ECS518U - Operating Systems
Week 11

Java Concurrency Utilities

Java Concurrency Utilities

Library to avoid using wait / notify (or threads) directly

java.util.concurrent

- Utility classes commonly useful in concurrent programming
- Executors
  - Standardized interface
  - Thread-like subsystems, including thread pools
- Queues
  - Thread-safe FIFO queue
- Synchronizers
  - Other synchronization idioms – Semaphore
- Concurrent Collections

http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html

Semaphores

- Counting semaphore

`Semaphore(int permits)`

Creates a Semaphore with the given number of permits and nonfair fairness setting.

`void acquire()`

Acquires a permit from this semaphore, blocking until one is available, or the thread is interrupted.

`void release()`

Releases a permit, returning it to the semaphore.

Queues

```
public class ArrayBlockingQueue<E>
   implements BlockingQueue<E>
```

- **`void put(E e)`**
  - Inserts element 'e' at the end of the queue, waiting for space to become available if the queue is full
- **`E take()`**
  - Retrieves and removes the head of the queue, waiting if necessary until an element is available

Producer-Consumer using a Queue

```
class Producer<E> extends Thread {
  // the queue
  private BlockingQueue<E> queue ;
  // constructor
  Producer(BlockingQueue<E> q) {
     this.queue = q ;
  }
  // run method
  public void run() {
    while (true) {
      E item = produce() ; // producing
      queue.put(item) ;
    }
  }
}
```

Producer-Consumer using a Queue

```
class Consumer<E> extends Thread {
  // the queue
  private BlockingQueue<E> queue ;
  // constructor
  Consumer(BlockingQueue<E> q) {
     this.queue = q ;
  }
  // run method
  public void run() {
    while (true) {
      E item = queue.get() ;
      // consume
    }
  }
}
```

Producer-Consumer using a Queue

```
class Main {

  public static void main(String[] args) {
    // create the queue
    BlockingQueue<E> queue =
                              new ArrayBlockingQueue(10) ;
    // create the constructor and producer
    Consumer c = new Consumer(queue) ;
    Producer p = new Producer(queue) ;

    // start the constructor and producer
    c.start() ; p.start() ;
}
```

Executor – Interface

- **Executor interface**

`void execute(Runnable command)`

Executes the given command at some time in the future. The command may execute in a new thread, in a pooled thread, or in the calling thread, at the discretion of the Executor implementation.

- The interface 'ExecutorService' adds more methods
  - shutdown
  - awaitTermination

Executor – Thread Pool

- Executors class
  - Factory methods for 'Executor' classes

```
static ExecutorService
    newFixedThreadPool (int nThreads)
```

Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue.

Return a Result

Interface ExecutorService submit method

<T> Future<T> submit(Callable<T> task)

Submits a value-returning task for execution and returns a Future representing the pending results of the task. The Future's get method will return the task's result upon successful completion.

Parameters: task - the task to submit

Returns: a Future representing pending completion of the task

Interface Future<V> get method

V get()

Waits for the computation to complete, and then retrieves its result.

Returns: the computed result

Concurrent Collections

- Collections e.g. HashMap can be synchronized
  - Thread safe
  - BUT limited concurrency
- ConcurrentHashMap

A hash table supporting full concurrency of retrievals and … (some) … updates. This class obeys the same functional specification as Hashtable. However, even though all operations are thread-safe, retrieval operations do not entail locking … . This class is fully interoperable with Hashtable in programs that rely on its thread safety but not on its synchronization details.