Chapter 7

7.1

Classic Problems of Synchronization (Bounded Buffer, Readers Writers and Dining Philosophers)

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

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {
   ...
  /* produce an item in next_produced */
   ...
  wait(empty);
  wait(mutex);
   ...
  /* add next produced 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 next_consumed */
    ...
  signal(mutex);
  signal(empty);
    ...
  /* consume the item in next consumed */
    ...
} 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 considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore **rw_mutex** initialized to 1
  - Semaphore **mutex** initialized to 1
  - Integer **read_count** initialized to 0

- The structure of a writer process

```
do {
```

```
    wait(rw_mutex);
        ...
    /* writing is performed */
        ...
    signal(rw_mutex);
} while (true);
```

- The structure of a reader process

```
do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
      wait(rw_mutex);
    signal(mutex);
        ...
    /* reading is performed */
        ...
    wait(mutex);
    read count--;
    if (read_count == 0)
    signal(rw_mutex);
    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 the 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 alternating thinking and eating
  IF chopsticks are available they should be able to eat if they are hungry

- 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 6 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [6] initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
   wait (chopstick[i] );
       wait (chopStick[ (i + 1) % 6] );

              //  eat

       signal (chopstick[i] );
       signal (chopstick[ (i + 1) % 6] );

            //  think

} while (TRUE);
```

- What is the problem with this algorithm?

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
   wait(mutex);
   wait (chopstick[i] );
      wait (chopStick[ (i + 1) % 6] );
   signal(mutex);
```

```
// eat

signal (chopstick[i] );
signal (chopstick[ (i + 1) % 6] );

// think

} while (TRUE);
```

- mutex =1 (initially) only one can eat at a time.. over synchronizes the code.
- mutex =2,3,4 (initially) still has starvation
- mutex =6 back to deadlocks

- **Deadlock handling**
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section. We will present this solution later in monitors)
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

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 are possible.

1. What is the purpose of the mutex semaphore in the implementation of the bounded-buffer problem using semaphores?

A) It indicates the number of empty slots in the buffer.

B) It indicates the number of occupied slots in the buffer.

C) It controls access to the shared buffer.

D) It ensures mutual exclusion.

2. The first readers-writers problem ____.

A) requires that, once a writer is ready, that writer performs its write as soon as possible.

B) is not used to test synchronization primitives.

C) requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared database.

D) requires that no reader will be kept waiting unless a reader has already obtained permission to use the shared database.

3. How many philosophers may eat simultaneously in the Dining Philosophers problem with 5 philosophers?

A) 1

B) 2

C) 3

D) 5