Concepts in Concurrent Programming

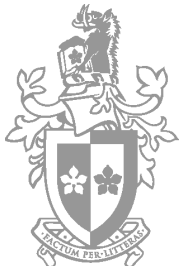
Concepts and Design Forces Documentation

COS30003–Advanced .NET Programming

Swinburne University of Technology

Alex Cummaudo 1744070

Semester 1, 2014



Contents

1 Design Forces in Concurrent Programming

Two views of a system compete with two 'correct' ways of programming of the sake of currency:

1. **Object-Centric Views** structure object collections together as a system of related and interconnected objects
2. **Activity-Centric Views** visualise the threads / activities that objects are affected by / perform, i.e. message-passing from one object across multiple threads

Therefore, an Object-Oriented system is a collection of both **objects** and **activities** (message-passing) behind those objects.

The Object-Centric viewpoint of a system has a fundamental concern with concurrency—no unintended behaviour should occur at runtime. This refers to the **safety** of a concurrent system. Safety is easier said than in practice, as concurrent programs do not execute sequentially for each thread (see Section 1.2.1) and may be blocked, and resumed at a later time. Therefore, although a system may compile and *appears* to be running fine, it may suddenly stop working (Section 1.4), or its output may not make any logical sense.

To prevent this, objects can be *locked down* for exclusive access to a single thread, or controlled via the use of concurrency utilities, thereby ensuring that the object is now safe from mutability from other threads. Herein lies the contending force of **liveness**.

Programs should be as active as possible—running as many activities at once to achieve the program's goals. By locking the object down, safety is acquired, but program liveness is sacrificed to do so—behaviour is now limited since the object can no longer be accessed by multiple threads and therefore cannot perform out multiple actions at once. Thus, too much safety leads to decreasing activity.

These two contending forces are the two conflicting design forces that programmers must balance in concurrent programs, and are discussed further below.

1.1 Safety

Safe concurrent design ensures that errors due to mutability of objects from contenting threads are resolved using, for example, exclusive access to the object from one thread at a time. The safest program will ensure that no objects execute any methods at all, although this will guarantee a program that has no liveness, and therefore achieves nothing.

In ways, this is similar to type-checking in a static language; a variable with a certain type will not be misrepresented if misused as a different type elsewhere. However, a compiler will point this out for the programmer—the *principle* behind ensuring a type will not be misused is similar for safety. Likewise, the *principle* that an object will not be misused by two contending threads leading to corruption of the object by both threads is key to safety.

The primary difference between the two principles are that while the compiler can spot out misused types with type-checking, it cannot spot out poor access to contending threads to an object. Adding a temporary disabled access to these objects must be done manually by the programmer, who must ensure that both safety and liveness in their program is properly balanced. This can be achieved by introducing a range concurrency utilities (Section 2) to battle against issues posed by multi-threaded programs.

1.1.1 Consistency States

Concurrency utilities allow for objects to remain in a **consistent** state—where all internal and dependent values of the object / dependent objects have *logical* and *meaningful* values, not values which are mutated illogically by contending threads. The *meaningful* values of these fields stem from the abstraction process of the initial design of classes from their abstract concepts, known as a set of invariants. Objects are consistent when:

- their fields maintain their invariants—i.e., they are conform to logically acceptable rules
- when they do not initiate new actions where fields are *not* invariant—i.e., using logically unacceptable fields as conditions to initiate actions

Without these utilities, it cannot ensured that objects won't encounter inconsistency errors. **Race conditions**, where whichever thread wins the *race* to mutating a field first gets to mutate the field first, not by logic but by luck, will inevitably ruin the program's safety. This leads to inconstancies in fields and leaves them invariant; thus illogical field values result in the program, via conflicts in reading or writing those fields.

Consider the following example of an unsafe field in an object:

```
_balance = 5;     // Balance is logically 5
_balance += 1;    // Balance is logically 6 (performed by ThreadA)
_balance -= 3;    // Balance is logically 3 (performed by ThreadB)
```

- `ThreadA` wishes to increment the balance of a field `_balance` by 1
  - It reads the value of `_balance`, initially set to `5`, into a temporary value
  - It increments that temporary value to `6`
  - It is then interrupted, and context switching defines that `ThreadB` takes over
- Meanwhile, `ThreadB` wishes to decrement the balance of `_balance` by 3
  - It reads the value of `_balance` (still `5` since `ThreadA` never wrote its temporary value of `6` back into `_balance`) into a temporary value
  - It decrements the value of the temporary `5` by `3` to `2`
  - It writes the value of `2` back into `_balance`
- `ThreadA` is now no longer interrupted and continues its activity
  - It now writes its value of its original temporary value `6` back into `_balance`
- Balance now has an illogical value of `6` after following the logical arithmetic described above.
- It has been subjected to a race condition, where `ThreadA` won, but was interrupted, allowing `ThreadB` to make the `_balance` value no longer consistent with the invariants above.
- In actuality, the final value of `_balance` becomes `6`, and not `3`.

The primary error here is that each thread contains its own **local variables** stored in the temporary values. Those temporary and local values to each thread no longer match logically match their original values when they are written back into the context of the overall program, leading to the errors aforementioned.

1.2 Liveness

All actions should progress to completed program goals. A program that does not consider safety but only liveness will always invoke methods and mutate values with no care of the *consistency* of the values it mutates. This will inevitably cause failure in liveness since progress to the completed goals will result in inaccuracies—consider the example above, where the `_balance` never reaches its intended value of `3`.

1.2.1 Thread Execution Patterns

There are several methods to amend inaccuracies such as the one mentioned. **Blocking** threads will cause activity can stop progressing, in order to improve upon safety. While they impede upon the completion of the program (i.e., the liveness), they are required to maintain consistency in objects, allowing them to follow their invariants.

Frequent and short **thread-blocking** is natural in concurrent programming, and threads typically run via the following pattern:

1. Create the thread
2. Start the thread to make its state runnable
3. When determined by the OS scheduler:
    a. the thread is scheduled to begin executing
    b. the thread is scheduled to be interrupted and blocked for other threads to utilise processing time, causing the thread's runnable state to be `false`
    c. the thread is unblocked, allowing its state to be made runnable again—the thread continues back from step 2.
4. This pattern continues from steps 3 back to 2, until it fails or finishes its execution, where it is then terminated.

1.3 Blocking Mechanisms

This is achievable by utilising blocking mechanisms, such as:

- **Locks**: Allows for exclusive access to an object for one thread only by 'giving' that thread the lock to that object—the thread who holds that lock can safely mutate the object exclusively.
- **Waiting/Joining Threads**: A method is in one thread waits for another thread to finish processing before it can continue processing sequentially.
- **I/O**: A thread is blocked while it waits for input from a process or device

Threads may also not run due to:

- **CPU Exhaustion**: A thread will not run, even though its state is currently runnable, because all computational resources are currently exhausted-
- **Failure**: A thread stops due to error or an uncaught exception.

