**2025.1 Multicore Computing Project #2**

**-JAVA Concurrency Utilities-**

20223961 김수아

1. **BlockingQueue and ArrayBlockingQueue**
   1. Explanation
      1. BlockingQueue

BlockingQueue is an interface for implementing Thread-Safe queue. It let the thread wait when the thread try to add items when the queue is full or remove items when the queue is empty. It makes data exchange between different threads safe. Add/put/offer null value is not allowed. (resulting NullPointerException)

텍스트, 스크린샷, 도표, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

1 BlockingQueue illustration

* + 1. ArrayBlockingQueue

ArrayBlockingQueue is one of the implementations of BlockingQueue interface. There are additional implementations like LinkedBlockingQueue, LinkedBlockingDeque, LinkedTransferQueue, PriorityBlockingQueue, SynchronousQueue, DelayQueue.

ArrayBlockingQueue is a fixed-size array-based BlockingQueue, which

means its maximum size must be defined. Also It operates on FIFO (First In First Out).

* + 1. BlockingQueue Methods
       1. overview

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Throws Exception | Special Value | Blocks | TimesOut |
| Insert | add(object) | offer(o) | put(o) | offer(o, timeout, time unit) |
| Remove | remove(o) | poll(o) | take() | poll(timeout, time unit) |
| Examine | element() | peek() |  |  |

Throw Exception : Methods that throw an Exception if they cannot be executed immediately.

Special Value: Methods that return null or a Boolean value.

Blocks : Methods that put the current thread into a waiting state (blocked) if they cannot be executed immediately

Times Out : Methods that block for a specified amount of time

* + - 1. put()

When a thread calls put() during the queue is full, the thread waits until one or more element is removed from the queue.

* + - 1. take()

When a thread calls take() during the queue is empty, the thread waits until an element is added to the queue.

* 1. Implementation Example
     1. Code Explanation (using put(), take() methods)

It is a code that implements a simple producer-consumer relationship. The producer thread insert one message by calling put() method. The Consummer thread delete one message by calling take() method. It is impossible to remove something from empty queue or put something to full queue. To put the consumer/producer thread into waiting state, ArrayBlockingQueue is used.

As the queue size is fixed, the maximum size is set to 3 at main() function. Also there is put() or take() method. The key code sections are highlighted.

* + 1. Complete Code

package prob3;  
  
import java.util.concurrent.ArrayBlockingQueue;  
import java.util.concurrent.BlockingQueue;  
  
public class ex1 {  
 public static void main(String[] args) {  
 BlockingQueue<String> queue = new ArrayBlockingQueue<>(3);  
  
 Thread producer = new Thread(new Producer(queue));  
 Thread consumer = new Thread(new Consumer(queue));  
  
 producer.start();  
 consumer.start();  
 }  
}  
  
class Producer implements Runnable {  
 private BlockingQueue<String> queue;  
  
 public Producer(BlockingQueue<String> queue) {  
 this.queue = queue;  
 }  
  
 @Override  
 public void run() {  
 for (int i = 1; i <= 10; i++) {  
 String message = i + "th message";  
 System.*out*.println("Producer " + i);  
 try {  
 System.*out*.println("Trying to put: " + message);  
 queue.put(message);  
 System.*out*.println("Put " + message + " Done. Queue Size:" + queue.size());  
  
 Thread.*sleep*((int)(Math.*random*() \* 2000));  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}  
  
class Consumer implements Runnable {  
 private BlockingQueue<String> queue;  
  
 public Consumer(BlockingQueue<String> queue) {  
 this.queue = queue;  
 }  
  
 @Override  
 public void run() {  
 String message;  
  
 for (int i = 1; i <= 10; i++) {  
 System.*out*.println("Consumer " + i);  
 try {  
 System.*out*.println("Trying to take()");  
 message = queue.take();  
 System.*out*.println("take: " + message + " Done. Queue Size:" + queue.size());  
 Thread.*sleep*((int)(Math.*random*() \* 2000));  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

* 1. Execution Result
     1. Result Explanation

The output shows when the Consumer/Producer attempts or succeeds in performing an operation, and when successful, the current Queue size is displayed. In the highlighted section, when the queue size was 0, the consumer tried to perform a take() operatoin but failed and entered a waiting state. After messages were inserted into the queue by using put() method, the waiting state was released and take() success.

