Parallel Programming

Frühjahrssemester 2019 – Lecture Notes

Sven Pfiffner

Davos|Zürich February 19

Content

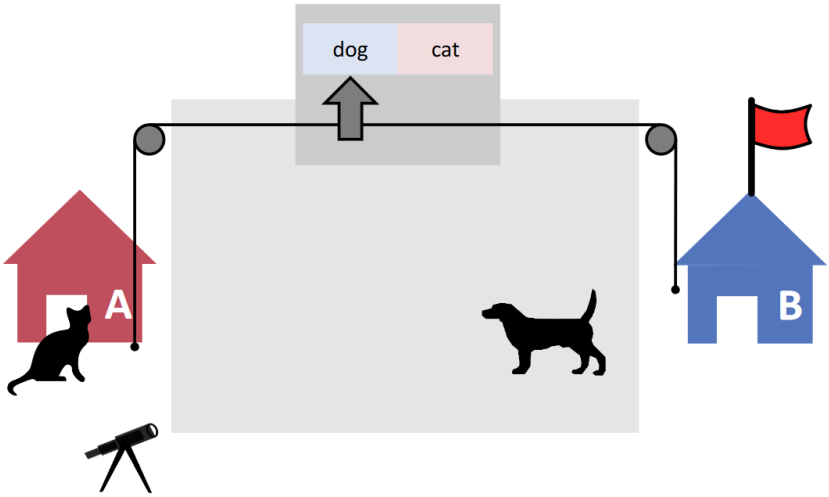
Why even bother?

According to Moores Law the number of transistors on a processing unit doubles every two years. However, this can not be satisfied forever, so computing power of a chip is limited. Hence, we may have to run programs on multiple processors at the same time → Parallel programming

Mutual Exclusion

A mutual exclusion is a program object that prevents simultaneous access to a shared resource. This concept is used in concurrent programming with a critical section, a piece of code in which processes or threads access a shared resource. Only one thread owns the mutex at a time, thus a mutex with a unique name is created when a program starts. When a thread holds a resource, it must lock the mutex from other threads to prevent concurrent access of the resource. Upon releasing the resource, the thread unlocks the mutex

We want to prevent multiple objects from accessing the same resource at the same time



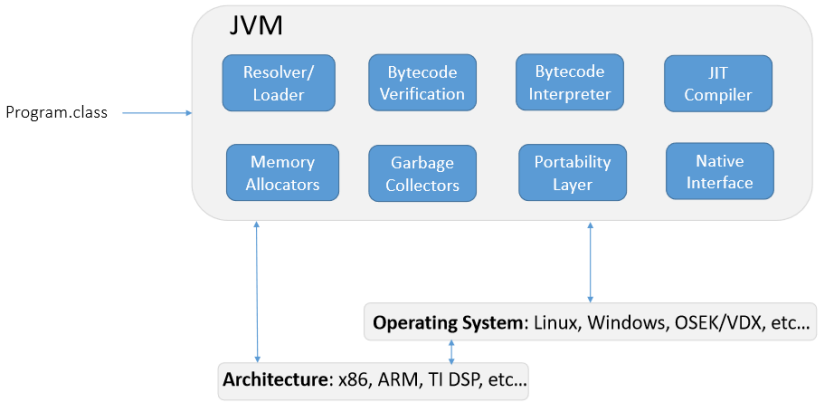
So, we use flags and similar concepts to let the objects communicate with each other

Java Virtual Machine

Note: This chapter is not part of the exam.

A Java virtual machine (JVM) is a virtual machine that enables a computer to run Java programs as well as programs written in other languages that are also compiled to Java bytecode. The JVM is detailed by a specification that formally describes what is required of a JVM implementation. Having a specification ensures interoperability of Java programs across different implementations so that program authors using the Java Development Kit (JDK) need not worry about idiosyncrasies of the underlying hardware platform.

