6.3 Java Monitors

• Java monitors
  - Primary mechanism for providing mutual exclusion and synchronization in multithreaded Java applications
  - Signal-and-continue monitors
    - Allow a thread to signal that the monitor will soon become available
    - Maintain a lock on monitor until thread exits monitor
  - Keyword synchronized
    - Imposes mutual exclusion on an object in Java

Java monitors

• notify is a variant of signal
• After c.notify is called
  - Calling process continues (Hansen’s semantics)
  - Woken-up process continues when caller exits

• Problem
  - Caller may wake up multiple processes, Pi, Pj, Pk, …
  - Pi could change condition on which Pj was blocked
- As the notifying process (p3) continues, other processes (p2) or even the notifier itself may change the condition of C1 before p1 enters critical section.

Java monitors

- Solution instead of:  
  - if (!condition) c.wait
  - use:  
  - while (!condition) c.wait

- signal vs notify  
  - (Beware: There is no universal terminology)  
  - signal may involve caller "stepping aside"  
  - notify usually has caller continuing  
  - signal "simpler," but notify may be more efficiently implemented  
  - Less context switching

JAVA Monitors

- An object may have synchronized method  
  - Only one thread at a time can execute a synchronized method on the object  
  - If there are several synchronized methods, only one synchronized method may be active on an object at once  
  - JAVA doesn't have conditional variables  
  - Each object can be viewed as having one conditional variable  
  - When a thread unblocked from wait, the condition need to be reevaluated every time before entering the critical section (use while loop for testing)  
  - Remember Hansen's semantics  

- Methods  
  - notify: choose one of the waiting threads in the queue  
  - notifyAll: awaken all the blocked threads putting them in the ready queue  
  - wait: calling thread is blocked and put in the waiting queue  
  - sleep: calling thread is blocked and be awakened later
JAVA Monitors

- Synchronized statement
  ```java
  void method () {
      ...
      synchronized (anyObject) {
          anyCode();
      }
  }
  ```

- Synchronized method
  ```java
  synchronized method () {
      anyCode();
  }
  ```

- Synchronized wrapper
  ```java
  List list = Collections.synchronizedList(new LinkedList(...));
  ```

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

- Java wait method
  - Calling thread releases lock on monitor
  - Waits for condition variable
  - After calling `wait()`, thread is placed in wait set
  - Thread remains in wait set until signaled by another thread

- Condition variable is implicit in Java
  - A thread may be signaled, reenter the monitor and find that the condition on which it waited has not been met

Conclusion

- Drawbacks of monitors
  - Programming language concept
    - The compiler must recognize them and arrange for the mutual exclusion somehow
    - Most popular programming languages do not have monitor
    - (On the contrary) what about semaphore?
      - Add two short assembly language routines to the library
      - The compilers do not have to know about the semaphore
    - Distributed systems (for both semaphores and monitors)
      - Approaches so far depended on the use of shared memory and TSL support
      - Is distributed system environment, nothing works!!

- Conclusion
  - Semaphores → too low level
  - Monitors → not usable except in a few programming languages
6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

- Threads issue signals via methods `notify` or `notifyAll`
  - `notify` method
    - wakes one thread waiting to enter the monitor
  - `notifyAll` method
    - Wakes all waiting threads

More Synchronization Problem

- To understand the nature of the concurrency and synchronization problems, we study some famous synchronization problems that are analogical to the real problems in many concurrent/distributed applications

- Solutions to those problems must satisfy
  - Fair
  - No starvation
  - No deadlock
Readers/Writers Problem

- Extension of basic CS problem (Courtois, Heymans, and Parnas, 1971)
- Two types of processes entering a CS:
  - Only one Writer may be inside CS, (exclusive) or
  - Any number of Readers may be inside CS

Readers-Writers Problem

- First reader-writer problem
  - While a writer is writing, the readers have to wait
  - The readers don’t have to wait for other readers to finish
  - When a writer is waiting, a new reader may start reading
  - Readers have priority over writers
  - (Problem) starvation of a writer

- Second reader-writer problem
  - When a writer is waiting, no new readers may start reading
  - Writers have priority over readers
  - (Problem) starvation of a reader

- Prevent starvation of either process type:
  1. A new reader should not start during a read sequence if there is a writer waiting
  2. All readers waiting at the end a writer operation should have priority over the next writer

Solution using monitor

```java
monitor Readers_Writers {
    int readCount = 0, writing = 0;
    condition OK_R, OK_W;
    start_read() { 
        if (writing || !empty(OK_W)) OK_R.wait;
        readCount = readCount + 1;
        OK_R.signal;
    } 
    end_read() {
        readCount = readCount - 1;
        if (readCount == 0) OK_W.signal;
    } 
    start_write() {
        if ((readCount != 0) || writing) OK_W.wait;
        writing = 1;
    } 
    end_write() {
        writing = 0;
        if (!empty(OK_R)) OK_R.signal;
        else OK_W.signal;
    }
}
```
Dining Philosophers

Problem statement:
Five philosophers sit around a circular table. Each leads a simple life alternating between thinking and eating spaghetti. In front of each philosopher is a dish of spaghetti that is constantly replenished by a dedicated wait staff. There are exactly five forks on the table, one between each adjacent pair of philosophers. Eating spaghetti (in the most proper manner) requires that a philosopher use both adjacent forks (simultaneously). Develop a concurrent program free of deadlock and indefinite postponement that models the activities of the philosophers.