* + 1. Complete Result

Consumer 1

Trying to take()

Producer 1

Trying to put: 1th message

Put 1th message Done. Queue Size:1

take: 1th message Done. Queue Size:0

Consumer 2

Trying to take()

Producer 2

Trying to put: 2th message

Put 2th message Done. Queue Size:1

take: 2th message Done. Queue Size:0

Producer 3

Trying to put: 3th message

Put 3th message Done. Queue Size:1

Consumer 3

Trying to take()

take: 3th message Done. Queue Size:0

Consumer 4

Trying to take()

Producer 4

Trying to put: 4th message

Put 4th message Done. Queue Size:1

take: 4th message Done. Queue Size:0

Consumer 5

Trying to take()

Producer 5

Trying to put: 5th message

take: 5th message Done. Queue Size:0

Put 5th message Done. Queue Size:0

Producer 6

Trying to put: 6th message

Put 6th message Done. Queue Size:1

Producer 7

Trying to put: 7th message

Put 7th message Done. Queue Size:2

Consumer 6

Trying to take()

take: 6th message Done. Queue Size:1

Consumer 7

Trying to take()

take: 7th message Done. Queue Size:0

Producer 8

Trying to put: 8th message

Put 8th message Done. Queue Size:1

Consumer 8

Trying to take()

take: 8th message Done. Queue Size:0

Producer 9

Trying to put: 9th message

Put 9th message Done. Queue Size:1

Consumer 9

Trying to take()

take: 9th message Done. Queue Size:0

Producer 10

Trying to put: 10th message

Put 10th message Done. Queue Size:1

Consumer 10

Trying to take()

take: 10th message Done. Queue Size:0

1. **ReadWriteLock**
   1. Explanation
      1. overview

ReadWriteLock provides parallelism for reading operations and data consistency for writing operations. It is an interface for concurrent control in multithreaded environments in Java. It provides different types of locks for reading and writing operations. Multiple threads can perform reading operations simultaneously, but writing operations can be performed exclusively by getting a whiteLock.

The main implementation is ReentrantReadWriteLock. This class provides reentrant functionality and allows setting fairness policies for read and write locks.

* + 1. Lock()

Acquires the lock. When one thread holds the lock, other threads are blocked until they can acquire the lock..

* + 1. Unlock()

Releases the lock held by the thread.

* + 1. readLock()

Returns a lock object for reading operations. Multiple threads can acquire it simultaneously.

* + 1. writeLock()

Returns a lock object for writing operations. Only one thread can acquire it at a time.

* 1. Implementation Example
     1. Code Explanation

The code perform read/write operations on SharedData safely. It creates 5 reader threads, 3 writer threads and they run simultaneously. Reader thread reads shared data by using readLock(). Writer thread use writeLock() to modify shared data. During modifying time, the others can’t access the date.

Each thread runs the task after getting a lock. And release the lock in the ‘finally’ block. This ensures that the lock is released even if exception occurs.

* + 1. Complete Code

package prob3;  
  
import java.util.concurrent.locks.ReadWriteLock;  
import java.util.concurrent.locks.ReentrantReadWriteLock;  
  
public class ex2 {  
 public static void main(String[] args) {  
 SharedData data = new SharedData();  
  
 for (int i = 0; i < 5; i++) {  
 Thread reader = new Thread(new Reader(data, i));  
 reader.start();  
 }  
  
 for (int i = 0; i < 3; i++) {  
 Thread writer = new Thread(new Writer(data, i));  
 writer.start();  
 }  
 }  
}  
  
class SharedData {  
 private int data = 0;  
 private ReadWriteLock lock = new ReentrantReadWriteLock();  
  
