G52CON Revision Guide

Craig Knott

Version 1.0
May 16, 2013

Contents

1 What Is Concurrency?

First and foremost, what is concurrency and why do we use it? Concurrency is a property in which several computations execute simultaneously, often with the ability to interact with one another. It is useful to be able to do several things at once, specifically in circumstances where latency, parallelism and distribution are important. Concurrency can be implemented on single processors, which means while certain programs are waiting to continue, other programs can use the processor. An example of the use of a concurrency can be seen in many file downloading applications. Without concurrency, is it difficult to interact with the client (it would be to busy downloading the file to allow you to interact with it, for instance to cancel the download), and the server would only be able to handle a single download at any one time (each subsequent download would be required to wait for the previous one to finish). However, concurrent implementations would allow for this client interaction, and for servers to handle multiple clients. We use concurrency when we multiple actors are accessing a shared resource without any synchronisations, and when at least one thread modifies data that others are reading. In summary, concurrent programs are ones which have multiple actors to manipulate shared resources, our job is to ensure the correct results are in place to make this interaction go without complication.

All programs are a single thread that executes a sequence of instructions in a given order, this is known as a *sequential program*, or an "Ordinary Programm". Concurrent programs extend this to be able to handle multiple processes (threads of controls), each of which is a sequential program. These are executed on *Multiprocess systems*, or *Parallel Programs*, that execute many sequential or concurrent processes in parallel. It's also worth noting there are *distributed systems* that execute processing on interconnected systems with no shared memory.

The problem with programming at the moment is that faster processors are more difficult to produce. To solve this, we use processors that have multiple *cores* on them. Research is ongoing as to how we can use these multicore systems to efficiently speed up computing, this is known as the *parallel challenge.*

There are two main implementations of concurrency; shared memory, where execution of concurrent processes are run on one or more processors with some access to shared memory in which communication takes place by the reading and writing of this shared memory; and distributed processing, where concurrent processes are run on separate processes and communication takes place in the form of message passing.

We can further distinguish between: multiple programming, where execution takes place by time sharing on a single processor (simulated concurrency); and multiprocessing, where execution is achieved by running processes on separate processors, each with access to a shared memory (true parallelism).

Concurrent programs are naturally more complex than single-threaded programs, for a variety of reasons: When more than one activity can occur at a time, execution becomes non-deterministic; code may execute in surprising orders, only those explicitly ruled out are not allowed; and a field set to one value in one line of code may have a different value before the next line of code in the same process is executed. We are required to learn new programming techniques to effectively write concurrent programs.

1.1 Race Conditions

Race conditions are when two threads are accessing the same data without any rules for its access; and in particular when the output is dependant of the sequence or timing of these accesses. Programs with these race conditions in them are called *indeterminate*, meaning their outcome is unpredictable, due to the different possible ways in which the sequences of instructions can be interleaved - this makes them extremely difficult to debug. In contrast, a deterministic program is one where the results will be the same each time it is run.

So, how do we handle these race conditions? We need some mechanism for controlling access to shared resources when dealing with concurrecny, the aim of this is to reintroduce the determinism in programming. For this, we need some form of synchronisation; which is implemented through use of critical sections. These are sections of code that access shared resources, and are required to use mutual exclusion to synchronise their execution. Any time in which a program reads, modifies, and writes a shared variable, it is important to have this as a critical section, so no interrupts occur during this process, preventing any undesired results.

1.2 The role of the Operating System

There are two main functions of the operating system when considering concurrent programs: as an extended machine, providing a clean abstract set of resources instead of complex details of hardware implementations; and as a resource manager, allowing multiple programs to run at the same time and the manage and protect the various resources of the system. Processes are described to the operating system by a data structure called a process descriptor. This includes private data of the process, the actual code, and the state of registers and the program counter. These processes themselves can be in three possible states, as explained in figure 1.

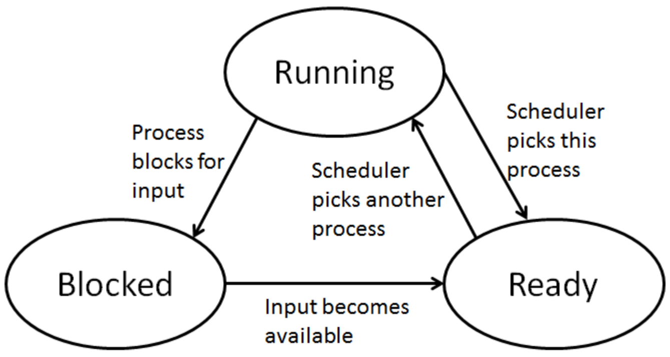

Figure 1: The states of a process

The *running* state is when a process is using the CPU (only one can do this at any one time), *ready* is when the process is runnable, but temporarily stopped to let another process run, and *blocked* is when a process is unable to run until some external event takes place. It is important to remember the difference between a process and a program. A process is a dynamic entity representing actual computations, and a program is a static entity representing potential computations.

1.3 Scheduling

When two or more processes are simultaneously in the read state, and are then competing for the CPU, a choice has to be made as to which process to run next. This is known as a multiprogramming system. Choosing which process is next is done through a scheduler, and its scheduling algorithm. There are four situations in which scheduling should take place:

- When a new process is created
- When a process exists
- When a processes blocks (due to I/O)
- When an I/O interruption occurs

Further to this, there are two different classifications of scheduling algorithm:

Pre-Emptive
When the process is run for up to some length of time, and is suspended at the end of this time if it is not already finished

Non Pre-Emptive
When the process runs until it is blocked

There are several different types of systems that each require different things from the scheduling algorithm they use, along with some properties that all systems require:

- **All Systems**
  Each process much be given a fair share of the CPU, the stated policy must be carried out, and all parts of the system must be kept busy. These are known as *fairness*, *policy enforcement* and *balance* respectively.
- **Batch Systems**
  Jobs are bundled together with the instructions necessary to allow them to be processed without intervention. This system must maximise jobs per hour (throughput), minimise time between submission and termination (turnaround time), and keep the CPU busy at all times (CPU Utilisation).
- **Interactive Systems**
  Modern PCs, multimedia systems and server that have interactive users and need to respond to events well. These systems must respond quickly to requests (response time), and meet the users' expectations (proportionality).
- **Real-Time Systems**
  External physical devices generate stimuli, to which the computer must react appropriately in a very short amount time. These systems must avoid losing data (meet deadlines), and avoid quality degradation in multimedia systems (predictability).

1.4 Scheduling Algorithms

1.4.1 Interactive Systems

These systems have impatient users, that need fast responses. This is why some level of pre-emption is required. Some algorithms are discussed below:

Round-Robin Scheduling
In this algorithm, each process is assigned a time interval, in which time it is allowed to run. The scheduler must maintain a list of runnable processes, and when a process uses up its *quantum* (its quota of runnable time), it is put at the end of this list. This means each algorithm is run in turn, and the processing can continue until the list is empty.

Priority Scheduling
Each process is assigned a priority, and the process with the highest priority is the one which runs. After each running, the priorities are changed. It can be useful to group multiple processes together, each with the same priority, and then execute these processes using the Round-Robin algorithm discussed above.

There are many other algorithms for interactive scheduling, including: multiple queues, shortest process next, guaranteed scheduling, lottery scheduling, and fair-share scheduling.

1.4.2 Batch System Scheduling

In these systems, there are no impatiently waiting uses, so non-pre-emptive algorithms are acceptable.

First-come first-served
This is by far the simplest scheduling algorithm, processes are assigned the to the CPU in the order they request it, with no interruption. This means there is a single queue of ready processes, and as jobs come in, they are put to the end of this queue. When the currently running process blocks, the first process on the queue is then run. This is a non-pre-emptive algorithm.

Shortest Job First
Another non-pre-emptive algorithm, which assumes that run times of the processes are known in advance and is optimal when all jobs are available simultaneously. It determines the optimal running time by using the average of the cumulative sum of the jobs to be processed. For instance, assume we have 4 jobs: B C, and D, each with a run time of 4 minutes, and A with a run time of 8 minutes. Two different approaches of executing these algorithms is shown below.

Original Order

$$\begin{aligned}
& \{A, B, C, D\} \\
= \quad & \{ \quad \text{Substitute Values} \quad \} \\
& \{8, 4, 4, 4\} \\
= \quad & \{ \quad \text{Apply Cumulative Sums} \quad \} \\
& \{8, 12, 16, 20\} \\
= \quad & \{ \quad \text{Take Average} \quad \} \\
& 14
\end{aligned}$$

Shortest First

$$\begin{array}{ll}
 & \{B, C, D, A\} \\
= & \{\ \text{Substitute Values}\ \} \\
 & \{4, 4, 4, 8\} \\
= & \{\ \text{Apply Cumulative Sums}\ \} \\
 & \{4, 8, 12, 20\} \\
= & \{\ \text{Take Average}\ \} \\
 & 11
\end{array}$$

As you can see from the above, the second ordering has a lower average cumulative sum time, and this is the optimal way to schedule these jobs (using this algorithm)

1.5 Atomic Instructions

An *atomic action* is one that appears to take place as a single *indivisible* operation. During an atomic action, a process switch cannot happen, meaning no other actions can be interleaved with an atomic action and no other process can interfere with the manipulation of data by an atomic action. When a processor is running a program, process switches can only occur between the end of one atomic action and the beginning of another. The majority of machine instructions of atomic, and these are known as fine-grained atomic actions. They are actions that can be implemented directly as uninterruptible machine instructions (for instance, loading and storing in registers). Some sequence of machine instructions can also be atomic, these are known as coarse-grained atomic actions, which are sequences of fine grained atomic actions that cannot be interrupted.

Like mentioned, memory access is atomic, meaning reading and writing a single memory location is a fine-grained atomic operation (known as memory interlock). However, it's important to remember that access to non-basic types (like strings and doubles) are often not atomic, and items that are packed two or more to a ward (like strings and bitvectors) cannot be assured to be atomic. Few programming language actually specify anything about the indivisibility of variable accesses, leaving this as an implementation issue.

In addition to memory interlock, most modern CPUs provide other special invisible instructions:

- Exchange
  The swapping of a register and a variable.

  ```
  x <–> r
  ```

- Increment/Decrement
  Increasing or decreasing a variable.

  ```
  public int INC (int x){
          int v = x;
          x = x + 1;
          return v;
  }
  ```

- Test and Set
  Write to a memory location and return its old value.

  ```
  public boolean TS (boolean v){
          boolean x = v;
          v = true;
          return v;
  }
  ```

- Compare and Swap
  Compare the contents of a memory location to a given value and, if they are the same, modify the contents of that memory location to a new, given, value.

  ```
  public boolean CAS (int x, value v, value n){
          if(x == v){
                  x = n;
                  return true;
          } else {
                  return false;
          }
  }
  ```

- Load-Link/Store-Conditional
  A pair of instructions, in which Load-Link returns the current value of a memory location, and the store-conditional will store a new value in that location only if no updates have occurred to the location since the load-link.

  ```
  value v = LL(int x);

  public boolean SC(int x, value v, value n){
          if (x == v){
                  x = n;
                  return true;
          } else {
                  return false;
          }
  }
  ```

These special machine instructions can be used to solve some very simple mutual exclusion problems, such as the single word readers and writers problem, and the shared counters problem. The single word reader and writer problem is where several processes read a shared variable and several processes write to same variable - but no process can do both. This can be solved by storing the variable in a single word, which will mean the memory unit will ensure mutual exclusion for all accesses to the variable. The shared counter problem is where several processes each increment a shared counter, and this can be solved by also storing the variable in a single word, and only allowing a special atomic increment instruction increase the value of it, combined with the mutually exclusive reading of single memory location).

