Chapter 6: 
Process Synchronization
Outline

• 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**

• Example
  – The consumer-producer problem
  – **count** is shared by producer and consumer and is updated concurrently

• **Race condition:**
  – Several processes concurrently access and manipulate the same data
  – Outcome of the execution relies on the particular order in which the access takes place

• **Sol.: Process synchronization and coordination**
  – Ensure **only one process at a time** can be manipulating the shared data
Bounded Buffer

Shared data

```c
#define BUFFER_SIZE 10
typedef struct {
    ...
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```
Bounded Buffer (Cont.)

```c
Producer process

item nextProduced;

while (1) {
    while (counter == BUFFER_SIZE) {
        /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
    }
}
```
Bounded Buffer (Cont.)

```c
Consumer process

item nextConsumed;

while (1) {
    while (counter == 0)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```
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:
  ```
  T0: producer execute register1 = count  {register1 = 5}
  T1: producer execute register1 = register1 + 1  {register1 = 6}
  T2: consumer execute register2 = count  {register2 = 5}
  T3: consumer execute register2 = register2 - 1  {register2 = 4}
  T4: producer execute count = register1  {count = 6}
  T5: consumer execute count = register2  {count = 4}
```
The Critical Section Problem

• $n$ processes all competing to use some shared data
• Each process has a code segment in which the shared data is accessed.
  – This segment is called a critical section
• Goal
  – When one process is executing in its critical section, no other process is allowed to execute in its critical section
  – No two processes are executing in their critical sections at the same time
The Critical Section Problem (Cont.)

- Each process must request the permission to enter its CS
- The section of code implementing this request is called the entry section
- The critical section (CS) might be followed by an exit section
- The remaining code is the remainder section
Solution to Critical-Section Problem

- **Mutual Exclusion**
  - At any time, at most one process can be in its critical section (CS)

- **Progress:**
  - If no process is executing in its CS and there exist some processes that wish to enter their CS...
  - The selection of the processes that will enter the critical section next cannot be postponed indefinitely.

- **Bounded Waiting**
  - After a process has made a request to enter its CS
  - A bound must exist on the number of times that the process is allowed to enter its critical sections and before that request is granted
    - Otherwise the process will suffer from *starvation*
Note: Preemptive and Nonpreemptive Kernel

- Kernel, like applications, may also face race conditions
  - Two kernel threads accesses the same file structure concurrently
- **Nonpreemptive kernel**
  - No race condition problem
  - Only one process is active in the kernel at a time
  - Windows XP and Linux prior to 2.6
- **Preemptive kernel**
  - Face race condition problem
  - Need to be designed carefully
  - Suitable for real-time programming and more responsive
  - Linux 2.6
Solutions to Race Conditions

• Software solutions
  – Peterson’s solutions

• Hardware solutions
  – Rely on some special machine instructions

• Operation System solutions
  – Provide some functions and data structures to the programmer
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;
    • Whose turn it is to enter the critical section
  – Boolean flag[2]
    • Indicate if a process is ready to enter the critical section.
    • flag[0] = 1, implies that process $P_0$ is ready to enter CS
Peterson’s Solution