 public int readData() { //multiple threads can lock the lock for reading  
 lock.readLock().lock();  
 try {  
 System.*out*.println(Thread.*currentThread*().getName() + " start reading : " + data);  
 Thread.*sleep*(1000);  
 System.*out*.println(Thread.*currentThread*().getName() + " finish reading : " + data);  
 return data;  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 return -1;  
 } finally {  
 lock.readLock().unlock();  
 }  
 }  
  
 public void writeData(int newValue) { //only one thread at a time can lock the lock for writing.  
 lock.writeLock().lock();  
 try {  
 System.*out*.println(Thread.*currentThread*().getName() + " start writing ");  
 Thread.*sleep*(2000);  
 this.data = newValue;  
 System.*out*.println(Thread.*currentThread*().getName() + " finish writing : " + data);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } finally {  
 lock.writeLock().unlock();  
 }  
 }  
}  
  
class Reader implements Runnable {  
 private SharedData data;  
 private int id;  
  
 public Reader(SharedData data, int id) {  
 this.data = data;  
 this.id = id;  
 }  
  
 @Override  
 public void run() {  
 for (int i = 0; i < 3; i++) {  
 int value = data.readData();  
 System.*out*.println("Reader " + id + " value : " + value);  
 try {  
 Thread.*sleep*((int)(Math.*random*() \* 1000));  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}  
  
class Writer implements Runnable {  
 private SharedData data;  
 private int id;  
  
 public Writer(SharedData data, int id) {  
 this.data = data;  
 this.id = id;  
 }  
  
 @Override  
 public void run() {  
 for (int i = 0; i < 2; i++) {  
 int newValue = (int)(Math.*random*() \* 100);  
 data.writeData(newValue);  
 System.*out*.println("Writer " + id + " edited value : " + newValue);  
 try {  
 Thread.*sleep*((int)(Math.*random*() \* 2000));  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

* 1. Execution Result
     1. Result Explanation
        1. Multiple reader threads can read simultaneously (first highlighted section)
        2. When a writer thread acquires the writeLock, all other threads (both readers and writers) become waiting state. (second highlighted section)
        3. Reader threads run in parallel.
     2. Complete Result

Thread-3 start reading : 0

Thread-2 start reading : 0

Thread-4 start reading : 0

Thread-1 start reading : 0

Thread-0 start reading : 0

Thread-1 finish reading : 0

Thread-0 finish reading : 0

Thread-4 finish reading : 0

Thread-2 finish reading : 0

Thread-3 finish reading : 0

Thread-5 start writing

Reader 2 value : 0

Reader 3 value : 0

Reader 0 value : 0

Reader 4 value : 0

Reader 1 value : 0

Thread-5 finish writing : 46

Thread-7 start writing

Writer 0 edited value : 46

Thread-7 finish writing : 36

Thread-6 start writing

Writer 2 edited value : 36

Thread-6 finish writing : 14

Writer 1 edited value : 14

Thread-4 start reading : 14

Thread-3 start reading : 14

Thread-1 start reading : 14

Thread-0 start reading : 14

Thread-2 start reading : 14

Thread-3 finish reading : 14

Reader 3 value : 14

Thread-1 finish reading : 14

Reader 1 value : 14

Thread-4 finish reading : 14

Reader 4 value : 14

Thread-2 finish reading : 14

Thread-0 finish reading : 14

Reader 0 value : 14

Reader 2 value : 14

Thread-5 start writing

Thread-5 finish writing : 28

Writer 0 edited value : 28

Thread-7 start writing

Thread-7 finish writing : 91

Writer 2 edited value : 91

Thread-6 start writing

Thread-6 finish writing : 15

Writer 1 edited value : 15

Thread-4 start reading : 15

Thread-0 start reading : 15

Thread-2 start reading : 15

Thread-3 start reading : 15

Thread-1 start reading : 15

Thread-0 finish reading : 15

Reader 0 value : 15

Thread-2 finish reading : 15

Reader 2 value : 15

Thread-4 finish reading : 15

Reader 4 value : 15

Thread-3 finish reading : 15

Thread-1 finish reading : 15

Reader 1 value : 15

Reader 3 value : 15

1. **AtomicInteger**
   1. Explanation
      1. overview

AtomicInteger is a class that included in java.util.concurrent.atomic package. It is an integer variable that can do atomic operations. In a multithread environment, it supplies thread-safe integer operations.

