Building Scalable Distributed Systems

Ian Gorton

ISBN:

CHAPTER 3

An Overview of Concurrent Systems

Scaling a system naturally involves adding multiple independently moving parts. We run our servers on multiple machines, our databases across multiple storage nodes, all in the quest of adding more capacity. Consequently, our solutions are distributed across multiple locations, with each processing events concurrently.

Any distributed system is hence by definition a concurrent system, even if each node is processing events one at a time. The behavior of the various nodes has to be coordinated in order to make the application behave as desired. As we'll see later, coordinating nodes in a distributed system is fraught with dangers. Luckily, our industry has matured sufficiently to provide complex, powerful software frameworks that hide many of these distributed system nasties (most of the time!) from our applications.

This chapter is concerned with concurrent behavior in our systems on a single node. By explicitly writing our software to perform multiple actions concurrently, we can optimize the processing on a single node, and hence increase our processing capacity both locally and system wide. We'll use the Java 7.0 concurrency capabilities for examples, as these are at a lower level of abstraction than those introduced in Java 8.0. Knowing how concurrent systems operate 'closer to the machine' is useful foundational knowledge when building concurrent and distributed systems.

A final point. This chapter is a concurrency primer. It won't teach you everything you need to know to build complex, high performance concurrent systems. It will also be useful if your experience writing concurrent programs is rusty, or you have written concurrent code in another programming language. The further reading section points you to more comprehensive coverage of this topic for those who wish to delve deeper.

Why Concurrency?

Think of a busy coffee shop. If everyone orders a simple coffee, then the barista can quickly and consistently deliver each drink. Suddenly, the person in front of you orders a soy, vanilla, no sugar, quadruple shot iced brew. Everyone in line sighs and starts reading their social media, In two minutes the line is out of the door.

Processing requests in Web applications is analogous to our coffee example. In a coffee shop, we enlist the help of a new barista to simultaneously make coffees on a different machine to keep the line length in control and serve customers quickly. In software, to make applications responsive, we need to somehow process requests in our server an overlapping manner.

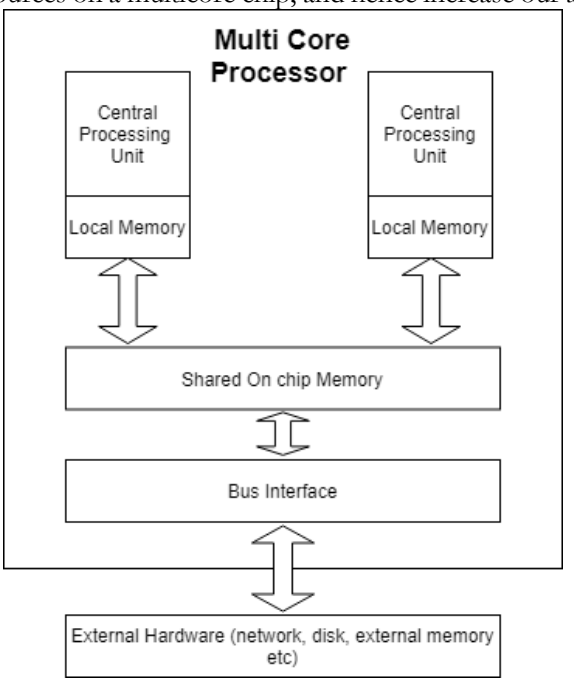
In the good old days of computing, each CPU was only able to execute a single machine instruction at any instant. If our server application runs on such a CPU, why do we need to structure our software systems to execute multiple instructions concurrently? It all seems slightly pointless.

There is actually a very good reason. Virtually every program does more than just execute machine instructions. For example, when a program attempts to read from a file or send a message on the network, it must interact with the hardware subsystem (disk, network card) that is peripheral to the CPU. Reading data from a modern hard disk takes around 10 milliseconds (ms). During this time, the program must wait for the data to be available for processing.

Now, even an ancient CPU such as a circa 1988 Intel 80386[1] can execute more than 10 million instructions per second (mips). 10ms is 1/100th of a second. How many instructions could our 80386 execute in 1/100th second. Do the math. It's a lot! A lot of wasted processing capacity, in fact.

This is how operating systems such as Linux can run multiple programs on a single CPU. While one program is waiting for an input-output (I-O) event, the operating system schedules another program to execute. By explicitly structuring our software to have multiple activities that can be executed in parallel, the operating system can schedule tasks that have work to do while others wait for I-O. We'll see in more detail how this works with Java later in this chapter.

In 2001, IBM introduced the world's first multicore processor, a chip with two CPUs – see Figure 1. Today, even my laptop has 16 CPUs, or cores as they are commonly known. With a multicore chip, a software system that is structured to have multiple parallel activities can be executed in parallel. Well, up the number of available cores, anyway. In this way, we can fully utilize the processing resources on a multicore chip, and hence increase our application's capacity.



[1] https://en.wikipedia.org/wiki/Intel_80386

Figure 1 Simplified view of a multicore processor

The primary way to structure a software system as concurrent activities is to use *threads*. Every programming language has its own threading mechanism. The underlying semantics of all these mechanisms are similar – there are only a few primary threading models in mainstream use – but obviously the syntax varies by language. In the following sections, we'll see how threads are supported in Java.

Threads in Java

Every software process has a single thread of execution by default. This is the thread that the operating system manages when it schedules the process for execution. In Java, the `main()` function you specify as the entry point to your code defines the behavior of this thread. This single thread has access to the program's environment and resources such as open file handles and network connections. As the program calls methods in objects instantiated in the code, the runtime stack is used to pass parameters and manage variable scopes. Standard programming language run time stuff, basically. This is a sequential process.

In your systems, you can use programming language features to create and execute additional threads. Each thread is an independent sequence of execution and has its own runtime stack to manage local object creation and method calls. Each thread also has access to the process' global data and environment. A simple depiction of this scheme is shown in Figure 2.