- Combined shared variables of algorithms 1 and 2.
- Process $P_i$
  ```
  do {
    flag [i]:= true;
    turn = j;
    while (flag [j] and turn = j) ;
    critical section
    flag [i] = false;
    remainder section
  } while (1);
- Meets all three requirements; solves the critical-section problem for two processes.
Peterson’s Algorithm Global View

Process P0:
Do {
    flag[0]:="true;  // 0 wants in
    turn:= 1;       // 0 gives a chance to 1
    while (flag[1]&&turn=1) { CS
        flag[0]:="false; // 0 no longer wants in
        RS
        While (1);      // 0 no longer wants in
    }
}

Process P1:
Do {
    flag[1]:="true;  // 1 wants in
    turn:=0;        // 1 gives a chance to 0
    while (flag[0]&&turn=0) { CS
        flag[1]:="false; // 1 no longer wants in
        RS
        While (1);      // 1 no longer wants in
    }
}
Proof of Correctness

• Mutual exclusion is preserved since:
  – turn can only be 1 or 0 but cannot both

• Progress
  – Suppose $P_i$ wishes to enter CS
    • If flag[$j$]=false, $P_i$ can enter
    • Otherwise, turn allows either $P_i$ or $P_j$ to enter

• Bounded waiting:
  – At most one entry of $P_j$
Drawbacks of Software Solutions

• Prone to error if not careful

• Processes that are requesting to enter in their critical section are busy waiting
  – Consuming processor time needlessly

• If Critical Sections are long, it would be more efficient to block processes that are waiting...
Synchronization Hardware

• Many systems provide hardware support for critical section code

• Uniprocessors – could **disable interrupts**
  – Currently running code would execute without preemption
  – Cannot be applied on multiprocessor systems

• Modern machines provide special **atomic** hardware instructions even in MP systems
  • Atomic = non-interruptable
    – **TestAndSet**: test memory word and set value
    – **Swap**: swap contents of two memory words
Hardware Solutions: Interrupt Disabling

Process Pi:
repeat
  disable interrupts
critical section
enable interrupts
remainder section
forever
TestAndndSet Instruction

Definition: copy \textit{target} to \textit{rv} and set \textit{target} to 1

\begin{verbatim}
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
\end{verbatim}
Solution using TestAndSet

- Shared data:
  ```java
  boolean lock = false;
  ```

- Process $P_i$
  ```java
  do {
    while (TestAndSet(lock)) ;
    critical section
    lock = false;
    remainder section
  }
  ```
Hardware Solutions: Interrupt
Disabling in AL

```
enter_region:
    tsl  rv, target           ; copy target to rv and set target to 1
    cmp rv, #0               ; was target zero
    jnz  enter_region        ; if it was non zero, lock was set, so loop
    ret                       ; return to caller; enter critical region

leave_region:
    mov  target, #0          ; store a 0 in target
    ret                       ; return to caller
```
Swap Instruction

- Atomically swap two variables.

```c
void Swap(boolean &a, boolean &b) {
    boolean temp = a;
    a = b;
    b = temp;
}
```
Solution using Swap

- Shared data (initialized to false):
  ```java
  boolean lock;
  boolean waiting[n];
  ```

- Process $P_i$
  ```java
  do {
    key = true;
    while (key == true)
      Swap(lock,key);
    critical section
    lock = false;
  remainder section
  }
  ```
Bounded-Waiting Mutual Exclusion with TestAndSet

- Previous version by TestAndSet only guarantees mutual exclusion
  - But do not satisfy the bounded-waiting requirement
- Common data structures – initialized to false
  - boolean waiting[n];
  - boolean lock;
- Mutual Exclusion: \( P_i \) can enter CS only if either
  - waiting\([i]\)==false or key==false
    - key becomes false only if TestAndSet is executed
      - First process to execute TestAndSet find key==false; others wait
      - waiting\([i]\] becomes false only if other process leaves CS
        - Only one waiting\([i]\] is set to false
- Progress: similar to mutual exclusion
- Bounded waiting: waiting in the \textbf{cyclic order}
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(1);
Semaphore
Problems in Previous Approaches

• Busy waiting
  – Previous approaches, including peterson’s solutions and synchronization hardware, require busy waiting
  – Wasting CPU cycles in the loops of enter section

• Complication
  – They are complicated for application programmers to use
Semaphore

- Semaphore: synchronization tool provided by the OS that does not require busy waiting
- **Semaphore $S$ – integer variable**
- Can only be accessed via two atomic (indivisible) operations:
  ```
  wait (S): {
    while $S \leq 0$
    ; // no-op
    $S$--;
  }
  ```
  ```
  signal (S): {
    $S$++;
  }
  ```
- Thus, when one process modifies the semaphore value, no other process can simultaneously modify the same semaphore value
Semaphore as General Synchronization Tool

• **Semaphore**
  – **Counting semaphore** – integer value can range over an unrestricted domain
    • Control access to a given resource consisting of a finite number of instances
    • Semaphore value is initialized to the number of resource available
  – **Binary semaphore** – integer value can range only between 0 and 1;
    • Also known as **mutex locks**, since provide **mutual exclusion**
    • Can be used to deal with the critical-section problem
    • Binary semaphore is initialized to 1
Mutual-Exclusion Implementation with Semaphores

- For \( n \) processes, initialize \( S \) value to 1
- Then only 1 process is allowed into CS (mutual exclusion)

