Chapter 6: Process Synchronization
Module 6: Process 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.
while (true) {
  /* produce an item and put in nextProduced */
  while (counter == BUFFER_SIZE)
    ; // do nothing
  buffer [in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
  counter++;
}
while (true) {
  while (counter == 0)
    ; // do nothing
  nextConsumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  counter--;

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

- `counter++` could be implemented as

  ```
  register1 = counter
  register1 = register1 + 1
  counter = register1
  ```

- `counter--` could be implemented as

  ```
  register2 = counter
  register2 = register2 - 1
  count = 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`  \{count = 6 \}
  S5: consumer execute `counter = register2`  \{count = 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**
- Especially challenging with preemptive kernels
Critical Section

- General structure of process $p_i$ is

```c
do {
    entry section
    critical section
    exit section
    remainder section
} while (TRUE);
```

Figure 6.1 General structure of a typical process $p_i$. 
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 $P_i$

\[\begin{array}{l}
do \\
\begin{array}{l}
\text{flag[i] = TRUE;}
\text{turn = j;}
\text{while (flag[j] && turn == j);}
\end{array} \\
\text{critical section}
\begin{array}{l}
\text{flag[i] = FALSE;}
\end{array} \\
\text{remainder section}
\end{array} \\
\text{while (TRUE);}\]

- Provable that
  1. Mutual exclusion is preserved
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met
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;
}
```
Solution using TestAndSet

- 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()

\[
\text{do } \{
\begin{align*}
\text{waiting}[i] &= \text{TRUE}; \\
\text{key} &= \text{TRUE}; \\
\text{while } (\text{waiting}[i] \&\& \text{key}) & \text{ key = TestAndSet(&lock);} \\
\text{waiting}[i] &= \text{FALSE}; \\
\ & \text{// critical section} \\
\ & j = (i + 1) \% n; \\
\ & \text{while } ((j \neq i) \&\& \!\! \text{waiting}[j]) \\
\ & \ & j = (j + 1) \% n; \\
\ & \text{if } (j == i) \\
\ & \ & \text{lock} = \text{FALSE}; \\
\ & \text{else} \\
\ & \ & \text{waiting}[j] = \text{FALSE}; \\
\ & \ & \text{// remainder section} \\
\} \text{ while (TRUE);} \\
\]
Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore S – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations
  - wait (S) {
    while $S \leq 0$
    ; // 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

  ```c
  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

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

- **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
  - 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, each can hold one item
- Semaphore `mutex` initialized to the value 1
- Semaphore `full` initialized to the value 0
- Semaphore `empty` initialized to the value $N$
The structure of the producer process

\[
do \{ \\
  // produce an item in nextp \\
  wait (empty); \\
  wait (mutex); \\
  // add the item to the buffer \\
  signal (mutex); \\
  signal (full); \\
} while (TRUE);
\]
The structure of the consumer process

```c
    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

- Several variations of how readers and writers are treated – all involve priorities

- Shared Data
  - Data set
  - Semaphore \texttt{mutex} initialized to 1
  - Semaphore \texttt{wrt} initialized to 1
  - Integer \texttt{readcount} initialized to 0
The structure of a writer process

```c

do {
    wait (wrt) ;

    // writing is performed

    signal (wrt) ;
} while (TRUE);

```
The structure of a reader process

```
do {
    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);
```
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 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 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?
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
Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