In multiprocessing implementations, the set of atomic instruction is (understandably) different, and those instructions that are atomic on a single processor do not provide mutual exclusion between multiple processors. In a multiprocessor machine, the memory lock instruction must have an extra constraint on it: no other processors are permitted access to the shared memory during the execution of the instruction following the memory lock instruction. This means memory locked instructions are effectively indivisible and therefore mutually exclusive across all processors. However, memory lock instructions have to be used with care, to avoid locking other processors out for unacceptable amounts of time.

1.6 Fine-grained and course-grained instructions

As mentioned above, fine-grained instructions are single indivisible instructions, and coarse-grained instructions are multiple fine-grained instructions that have the indivisibility property for the entire group. The problem with fine-grained atomic actions is that they are not very useful for the applications programmer: they don't work for multiprocessor implementation unless we can lock memory, the set of atomic actions varies from machine to machine, and we cannot assume that a compiler with generate a particular sequence of machine instructions from a given high-level statement. To write critical section of more than a single machine instruction, we need some way of concatenating fine-grained atomic actions, and this is was coarse-grained atomic actions are for. These can be implemented at the hardware level by disabling interrupts or by defined a mutual exclusion protocol.

1.6.1 Disabling Interrupts

We can ensure mutual exclusion between critical sections in a multiprogramming implementation by disabling interrupts in a critical section. However this has several disadvantages: it is available only in privileged mode, it excludes all other processes (reducing concurrency), and it doesn't work in multiprocessing implementations (disabling interrupts is local to one processor). There are, however, some specific occasions where it is useful to disable interrupts, these are: when writing operating systems, for dedicated systems or bare machines such as embedded systems, and simple processors which don't provide support for multi-user systems. However, it is still not a very useful approach for the application programmer.

1.6.2 Defining a Mutual Exclusion Protocol

A much better way of solving the mutual exclusion problem is to adopt a standard Computer Science approach, and design a protocol for it. The protocol will have a sequence of instructions that are executed before and possible after the critical section, utilising the memory interlock property. Before we continue with this, it's important to see why we need the protocol. Imagine you have the following two segments of code, from two process to run concurrently, in which y and z are initialised to 0 are shared between the two:

Process 1

```
// some code

int x;
x = y + z;

// some other code
```

Process 2

```
// some code

y = 1;
x = 2;

// some other code
```

These two processes could result in one of three possible interleaving sequences:

```
x = y + z;          y = 1;              y = 1;
y = 1;              x = y + z;          z = 2;
z = 2;              z = 2;              x = y + z;
```

In which the value of x can be either 0, 1, or 3 (respectively), dependent on which sequence is generated.

1.7 Concurrent Abstractions

To help us understand and think about conrurent programming, it is useful to use abstractions, as with most things in Computer Science. As a reminder, an abstraction is a method to generalise conclusions, and in programming, abstractions are achieved by encapsulation (dividing software into public specification and private implementation). The Concurrent Programming Abstraction is needed to make it possible to reason about the interaction between programs and their environment. The challenge with this, however, is to identify what the essential aspects of project interaction are, and what aspects are non-essential.

The Atomic Instruction is the basic building block of the concurrent programming abstraction. This is provided at the hardware implementation and operating system levels, and at the very least, every computer will provide some form of memory interlock.

Concurrent programs must have processes that can cooperate and communicate with one another. This can be achieved using shared variables, or message passing (depending on what form of concurrent implementation is being used). The biggest problem with concurrent programming is synchronising these communications, and the solutions to this are Mutual Exclusion (ensuring statements in different processes cannot execute at the same time) and Condition Synchronisation (delaying a process until some condition is true, usually one processes waits for an event signalled by another process). An example of the important of synchronisation is summarised in the shared memory producer-consumer problem. In this problem, one process (the producer) writes data to a single memory location, and the other process (the consumer) reads data from this memory location. Problems can arise when either the consumer attempts to consume nothing, or when the producer overwrites the memory location.

1.7.1 Interleaving

Any concurrent program can be represented as an interleaving sequence of atomic instructions (that are ordered randomly, unless there is a specific action enforcing a specific ordering). The execution time of atomic instructions is ignore, and thus we achieve hardware independence; only the order of execution is considered. Unfortunately, this means there are many possible interleaving sequences that we must consider. If, at any point, the execution time of a process is sufficiently important, this can be represented as one of the processes in the program with a high priority.

The main advantage of using the interleaved sequence approach is that we can consider the individual processes as executing independently, unless the following occurs: contention (processes compete for resources), or synchronisation/communication (processes exchange data, or negotiate order of execution). The real challenge is how to ensure the correct execution of all interleaving sequence for a given program. As mentioned earlier, the number of interleaving sequences can potentially be very large, the formula of which is $\binom{2n}{n}$, where n is the number of instructions for a process.

$$
\begin{aligned}
& 5\ \textit{Instructions} \\
= & \quad \{ \quad \text{Construct Choose Function} \quad \} \\
& \binom{10}{5} \\
= & \quad \{ \quad \text{Convert to Fraction} \quad \} \\
& \frac{10!}{5! \times 5!} \\
= & \quad \{ \quad \text{Apply factorial} \quad \} \\
& \frac{3,628,800}{120 \times 120} \\
= & \quad \{ \quad \text{Apply Multiplication} \quad \} \\
& \frac{3,628,800}{14,400} \\
= & \quad \{ \quad \text{Apply Division} \quad \} \\
& 252
\end{aligned}
$$

As you can imagine, this number rapidly increasing (a program with 40 instructions, for instance, has 1.07×10^{23} instructions).

If instructions from different processes are arbitrarily interleaved, problems can occur. Any interleaving that is not explicitly prohibited is allowed. Inevitably, some interleavings has undesirable results, like interference that can occur when two processes read and write shared variables in an unpredictable order, and produce unpredictable results.

Ornamental Gardens Problem
For instance, imagine the ornamental gardens problems. In which a stadium is monitoring the entering of customers into their ornamental gardens from two different turnstiles, that both update the same variable. The code for which is:

```
// West Turnstile                    // East Turnstile

init1;                               init2;
while(true){                         while(true){
        // Wait for turnstile                // Wait for turnstile
        count = count + 1;                   count = count + 1;
        // Other code                        // Other code
}                                    }
```

Where count is initialised to 0.

Now imagine if 10 people had been to the garden already, and as such the value of count would be 10.

West Turnstile	**East Turnstile**
count++;	count++;
1. *count* is loaded into a register ($r == 10$)	
	2. *count* is loaded into register ($r == 10$)
	3. Register is incremented ($r == 11$)
4. Register is incremented ($r == 11$)	
	5. Register value is stored into *count* ($count == 11$)
6. Register value is stored into *count* ($count == 11$)	

The problem with this scenario is that two people entered the garden, but only one was recorded; the other was lost. This is known as interference (see figure 2)

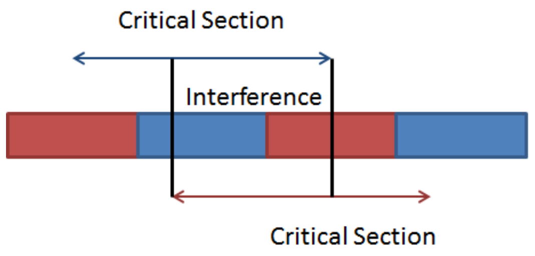

Figure 2: A diagram showing two processes interfering with one another

To avoid interference, we need to ensure that no two processes access a shared variable at the same time. we do this by marking such section of code as *critical*, and require that no two processes are executing critical code at the same time, this is known as *mutual exclusion*. This ensures the correct behaviour of the program, and states that once one process enters a critical section, no other process may enter a critical section until this first process has exited its critical section. Bare in mind that mutual exclusion is a constraint on the execution of processes which applies between the *critical sections*, and not the processes themselves; two processes are free to be interleaved even if they have critical sections, it is only their critical sections that should never overlap. Further to this, the critical section of one process *can* be interleaved with the *non* critical section of another.

Sometimes to ensure mutual exclusion, one (or more) process may have to wait to enter their critical section. If one process is in their critical section when a second attempts to enter its own, the second will wait. This prevents the interleaving of critical sections, and while this sounds like the execution time is increased, that is not true; the same instructions are executed, just in a different order.

In order to reduce unnecessary delays, there are different *classes* of critical sections. Not all critical sections need to be mutually exclusive of each other, as they could both be doing something that has no effect on the other. A class of critical sections is a set of critical sections, all of which must be mutually exclusive with only others in the same class. Further to this, critical sections in different *classes* need not be mutually exclusive.

There are four main methods of implementing critical sections, which will be covered in this guide

1. Locks
   Primitive, Minimal Semantics

2. Semaphores (covered later)
   Basic, easy to understand, hard to program with

3. Monitors (covered later)
   High level, requires support from the language, implicit operations, easy to program with

4. Messages and Message passing (covered later)
   Simple model of communication based on atomic transfer of data across a shared channel.

The general outline for a mutual exclusion protocol is as follows:

```
Non-Critical Section();
Pre-Protocol();
Critical-Sections();
Post-Protocol();
Non-Critical Section();
```

The one constraint of this mutual exclusion template is that the only segment that can be executed over infinite time are the non-critical sections, all other sections must execute in finite time.

1.8 Locks

Locks have two states, *held* and *not held*; and the meaning of these is intuitively simple. A lock is held when a thread is in the critical section, and not held when on threads are in the critical section. There are also two operations that a lock needs to have, *acquire* which marks a lock as held when it is not held, or if it is already held, will wait until it is no longer held; and release, which is used to mark a lock at not held.

When multiple threads call the acquire operation, one will succeed and can operate its critical section. All the other threads will wait until the first calls the release operation. At this point, one of the other threads may lock and run its critical section, and this process is repeated until all locks are resolved. They are declared just like variables, and are used to surround a critical section (must like the P and V operations of semaphores, discussed later).

Acquiring a lock only blocks thread trying to acquire the same lock. This means we can have different threads in different critical sections at the same time. If a piece of data is to be accessed by multiple threads, they must all have the same lock.

1.9 Deadlocks, Livelocks and Starvation

These three policies each cause some negative impact on the system, and it is important to eradicate them when writing a concurrent system. The effect of each one is:

- Starvation
  A policy that can leave some thread executing in some situation; causing one or more threads to wait. Eventually, starvation *can* end.