Modules of a JVM

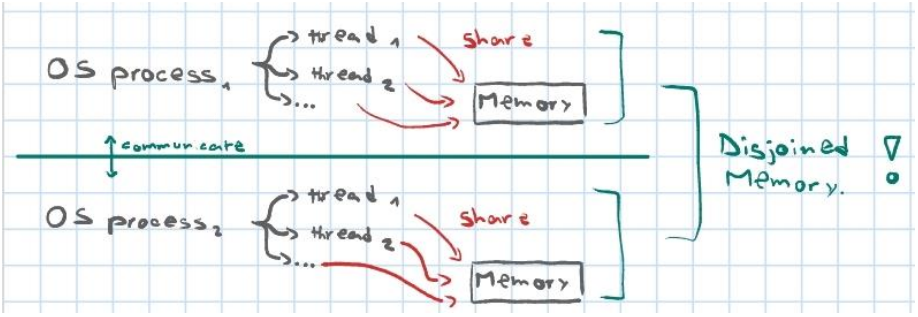


Resolver/Loader	Loads class files and sets up their internal memory. There is a lazy and an eager approach to class loading. Most common JVM's use the lazy approach since it is generally faster
Bytecode Verification	Verifies bytecode. Note: Automated verification is undecidable. All JVM's try to implement a module which recognises as many legal bytecodes as possible.
Bytecode Interpreter	Interprets bytecode using a stack and local variables
JIT Compiler	Compiles bytecode into machine code on demand (e.g. frequently used code), which happens during execution. Note: Since the JIT compiler monitors execution it produces an overhead. Most JVM's use 100's of optimisations to make the JIT as efficient as possible
Memory Allocators	Allocates memory. Since multiple threads may allocate memory at the same time this process must be concurrent. A sequential approach may lead to major pause times
Garbage Collectors	Main objective of Garbage Collector is to free heap memory by destroying unreachable objects.
Portability Layer	Handles communication with different system architectures and operating systems. (Portability is how easily one can I take a program and run it on all of the platforms one cares about.)
Native Interface	A foreign function interface programming framework that enables Java code to call and be called by native applications (programs specific to a hardware and operating system platform) and libraries written in other languages such as C, C++ and assembly.

Basics of Multi-Threading

In order to fully understand the principles of parallel programming in Java, one needs to understand what threads are and how they work.

A thread or a thread of executions is a sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. It is important to know that the implementation of threads/processes differs between operating systems. There can be multiple threads running within one process, executing concurrently and sharing resources such as memory. Different processes and threads which run in different processes do not share these resources but can communicate with each other.



Concept of Multiple threads running in different processes

Sequential vs. parallel composition

Sequential composition (denoted: $x = 1;$) means that we run different operations after each other while parallel composition (denoted: $x = 1; || y = 1;$) means that they run at the same time (in parallel; for example, within two different threads).

Native threads and Green Threads

There can run multiple green threads on the same OS-thread but there can only run one native thread per OS-thread. Green threads are significantly faster than native threads when having more active threads than processors.

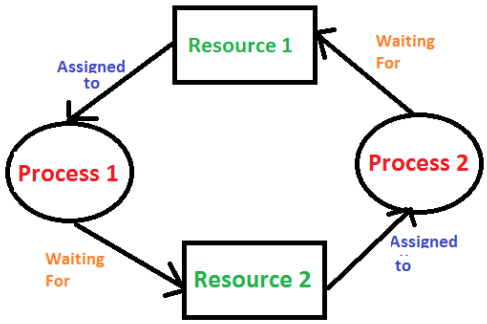
Java initially had support for green threads but unlike most modern green threading implementations it could not scale over multiple processors, making Java unable to utilise multiple cores. Nowadays Java removed green thread support to rely fully on native threading.

Common Parallelism Errors

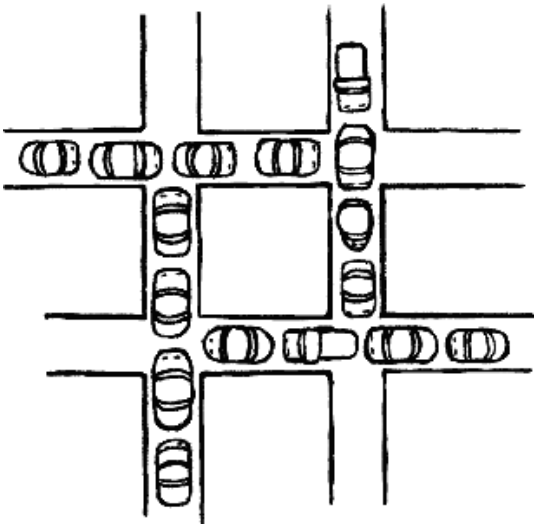
Deadlock

A deadlock (or cyclic waiting) occurs if two or more processes/threads become stuck because they wait for another involved thread to unlock a certain resource.

A deadlock is a situation, where a set of processes stops operating because each process starts waiting for another process in the set to perform a certain operation.



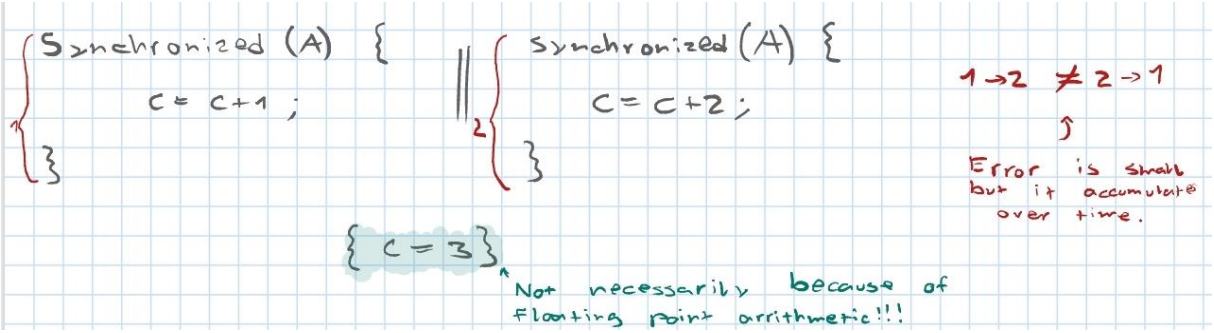
Graphical representation of a deadlock



Deadlock: Analogy in traffic

Floating point associativity

It is important to keep in mind that floating point operations are not associative. This means that $(x + y) + z$ may not be the same as $x + (y + z)$. When constructing parallel programs, we have to contemplate that operations are not sequential, which means that such errors can easily occur.



