Chapter 6: Process Synchronization
Module 6: Process Synchronization

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

- Concurrent access to shared data may result in data inconsistency → Multiple threads in a single process

- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
Background -2-

- **Disjoint** threads use disjoint sets of variables, and do not use shared resources.
  - The progress of one thread is independent of all other threads, to which it is disjoint.

- **Non-disjoint** threads influence each other by using shared data and/or resources.

- Two possibilities:
  - **Competing** threads: compete for access to the data resources
  - **Cooperating** threads: e.g. producer / consumer model

- Without synchronization, the effect of non-disjoint parallel threads influencing each other is not predictable and not reproducible.
Example -1-

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.
The Producer Consumer Problem:

A producer process "produces" information "consumed" by a consumer process.

Here are the variables needed to define the problem:

```
#define BUFFER_SIZE 10
typedef struct {
    DATA data;
} item;
item buffer[BUFFER_SIZE];
int in = 0; // Location of next input to buffer
int out = 0; // Location of next removal from buffer
int count = 0; // Number of buffers currently full
```
Producer / Consumer

Producer:
item nextProduced;
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:
item nextConsumed;
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: producer execute `count = register1` {count = 6}
  S5: consumer execute `count = register2` {count = 4}
  ```
  - count = 4 after count++ and count--, even though we started with count = 5
  - Easy question: what other values can count be from doing this incorrectly?

- Obviously, we would like to have count++ execute, followed by count-- (or vice versa)
Example: A database with personal data is accessed by several applications, e.g., an address management and a salary management program.

<table>
<thead>
<tr>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miller</td>
<td>4000.--</td>
</tr>
</tbody>
</table>

- **address management**
  - read record
    - Miller 4000.--
  - change name
    - Smith 4000.--
  - write record

- **salary management**
  - read record
    - Miller 4000.--
  - change salary
    - Miller 4180.--
  - write record

- Possible final values:
  - Miller 4180.-- or Smith 4000.--
- Correct would be:
  - Smith 4180.--
Race Conditions

- A race condition is where multiple processes/threads concurrently read and write to a shared memory location and the result depends on the order of the execution.

- The part of the program, in which race conditions can occur, is called critical section.
Critical Sections

- A **critical section** is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.

- The goal is to provide a mechanism by which only one instance of a critical section is executing for a particular shared resource.

- Unfortunately, it is often very difficult to detect critical section bugs.
A Critical Section Environment contains:

- **Entry Section** Code requesting entry into the critical section.
- **Critical Section** Code in which only one process can execute at any one time.
- **Exit Section** The end of the critical section, releasing or allowing others in.
- **Remainder Section** Rest of the code AFTER the critical section.

```c
do {
    Entry Section;
    critical section;
    Exit Section;
    remainder sections;
} while (true);
```
A solution to the critical-section problem must satisfy the following requirements:

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, or limit must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
   - Assume that each process executes at a nonzero speed.
   - No assumption concerning relative speed of the $N$ processes.
Critical section problem

- an example, a kernel data structure that maintains a List of all open files in the system. If it will be accessed simultaneously by two processes this will result in a race condition.

- Two general approaches to handle CS in OS:
  - Preemptive kernels e.g. Linux (carefully designed).
  - Nonpreemptive kernels e.g. WinXP (free from race conditions?)
Peterson’s Solution

- Two processes solution
- It provides a good algorithmic description of solving the critical-section problem
- Algorithm is only for 2 processes at a time
- Processes are
  - $P_0$
  - $P_1$
- Or can also be represented as $P_i$ and $P_j$
- i.e. $j=1-i$
Let the processes share common integer variable $turn$

Let $int\ turn = 0$ (or 1)

If process $turn == i$, $P_i$ is allowed to execute in critical section

But,

- Guarantees mutual exclusion.
- Does not guarantee progress --- enforces strict alternation of processes entering CS's
- (if $P_j$ decides not to re-enter or crashes outside CS, then $P_i$ cannot ever get in).

Sonali C.
2 process - Algorithm 2

- Shared variables
  - boolean flag[2]; // “interest” bits
    initially flag [0] = flag [1] = false.
  - flag [i] = true ⇒ \( P_i \) declares interest in entering its critical section

- Process \( P_i \) // where the “other” process is \( P_j \)

```plaintext
do {
  flag[i] = true; // declare your own interest
  while (flag[j]); // wait if the other guy is interested
  critical section
  flag [i] = false; // declare that you lost interest
  remainder section // allows other guy to enter
} while (1);
```
2-PROCESS - Algorithm 2

- Satisfies mutual exclusion, but not progress requirement.
  - If flag[i] == flag[j] == true, then deadlock - no progress
  - but barring this event, a non-CS guy cannot block you from entering
  - Can make consecutive re-entries to CS if other not interested
TWO-PROCESS -Algorithm 3

- Combined shared variables & approaches of algorithms 1 and 2.

