7. Synchronization Examples

ECE30021/ITP30002 Operating Systems

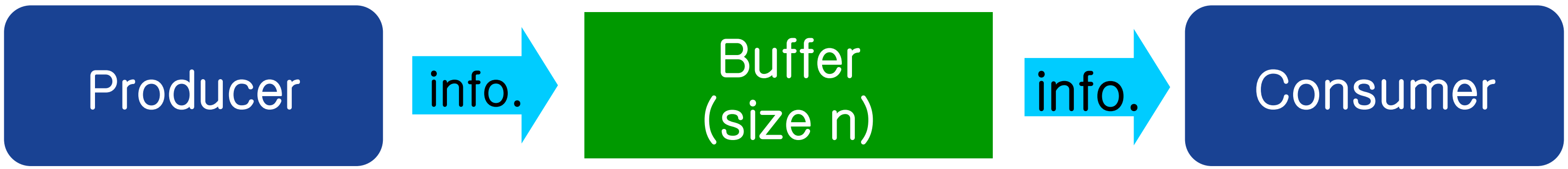
Agenda

- Classical Problems of Synchronization
- Synchronization within the Kernel
- POSIX Synchronization
- Synchronization in Java
- Alternative Approaches

Classical Problems of Synchronization

- The bounded-buffer problem
- The readers-writers problem
- The dining-philosophers problem

The Bounded-Buffer Problem



- If the buffer is full, the producer must wait until the consumer deletes an item.
  - Producer needs an empty space
  - # of empty slot is represented by a semaphore *empty*
- If the buffer is empty, the consumer must wait until the producer adds an item.
  - Consumer needs an item
  - # of item is represented by a semaphore *full*

The Bounded-Buffer Problem

- Producer-consumer problem with bounded buffer
  - Semaphores: full = 0, empty = n, mutex = 1;
  - Producer

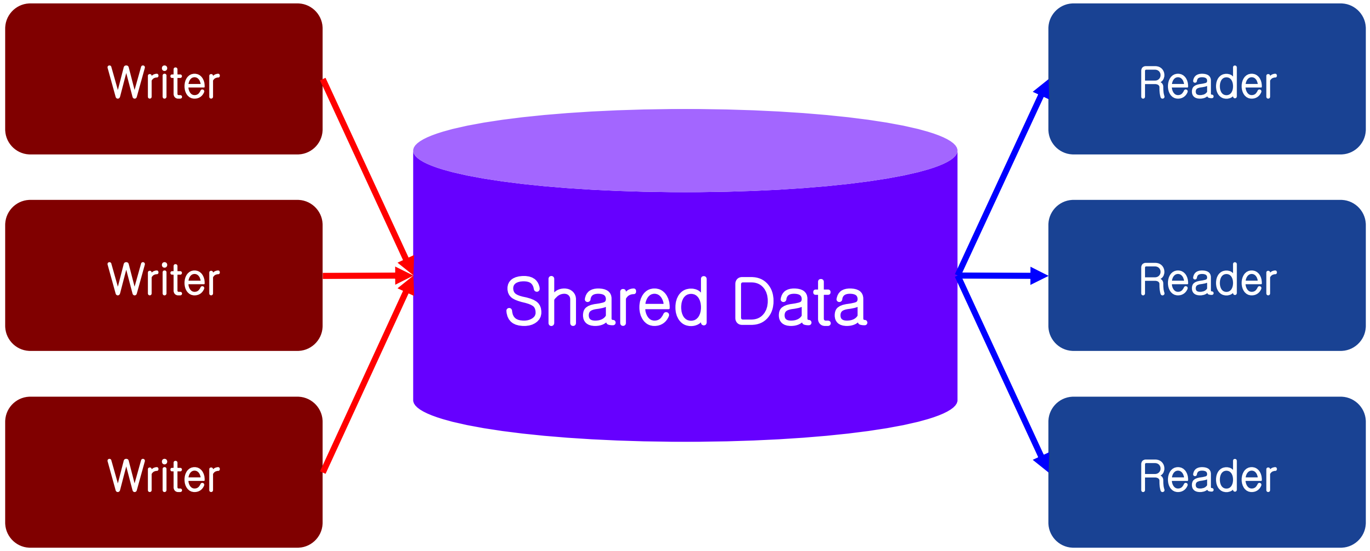
```
do {
        …
    produce an item in nextp
        …
    wait(empty);
    wait(mutex);      // acquire()
        …
    add nextp to buffer
        …
    signal(mutex);    // release()
    signal(full);
} while (1);
```

  - Consumer

```
do {
    wait(full);
    wait(mutex);      // acquire()
        …
    remove an item from buffer to nextc
        …
    signal(mutex);    // release()
    signal(empty);
        …
    consume the item in nextc
        …
} while (1);
```

The Readers-Writers Problem

- There are multiple readers and writers to access a shared data
  - Readers can access database simultaneously.
  - When a writer is accessing the shared data, no other thread can access it.



