[module lab 2.4]

BASIC LIBRARY SUPPORT
FOR THREAD COORDINATION
CONCURRENT BUILDING BLOCKS

• The Java platform libraries (Java 5.0 & Java 6.0) include a rich set of concurrent building blocks such as thread-safe collections and a variety of synchronizers that can coordinate the control flow of cooperating threads
  – Synchronized Collections
  – Concurrent Collections
  – Synchronizers
SYNCHRONIZED COLLECTIONS

- *Synchronized wrappers*
  - created by `Collections.synchronizedXXX` factory methods
  - achieving thread-safety by
    - encapsulating the state
    - synchronizing every public method
  > achieving safety by serializing all access to the collection’s state

- **Problems**
  - need to use additional client-side locking to guard compound actions
    - common compound actions include iteration, navigation, conditional operations such as put-if-absent
  - the object to be used for client-side locking is the synchronized collection object itself
  - performance problems
    - locking the collection for long-term operations, such as iteration...
    - strongly limiting concurrency
CONCURRENT COLLECTIONS

- Introduced with Java 5.0 and designed for concurrent access from multiple threads
  - greatly improving scalability and performance with respect to synchronized collections
- Main classes
  - `ConcurrentHashMap`
    - replacement for synchronized hash-based `Map` implementations
  - `CopyOnWriteArrayList`
    - a replacement for synchronized `List` implementations
  - `Queue` and `BlockingQueue`
    - interfaces with a different kinds of implementations available
BLOCKING QUEUE

- Provides blocking `put` / `take` methods + timed equivalent `offer` / `poll`
  - if the queue is full, `put` blocks until space becomes available
  - if the queue is empty, `take` blocks until an element is available
- Queue can be **bounded** and unbounded
  - unbounded queue are never full
- Bounded queue as a basic building block for *producer-consumer design pattern*
  - powerful resource management tool for building reliable applications
    - making programs more robust to overload by throttling activities that threaten to produce more work than can be handled
- Different classes implementing `BlockingQueue`
  - `LinkedBlockingQueue`, `ArrayBlockingQueue`, `PriorityBlockingQueue`,...
EXAMPLE: DESKTOP SEARCH

- A concurrent program scanning local drives for documents and indexes them for later searching
  - similar to Google Desktop or the Window Indexing Service
- Two agents + work queue
  - *File Crawler*
    - producer searching a file hierarchy for files meeting an indexing criterion and putting their names on the work queue
  - *Indexer*
    - consumer taking the file names from the queue and indexes them
- Benefits of the concurrent architecture (vs. sequential)
  - decomposing the overall problem in simple problems
    - increasing readability and reusability of the solution
  - several performance benefits
    - producers and consumers can execute concurrently (possibly in parallel)
    - good also in the case of mono-processor architecture, if the processes are I/O bound + CPU bound
public class FileCrawler extends Thread {
    private final BlockingQueue<File> fileQueue;
    private final FileFilter fileFilter;
    private final File root;

    public FileCrawler(BlockingQueue<File> q, FileFilter f, File r){
        fileQueue = q;
        fileFilter = f;
        root = r;
    }

    public void run() {
        try {
            crawl(root);
        } catch (InterruptedException ex) {
            Thread.currentThread().interrupt();
        }
    }

    private void crawl(File root) throws InterruptedException {
        File[] entries = root.listFiles(fileFilter);
        if (entries != null) {
            for (File entry : entries) {
                if (entry.isDirectory()) {
                    crawl(entry);
                } else if (!alreadyIndexed(entry)) {
                    fileQueue.put(entry);
                }
            }
        }
    }

    ...
public class Indexer extends Thread {
    private final BlockingQueue<File> fileQueue;

    public Indexer(BlockingQueue<File> q){
        fileQueue = q;
    }

    public void run(){
        try {
            while (true) {
                indexFile(queue.take);
            }
        } catch (InterruptedException ex){
            Thread.currentThread().interrupt();
        }
    }
}