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

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

    Initialization code (...) { ... }
}
```
Schematic view of a Monitor

- Shared data
- Operations
- Initialization code
- Entry queue
Condition Variables

- condition x, y;

- Two operations on a condition variable:
  - x.wait() – a process that invokes the operation is suspended until x.signal()
  - x.signal() – resumes one of processes (if any) that invoked x.wait()
    - If no x.wait() on the variable, then it has no effect on the variable
Monitor with Condition Variables

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

- If process P invokes `x.signal()`, with Q in `x.wait()` state, what should happen next?
  - If Q is resumed, then P must wait

- Options include
  - **Signal and wait** – P waits until Q leaves monitor or waits for another condition
  - **Signal and continue** – Q waits until P leaves the monitor or waits for another condition

- Both have pros and cons – language implementer can decide
- Monitors implemented in Concurrent Pascal compromise
  - P executing signal immediately leaves the monitor, Q is resumed
- Implemented in other languages including Mesa, C#, Java
Solution to Dining Philosophers

monitor DiningPhilosophers
{
    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;
}
Each philosopher $i$ invokes the operations `pickup()` and `putdown()` in the following sequence:

```
DiningPhilosophers.pickup (i);
EAT
DiningPhilosophers.putdown (i);
```

- No deadlock, but starvation is possible
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 – Condition Variables

For each condition variable $x$, we have:

```c
semaphore x_sem; // (initially = 0)
int x-count = 0;
```

The operation $x.wait$ can be implemented as:

```c
x-count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x-count--;
```
Monitor Implementation (Cont.)

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

```c
if (x-count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```
Resuming Processes within a Monitor

- If several processes queued on condition x, and x.signal() executed, which should be resumed?

- FCFS frequently not adequate

- **conditional-wait** construct of the form x.wait(c)
  - Where c is **priority number**
  - Process with lowest number (highest priority) is scheduled next
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
  - Starts as a standard semaphore spin-lock
  - If lock held, and by a thread running on another CPU, spins
  - If lock held by non-run-state thread, block and sleep waiting for signal of lock being released

- Uses **condition variables**

- Uses **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
  - Turnstiles are per-lock-holding-thread, not per-object

- Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile
Windows XP 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
  - spinlocks
  - reader-writer versions of both

- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption
Pthreads Synchronization

- Pthreads API is OS-independent

- It provides:
  - mutex locks
  - condition variables

- Non-portable extensions include:
  - read-write locks
  - spinlocks
The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set

- Example
  - System has 2 disk drives
  - \( P_1 \) and \( P_2 \) each hold one disk drive and each needs another one

- Example
  - semaphores \( A \) and \( B \), initialized to 1
  - \( P_0 \) \( P_1 \)
  - \( \text{wait}(A) \); \( \text{wait}(B) \) \( \text{wait}(A) \)
Bridge Crossing Example

- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible
- Note – Most OSes do not prevent or deal with deadlocks
System Model

- Resource types $R_1, R_2, \ldots, R_m$
  
  $CPU$ cycles, $memory$ space, $I/O$ devices

- Each resource type $R_i$ has $W_i$ instances.

- Each process utilizes a resource as follows:
  - request
  - use
  - release
Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously

- **Mutual exclusion**: only one process at a time can use a resource

- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes

- **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task

- **Circular wait**: there exists a set \( \{P_0, P_1, \ldots, P_n\} \) of waiting processes such that \( P_0 \) is waiting for a resource that is held by \( P_1 \), \( P_1 \) is waiting for a resource that is held by \( P_2 \), \ldots, \( P_{n-1} \) is waiting for a resource that is held by \( P_n \), and \( P_n \) is waiting for a resource that is held by \( P_0 \).
Resource-Allocation Graph

A set of vertices $V$ and a set of edges $E$

- $V$ is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system

- request edge – directed edge $P_i \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$
Resource-Allocation Graph (Cont.)

- Process

- Resource Type with 4 instances

- \( P_i \) requests instance of \( R_j \)

- \( P_i \) is holding an instance of \( R_j \)
Example of a Resource Allocation Graph
Resource Allocation Graph With A Deadlock
Graph With A Cycle But No Deadlock
Basic Facts

- If graph contains no cycles ⇒ no deadlock
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock
Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX
Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources

- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none
  - Low resource utilization; starvation possible
Deadlock Prevention (Cont.)

- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration
Deadlock Avoidance

Requires that the system has some additional a priori information available

- Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes
End of Chapter 6