- Livelock
  A policy that makes some threads do some things endlessly, without making any progress. In this case, no threads make any progress

- Deadlock
  A policy that leaves all threads stuck; no thread can do anything at all, meaning they all wait forever. This can happen if everything thread in a set of threads is waiting for an event, but this event can only be caused by on one of the threads in the set. Note that if deadlock occurs, this implicitly means starvation is also occurring; but deadlock will never end (without external input).

2 Proving Concurrent Correctness

Proving the correctness of our programs is important so we can be assured they are safe to run. For regular programs, there are two broad categories of correctness:

- **Partial Correctness**
  IF the conditions on the input hold AND the program terminates
  THEN it delivers the correct output
- **Total Correctness**
  IF the conditions on the input hold
  THEN the program terminates AND it delivers the correct output

Unfortunately, these methods of evaluating correctness aren't as useful when we consider concurrent programs. With these rules, non-terminating programs would always be partially correct, which is a problem considering how many concurrent programs are run as an infinite loop.

Instead of the two aforementioned correctness methods, we must abide by four principles, split into two categories, these are:

- **Safety**
  These properties must always hold
  - Mutual Exclusion must be abided by
    At most one process at a a time is executing its critical section
  - There must be no deadlock or livelock (there must be progress)
    If no process is in its critical section, and two or more attempt to enter their critical sections, atleast one will succeed. In addition to this, any thread outside a critical section cannot stop another from entering its respective critical section. In a deadlock, the process is blocked and waiting on some condition (and is not using CPU time), and in live lock, the process is still alive, and waiting on some condition (actively using CPU time).
- **Liveness**
  These properties must eventually hold
  - No individual starvation (Eventual Entry)
    If a process is attempting to enter its critical section, it will eventually succeed.
  - Fairness to all processes must be granted (Absence of unnecessary delay)
    Processes in their non-critical sections will not prevents other attempting to enter their respectively critical section from doing so.

Before we move on, there are a few more scheduling terms that you should be familiar with (also remember that scheduling is the order that the scheduler puts processes in).

- **Weak Fairness**
  If a process makes a maintains (continuously) a request, it will granted. This guarantees that if a condition becomes true and stay true forever, the action will eventually be executed.

- **Strong Fairness**
  If a process make infinitely often a request, it will granted. Guarantees that if the conditions becomes true infinitely often, then the action will eventually be executed. This means the condition may alternative between true and false, thus providing a scope for choice.

- **Linear Waiting**
  If a process makes a request, it will be granted before any other process is granted a request more than once.

- **FIFO (First In, First Out**
  If a process makes a request, it will be granted in the order of received requests.

In the following sections, we will go through how we can prove or disprove the relating properties. We will do this (in general), through the process of induction, which can be stated as the following:
$P(0)$ - Base Case (The beginning of the programs execution)
$P(n)$ - Assumed True
$P(n+1)$ - To be proven

We must prove (in general), that from a given state, the next state will not produce any valid transitions that would disprove the property we are attempting to prove.

Finally, the last piece of knowledge you should have is the meaning of the following two symbols that will be used in the following sections:

$\Box P$ - P Will *always* be true

$\Diamond P$ - P Will *eventually* be true

2.1 Mutual Exclusion

The first property to prove is mutual exclusion. Note that, while I have listed these properties in this order, it does not matter which order you prove them in; it is only important that any disproved properties invalidate the correctness of the algorithm at hand. In this property, we are looking at whether or not two critical sections can be accessed at the same time. This is defined at the following property (for a two process concurrent program):

$$\neg(at(Crit1) \wedge at(Crit2))$$

Where *Crit1* and *Crit2* would be the corresponding line number to the critical section for the respective method.

The best way to see how this principle works would be to actually go through and example, which can be found below.
Firstly, our algorithm in question is displayed below, it is known as a *simple spin-lock* that uses *turns* (Note that the algorithms have taken the ADA syntax and made it more friendly):

Process 1

```
loop {
a1      Non-Critical();
b1      loop { } Until (Turn = 1);
c1      Critical();
d1      Turn = 2;
} end loop
```

Process 2

```
loop {
a2      Non-Critical();
b2      loop { } Until (Turn = 2);
c2      Critical();
d2      Turn = 1;
} end loop
```

Firstly, we must identify the property we are trying to prove, which is that mutual exclusion holds. This is formally specified (for this algorithm) as:

$$\neg(at(c1) \wedge at(c2))$$

Next, we look at the $P(0)$ property. At, $P(0)$ (the beginning of the processes execution), the processes are at $a1$ and $a2$, respectively. This is a valid move, and doesn't break the property of mutual exclusion, and such the base case is true.

Next up, our inductive case. We must take a state in the process execution, and then prove that the next state would not cause invalidation. For this, we draw up four columns.

Process 1	Process 2	Process 1	Process 2

These columns represent the state of each of the processes, as they are run concurrently, and allow us to look at each algorithm individually. We now have to determine the position of the algorithm we are not analysing, and the variables that this algorithm would have set. As we are looking at mutual exclusion, it is only useful to analyse when the unobserved algorithm is in its critical section, and then test to see if the observed algorithm hits their critical section during this time. For this reason, we state that the unobserved algorithms are in their respectively critical sections (*c1* and *c2*). At this point, this revision guide differs from Andrzejs', as he list Process 1 as the unobserved process, and Process 2 as the observed process, and I have used Process 1 to represent the first process, and Process 2 to represent the second process.

Process 1	Process 2	Process 1	Process 2
at(c1)			at(c2)

Now we must specify the values of the system variables when the processes are in their critical sections. In our algorithm, the spin locks will release Process 1 when *turn = 1*, and Process 2 is released (and can enter its critical section) when *turn = 2*, these are added to the table.

Process 1	Process 2	Process 1	Process 2
at(c1)			at(c2)
Turn = 1			Turn = 2

We have now specified the unobserved algorithm, and must move onto the observed algorithm. For this, we must list every state that the observed algorithm can be in. This is where things become slightly more confusing. Intuitively, you would just list every line - this is *incorrect.* We do not include the critical section of Process 2, as we are attempting to prove whether or not Process 2 will enter this state, and not whether it starts in it. The second thing to note is that a loop or conditional can go to one of two places. It can either exit and progress to the next line, or continue running, and return to the first line in the loop. For this reason, any loops are listed twice, as follows:

Process 1	Process 2	Process 1	Process 2
at(c1)	a2	a1	at(c2)
Turn = 1	b2	b1	Turn = 2
	b2	b1	
	d2	d1	

Now, for each state in the observed algorithm, we must work out its next state. This is generally easy for the basic states, but loops and conditionals require a little more thought. However, it is generally the case that the end of the loop will go to the start of the loop or the line after this loop. In our case, we have a single line loop, which means it goes to itself, or the next line (this is also why the last line of the algorithm loops to the beginning line).

Process 1	Process 2	Process 1	Process 2
at(c1)	a2 → b2	a1 → b1	at(c2)
Turn = 1	b2 → b2	b1 → b1	Turn = 2
	b2 → c2	b1 → c1	
	d2 → a2	d1 → a1	

Now that we have a list of all the possible transitions that can be made, we can start evaluating whether they disprove the mutual exclusion property. The only moves in a mutual exclusion proof, that have the potential to disprove the property are those that transition into a critical section. In our case, this is b1 → c1, and b2 → c2. We highlight these in some way (I have made them red).

Process 1	Process 2	Process 1	Process 2
at(c1)	a2 → b2	a1 → b1	at(c2)
Turn = 1	b2 → b2	b1 → b1	Turn = 2
	b2 → c2	b1 → c1	
	d2 → a2	d1 → a1	

At these points, both of the programs would be in a critical section at the same time, and thus disproving the property. But, before we can say this for certain, we must ascertain as to whether or not the the moves made are *valid* - in terms of the variables that the unobserved process has set. For instance, our transition of b1 → c1 is not a valid move, as the variable *Turn* is set to the value of 2, and the spin lock is not released until *Turn* is equal to 1. We mark the valid and invalid moves.

Process 1	Process 2	Process 1	Process 2
at(c1)	a2 → b2 ✔	a1 → b1 ✔	at(c2)
Turn = 1	b2 → b2 ✔	b1 → b1 ✔	Turn = 2
	b2 → c2 ×	b1 → c1 ×	
	d2 → a2 ✔	d1 → a1 ✔	

Only now can we say whether or not our property holds. We say that, if there are any valid moves, that would result in the property being broken (any bold move that has a ✔), the property is not held. However, if all the moves that would break the property are invalid, or if there aren't any, the property holds. In this case, our property holds, as our two breaking cases are both invalid.
This process can be summarised to the following steps:

1. Determine the property to prove
2. Determine is the base case in invalid
3. Draw out the 4-Column table to represent the processes
4. Determine the position of the unobserved process, and the value of all variables in the system at that point
5. List all states the observed process may be in
6. List the transitions that these states can make
7. Mark all moves that may break the property at hand
8. Mark all moves that are invalid due to system variables
9. If any moves that break the property are valid, the property is invalid

2.2 Absence of Deadlock/Live lock

We can now prove the remaining properties of concurrent programs either true or false, using the same method as we did for mutual exclusion; with the single substitution of the property to prove. The next one we will look at is the absence of deadlocks or live locks. This property can be generalised to:

$$\neg(\Box\ at(\textit{Some Line Loop}) \wedge \Box\ at(\textit{Some Line Loop}))\ \neg(\Box\ at(\textit{Some Lines}) \wedge \Box\ at(\textit{Some Lines}))$$

Where deadlocks are caused when programs are stuck at one specific line (generally a single line loop), and cannot progress - but this one line is not using CPU time. Where as live locks are where the program is actively doing some computation, but cannot progress (like a multiline loop). In our example, we do not have a multiline loop, and as such the property of absence of live lock is maintained. We do however, have a single line loop, which could cause a deadlock. This would be specified as such:

$$\neg(\Box\ at(b1) \wedge \Box\ at(b2))$$

We can now use the formal process defined above to prove that this is an invalid; but it is sometimes useful to think more sensibly, and see that certain properties simply cannot be true. For instance, in our algorithm, for both loops to be running at the same time, *Turn* would need to have the value of 1 and 2 at the same, which is impossible - proving this property to be trivially true. Note that, when we only have a single line loop, we merge the properties of live and dead lock into one proof.

Below we specify a second algorithm, which does have a multiline loop in it, to show the process of proving the live lock absence property.

```
loop {                                    loop {
a1      Non-Critical();                   a2      Non-Critical();
b1      T1 = 0;                           b2      T2 = 0;
c1      loop until T2 = 1 {               c2      loop until T1 = 1 {
d1              T1 = 1;                   d2              T2 = 1;
e1              T1 = 0;                   e2              T2 = 0;
        }                                         }
f1      Critical();                       f2      Critical();
g1      T1 = 1;                           g2      T2 = 1;
} end loop                                } end loop
```

If we now go through our formal process, we can prove our live lock property either true or false. Firstly, we define the property to prove:

$$\neg(\Box\ at(c1, d1, e1) \wedge \Box\ at(c2, d2, e2))$$

We then look at the base case of $P(0)$. At the first step of execution, our processes are at $a1$ and $a2$, and not stuck in our loops - proving the base case true. Now we may draw our column chart.