BlockingQueue<File> queue = new LinkedBlockingQueue<File>(BOUND);
FileFilter filter = new FileFilter(){
    public boolean accept(File file){ return true; }
}

for (File root: roots){
    new FileCrawler(queue,filter,root).start();
}

for (File root: N_CONSUMERS){
    new Indexer(queue).start();
}
...
DEQUES AND WORK STEALING

• **Deque** and **BlockingDeque** data structure
  – introduced with Java 6.0
  – double-ended queue that allows for efficient insertion and removal from both the head and the tail
  – implementations: ArrayDeque and LinkedBlockingDeque

• Used for *work stealing* design pattern
  – similar to producers-consumers
  – each consumer has its own deque
  – if a consumer exhausts the work in its own deque, it can steal work from the *tail* of someone else’s deque

• More scalable than producers-consumers
  – workers don’t contend for a shared work queue
    • most of the time they access only their own deque, reducing contention
  – when accessing to others’ deque, the access is from the tail, not from the head
    • further reducing contention
SYNCHRONIZERS

• A synchronizer is any object that coordinates the control flow of threads based on its state
  – blocking queue can function as synchronizers
• Very important building blocks of concurrent applications
  – passive component encapsulating coordination functionalities
• All synchronizers share certain structural properties
  – encapsulating state that determines whether threads arriving at the synchronizers should be allowed to pass or forced to wait
  – providing methods to manipulate that state
  – providing methods to wait efficiently for the synchronizer to enter in the desired state
• Main types provided with Java library
  – Locks
  – Semaphores
  – Latches
  – Barriers
LOCKS

• Providing explicit lock functionality
  – vs. intrinsic lock given by synchronized blocks

• Lock interface and ReentrantLock implementation

```java
public interface Lock {
    void lock();
    void lockInterruptibly() throws InterruptedException;
    boolean tryLock();
    boolean tryLock(long timeout, TimeUnit unit) throws InterruptedException;
    void unlock();
    Condition newCondition();
}
```

• Typical usage:

```java
Lock lock = new ReentrantLock();
...
lock.lock();
try {
    // update shared object state
    // catch exception and restore invariants if necessary
} finally {
    lock.unlock();
}
```
POLLED AND TIMED LOCK ACQUISITION

• Using `tryLock` for polled and timed lock acquisition to have more sophisticated error recovery

```java
public boolean transferMoney(Account from, Account to, Amount am) throws InsufficientFundException, InterruptedException {
    while (true) {
        if (from.lock.tryLock()) {
            try {
                if (to.lock.tryLock()) {
                    try {
                        if (from.getBalance().compareTo(am) < 0) {
                            throw new InsufficientFundException();
                        } else {
                            from.debit(am);
                            to.credit(am);
                            return true;
                        }
                    } finally {
                        to.lock.unlock();
                    }
                }
            } finally {
                from.lock.unlock();
            }
        }
    }
}
```
EXPLICIT VS. INTRINSIC LOCKS

• Intrinsic locking (synchronized blocks) works fine in most situations but has some functional limitations
  – it is not possible to interrupt a thread waiting to acquire a lock..
  – ..or to attempt to acquire a lock without being willing to wait it forever
• In this case explicit locks can be used...
  – managing interruption
  – specifying bounded wait time
• ..with a strong discipline that must be followed by the programmers
  – explicit unlocking locks, for every possible scenario
• Performance comparison
  – in Java 5.0 explicit locks outperform intrinsic locks
    • ReentrantLock throughput about 4 times than intrinsic lock
  – in Java 6.0 same performance
SEMAPHORES

• Implementation of Dijkstra’s basic semaphore construct
• **Semaphore** class
  – created specifying a number of virtual *permits*
  – **acquire** + **release** method
  – possibility to enforce *fairness*
LATCHES

• A *latch* is a synchronizer that can delay the progress of a thread until it reaches its *terminal* state

• Function as a *gate*
  – until the latch reaches the terminal state, the gate is closed and no thread can pass
  – in the terminal state the gate opens allowing all threads to pass
  – once the latch reaches the terminal state, it cannot change the state again and so it remains open forever

• **CountDownLatch** class
  – CountDownLatch(int count)
    • to initialize the latch with a specific count
  – **countDown**
    • method to decrement the count
  – **await**
    • method that causes the current thread to wait until the latch has counted down to zero, unless the thread is interrupted.
LATCHES USE

• Used to ensure that certain activities do not proceed until other one-time activities complete.

