Techniques for developing verified concurrent programs based on monitors and semaphores

by Fazilatunnessa

© Fazilatunnessa

A thesis submitted to the

School of Graduate Studies

in partial fulfilment of the

requirements for the degree of

Master of Engineering (Computer Engineering)

Faculty of Engineering and Applied Science

January 2012

St. John's Newfoundland

Abstract

In concurrent programming, mutual exclusion algorithms are used to avoid the simultaneous access of a common resource. Monitors are objects that can be used safely by more than one thread, as their methods are executed with mutual exclusion. In order for threads to wait for some condition to be met, monitors also provide a mechanism for threads to temporarily give up exclusive access. Monitors also have a mechanism for signaling other threads that some condition has been met.

In this thesis, a general approach to monitors specification and verification code is developed which can be used for solving synchronization problems in an operating system. Specifications are given at the level of C code using the annotation language of Microsoft's Verifying C Compiler (VCC). VCC takes the annotated C program and tries to prove that the program meets these specifications. Later the proposed methodology is demonstrated with example applications.

Acknowledgements

I would like to express my heartiest and sincerest gratitude to my supervisors, Dr. Theodore S. Norvell and Dr. Dennis K. Peters for their intellectual guidance, tireless support, constructive suggestions, inspiration, and close supervision throughout the entire period of my M.Eng. Program.

I would like to thank my course instructors of Memorial University for their valuable lectures and course materials which directly helped me in my research. I also would like to thank Ms. Moya Crocker, Graduate office, Faculty of Engineering and Applied Science and Gerard Brake, ECE, for their administrative and technical assistance during the course of my program.

Last but not the least, I owe my deepest thanks to my beloved husband Sakib Mahmood, my daughter Shadida Mahmood and to my beloved parents, Md. Fazlul Hoque & Mrs. Shakera Houque.

I would like to dedicate my thesis to my elder brother AHSANUL HOQUE RINKU.

Contents

List of Tables

List of Figures

List of Listings

List of Algorithms

Chapter 0

Introduction

0.0 General Background

In order to meet modern-day requirements, software systems are evolving rapidly, and so is their complexity. Therefore a major challenge for the software developers is to develop software that is highly reliable. Because of increasing software complexity, the possibility of errors is increasing. To avoid propagation and compounding of errors, it is preferable to identify the errors in the earlier stages of software development. One of the robust ways of error identification is using formal methods. Formal methods are methods that are languages, techniques and tools based on mathematics.

Computer systems can be shared among many programs and programs to access any of its resources (main store, consoles, etc.) at any time. Computer designers construct various scheduling algorithms for these resources. Each class of resource has its own scheduler. Each scheduler consists of a certain amount of local data as well as some procedures and functions. The procedures and functions that are called by threads that need to acquire and release the resources. Such a combination of data and procedures are called a *monitor* [Hoare, 1974]. To handle mutual exclusion, a data structure named a *semaphore* [Hoare, 1974] is used. Monitors are built on top of semaphores which are a more primitive mechaism for mutual exclusion.

A monitor is an object or module in concurrent programming that is built to be used safely by more than one thread or task. The defining characteristic of a monitor is that its methods are executed with mutual exclusion. That is, at most one thread may be executing any of its methods at each point in time. This mutual exclusion greatly simplifies reasoning about the implementation of monitors compared with code that may be executed in parallel. Monitors and semaphores often form the basis of operating system kernels.

VCC (The Verify C Compiler) [Cohen *et al.*, 2009] is a mechanical verifier for concurrent C programs. As input it takes an annotated (with function specifications, data invariants, loop invariants, ghost code, etc.) C program. If it succeeds to prove the annotations correct, then VCC ensures that the program actually meets its specifications.

In this thesis an approach will be described to verify concurrent programs built with monitors and semaphores.

0.0.0 Software Specification

A system has a set of properties which are known as system requirements. The categories of system properties may include behaviour (functional, timing) of the system. The process that defines these properties is called a specification. A specification can be defined in terms of mathematical syntax and semantics.

Norvell [Norvell, 2009] has defined the behavioural specifications as: i) behaviours the system could engage in and ii) behaviours the system cannot engage in.

He defined *signature* as a function that maps names to nonempty sets and the behavioural specification as a pair of Σ, f where Σ is a signature and f is a boolean function such that $b \in \ dom(f)$ (domain of function f), for all $b : \Sigma$ (behavior b belongs to Σ).

If $f(b) =$ true, then the specification (Σ, f) accepts behaviour b.

If $f(b) =$ false, we say that the specification (Σ, f) rejects behaviour b.

0.0.1 States and Behaviours

States of the computer are modeled as mappings from variable names to values.

For example, if the variable names x and y are of type int, then example states include

$i = \{``x" \mapsto 10, ``y" \mapsto 5\}$

$o = \{``x" \mapsto 6, ``y" \mapsto 5\}$

Behaviour consists of two states, an initial state and a final state. Following is the example of behaviour,

$i \dagger o = \{``x" \mapsto 10, ``y" \mapsto 5, ``x'" \ \mapsto 6, \ ``y'" \mapsto 5\}$; where $i \dagger o$ is the combination of two behaviors i & o

Both input and output states belong to the same signature Σ. Therefore behaviours belong to $\Sigma \dagger \ \Sigma$.

0.0.2 Examples of specifications

Specifications are often of the form

$\langle P \Rightarrow\ Q \rangle$

P represents an assumption about the input. Here P is called the precondition and Q is called the postcondition. For inputs, where P is false, the expression is simplified as follows:

$P \Rightarrow\ Q$

$=$

$false \Rightarrow Q$

$=$

$true$

For such inputs, the specification imposes no restrictions on the output.

The specification of the problem of computing the minimum of two natural numbers (x, y) is as follows:

$\Sigma = \{$"x"$\mapsto N,$"y"$\mapsto N\}$; where N is a set of natural numbers.

$f = \langle x' = min(x, y) \rangle$; where x' is the result value

The specification of the problem of computing the greatest common denominator of two natural numbers is as follows:

$\Sigma = \{$"x"$\mapsto N,$"y"$\mapsto N\}$

$gcd(x, y) = \{x\ if\ y = 0\}$

$gcd(x, y) = \{gcd(y, x\ mod\ y); otherwise\}$

$f = \langle x' = gcd(x, y) \rangle$; where x' is the result value and gcd(x, y) is the function that returns the greatest common denominator of x & y

Let x be the initial value of a program variable and x' be the final value of the same variable. Similarly let y be the initial values of program and y' be the final value of the variable.

Let $\Sigma = \{“x” \mapsto Z, “y” \mapsto Z, “\acute{x}” \mapsto Z, “\acute{y}” \mapsto Z\}; x, \acute{x}, y, \acute{y}$ all are integer variables.

Let $f = \langle x' = 0 \wedge y' = y \rangle$; where f is the function

(Σ, f) is a specification that accepts behaviour,

$\{“x” \mapsto -3, “y” \mapsto 5, “x'” \mapsto 0, “y'” \mapsto 5\}$

But it rejects behaviour,

$\{“x” \mapsto -3, “y” \mapsto -3, “x'” \mapsto 1, “y'” \mapsto -3\}$

0.0.3 Uses of Specifications

Specifications are useful for a number of purposes [Norvell, 2010]:

- Documentation: Managers of large software projects first started to understand the importance of having precise documentation for software products as it was time consuming to obtain reliable software without documentation. Documentation has become the important part of software development.

- Requirements Specification: A specification can be used to describe all the ways that it is acceptable for a system, which may not yet have been built, to behave.

- Testing: After implementation of software, the required specification can be compared with its behaviour. If the system does not behave according to the specification, then an error has occurred. For example, if a behaviour $b : \Sigma$ is observed, and the system specification is (Σ, g), then $\neg g(b)$ indicates an error.

- Verification: Verification is a kind of engineering activity which can be performed with different levels of confidence as well as in different ways. It is a tool/technique which ensures software consistency with its formal specifications. The system is called verified if each behaviour (that the system could engage in) is acceptable to its specification. For an example the specification (Σ, g) refines a specification (Σ, f) if

$\forall b : \Sigma \cdot g(b) \Rightarrow f(b)$; where "$\cdot$" means "such that"

So if the above formula is proven then it has been proved that a system described by(Σ, g) meets the specification (Σ, f).

0.1 Objectives of the thesis:

The aim of the present work is to verify some aspects of kernel code. The main set of objectives of this thesis is to:

- Develop a method for verifying certain kinds of code using semaphores.
- Develop a method for verifying monitors.
- Demonstrate these methods with example applications.

0.2 Organization of the Thesis

This thesis is composed of five chapters. The first chapter (Chapter 0) addresses the general background, objective and scope of the proposed research work.

Chapter 1 presents a brief review and application of verification of concurrent programs. The chapter covers the proofs and theorems of concurrent programs. Related works in the context of operating system verification are presented. Also some examples of the different VCC annotations are described.

Chapter 2 presents the detailed background of semaphores and monitors using examples.

Chapter 3 describes the implementation and verification of s*emaphores* and *monitors* without condition variables. As an example, it presents the verification of *Time of Day* example.

Chapter 4 describes the implementation and verification of *monitors* with condition variables. As an example, it presents the verification of *Producer/Consumer Bounded Buffer.*

Chapter 5 summarizes and concludes the findings of the present work. This chapter also lists original contributions of this thesis along with some guide-lines for future work.

Chapter 1

Background and Related Work

In this chapter the background of concurrent programming is described. Evolution of formal verification is presented in the context of operating system verification. The verification tool VCC [Cohen *et al.*, 2009] is introduced in this chapter. VCC annotations and specifications used in this thesis are also described in this chapter in details using relevant examples.

1.0 Background

A concurrent program is a set of sequential programs, which are designed as collections of interacting computational processes that may be executed in parallel. On a single processor, concurrent programs can be executed by interleaving the execution steps of each computational process. On multi-processors, they can be executed in parallel by assigning each computational process to one of a set of processors that may be colocated or distributed across a network.

In general, the term *process* is usually used in the theory of concurrency whereas the term *thread* is used in programming languages or libraries. However, there is a distinction that can be made between the two terms with respect to the *address space.* A *process* runs in its own *address space,* that is managed by the operating system. A *thread* runs within the *address space* of a single *process.*

Ensuring the correct sequencing of the interactions or communications between computational processes and coordinating access to resources that are shared among processes are the main challenges in designing concurrent programs.

According to *Stallings* [Stallings, 1992], concurrency arises in three different contexts:

- Multiple applications at the same time (which is known as multiprogramming) allows sharing the processing time among a number of active applications.
- Some programs are structured as a set of concurrent processes, which is an extension of the principle of modular design and structured programming.
- Operating systems are often implemented as a set of concurrent *processes* or *threads.*

1.0.0 Difficulties with Concurrency

Maintaining synchronization and communication in concurrent programming is an important goal. A number of different approaches, such as implementing each computational process as an operating system process, or implementing the computational processes as a set of threads within a single operating system process, can be used.