Process 1	Process 2	Process 1	Process 2

Now we add the variables and position of the unobserved process into this table. Baring in mind that the variables will be different depending on the

Process 1	Process 2	Process 1	Process 2
at(c1)			at(c2)
T2 = 0			T1 = 0
T1 = 0			T2 = 0
at(d1)			at(d1)
T1 = 0			T2 = 0
at(e1)			at(e1)
T1 = 1			T2 = 1

We now list all of the states that the observed process can be in. Excluding any states that we are at in the unobserved process.

Process 1	Process 2	Process 1	Process 2
at(c1)	a2	a1	at(c2)
T2 = 0	b2	b1	T1 = 0
T1 = 0	f2	f1	T2 = 0
at(d1)	g2	g1	at(d1)
T1 = 0			T2 = 0
at(e1)			at(e1)
T1 = 1			T2 = 1

Followed by their next state.

Process 1	Process 2	Process 1	Process 2
at(c1)	a2 → b2	a1 → b1	at(c2)
T2 = 0	b2 → c2	b1 → c1	T1 = 0
T1 = 0	f2 → g2	f1 → g1	T2 = 0
at(d1)	g2 → a2	g1 → a1	at(d1)
T1 = 0			T2 = 0
at(e1)			at(e1)
T1 = 1			T2 = 1

This is followed by marking any states that could break the property (those that enter the loop).

Process 1	Process 2	Process 1	Process 2
at(c1)	a2 → b2	a1 → b1	at(c2)
T2 = 0	b2 → c2	b1 → c1	T1 = 0
T1 = 0	f2 → g2	f1 → g1	T2 = 0
at(d1)	g2 → a2	g1 → a1	at(d1)
T1 = 0			T2 = 0
at(e1)			at(e1)
T1 = 1			T2 = 1

And finish by marking all valid moves.

Process 1	Process 2	Process 1	Process 2
at(c1)	a2 → b2 ✔	a1 → b1 ✔	at(c2)
T2 = 0	b2 → c2 ✔	b1 → c1 ✔	T1 = 0
T1 = 0	f2 → g2 ✔	f1 → g1 ✔	T2 = 0
at(d1)	g2 → a2 ✔	g1 → a1 ✔	at(d1)
T1 = 0			T2 = 0
at(e1)			at(e1)
T1 = 1			T2 = 1

What's this!? We have a red line ticked? This means that a transition that breaks our property is valid; making the property invalid. At this point, the entire concurrent correctness of the program is negated.

2.3 Eventual Entry

The last two properties are much easier to prove using common sense. The first of these is eventual entry. This property states that if a process has a critical section, it will eventually execute it. It has the additional constraint that the second process is in contention for this critical section. It is a good idea to prove this for all cases, for instance using both values of *Turn*. For our first algorithm, we can say the property is true because of the following statements.

To Prove: $\Diamond\ at(c1)$

In the instance where $Turn = 1$

$$\begin{aligned} & at(b1) \wedge Turn = 1 \\ = \quad & \{ \quad \text{Progression of Algorithm} \quad \} \\ & at(c1) \end{aligned}$$

In the instance where $Turn = 2$

$$\begin{aligned} & at(b1) \wedge Turn = 2 \\ = \quad & \{ \quad \text{Implication that Turn is equal to 2} \quad \} \\ & Turn = 2 \\ = \quad & \{ \quad \text{Second process will eventually end loop, and change value of Turn} \quad \} \\ & \Diamond\ Turn = 1 \\ = \quad & \{ \quad \text{First process will use new Turn value to end loop} \quad \} \\ & \Diamond\ at(c1) \end{aligned}$$

2.4 Absence of Unnecessary Delay

The final property to prove for a program to be concurrently correct is the absence of unnecessary delay. This means that when the unobserved process is not in contention, or is terminated, the observed process will eventually enter the critical section. This basically means we must observe the variables of the second process as they would be after reaching the final line of execution. For instance, in our second algorithm expressed above, at the end of execution, *T2* would be 1, and *T1* would be 1. We now looked at Process 1, and whether or not these variables would affect whether or not it could reach its critical state. Which it does, proving this property true.

$$\begin{aligned} & at(b1), T1 = 1, T2 = 1 \\ = \quad & \{ \quad \text{Algorithm Progress} \quad \} \\ & at(c1), T1 = 1, T2 = 1 \\ = \quad & \{ \quad \text{Algorithm Progress (loop skipped)} \quad \} \\ & at(f1) \end{aligned}$$

Now that we know how to prove the correctness of algorithms, and what these properties mean, we will look at some algorithms that are used to implement concurrent correctness.

3 Concurrent Algorithms and their Properties

Some of the most well known and well covered concurrent algorithms are listed in this section, it would be wise to learn them.

3.1 Dekker's Algorithm

```
Integer T1 = 1 (of 0 and 1)
Integer T2 = 1 (of 0 and 1)
Integer Turn = 1 (of 1 and 2)


loop {                                  loop {
  Non-Critical();                         Non-Critical();
  T1 = 0;                                 T2 = 0;
  loop until T2 = 1 {                     loop until T1 = 1 {
    if ( Turn == 2 ){                       if ( Turn == 1 ){
      T1 = 1;                                 T2 = 1;
      loop until Turn = 1 { }                 loop until Turn = 2 { }
      T1 = 0;                                 T2 = 0;
    }                                       }
  }                                       }
  Critical();                             Critical();
  T1 = 1;                                 T2 = 1;
  Turn = 2;                               Turn = 1;
} end loop                              } end loop
```

3.2 Peterson's Algorithm

```
Integer T1 = 1 (of 0 and 1)
Integer T2 = 1 (of 0 and 1)
Integer Turn = 1 (of 1 and 2)


loop {                                  loop {
  Non-Critical();                         Non-Critical();
  T1 = 0;                                 T2 = 0;
  Turn = 2;                               Turn = 1;
  loop until T2 = 1 || Turn = 1 { }       loop until T1 = 1 || Turn = 2 { }
  Critical();                             Critical();
  T1 = 1;                                 T2 = 1;
} end loop                              } end loop
```

3.3 Bakery Algorithm

```
Integer N1 = 0
Integer N2 = 0
```

```
loop {                                    loop {
  Non-Critical();                           Non-Critical();
  N1 = 1;                                   N2 = 1;
  N1 = N2 + 1;                              N2 = N1 + 1;
  loop until N2 = 0 || N1 <= N2 { }         loop until N1 = 0 || N2 < N1 { }
  Critical();                               Critical();
  N1 = 0;                                   N2 = 0;
} end loop                                } end loop
```

3.4 N-Bakery Algorithm

```
Integer[] N; Integer[] C;
Integer I = Task id;

loop {
  Non-Critical-I();
  C[I] = 1;
  N[I] = 1 + Max(N);
  C[I] = 0;
  for J in 1..N {
    if (J != I) {
      loop until C[I] = 0 { }
      loop until (N[J] = 0 || N[I] < N[J] || (N[I] = N[J] && I < J)){ }
    }
  }
  Critical-I();
  N[I] = 0;
} end loop
```

3.5 Hardware Assisted Mutual Exclusion

```
Integer C = 0; Integer Li;

Test-And-Set(Integer Li){    /* A special hardware atomic instruction */
  Li = C;
  C = 1;
}

loop {
  Non-Critical-I();
  loop until Li = 0 {
    Test-And-Set(Li);
  }
  Critical-I();
  C = 0;
} end loop
```

4 Concurrent Problems

There are also a number of well known concurrency problems, which are discussed briefly below. Like the previous section, this probably isn't going to be on the exam, but is useful to known regardless.

4.1 Ornamental Gardens

In this problem, we have two entrances to a garden. Each of these entrances independently counts the number of people that come through, and updates a shared variable. Without concurrent principles intact, it is possible for the entrances to overwrite a count that the other entrance has just register, effectively losing a person.

4.2 Dining Philosophers

In this problem, we have five silent philosophers at a table, with a bowl of spaghetti in the middle and a fork is placed between each pair of philosophers. A philosopher can only eat if he has both of the forks either side of him, and a fork can only be used if it is not being used by another philosopher. This problem demonstrates the problem of avoiding deadlock.

4.3 Sleeping Barber

In this problem, we have one barber, with one barber chair, and a waiting room with some number of chairs in it. When the barer finishes cutting a customer's hair, he checks to see if there are any more in the waiting room. If there are, he brings one in and cuts their hair. Otherwise, he returns to his chair for a short nap. When a new customer arrives, if the barber is asleep, the customer will wake him up, and receive a haircut. If the barber is busy cutting hair, the customer will take a free chair in the waiting room, if there is one. If there is no free chair, the customer leaves. The problem with this is that all of these takes take an unknown amount of time, for instance, it is entirely possible that a customer may arrive, see the barber is cutting hair, and go to the waiting room. If the barber finished the hair cut, and gets to the waiting room before the customer does, he will see no one is there, and go take a nap. This customer will now be waiting in the waiting room until another customer comes in to wake the now sleeping barber.

4.4 Producer-Consumer

In this problem, we have two processes, and a producer and a consumer. The producer creates some entity and stores it in some fixed length buffer. The consumer removes data from this same buffer, one piece at a time. The problem is to ensure that the producer does not more data items that the buffer can fit, and that the consumer does not attempt to consume when there is no data in the buffer.

4.5 Reader-Writer

In this final problem, we consider when many threads must access the same shared memory location at the same time; with some reading, and some writing. No process many access the shared area for reading or writing while another process is in the act of writing to it (meaning multiple threads may be reading it at any one time).

5 Semaphores

So far, the best way we have found to ensure mutual exclusion is through use of global variables, and use of "busy waiting". The problem with this, is that there can be a significant waste of CPU time, and errors can occur due to the use of global variables. For the example of spin locks, the time spinning is necessary to ensure mutual exclusion, but all processes that are spinning are doing no useful work while another process is in its critical section. The scheduler, however, does not know this and will constantly try to run the other processes, even while another is in its critical section. This can slow down the program if the critical sections are large, or if there is a lot of contention (With 10 processes competing to access their critical sections, we could end up using 90% or more of the CPU). This is why we introduce the concept of *Semaphores*. In this section we will learn what a semaphore is, when to use them, and when we should use other methods, like locks. The problem with locks, is that they only provide mutual exclusion, and as we've seen before, there are four main principles of concurrency that we need to uphold.

A semaphore is an integer variable which can take only non-negative values. Once it has been given its initial value, there are only two permissible actions can can be applied to a semaphore, *s* (note that these are atomic actions)[1].

- $P(s)$ (also known as *wait*)

  ```
  if ( s > 0 ) {
    s = s - 1;
  } else {
    suspend execution of process that called P(s)
  }
  ```

- $V(s)$ (also known as *signal*)

  ```
  if some process , p, is suspended by a previous P(s) on this semaphore {
    resume p
  } else {
    s = s + 1;
  }
  ```

A *general semaphore* can have any non-negative value; where as a *binary semaphore* can only have values of 0 or 1. A V operation on a binary semaphore which already has a value of 1 would not be successful.