Dining Philosopher behavior.

```c
void typicalPhilosopher()
{
    // Infinite loop
    while (true)
    {
        think();
        eat();
    } // end while
} // end typicalPhilosopher
```

Dining Philosophers

- Each philosopher needs both forks to eat
- Requirements
  - No philosopher must starve
  - Free of deadlock
  - Guarantee fairness
  - Free of indefinite postponement
  - Enforce mutual exclusion
  - Two philosophers cannot use the same fork at once
  - Non-neighbors may eat at the same time
- The problems of mutual exclusion, deadlock and indefinite postponement lie in the implementation of method eat.
Dining Philosophers

Implementation of method eat.

```c
void eat()
{
    pickUpLeftFork();
    pickUpRightFork();
    eatForSomeTime();
    putDownRightFork();
    putDownLeftFork();
} // eat
```

Dining philosophers

```c
philosopher(i) {
    while (1) {
        think(i);
        grab_forks(i);
        eat(i);
        return_forks(i);
    }

    grab_forks(i)  { P(f[i]); P(f[(i+1)%5]) }
    return_forks(i) { V(f[i]); V(f[(i+1)%5]) }
```

Problems with the suggested solution

- Possible deadlock
  - What if all five philosophers become hungry simultaneously

- Alternative solutions
  - [solution 1] Use a counter:
    - at most n-1 philosophers may attempt to grab forks at the table
  - [solution 2] One philosopher requests forks in reverse order
    - Have one process pick up its right fork instead of left fork to break the circular wait
    - May violate concurrency requirements while one process is eating blocking all others in chain
Problems with the suggested solution

- Alternative solutions
  - [solution 3] Divide philosophers into two groups:
    - Odd philosophers: pick up their left forks first
    - Even philosophers: pick up their right forks first
  - [solution 4] Requests resources at once
    - Allow them to pick up their forks only if both forks are available

- Does eliminating deadlock solve all the problems?
  - What about starvation? A philosopher may starve to death

---

Dining Philosophers (1)

```c
#define N 5 /* number of philosophers */
#define LEFT (N-1) /* number of i's left neighbor */
#define RIGHT (N-1) /* number of i's right neighbor */
#define THINKING 0 /* philosopher is thinking */
#define HUNGRY 1 /* philosopher is thinking to get fork */
#define EATING 2 /* philosopher is eating */

typedef int semaphore; /* semaphores are a special kind of int */
const int mutex = 1; /* mutual exclusion for critical regions */

void philosophers(int i) /* i philosopher number, from 0 to N-1 */
{
  while(TRUE) /* repeat forever */
  {
    think(); /* philosopher is thinking */
    take_forks(); /* acquire two forks or block */
    eat(); /* yum-yum, spaghetti */
    put_forks(); /* put both forks back on table */
  }
}
```

Dining Philosophers (2)

```c
void take_fork(int i) /* i philosopher number, from 0 to N-1 */
{
  down(mutex); /* enter critical region */
  state[i] = HUNGRY; /* record fact that philosopher i is hungry */
  test(); /* try to acquire 2 forks */
  up(mutex); /* exit critical region */
  down(mutex); /* block if forks were not acquired */
}

void put_forks() /* i philosopher number, from 0 to N-1 */
{
  down(mutex); /* enter critical region */
  state[i] = THINKING; /* philosopher has finished eating */
  test(LEFT); /* see if left neighbor can now eat */
  test(RIGHT); /* see if right neighbor can now eat */
  up(mutex); /* exit critical region */
}

void test() /* i philosopher number, from 0 to N-1 */
{
  if (state[i] == HUNGRY && state[LEFT] == EATING && state[RIGHT] == EATING) /* i */
    state[i] = EATING;
}
```
(solution 4) Allow them to pick up their forks only if both forks are available

```c
monitor dp{
    state[5] = {thinking, thinking, thinking, thinking, thinking};
    condition self[5];

    pickup(int i) {
        state[i] = hungry;
        test(i);
        if (state[i] == eating)
            state[i].wait();
    }

    putdown(int i) {
        state[i] = thinking;
        test((i+4)%5);
    }

    test(int i) {
        if ((state[(i+4)%5] != eating) &&
            (state[i] == hungry) &&
            (state[(i+1)%5] != eating)) {
            state[i] = eating;
            self[i].signal();
        }
    }
}

dp.pickup(i);
...
... dp.putdown(i);
... eat
...