There are some difficulties in concurrency [Stallings, 1992], such as,

- The relative speed of execution of processes depends on the i) activities of other processes, ii) way in which the operating system handles interrupts as well as iii) scheduling policies of the operating system.
- If two processes share global variables to read and write, then the order of access become critical.
- Managing the allocation of resources optimally is the other hard task for the operating system.
- It is difficult to find errors in programming, as the results are not deterministic and not reproducible.

1.0.1 Atomic Statements

The concurrent programming abstraction deals with interleaved sequences execution of the atomic statements. The important property of atomic statements is that, if there are two processes accessing the same variable simultaneously, the output should be same as if they had been executed sequentially in some order. It is important to specify the atomic statements precisely because the correctness of an algorithm depends on this specification.

1.0.2 Competition Among Processes for Resources

The concepts of concurrent programs become a concern when the conflict of processes occurs when sharing the same resource. For example, although two processes are trying to access the same shared resource (such as I/O devices, memory, processor time, clock, etc), each process is unaware of the existence of the other processes.

There are three control problems to be concerned within the case of competing processes [Ben-Ari, 2006]:

- *Mutual exclusion*: When two or more processes require access to a single *non-sharable* resource such a resource is called a *critical* resource and the part of program that uses this resource is called a *critical section* of the program. For example, one printer used by two or more processes, is a single *non sharable* resource. During the course of execution, each *process* will be sending commands to the I/O device, receiving status information, sending data, and/or receiving data. But only one *process* is allowed in the *critical* section at a time. In the case of a printer only one individual process should have control over the printer while printing the file to prevent the chance of interleaving lines from different processes.

- *Deadlock*: The attempt to ensure mutual exclusion may lead to deadlock sometimes. For an example, there are two processes *P1* and *P2*, and two resources *R1* and *R2* and both processes need both resources to perform part of their function. If the operating system assigns *R1* to *P2* and *R2* to *P1*, then there is a possibility of the deadlock that is shown in Figure 1.0.

  P1 is waiting for *R1* and process *P2* is waiting for *R2*. Niether process will release the resource that is owned by them until they can get access to the other resource. The scenario is called deadlock.

- *Starvation*: The final control problem is known as starvation. For example, there are three processes *P1*, *P2* and *P3*. Each process requires periodic access

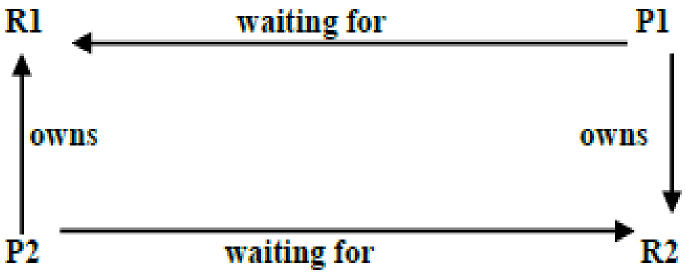


Figure 1.0: Deadlock in processes

to the resource *R*. *P1* gets the access of the resource and the other two processes are delayed. When *P1* exits its critical section, the operating system allows *P3* to access *R*. In the meantime *P1* again requires access before completing the critical section of *P3*. If the operating system grants the access to *P1* after *P3* alternately, then *P2* will be indefinitely denied access to the resource and this is referred to as starvation.

1.0.3 Properties of Concurrent program

There are two kinds of properties in concurrent programming to satisfy: i) Safety Properties and ii) Liveliness Properties.

1.0.3.0 Safety Properties

S*afety* properties require that nothing bad will happen during the execution of a system. Three different examples of safety properties are as follows:

- *Partial Correctness*: If the precondition is true at the beginning of the program then the program will never terminate with the false postcondition.

- *Absence of deadlock*: The program will not enter such a state in which there is no possibility of further progress.

- *Mutual exclusion*: Two different processes can not be enter their critical section at the same time.

1.0.3.1 Liveness Properties

Liveness properties state that something good eventually does happen, which means that the program eventually enters a desirable state. Program termination is the most important *liveness* property. The lots of formal efforts have been given to handle this property. Owicki and Lamport [Owicki and Lamport, 1982] have defined some other kinds of *liveness* properties. For example:

- Each request for service will eventually be answered.

- A message will eventually reach its destination.

- A process will eventually enter its critical section.

1.1 Related Work

In this section, evolution of formal verification is described in the context of operating system verification.

1.1.0 Evolution of formal verification

Formal proof comes along with the human error as well. James Reason [Reason, 1990] described that the large, complex, highly detailed formal proofs is the worst combination for human error. So it is expected to have errors in large proofs. The probability is high if the proofs are constructed by hand. This problem can be solved by machine checked proofs. But then the correctness of the theorem prover comes to the focus.

With respect to that problem, a number of formal methods tools are developed carefully. Some of the tools are made based on a formally analyzed algorithm. As an example, tools in this kind are *PVS* [Owre *et al.*, 1996], *ACL2* [Kaufmann *et al.*, 2000], the *B-tool* [Abrial, 1996] and most popular model checkers, first-order automatic provers and static analysis tools.

Klein [Klein, 2009] has given a brief overview of the operating system verification projects surveyed in his paper.

a. *UCLA*

Walker *et al.* [Walker *et al.*, 1979] presented a report on the specification and verification of *UCLA Secure Data Unix*. *UCLA Secure Data Unix* is an operating system that was aimed at providing a standard *UNIX* interface to applications. The verification effort in this project was focused on the kernel of the *OS*. All of the specifications in *UCLA Secure Data Unix* were represented as state machines which have a set of possible states, a current state, and a set of possible transitions between those states. Later Walker *et al.* proved that the specifications are consistent with each other.

b. *PSOS*

The provably secure operating system (*PSOS*) is a hardware/software co-design with demonstrable security properties [Neumann and Feiertag, 2003]. To design PSOS, which basically used a layered architecture, the project initially developed the Hierarchical Development Method (*HDM*) [Robinson and Levitt, 1977] with its specification and assertion language *SPECIAL*. Principles that are used in implementation of *PSOS* such as, encapsulation and information hiding are known as typical techniques nowadays. The design methodology of PSOS is used for the implementation of the Kernelized Secure Operating System (*KSOS*) [McCauley and Drongowski, 1979] by Ford Aerospace. The Secure Ada Target (*SAT*) [Haigh and Young, 1987] and the Logical Coprocessor Kernel (*LOCK*) [Saydjari *et al.*, 1987] are also inspired by the *PSOS* design and methodology.

c. *KIT*

Kit is a small operating system kernel written for a uni-processor computer with a simple von Neumann architecture [Bevier, 1989]. *KIT* stands for kernel for isolated tasks, which is the main service of it. The kernel of *KIT* consists of 620 lines of assembler source code and 300 lines of actual assembler instructions. Hence the kernel is extremely small and purposely very simple. It is also significant because it is the first formally verified kernel. In *KIT*, the verification was performed using the *Boyer–Moore* theorem prover [Boyer and Moore, 1988] along with the prototype of the *ACL2* prover.

d. *VFiasco*

The *VFiasco* (Verified Fiasco) project started in November 2001. Hohmuth *et al.* [Hohmuth *et al.*, 2002a] presented the main ideas and the approach of the project in 2002. *Fiasco* [Hohmuth and Härtig, 2001] is a binary compatible re-implementation of the high performance, second generation microkernel *L4*. One of the contributions of the *VFiasco* project is the modelling of the *C++* language for the verification of low-level code. The methodology translates a *C++* program directly into its semantics in the theorem prover *PVS*.

e. *EROS / Coyotos*

The *EROS* (Extremely Reliable Operating System) [Shapiro *et al.*, 1999] system is a second-generation microkernel. Shapiro & Weber [Shapiro and Weber, 2000] first formalised and analysed its security model in a pen-and-paper proof. The take-grant model of capability distribution [Lipton and Snyder, 1977] is used in the model. The *Coyotos* project [Shapiro, 2008] is the result where Shapiro designed and implemented a new kernel, as well as designed the proposed implementation language *BitC*.

f. *Verisoft*

In pervasive verification, the correctness of the compiler is verified formally in each step. There is a complete, unbroken formal chain from hardware to applications. The *Verisoft* [Alkassar *et al.*, 2008] project is a significant work which demonstrates the pervasive formal verification [Bevier *et al.*, 1989] of a whole computer system (both the hardware & software).

g. *L4.verified/seL4*

The *L4* verification project combines two different projects: i) *seL4* and ii) *L4*.verified. The *seL4* (secure embedded L4) kernel [Elphinstone *et al.*, 2007] is an evolution of the *L4* microkernel [Liedtke, 1995] with efficient support for security and embedded systems. In the *seL4* methodology, the prototype is written in *Haskell* [Peyton Jones, 2003], which is a high-level programming language that is efficiently executable and similar to the notation of the theorem prover. As *Haskell* is very close to *Isabelle/HOL* [Nipkow *et al.*, 2002], it can be automatically translated into the theorem prover. As an extension of the two projects, a hardware simulator generator was also developed. The simulator takes an instruction set specification and a simple

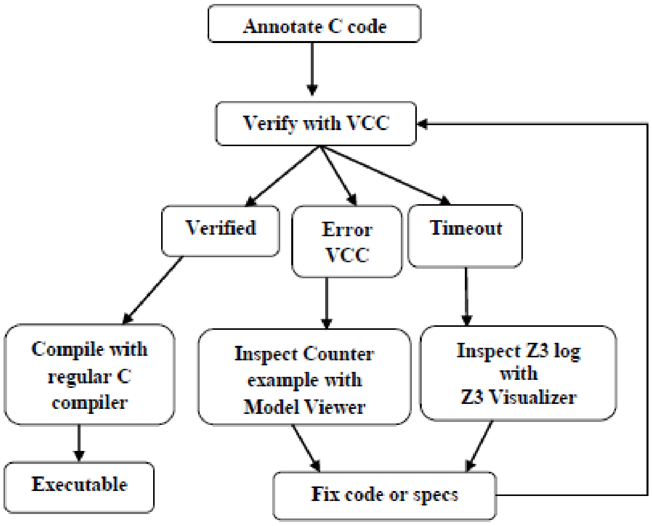

Annotate C code
Verify with VCC
Verified
Error VCC
Timeout
Compile with regular C compiler
Inspect Counter example with Model Viewer
Inspect Z3 log with Z3 Visualizer
Executable
Fix code or specs

Figure 1.1: Flow diagram of VCC

device description and turns into an efficient instruction-level simulator as well as an *Isabelle/HOL* formalisation. This hardware formalisation can then form the source for assembly level verification.

1.2 VCC Specification

The Verifier for Concurrent *C* (*VCC*) is a verifier tool for concurrent C programs. It is developed at Microsoft Research, Redmond, USA, and the European Microsoft Innovation Center (EIMC), Aachen, Germany [Cohen *et al.*, 2009]. *Hypervisor* is a thin layer of software between hardware and operating system (OS) that runs directly on *x64* hardware. It turns a single real multiprocessor *x64* machine into a number of virtual multiprocessor *x64* machines. *VCC* is used to verify the Microsoft Hyper-V hypervisor software [Cohen *et al.*, 2009].