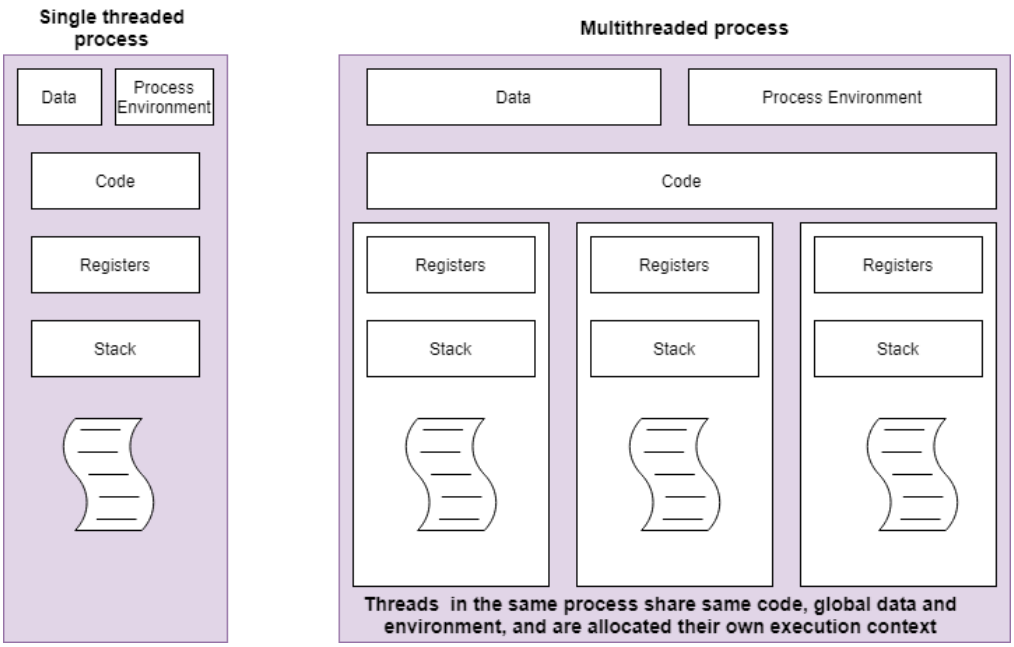


Figure 2 Comparing a single and multithreaded process

In Java, we can define a thread using a class that implements the `Runnable` interface and defines a `run()` method. A simple example is depicted in Figure 3.

```
1.  class NamingThread implements Runnable {
```

```
2.
3.    private String name;
4.
5.    public NamingThread(String threadName) {
6.      name = threadName ;
7.      System.out.println("Constructor called: " + threadName) ;
8.    }
9.
10.   public void run() {
11.     //Display info about this  thread
12.     System.out.println("Run called : " + name);
13.     System.out.println(name + " : " + Thread.currentThread());
14.     // and now terminate  ....
15.   }
16. }
17.
```

Figure 3 A Simple Java Thread class

To execute the thread, we need to construct a `Thread` object using an instance of our `Runnable` and call the `start()` method to invoke the code in its own execution context. This is shown in Figure 4, along with the output of running the code. Note this example has two threads – the `main()` thread and our own `NamingThread`. The main thread starts the `NamingThread`, which executes asynchronously, and then waits for 1 second to give our `run()` method in `NamingThread` ample time to complete. This order of execution can be seen from examining the outputs in Figure 4.

For illustration, we also call the static `currentThread()` method, which returns a string containing:

- The system generated thread identifier
- The thread priority, which by default is 5 for all threads. We'll cover priorities later.
- The identifier of the parent thread – in this example both parent threads are `main`

Note to instantiate a thread, we call the `start()` method, not the `run()` method we define in the `Runnable`. The `start()` method contains the internal system magic to create the execution context for a separate thread to execute (e.g. see Figure 2). If we call `run()` directly, the code will execute, but no new thread will be created. The `run()` method will execute as part of the `main` thread, just like any other Java method invocation that you know and love. You will still have a single threaded code.

```
1. public static void main(String[] args) {
2.
3.    NamingThread name0 = new NamingThread("My first thread");
4.
5.    //Create the thread
6.    Thread t0 = new Thread (name0);
7.
8.    // start the threads
9.    t0.start();
10.
11.   //delay the main thread for a second (1000 miliiseconds)
12.   try {
13.     Thread.currentThread().sleep(1000);
14.   } catch (InterruptedException e) {
```

```
15.       }
16.
17.       //Display info about the main thread and terminate
18.       System.out.println(Thread.currentThread());
19.     }
20.
21. ===EXECUTION OUTPUT===
22. Constructor called: My first thread
23. Run called : My first thread
24. My first thread : Thread[Thread-0,5,main]
25. Thread[main,5,main]
```

Figure 4 Creating and executing a thread

In the example, we use `sleep()` to make sure the `NamimgThread` will terminate before the `main` thread. Coordinating two threads by delaying for an absolute time period (e.g. 1 second in our example) is not a very robust mechanism. What if for some reason - a slower CPU, a long delay reading disk, additional complex logic in the method – our thread doesn't terminate in the expected time frame? In this case, `main` will terminate first – this is not we intend. In general, if you are using absolute times for thread coordination, you are doing it wrong. Almost always. Like 99.999% of the time.

A simple and robust mechanism for one thread to wait until another has completed its work is to use the `join()` method. In Figure 4, we could replace the `try-catch` block with:

```
t0.join();
```

This method causes the calling thread (in our case, `main`) to block until the thread referenced by `t0` terminates. If the referenced thread has terminated before the call to `join()`, then the method call returns immediately. In this way we can coordinate, or synchronize, the behavior of multiple threads. Synchronization of multiple threads is in fact the major focus of rest of this chapter.

Order of Thread Execution

TL;DR The system scheduler (in Java, thie lives in the JVM) controls the order of thread execution. From the programmer's perspective, the order of execution is *non-deterministic*. Get used to that term, we'll use it a lot. The concept of non-determinism is fundamental to understanding multithreaded code.

Let's illustrate this by building on our earlier example. Instead of creating a single `NamingThread`, let's create and start up a few. 3 in fact, as shown in Figure 5:

```
1.  NamingThread name0 = new NamingThread("thread0");
2.        NamingThread name1 = new NamingThread("thread1");
3.        NamingThread name2 = new NamingThread("thread2");
4.
5.        //Create the threads
6.        Thread t0 = new Thread (name0);
7.        Thread t1 = new Thread (name1);
8.        Thread t2 = new Thread (name2);
9.
```

```
10.       // start the threads
11.       t0.start();
12.       t1.start();
13.       t2.start();
14.
15. ===EXECUTION OUTPUT===
16. Run called : thread0
17. thread0 : Thread[Thread-0,5,main]
18. Run called : thread2
19. Run called : thread1
20. thread1 : Thread[Thread-1,5,main]
21. thread2 : Thread[Thread-2,5,main]
22. Thread[main,5,main]
```

Figure 5 Multiple threads exhibiting non-deterministic behavior

The output shown in Figure 5 is a sample from just one execution. You can see the code starts three threads sequentially, namely *t0*, *t1* and *t2* (lines 11-13). Looking at the execution trace, we see thread *t0* completes (line 17) before the others start. Next *t2's* run() method is called (line 18) followed by *t1's* run() method, even though *t1* was started before *t2*. Thread *t1* then runs to completion (line 20) before *t1*, and eventually the *main* thread and the program terminate.

