Chapter 6: Process Synchronization
Objectives

- Introduce Concept of Critical-Section Problem
- Hardware and Software Solutions of Critical-Section Problem
- Concept of Atomic Transaction
Background
Producer-Consumer Problem Revisited

- Model illustrates:
  - Asynchronous Processes and/or Threads (lightweight processes) share data memory

- Producer code:

  ```c
  while (true)
  {
      /* Produce an item in nextProduced */
      while (counter == BUFFER_SIZE) ; /* do nothing */
      buffer[in] = nextProduced;
      in = (in + 1) % BUFFER_SIZE;
      counter++;
  }
  
  Shared memory: bounded circular buffer
  ```

- Consumer code:

  ```c
  while (true)
  {
      while (counter == 0) ; /* do nothing */
      nextConsumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      counter--;
      /* Consume item in nextConsumed */
  }
  ```
Background

Producer-Consumer Problem Revisited

- Concurrent executions of Producer and Consumer Routines yields
  - Possible inconsistency in value of counter
    - Let’s examine scenarios when current value is 4
      - counter = 3-4-5 ?
    - Root cause:
      - Both processes allowed to manipulate counter variable concurrently and outcome depended on the order of executions (Race Condition)
  - Solution Approach
    - Ensure only one process at time can manipulate the counter variable
      - We will talk about this shortly
Critical Section Problem

General Structure of Typical Process

Process:
do
{
  entry selection
  <CRITICAL SECTION>
  exit section
  <Remainder section>
} while (TRUE);
Mutual Exclusion Concepts

- **Race Condition**
  - Several processes access and manipulate the same data concurrently, and the outcome of execution depends on the order in which access takes place.

- **Critical section**
  - A segment of code in which a process has exclusive access to manipulate one or more shared resources.
    - Write to a file, update a table, change a common variables
    - The general structure of a typical process:
      - `<entry section> CRITICAL SECTION <exit section> Remainder section`

- **Mutual Exclusion**
  - When several processes try to use the same set of resources, how do we ensure that they gain access to the resources only one at a time.
Critical Section Problem
Solution Approach

Requirements

■ Mutual Exclusion
  - Allow one and only one process at a time to execute in the CRITICAL SECTION of the code

■ Progress
  - A process that is not requesting entry to CRITICAL SECTION of its code should not block the process (or processes) that are requesting entry into theirs

■ Bounded waiting
  - A process should not wait indefinitely before it can enter the CRITICAL SECTION
Critical Section Problem

Peterson’s Solution

- Restricted to two Processes \((P_0, P_1)\)
  - Processes alternate executions between Critical sections and Remainder section
  - Shared data items:
    - `int turn; /* turn = 1 \implies P_1 is allowed to execute critical section */`
    - `boolean flag[2]; /* flag[0] = true \implies P_0 is ready to enter critical section */`

Process 0:
```c
do
{
    flag[0] = TRUE;
    turn = 1
    while (flag[1] && turn == 1);
    <CRITICAL SECTION>
    flag[0] = FALSE;
    <Remainder section>
} while (TRUE);
```

Process 1:
```c
do
{
    flag[1] = TRUE;
    turn = 0
    while (flag[0] && turn == 0);
    <CRITICAL SECTION>
    flag[1] = FALSE;
    <Remainder section>
} while (TRUE);
```
Critical Section Problem

Solution

- To Avoid race conditions:
  - Process must acquire a lock before entering the critical section

\begin{verbatim}
Process:
do
{
     Acquire lock
     <CRITICAL SECTION>
     Release lock
     <Remainder section>
} while (TRUE);
\end{verbatim}
Semaphores

- A special Synchronization variable S
  - Defines an event, which is announced to the OS
  - Process wishes to wait for the event
    - Calls \texttt{wait(s)}
  - Process executing the event, after event completes
    - Calls \texttt{signal(s)}

- OS ensures that only one process can modify S at a time
Semaphore Implementation

- Semaphore “s” is initially set to 1
  - Before entering the critical section, process calls `wait (s)`
    - `s = s -1`
    - if (s < 0)
      - Block the process that called `wait (s)` on a queue associated with a semaphore s
      - Else
        - Allow the process that called `wait(s)` continue into critical section
  - After exiting critical section, process calls `signal (s)`
    - `s = s + 1`
    - if (s <= 0) then
      - Wake up one of the threads that called `wait (s)` and run it so it continues into critical section
Semaphore Implementation

Sample code

Process $P_0$:

```c
{ 
    t() { 
        while (true) { 
            wait(s); 
            <CRITICAL SECTION> 
            signal(s); 
            <Remainder section> 
        } 
    } 
}
```

Process $P_1$:

```c
{ 
    p() { 
        while (true) { 
            wait(s); 
            <CRITICAL SECTION> 
            signal(s); 
            <Remainder section> 
        } 
    } 
}
```

**OS** ensures that no two processes can execute `wait()` and `signal()` on the same semaphore at the same time.
Semaphore Implementation So far…

Busy Waiting

Recall:

- Process $P_0$:

  ```
  t()
  while (true){
    wait (s);
    <CRITICAL SECTION>
    signal (s);
    <Remainder section>
  }
  }
  ```

Process $P_1$ spins:

While Process 0 is in CRITICAL SECTION:
- Process 1 loops continuously in ENTRY SECTION of its code ($s <= 0$)
  - Wastes CPU cycles
  - Semaphore Type: SPINLOCK

Counting Semaphore and Binary Semaphore may cause Spinlock

How do we avoid Spinlock?
1. Define semaphore type:

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

2. Modify wait ( ) semaphore operation:
   - If Process P₁’s semaphore value in entry section is not positive
   - Process P₁ puts itself in a waiting queue by invoking system call block( )