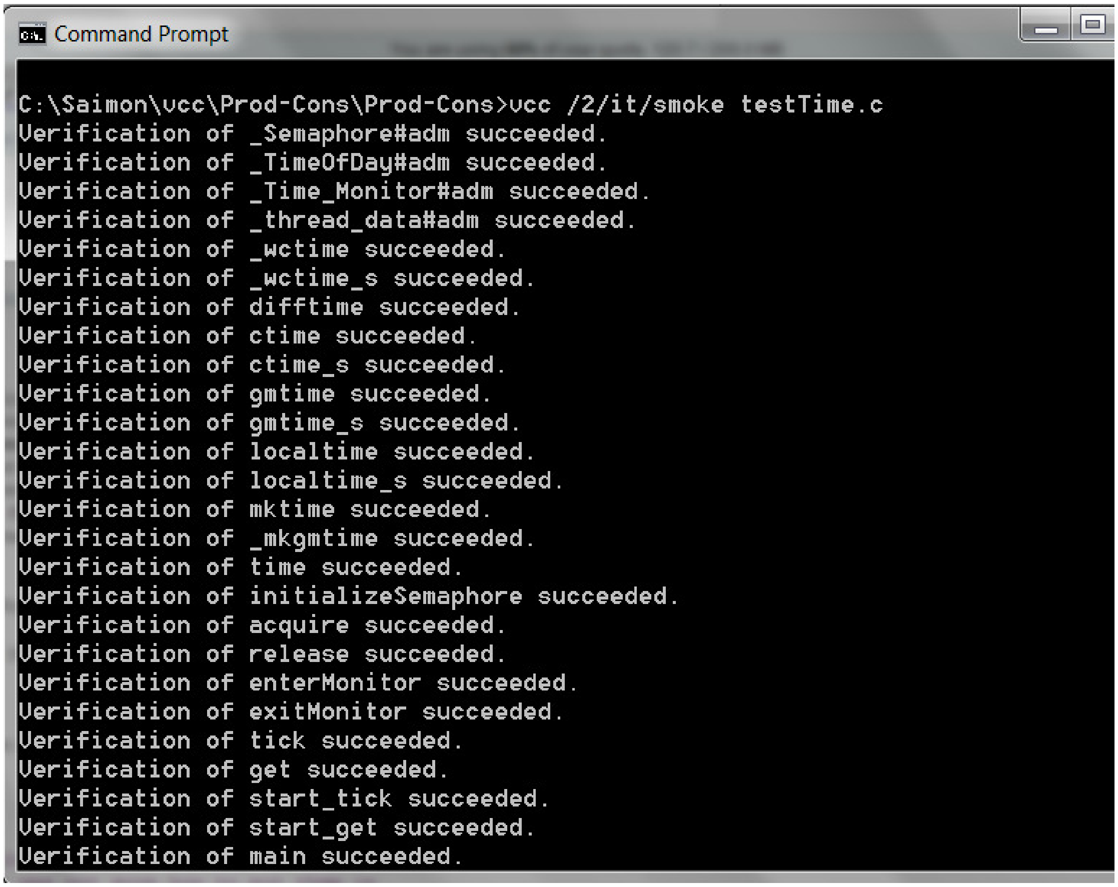
Figure 1.1 describes the work flow of *VCC*. *VCC* extends *C* with annotations giving function pre-conditions and post-conditions, assertions, data invariants, loop invariants and ghost code. Then it attempts to prove the correctness of the annotations. It allows users to add specifications and other annotations directly into the *C* source code. Specification (or ghost) code and objects are used to support verification.

The annotated program can be regularly compiled by using conditional compilation. *VCC* translates the annotated program into the *Boogie language* [Barnett *et al.*, 2006]. The Boogie tool generates verification conditions for partial correctness, passes them to the automatic theorem prover *Z3* [De Moura and Bjorner, 2008] and then *Z3* proves the generated verification conditions.

If any error is found, *VCC* reports that it is unable to verify the correctness of one or more of the annotations. In this case the code can be inspected usng the VCC Model Viewer.

In this thesis *VCC* is used with Microsoft Visual Studio 2008 (VS2008) editor. The code can be updated and the old syntax of *VCC* can be compiled and verified using VS2008. However, the code using new syntex of *VCC* can not be verified in VS2008. All the implementations of this thesis (that are verified by *VCC*) are compiled from command line. The interaction with *VCC* from command line is shown in Figure 1.2.

The */2* option in Figure 1.2 is used to verify the new syntext of *VCC.* The trigger inference for the more complex invariant is enabled by the */it* flag. Using the */smoke* option is used to do *smoke* tests which allows to find unreachable code. It is useful to find the inconsistencies in the specification [Schulte *et al.*, 2010].



```
C:\Saimon\vcc\Prod-Cons\Prod-Cons>vcc /2/it/smoke testTime.c
Verification of _Semaphore#adm succeeded.
Verification of _TimeOfDay#adm succeeded.
Verification of _Time_Monitor#adm succeeded.
Verification of _thread_data#adm succeeded.
Verification of _wctime succeeded.
Verification of _wctime_s succeeded.
Verification of difftime succeeded.
Verification of ctime succeeded.
Verification of ctime_s succeeded.
Verification of gmtime succeeded.
Verification of gmtime_s succeeded.
Verification of localtime succeeded.
Verification of localtime_s succeeded.
Verification of mktime succeeded.
Verification of _mkgmtime succeeded.
Verification of time succeeded.
Verification of initializeSemaphore succeeded.
Verification of acquire succeeded.
Verification of release succeeded.
Verification of enterMonitor succeeded.
Verification of exitMonitor succeeded.
Verification of tick succeeded.
Verification of get succeeded.
Verification of start_tick succeeded.
Verification of start_get succeeded.
Verification of main succeeded.
```

Figure 1.2: Running VCC from command line

1.2.0 Invariants

C types (structs and unions) can be annotated with single or two-state invariants [Cohen *et al.*, 2009]. For all closed objects, these invariants are required to hold and are called system invariants. In the case of single-state invariants, the invariant must hold in each state of the system. For two-state invariants, invariant must hold for each pair of successive states. Invariants are the mechanism to enforce data consistency. The type invariant describes how properly the objects of that type behave.

1.2.0.0 Example with Invariants

Below is an simple example of using invariants in VCC.

```
typedef struct _MonitoredBuffer{
    int buff[CAPACITY];
    int size ;
    _(invariant \this->size >=0 && \this->size <= CAPACITY)
} MonitoredBuffer;
```

The MonitoredBuffer structure consists of an array *buff* of size *CAPACITY*. The *size* parameter is used for the size of the buff. The invariant of *MonitoredBuffer* states that, the size of the buff is in the range from *0* to *CAPACITY*.

1.2.1 Consistency of an Object

The field *\consistent* is defined for every object. The invariants need to hold only when the *\consistent* field is true. Initially the field is *false*. It must be set to *false* before disposing objects.

In addition to the \consistent field, each object has an owner field. The owner of object, *obj,* is defined as *obj->\owner.* This field is of type \object, which is a type of pointers to objects. VCC provides objects, of \thread type, to represent threads of execution, so that threads can also own objects. If a thread owns the object *obj*, it can change the ownership of object *obj.*

While verifying the body of a function, VCC assumes that the function is being executed by some particular thread. The \thread object representing it is referred to as \me.

The followings are some rules of *ownership* and *consistency* [Schulte *et al.*, 2010]:

- On every atomic step of the program, the invariants of all the consistent objects have to hold.

- Only the owning thread can modify fields of an inconsistent object.

- Threads can own themselves.

- Only threads can own inconsistent objects.

From the above rules, objects can be updated in two ways:

In the first case, i) the updated object is *consistent,* ii) the update is *atomic* and iii) the update preserves the invariant of the object.

In the second case, i) the updated object is inconsistent and ii) the update is performed by the *owning* thread.

1.2.2 Wrap/Unwrap Protocol

If any field of an object is annotated with volatile keyword, then that field can be written also when the object is consistent. The non-volatile fields of an object can only be changed after it has been made *inconsistent*. This is performed by the *unwrap* operation. As making the object *inconsistent* is an update, the thread needs to *own* it first.

While *wrapping* an object, VCC does the following steps:

- Assert that the object is *unwrapped. Unwrapped* objects are owned by *me()* (definition is given at section 1.2.4) and are not consistent.
- Assert the invariant is true.
- Set the *consistent* field to true.

The *unwrap* operation does the opposite steps as follows:

- Assert that the object is in the *writes* set.
- Assert that the object is *wrapped.*
- Assume the invariant is true.
- Set the *consistent* field to false.
- Add the *span* of the object (i.e. all its fields) to the *writes* set.

1.2.2.0 Examples with wrap/unwrap protocol:

As discussed in this section, an example of *deposit* method is presented in Listing 1.0 to show the use of *wrap* and *unwrap* syntax in VCC. The method is used to deposit a value to the buffer. Here *theBuffer* is the *MonitoredBuffer* object. The invariant of *theBuffer* is shown below:

$\_(invariant \ \backslash this \rightarrow size \geq 0 \ \&\& \ \backslash this \rightarrow size \leq CAPACITY)$

To change the consistent buffer object; i) the thread needs to *unwrap* it first, ii) update the fields and iii) then *wrap* it again. So the buffer can remain *consistent*.

```
void deposit(int value, Buffer *buf)
_(requires \wrapped(buf))
_(requires buf->size < CAPACITY)
_(ensures \wrapped(buf))
{
     _(unwrap &theBuffer);
   theBuffer.buff[theBuffer.size] = value;
   theBuffer.size = theBuffer.size + 1;
   _(wrap &theBuffer);
}
```

Listing 1.0: Deposit method

1.2.3 Accessing Objects

In different states, the access permissions to objects are different. A *mutable* object can be read from or written to. However, *write* access is only allowed if the object has become *mutable* within the current function or is listed in the function's *writes()* set. *Unwrapping* an object is allowed if the object is listed in the function's *writes()* set or has become wrapped in the current function (by unwrapping its parent object). *Closed* objects that are transitively owned by the *current* thread are considered *thread-local* (if no intermediate object has a volatile owns set), and their *non-volatile* fields can be read. Write access to *non-volatile* fields of *closed* objects is forbidden. Access to the *volatile* fields of *closed* objects requires a guarantee that the object will not be opened by some other thread prior to the access.

There are two ways to obtain such a guarantee [Schulte *et al.*, 2010]:

- While the object is transitively owned by the *current* thread, no other thread can open it because thread *ownership* is a precondition to *unwrapping.*
- While there are *claims* (described in the next section) on an object, indicated by a *non-zero* claim count, it also may not be opened. Thus, a valid *claim* on an object can be used to justify a volatile access to an object.

1.2.4 Objects and Ownership

The ownership model in *VCC* is based on the one that is used in *Spec#* [Barnett *et al.*, 2004]. Each object has a special *owner* field that links to the object which owns it. This can be either an ordinary object or a thread. Threads are also considered to be objects. In *VCC*, only one thread is considered. This thread is called *current thread* or *me()* in *VCC.*

Objects that are closed and owned by *me()* are called *wrapped.* Objects that are open and owned by *me()* are called *mutable.* It is permitted for *me()* to modify the non-volatile fields of *mutable* objects.

Closing an object that is owned by *me* is called *wrapping*, whereas the opposite operation is called *unwrapping.* All objects that are transitively owned by a *closed* object are closed as well.