The Readers-Writers Problem

- Behavior of a writer
  - If a thread is in the critical section, all writers must wait.
  - The writer can enter the critical section only when no thread is in its critical section.
    - It should prevent all threads from entering the critical section.

- Behavior of a reader
  - If no writer is in its critical section, the reader can enter the critical section.
  - Otherwise, the reader should wait until the writer leaves the critical section.
  - When a reader is in its critical section, any reader can enter the critical section, but no writer can.
    - Condition for the first reader is different from the following readers.

The Readers-Writers Problem

- Shared data
  - semaphore mutex=1, wrt=1;
  - int readcount = 0;
    - # of readers in critical section
- Writer

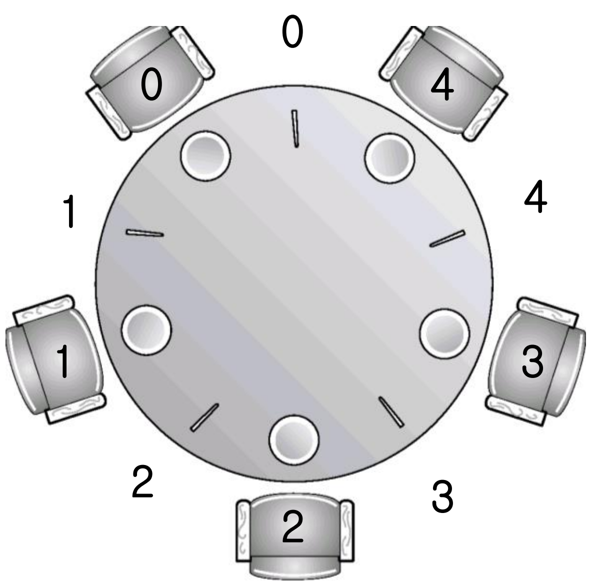
```
wait(wrt);
   …
writing is performed
   …
signal(wrt);
```

- Reader

```
wait(mutex);
readcount++;
if (readcount == 1)
      wait(wrt);
signal(mutex);
      …
reading is performed
      …
wait(mutex);
readcount--;
if (readcount == 0)
      signal(wrt);
signal(mutex);
```

The Dining Philosophers Problem

- Problem definition
  - 5 philosophers sitting on a circular table
    - Thinking or eating
  - 5 bowls, 5 single chopsticks
  - No interaction with colleagues
  - To eat, the philosopher should pick up two chopsticks closest to her
  - A philosopher can pick up only one chopstick at a time
  - When she finish eating, she release chopsticks



- Solution should be deadlock-free and starvation free

The Dining Philosophers Problem

- A possible solution, but deadlock can occur

```
do {
    wait(chopstick[i]);                 // pick up left chopstick
    wait(chopstick[(i+1) % 5]);         // pick up right chopstick
     ...
    eat
     ...
    signal(chopstick[i]);               // release left chopstick
    signal(chopstick[(i+1) % 5]);       // release right chopstick
    ...
    think
    ...
} while (TRUE);
```

Deadlock

- Deadlock

P_0	P_1
wait(S);	*wait(Q);*
wait(Q);	*wait(S);*
⋮	⋮
signal(S);	*signal(Q);*
signal(Q);	*signal(S);*

- Necessary conditions for deadlock
  1. Mutual exclusion
  2. Hold and wait 동시에 2개를 잡는다
  3. No preemption [illegible]
  4. Circular wait 짝수 오른쪽, 홀수 왼쪽

Dining Philosophers Solution Using Monitor

- Data structures
  - enum { thinking, hungry, eating } state[5];
  - condition self[5];

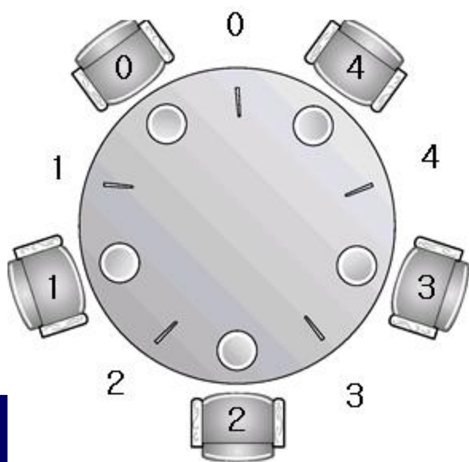
- Process of i-th philosopher implemented by a monitor dp

```
dp.pickup(i);                  // entry section
    …
    Eat                        // critical section
    …
dp.putdown(i);                 // exit section
```

