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
Background

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

- 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.
while (true)

    /* produce an item and put in nextProduced */
    while (count == BUFFER_SIZE)
        ; // do nothing
    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
while (1) {

    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: producer execute `count = register1` {count = 6}
S5: consumer execute `count = register2` {count = 4}
Race Condition (cont.)

- **Race condition**
  - The situation where several processes access and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.

- To prevent race conditions, concurrent processes must be **synchronized**.
The Critical-Section Problem

- $n$ processes all competing to use some shared data

- Each process has a code segment, called *critical section*, in which the shared data is accessed.

- Problem — ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
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.
Initial Attempts to Solve The Problem

- Only 2 processes, $P_0$ and $P_1$
- General structure of process $P_i$ (other process $P_i$)

\[
do \{
    \text{entry section}
    \text{critical section}
    \text{exit section}
    \text{remainder section}
\}\text{while (1);}
\]

- Processes may share some common variables to synchronize their actions.
Algorithm 1

- **Shared variables:**
  - `int turn;`
  - Initially `turn = 0`
  - `turn == i` \(\Rightarrow\) \(P_i\) can enter its critical section

- **Process \(P_i\):**
  - `do` {
    - `while (turn != i);`
    - critical section
    - `turn = j;`
    - remainder section
  } `while (1);`

- Satisfies mutual exclusion, but not progress
Algorithm 2

- Shared variables
  - boolean flag[2];
    initially flag[0] = flag[1] = false.
  - flag[i] = true \(\Rightarrow\) \(P_i\) ready to enter its critical section

- Process \(P_i\)
  
  do { 
    flag[i] := true;
    while (flag[i]);
    critical section
    flag[i] = false;
    remainder section
  } while (1);

- Satisfies mutual exclusion, but not progress requirement.
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 $P_i$

\[
\text{do } \left\{ \\
\quad \text{flag}[i] = \text{TRUE}; \\
\quad \text{turn} = j; \\
\quad \text{while ( flag}[j] && \text{turn} == j); \\
\quad \text{CRITICAL SECTION} \\
\quad \text{flag}[i] = \text{FALSE}; \\
\quad \text{REMAINDER SECTION} \\
\text{\} while (\text{TRUE});}
\]
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

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

Definition:

```c
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```
Shared boolean variable lock, initialized to false.

Solution:

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

                //  critical section

                lock = FALSE;

                //  remainder section

            } while ( TRUE);
```
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
  } while ( TRUE);
  ```
Bounded-waiting Mutual Exclusion with TestAndSet()

```c
    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;
    } while (TRUE);
```

When a process leaves CS, it scans the waiting list in cyclic order. (bounded-waiting)
Semaphore

- Synchronization tool that does not require busy waiting

- Semaphore $S$ — integer variable

- Two standard operations modify $S$: \texttt{wait()} and \texttt{signal()}
  - Originally called \texttt{P()} and \texttt{V()}

- Less complicated
Semaphore

- Can only be accessed via two indivisible (atomic) operations
  - `wait (S) {`
    - `while S <= 0`
      - `; // no-op`
      - `S--;`
  - `}
  - `signal (S) {`
    - `S++;`
  - `}`
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 S; // initialized to 1`
  - `wait (S);`
  - `Critical Section`
  - `signal (S);`
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 (S){
    value--;
    if (value < 0) {
      add this process to waiting queue
      block(); }
  }
  ```

- Implementation of signal:

  ```c
  Signal (S){
    value++;
    if (value <= 0) {
      remove a process P from the waiting queue
      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**
  
  \[
  \begin{align*}
  &P_0 \\
  &\text{wait (S)}; \\
  &\text{wait (Q)}; \\
  &\text{wait (Q)}; \\
  &\text{wait (S)}; \\
  &\vdots \\
  &\text{signal (S)}; \\
  &\text{signal (Q)};
  \end{align*}
  \]

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

- **Starvation**
  - Indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.
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 $\text{mutex}$ initialized to the value 1
- Semaphore $\text{full}$ initialized to the value 0
- Semaphore $\text{empty}$ initialized to the value $N$. 

Semaphore full initialized to the value 0.
Bounded Buffer Problem (Cont.)

- The structure of the producer process

```c
    do {
        // produce an item
        wait (empty);
        wait (mutex);
        // add the item to the buffer
        add_item(buffer);
        signal (mutex);
        signal (full);
    } while (true);
```
The structure of the consumer process

```c
do {
    wait (full);
    wait (mutex);

    // remove an item from buffer
    signal (mutex);
    signal (empty);

    // consume the removed item
}
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
{
    wait (wrt);

    // writing is performed

    signal (wrt);

} while (true)
```
The structure of a reader process

\[
\text{do } \{ \\
\text{wait (mutex) ;} \\
\text{readcount ++ ;} \\
\text{if (readercount == 1) wait (wrt) ;} \\
\text{signal (mutex)} \\
\}
\]

// reading is performed

\[
\text{wait (mutex) ;} \\
\text{readcount -- ;} \\
\text{if (readacount == 0) signal (wrt) ;} \\
\text{signal (mutex) ;} \\
\}
\] while (true)
Dining-Philosophers Problem

- **Shared data**
  - Bowl of rice (data set)
  - Semaphore **chopstick** [5] initialized to 1
Dining-Philosophers Problem (Cont.)

- The structure of Philosopher $i$:

  ```c
  Do {
    wait (chopstick[i]);
    wait (chopStick[(i + 1) % 5]);
    // eat
    signal (chopstick[i]);
    signal (chopstick[(i + 1) % 5]);
    // think
  } 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 ( ....) { ... }
    ...
}
```
Schematic View of a Monitor
Condition Variables

- condition $x, y$;

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

![Diagram of monitor with condition variables]

- Queues associated with x, y conditions
- Shared data
- Entry queue
- Operations
- Initialization code
Condition Variables with Semaphores

mutex = 1; next=0; next_count=0; x_sem = 0; x_count = 0;

Monitor Procedure F
{
mutex = 1;
wait(mutex);
Body of F
if(next_count > 0)
    signal(next);
else
    signal(mutex);
}

x.wait()
{
x_count++;
if(next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
xcount--;
}

x.signal()
{
next_count++;
signal(x_sem);
wait(next);
next_count--;
}
Condition Variables with Semaphores

Monitor Example {

Proc1() {
    P1A;
    x.wait();
    P1B;
}

Proc2() {
    P2A;
    x.signal();
    P2B;
}

Proc1 

wait(mutex)

P1A

x.wait()

signal(mutex)

wait(x_sem) {suspend}

Proc2

wait(mutex) {suspend}

P1A

wait(mutex) {suspend}

wait(x_sem) {suspend}

P2A

x.signal()

signal(x_sem)

wait(next) {suspend}

P2B

signal(next)

P1B

signal(next)

P2B

signal(mutex)
Solution to Dining Philosophers

dp.pickup (i)

.....

Eat

....

dp.putdown(i)
Solution to Dining Philosophers

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

  void pickup (int i) {
    state[i] = HUNGRY;
    test(i);
    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);
  }
}
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;
}
Conditional wait

- **Conditional-wait construct**: `x.wait(c);`
  - `c` — integer expression evaluated when the `wait` operation is executed.
  - value of `c` *(a priority number)* stored with the name of the process that is suspended.
  - when `x.signal` is executed, process with smallest associated priority number is resumed next.
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
  - Spinlock <-> block

- 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
  - A queue structure according to a priority inheritance protocol
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
  - Signaled, nonsignaled

- Dispatcher objects may also provide events
  - An event acts much like a condition variable
Linux Synchronization

- Linux:
  - disables interrupts to implement short critical sections

- 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.
- Write-ahead logging
  - Each log record describes a single operation of a transaction write.
  - (Transaction Name, Data Item Name, Old Value, New Value).
  - <Ti starts>........<Ti commits>
  - Log entry must reach stable storage before operation on data occurs
  - Penalty: two physical writes are required for every logical write request.
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

- To avoid the time-consuming searching process, checkpoints are used.
  - 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

- Serializability
  - concurrent execution of transactions = transactions executed serially
  - 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

- The execution result of a nonserial schedule is not necessarily incorrect.
Serializability (cont.)

- Oi and Oj **conflict**
  - access the same data item and at least one of these operations is a write operation.
  - If Oi and Oj are operations of different transactions and do not conflict, we can swap Oi, Oj for a new schedule S’.
  - If S can become S’ via swapping nonconflicting operations, S is **conflict serializable**.
## Serializability

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
<th>T0</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A)</td>
<td>Write(A)</td>
<td>Read(A)</td>
<td>Write(A)</td>
</tr>
<tr>
<td>Write(A)</td>
<td>Read(A)</td>
<td>Write(A)</td>
<td>Read(A)</td>
</tr>
<tr>
<td>Read(B)</td>
<td>Write(B)</td>
<td>Read(B)</td>
<td>Write(B)</td>
</tr>
<tr>
<td>Write(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Conflicting transactions: Write(B), Read(B)

**Conflict serializable**
Locking Protocol

- Ensure serializability by associating lock with each data item
  - Follow locking protocol for access control

- Locks
  - **Shared** – $T_i$ has shared-mode lock (S) on item $Q$, $T_i$ can read $Q$ but not write $Q$
  - **Exclusive** – $T_i$ has exclusive-mode lock (X) on $Q$, $T_i$ 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 $T_i$ associated with timestamp $TS(T_i)$ before $T_i$ starts
  - $TS(T_i) < TS(T_j)$ if $T_i$ entered system before $T_j$
  - $TS$ can be generated from system clock or as logical counter incremented at each entry of transaction

- Timestamps determine serializability order
  - If $TS(T_i) < TS(T_j)$, system must ensure produced schedule equivalent to serial schedule where $T_i$ appears before $T_j$
Implementation

- **Timestamp protocol**
  - A fixed timestamp $TS(Ti)$: system clock or logical counter before $Ti$ starts.
  - $W$-timestamp($Q$): the largest timestamp of write($Q$) (successfully)
  - $R$-timestamp($Q$): the largest timestamp of read($Q$) (successfully)

  - For read, if $Ts(Ti) < W$-timestamp($Q$): read is rejected, $Ti$ roll back
  - For write, if $Ts(Ti) < R$-timestamp($Q$): write is rejected, $Ti$ roll back
  - For write, if $Ts(Ti) < W$-timestamp($Q$): write is rejected, $Ti$ roll back
  - Any rolled back transaction $Ti$ is assigned new timestamp and restarted

- **Algorithm ensures conflict serializability and freedom from deadlock**
END OF CHAPTER 6