Expression Annotation Key	**Meaning**
\set in(o,S)	set membership
\old(e)	refer to pre-state
_ (unchanged e)	$e \equiv old(e)$

Table 1.0: VCC expression annotation constructs

Claims Annotaion Key	**Meaning**
o->\claim_ count	claim reference count
\make_ claim(),\destroy_ claim ()	claim creation / destruction
\claims_ obj(c,o)	assert target object of a claim
\claims(c,e)	assert the property of a claim stays
_ (unchanged e)	unchanged during claim's life time

Table 1.1: VCC claims annotation constructs

Ghost Annotation Key	**Meaning**
_ (ghost)	ghost parameter, variable, or function
_ (ghost \claim)	claim ghost parameter
_ (out x)	by-reference ghost parameter

Table 1.2: VCC ghost annotation constructs

Object Annotation Key	**Meaning**
_(invariant e)	object invariant
\wrap(o),\unwrap(o)	opening and closing objects
owner(o), owns(o)	owner and owns set
\span(o)	primitive fields of an object
\this	reference to object itself

Table 1.3: VCC object annotation constructs

Function Annotation Key	**Meaning**
_(requires e)	precondition
_(ensures e)	postcondition
_(writes o)	function writes to addresses in sets

Table 1.4: VCC function annotation constructs

Some of the key annotation constructs along with their meaning that are used in this thesis are grouped in Table 1.0, Table 1.1, Table 1.2, Table 1.3 and Table 1.4; where o is the pointer to any object, e is the expression, x is the ghost parameter. The use of these annotations will be described in the next sections.

Some key terminologies that are used in this thesis are grouped in table 1.5.

1.2.5 Concurrency

VCC allows concurrent access to data that is marked as *volatile* in the typestate. *Volatile* fields of an object can be updated concurrently by multiple threads. These fields can be changed while the object is closed.

Volatile fields can only be changed by *atomic* writes, which must respect the two-state invariant of the object. Every update on a *volatile* field must be surrounded by an *atomic-block.* This block tells the prover to check the invariants that may be affected by the update. An *atomic-block* may contain at most one update of a *volatile* of the *real code.* However, any number of updates to *volatile ghost* fields can be done as well.

1.2.6 Object States and Transition

All other fields of open objects can be changed except for *volatile* fields. Objects that are in the transitive *ownership* of a *closed* object are closed. Figure 1.3 [Hillebrand and Leinenbach, 2009] illustrates object states and their transitions during an object's life time.

After the creation of the object, it is *open* and owned by the special object *me(),*

Terminology	**Meaning(**In respect of pointer *o* of any object**)**
me	Represent the current thread
fresh(o)	Object that does not alias with any other existing object is called *fresh*
mutable(o)	Object that is not *closed*, owned by *me(), fresh* and *claim_ count* is zero is called *mutable*
wrapped(o)	*o→consistent* && *claim_ count(o)* == 0 && *owner(o)==me()*
closed(o)	Object *o* is *o→consistent* when *o* is *wrapped*; thus its invariant holds
unwrapped(o)	*!closed(o)* && *owner(o) == me()*
thread_ local(o)	Object that is known to be valid and not concurrently modified by other thread is called *thread_ local*
writable(o)	Object *o* is *writable* if it is either *mutable* or *o* is mentioned in the writes clause_ *(writeso),* of the function

Table 1.5: Some useful VCC terminology

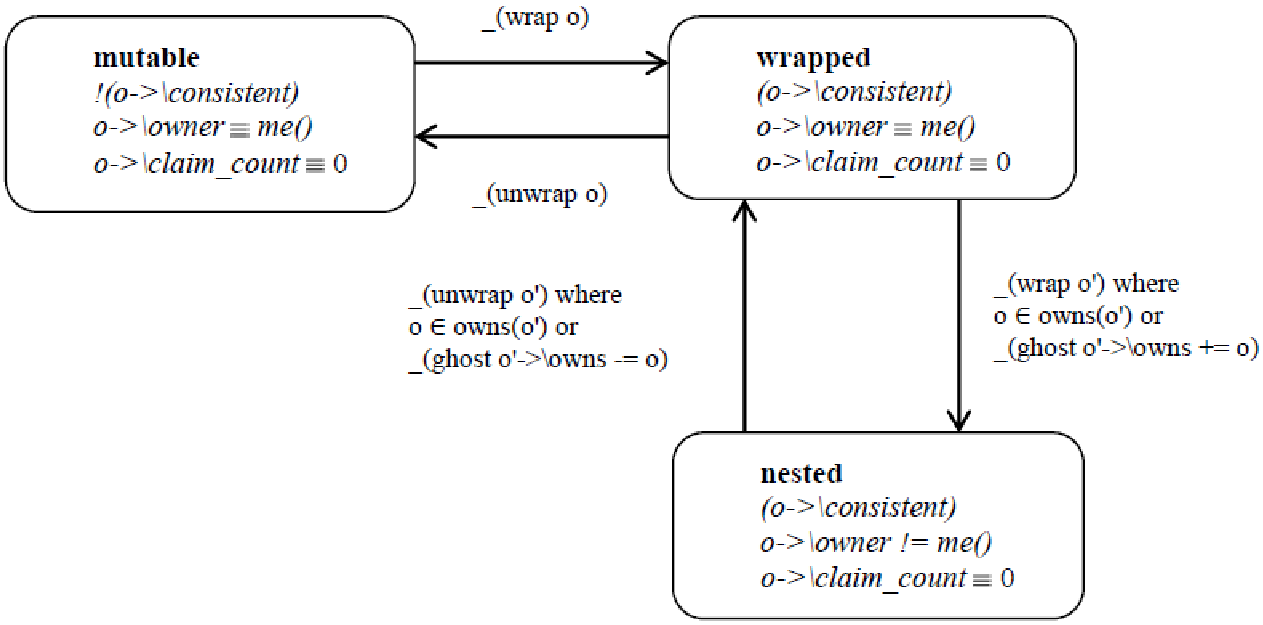


Figure 1.3: Objects states, transitions, and access permissions

representing the *current* thread. It is also considered as *fresh* object. This state is called *mutable*. In this state, the *claim count* of the object is zero, which means there are no *claims* (which will be described in later sections) on that object.

Closed objects can be added to as well as removed from another object's ownership via the following two operations respectively:

- $\_(ghost\ o' - > \backslash owns + = o)$
- $\_(ghost\ o' - > \backslash owns - = o)$

If any other objects containing the object o in its ownership set, becomes (or is already) closed, then o is called *nested*.

1.2.7 Claims

A *claim* [Hillebrand and Leinenbach, 2009] is associated with a number of *closed* objects (those objects also can be other claims). If a thread owns a claim *c* for an object *o*, it can be sure that *o* is consistent. This means that its nonvolatile fields will not change and its volatile fields will change only in ways described by the two-state invariant of the object.

Every time a *claim* on an object is created or destroyed, the object's claim count is incremented or decremented respectively. As a precondition to open an object, its *claim count* is required to be zero. This will guarantee that claimed objects remain closed.

Types which are claimable, need to be declared with the _*(claimable)* type modifier parameter. An object which is not declared with the modifier _*(claimable)* can be assumed to always have a zero *claim* count. The operation *claim(o1, . . . , oN, p)* returns a *fresh* claim referencing objects *o1* to *oN* with a claimed property *p*. While creating the claim, VCC checks the following preconditions: i) *write* permissions for the referenced objects exist, ii) the objects are *closed*, iii) the claimed property holds initially and under interference. If these preconditions are met, a *valid fresh* claim with the claimed property is returned and the *claim* counts of the objects are incremented.

The *claim* can be made or created by using the following syntax,

$\_(ghost\ \backslash claim\ c = \backslash make\_claim(\{o1, ..., oN\}, P);)$

A *claim* can be disposed by the folowing way,

$\_(ghost\ \backslash destroy\_claim(c, \{o1, ..., oN\});)$

The operation \destroy_ claim (c, {o1, . . . , oN}) destroys claim *c* and dereferences object *o1* to *oN*. For \destroy_ claim(), write permissions for the referenced objects and the *claim* must exist and as \destroy_ claim() is a special kind of *unwrap* operation, the *claim* itself must have a *claim count* of zero.

1.2.8 Atomic Blocks

Volatile fields of *closed* object can only be changed inside *atomic* blocks. This block tells the prover to check the invariants that may be affected by the update.

The *atomic* blocks are written as follows:

$\_(atomic\ c, obj)\{\};$

The keyword is followed by a list of claims and pointers. The pointers are required to point to objects that must be *closed* before the *atomic* block.

Each *atomic* block is allowed to do *at most one* atomic physical read or write operation and any number of *ghost* state updates (including creation of new claims). Within an atomic block only fields within *spans* of the objects listed may be changed. The two state invariants of those objects must be respected by the entire *atomic* transition.

1.2.8.0 Examples with atomics and claims:

Atomic blocks allow modification of the listed objects and check whether their invariants are preserved. However the update happens *at once* from the point of view of other threads.

The following is the example of the *Release()* method of a *semaphore* class where the current thread will give the *ownership* of an owned object to the semaphore. The claim c, claims that the semaphore is *consistent*.

The invariant of semaphore is as follows:

$\_(invariants == 0 \;||\; s == 1)$

$\_(invariants == 1 \Leftrightarrow \backslash mine(protected\_obj))$

The volatile field s, is updated inside the atomic block. As the claim says that the semaphore will be *consistent*, VCC checks the consistency of semaphore at the end of the *atomic block*.

```
void Release(struct Semaphore *l _(ghost \claim c))
_(always c, l->\consistent)
_(requires l->protected_obj != c)
_(writes l->protected_obj)
_(requires \wrapped(l->protected_obj))
{
_(atomic c, l) {
     l->s = 1;
     _(ghost l->\owns += l->protected_obj)}
}
```

1.3 Summary

A brief description and application of verification of concurrent programs with some related works are shown in this chapter. Also the verification tool *VCC* (which is used in this thesis) and its applications are described with examples. The next chapter will be focused on the background of semaphores & monitors and their applications.

Chapter 2

Background on Semaphores & Monitors

In this chapter, the background of *semaphores* and *monitors* are described with their applications.

2.0 Semaphore

A semaphore is a variable or abstract data type. It gives the facility to provide a simple abstraction for controlling access by multiple processes that share a common resource in a parallel programming environment. One way to use semaphore is to track the number of units of a resource that are available. On demand that number can be adjusted or wait until a unit of the resource becomes available. Semaphores which allow an arbitrary resource count are called counting semaphores, while semaphores which are restricted to the values 0 and 1 are called binary semaphores.

A semaphore is like an integer variable. However it does differ from integer variable in some ways:

- When the semaphore is created, its value can be initialized to any nonnegative integer.
- The value can be atomically incremented (increased by one) or decremented (decreased by one).
- When a thread attempts to decrement the semaphore, the thread must wait until the semaphore is positive.
- When a thread increments the semaphore, if there are other threads waiting, one of the waiting threads gets unblocked.