Digression: Why is that?

This chapter is a summary of the article: **techradar.com|why-computers-suck-at-maths**

Computers perform calculations in quite a different way from the methods that humans use to do arithmetic – and that means that they habitually come up with the wrong answer. […]

Although computers can handle integers (whole numbers), for general-purpose arithmetic they store numbers in floating point format because it's so much more efficient in memory use.

Let's take the double precision floating point representation as an example. It uses 64 bits to store each number and permits values from about -10308 to 10308 (minus and plus 1 followed by 308 zeros, respectively) to be stored. Furthermore, fractional values as small as plus or minus 10-308 (that's a decimal point followed by 307 zeros and then a 1) can be stored.

The secret to this apparently amazing efficiency is approximation. Of those 64 bits, one represents the sign (so whether the value is positive or negative), 52 bits represent the mantissa (that's the actual numbers) and the remaining 11 bits represent the exponent (how many zeros there are or where the decimal point is). So, although a much greater range of numbers can be stored using floating point notation, the precision is less than what can be achieved in integer format, since only 52 bits are available. In fact, 52 bits of binary information represents a 16-bit decimal number, **so any values that differ only in their 17th decimal point will actually be seen as identical.**

We can also look at an analogy to this in within the decimal numbers. Suppose you want to compute $\frac{1}{3}+\frac{1}{3}+\frac{1}{3}$ and have a system that can represent numbers with at most 4 digits. Even though it its obvious for us that the solution to the calculation is 1 ,said system, because it is limited to a given number of digits and hence can not work with the concept of recursion, will compute

$$3333 * 10^{-4} + 3333 * 10^{-4} + 3333 * 10^{-4} = 3333 * 3 * 10^{-4} = 0.9999 \neq 1$$

Exceptions thrown in synchronized blocks

If an exception gets thrown within a synchronized block the synchronized object will be released as if the synchronized scope ends right at the point where the exception is thrown. After that Java catches the exception and executes the error handle if there is one. If not, then the exception is propagated back to the caller of the synchronized block.

Wait, Notify and NotifyAll

The object class in java contains three final methods that allow threads to communicate about the lock status of a resource. These methods are **.wait()**, **.notify()** and **.notifyAll()**. Since they are of utmost importance for parallel programming in java, we will briefly cover each one of them below

Wait

Once the **.wait()** function is called on an object, the current thread is forced to wait until some other thread invokes **.notify()** or **.notifyAll()** on the same object.

```
class ThreadA {
	static void main(String[] args) {
		ThreadB b = new ThreadB();
		b.start();

		synchronized(b) {
			try{
				System.out.println("Waiting for b");
				b.wait();
			} catch(InterruptedException e) {
				e.printStackTrace();
			}
			System.out.println("Done!");
		}
	}
}
```

Notify

For all threads waiting on this object's monitor the **.notify()** method notifies any one of them to wake up arbitrarily. Note that the choice of which thread to wake up is non-deterministic and depends on the tread implementation of the current OS.

```
class ThreadB extends Thread{
	@Override
	public void run() {
		synchronized(this) {
			//Do some stuff
			this.notify();
		}
	}
}
```

NotifyAll

This method works quite similar as **.notify()**, but does not wake one arbitrary thread, but all threads that are waiting on this object's monitor. The awakened threads will complete in the usual manner – like any other thread.

```
class ThreadB extends Thread{
	@Override
	public void run() {
		synchronized(this) {
			//Do some stuff
			this.notifyAll();
		}
	}
}
```

Von Neumann Architecture

The von Neumann architecture is a computer architecture based on a description by John von Neumann and others. Basically, a von Neumann models consists of:

- A processing unit that contains an arithmetic logic unit and processor registers
- A control unit that contains an instruction register and program counter
- Memory that stores data and instructions
- External mass storage
- I/O mechanisms

System level parallelism

Even though we can parallelise our programs using threads and similar concepts, a lot of parallelisation is already done by the system (CPU, Operating System…).

Vectorisation

Automatic vectorization converts a computer program (instruction) from a scalar implementation (process one operation at a time), to a vector implementation (process multiple operation at a time).

Example:

When we write a loop like this:

```
for( i=0; i<4; i++ ){
      c[i] = a[i] + b[i]
}
```

The system performs following operation:

$$c = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} + \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

Instruction level parallelisation (ILP)

Instruction level parallelism (ILP) tries to run as many (actually sequential) instructions in parallel.

