Multiprocessor programming

Michael Lundquist

April 17, 2019

1 Introduction

In modern computing, multi-core processors are almost ubiquitous. Even the 35$ Raspberry Pi 3 B+ comes equip with a quad-core processor. This new paradigm of computing comes with new challenges, but grand new possibilities. This paper covers some common possibilities, challenges and solutions to the challenges faced in this new environment. When applicable, this paper has tested java programs to demonstrate some of these concepts.

1.1 Possibilities

Having multi-core processors allows multiple processes to run *in parallel*. When multiple processes run in parallel, their programs *speed up*. However, in many situations, programs can't run in parallel for various reasons. Still, the multi-core architecture can speed up non-parallel tasks by diverting them to core that runs at a faster speed. Truly, multiple cores is here to stay.

1.2 Amdahl's Law

Amdahl's law[1] is a formula for calculating how much a program speeds up. We say a program with n processors speeds up S times. This formula also has the factor p, which is the ratio of the program that can be run in parallel. If $p = .98$, 98% of the program can be run in parallel.

$$S = \frac{1}{1 - p + \frac{p}{n}}$$

This law provides a quick and dirty approach to determining if it's worth it to run a program on a multi-cored system. Furthermore, it has some surprising results. For example, if 95% of a program can be parallelized, and you run it on a 1080Ti with its 57344 threads, you only see a speed up of about 20. While S

[1] Amdahl, 1967, the cited paper is the law's origin.

was only 20 in the last example, $\frac{p}{n}$ ran *nearly instantly*. With this information, our goal is to maximize p.[2]

2 Challenges

Speedup through parallelization is fantastic, but parallelization naturally comes with new challenges. For example, work coordination takes on a new form in multi-threaded applications where threads must be persistently independent and isolated, but still coordinate the use or resources. This section will discuss these challenges and discuss some requirements that programmers use to avoid these challenges. Finally this section will provide a sample solution to the dining philosophers problem[3] that demonstrates a situation where these issues could arise.

2.1 deadlock

Deadlocks occur when a thread is permanently blocked from executing (entering its critical section) by other threads. Without locks (see below), this would happen when multiple processes can't resolve competitions for resources. To prevent deadlock, programmers grant mutually exclusive access to resources. When a program has mutually exclusive access to a resource, it's said to be in its *critical section*.

A critical section is *well formed* if it has the following three properties. First, it has its own unique lock. Each process needs its own lock to prevent *covering states* where the thread is about to use a resource, but has no way of telling other threads about this intention. Second, the thread lock when entering their critical section to ensure mutual exclusion. Third, threads unlock when exiting

[2]djna, 2019, Stackoverflow user, djna, helps another user, Monster, maximize p.
[3]Dijkstra, 1965, produced by dijkstra.

their critical section to allow other threads to ensure mutual exclusion.

2.2 starvation

The last requirement of a well-formed critical section to unlock prevents *starvation*. Starvation is when a thread a process can't *ever* enter its critical section. Freedom from starvation is guaranteed if your program has no deadlocks and threads enter their respective critical sections in a first-come first-serve order. Note, freedom from starvation is important, but not always guaranteed.

2.3 fairness

If a thread is starvation free, it will eventually execute, but for a thread to have a truly *fair* (equal) chance of executing its critical section, it cannot wait an unfair amount of time. Fairness is often accomplished by putting *bounds* on how long your program can wait at any point. If bounded wait times are ensured on all parts a thread's life-cycle, we call the cycle *wait free*.

2.4 Dining Philosophers Example

The dining philosophers[4] problem is an example of how deadlock and therefore starvation could occur. In the problem a group of philosophers sit around a table and eat. The philosophers must share a chopstick with a philosopher on their right, and a philosopher on their right. If every philosopher is greedy and grabs one chopstick, deadlock will occur and the philosophers will starve and die.