Some consequences that must be taken care of while using semaphore are as follows:

- Without decrementing the semaphore there is no way to know whether the thread will be blocked or not.
- When one thread increments a semaphore and another thread gets woken up, they can both run concurrently. However there is no way to know which thread will continue instantly.
- The signaler thread (thread that notifies the sleeping thread to wake up) cannot know how many threads are waiting after it sends a signal (see section 2.1.0). The number of waiting threads can be zero or one.
- The semaphore is always nonnegative.

2.0.0 Syntax

In many programming environments, an implementation of semaphores is available as part of the programming language or the operating system. The capabilities as well as syntaxes vary in different implementations. The semaphore is a shared integer variable s, manipulated by following operations:

- Constructor

  Following is the pseudo-code to create a new semaphore,

entrance = Semaphore(i) ; where i>=0. Default i=0

Semaphore(i) is a constructor. It creates and returns a new Semaphore. The initial value of the semaphore is passed as a parameter i, to the constructor.

- P & V operation

The terms P & V operations were proposed by Dijkstra [Dijkstra, 1971]. The methods are described as follows:

P(s): In this method a process decrements s (where, $s > 0$) by one. If s is 0, the process must wait until s is positive so that it can be decremented and the process can proceed.

$P(s) : \langle await(s > 0)s = s - 1; \rangle$

V(s): In this method a process increments s by one. If s is 0, and there are one or more processes waiting in *P(s)* method, one of them can complete *P(s)* and proceed.

$V(s) : \langle s = s + 1; \rangle$

Each semaphore has an associated queue of processes. The queue is usually a FIFO (first-in first-out) queue. If a process performs a P operation on a semaphore and the value of the semaphore is zero then the process is added to the semaphore's queue. When another process increments the semaphore by performing a V operation, and there are processes on the queue, one of them is removed from the queue and resumes execution. If processes have different priorities, the queue may be ordered by priority so that the highest priority process is taken from the queue first.

There are two types of semaphore used in designing synchronization algorithms.

- General Semaphore: Such a semaphore can have any values ≥ 0. Any number of processes can complete the P operation and proceed without any delay.

- Binary Semaphore (also called a mutex): Such a semaphore is confined to be either 0 or 1. One process can only proceed with the P operation at a time. If the value is 0 it has to be wait in P operation. For a binary semaphore, V can be called at any time and sets the semaphore to 1.

2.0.1 Semaphore Invariant

A semaphore satisfies the following invariants [Ben-Ari, 2006]:

- The invariant of semaphore is the value, s, should be non-negative.

  $s \geq 0$

- $s = s0 + \#V - \#P$

s0 is the initial value of the semaphore, #V is the number of V operations executed on s, and #P is the number of completed P operations executed on s.

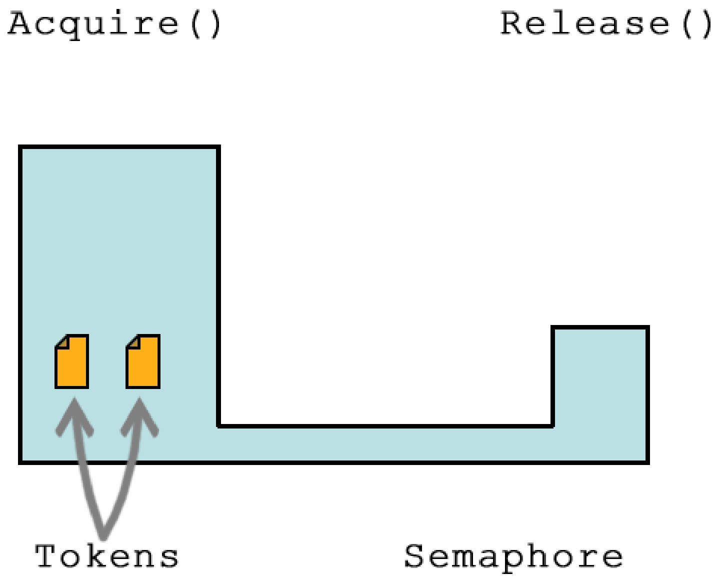


Figure 2.0: The semaphore creates two tokens

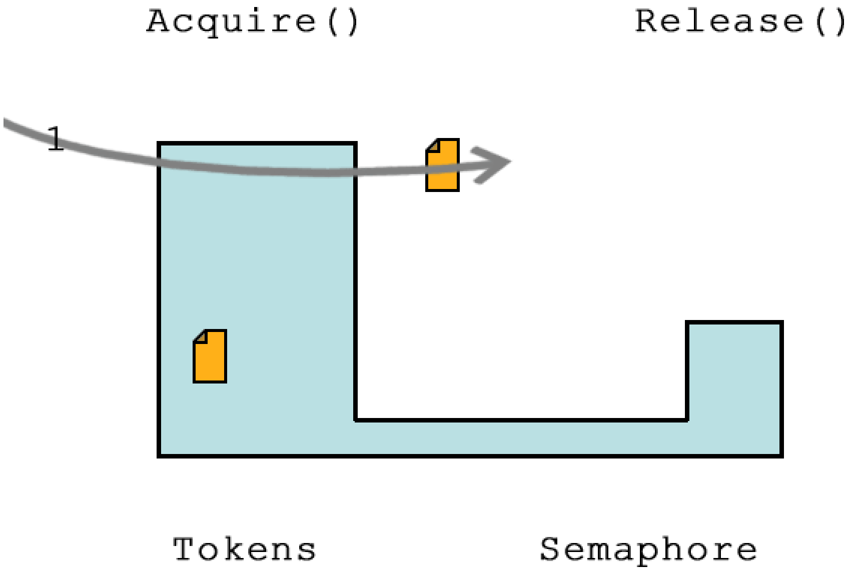


Figure 2.1: Thread1 acquires one token.

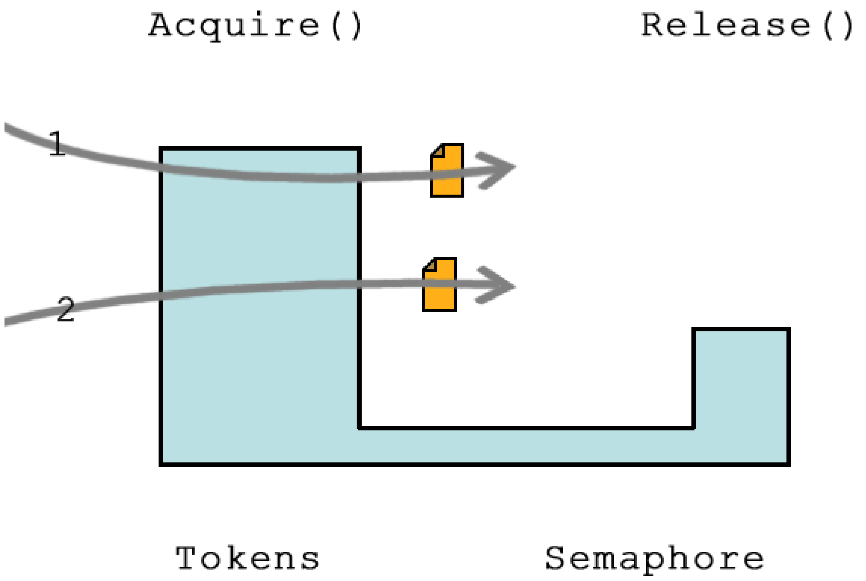


Figure 2.2: Thread2 takes another token.

2.0.2 Semaphore Illustration

The semaphore operations are described by using an illustrative example to show how the semaphore is created as well as how P & V operations are used.

In this example, the semaphore is created with two tokens using the constructer *Semaphore(i)* where $i = 2$. The two tokens are shown in Figure 2.0. Thread1 comes in and acquires the semaphore (using P operation), as shown in Figure 2.1. As there is a token available, it reduces the number of tokens and proceeds. In Figure 2.2, Thread2 comes in and acquires the last token. Now as there are no other tokens left, Thread3 will be blocked as shown in Figure 2.3. When Thread1 completes its task, it releases the token back to the semaphore (using V operation), as shown in Figure 2.4. Now Thread3 can acquires the token and proceed which is shown in Figure 2.5. In Figure 2.6,Thread2 completes its task and releases its token. Finally, Thread3

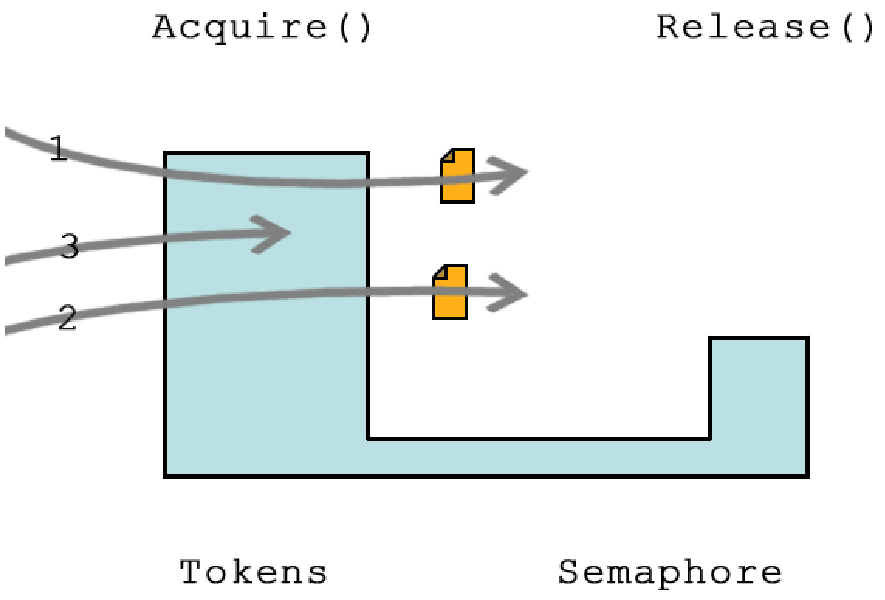


Figure 2.3: Thread3 is blocked.

completes its task and releases its token, as shown in Figure 2.7.

2.1 Monitors

Semaphores provide a simple yet powerful and flexible tool for enforcing mutual exclusion and for coordinating processes. However the *await* and *signal* (which is defined in section 2.1.0) operations may be scattered throughout a program and it is hard to see the overall effect of these operations on the semaphores that are affected by these operations.