1.4 Failure To Unblock

Threads may fail to unblock, and remain in a constant un-runnable state, where poor blocking design has been used.

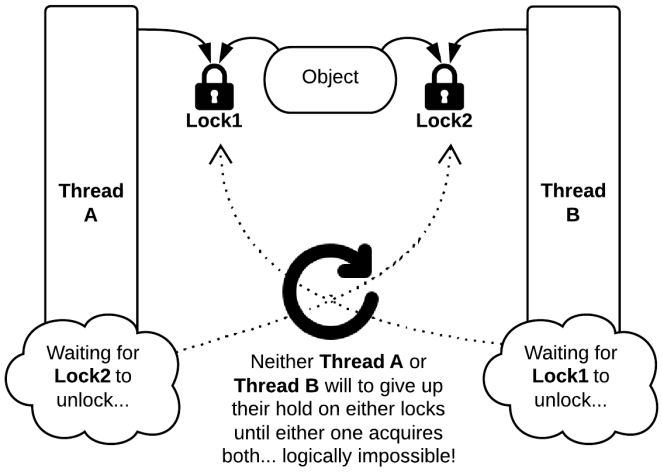


Figure 1.1: Both locks for one object are acquired by different threads. Neither thread will give up their lock for the other thread. The program stalls and encounters a *Deadlock*.

Deadlocks are a common cause of threads never being able to unblock. They usually follow the following pattern:

- **Mutual Exclusion**: Both threads have acquired two different locks of a object.
- **Hold and Wait**: Both threads hold and wait for the other lock to be available
- **No pre-emption**: Threads will wait forever to obtain the lock it is waiting for—they will not unlock for the sake of another thread
- **Circular dependence**: Both threads are dependent on each other giving up their lock holds, which will never occur due to their lack of pre-emption.

Deadlocks can be avoided however; where an object is shared across different threads, ensure that locks are applied:

1. When an object's field is going to be updated

2. When an object's field is going to be accessed

Secondly, ensure that a lock is not applied where it will invoke methods on other locks which have the capacity for a deadlock.

1.5 Performance and Reusability

Locking down objects to block threads severely has a drastic affect on the Performance of a program. Performance extends the liveness concerns of a program. While the liveness design force suggest that programs should eventually reach their goal and finish execution, performance extends that goal by suggesting that programs should be executed *soon and quickly*. As Lea (2000, p.48) outlines, better performance through increased throughput allows more operations to occur per unit of time. This is hindered when too many locks have been used in a program—the more methods that unnecessarily block waiting for locks to be freed, the slower the program will perform. Hence, questioning the need for locks in certain sections of code need to be assessed.

Reusability is the ability for developers to use concurrency utilities in as many ways as possible. This also means that the thread features supported by the runtime environment should be addressed into concurrency utilities also. Within the concurrency utilities discussed in this report, this is achieved mainly via Thread Interruption Exceptions (TIEs) that occur after a thread is interrupted.

To support TIEs, any changes in the state of the concurrency utility needs to be reversed to undo any actions that the thread being interrupted has caused on the utility. This may involve decreasing the number of threads waiting on the utility, for example, since the interrupting thread is now leaving. This is further discussed in later sections of this report.

2 Concurrency Utilities

2.1 Semaphore

2.1.1 Solving Non-Determinism

Semaphores follow the Guarded Suspension principle, which states that:

1. **Dependent or Consumer Threads** will have their threads 'stalled' as they *wait* for a supplier to produce data to signal a pulse for them to continue
2. **Supplier or Producer Threads** will create some form of data, then signalling to all threads that this production has occurred.

This is commonly known as the "Check and Act" process—one or more thread(s) *check* to see if it/they should continue processing, while the other(s) *act* where a certain condition is met. As such, one thread is now able to *depend* on another thread, meaning that threads can work interdependently, and not mistakenly mutate objects if they are trying to access or read from the same values at once (see Section 1.1.1).

In turn, the **non-determinism issue** is solved using semaphores; traditional, single-threaded programs work by the fundamental principles of sequence, selection and repetition. While the two later principles are still obeyed with multi-threaded programs, sequence is not, meaning that "it is not possible to tell, by looking at the program, what will happen when it executes" (Downey, 2005, p.4).

But with a semaphore's `Acquire` and `Release` methods, it is possible to determine execution paths in a multi-threaded environment—the names for these methods help to clarify their meaning and purpose in regards to non-determinism:

- **Acquire**—a thread acquires a semaphore's permission to perform a certain task. It seeks permission to become the semaphore *consumer*.
- **Release**—a thread releases the semaphore's permission once it has completed its task. It releases its *consumer* permission or signals that it has finished *producing* something, allowing a consumer to respond accordingly.

Hence, semaphore's provide a mechanism to deal with the non-deterministic, random flow of concurrent programs via protecting a program against **race conditions** (Section 1.1.1)

which then ensure that one task must be completed by one thread before another task can be completed by another. This, in-turn, provides a sense of *structured flow* for competing threads, thereby allowing the programmer to determine *the order* in which some tasks need to be completed.

In doing so, **safety** (Section 1.1) of the program is increased—by following such a structured flow, threads have fewer chances to inadvertently mutate an object by accessing and writing to it concurrently from two separate threads.

Adversely, the **liveness** (Section 1.2) is forgone—the key aim to complete program goals as soon as possible is suspended whenever the program is forced to stall by threads waiting upon other threads.

2.1.2 .NET Components

`lock` and `readonly` Keywords The `lock` keyword is a mechanism to make an object exclusively accessed by a single thread in a multi-threaded environment, making its block **atomic**.

In particular with the Semaphore class, an arbitrary object[1] `_tokenLock` is locked whenever a thread wishes to access the integer `_tokenCount` of the semaphore (either by reading it or updating it).

To ensure that the `_tokenLock` is not modified in anyway possible by contending threads (see Section 1.1.1), the lock is defined as a `readonly` object. Threads must first acquire the token's lock before proceeding, and making the lock `readonly` fortifies safety within the program by removing any chance of mutation from the threads.

`Monitor` class The `Monitor` class is included under the `System.Threading` package, and grants locks to specific objects so that only a single thread can use that object (Microsoft Corporation, 2014). The `lock` keyword shortens the `Monitor` class by automatically adding `Enter` and `Exit` methods within a try/catch block.

Within the semaphore, two methods are called on this class:

1. `Wait`—This is used within the `while (_tokenCount == 0)` block. Threads are forced to `Wait` while[2] there are no more tokens for the semaphore to distribute—where this

[1] `lock` can only wrap reference types, such as objects, and not value types, such as a non-pointer integer.
[2] A `while` loop is placed here, since when the thread is pulsed, it would not check to see if `_tokenCount` is still zero should the thread lose the race against other contending threads waiting on the `_tokenLock`.

occurs, threads release their hold on `_tokenLock` (so that other threads may attempt to `Acquire` the semaphore, otherwise they would be unable to since the `_tokenLock` is still locked) and then wait on `_tokenLock` for a `Pulse`, in order to continue with their task. Threads which are `Waiting` or **blocked** cause a decrease in program liveness, since the thread is blocked from continuing any execution due to other threads that have already acquired that semaphore.