Consider the following program:

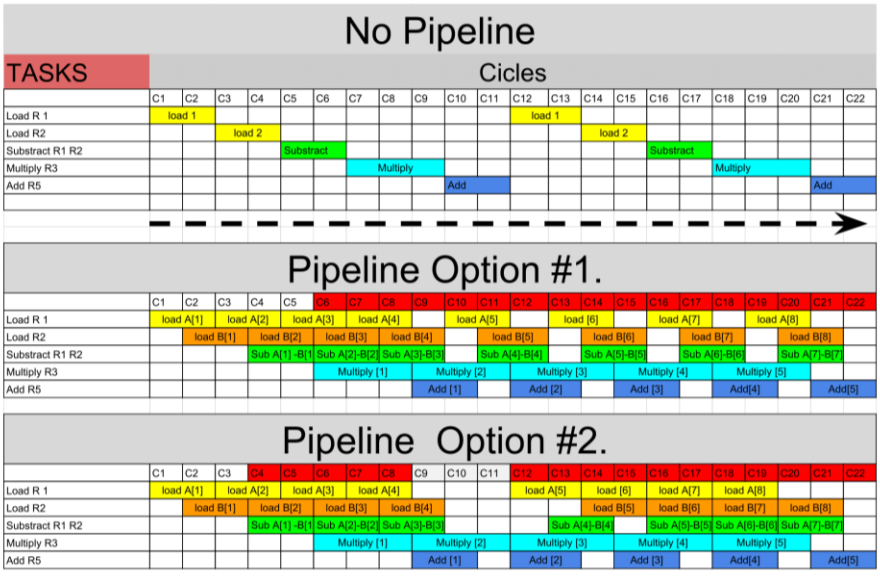
```
e = a+b;
f = c+d;
g = e*f;
```

The operation that gives g its value is dependant on the values of e and f. However, both e and f can be computed simultaneously since they do not depend on any other operation.

Ordinary programs are written to execute instructions in sequence; one after the other, in the order as written by the programmer. ILP allows the compiler and the processor to overlap the execution of multiple instructions or even to change the order in which instructions are executed.

Pipelining

Pipelining is a technique used by most modern CPU's to improve execution speed of sequential instructions. Pipelining uses a so-called pipeline to divide instruction-sets into smaller sub-problems that can be processed in parallel.



Pipelining of an instruction set with 5 elements

Throughput

The throughput of a pipeline describes the quantity of instructions that can be processed within a given unit of time

Imagine you work at a car factory that produces 100 cars a day. The throughput of the factory equals:

$$100\frac{cars}{day}, 4\frac{cars}{hour} \dots$$

Latency

The latency of a pipeline describes the time it takes for an instruction to be processed

Imagine you work at a car factory where producing a car takes 15 minutes. The latency of the factory equals $\frac{1}{4}h$.

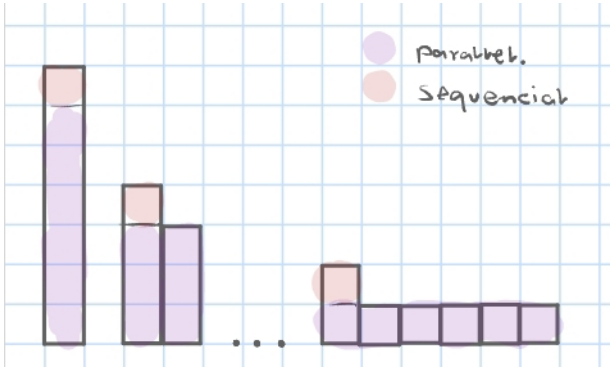
Increasing amount of threads

At this point we may wonder what happens if we have an unlimited amount of threads to use. Will execution time of parallel programs become infinitely small?

Amdahl's Law

Amdahl's Law is rather pessimistic and focuses on the amount of time that can be saved by parallelising the same instruction with an increasing amount of threads. Let W_{seq} be the percentage of sequential work in a program and W_{par} the percentage of parallelized work in a program. According to Amdahl the time needed to process W_{par} decreases with increasing amount of threads. We get following formula for speedup using p threads:

$$S_p = \frac{W_{seq} + W_{par}}{W_{seq} + \frac{W_{par}}{p}}$$

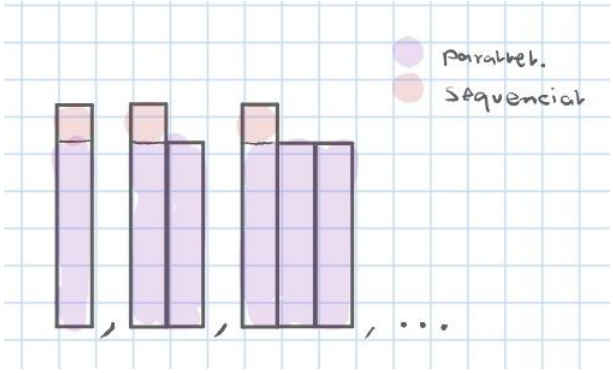


Amdahl's Law Visualized

Gustafson's Law

Gustafson's Law is more optimistic than Amdahl's and focuses on the amount of work that can be done in a given amount of time by parallelising the same instruction with an increasing amount of threads. Let W_{seq} be the percentage of sequential work in a program and W_{par} the percentage of parallelized work in a program. According to Gustafson the amount of W_{par} that can be processed increases with increasing amount of threads. We get following formula for speedup using p threads:

$$S_p = W_{seq} + p(W_{par})$$



Gustafson's Law Visualized

Task parallelism

ExecutorService

The ExecutorService is a framework provided by the JDK which simplifies the execution of tasks in asynchronous mode. Generally speaking, ExecutorService automatically provides a pool of threads and API for assigning tasks to it.