Dining Philosophers Solution Using Monitor

```
monitor diningPhilosophers {
     int state[5];
     static final int THINKING = 0;
     static final int HUNGRY = 1;
     static final int EATING = 2;
     condition self[5];

     void initialization_code {
        for (int i = 0; i < 5; i++)
           state[i] = THINKING;
    }

    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();
       }
    }
}
```

Deadlock-free, but not starvation-free



Agenda

- Classical Problems of Synchronization
- <u>Synchronization within the Kernel</u>
- POSIX Synchronization
- Synchronization in Java
- Alternative Approaches

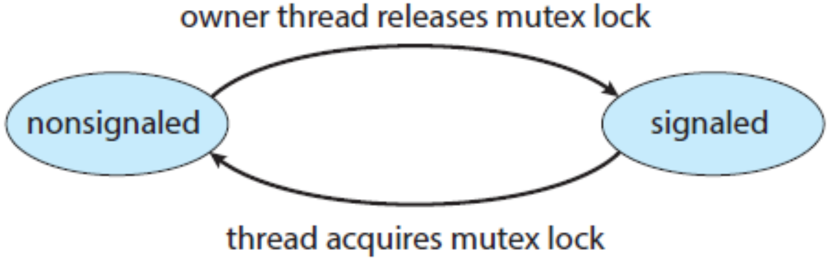
Synchronization in Windows Kernel

- Windows kernel
  - A multithreaded kernel
  - Provides support for real-time applications
  - Provides support for multiple processors

- Accessing global variables
  - Single-processor system: mask interrupt (kerel 모드에서만 가능)
  - Multi-processor system: spinlocks
    - Only to protect short code segments
    - The kernel ensures that a thread will never be preempted while holding a spinlock

Synchronization in Windows

- For thread synchronization outside the kernel, windows provides dispatcher objects
  - Mutex locks, semaphores, events, timers
- States of dispatcher objects
  - Signaled state: the object is available
    - Thread will not block when acquiring the object.
  - Nonsignaled state: the object is not available
    - Thread will block when attempting to acquire the object.
    - Its state changes from ready to waiting, and the thread is placed in a waiting queue for that object.



Synchronization in Windows

- Critical section object: a user-mode mutex that can often be acquired and released without kernel intervention.
  - A critical-section object first uses a spinlock while waiting for the other thread to release the object.
    - ➔ Does not allocate kernel object. (efficient)
  - If it spins too long, the acquiring thread will then allocate a kernel mutex and yield its CPU.

Synchronization in Linux Kernel

- From v.2.6., Linux kernel is fully preemptive
- Atomic integer (atomic_t)
  - All math operations are performed without interruption

Ex) atomic_t counter;
int value;

Atomic Operation	*Effect*
`atomic_set(&counter,5);`	`counter = 5`
`atomic_add(10,&counter);`	`counter = counter + 10`
`atomic_sub(4,&counter);`	`counter = counter - 4`
`atomic_inc(&counter);`	`counter = counter + 1`
`value = atomic_read(&counter);`	`value = 12`

Synchronization in Linux Kernel

- Mutex locks
  - int mutex_init(mutex_t *mp, int type, void *arg);
  - int mutex_lock(mutex_t *mp);
  - int mutex_trylock(mutex_t *mp);
  - int mutex_unlock(mutex_t *mp);
  - int mutex_consistent(mutex_t *mp);
    - Makes a mutex consistent
  - int mutex_destroy(mutex_t *mp);

Synchronization in Linux Kernel

- Spinlocks
  - spin_lock(), spin_unlock(), etc.
  - Useful for short duration
  - Inappropriate on single processing core
- Enabling/disabling kernel preemption
  - preempt_disable(), preempt_enable()

Single processor	Multiprocessor
Disable kernel preemption	Acquire spinlock
Enable kernel preemption	Release spinlock

Agenda

- Classical Problems of Synchronization
- Synchronization within the Kernel
- <u>POSIX Synchronization</u>
- Synchronization in Java
- Alternative Approaches

POSIX Synchronization

- POSIX mutex locks
  - Creation and initialization

```
#include <pthread.h>

pthread_mutex_t mutex;                    // global declaration
pthread_mutex_init(&mutex,NULL);   // call before first lock
```

pthread_mutex_t;

or

pthread_mutex_destroy();

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

  - Critical section

```
pthread_mutex_lock(&mutex);

/* critical section */

pthread_mutex_unlock(&mutex);
```

POSIX Synchronization

- Named semaphores
  - Multiple unrelated processes can easily use a common semaphore
  - Creation and initialization

```
#include <semaphore.h>
sem_t *sem;                    // global declaration

sem = sem_open("SEM", O_CREAT, 0666, 1);
```

?