This is just one possible order of execution. If we run this program again, we will almost certainly see a different execution trace. This is because the JVM scheduler is deciding which thread to execute, and for how long. Simply, once the scheduler has given a thread an execution time slot on the CPU, it interrupts the thread and schedules another one to run. This ensures each threads is given an opportunity to make progress. Hence the threads run independently and asynchronously until completion, and the scheduler decides which thread runs when based on a scheduling algorithm.

We will examine the basic scheduling algorithm used later in this chapter. But for now, there is a major implication for programmers, namely, regardless of the order of thread execution, your code should produce correct results. Sounds easy?

Read on.

Problems with Thread – Race Conditions

Non-deterministic execution of threads implies that the code statements that comprise the threads:

- Will execute sequentially as defined for each thread
- Can be overlapped in any order across threads, as how many statements are executed for each thread execution slot is the scheduler.

Hence, when many threads are executed on a single processor, their execution is *interleaved*. The executes some steps from one thread, then performs some steps from another, and so on. If we are executing on a multicore CPU, then we can execute one thread per core. The statements of each thread execution are still however interleaved in a non-deterministic manner.

Now, if every thread simply does its own thing, and is completely independent, this is not a problem. Each thread executes until it terminates, as in our trivial example in Figure 5. Piece of cake! Why are these thread things meant to be complex?

Unfortunately, totally independent threads are not how most multithreaded systems behave.

If you refer back to Figure 2, you will see that multiple threads share the global data within a process. In Java this is both global and static data.

Threads can use shared data structures to coordinate their work and communicate status across threads. For example, we may have threads handling requests from a Web clients, one thread per request. We also want to keep a running total of how many requests we process each day. When a thread completes a request, it increments a global *RequestCounter* object that all threads share and update after each request. At the end of the day, we know how many requests were processed. A lovely solution indeed.

Figure 6 shows a very simple implementation that mimics our example scenario by creating 50k threads to update a shared counter. Note we use a lambda function for brevity to create the threads, and a 'don't do this at home' 5 second delay in `main` to allow the threads to finish.

What you can do at home is run this code a few times and see what results you get. In 10 executions my mean was 49995. I didn't once get the correct answer, namely 50000. Weird.

Why?

```
1. public class RequestCounter {
2.   final static private int NUMTHREADS = 50000;
3.   private int count = 0;
4.
5.   public  void inc() {
6.     count++;
7.   }
8.
9.   public int getVal() {
10.    return this.count;
11.  }
12.
13.  public static void main(String[] args) throws InterruptedException
   {
14.    final RequestCounter counter = new RequestCounter();
15.
16.    for (int i = 0; i < NUMTHREADS; i++) {
17.      // lambda runnable creation
18.      Runnable thread = () -> {counter.inc(); };
19.      new Thread(thread).start();
20.    }
21.
22.    Thread.sleep(5000);
23.    System.out.println("Value should be " + NUMTHREADS + "It is: " +
   counter.getVal());
24.  }
25. }
```

Figure 6 Example of a race condition

The answer lies in how abstract, high-level programming language statements, in Java in this case, are executed on a machine. To perform an increment of a counter, the CPU must (1) load the current value into a register, (2) increment the register value, and (3) write the results back to the original memory location.

As Figure 7 shows, at the machine level these three operations are not indistinguishable, or as more commonly known, atomic. One thread can load the value into the register, but before it writes the incremented value back, the scheduler interrupts it and allows another thread to start. This thread loads the old value of the counter and writes back the incremented value. Eventually

the original thread executes again, and writes back its incremented value, which just happens to be the same as what is already in memory.

We've lost an update. From our 10 tests of the code in Figure 6, we see this is happening on average 5 times in 50000 increments. Hence such events are rare, but even if it happens 1 time in 10 million, you still have an incorrect result.

Thread 1	Thread 2
Reads (x) into register	
Register value + 6	
Writes register value to (x)	
	Reads (x) into register
	Register value + 1
	Writes register value to (x)

Thread 1	Thread 2
Reads (x) into register	
Register value + 6	
	Reads (x) into register
	Register value + 1
	Writes register value to (x)
Writes register value to (x)	

Figure 7 Increments are not atomic at the machine level

This is called a race condition. Race conditions can occur whenever multiple threads make changes to some shared state, in this case a simple counter. Essentially, different interleaving sof the threads can produce different results. Race conditions are insidious, evil errors, because their occurrence is typically rare, and they can be hard to detect as most of the time the answer will be correct. Try running the code in Figure 6 with 1000 threads instead of 50000, and you will see this in action. I got the correct answer four times out of 5.

Same code. Occasionally different results. Like I said – race conditions evil! Luckily, eradicating them is straightforward if you take a few precautions.

The key is to identify and protect *critical sections*. A critical section is a section of code that updates shared data structures, and hence must be executed atomically if accessed by multiple threads. Our example of incrementing a shared counter is an example of a critical section. What about removing an item from a list? We need to delete the head node of the list, and move the reference to the head of the list from the removed node to the next node in the list. Both operations must be performed atomically to maintain the integrity of the list. It's a critical section.

In Java, the `synchronized` keyword defines a critical section. If use to decorate a method, then when multiple threads attempt to call that method on the same shared object, only one is permitted to enter the critical section. All others block until the thread exits the synchronized method, at which point the scheduler chooses the next thread to execute the critical section. We say the execution of the critical section is serialized, as only one thread at a time can be executing the code inside it.

To fix the example in Figure 6, we therefore just need to identify the `inc()` method as a critical section, ie:

```
synchronized public void inc() {
    count++;
  }
```

Test it out as many times as you like. You'll always get the correct answer. Slightly more

formally, this means any interleaving of the threads thar the scheduler throws at us will always produce the correct results.