2. `PulseAll`—This is used when the semaphore is `Released` by a thread, where a signal is made to all threads waiting on the `_tokenLock`. The pulse signals that `_tokenCount` was incremented, and that one less thread is now using the semaphore. The semaphore pulses *all* blocked threads to signify that a token is now available for one of the blocked thread's use. Threads then contend with each other in order to become unblocked and continue with their task (i.e., a race for whichever thread can acquire that token and hence safely continue).

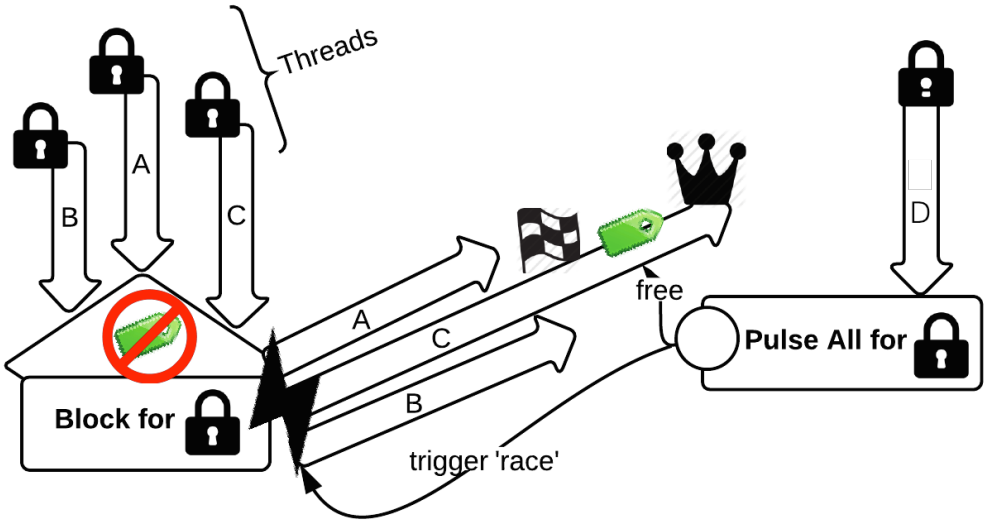


Figure 2.1: Race between waiting threads, and capturing a semaphore's token.

Figure 2.1 illustrates that Threads A, B and C arrive to Acquire the semaphore, but are forced to wait on the lock since there aren't any tokens available. Thread D finishes its task, and Releases the semaphore, pulsing all three threads waiting on the lock and 'frees' up a token. This triggers the race between the three threads, and C wins (possibly because A

and B were scheduled to be blocked 'along the way'). C captures the token and it gets to perform its task using the semaphore.

2.1.3 Performance and Reusability

To ensure for better performance and reusability within the semaphore utility, TIEs (Thread Interruption Exceptions) have been addressed accordingly. This is done by catching any TIEs when under the `Monitor.Wait` block. However, as a TIE is caught, all changes to the state of the Semaphore need to be reversed.

Using a count for the `_waitingThreads`, performance can be tracked and the state of the threads waiting on the semaphore is now accessible. This now means that, rather than unnecessarily pulsing all threads on the lock, pulsing just once (given that there are waiting threads to pulse!) reduces now suffices.

Delaying interrupts extends the reusability factor of the semaphore as well. Since TIEs may be thrown whilst a thread is being released, therefore never actually releasing the semaphore itself, then a `ForceRelease` method can be introduced which *delays* that interrupt, and keeps trying to `Release` the semaphore. If the `Release` method throws a TIE, then the TIE is delayed until a later time, and the `Release` method is called again, and again until the thread is finally unblocked. When `Release` is finally called without throwing a TIE, but a TIE has been delayed, then the original TIE (that has been delayed) can now finally be thrown, but the Semaphore has forcefully been released.

In addition to `ForceRelease`, `TryAcquire` similarly improves performance by trying to acquire the Semaphore in a given time limit. Rather than waiting forever, which can block a thread and hinder performance, trying to acquire the semaphore is a better approach—if the semaphore cannot be acquired within the specified time limit, then the thread can stop trying to acquire that semaphore and move onto to something else. This is well used for threads that may need to acquire a semaphore, but whose liveability do not depend on acquiring that semaphore; the thread *could* acquire the semaphore to complete a task, but if it waits too long, then performance will be hindered. Therefore, to improve performance, the thread gives up waiting entirely. The `TryAcquire`, hence, extends the Semaphore and provides this functionality, factoring in even further design forces.

2.2 Channel

2.2.1 Solving Thread-Shared Variables

A channel solves thread-safe queuing between semaphores. While queues allow objects to be 'transferred' or used from one thread to another, programs will encounter **shared variable** issues during this transfer. The data being exchanged between these threads will need to ensure that no thread is currently writing (*putting*) data onto the queue, and that another data is reading data (`taking`) data off the queue. The safety of the data that travels along the queue can easily overlooked where a thread-safe channel is not being used for this exchange.

Like the example posed in Section 1.1.1, mutation of the data as it travels between each thread will cause safety issues for the integrity of that data as each thread attempts to write and read data simultaneously. Consider the following example:

1. Thread A wishes to place an object, say a string initialised with the value `"keys"`, onto the queue. It begins to put the contents of `"keys"` onto the queue.

2. Thread A is then blocked by the scheduler as it is writing, and has only in fact written `"key"` onto the queue.

3. Thread B reads the contents off the queue, and stores the result `"key"` into a local variable it contains.

4. Thread A is now unblocked by the scheduler, still mid-way writing its initial value, `"keys"`, onto the queue. It continues where it left off, and writes `"s"` onto the queue.

5. Thread B reads the new contents of another object off the queue, this time storing the value `"s"` into a local variable.

As illustrated, it is easy for objects to be mutated as they travel along the queue between threads. Firstly, writing and reading values from the queue have ignored **atomic** principles (where actions in a thread must be executed without being blocked until the action has completed), and thus the action of writing and reading may be blocked by the scheduler at any moment (as was the case illustrated in Thread A).

But this isn't the only issue. Threads should only be able to take objects off queues once it can be ensured that there are values initially there in the first place. There is no way to safely dequeue objects when there are multiple threads trying to dequeue at the same time.

By introducing a semaphore into the equation, we can solve this second issue posed by introducing a `_takePermission` semaphore, which, when used by threads:

1. `Release`s permission when an object is queued onto the channel, and
2. `Acquire`s permission to take an object queued onto the channel off.

