Operating System

Ch07: Synchronization example

BeomSeok Kim
Department of Computer Engineering
KyungHee University
passion0822@khu.ac.kr

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded Buffer Problem

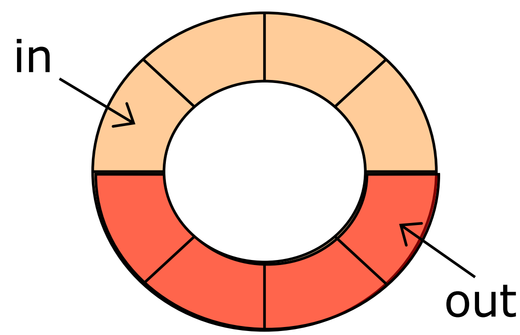
- No synchronization

Producer

```
void produce(data)
{

   while (count==N) ;
   buffer[in] = data;
   in = (in+1) % N;
   count++;

}
```

```
int count;
```

```
struct item buffer[N];
int in, out, count;
```



Consumer

```
void consume(data)
{

   while (count==0) ;
   data = buffer[out];
   out = (out+1) % N;
   count--;

}
```

Bounded Buffer Problem

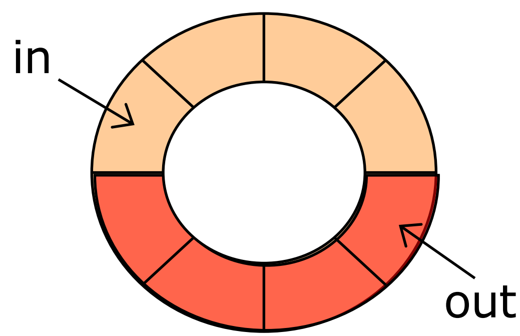
- Implementation with semaphores

Producer

```
void produce(data)
{

   wait (empty);
   wait (mutex);
   buffer[in] = data;
   in = (in+1) % N;
   signal (mutex);
   signal (full);

}
```

```
Semaphore
    mutex = 1;
    empty = N;
    full = 0;
```

```
struct item buffer[N];
int in, out;
```



Consumer

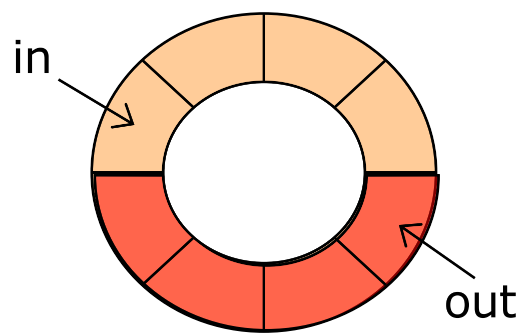
```
void consume(data)
{

   wait (full);
   wait (mutex);
   data = buffer[out];
   out = (out+1) % N;
   signal (mutex);
   signal (empty);

}
```

Bounded Buffer Problem

- Implementation with mutex lock and condition variables

```
MutexLock mutex;
CondVar not_full,
        not_empty;
```

```
struct item buffer[N];
int in, out, count;
```



Producer

```
void produce(data)
{

   lock (mutex);
   while (count==N)
      wait (not_full);
   buffer[in] = data;
   in = (in+1) % N;
   count++;
   signal (not_empty);
   unlock (mutex);
}
```

Consumer

```
void consume(data)
{
   lock (mutex);
   while (count==0)
      wait (not_empty);
   data = buffer[out];
   out = (out+1) % N;
   count--;
   signal (not_full);
   unlock (mutex);
}
```

Readers-Writers Problem

- Readers-Writers problem
  - ✓ An object is shared among several threads
  - ✓ Some threads only read the object, others only write it
  - ✓ We can allow multiple readers at a time
  - ✓ We can only allow one writer at a time
  - ✓ Two cases
    - ➢ No reader should wait for other readers to finish simply because a writer is waiting
    - ➢ Once a writer is ready, that writer performs its write ASAP

- Implementation with semaphores
  - ✓ readcount - # of threads reading object
  - ✓ mutex – control access to readcount
  - ✓ wrt – exclusive writing or reading