My group of philosophers decided to courteous. When they sit down, they decide who starts eating. Then one at a time, the philosophers either: stop eating if they are, or tell their neighbors to stop, and take their chopstick. If a

[4]Dijkstra, 1965, original notes on concurrency from dijkstra.

philosopher was told to stop eating, he skips his turn. The emerging behavior from this algorithm is a deadlock-free, starvation-free and fair solution where each philosopher can eat.

```
import java.util.ArrayList;
import java.util.Arrays;

/**
 * Here's an implementation of the classic
 */
public class Philosopher{
    //some constants

    // tells the program how many philosophers
    public static final int NUMB_PHILOSOPHERS = 10;
    // tells the program how many clock ticks to take
    public static final int NUMB_STEPS = 5;

    //if this
    public boolean eating = false;

    //This philosopher's neighbors
    public Philosopher next_philosopher;
    public Philosopher last_philosopher;

    //The total list of philosophers. Mostly used to make printing
        easier
    public static ArrayList<Philosopher> philosophersList = new
        ArrayList<Philosopher>();
```

```
public static void main(String args[]){
    setTable();
    passForks();
}

/**
 * Sets the philosophers up in a circularly linked list
 *
*/
private static void setTable(){
    Philosopher head = new Philosopher( null);
    Philosopher prevPhilosopher = new Philosopher( head );
    head.next_philosopher = prevPhilosopher;
    prevPhilosopher.eating = true;

    for (int i = 0; i < NUMB_PHILOSOPHERS; i++){
        prevPhilosopher.next_philosopher = new Philosopher(
            prevPhilosopher );
        prevPhilosopher = prevPhilosopher.next_philosopher;
        if(i < NUMB_PHILOSOPHERS - 1){
            if(prevPhilosopher.last_philosopher.eating == false){
                prevPhilosopher.eating = true;
            }
        }
        philosophersList.add(prevPhilosopher);
    }
    prevPhilosopher.next_philosopher = head;
    head.last_philosopher = prevPhilosopher;
```

```
}

/**
 * algorithm. Basically each philosopher:
 *   if eating: Stops eating
 *   if not: tell the next to stop, and start. Skip the next.
 *
 * Prints at every step to show how it works
 */
private static void passForks(){
    for (int i = 0; i < NUMB_STEPS; i ++){
        //The clock is ticking, start eating
        if(head.eating){
            head.eating = false;
            head = head.next_philosopher;
        }else{
            head.next_philosopher.eating = false;
            head.eating = true;
            head = head.next_philosopher.next_philosopher;
        }
        //print the philosophers out
        System.out.println(
            Arrays.toString(
                Philosopher.philosophersList.toArray()
            )
        );
    }
}

Philosopher(Philosopher prev){
```

```
            this.last_philosopher = prev;
        }


        public String toString(){
            if(eating){
                return "T";
            }else{
                return "F";
            }
        }
    }
```

2.4.0.1 Dining Philosopher Output. The above dining philosophers program's first five steps are shown below. Each index represents a philosopher. If the value at the index is T, the philosopher eats, otherwise they wait.

```
[F, T, F, T, F, T, F, T, F, F]
[T, F, F, T, F, T, F, T, F, F]
[T, F, T, F, F, T, F, T, F, F]
[T, F, T, F, T, F, F, T, F, F]
[T, F, T, F, T, F, T, F, F, F]
```

3 Solutions to Challenges

The dining philosophers problem gives an example where the challenges discussed above are faced. These challenges are regularly faced in multi-threaded programs where programs are competing for resources as the philosophers competed for chopsticks. This section will discuss some general rules and techniques for overcoming the discussed challenges. The first technique we use is locking,

we then discuss protocols for avoiding the challenges.

3.1 Locking Algorithms

Well-formed critical sections start, lock, run, then unlock. Remember, critical sections are how mutual exclusion, a fundamental property of multi-threading is performed. Locks are used by a thread to request entry into its critical section. If another thread is in its critical section, the lock will block the thread until the lock has determined to unblock the thread. Although locks are a simple concept, they're difficult to design. Along with the issues discussed above lock algorithms are often called simultaneously by two different threads.

3.1.1 Filter Lock

The filter lock uses two parallel arrays to queue threads. When a thread calls lock, it enters into level 0 of each of the arrays. If a thread is already at position 0, one of the arrays has a variable to detect this and the thread that's currently at position 0 gets bumped up to the next position. As you can see, it's very important to have enough space for each thread. While the position ahead of the current thread is open, thread moves up to that position. Eventually, the thread hits the last position in the arrays and runs. The important thing to understand about the filter lock is deadlock-free, and first-in first-out, furthermore simultaneous calls to the filter lock are safe.

Peterson Lock. The filter lock is an adaption of the Peterson lock[5] to n variables. Peterson pioneered the idea of reading and writing a shared variable to prevent deadlock and flags to prevent starvation.