  - Critical-section

wait

```
sem_wait(sem);                 // acquire the semaphore

/* critical section */

sem_post(sem);                 // release the semaphore
```

signal

named semaphore : sem_close( ), sem_unlink( )

POSIX Synchronization

- Unnamed semaphores
  - Creation and initialization

```
#include <semaphore.h>
sem_t sem;                 // global declaration

sem_init(&sem, 0, 1);
```

?

unnamed semaphore:
sem_destroy()

  - Critical-section

```
sem_wait(&sem);            // acquire the semaphore

/* critical section */

sem_post(&sem);            // release the semaphore
```

POSIX Synchronization

- Condition variables
  - Combined with mutex locks instead of monitors
  - Initialization

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex, NULL);
pthread_cond_init(&cond_var, NULL);
```

  - Critical-section

```
Thread A
    pthread_mutex_lock(&mutex);
    while (a != b)
        pthread_cond_wait(&cond_var, &mutex);
    pthread_mutex_unlock(&mutex);
```

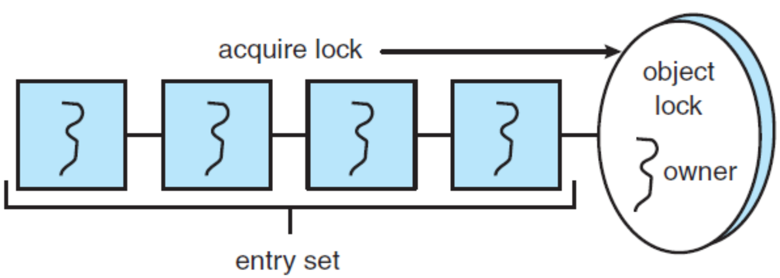
```
Thread B
    pthread_mutex_lock(&mutex);
    a = b;
    pthread_cond_signal(&cond_var);
    pthread_mutex_unlock(&mutex);
```

Agenda

- Classical Problems of Synchronization
- Synchronization within the Kernel
- POSIX Synchronization
- <u>Synchronization in Java</u>
- Alternative Approaches

Synchronization in Java

- Java monitors
  - Synchronized methods
  - Entry set
    - Threads waiting for the lock
  - Releasing the lock
    - Resumes an arbitrarily chosen thread
    - In practice, FIFO policy




```
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 */
   }
}
```

Synchronization in Java

- Producer-Consumer in Java

```
/* Producers call this method */
public synchronized void insert(E item) {
   while (count == BUFFER_SIZE) {
      try {
         wait();
      }
      catch (InterruptedException ie) { }
   }

   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
   count++;

   notify();
}
```

```
/* Consumers call this method */
public synchronized E remove() {
   E item;

   while (count == 0) {
      try {
         wait();
      }
      catch (InterruptedException ie) { }
   }

   item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   count--;

   notify();

   return item;
}
```

Synchronization in Java

- Java monitors
  - If the thread that has the lock is unable to continue, it calls wait().

    Ex) Calling insert() when the buffer is full

    1. The thread releases the lock for the object.
    2. The state of the thread is set to blocked.
    3. The thread is placed in the wait set for the object.
  - Other thread can call notify() to wake up a waiting thread
    1. Picks an arbitrary thread T from the list of threads in the wait set
    2. Moves T from the wait set to the entry set
       - When the lock is released, JVM arbitrarily chooses a thread to run
    3. Sets the state of T from blocked to runnable

Synchronization in Java

- Semaphores
  - Constructor
    - Semaphore(int value);
  - Critical-section

```
Semaphore sem = new Semaphore(1);

try {
   sem.acquire();
   /* critical section */
}
catch (InterruptedException ie) { }
finally {
   sem.release();
}
```

Agenda

- Classical Problems of Synchronization
- Synchronization within the Kernel
- POSIX Synchronization
- Synchronization in Java
- <u>Alternative Approaches</u>

Transactional Memory

- Memory transaction: a sequence of memory read-write operations that are atomic.
  - Originated from database theory
  - SW transactional memory(STM) / HW transactional memory(HTM)

Ex)

```
void update ()
{
    atomic {
        /* modify shared data */
    }
}
```

  - Traditional implementation

```
void update ()
{
    acquire();
    /* modify shared data */
    release();
}
```

OpenMP

- Parallel region
  - #pragma omp parallel

- Critical section
  - #pragma omp critical

Ex)

```
void update(int value)
{
    #pragma omp critical
    {
        counter += value;
    }
}
```