Readers-Writers Problem

- Implementation with semaphores

```
// number of readers
int readcount = 0;
// mutex for readcount
Semaphore mutex = 1;
// mutex for reading/writing
Semaphore wrt = 1;

void Writer ()
{
   wait (wrt);
   …
   Write
   …
   signal (wrt);
}
```

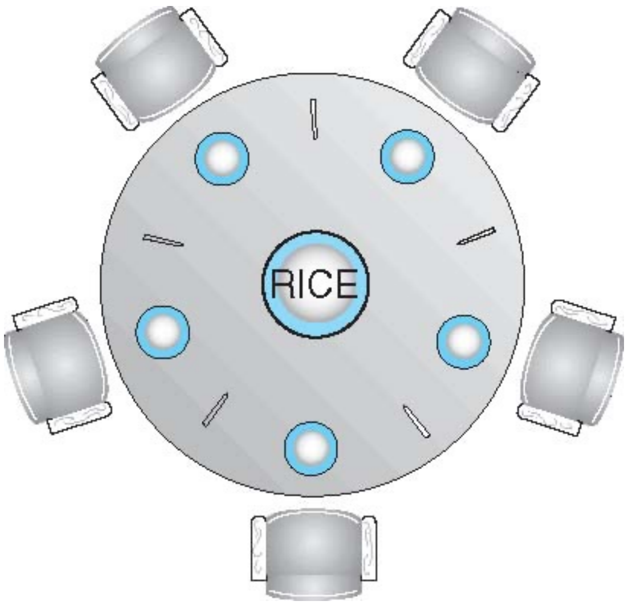
```
void Reader ()
{
   wait (mutex);
   readcount++;
   if (readcount == 1)
      wait (wrt);
   signal (mutex);
   …
   Read
   …
   wait (mutex);
   readcount--;
   if (readcount == 0)
      signal (wrt);
   signal (mutex);
}
```

Readers-Writers Problem

- If there is a writer
  - ✓ The first reader blocks on wrt
  - ✓ All other readers will then block on mutex
- Once a writer exits, all readers can fall through
  - ✓ Which reader gets to go first?
- The last reader to exit signals waiting writer
  - ✓ Can new readers get in while writer is waiting?
- When writers exits, if there is both a reader and writer waiting, which one goes next is up to scheduler

Dining Philosopher

- Dining philosopher problem
  - ✓ Dijkstra, 1965
  - ✓ Life of a philosopher: Repeat forever
    - ➢ Thinking
    - ➢ Getting hungry
    - ➢ Getting two chopsticks
    - ➢ Eating



Dining Philosopher

- A simple solution

```
Semaphore chopstick[N];  // initialized to 1
void philosopher (int i)
{
    while (1)  {
        think ();
        wait (chopstick[i]);
        wait (chopstick[(i+1) % N];
        eat ();
        signal (chopstick[i]);
        signal (chopstick[(i+1) % N];
    }
}
```

$\Rightarrow$ Problem: causes deadlock

Dining Philosopher

- Deadlock-free version: starvation?

```
#define N         5
#define L(i)      ((i+N-1)%N)
#define R(i)      ((i+1)%N)
void philosopher (int i)  {
  while (1)  {
    think ();
    pickup (i);
    eat();
    putdown (i);
  }
}
void test (int i)  {
  if (state[i]==HUNGRY &&
      state[L(i)]!=EATING &&
      state[R(i)]!=EATING)  {
     state[i] = EATING;
     signal (s[i]);
 }
```

```
Semaphore mutex = 1;
Semaphore s[N];
int state[N];

void pickup (int i)  {
  wait (mutex);
  state[i] = HUNGRY;
  test (i);
  signal (mutex);
  wait (s[i]);
}
void putdown (int i)  {
  wait (mutex);
  state[i] = THINKING;
  test (L(i));
  test (R(i));
  signal (mutex);
}
```

Dining Philosophers

- Monitor implementation

```
monitor dp
{
    enum {THINKING, HUNGRY, EATING} state[5];
    condition self[5];
    void pickup(int i);
    void putdown(int i);
    void test(int i);
    void init() {
        for (int i = 0; i < 5; i++)
        state[i] = EATING;
    }
}
```