Instantiation

The easiest way to create ExecutorService is to use one of the factory methods of the Executors class

```
ExecutorService executor = Executors.newFixedThreadPool(10);
```

Because ExecutorService is an interface, an instance of any its implementations can be used. For example, the ThreadPoolExecutor class has a few constructors which can be used to configure an executor service and its internal pool.

```
ExecutorService executor =
        new ThreadPoolExecutor(1,1,0L, TimeUnit.MILLISECONDS,
        new LinkedBlockingQueue<Runnable>())
```

Assigning Tasks

ExecutorService can execute both Runnable and Callable tasks. Tasks can be assigned to the ExecutorService using several methods, including ***execute(), submit(), invokeAny()*** and ***invokeAll().***

execute() is void, and it does not give any possibility to get the result of task's execution

```
executor.execute(runnableTask);
```

submit() submits a Callable or Runnable task to an ExecutorService and returns a result of type Future

```
Future<String> future = executor.submit(callableTask);
```

invokeAny() assigns a collection of tasks to an ExecutorService, executes each one and returns the result of a successful execution of one arbitrary task

```
String result = executor.invokeAny(callableTasks);
```

invokeAll() assigns a collection of tasks to an ExecutorService, causing each to be executed, and returns the result of all task executions in the form of a list of objects of type Future

```
List<Future<String>> futures = executor.invokeAll(callableTasks);
```

Shutdown

In general, the ExecutorService will not be automatically destroyed when there is no task to process. It will stay alive and wait for new work to do.

shutdown() makes the ExecutorService stop accepting tasks and shuts it down after all threads finish.

```
executor.shutdown();
```

shutdownNow() tries to destroy the ExecutorService immediately and returns a list of non processed tasks

```
List<Runnable> notExecutedTasks = executor.shutdownNow()
```

ForkJoin Framework

The ForkJoin framework provides tools to help speed up parallel processing by attempting to use all available processor cores – which is accomplished through a divide and conquer approach.

The framework first **"forks"**, recursively breaking the task into smaller independent subtasks until they are simple enough to be executed sequentially. After that, the **"join"** part begins, in which results of all subtasks are recursively joined into a single result, or in the case of a task which returns void, the program simply waits until every subtask is executed.

ForkJoinPool

The ForkJoinPool is an implementation of the ExecutorService that manages worker threads and provides us with tools to get information about the thread pool state and performance.

The ForkJoinPool doesn't create a separate thread for every single subtask. Instead, each thread in the pool has its own double ended queue which stores tasks. **(work-stealing algorithm)**

Instantiation

To use the framework, we need a ForkJoinPool, which can easily be instantiated with a given parallelism level (the number of cores that the pool may use at most)

```
ForkJoinPool pool = new ForkJoinPool(2);
```

Creating tasks

ForkJoinTask is the base type for tasks executed inside ForkJoinPool. In practice, one of its two subclasses should be extended: the RecursiveAction for void tasks and the RecursiveTask<V> for tasks that return a value of type V. They both have an abstract method ***compute()*** in which the task's logic is defined.

Submitting tasks

The most convenient way to submit a task to the thread pool is the ***invoke()*** method, which forks the task and waits for the result.

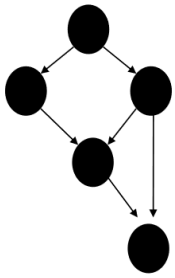
```
int result = pool.invoke(costumRecursiveTask);
```

We can also use the ***invokeAll()*** method to submit a sequence of tasks to the pool. It takes tasks as parameters, forks them and returns a collection of Future objects in the order in which they were produced

Algorithms as DAG's

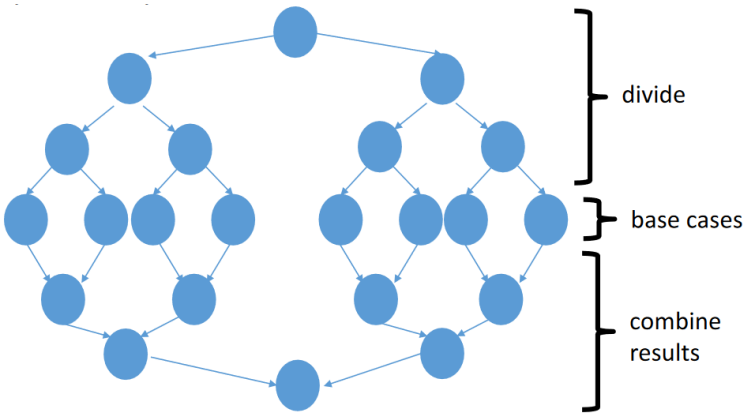
Like all algorithms, parallel algorithms should be correct and efficient. But how do we analyze behaviour of parallel algorithms?