```c
Shared data:
semaphore mutex; //initially mutex = 1

Process \( Pi \):

do {
    wait(mutex);
    critical section
    signal(mutex);
    remainder section
} while (1);
```
Using Semaphores to Synchronize Processes

- Two processes: P1 and P2
- Statement S1 in P1 needs to be performed before statement S2 in P2
- synch is a semaphore
- Initialize synch to 0

P1:
- S1;
- signal(synch);

P2:
- wait(synch);
- S2;
Semaphore Implementation

• Disadvantage of previous semaphore solution is **busy waiting**.
• Have the same problem as previous approaches
• This type of semaphore is also call a **spinlock**
  – “spin” while waiting for the lock
  – Useful in MP if critical section is short
    • No context switch overhead and other processor can execute the CS
• To avoid busy waiting: when a process has to wait, it will be put in a blocked queue
• Redefine a semaphore as a record

```c
typedef struct {
    int value;
    struct process *L;
} semaphore;
```
Semaphore Implementation (Cont.)

wait(S):
    S.value--;
    if (S.value < 0) {
        add this process to S.L;
        block;
    }

signal(S):
    S.value++;
    if (S.value <= 0) {
        remove a process P from S.L;
        wakeup(P);
    }
Semaphore Implementation (Cont.)

- `signal(S)` and `wait(S)` should be **atomic**. This situation is a CS problem.
- **Solutions**
  - **uniprocessor** system
    - Disable interrupts during the execution of signal and wait operations
  - **multiprocessor** system, interrupt does not work here
    - Use previous software or hardware schemes.
    - The amount of busy waiting should be small
- **Note that busy waiting have not be completely eliminated.**
  - It is removed from CSs of application programs.
  - It is limited to CSs of signal and wait operations
    - Which are short. Typically 10 instructions
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.
- **Example:** Let $S$ and $Q$ be two semaphores initialized to 1

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

- **Starvation** – indefinite blocking.
  - A process may never be removed from the semaphore queue in which it is suspended.
  - For example, we add and remove processes from the semaphore queue in LIFO order.
Classical Problems of Synchronization

• Bounded-Buffer Problem

• Readers and Writers Problem

• Dining-Philosophers Problem

• The three problems are used for testing nearly every newly proposed synchronization scheme
Bounded-Buffer Problem

• \( N \) buffers, each can hold one item
• Three semaphores
  – Semaphore \textit{mutex} initialized to the value 1
    • Synchronize to access buffer
  – Semaphore \textit{full} initialized to the value 0
    • blocked if buffer is full
  – Semaphore \textit{empty} initialized to the value \( N \).
    • Blocked if buffer is empty
Bounded-Buffer Problem (Cont.) – Producer

```c

do {
    ... 
    produce an item in nextp
    ... 
    wait(empty);
    wait(mutex);
    ... 
    add nextp to buffer
    ... 
    signal(mutex);
    signal(full);
} while (1);
```
Bounded-Buffer Problem (Cont.)

do {
    wait(full)
    wait(mutex);
    ...
    remove an item from buffer to nextc
    ...
    signal(mutex);
    signal(empty);
    ...
    consume the item in nextc
    ...
} while (1);
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.

• Goal
  – Allow multiple readers to read at the same time.
  – But only one single writer can access the shared data at the same time.

• Shared Data
  – Data set
  – Semaphore `mutex` initialized to 1.
    • Synchronize access to `readcount`
  – Semaphore `wrt` initialized to 1.
    • Synchronize writer to write and data set
  – Integer `readcount` initialized to 0.
    • Keep track of how many readers are currently reading the data set
Readers and Writers Problem (Cont.)