Dining Philosophers

- Monitor implementation

```
void pickup(int i) {
    state[i] = HUNGRY;
    test(i);
    if (state[i] != EATING)
        self[i].wait();
}
```

```
void putdown(int i) {
    state[i] = THINKING;
    // test left and right
    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();
    }
}
```

Synchronization Tools in Real World

- POSIX synchronization
  - ✓ POSIX semaphores
  - ✓ POSIX mutex locks (mutexes)
  - ✓ POSIX condition variables
- Java synchronization
  - ✓ Monitor: synchronized
- C/C++/Java
  - ✓ Semaphores
  - ✓ Mutexes & Condition variables

POSIX Semaphores

- POSIX semaphore libraries
  - ✓ `#include <semaphore.h>`
  - ✓ `int sem_init(sem_t *sem, int pshared, unsigned int value);`
  - ✓ `int sem_wait(sem_t *sem);`
  - ✓ `int sem_trywait(sem_t *sem);`
  - ✓ `int sem_post(sem_t *sem);`
  - ✓ `int sem_getvalue(sem_t *sem, int *sval);`
  - ✓ `int sem_destroy(sem_t *sem);`
  - ✓ return: 0 if OK, non-zero value on error

POSIX Mutex Locks

- Pthread libraries for mutexes
  - ✓ `#include <pthread.h>`
  - ✓ `int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *mattr);`
  - ✓ `int pthread_mutex_destroy(pthread_mutex_t *mutex);`
  - ✓ `int pthread_mutex_lock(pthread_mutex_t *mutex);`
  - ✓ `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
  - ✓ `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
  - ✓ return: 0 if OK, non-zero value on error

POSIX Condition Variables

- Pthread libraries for condition variables
  - ✓ `#include <pthread.h>`
  - ✓ `int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cattr);`
  - ✓ `int pthread_cond_destroy(pthread_cond_t *cond);`
  - ✓ `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`
  - ✓ `int pthread_cond_signal(pthread_cond_t *cond);`
  - ✓ `int pthread_cond_broadcast(pthread_cond_t *cond);`
  - ✓ return: 0 if OK, non-zero value on error

Bounded Buffer with POSIX Semaphores

```
sem_t mutex, full, empty;
/*   sem_init (&mutex, 0, 1);
     sem_init (&full, 0, 0); sem_init (&empty, 0, N);   */
buffer resources[N];

void producer (resource x)  {
   sem_wait (&empty);
   sem_wait (&mutex);
   add “x” to array “resources”;
   sem_post (&mutex);
   sem_post (&full);
}
void consumer (resource *x)  {
   sem_wait (&full);
   sem_wait (&mutex);
   *x = get resource from array “resources”
   sem_post (&mutex);
   sem_post (&empty);
}
```

Bounded Buffer with POSIX Mutexes & Cond. Vars

```
pthread_mutex_t mutex;
pthread_cond_t not_full, not_empty;
buffer resources[N];
void producer (resource x)  {
    pthread_mutex_lock (&mutex);
    while (array "resources" is full)
        pthread_cond_wait (&not_full, &mutex);
    add "x" to array "resources";
    pthread_cond_signal (&not_empty);
    pthread_mutex_unlock (&mutex);
}
void consumer (resource *x)  {
    pthread_mutex_lock (&mutex);
    while (array "resources" is empty)
        pthread_cond_wait (&not_empty, &mutex);
    *x = get resource from array "resources"
    pthread_cond_signal (&not_full);
    pthread_mutex_unlock (&mutex);
}
```

Bounded Buffer with Java Monitor

```
public class BoundedBuffer<E>
{
   private static final int BUFFER_SIZE = 5;

   private int count, in, out;
   private E[] buffer;

   public BoundedBuffer() {
      count = 0;
      in = 0;
      out = 0;
      buffer = (E[]) new Object[BUFFER_SIZE];
   }

   /* Producers call this method */
   public synchronized void insert(E item) {
      /* See Figure 7.11 */
   }

   /* Consumers call this method */
   public synchronized E remove() {
      /* See Figure 7.11 */
   }
}
```

Thank You!
Q&A