If system fails, restore state of all updated data via log
  - If log contains \(<T_i \text{ starts}>\) without \(<T_i \text{ commits}>\), \(\text{undo}(T_i)\)
  - If log contains \(<T_i \text{ starts}>\) and \(<T_i \text{ commits}>\), \(\text{redo}(T_i)\)
Checkpoints

- Log could become long, and recovery could take long
- Checkpoints shorten log and recovery time.
- Checkpoint scheme:
  1. Output all log records currently in volatile storage to stable storage
  2. Output all modified data from volatile to stable storage
  3. Output a log record <checkpoint> to the log on stable storage
- Now recovery only includes Ti, such that Ti started executing before the most recent checkpoint, and all transactions after Ti. All other transactions already on stable storage.

Concurrent Transactions

- Must be equivalent to serial execution — serializability
- Could perform all transactions in critical section
  - Inefficient, too restrictive
- Concurrency-control algorithms provide serializability
# Serializability

- Consider two data items A and B
- Consider Transactions T₀ and T₁
- Execute T₀, T₁ atomically
- Execution sequence called schedule
- Atomically executed transaction order called serial schedule
- For N transactions, there are N! valid serial schedules

---

## Schedule 1: T₀ then T₁

<table>
<thead>
<tr>
<th>T₀</th>
<th>T₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>write(A)</td>
<td>write(A)</td>
</tr>
<tr>
<td>read(B)</td>
<td>read(B)</td>
</tr>
<tr>
<td>write(B)</td>
<td>write(B)</td>
</tr>
</tbody>
</table>
Nonserial Schedule

- **Nonserial schedule** allows overlapped execute
  - Resulting execution not necessarily incorrect
- Consider schedule S, operations O_i, O_j
  - **Conflict** if access same data item, with at least one write
- If O_i, O_j consecutive and operations of different transactions & O_i and O_j don’t conflict
  - Then S’ with swapped order O_j O_i equivalent to S
- If S can become S’ via swapping nonconflicting operations
  - S is **conflict serializable**

Schedule 2: Concurrent Serializable Schedule

<table>
<thead>
<tr>
<th>T_0</th>
<th>T_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>write(A)</td>
<td>write(A)</td>
</tr>
<tr>
<td>read(B)</td>
<td>read(B)</td>
</tr>
<tr>
<td>write(B)</td>
<td>write(B)</td>
</tr>
</tbody>
</table>
Locking Protocol

- Ensure serializability by associating lock with each data item
  - Follow locking protocol for access control
- Locks
  - **Shared** – T₁ has shared-mode lock (S) on item Q, T₁ can read Q but not write Q
  - **Exclusive** – T₁ has exclusive-mode lock (X) on Q, T₁ can read and write Q
- Require every transaction on item Q acquire appropriate lock
- If lock already held, new request may have to wait
  - Similar to readers-writers algorithm

Two-phase Locking Protocol

- Generally ensures conflict serializability
- Each transaction issues lock and unlock requests in two phases
  - **Growing** – obtaining locks
  - **Shrinking** – releasing locks
- Does not prevent deadlock
Timestamp-based Protocols

- Select order among transactions in advance – timestamp-ordering
- Transaction Ti associated with timestamp TS(Ti) before Ti starts
  - TS(Tj) < TS(Ti) if Ti entered system before Tj
  - TS can be generated from system clock or as logical counter incremented at each entry of transaction
- Timestamps determine serializability order
  - If TS(Ti) < TS(Tj), system must ensure produced schedule equivalent to serial schedule where Ti appears before Tj

Timestamp-based Protocol Implementation

- Data item Q gets two timestamps
  - W-timestamp(Q) – largest timestamp of any transaction that executed write(Q) successfully
  - R-timestamp(Q) – largest timestamp of successful read(Q)
  - Updated whenever read(Q) or write(Q) executed
- Timestamp-ordering protocol assures any conflicting read and write executed in timestamp order
- Suppose Ti executes read(Q)
  - If TS(Ti) < W-timestamp(Q), Ti needs to read value of Q that was already overwritten
    - read operation rejected and Ti rolled back
  - If TS(Ti) ≥ W-timestamp(Q)
    - read executed, R-timestamp(Q) set to max(R-timestamp(Q), TS(Ti))

Silberschatz, Galvin and Gagne ©2009
## Timestamp-ordering Protocol

Suppose Ti executes write(Q)

- If TS(Ti) < R-timestamp(Q), value Q produced by Ti was needed previously and Ti assumed it would never be produced
  - **Write** operation rejected, Ti rolled back
- If TS(Ti) < W-timestamp(Q), Ti attempting to write obsolete value of Q
  - **Write** operation rejected and Ti rolled back
- Otherwise, **write** executed

- Any rolled back transaction Ti is assigned new timestamp and restarted
- Algorithm ensures conflict serializability and freedom from deadlock

---

## Schedule Possible Under Timestamp Protocol

<table>
<thead>
<tr>
<th>T₂</th>
<th>T₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(B)</td>
<td>read(B)</td>
</tr>
<tr>
<td>read(A)</td>
<td>write(B)</td>
</tr>
<tr>
<td></td>
<td>read(A)</td>
</tr>
<tr>
<td></td>
<td>write(A)</td>
</tr>
</tbody>
</table>
End of Chapter 6