```plaintext
reader

wait(mutex); // access readcount
readcount++;
if (readcount == 1)
    wait(wrt);
signal(mutex);

... reading is performed

wait(mutex);
readcount--;
if (readcount == 0)
    signal(wrt);
signal(mutex);
```

writer

```plaintext
wait(wrt);
...
writing is performed
...
signal(wrt);
```
Dining-Philosophers Problem

• Give philosophers, who only eat and think
  – To eat, two chopsticks are required
    • But taking one chopstick at a time

• We have only 5 forks
The Dining Philosophers Problem (Cont.)

Shared data:
semaphore chopstick[5];

Initially, all values are 1

```
Philosopher i:
  do {
    wait(chopstick[i])
    wait(chopstick[(i+1) % 5])
    ...  
    eat    
    ...  
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    ...  
    think  
    ...  
  } while (1);
```
Dining-Philosophers Problem (Cont.)

• The previous solution may cause **deadlock**
  – When all philosophers become hungry simultaneously, and each grabs her left chopstick

• Possible remedies to deadlock
  – Allow at most **four** philosophers to be sitting simultaneously
  – Allow a philosopher to pick up her chopsticks only if both chopsticks are available
    • To do this she must pick them up in a CS
  – Use an asymmetric solution
    • An **odd** philosopher picks up first **left** chopstick; then **right** chopstick,
    • An **even** philosopher picks up first **right** chopstick; then **left** chopstick

• But above remedies may also **starvation**
Problems with Semaphores

- Semaphores provide a convenient and effective mechanism for process synchronization.
- However, incorrect use may result in timing errors
  - Interchange the order of wait() and signal()
    - Violate the mutual exclusion
  - Replace signal() with wait()
    - Deadlock will occur
  - Omit wait() or signal()
    - Wither mutual exclusion is violated or a deadlock will occur
- Solution: high-level language constructs like `monitor`
Problems with Semaphores

- Incorrect order (not mutual exclusive)
- Typing error (deadlock)
- Forgotten
Monitor

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Found in many concurrent programming languages
  - Concurrent Pascal, Modula-3, uC++, Java...
- Can be implemented by semaphores...
- Only one process may be active within the monitor at a time
Monitors (Cont.)

• Encapsulate private data with public method to operate on that data

• The representation of a monitor type consists of
  – Variables whose values define the state of an instance of the type
    • These variables can be accessed by only the local procedures
  – Procedures or functions that operate on those variables
    • A procedure within a monitor can access only variables defined in the monitor and the formal parameters
Monitor Structure

```plaintext
monitor monitor-name
{
  shared variable declarations
  procedure body P1 (...) {
      ...
  }
  procedure body P2 (...) {
      ...
  }
  procedure body Pn (...) {
      ...
  }
  {
    initialization code
  }
}
```
Schematic View of a Monitor
Monitors with Condition Variables

• Previous monitor is not sufficient powerful to model some synchronization mechanisms
• Sol. Condition Variable
  – To allow a process to wait within the monitor
• A condition variable must be declared as condition x, y;
• Condition variable can only be invoked by wait() and signal() operations
  – x.wait();
    • The process invoking this operation is suspended until another process invokes
  – x.signal();
    • Resume exactly one suspended process.
    • If no process is suspended, then the signal operation has no effect.
Monitor with Condition Variables

queues associated with $x$, $y$ conditions

shared data

operations

initialization code

entry queue
Monitors with Condition Variables (Cont.)

• Suppose Q executes `x.wait()` and is blocked
• Then P executes `x.signal()` that wakes up the Q
• Now, both P and Q are active, one must wait.
• Two possibilities for the implementation:
  – **Signal and wait**: P either waits Q leaves, or waits another condition
  – **Signal and continue**: Q either waits P leaves, or waits another condition
Dining-Philosophers Solutions Using Monitors

• A **Deadlock-free Solution:**
  – A philosopher is allowed to pick up her chopsticks only if both of them are available

• **data structure**
  – **state:** array[0..4] of (thinking, hungry, eating);
    • $state[i]$ is set to eating only if both its two neighbors are not eating now
  – **self:** array[0..4] of condition;
    • Philosopher $i$ uses $self[i]$ to wait for chopsticks when she is unable to obtain the chopsticks the needs
Dining-Philosophers Solutions Using Monitors (Cont.)

• The solution
  – declare a monitor dp of type dining-philosophers
    
    \[
    \text{dp: dining-philosophers}
    \]

  – To eat, philosopher P\textsubscript{i} performs
    
    \[
    \text{dp.pickup(i);} \\
    \text{...} \\
    \text{eat} \\
    \text{...} \\
    \text{dp.putdown(i);} 
    \]

• Although deadlock free but a philosopher may be started to death

  From other authors
monitor dp
{

enum { THINKING; HUNGRY, EATING) state [5] ;
condition self [5];

void pickup (int i) {
    state[i] = HUNGRY;
    test(i);                 // try to eat
    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 (Cont)

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;
}
An illustration

P₃

P₀

P₁

P₂

P₄

(time 1)

(time 2)

(time 3)

eating;

hungry

self[2].wait;

test (0);  
test (2);  
if P₃ is not eating ⇒ self[2].signal

From other authors
Monitor Implementation Using Semaphores

- Variables

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

- Each external 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 Using Semaphores

- **Variables**
  - `semaphore mutex;`
    - For each monitor (initially = 1)
  - `semaphore next;`
    - Used by a signaling process since it may be waited (initially = 0)
  - `int next-count = 0;`
    - // number of process suspended on `next`
- Each external procedure $F$ will be replaced by