Consider the channel now process now, suitably updated with these introduced concurrency utilities:

1. The `_takePermission` semaphore is initialised when the channel is created with a `_tokenCount` of zero—no threads can take things off the channel as there are no tokens for `_takePermission` to give.
2. Thread A wishes to place an object onto the queue. It ensures **atomicity** by using an arbitrary object (`_dataLock`) as a lock to safely place the data on without worrying if the data will be fully written on or not / if the scheduler will block it.
3. Thread A is blocked after it has finished writing the contents.
4. Thread B wishes to take the object off the queue. It seeks to `Acquire` `_takePermission`, but finds that its `_tokenCount` is zero. It `Wait`s on `_takePermission`'s `_tokenLock`, while the `_tokenCount` remains at zero (see Section 2.1).
5. Thread A is now unblocked. It now tells the `_takePermission` to `Release`, as its last action (before blocking) was to finish writing the data onto the queue.
   (a) `_takePermission` increments it `_tokenCount` to one.
   (b) `_takePermission` pulses all threads waiting on its `_tokenLock`, which happens to include the blocked Thread B.
6. Thread B would now contend with any other threads waiting on `_tokenLock` as it has been pulsed by Thread A. Since it is the only thread waiting on the lock in this scenario, it wins in a race against itself, and it then decrements the `_tokenCount` back to zero[3].

[3]Note that in another scenario with more contending threads, these other threads would still be waiting in the while loop at this stage, since they would only see a `_tokenCount` that has been decremented back to zero from the winning Thread B.

7. Thread B successfully dequeues the object that Thread A placed.

While the safety of this program has improved, this is, once again, done at the sake of liveness—we see that other threads would still be waiting for some data to be enqueued, and thus are blocked until this is the case. Nonetheless, threads which attempt to take *nothing* off the thread when two threads attempt to concurrently take something off will drastically reduce the program's safety. As demonstrated, the balance between safety and liveness is a fine one.

2.2.2 Bound Channels

A bound channel would work in an inverse fashion to a standalone channel when it came to reaching its upper limits—threads must be blocked before they attempt to overfill the bound channel with data when it has reached its limits.

Solving this requires another semaphore, `_putPermission`. This semaphore is initialised with the limit count of the bound channel so that we limit the number of threads that can place items on the channel, which comes into play whenever the bound channel is full.

Whenever a thread wishes to place things onto the queue now, it must `Acquire` permission from `_putPermission` to do so:

- If the bound channel is not full, then the thread will decrement `_putPermission` by one, and continue to enqueue that object.
- If the bound channel is full, then `_putPermission`'s `_tokenCount` must be at zero (since too many decrementations have occurred on that count). This means that the thread starts to `Wait` until the token count is raised above zero.

Bound channels still work entirely like any other normal channel—it still contains a `_takePermission` to ensure that data that *actually exists* can be taken off. However, when the bound channel has data taken off it, it `Release`s the `_putPermission`, thereby pulsing any blocked threads waiting to put data onto the queue.

2.2.3 Performance Factors

Similar to Section 2.1.3, the Channel can *Poll* to get object off the channel in a given period of time. This makes use of the semaphore's `TryAcquire` method, whereby the `Poll` method

will wait for a given period of time to receive an object. Rather than hindering a thread's performance and making it wait forever, the thread can instead just wait for this period and if it does not receive anything it can continue on with another task. Originally, a Channel's `Take` method will wait *indefinitely* for an object, which has a big adverse impact on performance if it is not imperative that the thread waits for so long for an object (i.e., an object that is not imperative to the thread's liveness.)

Likewise, a Bound Channel has a similar feature with its *Offer* method; it will `Offer` to place something on a channel within a given amount of time. If the thread waits for anything over this time, then it simply gives up that offer and does not place the object onto the Bound Channel at all. This is particularly useful for a heavily bounded channel that is frequently full; rather than hindering the thread's performance and making it wait indefinitely for it to place its object on the Channel, it will wait only a given specified time and give up if it that timeout is reached. This improves the performance in very much the same way as `Poll` does with a standard channel.

2.2.4 .NET Components

A `Queue` object is required under the `System.Collections.Generic` namespace to actually place the data on.

2.3 Mutex

Protecting a critical section, that is, a section of code that requires atomicity, can usually be achieved with a `lock` statement. The lock ensures that all other threads trying to access the locked object must wait for the thread which has the 'lock' of that object to finish, and give that lock back.

While this is handy in the .NET framework via the `Monitor` class—see Section 2.1.2—not all frameworks provide such a simple mechanism. A mutex solves atomicity in a *generalised* sense; it is useful to develop and use in languages where such concurrency controls are not as easily accessible.

Mutex's achieve this by limiting the functionality of a semaphore (via inheritance). They are essentially constrained semaphores that have either one or no tokens at all times; when a mutex is acquired, a thread will take that one and only token, leaving all other tokens to be blocked until the first thread releases the mutex and puts that only token back for other

threads to use.

This thereby achieves a single thread taking, and subsequently putting back, the token—other threads block as they wait for the mutex to be released, and this in turn allows for atomicity while the executing and acquired thread runs.

A major difference, however, is that—as a mutex is just a semaphore, which in turn is any other object—its syntax does allow for *different* threads to acquire and release the mutex's lock.

Consider this example:

1. Thread A acquires the mutex lock—called `m`.
2. Inside the critical section in which Thread A is now running:
   (a) A predefined variable `data`—initialised with a value zero—is incremented by 1.
   (b) Thread A creates a new Thread, call it Thread X:
      i. Thread X's execution method passes in `m` that Thread A has locked, as well as the value of `data` by reference.
      ii. Thread X's method subtracts `data` by 1.
      iii. Thread X releases the mutex `m`.
3. After creating Thread X, Thread A calls the `Start` method on Thread X.
4. Thread X concurrently decrements `data`.
5. Thread A releases `m`.
6. Thread X releases `m` again as per 2(b)iii.

```
private int _data;
private Mutex _m;

private void AStart()
{
    // Mutex acquired here in thread A
    _m.Acquire();
    _data++;
    Thread x = new Thread(XStart);
    x.Start();
    Thread.Sleep(1);
```

```
        // Output of _data may be 1 or 0, depending if X ran fast enough
        Console.WriteLine(_data);
    }

    private void XStart()
    {
        _data--;
        // Mutex released here in thread X
        _m.Release();
    }

    // Main thread starts here
    public void RunTest()
    {
        _data = 0;
        _m = new Mutex();

        Thread a = new Thread(AStart);
        a.Start();
    }
```

As shown, there are two issues: when Thread A has finished with its critical section, the value of `data` (expectedly one) is in fact zero, due to the decrementation that occurred in Step 4. Atomicity rules have been violated by creating the second thread—Thread X never acquired the mutex `m`, yet it still had access to the variables that are concurrently being executing on in Thread A.