The monitor is a programming-language structure that provides equivalent functionality to semaphores, however monitors are easier to reason about. A key point is that monitors are object oriented. The monitor structure has been implemented

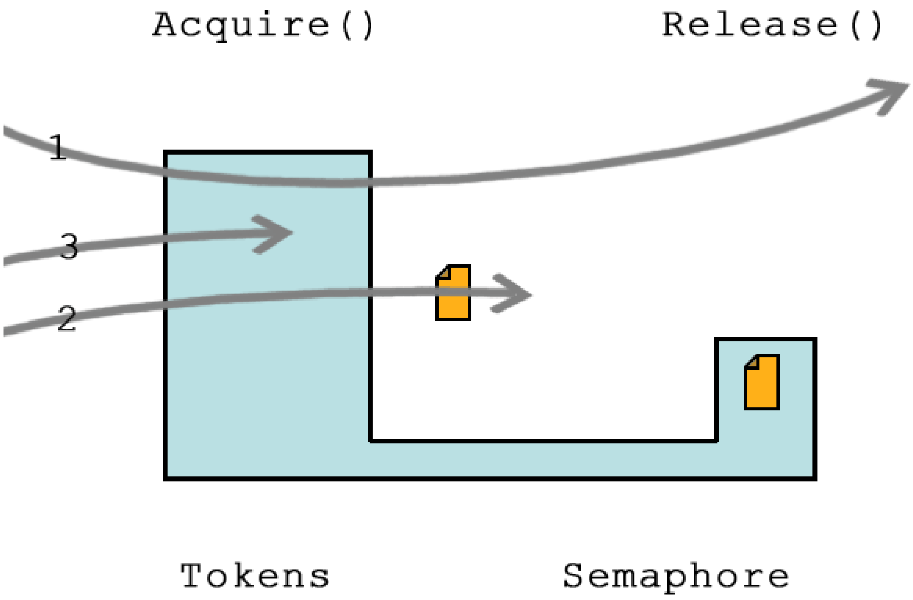


Figure 2.4: Thread1 releases one token.

in several programming language including Concurrent Pascal, Pascal-Plus, Mesa, as well as Java [Stallings, 1992].

A monitor is an object which may be used safely by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion. That is, at each point in time, at most one thread may be executing any of its methods. This mutual exclusion greatly simplifies reasoning about the implementation of monitors compared with code that may be executed in parallel.

The monitor consists of i) one or more procedures, ii) an initialization sequence, and iii) local data. The main characteristics of a monitor are the following:

- The local data variables are accessible only by the procedures of the monitor. The variables may not be accessed by any external procedures.

- The procedures of the monitor are used to enter the monitor.

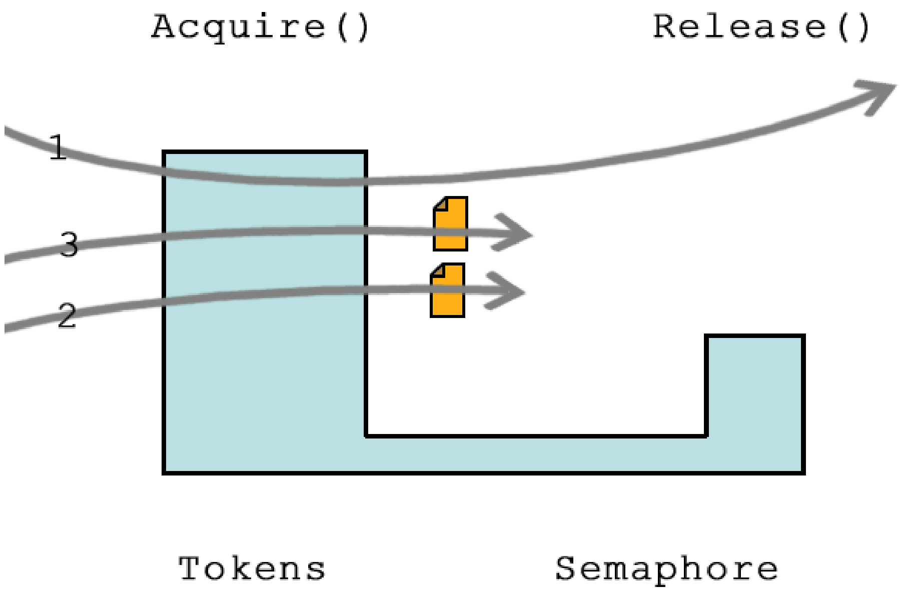


Figure 2.5: Thread3 acquires the token.

- Only one process can be executed in the monitor at a time.

The mutual exclusion ensures that one process may access the shared data structure at a time. There may be a case where one process is blocked in the monitor until some condition is satisfied. In that case the process should wait until the condition is satisfied. But also the process should leave the monitor so that some other process may enter. At a later time, when the condition is satisfied, the process may be allowed to re-enter the monitor at the same point at which it was suspended.

2.1.0 Condition Variable:

Monitors support synchronization by the use of condition variables that are contained within the monitor and associated only within the monitor. Each condition variable is associated with a condition. Threads may leave the monitor while waiting on a

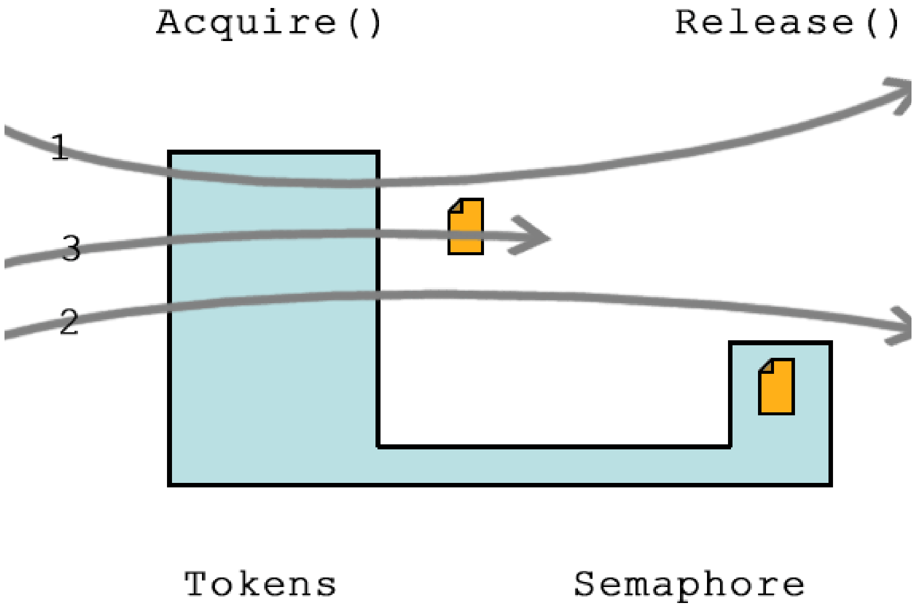


Figure 2.6: Thread2 releases the token.

condition variable for the condition to become true. Other threads may enter the monitor for execution. When the condition becomes true, the executing thread may signal the condition variable.

Blocking condition variables were first proposed by Hoare [Hoare, 1974] and Brinch Hansen [Hansen, 1973]. Monitor with blocking condition variables are often called *Hoare style* monitors.

There are two functions to operate on condition variables:

- *await(c)*: Suspends execution of the calling process on condition variable *c*. The monitor is now available for use by another process.
- *signal(c)*: Resumes execution of some process suspended on an *await* on the same condition variable. If there are several such process, one of them will be

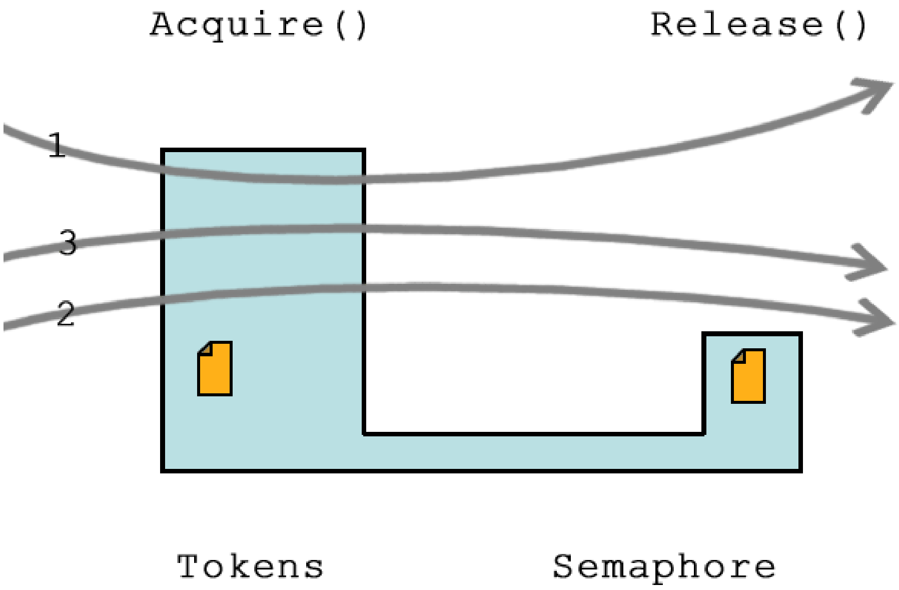


Figure 2.7: Thread3 releases the token.

chosen. If there is no such process, then the current process will proceed.

The structure of a monitor [Stallings, 1992], is shown in Figure 2.8. The process can enter the monitor by invoking any of its procedures. The monitor entry point is guarded so that only one process can get in at a time.

Once a process is in the monitor, it may temporarily suspend itself on condition c by issuing *await(c)*. It is then placed in a queue of processes waiting to re-enter the monitor when the condition changes.

If a process that is executing in the monitor causes a change in a condition, it may issue *signal(c)* on the corresponding condition variable c, which alerts the corresponding condition queue that the condition has changed to true.

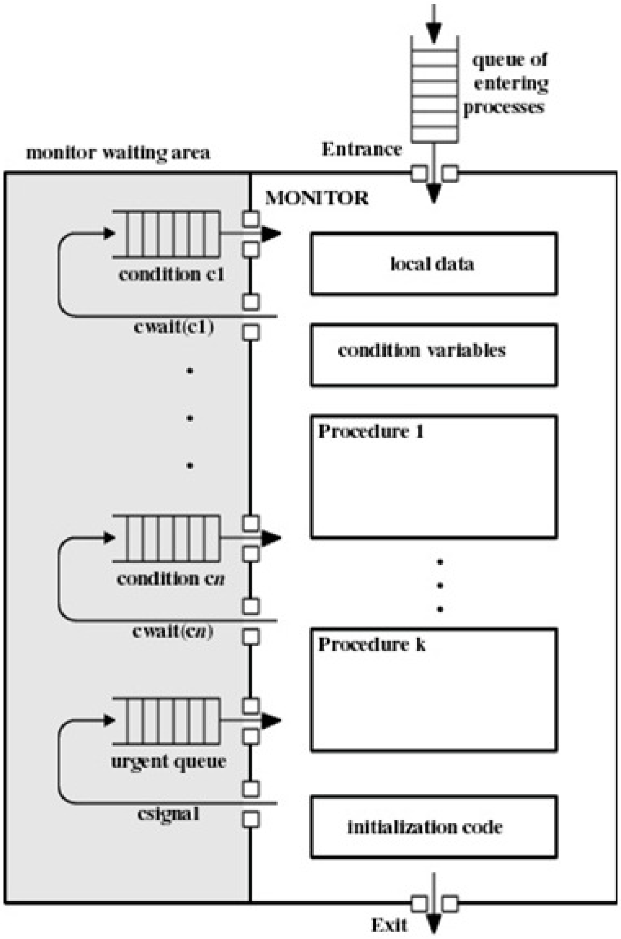


Figure 2.8: Structure of a Monitor

2.1.1 Proof Rules of Wait & Signal

The mutual exclusion on the code of a monitor ensures that procedure calls follow each other in time as in a sequential programming. An invariant *I*, is associated with the local data of a monitor to describe some condition which will be true of this data before and after every procedure call. The invariant *I*, must also be made true after initialization of the data and before every wait instruction. Otherwise the next following procedure call will not find the local data in its expected state.