The synchronized keyword can also be applied to blocks of statements within a method. For example, we could rewrite the above example as:

```
public void inc() {
    synchronized(this){
            count++;
        }
}
```

Underneath the covers, every Java object has a monitor lock, sometimes known as an intrinsic lock, as part of its runtime representation. To enter a synchronized method or block of statements as shown above, a thread must acquire the monitor lock. Only one thread can own the lock at any time, and hence execution is serialized. This, very basically, is how Java implements critical sections.

Aa a rule of thumb, we should keep critical sections as small as possible so that the serialized code is minimized. This can have positive impacts on performance and hence scalability. We'll return to this topic later, but if you are interested, have a read about Amdahl's Law[2] for an explanation on why this is desirable.

Problems with Thread – Deadlock

To ensure correct results in multithreaded code, we have seen we have to restrict non-determinism to serialize access to critical sections. This avoids race conditions. However, if we are not careful, we can write code that restricts non-determinism so much that our program stops. This is formally known as a deadlock.

A deadlock occurs when two threads acquire exclusive access to a resource that the other thread also needs to make progress. The scenario below shows how this can occur:

Two threads sharing access to two shared variables via synchronized blocks

1. *thread 1: enters critical section A*
2. *thread 2: enters critical section B*
3. *thread 1: blocks on entry to critical section B*
4. *thread 2: blocks on entry to critical section A*
5. *Both threads wait forever* ☹

A deadlock, also known as a deadly embrace, causes a program to stop. It doesn't take a vivid imagination to realize that this can cause all sorts of undesirable outcomes. I'm happily texting away while my autonomous vehicle drives me to the bar. Suddenly, the code deadlocks. It won't end well.

Deadlocks occur in more subtle circumstances than the simple example above. The classic example is the Dining Philosophers[3] problem. The story goes like this.

Five philosophers sit around a shared table. Being philosophers, they spend a lot of time thinking deeply. In between bouts of deep thinking, they replenish their brain functions by eating

[2] https://en.wikipedia.org/wiki/Amdahl%27s_law

[3] https://en.wikipedia.org/wiki/Dining_philosophers_problem

from a plate of food that sits in front of them. Hence a philosopher is either eating or thinking, or transitioning between these two states.

In addition, the philosophers must all be very close, highly dexterous and all COVID19 vaccinated friends, as they share chop sticks to eat with. Only five chopsticks are on the table, placed between each philosopher. When one wishes to eat, they follow a protocol of picking up their left chopstick first, then their right chopstick. Once they are ready to think again, they first return the right chopstick, then the left.

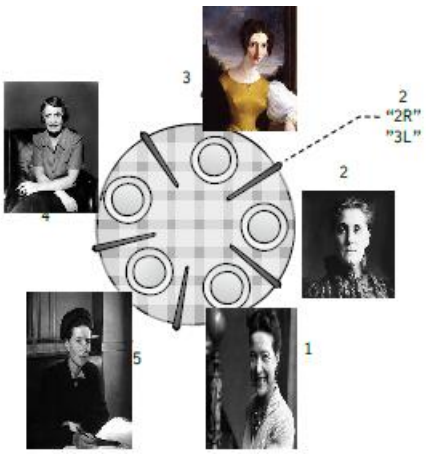


Figure 8 The Dining Philosophers Problem

Figure 8 depicts our all-female philosophers, each identified by a unique number. As each is either concurrently eating or thinking, we can model each philosopher as a thread. The code is shown in Figure 9. The shared chopsticks are represented by instances of the Java `Object` class. As only one object can hold the monitor lock on an object, they are used as entry conditions to the critical sections in which the philosophers acquire the chopsticks they need to eat. After eating, the chopsticks are returned to the table and the lock is released on each so that neighboring philosophers can eat whenever they are ready.

```
1.  public class Philosopher implements Runnable {
2.
3.    private final Object leftChopStick;
4.    private final Object rightChopStick;
5.
6.    Philosopher(Object leftChopStick, Object rightChopStick) {
7.      this.leftChopStick = leftChopStick;
8.      this.rightChopStick = rightChopStick;
9.    }
10.   private void LogEvent(String event) throws InterruptedException {
11.     System.out.println(Thread.currentThread()
12.                                     .getName() + " " + event);
13.     Thread.sleep(1000);
14.   }
15.
16.   public void run() {
17.     try {
18.       while (true) {
19.         LogEvent(": Thinking deeply");
```

```
20.          synchronized (leftChopStick) {
21.            LogEvent( ": Picked up left chop stick");
22.            synchronized (rightChopStick) {
23.              LogEvent(": Picked up right chopstick - eating");
24.              LogEvent(": Put down right chopstick");
25.            }
26.            LogEvent(": Put down left chopstick. Ate too much");
27.          }
28.        } // end while
29.      } catch (InterruptedException e) {
30.         Thread.currentThread().interrupt();
31.    }
32.  }
33. }
```

Figure 9 A Philosopher thread

To bring our philosophers to life, we must instantiate a thread for each and give each philosopher access to its neighboring chopsticks. This is done through the thread constructor call on line 16 in Figure 10. In the `for` loop we create five philosophers and start these as independent threads, where each chopstick is accessible to two threads, one as a left chopstick, and one as a right.

```
1.  private final static int NUMCHOPSTICKS = 5 ;
2.  private final static int NUMPHILOSOPHERS = 5;
3.  public static void main(String[] args) throws Exception {
4.
5.    final Philosopher[] ph = new Philosopher[NUMPHILOSOPHERS];
6.    Object[] chopSticks = new Object[NUMCHOPSTICKS];
7.
8.    for (int i = 0; i < NUMCHOPSTICKS; i++) {
9.      chopSticks[i] = new Object();
10.   }
11.
12.   for (int i = 0; i < NUMPHILOSOPHERS; i++) {
13.     Object leftChopStick = chopSticks[i];
14.     Object rightChopStick = chopSticks[(i + 1) % chopSticks.length];
15.
16.     ph[i] = new Philosopher(leftChopStick, rightChopStick);
17.             }
18.
19.     Thread th = new Thread(ph[i], "Philosopher " + (i + 1));
20.     th.start();
21.   }
22. }
```