```
wait(mutex);
body of F;
if next-count > 0
    signal(next)
else
    signal(mutex);
```

- Compiler
  - Wakeup a suspended process
  - Allow others to enter the monitor
Condition Variable 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 \texttt{x.signal} can be implemented as:

\begin{verbatim}
if (x-count > 0) {
    next-count++;
    signal(x-sem);
    wait(next);
    next-count--;
}
\end{verbatim}

• A signaling process must WAIT until the resumed process either leaves or waits.
• \texttt{NEXT}: the signaling processes may suspend themselves.
Resuming Processes Within a Monitor

- If several processes are suspended on condition x
  - \texttt{x.signal()} should resume which process
- \textit{Conditional-wait} construct: \texttt{x.wait(c)};
  - \texttt{c} – integer expression evaluated when the \texttt{wait}
    operation is executed.
  - value of \texttt{c} (a \textit{priority number}) stored with the name
    of the process that is suspended.
  - when \texttt{x.signal} is executed
    - Process with smallest associated priority number is resumed next.
Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads
Synchronization in Solaris

- Adaptive mutex, condition variables, semaphores, read-writer locks, and turnstiles
- **Adaptive mutexes**
  - Protecting data from short code segments
  - In MP, if the lock is held by another thread running in another CPU
    - Use spinlock
  - Else
    - The acquiring thread is blocked and go to sleep
- **Readers-writers locks**
  - If a data is usually accessed in a read-only manner
  - More efficient than semaphore since multiple threads can read data concurrently
    - Semaphores always serialize access to the data
- **Turnstiles**
  - Order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
Windows XP Synchronization

• In XP Kernel
  – Uses *interrupt masks* to protect access to global resources on uniprocessor systems
  – Uses *spinlocks* on multiprocessor systems

• For Applications
  – 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
Synchronization in Linux

• Linux Kernel
  – Uniprocessor:
    • Disable interrupt short critical sections
    • Semaphore: otherwise

  – Multiprocessor
    • Spin lock: if a lock is held for a short duration
    • Semaphore: otherwise
Pthreads Synchronization

- Pthreads API is OS-independent
- Pthread API provides:
  - mutex locks
  - condition variables
  - read-write locks

- Non-portable extensions include:
  - semaphore
  - spin locks
Atomic Transactions

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

• **Transactions:**
  – Collection of instructions or operations that performs single logical function
  – Transaction is series of **read** and **write** operations
  – This single logical unit of work is performed either **in its entirety, or not at all**
  – i.e., preserve **atomicity**
  – Terminated by **commit** (transaction successful) or **abort** (transaction failed) operation
  – Aborted transaction must be **rolled back** to undo any changes it performed
  – Example: funds transfer

• **Challenge is assuring atomicity:**
  – Computer system crashes or failures
Types of Storage Media

• Volatile storage – information stored here does not survive system crashes
  – Example: main memory, cache

• Nonvolatile storage – Information usually survives crashes but subject to failure
  – Example: disk and tape

• Stable storage – Information never lost
  – Not actually possible, so approximated via replication or RAID to devices with independent failure modes

In the following discussion, we assume an environment where failures results in the loss of information on volatile storage.
Log-Based Recovery

• Record all modifications made by a transaction to various data on stable storage

• Sol.: Most common is write-ahead logging by writing log on stable storage,
  – Each log record describes single transaction write operation, including
    • Transaction name
    • Data item name