FilterLock.java.

[5]Peterson, 1981, Peterson's original paper on the Peterson lock.

```
package com.os.seive;

/**
    * This class implements the filter lock.
    * The filter lock is a generalization of the Peterson lock
    * to n threads.
    *
    * Note: inputs (JOptionPane) are blocking so they prevent
        threading
    *
    * @author Michael Lundquist
    */
class FilterLock { // implements Lock{ (lock's unlock is different)
    public volatile boolean[] flag;
    //flag is an array containing locks
    public volatile long[] victim;
    //victim is the member variable array
    public int highestLevel;

    /**
        * Creates the parallel arrays flag and victim.
        * As in the Peterson lock, the flags prevent starvation
        * and the victim prevents deadlock.
        *
        * @param n the number of levels in the lock (number of
            possible threads)
        */
    public FilterLock(int n){
        victim = new long[n];//last to enter
```

```
    /*
    you don't need a victim at the
    highest level because you've already filtered out
    all possible deadlocks
    */
    flag = new boolean[n + 1];
    this.highestLevel = n;
}

/**
    * Blocks the calling thread until this thread is first in the
        filter.
    * This results in a fair allocation of resources.
    */
public void lock(){
    //loop to block for each level of the thread.
    for (
        int level = 0;
        level < this.highestLevel;
        level ++
    ){
        victim[level] = Thread.currentThread().getId();
        /**
            * It's important that you set the flag after the
            * victim so you know who should wait to set the flag if
            * two processes hit this method simultaneously
            */
        flag[level] = true;//acquire lock

        unlock(level - 1);//unlock the last level.
```

```
        while (
            flag[level + 1] &&
            victim[level] == Thread.currentThread().getId()
        ){
        /**
            * Block this thread until the next level is empty
                (prevents stravation using flag)
            * or the thread is pre-empted to move to the next
                level (prevents deadlock using victim).
            */
        }
    }
    /**
        * Then handle the highest level
        * You don't use victim because it could cause starvation
            at the last level.
        * Plus it's unnecessary, the filter was already big
            enough to handle all possible deadlock.
        * You simply add a lock and wait to be run.
        */

    flag[this.highestLevel] = true;
    unlock(this.highestLevel - 1);

    //****Wait for The Runnable to run the thread, then unlock!
        *******/

}

/**
```

```
        * Unlocks at this level of the filter.
        *
        * @param level (the level of the filter to unlock)
        */
    private void unlock(int level){
        if(level >= 0){
            flag[level] = false;
        }
    }

    /**
        * Unlocks in the way a user would think,
        * by unlocking the last level of the
        */
    public void unlock(){
        //further ensures the other thread will run
        unlock(this.highestLevel);
    }
}
```

Lock interface.

```
package com.os.seive;

/**
 * This class provides an interface to locking for you
 * so you can implement different types of locks
 */
public interface Lock{
    public void lock();
```

```
    public void unlock();
}
```

3.1.2 Multi-threaded counter

This is an example of counting to twenty on a shared variable using four threads. The algorithm uses a filter lock to ensure mutually exclusive access to and modification of the variable. If mutually exclusive access were not ensured in this simple example, then multiple threads would see the same value. Although we're not using the value for anything at the moment, ensuring data integrity is such a common problem in multithreading they have a name for it, *the readers-writers problem*.

