Chapter 5: Process Synchronization

adapted from Silberschatz, Galvin, Gagne

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
- 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.

Producer

```
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

```
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}

Critical-Section Problem

- Consider a system consisting of n processes $\{P_0, P_1, \ldots, P_{n-1}\}$
- Each process has a segment of code, called a *critical section*, in which the process may be changing common variables, updating a table, writing a file and so on.
- Critical-section problem:
  - When one process is executing in its critical section, no other process is allowed to execute in its critical section.
  - Or in other words, no two processes are executing in their critical section at the same time.

Critical-Section Problem

```
while (true)  {
        entry section
                //request permission to enter its critical section
        critical section
        exit section
                //notify exit of critical section
        remainder section
 }
```

Solution to Critical-Section Problem

Must satisfy all three requirements:

1. **Mutual Exclusion** - If process $\mathbf{P_i}$ 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

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

```
while (true)  {
        flag[i] = TRUE;
        turn = j;
        while (flag[j] && turn == j);
                critical section
        flag[i] = FALSE;
                remainder section
}
```

Algorithm for Process P_i

- Entry section
  - flag[i] = TRUE; // process P_i is ready
  - turn = j; // P_j can go first
  - while (flag[j] && turn == j); // wait if P_j is ready and P_j's turn
- Exit section
  - flag[i] = FALSE; // process P_i is out of critical section

Correctness of Peterson's Solution

- Mutual exclusion
  - flag[0] == flag[1] == true
  - turn == 0 or turn == 1, not both
- Progress
  - If only P_0 to enter critical section
    - flag[1] == false, thus P_0 enters critical section
  - If both P_0 and P_1 to enter critical section
    - flag[0] == flag[1] == true and (turn == 0 or turn == 1)
    - One of P_0 and P_1 will be selected
- Bounded-waiting
  - If both P_0 and P_1 to enter critical section, and P_1 selected first
  - When P_1 exit, flag[1] = false
    - If P_0 runs fast: flag[1] == false, P_0 enters critical section
    - If P_1 runs fast: flag[1] = true, but turn = 0, P_0 enters critical section

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Locking
  - Protecting the critical section through the use of locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - All other processors need to be “paused” when one of them enters critical section
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable

Atomic Instructions

- Different architectures have different implementation of atomic instructions
  - x86, ARM, MIPS
  - No standard interface and implementation
- Two categories:
  - test_and_set(): test memory word and set value
  - compare_and_swap(): swap contents of two memory words

Solution to Critical-section Problem Using Locks

```
do {
        acquire lock
                critical section
        release lock
                remainder section
} while (TRUE);
```

test_and_set

- Definition:

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

Solution using test_and_set

- Shared boolean variable lock., initialized to **FALSE**.
- Solution:

```
do {
        while ( test_and_set (&lock));   // do nothing

                critical section

        lock = FALSE;

                remainder section

} while (TRUE);
```

compare_and_swap Instruction

- Definition:

```
void compare_and_swap (int *value, int expected_value,
int new_value)
    {
        int temp = *value;
        if(*value == expected)
            *value = new_value;
        return temp;
    }
```

Solution using compare_and_swap()

- Shared intvariable *lock* initialized to 0
- Solution:

```
do {
        while(compare_and_swap (&lock, 0, 1) != 0);

                critical section

        lock = 0;

                remainder section

} while (TRUE);
```

Bounded-waiting

- Test_and_set() and compare_and_swap() satisfies *mutual exclusion and progress*
- Bounded-waiting is not satisfied
  - A process can wait indefinitely to enter its critical section while another process re-enters its critical section multiple times

Bounded-Waiting Mutual Exclusion

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
                key = test_and_set(&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 (TRUE);
```

Bounded-Waiting: Discussion

- Support n processes
  - boolean waiting[n]
  - boolean lock
    - initially FALSE
- A process can enter the critical section if either **waiting[i] == FALSE** or **key == FALSE**
  - key is local variable
  - All process must execute test_and_set() at least once
  - The first one call test_and_set() with lock==FALSE wins
    - key = FALSE
  - lock == TRUE after the first process executes test_and_set()
    - key = TRUE
- *Mutual exclusion* and *progress* are satisfied

Bounded-Waiting: Discussion (Cont)

- When a process leaves the critical section
- It scans the array waiting[n] in a cyclic order (i+1, i+2, … n-1, 0, 1, …, i-1)
- The first process with waiting[j] == TRUE enters the critical section next
- Bounded-waiting: Any process waiting to enter its critical section will do so within n-1 turns.
- If no other process to enter critical section: i==j
  - lock = FALSE

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
- 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

- 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

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

Semaphore for Sequentialization

- Process P_1 and P_2
  - P_1 has a statement S_1
  - P_2 has a statement S_2
- Requirement S_2 executed only after S_1 has completed
  - Semaphore *synch* // initialized to 0

- P_1
  - S_1

    signal (synch)
- P_2
  - wait (synch)

    S_2

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue.
  - typedef struct {

    int value;

    struct process *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:

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

- Implementation of signal:

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

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.
  - Need to disable interrupt on single-processor machine
  - use compare_and_wait() on multi-core architecture

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

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended
  - Maintain a list as a LIFO (last-in, first-out) queue

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.

Bounded Buffer Problem (Cont.)

- 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);
```

Bounded Buffer Problem (Cont.)

- The structure of the **consumer** process

```
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
- 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

```
do {
        wait (wrt) ;

        //    writing is performed

        signal (wrt) ;
} while (TRUE);
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

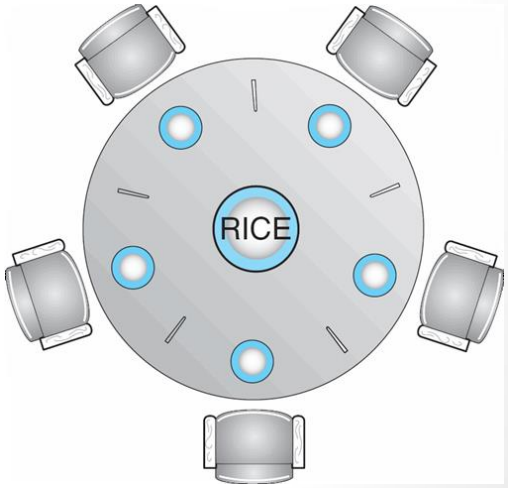
```
do {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
                wait (wrt) ;
        signal (mutex)

            // reading is performed

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

Dining-Philosophers Problem

- Five philosophers sitting at a round table
  - Bowl of rice (data set)
  - Five single chopsticks
- Eating
  - Pick up the two chopsticks that are closest to her
  - Pick up on chopsticks at a time
  - Put down both chopsticks when finish
- Share variables
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1



Dining-Philosophers Problem (Cont.)

- 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);
```

Dining-Philosophers Problem (Cont.)

- Deadlock may happen
  - All five philosophers are hungry at the same time
  - Each grabs her left chopstick
  - All elements of chopstick[] will become 0
- Solutions
  - Allow at most four philosophers to be sitting simultaneously at the table
  - Allow a philosopher to pick up her chopsticks only if both chopsticks are available (using critical sections)
  - Odd-numbered philosopher picks up first her left chopstick and then right, while even-numbered philosopher picks up first her right chopstick and then left

POSIX Synchronization

- Semaphores
  - Named Semaphore
  - Unnamed semaphore
- Pthread synchronization
  - mutex locks
  - condition variables

End of Chapter 5