    • Old value
    • New value
  – Special record: \(<T_i\text{ starts}>\) written to log when transaction \(T_i\) starts
  – Special record: \(<T_i\text{ commits}>\) written to log when \(T_i\) commits

• Log entry must reach stable storage before operation on data occurs
Log-Based Recovery (Cont.)

• Performance penalty
  – Two physical writes for every write operations
  – More storage is needed

• But, using the log, system can handle any volatile memory errors
Log-Based Recovery Algorithm

• The recovery algorithm uses two procedures
  – \textit{Undo}(T_i) restores value of all data updated by T_i
  – \textit{Redo}(T_i) sets values of all data in transaction T_i to new values

• \textit{Undo}(T_i) and \textit{redo}(T_i) must be \textbf{idempotent}
  – Multiple executions must have the same result as one execution

• If system fails, restore state of all updated data via \textit{log}
  – If log contains \textit{<T_i starts>} without \textit{<T_i commits>}, \textit{undo}(T_i)
  – If log contains \textit{<T_i starts>} and \textit{<T_i commits>}, \textit{redo}(T_i)
Log-Based Recovery Algorithm (Cont.)

• When a system failure occurs
  – Consult the log to determine those transactions that need to be redone or undone

• Problems of log-based recovery
  – Log could become long, and the search process is time consuming
  – Most transactions need to be redone
    • But they have already actually updated the data
    • Although no harm due to idempotency, it cause recovery to take longer

• Sol.: Checkpoints
Checkpoints

- 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 in volatile storage to stable storage
  3. Output a log record `<checkpoint>` to the log on stable storage
- A transaction $T_i$ that committed prior to the checkpoint
  - All modifications made by $T_i$ must have been written to stable storage
  - There is no need to redo operations on $T_i$.
- Now recovery only includes
  - $T_i$, such that $T_i$ started executing before the most recent checkpoint
    - .....<checkpoint> ....<$T_i$ start> ..... 
  - And all transactions after $T_i$
- All other transactions already on stable storage
Concurrent Atomic Transactions

- Concurrent executions of transactions
  - Must be equivalent serially in some arbitrary order
  - called *serializability*
- Could perform all transactions in a critical section
  - When a transaction starts, execute `wait(mutex)`
  - When the transaction commits or aborts, execute `signal(mutex)`
  - Inefficient, too restrictive
- Sol.: *Concurrency-control algorithms* also provide serializability
Serializability

• Consider two data items $A$ and $B$
• Consider Transactions $T_0$ and $T_1$
• Execute $T_0$, $T_1$ atomically
• Any one execution sequence called a $schedule$
• **Serial schedule**: a schedule in which each transaction is executed atomically
• For $N$ transactions, there are $N!$ valid serial schedules
## Schedule 1: $T_0$ then $T_1$

<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>
Nonserial Schedule

• Previous schedules assume serial execution
• **Nonserial schedule** allows *overlapped* execution of two or more transactions
  – Resulting execution not necessarily incorrect
• Consider a schedule $S$, operations $O_i$, $O_j$
  – **Conflict** if access same data item, with at least one write
• If $O_i$, $O_j$ are consecutive and are operations of different transactions and $O_i$ and $O_j$ don’t conflict
  – Then $S$’ with swapped order $O_j O_i$ equivalent to $S$
Schedule 2: Concurrent Serializable Schedule

<table>
<thead>
<tr>
<th></th>
<th>$T_0$</th>
<th>$T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>$read(A)$</td>
<td>$read(A)$</td>
</tr>
<tr>
<td>write</td>
<td>$write(A)$</td>
<td>$write(A)$</td>
</tr>
<tr>
<td>read</td>
<td>$read(B)$</td>
<td>$read(B)$</td>
</tr>
<tr>
<td>write</td>
<td>$write(B)$</td>
<td>$write(B)$</td>
</tr>
</tbody>
</table>
Conflict Serializable

• Example