[1]V stands for verhogen ("increase"), and P stands for the portmanteau *prolaag*, short "for probeer te verlagen", literally "try to reduce", or to parallel the terms used in the other case, "try to decrease". The reason for these Dutch phrases, is that the concept of semaphores was invented by Dutch computer scientist Edsger Dijkstra

Semaphores can be seen as an *abstract data type* that contain:

- A set of permissible values
- A set of permissible operations on instances of the type (P and V).

Where P and V are required to be implemented as atomic actions.

Reading and writing the semaphore value is itself a critical section, and the P and V operations must by mutually exclusive. Suppose we have a semaphore, s, which has value of 1, and two processes simultaneously attempt to execute P on s. Only one of these operations will be able to complete before the next V operation on s, which means the other process is suspended. The P and V operations are generally run in quick succession (with a critical section in the middle), so this prevents any concurrency problems from occurring. The semaphore operations between different semaphores do not need to be mutually exclusive.

At this point, it is worth noting that the definition of V does not specify which process is woken up if more than one process has been suspended by the same semaphore. This means there are issues with fairness to consider when working with semaphores.

5.1 The implementation of semaphores

The P and V methods of semaphores can be implementation using any of the mutual exclusion methods we have seen thus far (Peterson's, Dekker's, Special Hardware Instructions, Disabling Interrupts, e.g), each with their own advantages and disadvantages. When implementing specifically the V operations, there are several ways that the suspension of processes can be achieved: some poor methods, like "busy waiting"; and some better methods, like *blocking* the process, where processes are paused, waiting for some event to happen, but are not actively using CPU time.

We are quickly going to go through the a proof that mutual exclusion holds for a general semaphore implementation, using the method we specified earlier. The actual algorithms we will be using are specified below.

```
                    Semaphore S = 1;

initialise;                         initialise;
while ( true ) {                    while ( true ) {
a1  Non-Critical-1();               a2  Non-Critical-2();
b1  P(s);                           b2  P(s);
c1   Critical-1();                  c2   Critical-2();
d1  V(s);                           d2  V(s);
}                                   }
```

We begin, as usual, by specifying the problem we are solving.

$$\neg(at(c1) \wedge at(c2))$$

We then look at our base case, $P(0)$. At the first step of execution, the processes are at $a1$ and $a2$; which does not invalidate our property, making our base case true. Now we may draw our column chart.

Process 1	Process 2	Process 1	Process 2

Now we add the state that our unobserved process would be, and what variables would need to be true for this to be the case. We have skipped a step and also filled in the states that the observed algorithm could be in at this time.

Process 1	Process 2	Process 1	Process 2
at(c1)	a1	a2	at(c2)
S = 0	b1	b2	S = 0
	d1	d2	

We know list all states that would be reached after one step of the algorithm for each of the listed states. We also highlight all of the states that could cause the property to be falsified.

Process 1	Process 2	Process 1	Process 2
at(c1)	a1 → b1	a2 → b2	at(c2)
S = 0	b1 → c1	b2 → c2	S = 0
	d1 → a1	d2 → a2	

Finally, we mark each of the states in terms of their validity, in relation to the variable constraints on the observed algorithm.

Process 1	Process 2	Process 1	Process 2
at(c1)	a1 → b1 ✔	a2 → b2 ✔	at(c2)
S = 0	b1 → c1 ×	b2 → c2 ×	S = 0
	d1 → a1 ✔	d2 → a2 ✔	

At this point we can say with certainty that the property of mutual exclusion holds for this general semaphore solution, because the only states that disprove the property are invalid. I hope that it is clear to see *why* the semaphore provides concurrent protection. In our example, whenever one process calls the P function, it prevents the other process from progressing - as calling P on the second function will cause it to block. Only after the first process is finished, and calls the V function, will it be able to unblock and continue.

A useful metaphor for thinking about semaphores is that they are resource managers. In this scenario, if a process wishes to access the resource, it performs a P operation. If it is successful, it decrements the amount of remaining resources available, and continues. If there is no resources left, the process must wait.

When the process is finished with a resource it performs a V operation. If there were processes waiting on the resource, one is woken, but if not, the amount of resource is increased.

Properties of Semaphores	
Mutual Exclusion	True
Absence of deadlock/livelock	True
Absence of unnecessary delay	True
Eventual Entry	Guaranteed for 2 processes. Guaranteed for more that 2 process if semaphore is implemented with fairness

Semaphores also have the advantages that they work for n processes, are much simple to implement that other solutions, and avoid "busy waiting". As an example of the simplicity of semaphores, we can solve the problem of the ornamental garden by simply added a P and V commands around the counter increment, concurrency would be maintained.

5.2 Condition Synchronisation

Condition synchronisation is the method of delaying a process until some boolean value becomes true. This can, once again, be implemented with "busy waiting", but again this is inefficient. Semaphores, however, make for a much easier implementation of condition synchronisation. A good example of this is in the producer-consumer problem; we are delaying the consumer until there is some to consume, and we prevent the producer producing when there is no space.

We will quickly look at the producer-consumer problem in the event of an infinite buffer, with the following constraints:

The producer may produce a new item at any time

THe consumer may only consume when the buffer is not empty

All items produced are eventually consumed

The code for this is as follows:

```
Object[] buf = new Object[inf]
Semaphore n = 0;
```

```
// Producer
Object v = null;
Integer in = 0;
while ( true ) {
  buf[in] = v;
  in = in + 1;
  V(n);
}
```

```
// Consumer
Object w = null;
Integer out = 0;
while ( true ) {
  P(n);
  w = buf[out];
  out = out + 1;
}
```

I hope it is obvious in this solution that if the consumer attempts to consume without there being anything in the buffer, it will be blocked. The producer would then be fee to add to the buffer, resulting in a call of V, waking the consumer and allowing it to consume the new data. This solution ensures that no items are lost or duplicated in transit, items are consumed in the order they were produced, and all produced items are eventually consumed.

There are other variations of this problem: multiple producers and a single consumer, multiple producers and multiple consumers, and single producer and multiple consumer are all valid variations. In more advanced variations of the problem, and in multiprogramming and multiprocessing implementations, communications between the producers and consumers is dealt with by a *shared buffer*. A *buffer* is an area of memory used for temporary storage of data while it is in transit from one process to another. The producer writes to the buffer, and the consumer then reads from it. An example of this in with the UNIX piping system.

5.3 Synchronisation (of the General Producer-Consumer)

The general producer-consumer problems requires both mutual exclusion and condition synchronisation to function correctly. The mutual exclusion is used to ensure that more than one producer or consumer does not access the same buffer slot at the same time. Condition synchronisation is used to ensure that data is not read before is has been written, and that data is not overwritten before it has been read.

The properties that a general producer-consumer buffer-based solution should have are:

- no "items" are read from an empty buffer
- data items are only read once
- data items are not overwritten before they are read
- items are consumed in the order they are produced
- all items producers are eventually consumed
- the four concurrency principles (mutual exclusion, no deadlock/livelock, no unnecessary delay, and eventual entry)

These solutions can be judged on a number of different criteria, including: fairness, correctness, and efficiency. These solutions may use different buffer sizes, and even different protocols for synchronising access to these buffers.

As mentioned before, the infinite buffer solution is good, and only had one major problem to consider - ensuring the consumer did not get ahead of the producer. When we have a single sized buffer (useful for when we are implementing I/O on peripheral devices, or creating dedicated programs running on bare machines), we have two problems to consider: preventing the consumer getting ahead of the producer, and preventing the producer getting ahead of the consumer. The algorithm for implementing a single buffer producer-consumer solution is as follows:

```
Object buf;
binary semaphores empty = 1, full = 0;
```

```
// Producer
Object x = null;
while ( true ) {
  x = new data();
  P(empty);
  buf = x;
  V(full);
}
```

```
// Consumer
Object y = null;
while ( true ) {
  P(full);
  y = buf;
  V(empty)
  // use y
}
```

This single element buffer works well if the producer and consumer are producing at the same rate. In this case, processes don't have to wait very long to access the single buffer, and many low-level synchronisation problems are solved in way. Unfortunately, this is often not the case. In the case that the producer and consumer are only equal in speed *on average*, a larger buffer can significantly increase the performance (by reducing the number of times the processes will have to block).

There is another type of buffer to add to our roster, the *bounded buffer*. In this buffer, we use the metaphorical idea of a rounded array. In which the end connects to the beginning, containing n slots. The buffer contains a queue of items which have produced, but not yet been consumed, as displayed in figure 3.

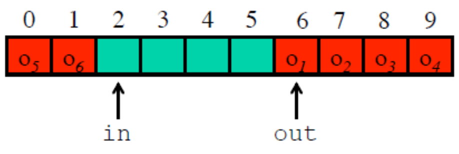

Figure 3: A bounded buffer example

In this example, *out* represents the item at the head of the queue, and *in* is the index of the first empty slot at the end of the queue. Bounded buffers are used for serial input and output streams in many operating systems (UNIX maintains queues of characters for I/O on all serial character devices such as keyboards, screens and printers, and UNIX pipes are also implemented using bounded buffers). The bounded buffer has all the same problems and constraints as the single buffer problem. Its implementation can be seen below:

```
integer n = BUFFER_SIZE;
Object[] buf = new Object[n];
general semaphores empty = n, full = 0;
```

```
// Producer
Object x = null;
integer in = 0;
while ( true ) {
  x = new Data();
  P(empty);
  buf[in] = x;
  in = (in + 1) % n;
  V(full);
}
```

```
// Consumer
Object y = null;
integer out = 0;
while ( true ) {
  P(full);
  y = buf[out];
  out = (out + 1) % n;
  V(empty)
  // use y
}
```

5.4 Dining Philosopher's Problem

As mentioned earlier, in this problem, we have five silent philosophers at a table, with a bowl of spaghetti in the middle and a fork placed between each pair of philosophers. A philosopher can only eat if he has both of the forks either side of him, and a fork can only be used if it is not being used by another philosopher. This problem demonstrates the problem of avoiding deadlock (starvation).

This can be implemented with relative ease by using semaphores:

```
binary semaphore[5] fork = {1, 1, 1, 1, 1}
```

```
// Philosophers 1-4
while (true) {
  // get right fork
  P(fork[i]);
  // get left fork
  P(fork[i+1]);
  // eat and release forks
  V(fork[i]);
  V(fork[i+1]);
  // think
}
```

```
// Philosophers 5, deals with wrapping
while (true) {
  // get right fork
  P(fork[1]);
  // get left fork
  P(fork[5]);
  // eat and release forks
  V(fork[1]);
  V(fork[5]);
  // think
}
```

5.5 Semaphores in Java

In Java 5, support for semaphores was added in the form of *java.util.concurrent's Semaphore* class.

This supports to P and V operations, named *acquire()* and *release()*. The Java implementation also allows for a fairness parameter; which, when true, ensures semaphore blocks processes are resumed in a FIFO order.