3. Modify signal ( ) semaphore operation:
   - When P₀ executes signal ( )
     - Restart a process in waiting queue by invoking system call wakeup( )
Semaphore Implementation
Avoiding Spinlock

- `wait()` semaphore operation

```c
wait (semaphore *S {  
    S->value --;  
    if S->value < 0 {  
        add process that called wait () to S->list;  
        block ();  
    }  
}  

Suspend the process that called wait
```

Operating Systems CS 33211
Semaphore Implementation
Avoiding Spinlock

signal ( ) semaphore operation

```c
signal (semaphore *S {
    S->value ++;
    if S->value <= 0 {
        remove a process P from S->list;
        wakeup (P);
    }
}
```

Which process?
P resumes execution of its critical section
Semaphore Implementation

- Process $P_0$:
  
  ```
  t()
  while (true) {
    wait (*s);
    <CRITICAL SECTION>
    signal (*s);
    <Remainder section>
  }
  }
  ```

- Process $P_1$:
  
  ```
  p()
  while (true) {
    wait (*s);
    <CRITICAL SECTION>
    signal (*s);
    <Remainder section>
  }
  ```
Deadlocks and Starvation

Waiting Queue

- **Deadlock**
  - Two or more processes wait for an event, `signal op`, that can be triggered by only one of the waiting processes

```
P_0
wait(S);
wait(Q);

\vdots

signal(S);
signal(Q);

P_1
wait(Q);
wait(S);

\vdots

signal(Q);
signal(S);
```

- **Starvation:**
  - When one processes wait indefinitely within the same semaphore
Classic Problems of Synchronization
Case #1: The Bounded-Buffer Problem

Why examine Classic Synch Problems?

The Bounded-Buffer Problem: (Producer/Consumer Processes)

- Assumptions
  - n-buffers, each holds one item
  - Semaphore (mutex: 0/1) provides mutual exclusion access to n-buffers
  - Semaphore (empty) ~ count of empty buffers: [empty \(\leftarrow n\)]
  - Semaphore (full) ~ count of full buffers: [full \(\leftarrow 0\)]

```plaintext
Producer Process:
do {
    /* Produce an item in next nexp */
    wait (empty)
    wait (mutex)
    ........
    /* add nexp to buffer */
    ........
    signal (mutex);
    signal (full);
} while (TRUE);
```

```plaintext
Consumer Process:
do {
    /* Produce an item in next nexp */
    wait (full)
    wait (mutex)
    ........
    /* remove item from buffer to nexp */
    ........
    signal (mutex);
    signal (empty);
} while (TRUE);
```
Classic Problems of Synchronization
Case #2: Readers-Writers Problem (dB)

Assumptions
- Assume dB is shared by n Concurrent Processes $R_i$, $i = 1, j; W_i$, $i = 1, k$; where $n = j + n$
- Processes R perform a read operation on dB
- Processes W perform dB update (read followed by write)

What happens to data item if Processes R access the shared data simultaneously?

What could happen to a shared db object if a W Process and another Processes R or W access the shared data simultaneously?
- Assume interrupts are not inhibited

Basic Solution Requirements
1. Writers (W) will have exclusive access to db
2. A Reader cannot be blocked unless Writer has permission to shared object
Classic Problems of Synchronization
Case #2: Readers-Writers Problem (dB)

The Basic Solution:

- Assumptions
  - Semaphore (wrt: 0/1) provides mutual exclusion for writers (used by readers) wrt ← 1
  - Semaphore (readcount) ~ count of process reading shared object: [readcount ← 0]
  - Semaphore (mutex) ~ provides mutual exclusion after updating readcount: [mutex ← 1]

Writer Processes:
- semaphore wrt
- do {
  - wait (wrt)
  - /* write to db */
  - ......
  - signal (wrt);
- } while (TRUE);

Reader Processes:
- semaphore mutex, wrt
- int readcount;
- do {
  - wait (mutex)
  - readcount++;
  - if (readcount == 1)
    - wait (wrt);
    - signal (mutex);
  - ......
  - /* read db */
    - readcount--;
    - if (readcount == 0)
      - signal (wrt);
      - signal (mutex);
- } while (TRUE);
Classic Problems of Synchronization
Case #3: Dinning-Philosophers Problem

- Consider 5 Philosophers
  - To Think and Eat (fried rice?)
  - Share a round table with 5 chairs
  - Each Philosopher has assigned chair and plate
  - Fried rice located at center of table

  - One chopstick between neighbor plates
- To eat Philosopher uses assigned chair, plate and grabs left chopstick followed by
  right chopstick on either side of his/her plate
- When Philosopher is not eating, s/he is thinking

- Problem:
  - Develop an algorithm to allow Philosophers to eat. Algorithm must satisfy:
    - Mutual Exclusion (One and only one Philosopher uses a chopstick at a time)
    - Avoid deadlock (Where everyone holds left chopstick and waits for right chopstick)
    - Avoid starvation (Everyone must eat eventually)
Condition variables coordinate events:

- Define one or more variables of type condition
  
  ```
  condition x, y;
  ```

- Supports two atomic operations
  
  - `x.wait()` /* Suspend the process invokes operation until `x.signal()` is invoked by another process*/
  
  - `x.signal()` /*resume one suspended process*/

- Now associate a semaphore and integer variable with each condition variable
Synchronization Mechanism
Condition Variables Implementation

- `x.wait( )`

```c
semaphore x_sem ← 0
integer x_count ← 0
x_count++{
    if (next_count > 0)
        signal (next)
    else
        signal (mutex)
    wait(x_sem)
xcount--;}
```

- `x.signal( )`

```c
semaphore x_sem ← 0
integer x_count ← 0
if (next_count > 0)
{
    next_count++; signal (next)
    signal (x_sem);
    wait (next);
    next_count--; }
```