```
For Process P_i

do {
    flag [i] = true;       // declare your interest to enter
    turn = j;    // assume it is the other’s turn-give PJ a chance
    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.

- Turn variable breaks any deadlock possibility of previous example, AND prevents “hogging” – $P_i$ setting turn to $j$ gives $P_j$ a chance after each pass of $P_i$’s CS
- Flag[ ] variable prevents getting locked out if other guy never re-enters or crashes outside and allows CS consecutive access other not interested in entering.
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 (TestAndSet())
    - Or swap contents of two memory words (Swap())
- TestAndSet is hard to program for end users
Semaphore

- **Semaphore: a synchronization tool**, A flag used to indicate that a routine cannot proceed if a shared resource is already in use by another routine.
- 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 <= 0`
      - `; // no-op`
    - `S--;`
  `}
  - `signal (S) {`
    - `S++;`
  `}`
Semaphore as General Synchronization Tool

- Two types:
  - **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 (they are locks that provide mutual exclusion)

- There are 3 uses for Semaphores:
Semaphores Usage

Usage1:

- Binary semaphores can solve the critical-section problem for \( n \) processes. The \( n \) processes share a semaphore, mutex, initialized to 1. Each process \( P_i \) is organized as:

  Semaphore mutex;    // initialized to 1
  do {
    wait (mutex);
    // Critical Section
    signal (mutex);
    // remainder section
  } while (TRUE);

- Only one process is allowed into Critical Section (mutual exclusion).
Semaphores Usage

Usage2:

- Counting semaphores control access to a given resource of many instances.
- Simply initialize the semaphore to $K$ (# of available resources).
- This allows $K$ processes to enter their critical-sections and use that resource at a time.
  - Each process wants to use a resource performs `wait()` operation → decrementing the count of the semaphore.
  - Each process releases a resource, it performs `signal()` operation → incrementing the count of the semaphore.
  - When the `count = 0`, all resources are being used. Any process wants to use a resource will block until `count > 0`. 
Semaphores Usage

Usage3:
- To solve synchronization problems.
- Example:
  - Two concurrent processes: P₁ and P₂
  - Statement S₁ in P₁ needs to be performed before statement S₃ in P₂
  - Need to make P₂ wait until P₁ tells “it is OK to proceed”
    - Define a semaphore “synch” → Initialize synch to 0.
  - In P₂:
    - wait(synch);
    - S₃;
  - And in P₁:
    - S₁;
    - signal(synch);
Semaphore Implementation

- The main disadvantage of the semaphore is that it requires **busy waiting**, which wastes CPU cycle that some other process might be able to use productively.

- This type of semaphore is also called a **spinlock** because the process “spins” while waiting for the lock.
Semaphore Implementation with no Busy waiting

- Each semaphore has an integer value and an associated waiting queue.

    typedef struct {
        int value;
        struct process *list;
    } semaphore;

- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue, and the state of the process is switched to the waiting state.
  - Then control is transferred to the CPU scheduler, which selects another process to execute.
  - **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--;  // Decrement the semaphore value
    if (S->value < 0) {
      add this process to S->list;  // Add the process to the list
      block();  // Block the process
    }
  }
  ```

- Implementation of signal:
  ```c
  signal(semaphore *S) {
    S->value++;  // Increment the semaphore value
    if (S->value <= 0) {
      remove a process P from S->list;  // Remove a process from the list
      wakeup(P);  // Wake up the process
    }
  }
  ```
A queue is used to hold processes waiting on a semaphore.
Semaphore

- It's critical that these be atomic. We must guarantee that no two processes can execute `wait()` and `signal()` operations on the same semaphore at the same time.
- In uniprocessors we can disable interrupts,
- But in multiprocessors other mechanisms for atomicity are needed.
Deadlock

- **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

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

- Suppose that \( P_0 \) executes \textit{wait(S)} and \( P_1 \) then executes \textit{wait(Q)}.

- When \( P_0 \) executes \textit{wait(Q)}, it must wait until \( P_1 \) executes \textit{signal(Q)}.

- Similarly, when \( P_1 \) executes \textit{wait(S)}, it must wait until \( P_0 \) executes \textit{signal(S)}.

- Since these \textit{signal()} operations cannot be executed, \( P_0 \) and \( P_1 \) are deadlocked.
Starvation

- 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
  - Bounded buffers P/C can be seen in e.g. streaming filters or packet switching in networks.

- Readers and Writers Problem
  - Database readers and writers: online reservation systems; file systems.

- Dining-Philosophers Problem
  - Dining philosophers could be a sequence of active database transactions that have a circular wait-for-lock dependence.

- The Sleeping Barber problem
  - Sleeping Barber is often generally thought of as a client-server relationship.
Bounded-Buffer Problem

- $N$ buffers, each can hold one item
- Semaphore `mutex` controls access to region and initialized to the value 1.
- Semaphore `full` counts full buffer slots and initialized to the value 0.
- Semaphore `empty` counts empty buffer slots and initialized to the value $N$. 
Bounded Buffer Problem (Cont.)

The structure of the producer process