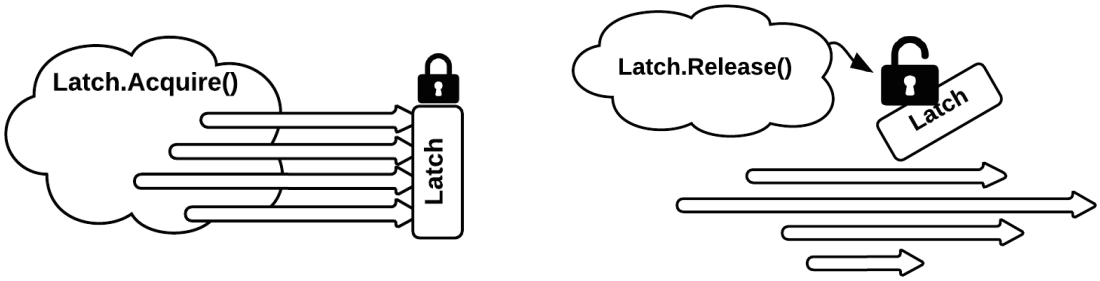
Thus, by introducing the flexibility of allowing mutex's being released by a different thread elsewhere, rather than in a simple, block-type structure as per `lock`, safety of the program has been violated. Creating and starting a second thread within the mutex's critical section violates atomicity principles; the programmer should be careful in implementing the mutex.

A resolution around this is to constrain the release method so that if it is called with an internal `_tokenCount` of zero (meaning it is being released while it is already unlocked), then an exception should be thrown.

2.4 Latch

A latch is a a simple mechanism that will block all threads which attempt to acquire it until it is released—when the latch is released, all threads that it previously blocked will concurrently continue their execution.

This is best illustrated in Figures 2.2a and 2.2b.



(a) Latch initially closed—threads blocked **(b)** Latch opens—floodgate of threads

Figure 2.2: Workings of a latch—closed and blocking all acquiring threads; open and letting all acquiring threads through.

Internally, the latch uses a semaphore initialised with a token count of zero. When threads attempt to acquire the latch, they will automatically be blocked by the semaphore's wait on no token loop. When the latch is released, so is its internal semaphore, allowing one of the blocked threads to continue executing. The thread's immediate next action is to tell the latch's semaphore to release another token for the next waiting thread (i.e., give back the token it just acquired); this achieves the effect of passing the semaphore's lock along to every thread waiting and allowing them to move on.

When there are no more threads waiting, the semaphore's token count is now greater than zero; when a new thread comes along to the latch and acquires it, it will automatically pass through, and (again) release the token it just acquired so that the next thread that comes along can do the same thing.

This one-instance unlock behaviour can be quite useful for keeping threads blocked until a certain condition is met. All threads will be held up until this condition, which may improve safety since these threads may have adverse affects on variables should they continue.

For example, a program has several threads waiting to process a set of data—sum, mean, standard deviation etc. Each calculation set is done by its own thread. For the calculations to begin, however, there needs to be some data to work with.

A latch is useful here; the threads waiting to calculate on the data can sit and wait (be blocked) for data input. When another thread has finished loading the set of data (e.g., from a file) then these calculations can begin; the loading-from-file thread can signal for the latch to be opened, thereby allowing all of the calculator-threads waiting at the latch to make it through and start executing.

While the liveness of the program was significantly hindered as the calculator-threads were blocked during their waiting period, safety was inevitably guaranteed—if these threads did not wait and began to work on a half-loaded set of data from the loader-thread, there would be incorrect calculations on that data since the data wasn't even loaded yet. Forcing the calculating threads to wait for their data is required for correctness.

2.5 Light Switch

Light Switches applies a First-In-Last-Out principle for threads onto a semaphore, meaning that a group of threads can collectively request permission, and release it when the group of threads have finished processing. Light switches are initialised with a single semaphore (or mutex) which it wraps so that threads can *collectively* yet safely acquire this semaphore once—as a group.

The first thread in the group to begin executing will acquire the light switch's permission (metaphorically turning the switch *on*) and the last thread to finish executing will release that light switch's permission (turning the switch *off*). While the switch is on, any thread which attempts to access the light switch's internal semaphore *directly* will be blocked, since there are still threads collectively inside the 'room' in which the light switch operates in.

Thus, whenever a group of threads are executing, and can do so as per the light switch's acquired permission, this group is live. External threads that then attempt to access the light switch's semaphore remain inactive, reducing the liveness of the program.

A light switch puts emphasis that threads which share some common purpose or task (e.g., a group of threads that read variables) should be concurrently active—i.e., their liveness shouldn't be hindered just because they're different threads; liveness should remain unaffected due to their common purpose. While this maintains safety from external threads while these grouped threads are active and are thus blocked (see Figure 2.5) it doesn't guarantee safety *internally* from the same group—ensuring that the group do not collectively mutate variables is a requirement within the light switch block. The light switch will *not* protect variables from the grouped threads that have acquired it, only from external ones. The collective group shouldn't mutate the same variables at once whilst they are in the 'room', as this would break safety principles. It does, however, provide safety against other threads which attempt to mutate variables inside a light switch block and are not part of the collective group.

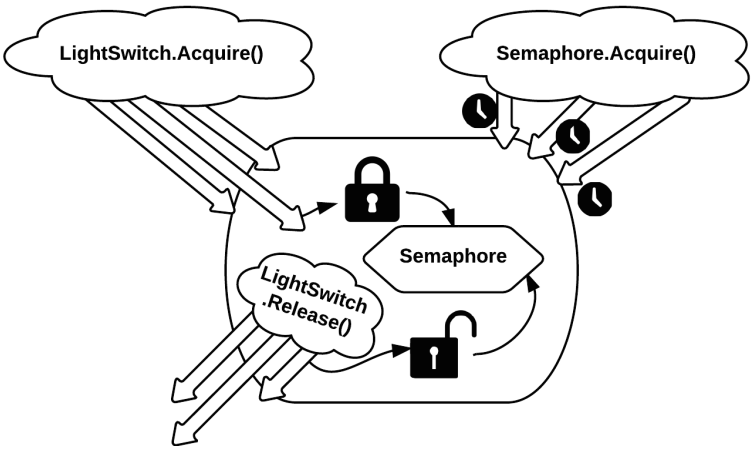


Figure 2.3: A group of threads that acquire a semaphore via a light switch have exclusive access (collectively) to that semaphore from other threads.

The group of threads shown in Figure 2.5 acquire the light switch collectively as each thread calls `LightSwitch.Acquire()` together. Notice the first thread that enters the 'room' (i.e., the first thread that successfully acquires the switch) will lock the light switch's internal semaphore on behalf of the entire group. When the last thread exists the 'room' (i.e., the last thread to release the light switch when all other threads in that group have already released it), then that last thread will release the switch's internal semaphore on behalf of the group. This means that other waiting threads on that *semaphore* (on the right of the figure) can now race to acquire the semaphore after the last thread in the group has left.