For subtracting from the AtomicInteger value, decrementAndGet(), getAndDecrement() methods are available.

* + 1. set(int i)

Set AtomicInteger value to i. It works as atomic even several threads access simultaneously.

* + 1. get()

It returns current value of AtomicInteger.

* + 1. getAndAdd(int i)

Add I to current value and return before-update value.

Atomically processes the step : read value -> update value -> return previous value

* + 1. addAndGet(int i)

Add I to current value and return updated value.

Atomically processes the step : read value -> update value -> return new value.

* 1. Implementation Example
     1. Code Explanation

Two threads try to update AtomicInteger value simultaneously. Each thread edits shared variable by using set(), getAndAdd(), addAndGet().

Two threads execute different task to one AtomicInteger object.

* + 1. Complete Code

package prob3;  
  
import java.util.concurrent.atomic.AtomicInteger;  
  
public class ex3 {  
 public static void main(String[] args) throws InterruptedException {  
 Counter counter = new Counter();  
  
 Thread t1 = new Thread(counter, "t1");  
 Thread t2 = new Thread(counter, "t2");  
  
 t1.start();  
 t2.start();  
  
 t1.join();  
 t2.join();  
  
 // Display the final counter value  
 System.*out*.println("Final : " + counter.getValue());  
 }  
}  
  
class Counter implements Runnable{  
 private AtomicInteger value = new AtomicInteger(0);  
  
 @Override  
 public void run() {  
 String threadName = Thread.*currentThread*().getName();  
  
 if (threadName.equals("t1")) {  
 setValue(10);  
 getAndIncrement(5);  
 incrementAndGet(3);  
 } else {  
 setValue(20);  
 getAndIncrement(7);  
 incrementAndGet(2);  
 }  
  
 System.*out*.println(threadName + " finished. Current value: " + getValue());  
 }  
  
 private int incrementAndGet(int i) {  
 int newVal = value.addAndGet(i);  
 System.*out*.println(Thread.*currentThread*().getName() + " addAndGet('" + i + "') return : " + newVal + ", current val : " + value.get());  
 return newVal;  
 }  
  
 private int getAndIncrement(int i) {  
 int prevValue = value.getAndAdd(i);  
 System.*out*.println(Thread.*currentThread*().getName() + " getAndAdd('" + i + "') return : " + prevValue + ", current val : " + value.get());  
 return prevValue;  
  
  
 }  
  
 public int getValue() {  
 return value.get();  
 }  
  
 public void setValue(int newValue){  
 value.set(newValue);  
 System.*out*.println(Thread.*currentThread*().getName() + " set to " + newValue);  
 }  
  
  
}

* 1. Execution Result
     1. Result Explanation

Two Threads execute at same time but the operations manage atomically. So data concurrency is maintained.

* + - 1. thread t2 overwrites value to 20 after thread t1 set the value to 10.
      2. t2 update value to 27 and return previous value(20).
      3. t1 update value to 32 and return previous value(27).
      4. t2 update value to 34 and return update value(34).
      5. t1 update value to 37 and return update value(37)
    1. Complete Result

t1 set to 10

t2 set to 20

t2 getAndAdd('7') return : 20, current val : 27

t1 getAndAdd('5') return : 27, current val : 32

t2 addAndGet('2') return : 34, current val : 34

t2 finished. Current value: 37

t1 addAndGet('3') return : 37, current val : 37

t1 finished. Current value: 37

Final : 37

1. **CylclicBarrier**
   1. Explanation
      1. Overview

CyclicBarrier is a barrier class for synchronization. Barrier is a certain point that multiple threads wait for each other. CyclicBarrier let the threads wait until certain number of threads reach at barrier. After last thread reach, the barrier is removed and every waiting threads execute simultaneously. In addition, cyclicBarrier can define action(barrier action) selectively. Barrier Action is run when every threads reach at barrier.