Figure 10 Dining Philosophers - deadlocked version

Running this code produces the following output on my first attempt. This was lucky. If you run the code you will almost certainly see different outputs, but the final outcome will be the same.

```
Philosopher 4 : Thinking deeply
Philosopher 5 : Thinking deeply
Philosopher 1 : Thinking deeply
```

```
Philosopher 2 : Thinking deeply
Philosopher 3 : Thinking deeply
Philosopher 4 : Picked up left chop stick
Philosopher 1 : Picked up left chop stick
Philosopher 3 : Picked up left chop stick
Philosopher 5 : Picked up left chop stick
Philosopher 2 : Picked up left chop stick
```

10 lines of output, then … nothing! We have a deadlock. This is a classic circular waiting deadlock. Imagine the following scenario:

1. Each philosopher indulges in a long thinking session
2. Simultaneously, they all decide they are hungry and reach for their left chop stick.
3. No philosopher can eat (proceed) as none can pick up their right chop stick

The philosophers in this situation would figure out some way to proceed by putting down a chop stick or two until one or more can eat. We can sometimes do this is our software by using timeouts on blocking operations. When the timeout expires a thread releases the critical section and retries, allowing other blocked threads a chance to proceed. This is not optimal though, as blocked threads hurt performance, and setting timeout values in an inexact science.

It's much better however to design a solution to be deadlock free. This means that one or more threads will always be able to make progress. With circular wait deadlocks, this can be achieved by imposing a resource allocation protocol on the shared resources, so that threads will not always request resources in the same order.

In the Dining Philosophers problem, we can do this by making sure one of our philosophers picks up their right chop stick first. Let's assume we instruct Philosopher 4 to do this. This leads to a possible sequence of operations such as below:

Philosopher 0 picks up left chopstick (chopStick[0]) then right (chopStick[1])
Philosopher 1 picks up left chopstick (chopStick[1]) then right (chopStick[2])
Philosopher 2 picks up left chopstick (chopStick[2]) then right (chopStick[3])
Philosopher 3 picks up left chopstick (chopStick[3]) then right (chopStick[4])
Philosopher 4 picks up left chopstick (chopStick[0]) then right (chopStick[4])

In this example, Philosopher 4 must block, as Philosopher 0 already has acquired access to chopstick[0]. With Philosopher 4 blocked, Philosopher 3 is assured access to chopstick[4] and can then proceed to satisfy their appetite.

The fix for the Dining Philosophers solution is shown in Figure 11.

```
1.  if (i == NUMPHILOSOPHERS - 1) {
2.    // The last philosopher picks up the right fork first
3.    ph[i] = new Philosopher(rightChopStick, leftChopStick);
4.  } else {
5.    // all others pick up the left chop stick first
6.    ph[i] = new Philosopher(leftChopStick, rightChopStick);
7.  }
8.              }
```

Figure 11 Solving the Dining Philosophers deadlock

More formally we are imposing an ordering on the acquisition of shared resources, such that:

chopStick[0]< chopStick[1]< chopStick[2]< chopStick[3]< chopStick[4]

This means each thread will always attempt to acquire chopstick[0] before chopstick[1], and chopstick[1] before chopstick[2], and so on. For philosopher 4, this means it will attempt to acquire chopstick[0] before chopstick[4], thus breaking the potential for a circular wait deadlock.

Deadlocks are a complicated topic and this section has just scratched the surface. You'll see deadlocks in many deployed systems, and we'll revisit them later when discussing concurrent database accessed and locking.

Thread States

Let's briefly describe the various states that a thread can have during its lifecycle. Multithreaded systems have a system scheduler that decides which threads to run when. In Java, the scheduler is known as a preemptive, priority-based scheduler. In short this means it chooses to execute the highest priority thread which wishes to run.

Every thread has a priority (by default 5, range 0 to 10). A thread inherits its priority from its parent thread. Higher priority threads get scheduled more frequently than lower priority threads, but in most applications having all threads as the default priority suffices.

Scheduling is a JVM-specific implementation detail based on the Java specification. But in general schedulers behave as follows.

The scheduler cycles threads through four states, based on their behavior. These are:

Created: A thread object has been created but its `start()` method has not been invoked. Once `start()` in invoked, the thread enters the runnable state.

Runnable: A thread is able to run. The scheduler will choose which thread(s) to execute in a FIFO manner. Threads then execute until they block (e.g. on a `synchronized` statement), execute a `yield()`, `suspend()` or `sleep()` statement, or the `run()` method terminates, or are preempted by the scheduler. Preemption occurs when a higher priority thread becomes runnable, or when a system-specific timeslice value expires. Timeslicing allows the scheduler to ensure that all threads eventually get chance to execute – no execution hungry threads can hog the CPU.

Blocked: A thread is blocked if it is waiting for a lock, a notification event to occur (e.g. sleep timer to expire, resume() method executed), or is waiting for a network or disk request to complete. When the event a blocked thread is waiting for occurs, it moves back to the runnable state.

Terminated: A thread's run() method has completed or it has called the stop() method. The thread will no longer be scheduled.

An illustration of this scheme is in Figure 12. The scheduler effectively maintains FIFO queue in the *Runnable* state for each thread priority. High priority threads are used typically to respond to events (eg an emergency timer), and execute for a short period of time. Low priority threads are used for background, ongoing tasks like checking for corruption of files on disk through recalculating checksums.

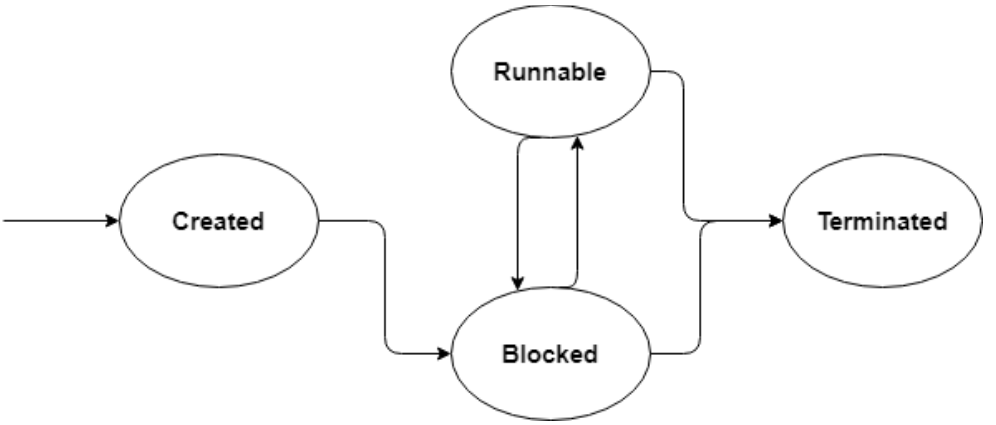


Figure 12 Threads states and transitions

Thread Coordination

There are many problems that require threads with different roles to coordinate their activities. Imagine a collection of threads that accept a document, do some processing on that document (e.g. generate a pdf), and then send the processed document to a shared printer pool. Each printer can only print one document at a time, so it reads from a shared print queue, printing documents in the order they arrive.