2.6 Reader Writer Lock

2.6.1 Improving Atomicity Liveness

A reader writer lock can significantly improve the liveness of a program—referring back to Section 1.1.1, it is noticeable that concurrent reads do not have any affect whatsoever on the safety of the program. If two threads wish to read the value of a variable, there are no adverse affects as there is with mutating variables. As long as the system maintains its invariants when a thread accesses a variable—which is always the case when a value is read, and not necessarily the case when it is written to—then safety remains unaffected.

A read action does not need to be atomic; locking a value down whenever it needs to

be read, thereby blocking other threads, reduces system liveness and gains no safety at all. Safety is paramount for write actions, but read actions are only unsafe if they are reading values which do not comply with their invariants (i.e., if they were mutated concurrently and not atomically).

Rather than locking read actions down, reader writer locks allow for concurrent reads to occur as many times as needed; when a write action is needed, then the read actions can be blocked, the write action can occur atomically, and then the read actions can continue. This is achievable using:

1. a mutex for write actions, and

2. a light switch for read actions.

The read write lock can be acquired and released as a writer or as a reader. When read actions are needed by one or multiple threads, then the light switch inside the lock will be acquired; as long as there are readers in the 'room', then they can consistently read values, improving system liveness without affecting system safety. When a writer thread wishes to acquire the lock, it will attempt to acquire the mutex used for the reader light switch.

As discussed in Section **??**, the writer is an external thread not part of the collective reader's group; it waits outside the 'room' until all readers have finished their tasks. As the last reader thread leaves that room and switches off the light switch, only then can the writer attempt to acquire the mutex. When it does, it can then write in an atomic fashion, maintaining system safety (since any new readers will wait for the writer to finish before it enters the room). The balance between liveness and safety is shifted towards liveness for readers with this lock.

2.6.2 Greedy Readers

An apparent problem exists by introducing the flexibility of having concurrent multiple readers or singular writers. Whilst the lock is more live for readers, there is a *potential* for writers to be completely blocked out from the lock at all times. Consider the following scenario:

1. A group of threads wish to read something and acquire the lock as readers.

2. All but one thread leaves; this last thread still acquires the lock and is still inside the 'room' inside the lock's light switch.

3. A thread wishes to acquire as a writer—it is blocked since the reader has still acquired the data mutex via the light switch.

4. More reader threads come back to acquire the light switch.

5. Steps 1 and 4 repeat forever.

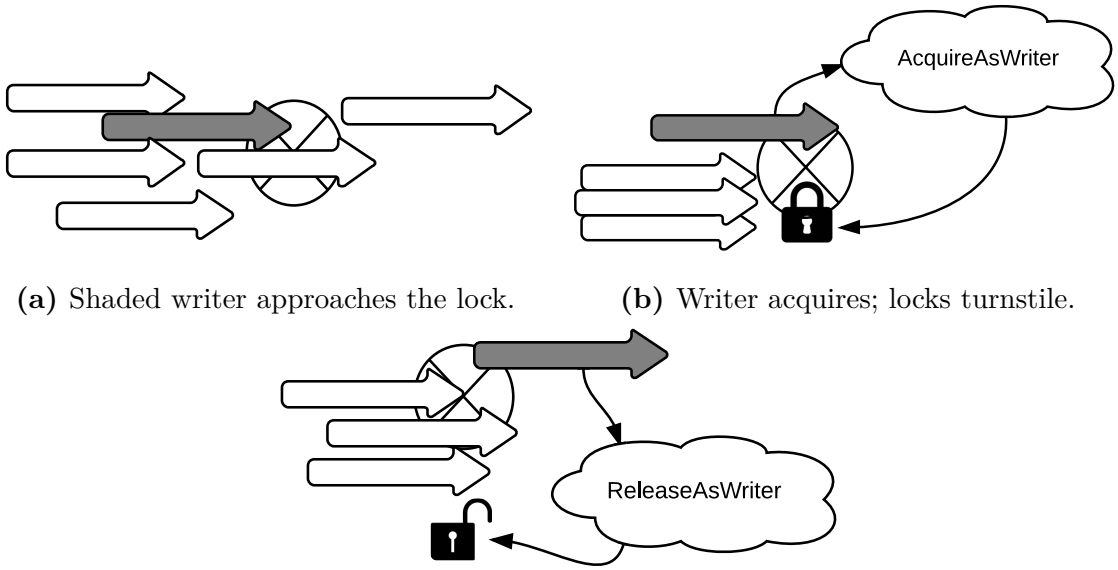
As demonstrated, writer threads will *always* be blocked out should there be a thread in there—writer liveness is thus compromised since there is the capacity for writers never to have a go at writing, even when they acquire the reader-writer lock.

A turnstile, say `_readersTurnstile`, must be introduced. This turnstile will allow readers to pass through as soon as they acquire the reader-writer lock as a reader (Figure 2.4a). However, writers will *also* acquire this turnstile, but not release it immediately like the readers do, thereby preventing any further readers from coming on in and preventing writers from writing (Figure 2.4b). Thus, by 'pinning down' the turnstile, readers cannot move past the turnstile, and thus they can't monopolize the light switch. Writers will release that pin when they are released—forcing readers to queue up at their turnstile when they acquire if a writer wants to acquire also, then releeasing that queue when the writer has finished (Figure 2.4c). This improves the liveness of writers, whilst maintaining the safety of the lock.

2.6.3 Improving Reader Liveness

The liveness of readers is less than it could potentially be. By solving the greedy-readers issue, the liveness of readers is degraded—the whole purpose of a reader-writer lock is to allow concurrent readers to read, thereby improving liveness. However, by introducing the turnstile in Section 2.6.2, writers that pin the turnstile down prevent *groups* of readers from passing through.

This introduces unnecessary atomicity for readers in the lock. Conside the example posed in Figure 2.5; if two readers and two writers come in, and the first writer acquires, the first reader acquires, then the second writer acquires, atomicity for the readers has implicitly occurred. While this has no impact on safety, the liveness of the lock should be improved



(a) Shaded writer approaches the lock.

(b) Writer acquires; locks turnstile.

(c) Writer releases; unlocks turnstile.

Figure 2.4: Solving the greedy reader problem with a reader's turnstile.

to allow *batches* of readers to read together; in the example, both readers should have read *concurrently*, while the writers should work atomically.

Solving this issue requires another turnstile—`_writersTurnstile`—and a count on the number of readers waiting to acquire the light switch—`_waitingReadersCount`.

The `_writersTurnstile` is pinned down by a writer when it is acquired; preventing new threads to acquire the lock as a writer and thereby preventing two writers from acquiring the lock. As readers come in to acquire the light switch (and are consequently blocked from the writer and its acquired mutex), the `_waitingReadersCount` is incremented; a reader is *waiting* for the writer to release its hold on the light switch's mutex.