```c
    do {
        // produce an item in nextp
        wait (empty); /* decrement empty count */
        wait (mutex); /* enter critical region */

        // add the item to the buffer
        signal (mutex); /* leave critical region */
        signal (full); /* increment count of full slots */
    } while (TRUE);
```
The structure of the consumer process

```c
do {
    wait (full); /* decrement full count */
    wait (mutex); /* enter critical region */

    // remove an item from buffer to nextc

    signal (mutex); /* leave critical region */
    signal (empty); /* increment count of empty slots */

    // consume the item in nextc

} while (TRUE);
```
Classical Problem 2:
The Readers-Writers Problem

Multiple readers or a single writer can use DB.
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.)

```c
BINARY_SEMAPHORE wrt = 1;
BINARY_SEMAPHORE mutex = 1;
int readcount = 0;

The structure of a writer process

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

The structure of a reader process

do {
    wait (mutex) ; /* Allow 1 reader in entry*/
    readcount ++ ;
    if (readcount == 1)
        wait (wrt) ; /* 1st reader locks writer */
    signal (mutex)
    // reading is performed
    wait (mutex) ;
    readcount  - - ;
    if (readcount  == 0)  signal (wrt) ;
    /*last reader frees writer */
    signal (mutex) ;
} while (TRUE);
```
Dining-Philosophers Problem

Shared data
- Bowl of rice (data set)
- Semaphore `chopstick [5]` initialized to 1
The Structure of Philosopher $i$

- Philosopher $i$
  while ( true ) {
    // get left chopstick
    wait(chopStick[i]);
    // get right chopstick
    wait(chopStick[(i + 1) % 5]);

    // eat for a while
    //return left chopstick
    signal(chopStick[i]);
    // return right chopstick
    signal(chopStick[(i + 1) % 5]);

    // think for awhile
  }

A deadlock occurs!
Dining-Philosophers Problem

- 5 philosophers with 5 chopsticks sit around a circular table. They each want to eat at random times and must pick up the chopsticks on their right and on their left.
- Clearly deadlock is rampant (and starvation possible.)
- Several solutions are possible:
  - Allow only 4 philosophers to be hungry at a time.
  - Allow pickup only if both chopsticks are available. (Done in critical section)
  - Odd # philosopher always picks up left chopstick 1st, even # philosopher always picks up right chopstick 1st.
The Sleeping Barber Problem

A barbershop consists of a waiting room with \( N \) chairs, and the barber room containing the barber chair. If there are no customers to be served the barber goes to sleep. If a customer enters the barbershop and all chairs are busy, then the customer leaves the shop. If the barber is busy, then the customer sits in one of the available free chairs. If the barber is asleep, the customer wakes the barber up.
The Sleeping Barber Problem

The following pseudo-code guarantees synchronization between barber and customer and is deadlock free, but may lead to starvation of a customer.

Semaphore Customers = 0
Semaphore Barber = 0
Semaphore accessSeats (mutex) = 1
int NumberofFreeSeats = N  //total number of seats

The Barber (Thread/Process):
while (true) { //runs in an infinite loop
    wait(Customers) //tries to acquire a customer - if none is available he goes to sleep
    wait(accessSeats) //at this time he has been awakened - want to modify the number of available seats
    NumberofFreeSeats++ //one chair gets free
    signal(Barber) //the barber is ready to cut
    signal(accessSeats) //we don't need the lock on the chairs anymore
    //here the barber is cutting hair }

The Customer (Thread/Process):
while (true) { //runs in an infinite loop
    wait(accessSeats) //tries to get access to the chairs
    if ( NumberofFreeSeats > 0 ) { //if there are any free seats
        NumberofFreeSeats-- //sitting down on a chair
        signal(Customers) //notify the barber, who's waiting that there is a customer
        signal(accessSeats) //don't need to lock the chairs anymore
        wait(Barber) //now it's this customer's turn, but wait if the barber is busy
        //here the customer is having his hair cut }
    else { //there are no free seats //tough luck
        signal(accessSeats) //but don't forget to release the lock on the seats
        //customer leaves without a haircut } }
Synchronization Examples

- Windows XP
- Linux
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 (they may notify a waiting thread when a desired condition occurs).
Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, a nonpreemptive kernel.
  - Version 2.6 and later, fully preemptive

- Linux provides:
  - semaphores
  - spin locks

<table>
<thead>
<tr>
<th></th>
<th>Single processor</th>
<th>Multiple processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable kernel preemption</td>
<td>Acquire spin lock</td>
<td></td>
</tr>
<tr>
<td>enable kernel preemption</td>
<td>Release spin lock</td>
<td></td>
</tr>
</tbody>
</table>