This printing problem is an illustration of the classic producer-consumer problem. Producers generate and send messages via a shared FIFO buffer to consumers. Consumers retrieve these messages, process them, and then go to try get more work from the buffer. A simple illustration of this problem is in Figure 13. It's a bit like a 24 hour, 365 day restaurant in New York, the kitchen keeps producing and the wait staff collect the food and deliver to hungry diners. Forever.

Like virtually all real things, the buffer has a limited capacity. Producers generate new items, but if the buffer is full, they must wait until some item(s) have been consumer before they can add the new item to the buffer. Similarly, if the consumers are consuming faster then the producers are producing, them they must wait if there are no items in the buffer, and somehow get alerted when new items arrive.

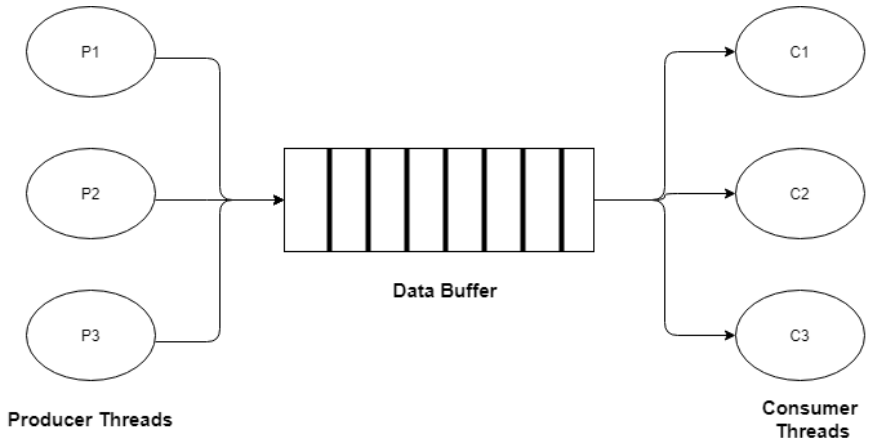


Figure 13 The Producer Consumer Problem

One way for a producer to wait for space in the buffer, or a consumer to wait for an item, is to keep retrying an operation. A producer could sleep for a second, and then retry the put operation until it succeeds. A consumer could do likewise.

This solution is called polling, or busy waiting. It works fine, but as the second name implies, each producer and consumer are using resources (CPU, memory, maybe network?) each time it retries and fails. If this is not a concern, then cool, but in scalable systems we're always aiming to optimize resource usage, and polling can be wasteful.

A better solution is for producers and consumers to block until their desired operation, put or get respectively, can succeed. Blocked threads consume no resources and hence provide an efficient solution. To facilitate this, thread programming models provide blocking operations that enable threads to 'signal' to other threads when an event occurs. With the producer-consumer problem, the basic scheme is as follows:

- When a producer adds an item to the buffer, it sends a signal to any blocked consumers to notify them that there is an item in the buffer
- When a consumer retrieves an item from the buffer, it sends a signal to any blocked producers to notify them there is capacity in the buffer for new items.

In Java, the two basic primitives are `wait()` and `notify()`. Briefly, they work like this:

- A thread may call wait() within a synchronized block if some condition it requires to hold is not true. For example, a thread may attempt to retrieve a message from a buffer, but if the buffer has no messages to retrieve, it calls wait() and blocks until another thread adds a message, sets the condition to true, and calls notify() on the same object.
- notify() wakes up a thread that has called wait() on the object.

These Java primitives are used to implement *guarded blocks*. Guarded blocks use a condition as a guard that must hold before a thread resumes the execution. The code snippet below shows how the guard condition, *empty*, is used to block a thread that is attempting to retrieve a message from an empty buffer.

```
1. while (empty) {
2.   try {
3.     System.out.println("Waiting for a message");
```

```
4.      wait();
5.    } catch (InterruptedException e) {}
6.  }
```

When another thread adds a message to the buffer, it executes notify() as in the code fragment below.

```
1. // Store message.
2. this.message = message;
3. empty = false;
4. // Notify consumer that message is available
5. notify();
```

The full implementation of this example is given in the code examples. There are a number of variations of the wait() and notify() methods, but these go beyond the scope of what we can cover in this overview. And luckily, Java provides us with abstractions that hide this complexity from out code.

An example that is pertinent to the producer-consumer problem is the `BlockingQueue` interface in `java.util.concurrent.BlockingQueue`. A `BlockingQueue` implementation provides a thread-safe object that can be used as the buffer in a producer-consumer scenario. There are 5 different implementations of the BlockingQueue interface. Let's use one of these, the LinkedBlockingQueue, to implement the producer-consumer. This is shown in Figure 14.

```
1.  class ProducerConsumer {
2.     public static void main(String[] args)
3.       BlockingQueue buffer = new LinkedBlockingQueue();
4.       Producer p = new Producer(buffer);
5.       Consumer c = new Consumer(buffer);
6.       new Thread(p).start();
7.       new Thread(c).start();
8.     }
9.   }
10.
11. class Producer implements Runnable {
12.    private final BlockingQueue buffer;
13.    public Producer(BlockingQueue q) { buffer = q; }
14.    public void run() {
15.      try {
16.        while (active) { buffer.put(produce()); }
17.      } catch (InterruptedException ex) { // handle exception}
18.    }
19.    Object produce() { // details omitted, sets active=false }
20.  }
21.
22.  class Consumer implements Runnable {
23.    private final BlockingQueue buffer;
24.    public Consumer(BlockingQueue q) { buffer = q; }
25.    public void run() {
26.      try {
27.        while (active) { consume(buffer.take()); }
28.      } catch (InterruptedException ex) { // handle exception }
29.    }
30.    void consume(Object x) {  // details omitted, sets active=false }
31.  }
```

```
32.
```

Figure 14 Producer-Consumer with a LinkedBlockingQueue

This solution absolves the programmer from having to be concerned with the implementation of coordinating access to the shared buffer, and greatly simplifies the code.