When a writer is going to be released, it checks for the number of waiting readers before it releases the `_writersTurnstile` and lets the next writer in for exclusive access; if there are waiting writers, it will use .NET's `Monitor` class to wait (on an arbitrary `_writerWait` object) and prevent future writers from coming in. This will give a chance for *multiple* waiting readers to acquire the light switch collectively as soon as its mutex is given up by a writer; when a reader has successfully acquired the light switch, it will indicate that there is now one less waiting reader waiting on the light switch; `_waitingReadersCount` is decremented,

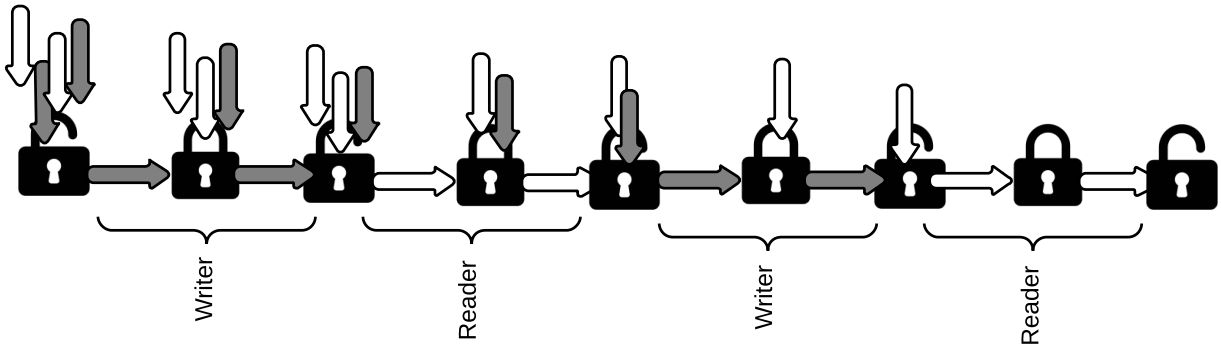


Figure 2.5: Atomicity has occurred for both readers and writers; atomicity is only applicable (for safety reasons) to writers and not to readers.

and threads waiting on `_writerWait` are pulsed via `Monitor`. This will wake the blocked writer thread, and release the `_writersTurnstile` to allow more writers in. Thus, writers are blocked if there are multiple readers waiting to acquire the light switch since the releasing writer will be blocked until there is one less reader waiting on that light switch.

2.7 Barrier

Barriers are a handy utility to have for parallel programming, where lots of threads concurrently execute the same task—e.g., calculations—on a single set of data split apart. Threads *arrive* at a barrier, and when a certain number arrive, then the barrier opens, to which those threads can begin executing that task.

As threads arrive at the barrier, they are blocked until the the 'captain' thread (or the thread that releases the barrier—i.e., the last thread in until the limit is reached) indicates that they should be released. Safety issues do occur when barriers are reused—should the captain thread release the barrier, it must be ensured that no other threads can go back into the barrier after the captain thread has released it (i.e., the *same* group of threads must execute and finish executing until it is made reusable again).

Consider the following example:

- A barrier with a limit of 4 is created.
- Three threads arrive at the barrier; all three are blocked at the barrier's semaphore `_goPermission`.

- The fourth, 'captain' thread arrives; it releases the block on the three waiting threads by releasing three tokens for those waiting threads.

- Another thread arrives at the barrier; it 'snatches' one of the released tokens for one of the original threads at the barrier; that original thread never got the token it deserved and is thus still blocked.

This is problematic for a parallel calculator; consider the four threads receive one fourth of a set of data each. Each of these threads need to calculate the sum of their subset. The fifth thread in this scenario arrives at the barrier to calculate the sum of the subsets; should the fifth thread arrive and *snatch* a subset calculator's deserving token, then that subset will never be calculated. Therefore, the fifth thread will calculate the sum on incomplete data (i.e., one fourth of the set of data was not calculated). The safety of the program is clearly undermined.

Solving this issue can be done by introducing a turnstile; `_lockdown`. When all non-captain threads arrive, they will approach that turnstile, initially unlocked, and pass straight through, but be blocked at the barrier's `_goPermission` semaphore. The captain thread will also go through, although once it realises that is the captain thread (i.e., `_threadCount == _limit`) it pins the turnstile down, preventing any more threads from coming in and acquiring the barrier's `_goPermission`—thus 'snatching' a token is now impossible for a new, incoming thread since it is waiting at `_lockdown`. The captain thread releases all the non-captain threads that came before it, allowing them past `_goPermission`. When the last non-captain thread is about to leave, it will unpin the `_lockdown` turnstile, thereby allowing the next batch of threads wanting to arrive at the barrier in.

2.8 Exchanger

Exchangers allow for two threads to exchange resources. Safety must be taken into account here; ensuring that the exact same two threads exchange *exactly* the same data is necessary.

Two semaphores, `_putPermission` and `_takePermission`, are required to control the exchange of data. The first thread that enters the exchange method block will seek the `_putPermission` acquisition in order to place its object into a temporary object slot. Once it has placed it, it releases the `_putPermission` for the second thread in the exchange. It will then acquire `_takePermission` (initially locked), and block until the second thread comes

in, reads in the temporary object slot into its return value, places its object down into the temporary object slot and releases both `_putPermission` and `_takePermission` (signalling to the first thread that it has placed its exchange object down). The first thread is unblocked and reads in the temporary object slot into its return value. Thus the exchange has occurred.

However, an exchanger must be locked down from *other* threads interacting with it as soon as *two* threads (i.e., the pair of threads exchanging data) have acquired it. This involves having a semaphore, called `_lockdown`, that is initialised with exactly two tokens. As soon as the first thread arrives, it will acquire one token, and wait for the second thread to exchange its data. When the second thread arrives, it will acquire the second token—thus locking down the Exchanger.

When a thread arrives and takes a token, it should always check to see if it is the first or second thread in the exchanger (using a `_threadCount` or similar). Should there be more than two in either case, an exception should be thrown—this boilerplates the exchanger since there should never be more than two threads within the `_lockdown` semaphore block.

Once the exchange has occurred between the pair of threads, both threads will release a token each for `_lockdown`; this will allow for the next pair to come and exchange (i.e., two threads waiting at the acquisition of `_lockdown` while the exchange was occurring can now continue).

2.9 FIFO Semaphore

A First-In-First-Out semaphore works in essentially the same way as a normal semaphore, however by using a .NET queue of Semaphores, it can be ensured that first threads in will also be the first threads out.

The main addition to the semaphore is a `_threadQueue` queue object. Whenever the semaphore is acquired, a new Semaphore is created and enqueued into the . This Semaphore will then be acquired whilst still in the `Acquire` method of the FIFO Semaphore, thereby blocking the thread at the Acquire method.