• Main examples:
  – ensuring that a computation does not proceed until resources it needs have been initialized
    • using a binary latch for each resource
  – ensuring that a service does not start until other services on which it depends have started
    • using a binary latch for each service
    • starting service S would involve first waiting on latches for other services on which S depends, and then releasing the S latch after startup completes
  – waiting all parties involved in an activity (e.g: players in a multi-player game) are ready to proceed
    • the latch reaches its terminal state after all the players are ready
class Driver { // ...
    void main() throws InterruptedException {
        CountDownLatch startSignal = new CountDownLatch(1);
        CountDownLatch doneSignal = new CountDownLatch(N);
        for (int i = 0; i < N; ++i) // create and start threads
            new Thread(new Worker(startSignal, doneSignal)).start();
        doSomethingElse();            // don't let run yet
        startSignal.countDown();      // let all threads proceed
        doSomethingElse();
        doneSignal.await();           // wait for all to finish
    }
}

class Worker implements Runnable {
    private final CountDownLatch startSignal;
    private final CountDownLatch doneSignal;
    Worker(CountDownLatch startSignal, CountDownLatch doneSignal) {
        this.startSignal = startSignal;
        this.doneSignal = doneSignal;
    }
    public void run() {
        try {
            startSignal.await();
            doWork();
            doneSignal.countDown();
        } catch (InterruptedException ex) {} // return;
    }
    void doWork() { ... }
}
ANOTHER EXAMPLE

class Driver2 { // ...
    void main() throws InterruptedException {
        CountDownLatch doneSignal = new CountDownLatch(N);
        Executor e = ...

        for (int i = 0; i < N; ++i) // create and start threads
            e.execute(new WorkerRunnable(doneSignal, i));

        doneSignal.await();       // wait for all to finish
    }
}

class WorkerRunnable implements Runnable {
    private final CountDownLatch doneSignal;
    private final int i;

    WorkerRunnable(CountDownLatch doneSignal, int i) {
        this.doneSignal = doneSignal;
        this.i = i;
    }

    public void run() {
        try {
            doWork(i);
            doneSignal.countDown();
        } catch (InterruptedException ex) {} // return;
    }

    void doWork() { ... }
}
BARRIERS

• Implementation of the *barrier synchronization*
  – similar to latches in that they block a group of threads until some event has occurred
  – the key difference is that in this case all the threads must come together at a barrier point *at the same time* in order to proceed

> Latches are for waiting for *events*, barriers for other *threads*
BARRIER IMPLEMENTATION

- **CyclicBarrier** class
  - allows a fixed number of parties to *rendezvous* repeatedly at a *barrier point*
  - `CyclicBarrier(int parties)`
    - creates a new CyclicBarrier that will trip when the given number of parties (threads) are waiting upon it, and does not perform a predefined action upon each barrier.
  - `CyclicBarrier(int parties, Runnable barrierAction)`
    - ...executing an action when the barrier is passed
  - `int await()`
    - waits until all parties have invoked `await` on this barrier.
    - the barrier is reset as soon as all threads met at the barrier point
  - `boolean isBroken()`
    - queries if this barrier is in a broken state, i.e. a thread blocked in `await` was interrupted
  - ...
class Solver {
    final int N;
    final float[][] data;
    final CyclicBarrier barrier;

    class Worker implements Runnable {
        int myRow;
        Worker(int row) { myRow = row; }
        public void run() {
            while (!done()) {
                processRow(myRow);
                try {
                    barrier.await();
                } catch (InterruptedException ex) {
                    return;
                } catch (BrokenBarrierException ex) {
                    return;
                }
            }
        }
    }

    public Solver(float[][] matrix) {
        data = matrix;
        N = matrix.length;
        barrier = new CyclicBarrier(N,
                new Runnable() {
            public void run() {
                mergeRows(...);
            }
        });
        for (int i = 0; i < N; ++i)
            new Thread(new Worker(i)).start();
        waitUntilDone();
    }
}