Counter.

```
package com.os.seive;

import java.util.Arrays;
import java.util.Vector;

import javax.swing.JOptionPane;

/**
 * This application counts from 0 to 20 using
 * 4 different thread. This algorithm is useful for demonstrating
 * how a lock works, and why locks are needed.
 *
 * @author Michael Lundquist
 */
public class Counter implements Runnable{
    //Constants. You can play around with these
    public static int MAX_COUNT = 20;
    public static final int NUMB_THREADS = 4;

    //Variables used in the program to account for which thread is
        doing what.
    public static volatile int curCount = 0;
    public static FilterLock locks = new
        FilterLock(Counter.NUMB_THREADS);
    public static Thread[] threads;
```

```
//ar stores which thread ID counted a number. Indexes are
    numbers counted,
//thread IDs are the values at the indexs.
static Vector<Long> ar = new Vector<>();

/**
 * Java's main method.
 * @param args
 */
public static void main(String[] args){
    threads = new Thread[NUMB_THREADS];
    Counter c;
    for(int i = 0; i < NUMB_THREADS; i++){
        c = new Counter();
        threads[i] = new Thread(c);
        threads[i].start();
    }

    while( Counter.curCount < Counter.MAX_COUNT ){
        //wait for the counting to finish
    }

    //join the threads back into the main program.
    for(int i = 0; i < NUMB_THREADS; i++){
        try{
            threads[i].join();
        }catch(InterruptedException e){
            e.printStackTrace();//???
        }
    }

    //prints out which thread did what
    JOptionPane.showMessageDialog(null,
        Arrays.toString(ar.toArray()));

}


/**
 * Run is the code that's threaded. It runs in a thread until it
    returns.
 *
 * @see Thread
 */
@Override
public void run() {
    while(Counter.curCount < Counter.MAX_COUNT){
        locks.lock();
        Thread curThread = Thread.currentThread();
```

```
                long threadID = curThread.getId();
                JOptionPane.showMessageDialog(null, "thread ID: " +
                    threadID
                   + " current count: " + Counter.curCount);
                Counter.curCount ++;
                ar.add(threadID);
                locks.unlock();
            }
        }
    }
```

Output. The above code results in the following output. Each index represents which thread modified the shared variable when its value was that of the index. As you can see the threads run in first-in first-out order and have no deadlocks.

```
[16, 17, 18, 19, 16, 17, 18, 19, 16, 17, 18,
19, 16, 17, 18, 19,16, 17, 18, 19, 16, 17, 18]
```

3.2 protocols

If all the threads and their corresponding objects follow certain protocols, then you can make certain guarantees about how the program will perform. Ultimately, the purpose of a protocol is to ensure correctness and progress.

3.2.1 correctness (safety)

Methods are correct if they produce the expected result. For example, a first-in first-out queue should dequeue objects in the same order they were enqueued. Methods that are correct in every defined state are called *total methods* whereas methods that are only correct in some states are called *partial methods*. The following are protocols that make correctness claims.

Sequential. Objects and methods that behave sequentially start, then do some processing, then end. This pattern gives programmers a simple way of understanding thread behavior.

Quiescent. Quiescent objects and methods that occur in a sequence have a waiting period between them. For example, if you enqueue, quiescence guarantees the object you enqueued is present when you dequeue.

linearizability. Methods and objects that are linearizable appear to occur instantaneously. This is the most safe but most restrictive protocol. With linearization, you can't have thread related errors, but you also can't do operations in parallel.

3.2.2 progress

Threads that ensure progress must at some point complete. Some protocols make progress claims by blocking, and some don't.

3.2.2.1 protocols that ensure progress. Sometimes threads *depend* on the underlying operating system to ensure progress. Locks depend on the underlying operating system to prevent deadlock and starvation. Generally dependent progress conditions rely heavily on locking and mutual exclusion and are therefore blocking.

Depending on the underlying operating system is often required, but whenever possible it's best to use progress conditions that are independent of the underlying operating system. For example, if the underlying operating system gets preempted, your entire program's state could get confused. Progress conditions that don't depend on the underlying operating system are generally non-blocking

4 conclusion

In summary, this paper discussed the advantages of multithreading, the challenges presented by multithreading, and some ways to overcome those challenges. The paper also included code example for the reader to experiment with.

References

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In Proceedings of the april 18-20, 1967, spring joint computer conferen (pp. 483–485). AFIPS '67 (Spring). doi:10.1145/1465482.1465560

Dijkstra, E. W. (1965). Cooperating sequential processes, technical report ewd-123.

djna. (2019). Thread isolation in java – stackoverflow. [Online; accessed 16-April-2019]. Retrieved from https://stackoverflow.com/questions/1288154/thread-isolation-in-java

Herlihy, M. (2011). The art of multiprocessor programming. Burlington, MA : Morgan Kaufmann.

Peterson, G. (1981). Myths about the mutual exclusion problem. Information Processing Letters, 12(3), 115–116. doi:https://doi.org/10.1016/0020-0190(81)90106-X

Wikipedia contributors. (2019). Amdahl's law — Wikipedia, the free encyclopedia. [Online; accessed 16-April-2019]. Retrieved from https://en.wikipedia.org/w/index.php?title=Amdahl%27s_law&oldid=890132303