6 Monitors

Monitors are a higher level abstraction than semaphores, and aim to rectify some of their problems. For instances, semaphores:

- Are low level
  This makes programming them very difficult, and means any slight mistakes can be catastrophic (A single forgotten V can cause deadlocks, and a single forgotten P can violate mutual exclusion).

- Are unstructured
  Synchronisation code is dispersed all through out the code, as opposed to being in a localised, well-defined region.

- Confuse conceptually distinct operations
  The same primitives are used for mutual exclusion and condition synchronisation.

Monitors are an abstract data type that represent a shared resources. Unlike semaphores that control access to a shared resource, the monitor encapsulates the shared resource. The monitor implements a shared data structure together with coarse grained atomic operations, which are used to manipulate the afford mentioned structure. A monitor has four main components:

1. A Set of private variables which represent the state of the resource

2. A Set of monitor procedures which provide the public interface to the resource

3. A Set of condition variables used to implement condition synchronisation

4. Initialisation code which initialises the private variables

These procedures manipulate the values of the private variables (known as condition variables) in the monitor. Only the names of the monitors procedures are visible outside the monitor. The only way a process can read or change the value of private monitor variable is by calling one of these procedures, which will update the private monitor variables shared among all monitors. Statements within monitor procedures or initialisation code may not access variables declared outside the monitor (unless they are passed as arguments to a monitor procedure).

The condition variables are used to delay a process that can't safely execute a monitor procedure until the monitors state satisfies some boolean condition (this is the monitor implementation of condition synchronisation). These variables are not visible outside the monitor and the only access to them is via special monitor operations within monitor operations. Like semaphores, the values of condition variables cannot be tested or assigned directly, even by the monitor procedures.

6.1 Synchronisation of Monitors

Synchronisation within monitors is achieved using monitors procedures and condition variables. Mutual exclusion is already implicit, by the very definition of monitors (as their methods are executed with mutual exclusion; only one process is allowed to be in the monitor executing at any one time). At most, one instance of one monitor procedure may be active in a monitor at a time. This means that if one process is executing a monitor procedure and another process calls a procedure of the same monitor, the second process will block and will be placed on the entry queue for that monitor. When the process in the monitor completes its monitor procedure call, mutual exclusion is passed to a blocked process on the entry queue. These entry queues are usually defined to be FIFO, so the first process to block will the be next to enter the monitor. This solves the critical section problem because, if all the shared state in the system is held in private monitor variables and all communication between processes is via calls to monitor procedures, then access to any part of the shared state by any process is guaranteed to be mutually exclusive of any other that accesses that part of the state. We can even implement the different classes of critical sections by using a different monitor for each class.

Condition synchronisation must be programmed explicitly using condition variables. The value of a condition is a delay queue of blocked processes waiting on some condition. If a call to a monitor procedure can't proceed until the monitor's state satisfies some boolean condition, the process that called the monitor procedure waits on the corresponding condition variable. When another process executes a monitor procedure that makes the condition true, it signal to the processes waiting on the condition variable. Condition variables are like semaphores used for condition synchronisation.

There are also monitor invariants to think about, which are conditions that must hold before and after the locking of a monitor. During a lock, these conditions may change, as no other processes can see this intermediate state.

6.2 Condition variable operations on variable *v*

- $wait(v)$
  If a process can't proceed, it blocks on a condition variable v, when calling this operation. The blocked process relinquishes exclusive access to the monitor and is appended to the end of the delay queue for v.
- $signal(v)$
  Processes blocked on a condition variable v are woken up when some *other* process performs a signal operation on v. This awakens the process at the front of the delay queue. If the delay queue is empty, signal will do nothing. It's worth noting that a signal is a hint something may have happened, and not a guarantee (things can change between the signalling and the waking of a process).
- $signal\_all(v)$
  Wake all processes on the delay queue for v and continue

When waiting on a conditional variable, a process will release its lock on the monitor and move to that condition variables queue. Where they stay until signalled (when at the top of the queue), or broadcast. When a monitor procedure calls *signal* on a condition variable, it wakes up the first blocked process in the delay queue waiting on the condition. There are two ways in which this can be done:

1. Signal and Wait
   The signaller waits until some later time and the signalled process executes now
2. Signal and Continue
   the signaller continues and the signalled process executes at some later time.

Below is a bounded buffer implemented with monitors.

```
Monitor BoundedBuffer {
 // Private Variables
 Object[] buf = new Object[n];
 integer out = 0,    // First full slot index
         in = 0,     // First empty slot index
         count = 0; // Number of full slots

 // Condition Variables
 condvar not_full ,  // Signalled when count < n
         not_empty; // Signalled when count > 0

 // Monitor Procedures (Signal and Continue)
 void append (Object data){
         while ( count == n) {
                 wait(not_full);
         }
         buf[in] = data;
         in = (in + 1) % n;
         count++;
         signal(not_empty);
 }

 void remove (Object &item) { // Pointer
         while ( count == 0 ) {
                 wait(not_empty);
         }
         item = buf[out];
         out = (out + 1) % n;
         count--;
         signal(not_full);
 }
}
```

In Brian Logan's lecture slides, "13-Monitors-I", beginning on page 21, he provides a trace of how this algorithm works. Which is explained in the points below, where each numbered point corresponds to the respective slider ("An example trace 5" from Brian's slides corresponds to point 5 here).

1. C1 attempts to remove an item from the accessible monitor
2. C1 is allowed access to the monitor, and the monitor closed to any other processes
3. P1 attempts to append a data item to the buffer in the (closed) monitor
4. P1 is unable to, due to it being closed, and it placed at the head of the entry queue
5. C2 attempts to remove an item from the closed monitor
6. But is also unable to, and it placed in the last slot of the entry queue. There is a process switch, and C1 continues execution
7. Due to count being 0, C1 is required to wait on the not_empty condition variable indefinitely
8. This causes C1 to block, and the monitor is once again accessible

9. P1, the head of the entry queue, is allowed access to the monitor
10. P1, due to the buffer not being full, puts its data item into the buffer
11. P1 now signals the not_empty buffer, to inform any waiting processes that the buffer is no longer empty
12. C1 is no longer waiting on the not_empty conditional variable, is unblocked, and joins the entry queue at the tail
13. P1 is now finished, and terminates. This leaves the monitor open
14. The head of the entry queue, C2, can now access the monitor, after which it closes again
15. As there is something in the buffer, C2 is able to remove it
16. C2 now signals the not_full condition variable
17. C2 terminates, and the monitor is accessible
18. C1 can now, once again, enter the monitor, and close it off to other processes
19. Unfortunately for poor C1, the buffer is again empty, and it must wait on the not_empty condition variable again
20. C1 joins the list of processes waiting on not_empty and blocks. The monitor is reopened.

6.3 Comparison with Semaphores

Monitors and semaphores both have the same expressive power: monitors can be used to simulate semaphores, and semaphores can be used to simulate monitors. What we gain with monitors is a higher level of abstraction.

6.4 Monitors in Java

Monitors form the basis of Java's support for shared memory concurrency. Mutual exclusion can be implemented in Java using the *synchronized* keyword; a synchronized method or block is executed under mutual exclusion with all other synchronized methods on the same object. Java provides basic operations for condition synchronisation in the form of *wait()*, *notify()* and *notifyAll()*, which are implemented with the *signal and continue* signalling discipline. Each object in Java has a single implicit condition variable and delay queue, known as the *wait set.*

6.5 Monitors to solve the Reader-Writer Problem

Given a shared file and a collection of processes that need to access and update it, we have readers, which are processes that only read the file, and writers, that need to both read and write the file. We assume the file is initially in a consistent state, and each read or write that executes in isolate transforms the file into a new consistent state. The reading of a file does change the resource; only the writing does, but the writing must be mutually exclusive of all readers and other writers.

This problem uses the concept of *selective mutual exclusion.* Where the readers need not mutually exclusive of other reading, but writers need be mutually exclusive of themselves and readers. When specifying a solution to this problem, we have to weight up correctness and efficiency. We could make all accesses to the file mutually exclusive. At most one process, whether reading or writing, would be able to access the file at a time. This is simple and correct, but, in general, the performance of such an approach is unacceptable. We need a way for the readers to be able to access the file concurrently.

To ensure correctness and also achieve maximum efficiency, we must require that: if a writer is writing to the file, no other writer may write to the file, and no reader may read it; and any number of readers may simultaneously read the file. We could do this using the "X-Preference" protocol. Where, given a sequence of X and Y, all X actions would be prioritised over all Y actions (or vice versa). For instance, say we had a sequence of requests that arrived in the following order:

$$R_1, R_2, W_1, R_3$$

If we were to use "Reader-Preference", all of the readers would come before the writers.

$$R_1, R_2, R_3, W_1$$

But if we were to use "Writer-Preference", all of the writers would come before the readers.

$$W_1, R_1, R_2, R_3$$

Both of these methods prevent a reader and a writer, or two writers to access the file at the same; but these are not *fair* solutions. For example, using readers' preference, as long as a single reader is active, no writer can gain access to the file (only other readers). If new readers arrive in rapid succession, the writer will be indefinitely delayed. A more fair solution would we: if there are waiting writers, then a new reader is required to wait for the termination at that writer; and, if there are reader waiting for termination of a write, they have priority over the next write. This prevents long streams of one process or the other, without any interleaving.

This problem can be solved with two distinct approaches:

- Mutual Exclusion
  Reader processes compete with the writer, and individual writer processes compete with readers and with other writers.
- Condition Synchronisation
  Reader processes must wait until no writer are accessing the file, and writer processes must wait until there are no readers or other writers accessing the file.
- Monitors
  Discussed below.

We will now discuss how monitors can be used to implement a solution. Even though the file is shared, we cannot encapsulate it in a monitor, because this would mean readers couldn't concurrently access it. Instead, the monitor is used to arbitrate access to the file, where the file itself is global to the readers and writers. An arbitration monitor grants permission to access the file. Processes inform the monitor when they want access to the file (access requests), and when they are finished with it (release requests). With two kinds of access request (read and writer) and two release request (read and write), the monitor would have four procedures:

$$\begin{aligned} startRead() \\ endRead() \\ startWrite() \\ endWrite() \end{aligned}$$

Which would be used like so:

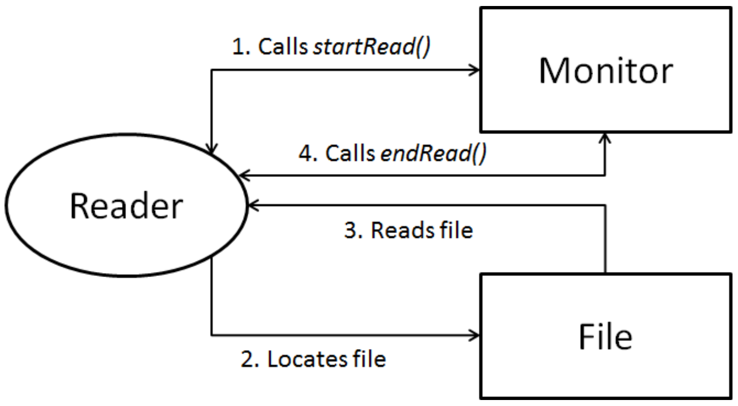

Figure 4: Process of interacting with an arbitration monitor

The actual code solution is as follows:

```
  Monitor ReadersWriters {
    boolean writing = false;
    integer readers = 0,
            waitingReaders = 0,
            waitingWriter = 0;

    condvar okToReader,
            okToWrite;
```

```
public void startRead(){
  if (writing or waitingWriters > 0){
    waitingReaders++;
    wait(okToRead);
    waitingReaders--;
  }
  readers++;
}

public void endRead(){
  readers--;
  if(readers == 0){
    signal(okToWrite);
  }
}




}
```

```
public void startWrite(){
  if (writing or readers > 0 or
        waitingReaders > 0 or
        waitingWriters > 0){
    waitingWriters++;
    wait(okToWrite);
    waitingWriters--;
  }
  writing = true;
}

public void endWrite(){
  writing = false;
  if (waitingReaders > 0){
    signal_all(okToRead);
  } else {
    signal(okToWrite);
  }
}
```

In practice, exclusive writers and concurrent reads are not sufficient to gain the necessary performance in large database systems. The internal structure of the database must be used to limit the are to which a write lock is imposed; this then behaves as if it were supporting a reader and writer protocol.