The `java.util.concurrent`[4] package is a treasure trove for building multithreaded Java solutions. The following sections will briefly highlight a few more powerful and extremely useful capabilities.

Thread Pools

Many multithreaded systems need to create and manage a collection of threads that perform similar tasks. For example, in the producer-consumer problem, we can have a collection of producer threads and a collection of consumer threads, all simultaneously adding and removing items, with coordinated access to the shared buffer.

These collections are known as thread pools. Thread pools comprise several worker threads, which typically perform a similar purpose and are managed as a collection. We could create a pool of producer threads which wait for an item to process, write the final product to the buffer, and then wait to accept another item to process. When we stop producing items, the pool can be shutdown in a safe manner, so no partially processed items are lost through an unanticipated exception.

In `java.util.concurrent`, thread pools are supported by the `ExecutorService` interface. This extends the base `Executor` interface with a set of methods to manage and terminate threads in pool. A simple producer-consumer example using a fixed size thread pool is shown in Figure 15 and Figure 16. The `Producer` class in Figure 15 is a `Runnable` that sends a single message to the buffer and then terminates. The `Consumer` simply takes messages from the buffer until an empty string is received, upon which it terminates.

```
1. class Producer implements Runnable {
2.
3.   private final BlockingQueue buffer;
4.
5.   public Producer(BlockingQueue q) { buffer = q; }
6.
7.   @Override
8.   public void run() {
9.
10.    try {
11.      sleep(1000);
12.      buffer.put("hello world");
13.
14.    } catch (InterruptedException ex) {
15.      // handle exception
16.    }
17.  }
```

[4] https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html

```
18.  }
19.
20.  class Consumer implements Runnable {
21.    private final BlockingQueue buffer;
22.
23.    public Consumer(BlockingQueue q) { buffer = q; }
24.
25.    @Override
26.     public void run() {
27.        boolean active = true;
28.        while (active) {
29.            try {
30.               String  s = (String) buffer.take();
31.               System.out.println(s);
32.               if (s.equals("")) active = false;
33.            } catch (InterruptedException ex) {
34.                / handle exception
35.            }
36.        } /
37.        System.out.println("Consumer terminating");
38.      }
39.   }
40.
```

Figure 15 Producer and Consumer for thread pool implementation

In Figure 16, we create a single consumer to take messages from the buffer. We then create fixed size thread pool of size 5 to manage our producers. This causes the JVM to pre-allocate five threads that can be used to execute any `Runnable` objects that are executed by the pool.

In the `for()` loop, we then use the `ExecutorService` to run 20 producers. As there are only 5 threads available in the thread pool, only a maximum of 5 producers will be executed simultaneously. All others are placed in wait queue which is managed by the thread pool. When a producer terminates, the next `Runnable` in the wait queue is executed using any an available thread in the pool.

Once we have requested all the producers to be executed by the thread pool, we call the shutdown() method on the pool. This tells the `ExecutorService` not to accept any more tasks to run. We next call the `awaitTermination()` method, which blocks the calling thread until all the threads managed by the thread pool are idle and no more work is waiting in the wait queue. Once `awaitTermination()` retrurns, we know all messages have been sent to the buffer, and hence send an empty string to the buffer which will act as a termination value for the consumer.

```
1.  public static void main(String[] args) throws
    InterruptedException
2.    {
3.      BlockingQueue buffer = new LinkedBlockingQueue();
4.
5.      //start a single consumer
6.      (new Thread(new Consumer(buffer))).start();
7.
8.      ExecutorService producerPool = Executors.newFixedThreadPool(5);
9.      for (int i = 0; i < 20; i++)
10.       {
```

```
11.         Producer producer = new Producer(buffer) ;
12.         System.out.println("Producer created" );
13.         producerPool.execute(producer);
14.       }
15.
16.       producerPool.shutdown();
17.       producerPool.awaitTermination(10, TimeUnit.SECONDS);
18.
19.       //send termination message toconsumer
20.       buffer.put("");
21.     }
```

Figure 16 Thread pool-based Producer Consumer solution

Like most topics in this chapter, there's many more sophisticated features in the `Executor` framework that can be used to create multithreaded programs. This description has just covered the basics. Thread pools are important as they enable our systems to rationalize the use of resources for threads. Every thread consumes memory, for example the stack size for a thread is typically around 1MB. Also, when we switch execution context to run a new thread, this consumes CPU cycles. If our systems create threads in an undisciplined manner, we will eventually run out of memory and the system will crash. Thread pools allow us to control the number of threads we create, and utilize them efficiently.

We'll revisit thread pools throughout the remainder of this book, as they are a key concept for efficient and scalable management of ever the increasing request loads that servers must satisfy.

Barrier Synchronization

I had a high school friend whose family, at dinner times, would not allow anyone to start eating until the whole family was seated at the table. I thought this was weird, but many years later it serves as a good analogy for the concept known as barrier synchronization. Eating commenced only after all family members arrived at the table.

Multithreaded systems often need to follow such a pattern of behavior. Imagine a multithreaded image processing system. An image arrives and a non-overlapping segment of the image is passed to each thread to perform some transformation upon – think Instagram filters on sterioids. The image is only fully processed when all threads have completed. In software systems, we use a mechanism called barrier synchronization to achieve this style of thread coordination.

The general scheme is shown in Figure 17. In this example, the `main()` thread creates four new threads and all proceed independently until they reach the point of execution defined by the barrier. As each thread arrives, it blocks. When all threads have arrived at this point, the barrier is released, and each thread can continue with its processing.