A program execution using fork and join can be seen as a DAG (directed acyclic graph) where the nodes represent pieces of work and the edges the "call direction" of the work (source must finish before destination starts).



- A `fork` "ends a node" and makes two outgoing edges
  - New thread
  - Continuation of current thread
- A `join` "ends a node" and makes a node with two incoming edges
  - Task just ended
  - Last node of thread joined on

As an example, lets look at a DAG of a basic divide-and-conquer algorithm, such as a the ForkJoin implementation of Mergesort:



Efficiency

Let T_p be the running time if there are P processors available. We define following properties of a task:

- Work: How long would it take 1 processor? $= T_1$
  - Just "sequentialize" the recursive forking
- Span: How long would it take an infinite number of processors? $= T_\infty$
  - The longest dependence-chain
  - Example: $O(\log n)$ for summing an array
  - Also called "critical path length" or "computational depth"

Designing parallel algorithms is about decreasing span without increasing work too much!

We know the work (T_1) and the span (T_∞) of an algorithm. But what if we want to compute the execution time for a fixed amount of threads T_p (e.g., for $p = 4$)?

An asymptotically optimal execution would be:

$$T_p = O\left(\frac{T_1}{P} + T_\infty\right)$$

The good news: The ForkJoin framework gives an expected-time guarantee of asymptotically optimal! Our job as ForkJoin Framework users is to pick a good algorithm and write a program. When run, said program creates a DAG of things to do and makes all the nodes a "small-ish" and approximately equal amount of work.

Summary

Task parallelism is a simple yet powerful parallel programming paradigm which is based on the idea of divide and conquer (parallel equivalent to recursion) and can be done by "hand" (discouraged) or using dedicated frameworks such as java's ForkJoin framework.

Getting good performance is not always easy, but the algorithms scale well with additional parallelism.

Locks

A lock object is a shared object that satisfies the following interface

```
Public interface Lock {
        public void lock();
        public void unlock();
}
```

The lock implementation ensures that given simultaneous acquires and/or releases, a correct thing will happen. Thus, a lock implements a mutual exclusion algorithm. Implementation of locks use special hardware and operating-system support, but for this course, it is sufficient to just take them as a primitive.

Re-entrant lock

As the name says, ReentrantLock allows threads to enter lock on a resource more than once. When the thread first enters lock, a hold count is set to one. Before unlocking the thread can re-enter into lock again and every time hold count is incremented by one. For every unlock request, hold count is decremented by one and when hold count is 9, the resource is unlocked.

Re-entrant Locks also offer a fairness parameter, by which the lock would abide by the order of the lock request i.e. after a thread unlocks the resource, the lock would go to the thread which has been waiting for the longest time. This fairness mode is set up by passing true to the constructor of the lock.

Semaphore

A semaphore controls access to a shared resource using a counter. If the counter is greater than zero, then access is allowed. IF it is zero, then access is denied. What the counter is counting are permits that allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore.

In general, to use a semaphore, the thread that wants access to the shared resource tries to acquire a permit.

- If the semaphore's count is greater than zero, then the thread acquires a permit, which causes the semaphore's count to be decremented.
- Otherwise, the thread will be blocked until a permit can be acquired.
- When the thread no longer needs an access to the shared resource, it releases the permit, which causes the semaphore's count to be incremented.
- If there is another thread waiting for a permit, then that thread will acquire a permit at that time.

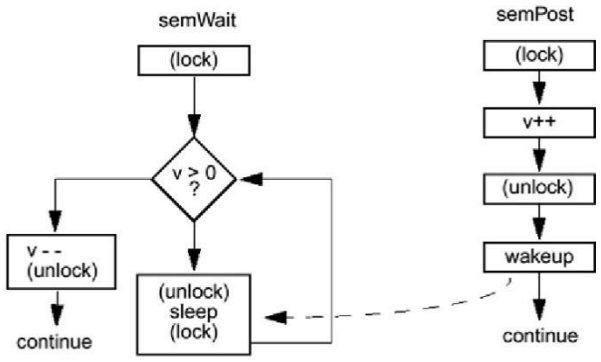


Diagram of a Semaphore's functionality

Example: Consider an ATM cubicle with 4 ATMs. Semaphore can make sure only 4 people can access simultaneously.

```
public class SemaphoreTest {

	//Max 4 people
	static Semaphore semaphore = new Semaphore(4);

	static class MyATMThread extends Thread {
		String name = "";

		MyATMThread(String name) {
			this.name = name;
		}

		public void run() {
			//...
			semaphore.acquire();
			//...
			semaphore.release();
		}
	}

	//...
}
```

Barriers

A barrier is a type of synchronization method. A barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.

When threads get to a barrier, they block until n of them (where n is a property of the barrier) reach the barrier. Unlike *.join()*, the synchronization is not waiting for one other thread to terminate, but rather n other threads to get to the barrier.