7 Synchronisation in Java

In Java, threads that access a shared variable take a local copy of it, which they are allowed to manipulate. This is saved in, what's known as, working memory; which is an abstraction of caches and registers, used to storing values. The Java memory model specifies when values must be transferred between main memory and per-thread working memory. There are three properties that must be considered:

- Atomicity
  Which instructions must have invisible effects? In unsynchronised code, all reads and writes corresponding to fields of any type other than long or double are guaranteed to be atomic.

- Visible
  Under what conditions are the effects of one thread visible to the others. In unsynchronised code, changes to fields made by one thread are not guaranteed to be visible to other threads. In synchronized code, any changes made in one synchronized method or block are visible with respect to other synchronized methods or blocks on the same object.

- Ordering
  Under what conditions will the effects of operations appear out of order to any given thread. In unsynchronised code, from the point of view of other threads, instructions may appear to be executed out of order. In synchronised code, the order of calls is preserved from the point of view of any other threads.

7.1 Volatile Fields

If a field is declared as *volatile*, a thread must reconcile its working copy of the field with the master copy every time it access that field. Reads and writes to volatile fields are guaranteed to be atomic (even for longs and doubles), new values are immediately propagated to other threads, and (from the point of view of other threads) the relative ordering of operations on volatile fields are preserved. However, these ordering and visibility effects surround only the reading and writing of a violate field (for instance, a read-write command, like "++" on a volatile field will not atomic).

7.2 Mutual Exclusion in Java

It can be difficult to implement mutual exclusion algorithms in Java, due to problems of visibility, ordering, scheduling and efficiency. We could use special machine instructions or atomic memory access, or even semaphores, and implement them as a Java class which implement the P and V operations from java.util.concurrent. Alternatively, we can use monitors, as there is a straightforward mapping from designs based on monitors to solutions using synchronized classes.

These synchronized classes are achieved in Java by use of the *synchronized* keyword, which is the built in method to implement mutual exclusion in Java. Both methods, and blocks can be synchronized. Every object has a lock (inherited from the *Object* class), which, when a synchronized method or block is called, must be obtained before execution can continue. At the end of execution, this lock will be released. This synchronized also allows for the implementation of coarse grained atomic actions.

When a thread invokes a synchronized method (say, *foo()*), on an object x, it tries to obtain a lock on this object x. If another thread already holds this lock, the thread invoking *foo()* will block. When this thread eventually obtains the lock, it executes the body of the method of then release the lock - even if the exit occurs due to an exception. When one thread releases a lock, another thread may acquire it (perhaps even the same thread); there are no guarantees about which thread will acquire the lock next or if any thread will ever acquire a specific lock. We say that locks are *re-entrant* if they can claim a lock multiple time without blocking on itself. If a lock is non re-entrant, it could grab a lock, and then block when attempting to grab it again - deadlocking itself.

7.2.1 Synchronized Methods

The synchronized keyword is not part of a method's signature, and is not automatically inherited when subclasses override superclass methods. When a method is synchronized in some subclass, it uses the same lock as in their super classes. Finally, methods in interfaces or constructors cannot be declared as synchronized.

7.2.2 Synchronized Blocks

Block synchronization is lower-level that method synchronization. Synchronized methods synchronize on an instance of the method's class, which allows for synchronization on *any* object. This then allows us to narrow the scope of a lock to only part of the code in a method, meaning a different object can be used to implement the lock.

7.3 Conditional Synchronisation in Java

There a multiple approaches to condition synchronisation in java: busy waiting, where processes sit in loops until a condition is true, semaphores, using the P and V operations to wait for a condition and signal when it occurs, or monitors, which have condition synchronisation that is explicitly programmed using condition variables and monitor operations (like wait and signal).

There is built in support for condition synchronisation in Java, using the *wait()*, *notify()* and *notifyAll()*. The *wait()* method is used to delay a thread; used in conjunction with a loop, this method can be used to cause a thread to block if some condition is false. When a method that changes the truth value of one of the condition variables, the *notify()* and *notifyAll()* methods can be used to wake up the blocked threads, so they can check if they are allowed to proceed depending on the new value of the condition variable. Java uses the Signal and Continue signalling discipline. All of these methods must be executed within a synchronized method of block. Specifically, the function of the methods are as follows:

- *wait()*
  Releases the lock on an object held by the calling thread. This threads then blocks and is added to the wait set for the object
- *notify()* and *notifyAll()*
  One (random) thread is awoken, or all threads are woken up. The thread that invoked these methods continues to hold the lock until it is finished executing. The awakened thread(s) remain blocked and execute at some future time when they can acquire the lock. The *notify()* method can be used to increase performance, if only thread needs to be woken, and *notifyAll()* is better when the threads are waiting on different conditions.

7.4 Interrupting a Thread

Each Thread object has an associated boolean interruption status. The *interrupt()* method sets the thread's interrupted status to true, and the *isInterrupted()* method returns true if the thread has been interrupted. Threads can periodically check their interrupted status, and if it is true, clean up and exit.

Threads which are blocked from called to *wait()*, *sleep()* or *join()* aren't runnable, and cannot check the value of their interrupted flag. Interrupted a thread which is not runnable aborts the thread, throws an *InterruptedException*, and sets the thread's interrupted status to false. As a result of this, calls to these methods are generally surround by try-catch blocks.

7.5 Monitor implementation in Java

Before continuing, here is the bounded buffer problem again, implemented this time in Java.

```
class BoundedBuffer {
  // Private Variables
  private Object buf;
  private int out = 0,    // First full slot index
  private int in = 0,     // First empty slot index
  private int count = 0; // Number of slots

  public BoundedBuffer(int n){
    buf = new Object[n];
  }

  //Monitor Procedures
  public synchronized void append (Object data){
    try {
      while ( count == n ){
        wait();
      }
    } catch (InterruptedException e){
      return;
    }

    buf[in] = data;
    in = (in + 1) % n;
    count++;
    notifyAll();
  }

  public synchronized Object remove () {
    try {
      while (count == 0){
        wait();
      }
    } catch (InterruptedException e) {
      return null;
    }
    Object item = buf[out];
    out = (out + 1) % n;
    count--;
    notifyAll();
    return item;
  }
}
```

7.6 Fully Synchronised Objects

The safest design strategy based on mutual exclusion, is the use of fully synchronized object. In these objects: all methods are synchronized, there are not public fields or other encapsulation violations, all methods are finite, all fields are initialised to a consistent state in constructors, and the state of the object is consistent at the beginning and end of each method.

Unfortunately, there are some issues with synchronized method and blocks. These issues are that: there is no way to back off from an attempt to acquire a lock (no times or cancellations), there is strict block structured locking, and there is no way to alter the semantics of a lock (for instance, read or write protection). One way these problems can be overcome is by using utility classes to control the locking instead.

Recall that threads periodically check their interrupt status, and shut down if they have been interrupted. A good place to check for this interrupt status is before calling a synchronized method; as we may spend a long time contending for the lock otherwise, which can result in threads being unresponsive to interrupts. However, even if we check for interrupts before attempting to acquire locks: a thread trying to acquire the lock must be prepared to wait indefinitely, and deadlocks are fatal and unrecoverable. Luckily, we can implement more flexible locking protocols using utility classes.

7.7 The Lock Interface

The utility class, java.util.concurrent defines a Lock interface and a number of classes that implement this interface. The interface specifies the following:

```
public interface Lock {
  void lock();
  void lockInterruptibly() throws InterruptedException;
  boolean tryLock();
  boolean tryLock(long timeout, TimeUnit unit) throws InterruptedException;
  void unlock();
  Condition newCondition();

}
```

We can use the Lock interface to replace the synchronized blocks we have been using thus far. Instead of:

```
synchronised ( this ) {
        // Body
}
```

We can use:

```
Lock lock = new ReentrantLock();
lock.lock();
try {
  //body
} finally {
  lock.unlock();
}
```

The Lock interface supports three time of lock acquisition, that overcome the problems of the synchronized keyword:

- Polled Lock Acquisition: *tryLock()*
  Allows control to be regained if all the required locks can't be acquired
- Timed Lock Acquisition: *tryLock(timeout)*
  Allows control to be reagined if the time available for an operation runs out
- Interruptible Lock Acquisition: *lockInterruptibly()*
  Allows an attempt to acquire a lock to be interrupted

Like mentioned, synchronized methods and blocks limit themselves to strict block structured locking. This means a lock is released in the same block as it was acquired, regardless of how control exists the block. While this can help prevent potential coding errors, it can be extremely inflexible. Instead, we use utility classes. When using the Lock interface, we can acquire locks while current locks are being held. This allows for extremely fine-grained locking and increases potential concurrency, but is generally only worthwhile in situations where there is a lot of contention.

The final problem of the synchronized keyword, is that there is no way to alter the semantics of a lock (for instance, the different requires for read and write protections). This makes it different to solve selective mutual exclusion problems, like the readers and writers problem. Once again, utility classes come to the rescue. There is a specific ReadWriteLock interface in java.util.concurrent. This comes with two functions:

```
public interface ReadWriteLock {
  Lock readLock();
  Lock writeLock();
}
```

This interface maintains a pair of associated Locks, one for read-only operations, and one for writing. The readLock may be held simultaneously by multiple reader threads, so long as there are no writers. The writeLock, however, is exclusive. Since the readLock and writeLock implement the Lock interface, the also support polled, timed, and interruptible locking.

A ReadWriteLock can allow for a greater level of concurrency in accessing shared data than that permitted mutual exclusion locking, provided that:

- The methods in a class can be separated into those that only read internally held data, and those that read and write
- Reading is not permitted while writing methods are executing
- The application has more readers than writers
- And the methods are time consuming, so it pays to introduce more overhead, in order to ensure concurrency among reader threads

7.8 Context Switching in Java

When a thread blocks, or another is scheduled, the JVM must perform a context switch. This involves saving the resisters of the suspended thread and loading the resisters of the newly scheduled thread - which takes time. A concurrent program runs faster if we can reduce the number of context switches that happen. Context switching can be minimised by delegating operations with different *wait()* conditions to different helper objects. Such helper objects serve as condition variables; places to put threads that need to wait and be notified.

