Chapter 6: Synchronization

6.1 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 `counter` that keeps track of the number of full buffers.
  - Initially, `counter` 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.

Producer

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

Consumer

```
while (true) {
  while (counter == 0)
    ; // do nothing
  nextConsumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  counter--;
  /* consume the item in nextConsumed */
}
```
6.2 The Critical-Section Problem

- **Critical section**: when one process is executing in a critical section, no other process is to be allowed to execute in its critical section
  - the process may be changing common variables, updating a table, writing a file, and so on
  - no two processes are executing in their critical sections at the same time
- **Examples**: kernel data structure
  - (a) preemptive kernels and (b) nonpreemptive kernels

```c
while (true) {
    flag[i] = TRUE;
    turn = j;
    while ( flag[j] && turn == j);
    CRITICAL SECTION
    flag[i] = FALSE;
    REMAINDER SECTION
}
```

**Algorithm for Process Pi**

6.3 Peterson’s Solution

- A classic software-based 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.
  - if `turn == i`, then process Pi is allowed to execute in its 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 Pi is ready!
In general we can state that any solution to the critical-section problem requires a simple tool – a lock

- acquire a lock before entering a critical section
- release the lock when it exits the critical section

```c
do {
    acquire lock
    CRITICAL SECTION
    release lock
    REMAINDER SECTION
} while (TRUE);
```

### 6.4 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
    - CLI/SETI in Intel IA-32
- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptable
  - TestAndSet: Either test memory word and set value
  - Swap: Or swap contents of two memory words

### TestAndSet Instruction

- Definition:

```c
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

- Atomic: two simultaneously TestAndSet instructions are executed sequentially in some arbitrary order
  - only one process can find FALSE for target

### Solution using TestAndSet

- Shared boolean variable lock, initialized to false.
- Solution:

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

    lock = FALSE;
    // remainder section
}
```
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
  while (true) {
    key = TRUE;
    while ( key == TRUE) Swap (&lock, &key );
    //    critical section
    lock = FALSE;
    //      remainder section
  }
  ```

6.5 Semaphore

- A synchronization tool that does not require busy waiting
  - higher-level, abstract the hardware complexity
- 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
  ```c
  while (S <= 0) ; // no-op
  S--;
  ```
  ```c
  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
  Semaphore synch;    //  initialized to 1
  ```
  ```c
  in Process P1:
  Semaphore synch;    //  initialized to 1
  S1;
  signal (synch);
  in Process P2:
  wait (synch);
  S2;
  ```
Semaphore Implementation

Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.

- Must guarantee that no two processes can execute **wait ()** and **signal ()** on the same semaphore at the same time
- **spinlock**: busy waiting semaphore in critical section implementation
  - disadvantage: waste CPU cycles
    > applications may spend lots of time in critical sections
  - advantage: no context switch that takes considerable time
    > good when locks are expected to be held for short times (short critical section)

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): # of waiting processes
- 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.)

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

- Implementation of wait:
  ```c
  wait (semaphore *S){
    S->value--;
    if (S->value < 0) {
      add this process to waiting queue S->list;
      block();
    }
  }
  ```

- Implementation of signal:
  ```c
  Signal (semaphore *S){
    S->value++;
    if (S->value <= 0) {
      remove a process P from the waiting queue 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

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait (S);</td>
<td>wait (Q);</td>
</tr>
<tr>
<td>wait (Q);</td>
<td>wait (S);</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>signal (S);</td>
<td>signal (Q);</td>
</tr>
<tr>
<td>signal (Q);</td>
<td>signal (S);</td>
</tr>
</tbody>
</table>

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.
6.6 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 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

```c
while (true) {
    // produce an item
    wait (empty);
    wait (mutex);
    // add the item to the buffer
    signal (mutex);
    signal (full);
}
```

The structure of the consumer process

```c
while (true) {
    // remove an item from buffer
    wait (full);
    wait (mutex);
    // add the item to the buffer
    signal (mutex);
    signal (empty);
    // consume the removed item
    signal (mutex);
    signal (full);
}
```

### 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.
- first readers-writers problem: no reader be kept waiting unless a writer has already obtained permission to use the shared object
  - No reader should wait for other readers to finish
- second readers-writers problem: once a writer is ready, that writer performs its write ASAP.
  - If a writer is waiting to access the object, no new readers may start reading

Solution to 1st readers-writers problem: 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
while (true) {
    // writing is performed
    wait (wrt);
    signal (wrt);
}
```

The structure of a reader process

```c
while (true) {
    wait (mutex);
    readcount ++ ;
    // reading is performed
    if (readercount == 1)  wait (wrt) ;
    signal (mutex);
    readcount - - ;
    if (redacount == 0)  signal (wrt) ;
    signal (mutex) ;
}
```

If a writer is in the CS and \( n \) readers are waiting, then 1 reader is queued on wrt, and \( n-1 \) readers are queued on mutex.
**Dining-Philosophers Problem**

- Five philosophers who think and eat. Five single chopsticks.
  - Pick two chopsticks closest to her before eating. Pick one at a time.
- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

**Dining-Philosophers Problem (Cont.)**

- The structure of Philosopher $i$:

  ```
  While (true) {
    wait (chopstick[i]);
    wait (chopStick[(i + 1) % 5]);
    // eat
    signal (chopstick[i]);
    signal (chopstick[(i + 1) % 5]);
    // think
  }
  ```

  - Deadlock when each philosopher tries to grab her right chopstick
    - deadlock-free solution shown in next section

**Problems with Semaphores**

- Correct use of semaphore operations:
  - mutex is initialized to 1
  - wait (mutex) .... signal (mutex)
- timing errors are difficult to detect since they happen only for some particular execution sequences
  - signal (mutex) .... wait (mutex)
  - wait (mutex) ... wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

**6.7 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

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (...) { .... }
  ...
  procedure Pn (...) {......}
  Initialization code { .... } { ... }
}
```
### Condition Variables

- **condition** `x, y`;

- Two operations on a condition variable:
  - `x.wait()` — a process that invokes the operation is suspended.
  - `x.signal()` — resumes one of processes (if any) that invoked `x.wait()`

### Solution to Dining Philosophers

```c
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((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;
    }
}
```
Solution to Dining Philosophers (cont)

- Each philosopher \(i\) invokes the operations `pickup()` and `putdown()` in the following sequence:
  ```
  dp.pickup (i)
  EAT
  dp.putdown (i)
  ```

Monitor 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--;
  ```
- The operation \(x.signal\) can be implemented as:
  ```
  if (x-count > 0) {
    next-count++;signal(x-sem);
    wait(next);
    next-count--;
  }
  ```

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.

Chapter 6: Synchronization

6.1 Background
6.2 The Critical-Section Problem
6.3 Peterson’s Solution
6.4 Synchronization Hardware
6.5 Semaphores
6.6 Classic Problems of Synchronization
6.7 Monitors
6.8 Synchronization Examples
6.9 Atomic Transactions