Java implements barriers with the CyclicBarrier Class. In practice, CyclicBarriers are used in programs in which we have a fixed number of threads that must wait for each other to reach a common point before continuing execution. The barrier is called cyclic because it can be re-used after the waiting threads are released.

Usage

The constructor for a CyclicBarrier is simple. It takes a single integer that denotes the number of threads that need to call the *.await()* method on the barrier instance to signify reaching the common execution point.

```
public CyclicBarrier(int parties)
```

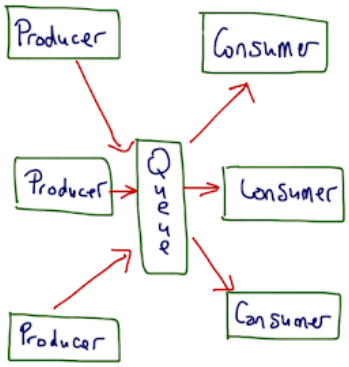
The threads that need to synchronize their execution are also called parties and calling the *.await()* method is how we can register that a certain thread has reached the barrier point. This call is synchronous and the thread calling this method suspends execution till a specified number of threads have called the same method on the barrier. This situation where the required number of threads have called *.await()*, is called tripping the barrier. Optionally, we can pass the second argument to the constructor, which is a Runnable instance. This has logic that would be run by the last thread that trips the barrier.

```
public CyclicBarrier(int parties, Runnable barrierAction)
```

Producer Consumer Pattern

The Producer Consumer pattern is an ideal way of separating work that needs to be done from the execution of that work. Basically, the Producer Consumer pattern contains two major components, which are usually linked by a queue. This means that the separation of the work that needs doing from the execution of that work is achieved by the Producer placing items of work on the queue for later processing instead of dealing with them the moment they are identified. The Consumer is then free to remove the work item from the queue for processing at any time in the future.

This decoupling means that Producers don't care how each item of work will be processed, how many consumers will be processing it or how many other producers there are. Likewise, consumers don't need to know where the work item came from, who put it in the queue, and how many other producers and consumers there are.



Producer Consumer Pattern Scheme

Example: A TV company sends a reporter to every game to feed live updates into a system and sent them back to the studio. On arriving at the studio, the updates will be placed in a queue before being displayed on the screen by a Teletype. This scenario may have many producers but only one or two consumers.

Producer

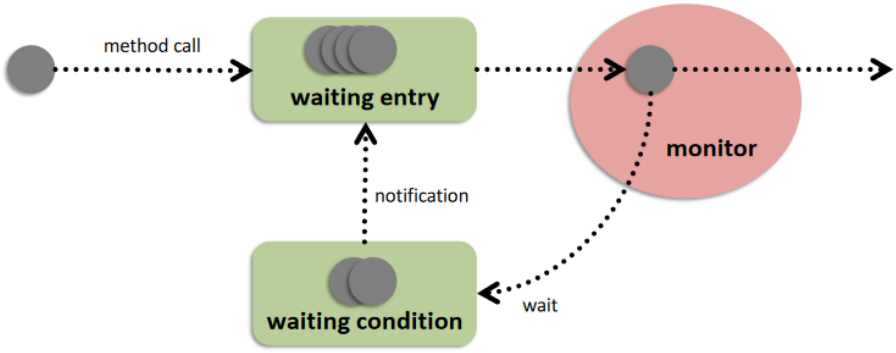
```
//.....
List<Message> matchUpdates = match.getUpdates();
for(Message m: matchUpdates) {
        queue.add(message);
}
//.....
```

Consumer

```
//.....
Message m = queue.take();
System.out.println(m.toString());
//.....
```

Monitors

We can think of a Monitor as an object that contains several attributes which can only be accessed using access methods where only one method can be active at a time. As soon as any access method is run the Monitor is "blocked" until execution is over. Any thread that tries to execute such a method while the monitor is blocked is submitted to a queue.



Threads entering the queue of a monitor

In order to use a Monitor in Java, we must use the **synchronized** keyword when declaring the access methods of a class. All instances of this class can then be seen as Monitors where each synchronized method is an access method. As soon as such a method is run by any thread no other thread can execute any access method of the object until the initial caller is done executing.

Note: The concept of monitors is extremely prone to race conditions since multiple threads can end up sending each other into wait condition forever. [See:

Example: We are developing a system for online banking that can add to and subtracting from multiple bank accounts simultaneously. In order to prevent race conditions in the account balances we declare the bank accounts as monitors with *add()* and *sub()* being access methods.

BankAccount

```
int balance;

//.....

public synchronized void add(int amount) {
        int newBalance = balance;
        newBalance += amount;
        balance = newBalance;
}

public synchronized void sub(int amount) {
        int newBalance = balance;
        newBalance -= amount;
        balance = newBalance;
}

//.....
```

Dining Philosopher|Deadlock]

Example: We are developing a system for online banking that can add to and subtracting from multiple bank accounts simultaneously. In order to prevent race conditions in the account balances we declare the bank accounts as monitors with *add()* and *sub()* being access methods.