With each condition variable b, an assertion B is associated, that describes the condition, under which a process waiting on b may be resumed. Since other processes may invoke a monitor procedure during a wait, a waiting process must ensure that the invariant I for the monitor is true prior to waiting.

The given proof rule for waits according to [Hoare, 1974] is:

$\{I\}$ *await(b)* $\{I \wedge B\}$; where the result of $I \wedge B$ is true iff both of the operands(I and B) are true

Since a signal can cause immediate resumption of a waiting process, the conditions $I \wedge B$ which are expected by that process must be made true before the signal. Moreover since B may be made false again by the resumed program, only I may be assumed true afterwards.

Thus the proof rule for a signal is:

$\{I \wedge B\}$ *signal(b)* $\{I\}$

2.2 Summary

The background of *semaphores* and *monitors* are described and explained with using examples. The use of condition variable in synchronization problem are also defined. In the next chapter the implementation and verification of *semaphores* and *monitors* without condition variable will be presented. All verifications that will be shown in upcoming chapters will be done using the VCC tool.

Chapter 3

Implementation and Verification Semaphores & Monitors without Conditions

In this chapter the implementation of *semaphores* and *monitor* are presented. A general approach to *monitors* specification and verification code will be proposed, which can be used for solving synchronization problems in operating systems.

3.0 Semaphore Implementation and Verification

In this section, the implementation and verification of semaphores are presented.

The annotated declaration of the semaphore data structure is shown in Listing 3.0. The data structure *Semaphore* contains a single volatile implementation variable called *s*. As it is a binary Semaphore, the value of *s* is confined to be either 0 or 1.

The semaphore contains one ghost variable, a generic object pointer *protected_ obj.*

The field *protected_ obj* is used for the identification of object that is protected by the semaphore. The pointer to the object that is shared by multiple threads is initialized to the *protected_ obj.* After initialization the semaphore will own the *protected_ obj.* A thread that needs to update the shared object, has to take ownership from the semaphore by acquiring from the semaphore (which is done by decrementing the implementation variable *s*). After updating the object ownership is given back to the semaphore from owning thread (that owns shared object) by releasing it to the semaphore (which is done by incrementing the implementation variable s).

```
_(claimable) _(volatile_owns) typedef struct _Semaphore {
    volatile int s;
    _(ghost \object protected_obj;)
    _(invariant s==0 || s==1 )
    _(invariant s == 1 <==> \mine(protected_obj))
}Semaphore;
```

Listing 3.0: The Semaphore Structure

3.0.0 Invariant

As it is a binary semaphore, the first invariant of the semaphore is that the implementation variable, *s*, can be either 0 or 1.

_(*invariant s* == 0 || *s* == 1)

The next invariant says that, if the value of the semaphore is set to 1 then the object protected by the semaphore is owned by the semaphore. The ownership will change only if any thread needs to access the object. Therefore, the thread needs to acquire the ownership from the semaphore.

$$\_(invariant\ s == 1 \iff \backslash mine(protected\_obj))$$

3.0.1 Initialization

From the implementation point of view, initializing a semaphore simply means to set its implementation variable s to 1. From a specification point of view, some extra tasks have to be done. The *initializeSemaphore* function ensures that the semaphore is *wrapped* and the *protected_obj* field is set. As the implementation variable of semaphore is set, semaphore owns the *protected_obj.* Any thread that updates the object is required to obtain ownership of it from the semaphore.

The initialization of semaphore is given in Listing 3.1.

```
void initializeSemaphore(Semaphore *sem _(ghost \object obj))
_(writes \span(sem))
_(writes obj)
_(requires \wrapped(obj))
_(ensures \wrapped(sem) && sem->protected_obj == obj)
_(ensures sem->s == 1)
{
    sem->s = 1;
    _(ghost {
        sem->protected_obj = obj;
        sem->\owns = {obj};
    _(wrap sem)
    })
}
```

Listing 3.1: Semaphore Initialization

3.0.2 Implementation of Semaphore *acquire* Method

Acquiring from a semaphore proceeds in two phases: i) wait until the variable s is set to 0 and ii) after the s has been set to 0, transfer ownership of the protected object. No new thread can get the access to the semaphore now. The *P()* method of semaphore is named as *acquire()* and the implementation variable s is set to 0 instead of decrementing.

Listing 3.2 is the annotated implementation of the function *acquire()* which is used to transfer the ownership of *protected_ object* to current thread from the semaphore. The claim c is a ghost parameter which guarantees the semaphore to be consistent using the _*(always)* clause. The function also ensures the caller that, when the function returns, the *protected_ object* is *wrapped* and *fresh*. Thus the *protected_ obj* is writable at the end of this function.

The specification ensures that after the call, the semaphore will have given up the ownership of the *protected_ obj* to the thread. Thus the thread can acquire the ownership of the *protected_ obj* from the semaphore.

```
void acquire(Semaphore *sem _(ghost \claim c))
_(always c, sem->\consistent)
_(ensures \wrapped(sem->protected_obj) && \fresh(sem->protected_obj))
{
    int stop = 0;
    do {
        _(atomic c, sem) {
            stop = InterlockedCompareExchange(&sem->s, 0, 1) == 1;
            _(ghost if (stop) sem->\owns -= sem->protected_obj)
        }
    }while (!stop);
}
```

Listing 3.2: Semaphore Acquire Method

The *InterlockedCompareAndExchange()* function is a compiler built-in, which on the *x86/x64* hardware translates to *cmpxchg* assembly instruction [Hillebrand and Leinenbach, 2009]. It takes a memory location and two values. If the memory location contains the first value, then it is replaced with the second one and the old value is returned. The entire operation is performed atomically.

The *InterlockedCompareExchange()* implementation shown in Listing 3.3 is used in this thesis only for the verification purpose. When the code is compiled, the function is translated to a single instruction.

```
_(atomic_inline) int InterlockedCompareExchange(volatile long
*Destination, long Exchange, long Comparand) {
        if (*Destination == Comparand) {
            *Destination = Exchange;
            return Comparand;
        } else {
            return *Destination;
        }
    }
```

Listing 3.3: InterlockedCompareExchange Method

3.0.3 Implementation of Semaphore *release* Method

Releasing the object to a semaphore also proceeds in two phases, i) setting the implementation variable s to 1 and ii) giving back the ownership of the object to the semaphore. Afterward another thread can get access to the semaphore. The *V()* method of the semaphore is named *release()*.

The release function is shown in Listing 3.4.

```
void release(Semaphore *sem _(ghost \claim c))
_(always c, sem->\consistent)
_(requires sem->protected_obj != c)
_(writes sem->protected_obj)
_(requires \wrapped(sem->protected_obj))
{
    _(atomic c, sem) {
        sem->s = 1;
        _(ghost sem->\owns += sem->protected_obj)
    }
}
```

Listing 3.4: Semaphore Release Method

The ghost claim parameter *c*, is passed to the *release()* function. The claim guarantees that the semaphore is consistent. It is required that the *protected_object* is *wrapped*. It is also required that the claim is not the *protected_object*, otherwise it could not ensure the claim is *wrapped* after the call.

After completing the *release()* function, the thread will have given the ownership of the *protected_obj* back to the semaphore.

3.1 Monitor Implementation and Verification

In this section the implementation and verification of monitor methods (without any condition variables) is presented. A thread will enter the monitor to access the shared object. Synchronization is done using the semaphore object. Each monitor has one pointer to the semaphore object. The thread needs to wait till the semaphore is available. Once the semaphore is available it can proceed to the monitor using its enter function. Only one thread will enter the monitor at a time. Thread will give the ownership back to the semaphore and leave the monitor using the exit function.

To bind together the semaphore with any monitored object, the monitored object is divided into two separate objects in this thesis: i) an object and ii) a monitor. The invariant of the Monitor requires that the semaphore be owned by the monitor. Any claim that claims the monitor is consistent will also claim that the semaphore is consistent. A second invariant requires that the semaphore protects the object.

In this thesis, monitor methods (entry and exit) are implemented as *enterMonitor()* and *exitMonitor()*.

To get access to the object each thread needs to enter the monitor using *enterMonitor()* function. This function ensures that the *protected_obj* (shared object) of the *entrance* semaphore is *wrapped*. The ownership of the object is transfered to the thread using this method.

To give ownership back to the semaphore, the *exitMonitor()* function is used.

A thread needs to use the *enterMonitor()* function to get access of the shared object and the *exitMonitor()* function to release the object. Only one thread can be executed in the monitor at a time which is satisfied by using the *entrance* semaphore.

To use the monitor functions the monitored object will own the entrance semaphore. Using the entrance semaphore the thread can enter and exit the monitor. The functions will be described with the example later.

3.2 Time of Day Example

The SI (International System of Units) based unit for time is the second. The larger units such as, minute and hour, are defined from the second; i) the minute is unit time equal to 60 seconds and ii) the hour is unit time equal to 60 minutes.

To get the correct time these three units have to be synchronized. For an example, the current time is 11 hours and 59 minutes. A thread T1 reads the value of hour and gets 11. A short time later, the time turns into 12 hours and 0 minutes. Thread T1 now reads the value of minute and gets the value 0. So T1 reads the time as 11 hours and 0 minutes. The proposed monitor synchronization methodology will be demonstrated by solving this problem.

To bind together the entrance semaphore with the Time object, the monitored object is divided into two separate objects: i) *Time* & ii) *TimeMonitor*.

3.2.0 The *Time* Structure

The structure *Time* has three implementation variables *hr*, *min* and *sec* representing three parts of time of a day.

The *Time* structure is shown in Listing 3.5.

```
typedef struct _TimeOfDay{
    int hr;
    int min;
    int sec;
    _(invariant 0 <= sec && sec < 60
            && 0 <= min && min < 60
            && 0 <= hr && hr < 24 )

}Time;
```

Listing 3.5: The Time Structure

3.2.0.0 Invariant of *Time* Structure

The invariant of the *Time* structure is straight forward. The variables *sec* and *min* are greater than and equal to 0 and less than 60. The variable *hr* is greater than and equal to 0 and less than 24. The invariant is shown in Listing 3.6.

```
    _(invariant 0 <= sec && sec < 60
            && 0 <= min && min < 60
            && 0 <= hr && hr < 24 )
```

Listing 3.6: Invariant of Time object

3.2.1 The *TimeMonitor* Structure

In this example the global structure *TimeMonitor* is used to own other objects. Each *TimeMonitor* object contains a *Time* object, *t,* and a Semaphore object, *entrance.* *TimeMonitor* owns the *entrance* semaphore.