  – `write(A)` in $T_1$ can be swapped with `read(B)` in $T_0$
  – The new schedule is equivalent to original schedule
    • Produce the same final system state
  – `read(B)` in $T_0$ can be swapped with `read(A)` in $T_1$
  – `write(B)` in $T_0$ can be swapped with `write(A)` in $T_1$
  – `write(B)` in $T_0$ can be swapped with `read(A)` in $T_1$
  – Finally, scheduler 2 is becomes schedule 1

• If $S$ can be transferred to a serial schedule $S'$ via swapping a series of nonconflicting operations
  – $S$ is conflict serializable
Locking Protocol

- One way to ensure serializability is to associate with each data item a lock
  - Each transaction follow a **locking protocol** for access control
- Two lock modes
  - **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
- To ensure serializability: **two-phase locking protocol**
Two-phase Locking Protocol

- Each transaction issues lock and unlock requests in two phases
  - **Growing phase** – obtaining locks but cannot release any lock
  - **Shrinking phase** – releasing locks but cannot obtain any lock
- Initially, a transaction is in the growing phase.
  - The transaction acquires locks as needed.
- Once the transaction releases a lock, it enters the shrinking phase
  - **No more lock requests can be issued**
- Ensures **conflict serializability**
- But does not prevent deadlock
  - If two processes acquire the same pair of locks in the opposite order
Two-Phase Locking
Timestamp-based Protocols

• Solutions to serializability
  – Locking: i.e., mutual exclusion on shared data
    • The execution order is determined at execution time
  – Timestamp: explicitly ordering operations
    • Select order among transactions in advance – timestamp-ordering
Timestamp-based Protocols

• Each Transaction $T_i$ is associated with timestamp $TS(T_i)$
  – $TS(T_i) < TS(T_j)$ if $T_i$ entered system before $T_j$
  – TS generation
    • From system clock or
    • Logical counter that is 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 $T_j$
Timestamp-based Protocol Implementation

• Each data item Q has two timestamps
  – **W-timestamp(Q)** – largest timestamp of any transaction that executed write(Q) successfully
  – **R-timestamp(Q)** – largest timestamp of any transaction that executed read(Q) successfully
  – Updated whenever read(Q) or write(Q) executed

• Timestamp-ordering protocol assures any conflicting read and write executed in timestamp order
Timestamp-ordering Protocol

- Suppose $T_i$ executes **read(Q)**
  - If $TS(T_i) < W$-timestamp(Q), $T_i$ needs to read value of Q that was already overwritten
    - **read** operation rejected and $T_i$ rolled back
  - If $TS(T_i) \geq W$-timestamp(Q)
    - **read** executed, R-timestamp(Q) set to $\max(R$-timestamp(Q), TS($T_i$))

- Suppose $T_i$ executes **write(Q)**
  - If $TS(T_i) < R$-timestamp(Q), value Q produced by $T_i$ was needed previously and $T_i$ assumed it would never be produced
    - **Write** operation rejected, $T_i$ rolled back
  - If $TS(T_i) < W$-timestamp(Q), $T_i$ attempting to write obsolete value of Q
    - **Write** operation rejected and $T_i$ rolled back
  - Otherwise, **write** executed
Schedule 3: A Schedule Possible Under Timestamp Protocol

<table>
<thead>
<tr>
<th>(T_2)</th>
<th>(T_3)</th>
<th>R_timestamp(B)</th>
<th>W_timestamp(B)</th>
<th>R_timestamp(A)</th>
<th>W_timestamp(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(B)</td>
<td>read(B)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>read(A)</td>
<td>write(A)</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Assume \(TS(T2) = 1\) and \(TS(T3) = 2\), thus, \(TS(T2) < TS(T3)\)

Schedule 3 is possible under the timestamp protocol
Timestamp-ordering Protocol

- Any rolled back transaction $T_i$ is assigned new timestamp and **restarted**
- Algorithm ensures conflict serializability
  - Conflicting operations are processed in timestamp order
- Algorithm freedom from deadlock
  - No transaction ever waits