텍스트, 스크린샷, 라인, 도표이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* + 1. Await()

This method let the threads wait at the barrier. InterruptedException occurs when thread is interrupted while waiting, BrokenBarrierException occurrs when barrier is broken.

* 1. Implementation Example
     1. Code Explanation

Three threads run through two barriers.

As barrier Action, a message is printed.

Each thread turns to a waiting thread when it reaches to barrier by calling await() method.

* + 1. Complete Code

package prob3;  
  
import java.util.concurrent.BrokenBarrierException;  
import java.util.concurrent.CyclicBarrier;  
  
public class ex4 {  
 public static void main(String[] args) {  
 int num = 3;  
 CyclicBarrier barrier = new CyclicBarrier(num, () ->  
 System.*out*.println("Done : threads start moving to next phase"));  
 /\*  
 Runnable barrierAction = new Runnable() {  
 public void run() {  
 System.out.println("Done : threads start moving to next phase");  
 }  
 };  
 CyclicBarrier barrier = new CyclicBarrier(2, barrierAction);  
  
 \*/  
  
 for (int i = 0; i < num; i++) {  
 Thread thread = new Thread(new thread(i+1, barrier));  
 thread.start();  
 }  
 }  
}  
  
class thread implements Runnable {  
 private int id;  
 private CyclicBarrier barrier;  
  
 public thread(int id, CyclicBarrier barrier) {  
 this.id = id;  
 this.barrier = barrier;  
 }  
  
 @Override  
 public void run() {  
 try {  
 System.*out*.println("thread " + id + " starts first cycle");  
 Thread.*sleep*(500 \* id);  
 System.*out*.println("thread " + id + " finishes first cycle");  
  
 System.*out*.println("thread " + id + " waits for others at first barrier");  
 barrier.await();  
  
 System.*out*.println("thread " + id + " starts second cycle");  
 Thread.*sleep*(300 \* id);  
 System.*out*.println("thread " + id + " finishes second cycle");  
  
 System.*out*.println("thread " + id + " waits for others at second barrier");  
 barrier.await();  
  
 System.*out*.println("thread " + id + " completed all work");  
  
 } catch (InterruptedException | BrokenBarrierException e) {  
 e.printStackTrace();  
 }  
 }  
}

* 1. Execution Result
     1. Result Explanation

Three thread work for different amounts of time. When every thread reach to barrier, a message is printed as barrier action. After the message, every thread simultaneously start new task after the barrier

* + 1. Complete Result

thread 1 starts first cycle

thread 2 starts first cycle

thread 3 starts first cycle

thread 1 finishes first cycle

thread 1 waits for others at first barrier

thread 2 finishes first cycle

thread 2 waits for others at first barrier

thread 3 finishes first cycle

thread 3 waits for others at first barrier

Done : threads start moving to next phase

thread 1 starts second cycle

thread 2 starts second cycle

thread 3 starts second cycle

thread 1 finishes second cycle

thread 1 waits for others at second barrier

thread 2 finishes second cycle

thread 2 waits for others at second barrier

thread 3 finishes second cycle

thread 3 waits for others at second barrier

Done : threads start moving to next phase

thread 1 completed all work

thread 2 completed all work

thread 3 completed all work

1. **ExecutorService, Executors, Callable, Future**
   1. Explanation
      1. ExecutorService

ExecutorService is an interface for executing asynchronous task. It manage thread pool and submit task to thread pool to run it. ExecutorServices return the result of task as a Future object.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* + 1. Executors

Executors is factory class to create ExecutorService instances. It can simplify thread pool creation/management. There are methods like newSingleThreadExecutor(), newFixedThreadPool(), newScheduledThreadPool(). Each method creates unique thread pool.