8 Remote Invocation

As mentioned near the beginning of this document, another method of communication between processes is *Message Passing.* In this, processes communicate by sending and receiving messages using special message passing primitives, that have inbuilt synchronisation. These methods are: *send(destination) message*, which sends *message* to another process, *destination*; and, *receive(source) message*, which indicates that a process is ready to receive a message, *message*, from another process, *source.*

If a process attempts to receive a message before one has been sent, it will block until there is a message for it to read. The differences are mainly in the behaviour of the sending process. This process uses one of three communication methods:

- Asynchronous
  The sending process continues without waiting for the message to be received (e.g Unix Socket, or java.net).
- Synchronous
  The sending process is delayed until the corresponding receive is executed (e.g CSP, or occam).
- Remote Invocation
  The sending process is delayed until a reply is received (e.g. RPC, or Extended Rendezvous).

There are some inherent problems with message passing with asynchronous and synchronous communication methods, that is, they assume only one-way communication. This means: messages are being transmitted in one direction only, from sender to receiver; message passing is well suited to problems in which the flow of information is essentially one-way (producer-consumer problems); two way information flow has to be programming with two explicit message exchanges (e.g. client server interactions).

With remote invocation, a processes executes a synchronous send, and waits until the reply is received. This combines aspects of monitors and synchronous message passings: interaction is via public procedures, and calling a procedure delays the caller. This provides two way communication from the caller to the process, and is implemented using message passing. We can implement remote invocation in two main ways:

1. Remote Procedure Call
   A new process is created to handle each call
2. Extended Rendezvous
   Requests are dealt with by an existing process.

These two methods can both be described by the abstract concept of a "Module". A module contains both processes, local procedures, and exported procedures in the following format:

- The header contains the signatures of the exported procedures
- The body contains local procedures and processes, local variables, and initialisation code
- At any point in time, a module contains zero or more processes
- Different modules may reside in different address spaces

The syntax for a module is as follows:

```
module <moduleName>
          // Header (signatures of exported procedures)
          export <procID1>(args);
          export <procID2>(args);
          ...

body
          // local variables
          // initialisation

          // implementations of exports module procedures
          // local procedures
          // local (background) processes
```

Communication between these modules is done through calls to exported procedures. Arguments and return values are passed as messages, and the sending and receiving of messages is implicit, rather than being explicitly programmed. Communication within these modules is similar to that of monitors, in that processes within a module can share variables and call procedures declared in that module.

8.1 Remote Procedure Calls

In Remote Procedure Calls, a module contains zero or more processes, and some number of exported procedures. Local processes are called background processes, and processes that result from remote calls to exported procedures which are called server processes.

There are two ways for server and background processes in an RPC module to have mutually exclusive access to shared variables and to synchronise with each other:

1. All processes in the same module execute with mutual exclusion (like as in monitors), with condition synchronisation being programmed explicitly using semaphores and/or condition variables, or;

2. Processes are executed concurrently within a module and both mutual exclusion and condition synchronisation are programmed explicitly using semaphores and/or condition variables.

Below, in figure 5, you can see the full communication between two modules and how a remote procedure call is dealt with. In this figure, Module 1 calls a remote procedure in Module 2, and blocks. Module 2 creates a new thread, with a server process, to deal with this, and its background process remains running. When the server process completes, the values are return to Module 1, and the process unblocks.

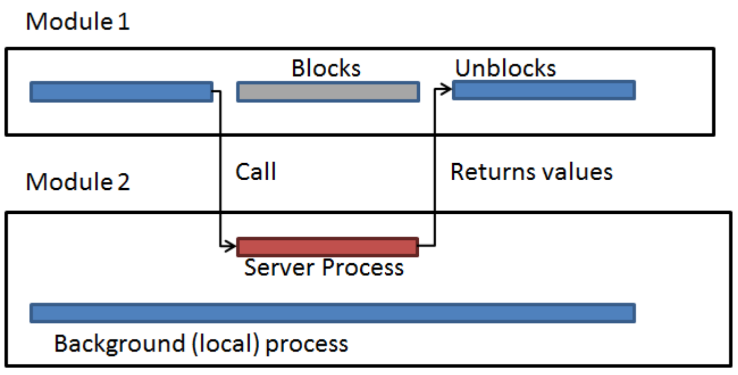

Figure 5: Full communication with RPC of two Modules

8.2 Extended Rendezvous

Extended Rendezvous combines communication and synchronisation. As with RPC, a process invokes an operation by means of a remote call, a server process waits for, and then acts on a single call, and these calls are served one at a time, rather than concurrently. The caller and server processes synchronise (rendezvous) on this call.

In extended rendezvous, a module contains a single process, and some export operations: the header, containing signatures of operations (or entry points) exported by the module; the body of a module consisting of a single process that services the call; and *accept statements*, that block the server process until there is at least one pending call of an exported operation. In addition to these, like RPC, arguments to the call, and any return values, are passed as messages between the caller and server processes.
Below, in figure 6, you can see the full communication between two modules and how an extended rendezvous is dealt with. In this figure, Module 1 synchronises with Module 2, and can then call a procedure in from Module 2. No new thread is created this time, instead, the Module 2 switches to dealing solely with this request; and Module 1 blocks. When the server process completes, the values are return to Module 1, and the process unblocks.

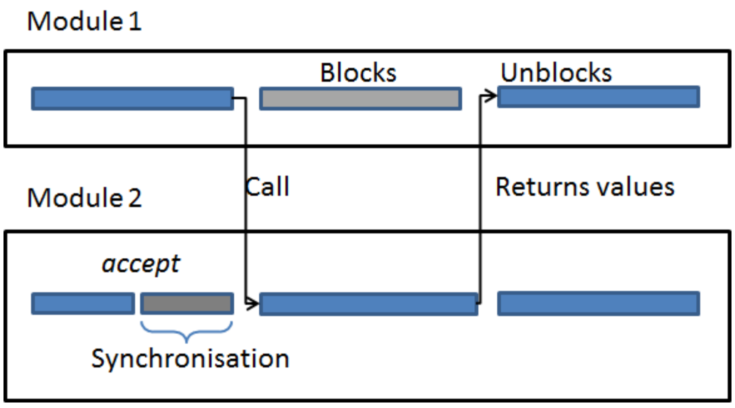

Figure 6: Full communication with ER of two Modules

9 Distributed Processing in Java

9.1 Java.rmi

The java.rmi package is how Java deals with remote produce calls (standing for Remote Method Invocation). Remote innovation is based on the model of a procedure call, and , in Java, all non-static methods must be invoked on an object. This means Java requires both remote methods and remote object, on which these methods can be invoked.

A remote object is one which has methods that can be invoked from other Java Virtual Machines; potentially on different hosts. A remote object must be described by one or more remote interfaces, each of which extend java.rmi.Remote (and have methods that must throw RemoteExceptions). Remote method invocation is the action of invoking a method of a remote interface on a remote object.

If you remember modules, there are several elements of this process that are very similar. for instance,m the Remote interface is similar to the header of a module, in that it contains signatures of exported procedures. Further to this, a class implementing a Remote interface is similar to the the body of the module, containing local a variables and methods, along with initialisation code. The processes in a module are the threads running on the target JVM.

The details of communication between Remote objects are handled by RMI, which uses Sockets and Serialization to implement the transfers of arguments and results.

9.2 The structure of RMI applications

The server creates some remote objects, make references to them accessible, and wits for clients to invoke remote methods of them.

The client get these remote references, either from the RMI register, or as a return value of a remote method, and invoke remote methods on them.

9.3 The RMI Registry

This is a particular object that is used to find references to remote objects. Once a remote object has been registered with the RMI Registry on some local host, clients from any host can look it up. This done through name, and allows the host to obtain a reference to it (known as a stub), and then invoke methods on it.

9.4 Stubs

The stub acts as a proxy for a remote object, and is responsible for carrying out method calls on the remote object. The process of invoking a stub method is as follows:

1. Connection is initiated, with the remote JVM containing the remote object
2. The method parameters are written and transmitted to the remote JVM
3. The results of the method invocation are then waited on
4. When these results are available, they are returned to the caller.

The arguments and returns values must be serializable objects. In addition to this: non-remote method arguments and results are passed by copying changes made by the object, and are not visible to other clients; but remote objects that are passed by reference, when changed by one client, will have this change visible to all clients.

Below (figure 7) is an diagram of the interaction between clients, the server, and the RMI registry. In this, the server creates stub, that the other elements of the system can use to then work with. The first case of this is with the RMI registry, which uses the stub to set up the naming and referencing of the remote objects that can be used. When a client wishes to interact with the server, it too takes a stub, and all method invocations are applied to the stub, instead of the actual server.

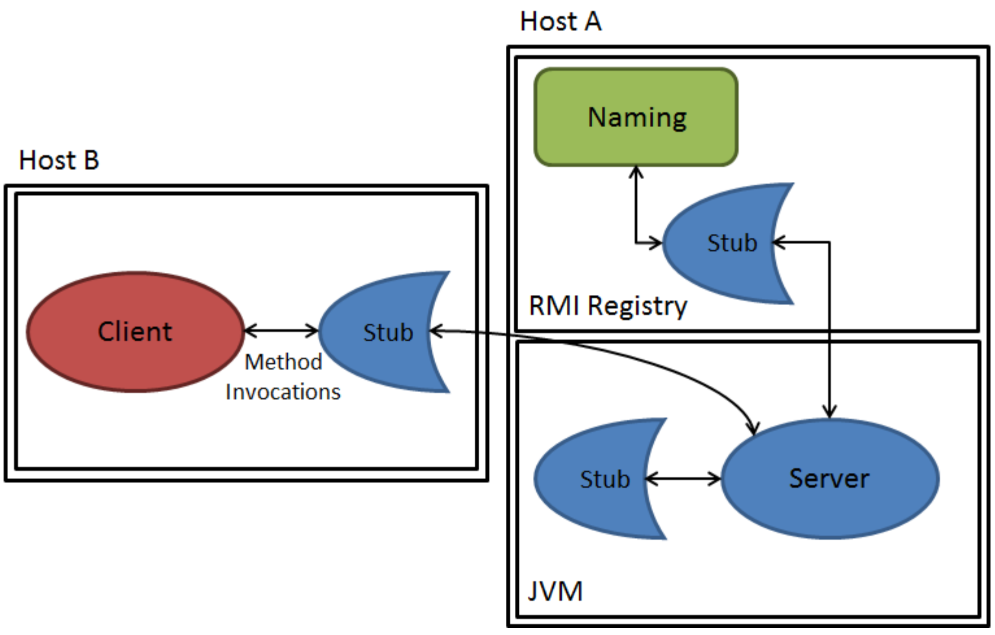

Figure 7: Interactions between clients, server, and RMI registry with stubs