When a FIFO Semaphore is released, a token is given back (as per the base semaphore). the queue is dequeued to gain access to the enqueued (and still blocked) Semaphore from above. Once dequeued, the Release method needs to acquire permission to allow the dequeued semaphore to be released. Initially, the FIFO-Semaphore already has this permission; it acquires the `_allowNextReleasePermission` semaphore, and then proceeds to release the

dequeued Semaphore. This unblocks the Semaphore still waiting in Acquire, which then takes a token, and releases permission to `_allowNextReleasePermission` so that the next first in semaphore can be released in the release method. The use of this second semaphore ensures that tokens are taken by the respective threads that acquired the FIFO Semaphore in order.bject.

FIFO semaphores are strong semaphores—their improved liveness, when compared to standard semaphores, stems from a guarantee that any threads that acquire it will *always* be unblocked (eventually). While this is rare for normal semaphores, it is still plausible; consider a thread that always fails to acquire a released token when a semaphore is released since new threads come in an 'snatch' the released token. With a FIFO semaphore, all threads are guaranteed to acquire a released token; therefore it provides a fail-safe to liveness issues wherein a thread that is blocked from the FIFO semaphore is ensured that it will be unblocked (depending on how many other threads have acquired the semaphore beforehand) and thus providing a guarantee to uphold that thread's liveness.

2.10 Active Objects

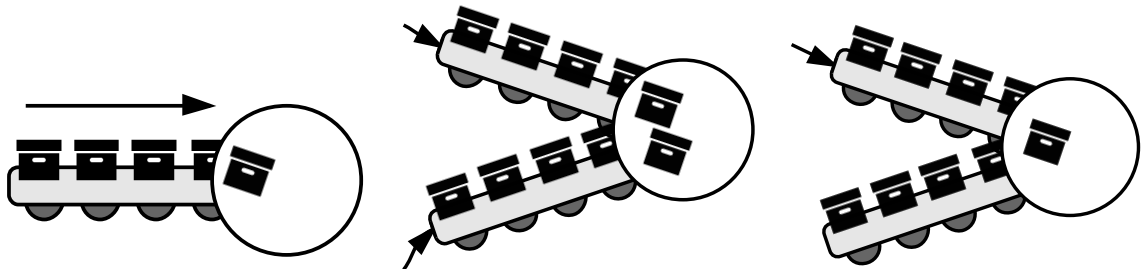
Active objects are an abstract class which are forever active (i.e., processing data in some way). Unlike passive, 'traditional' objects, that only process when they receive a message, active objects run on their own thread, are started once and once only (to activate that thread) and therein run a single task.

An active object's primary task is to receive data, process it in some way, and then send it off to another object. Typically, a concurrent program is made of a series of active objects each performing its own individual task; the data it receives is slightly modified or processed in such a way before it is handed over to the next active object. The active object never receives *direct* messages from other objects; instead it will only receive data from any other active object (it doesn't necessarily know the source) and use that data to perform work.

To hand data between one active object to the next, a channel (see Section 2.2) is used to link up one or more active objects with each other. There are typically at least two channels[4] for each active object—an incoming data channel and an outgoing data channel. The count and arrangement of these channels determines the behaviour of how the active object will interact with other active objects, while the arrangement of the linkages themselves deter-

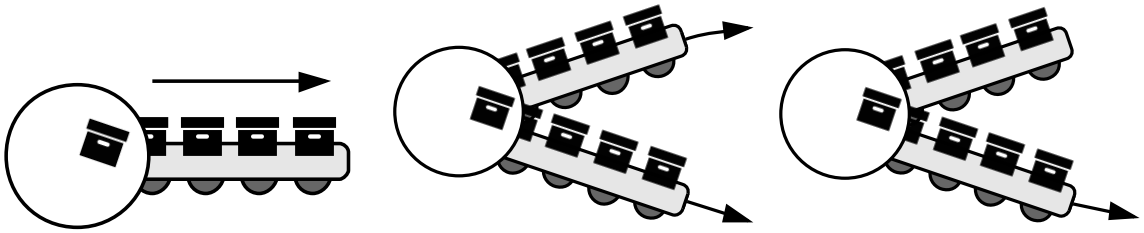
[4]There may be only one channel, in which case this object is an endpoint in the concurrent program.

mine how the active object achieves satisfactory liveness to reach the concurrent program's goal.



(a) Sink **(b)** Combiner **(c)** Selector

Figure 2.6: Different ways of arranging incoming channels between active objects.



(a) Source **(b)** Broadcaster **(c)** Router

Figure 2.7: Different ways of arranging outgoing channels between active objects.

Sink Sinks receive a single piece of data at a time, usually from a source.

Source Sources generate the work for other active objects by shooting single pieces of data off at a time.

Combiner Combiners process off multiple different channels at a time. A combiner doesn't care which channel it reads off, and treats data it reads off different channels as just one channel.

Broadcaster Broadcasters shoot data off to multiple different active objects. A broadcaster don't care which active objects it sends to—it an object is broadcast, it is sent off onto many different channels at once.

Selector A selector will selectively read off one of its incoming channels. An important factor with selectors is that they wait for data to be received from one channel exclusively, even if there is other data waiting from other incoming channels—this can cause liveness issues if programmers are not careful.

Router A router will selectively shoot data off one of its outgoing channels at a time. This means that only one of its receiving active objects will receive the same data, whilst the others do not receive it.

By using channels, in combination with one-way message passing, the threads in each active object are guaranteed not to mutate the same set of data; each unit of data is *handed over* to the next active object. The concept of one-way message passing is similar to normal object message passing, however, once the 'message' or data is sent to the next active object, there is no response from the receiving object—that is, sending data off an active object's outgoing channel is essentially sending the data through a black hole, never to be seen or processed by the same active object ever again. This 'black hole' principle is further reinforced by making the handed data a reference to `null` after the handing over is done; this is a protocol that all active object abide by to eliminate any safety issues that may arise in the program. Concurrency issues will arise if two threads work on the same data at the same time; therefore, to avoid this safety problem, once one active object is done working on its data, it shoots the data through this black hole—therefore the sending active object thread never touches that piece of data again, while the receiving active object will be able to safely work on it as soon as it receives it.

Thus, active objects have three key deciding factors that programmers face:

1. when should the active object take data off its receiving channel,
2. what should the active object do with that data, and
3. when should the active object let go of that data.

Typical examples of active objects include an listening for network traffic, writing to databases, printing to the console and so on. Each of these work on the same *kind* of data, do the *same task* with that data and has the capability of handing that data over to another active object.

References

A.B. Downey. *The Little Book of Semaphores*. 2005. URL `http://books.google.ee/books?id=pojKZwEACAAJ`.

Douglas Lea. *Concurrent programming in Java: design principles and patterns*. Addison-Wesley Professional, 2000.

Microsoft Corporation. Monitor Class, 2014. URL `http://msdn.microsoft.com/en-us/library/system.threading.monitor(v=vs.110).aspx`.