* + 1. Callable

Callable is an interface that return value, define tasks that can throw exception. The difference between Runnable and Callable is that Callable can return value. It have only one method, call().

* + 1. Future

Future is an interface that represents the result of an asynchronous task. Future object is returned if you submit callable task to ExecutorService. Future is used for checking state of task and choose waiting for done or getting result or canceling task. It contains get(), isDone(), cancel() methods.

* + 1. Methods
       1. newFixedThreadPool()

It is static method of Executors class. It creates a static thread pool. It creates and maintains certain size of thread. If an additional task is submitted while every thread is working, the additional task is wait in queue.

* + - 1. submit()

It is ExecutorService interface’s method. It is for submit some tasles to thread pool. It get Runnable/Callable object as paramenter, run asynchronously. Result is returned to Future object to see the task result lator.

* + - 1. call()

It is one and only method of Callable interface. It performs task and return result. It is similar to run() in Runnable but it can return value and throw exception. If executorService execute Callable object, call() is called.

* 1. Implementation Example
     1. Code Explanation

It is a program that computes the number of prime numbers between 1~200,000 using thread pool.

newFixedThreadPool() creates static size thread pool that consists of 6 (NUM\_THREADS) threads.

Callable : It performs task (compute number of prime numbers within certain range) and return result.

Future : Submit() method submits every callable works to thread pool. Returned Future objects are saved as list.

Shutdown() method quit the program after Future.get() method compute final result(total number of prime numbers).

* + 1. Complete Code

package prob3;  
  
import java.util.ArrayList;  
import java.util.List;  
import java.util.concurrent.\*;  
  
public class ex5{  
 private static final int *NUM\_END* = 200000; // default input  
 private static final int *NUM\_THREADS* = 6;  
  
 public static void main(String[] args){  
 ExecutorService executor = Executors.*newFixedThreadPool*(*NUM\_THREADS*);  
 List<Future<Integer>> futures = new ArrayList<>();  
  
 for (int i = 0; i < *NUM\_THREADS*; i++){  
 int start = i\*(*NUM\_END*/*NUM\_THREADS*) + 1;  
 int end;  
 if (i == *NUM\_THREADS* - 1) {  
 end = *NUM\_END*;  
 } else {  
 end = (i + 1) \* (*NUM\_END* / *NUM\_THREADS*);  
 }  
  
 PrimeCounter counter = new PrimeCounter(start, end);  
 Future<Integer> future = executor.submit(counter);  
 futures.add(future);  
  
 }  
  
 int totalPrimes = 0;  
 try{  
 for (Future<Integer> future : futures){  
 totalPrimes += future.get();  
 }  
 } catch (InterruptedException | ExecutionException e){e.printStackTrace();}  
  
 executor.shutdown();  
 System.*out*.println("answer is " + totalPrimes);  
 }  
}  
  
class PrimeCounter implements Callable<Integer>{  
 private final int start;  
 private final int end;  
  
 public PrimeCounter(int start, int end) {  
 this.start = start;  
 this.end = end;  
 }  
 @Override  
 public Integer call() {  
 int count = 0;  
 for (int i = start; i <= end; i++) {  
 if (*isPrime*(i)) {  
 count++;  
 }  
 }  
 System.*out*.println("Range [" + start + ", " + end + "]: found " + count + " primes");  
 return count;  
 }  
 private static boolean isPrime(int x) {  
 int i;  
 if (x<=1) return false;  
 for (i=2;i<x;i++) {  
 if (x%i == 0) return false;  
 }  
 return true;  
 }  
}

* 1. Execution Result
     1. Result Explanation

NUM\_THREADS threads compute the number of prime numbers within their own range independently. After computing, result is returnes. After all thread finish their task, total result is printed.

* + 1. Complete Result

Range [1, 33333]: found 3569 primes

Range [33334, 66666]: found 3076 primes

Range [66667, 99999]: found 2947 primes

Range [100000, 133332]: found 2852 primes

Range [133333, 166665]: found 2781 primes

Range [166666, 200000]: found 2759 primes

answer is 17984