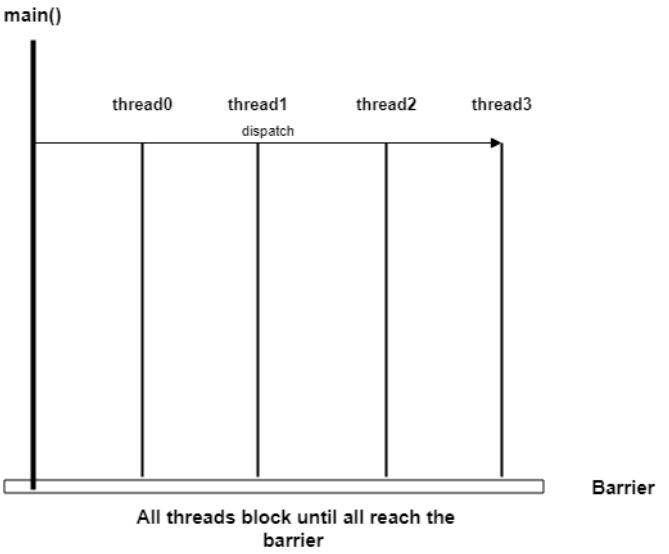


Figure 17 Barrier Synchronization

Java providers three primitives for barrier synchronization. Let's see how one of the three, namely the `CountDownLatch`, works.

When you create a `CountDownLatch`, you pass a value to its constructor that represents the number of threads that must bock at the barrier before they are all allowed to continue. This is called in the thread which is managing the barrier points for the system – in Figure 17this would be main().

```
CountDownLatch  nextPhaseSignal = new CountDownLatch(numThreads);
```

Next we create the worker threads that will perform some actions and then block at the barrier until they all complete. To do this, we need to pass each thread a reference to `CountDownLatch`.

```
for (int i = 0; i < numThreads; i++) {
            Thread worker = new Thread(new WorkerThread(nextPhaseSignal));
            worker.start();
        }
```

After launching the worker threads, the `main()` thread will call the `.await()` method to block until the latch is triggered by the worker threads.

```
nextPhaseSignal.await();
```

Each worker thread will complete its task and before exiting call the `.countDown()` method on the latch. This decrements the latch value. When the last thread calls `.countDown()` and the latch value becomes zero, all threads that have called .await() on the latch transition from the *blocked* to the *runnable* state. In our example this would be the `main()` thread. At this stage we are assured that all workers have completed their assigned task.

```
nextPhaseSignal.countDown();
```

Any subsequent calls to `.countDown()` will return immediately as the latch has been

effectively triggered. Note `.countDown()` is non-blocking, which is a useful property for applications in which threads have more work to do after reaching the barrier.

Our example illustrates using a `CountDownLatch` to block a single thread until a collection of threads have completed their work. We can invert this use case with a latch however if we initialize its value to one. Multiple threads could call `.await()` and block until another thread calls `.countDown()` to release all waiting threads. This example is analogous to a simple gate, which one thread opens to allow a collection of others to continue.

`CountDownLatch` is a simple barrier synchronizer. It's a single use tool, as the initializer value cannot be reset. More sophisticated features are provided by the `CyclicBarrier` and `Phaser` classes in Java and hopefully armed with the knowledge of how barrier synchronization works from this section, these will be straightforward to understand.

Thread-Safe Collections

Many programmers, once they delve into the wonders of multithreaded programs, are surprised to discover that the collections in the `java.util` package[5] are not thread safe. Why, I hear you ask? The answer, luckily, is simple. It's to do with performance. Calling synchronized methods incurs overheads. Hence to attain faster execution for single threaded programs, the collections are not thread-safe.

If you want to share an `ArrayList`, `Map` or 'your favorite data structure' from `java.util` across multiple threads, you must ensure modifications to the structure are placed in critical sections. This approach places the burden on the client of the collection to safely make updates, and hence is error prone – a programmer might forget to make modifications in a `synchronized` block.

It's always safer to use inherently thread-safe collections in out multithreaded code. For this reason, the Java collections framework provides a factory method that creates a thread-safe version of `java.util` collections. Here's an example of creating a thread-safe list.

```
List<String> list = Collections.synchronizedList(new ArrayList<>());
```

What's really happening here is that we are creating a wrapper around the base collection class, which has `synchronized` methods. These delegate the actual work to the original class, in a thread-safe manner of course. You can use this approach for any collection in the `java.util` package, and the general form is:

```
Collections.synchronized….(collection)
```

where "…." is `List, Map, Set`, and so on.

Of course, when using the synchronized wrappers, you pay the performance penalty for acquiring the monitor lock and serializing access from multiple threads. This means the whole collection is locked while a single thread makes a modification, greatly limiting concurrent performance (remember Amdahl's Law?). For this reason, Java 5.0 included the concurrent collections package, namely `java.util.concurrent`. It contains a rich collection of classes specifically designed for efficient multithreaded access.

In fact we've already seen one of these classes – the `LinkedBlockingQueue`. This uses a locking mechanism that enables items to be added to and removed from the queue in parallel. This finer grain locking mechanism utilizes the `java.util.concurrent.lock.Lock` class rather

[5] Except `Vector` and `HashTable`, which are legacy classes, thread safe and slow!

than the monitor lock approach. This allows multiple locks to be utilized on the same collection, hence enabling safe concurrent access.

Another extremely useful collection that provides this finer-grain locking is the `ConcurrentHashMap`. This provides the same methods as the non-thread safe `Hashtable`, but allows non-blocking reads and concurrent writes based on a `concurrencyLevel` value you can pass to the constructor (the default value is 16).

```
ConcurrentHashMap (int initialCapacity, float loadFactor,
                   int concurrencyLevel)
```

Internally, the hash table is divided into individually lockable segment, often known as shards. This means updates can be made concurrently to hash table entries in different shards of the collection, increasing performance.

There are a number of subtle issues concerning iterators and collections, but these are beyond the scope of this chapter. We'll however investigate some of these in the exercises at the end of the chapter.

Summary and Further Reading

This chapter has only brushed the surface of concurrency in general and its support in Java. The best book to continue learning more about the basic concepts of concurrency in is the classic "Java Concurrency in Practice" by Brian Goetz et al. If you understand everything in this book, you'll be writing pretty great concurrent code.

Java concurrency support has moved on considerably however since Java 5. In the world of Java 12 (or whatever version is current when you read this), there are new features such as `CompleteableFutures`, lambda expressions and parallel streams. The functional programming style introduced in Java 8.0 makes it easy to create concurrent solutions without directly creating and managing threads. A good source of knowledge for Java 8.0 features is "Mastering Concurrency Programming with Java 8" by Javier Fernández González.

Other excellent sources include:

- Doug Lea, Concurrent Programming in Java: Design Principles and Patterns, 2nd Edition
- Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft, Java 8 in Action: Lambdas, Streams, and functional-style programming, manning Publications, 1st Edition, 2014.