BankAccount

```
int balance;

//……..

public synchronized void add(int amount) {
        int newBalance = balance;
        newBalance += amount;
        balance = newBalance;
}

public synchronized void sub(int amount) {
        int newBalance = balance;
        newBalance -= amount;
        balance = newBalance;
}

//……..
```

Dining Philosophers

the dining philosophers problem is an example problem often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them.

Problem: Five silent philosophers sit at a round table with bowls of spaghetti. Forks are placed between each pair of adjacent philosophers. Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when they have both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After an individual philosopher finishes eating, they need to put down both forks so that the forks become available to others. A philosopher can take the fork on their right or the one on their left as they become available but cannot start eating before getting both forks.

The problem is how to design a concurrent algorithm such that no philosopher will starve; i.e., each can forever continue to alternate between eating and thinking, assuming that no philosopher can know when others may want to eat or think.

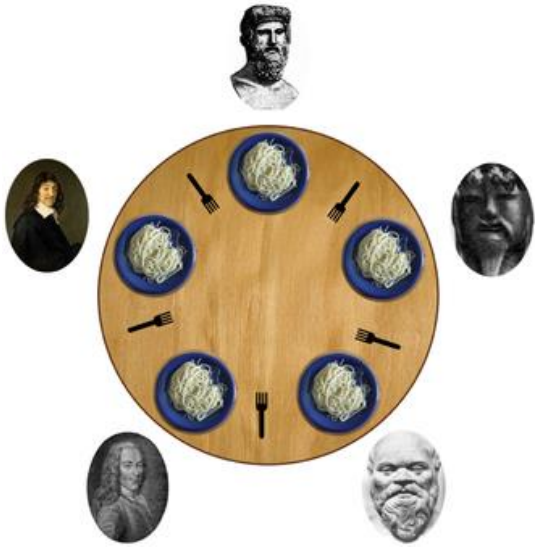


Illustration of the dining philosophers problem

Solutions:

Resource hierarchy solution: We assign a partial order to the forks and establish the convention that all resources will be requested in order, and that no two resources unrelated by order will ever be used by a single philosopher at the same time. We number the forks 1 through 5 and each philosopher will always pick up the lower numbered fork first.

Referee solution: In order to pick up the forks, a philosopher must ask permission of a referee. The referee gives permission to only one philosopher at a time until the philosopher has picked up both of the forks. In addition to introducing a new central entity (the referee), this approach can result in reduced parallelism: if a philosopher is eating and one of their neighbors is requesting the forks, all other philosophers must wait until this request has been fulfilled even if forks for them are still available.

Lock Granularity

The granularity of locks refers to how much of a class is locked at one time. In theory, a class can be locked entirely or as little as an atomic operation can be locked. Such extremes affect the concurrency (number of threads that can access the class) and locking overhead (amount of work to process lock requests).

To achieve optimum performance, a locking scheme must balance the needs of concurrency and overhead.

Coarse Grained

Coarse grained locking refers to a higher level of granularity. The amount of work required to obtain and manage locks is reduced. However, large-scale locks can degrade performance, by making other users wait until locks are released.

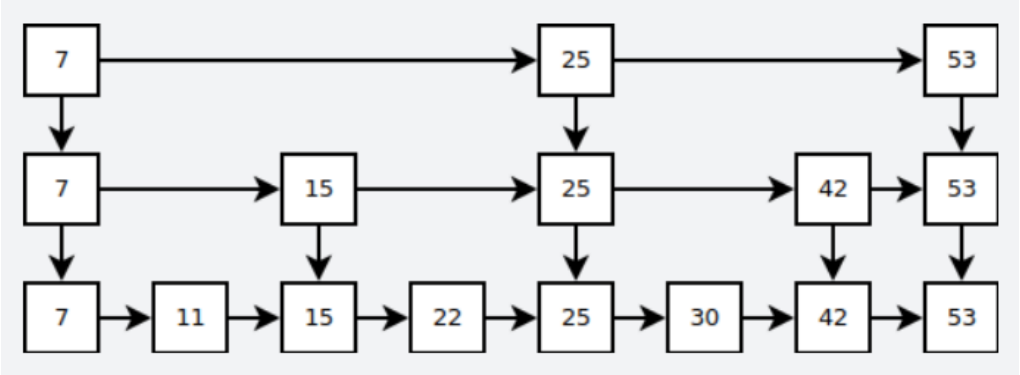
Fine Grained

Fine grained locking refers to a lower level of granularity. Decreasing the lock size makes more of the data accessible to other threads. However, finer granularity locks can also degrade performance, since more work is necessary to maintain and coordinate the increased number of locks.

Skip Lists

Skip lists are a linked-list-like structure which allows for fast search. It consists of a base list holding the elements, together with a tower of lists maintaining a linked hierarchy of subsequences each skipping over fewer elements.

Skip lists perform very well on rapid insertions because there are no rotations or reallocations (compared to tree like structures like AVL). They're also simpler to implement than both self-balancing binary search trees and hash tables.



A common implementation of the skip list

Our skip list consists of lists, stacked such that the n'th list visits a subset of the nodes the $n-1$'th list does. This subset is defined by a probability distribution.

Without Locks