The *TimeMonitor* structure is shown in Listing 3.7.

```
_(claimable) typedef struct _Time_Monitor
{ Time t ;
    Semaphore entrance;
    _(invariant \mine(&entrance))
    _(invariant entrance.protected_obj == &t)
}TimeMonitor;
```

Listing 3.7: The Time Monitor Structure

3.2.1.0 Invariant of *TimeMonitor* Structure

The invariants of *TimeMonitor* require that the *entrance* semaphore be owned by the monitor. As a result, any claim that claims that the *TimeMonitor* object is consistent will also claim that the *entrance* semaphore is consistent. The second invariant requires that the *entrance* semaphore protects the *Time* object.

```
    _(invariant \mine(&entrance))
    _(invariant entrance.protected_obj == &t)
```

Listing 3.8: Invariant of TimeMonitor object

3.2.1.1 Implementation of monitor *enterMonitor* method

Listing 3.9 is the annotated implementation of the function *enterMonitor(),* which is used to enter to the monitor. The claim *c* is the ghost parameter which is required and ensured to be valid and guarantees the consistency of the *monitor.* The function also ensures that, when the function returns, the *protected_ object* is *wrapped* and *fresh*. Thus the protected_obj is writable at the end of this function.

```
void enterMonitor(TimeMonitor *monitor _(ghost \claim c))
_(always c, (& monitor->entrance)->\consistent)
_(ensures \wrapped(monitor->entrance.protected_obj) )
_(ensures \fresh(monitor->entrance.protected_obj) )
{
    acquire(& monitor->entrance _(ghost c));
}
```

Listing 3.9: Enter function of Monitor

3.2.1.2 Implementation of monitor *exitMonitor* method

In Listing 3.10 the annotated imlementation and specification of the *exitMonitor()* function is given. The ghost parameter claim *c* is passed to the function which guarantees the consistency of the *monitor*. The function requires the claim to be *wrapped*. It is also required that the claim is not the *protected_ object*, otherwise it couldn't ensure that the claim is *wrapped* after the call.

```
void exitMonitor(TimeMonitor *monitor _(ghost \claim c))
_(always c, (& monitor->entrance)->\consistent)
_(requires \wrapped(monitor->entrance.protected_obj))
_(requires monitor->entrance.protected_obj != c)
_(writes monitor->entrance.protected_obj)
{
    release( & monitor->entrance _(ghost c));
}
```

Listing 3.10: Exit function of Monitor

3.2.2 Implementation & Specification of *tick* method

The *tick* routine is used to update the time in each sec. It receives two parameters; i) *TimeMonitor, monitor* and ii) *ghost* claim parameter, *c*. The specification states that the claim *c* requires and ensures to be valid and guarantees the consistency of the *monitor.*

To update the time, any thread needs to enter the monitor first. Once the thread gets access to the monitor it will *unwrap* the object *t*, change it and then *wrap* it again. Later it will exit from the monitor.

The *tick* routine is shown in Listing 3.11.

```
void tick(TimeMonitor *monitor _(ghost \claim c))
_(always c, monitor->\consistent)
{
    enterMonitor(monitor _(ghost c));
    _(unwrap &monitor->t);
        monitor->t.sec += 1;
        monitor->t.min += monitor->t.sec/60;
        monitor->t.hr += monitor->t.min/60;
        monitor->t.sec = monitor->t.sec % 60;
        monitor->t.min = monitor->t.min % 60;
        monitor->t.hr = monitor->t.hr % 24;
    _(wrap &monitor->t);
    exitMonitor(monitor _(ghost c));
}
```

Listing 3.11: The tick method

3.2.3 Implementation & Specification of *get* method

The *get* method is used to retrieve the time. It also receives two parameters similar to those of tick; i) *TimeMonitor, monitor* and ii) *ghost* claim parameter, *c.* The specification states that the claim *c* requires and ensures to be valid and guarantees the consistency of the *monitor.*

To get the current time, any thread needs to enter the monitor first. Once the thread gets access to the monitor, it will *unwrap* the object *t*, put the values (*hr*, *min* and *sec*) in the passed in array (*time*) and then *wrap* the monitor again. Later it will exit from the monitor.

The *get* routine is shown in Listing 3.12.

```
void get(int time[], TimeMonitor *monitor _(ghost \claim c))
_(writes \array_range(time,3) )
_(always c, monitor->\consistent)
{
        enterMonitor(monitor _(ghost c));
        _(unwrap &monitor->t);
                time[0] = monitor->t.sec;
                time[1] = monitor->t.min;
                time[2] = monitor->t.hr;
        _(wrap &monitor->t);
        exitMonitor(monitor _(ghost c));
}
```

Listing 3.12: The get method

3.2.4 Pthreads in *TimeOfDay* Example

Pthreads [Barney, 2011] are used in synchronization methods of real time operating systems like *RTAI* [Bucher *et al.*, 2003], FreeRTOS [Barry, 2004] etc. which are written in C. To test the *TimeOfDay* example, *pthreads* is used to create multiple threads that share a time monitor.

The *pthread_create()* function only allows one argument to pass to the starting thread. In this example, each thread takes two parameters; i) a pointer to the *TimeMonitor* object and ii) a pointer to the *claim* object.

To pass these two parameters, the *ThreadData* structure is implemented which contains a pointer to the *TimeMonitor* object *monitor*. It contains a claim, *claim* as well.

The pointer to *ThreadData* structure is passed to the starting thread by the *pthread_create()* function.

The *ThreadData* structure is shown in Listing 4.16.

```
typedef struct _thread_data{
    TimeMonitor * monitor;
    int count;
    bool spin;
    _(ghost \claim claim;)
    _(invariant \claims_object(claim, monitor))
}ThreadData ;
```

Listing 3.13: The ThreadData Structure

To start the tick and get threads, *start_tick()* and *start_get()* functions are used respectively. The specification of these two methods requires the *ThreadData* object *param* to be *wrapped* and also maintains the consistency of the *TimeMonitor* object (using the claim pointer *claim*). The *start_tick()* and *start_get()* routines are shown in Listing 3.14 and Listing 3.15 respectively.

```
void *start_tick (void *param)
_(requires \wrapped(((ThreadData *) param)))
_(always ((ThreadData *) param)->claim, ((ThreadData *) param)->monitor↩
↪->\consistent)
{
    int i ;
    TimeMonitor *monitor = ((ThreadData *) param)->monitor ;
    #ifndef VERIFY
        while( ! go ) {}
    #endif
    for (i=0 ; i < NUM_OF_TICKS ; i++)
    {
        tick(monitor _(ghost ((ThreadData *) param)->claim));
    }
    return 0 ;
}
```

Listing 3.14: Start Routine of tick

```
void *start_get (TimeMonitor *monitor _(ghost \claim c))
_(always c, monitor->\consistent )
{
    int time[3],i;
    #ifndef VERIFY
```

#of *tick* threads	**# of *get* threads**	**monitor.t.sec**	**monitor.t.min**	**monitor.t.hr**
20	20	20	33	3
45	45	0	0	2
100	100	40	46	17

Table 3.0: Results derived from TimeOfDay example - Value of monitor.t

```
        while( ! go ) {}
    #endif
    for (i=0 ; i < NUM_OF_GETS ; i++)
    _(writes \array_range(time,3) )
    {
        get(time, monitor _(ghost c) );
        if( time[0] >= 60 || time[1] >= 60 || time[2] >= 24 )
        {
            #ifndef VERIFY
                printf("Get␣FAILED") ;
            #endif
        }
    }
}
```

Listing 3.15: Start Routine of get

The full implementation of *timeOfDay* is added in the apendix. The results of testing *timeOfDay* example is splitted into two tables shown in Table 3.0 and Table 3.1.

An oracle function is written for the testing purpose. The number of created threads are defined by the number of *tick* threads and the number of *get* threads (In Table 3.0). Each thread performs a number of ticks and gets respectively (e.g. 100,000 ticks & 100,000 gets).

The value of time after all the ticks and gets for each various cases are given in

#of *tick* threads	**# of *get* threads**	**sec**	**min**	**hr**
20	20	20	33	3
45	45	0	0	2
100	100	40	46	17

Table 3.1: Results derived from TimeOfDay example - Expected value evaluated by countSum

Table 3.0. The total number of ticks are counted using the *countSum* variable. It is used to evaluate the expected time (sec=countSum%60, min=(countSum% 3600)/60, hr=(countSum/3600)%24). The result is shown in Table 3.1. After comaparing Table 3.0 and Table 3.1 it can be derived that the program evaluated the expected time.

3.3 Summary

The implementation of semaphores and monitors with their verification (using *VCC* tool) are shown in this chapter. The *timeOfDay* example is used to demonstrate the general approach (that is proposed in this chapter) to *monitor* specification and verification. Codes are compiled using *VCC* tool. The results are added in the chapter.

Chapter 4

Implementation and Verification Semaphores & Monitors with Conditions

The producer-consumer problem (which is also known as the bounded-buffer problem) is a classical multi-process synchronization problem in computer science. In this chapter the implementation and specification of the producer-consumer problem has been verified using the proposed *monitor* specification code (explained in the previous chapter).

4.0 Background

The threads of a multithreaded programs follow many patterns. In the common pattern some threads are *producers* and some are *consumers*. *Producers* create items of some kind and add them to a data structure. On the other hand, *consumers* remove the items and process them.

The producer-consumer problem describes two categories of processes: *producers* and *consumers.* The threads share a common and fixed-size buffer. A *producer*'s job is to generate a piece of data and put it into the buffer. A *consumer* consumes the data (also removing it from the buffer) one piece at a time. The problem is to make sure that the *producers* cannot add data into the buffer when the buffer is *full* and that the *consumers* cannot remove data from an *empty* buffer.

4.0.0 The Producer/Consumer Bounded-Buffer solution using *Monitor*

There are many solutions for this problem based on different synchonization mechanisms. In this chapter, a solution to the *Producer/Consumer Bounder Buffer* problem using a monitor is described, implemented, and verified. In Algorithm 4.0 the solution is presented as was proposed by C. A. R. Hoare [Hoare, 1974] .

The monitor module, *boundedbuffe*r, controls the buffer used to store and retrieve characters. In this example, there are two conditions that for which threads may need to wait. The monitor includes two condition variables: *notfull* is true when there is room to add at least one element to the buffer, and *notempty* is true when there is at least one element in the buffer.

A producer can add an element to the buffer only by means of the procedure *append* inside the monitor. However a procedure does not immediately access the buffer. The procedure first checks the condition variable *notfull* to determine whether there is space available in the buffer. If not, the process executing the monitor is suspended on that condition variable. Some other process (producer or consumer) may now enter the monitor. Later, when the buffer is no longer full, the suspended process may be removed from the queue, reactivated, and eventually the processing is resumed. After placing an element in the buffer, the process signals the *notempty* condition.

A similar description can be made of the *consumer* processes. A consumer can remove an element from the buffer only by means of the procedure *remove* inside the monitor. The procedure does not immediately access the buffer. The procedure first checks the condition *notempty* to determine whether it is removing an element from an empty buffer. If the buffer is empty, the process executing the monitor is suspended on that condition. Some other process (producer or consumer) may now enter the monitor. Later, when the buffer is *not-empty*, the suspended process may be removed from the queue, reactivated, and eventually the processing is resumed. After removing an element from the buffer, the process signals the *notfull* condition.

The pseudo code algorithm of the described solution of Producer/Consumer Bounded-Buffer problem is shown in Algorithm 4.0.

4.1 The Producer/Consumer BoundedBuffer Implementation and Specification

To bind together the entrance semaphore with the *buffer* object, the monitored object is divided into two separate objects: i) *MonitoredBuffer* & ii) *BufferMonitor*.

```
boundedbuffer:monitor
begin buffer:array0..N-1of portion;
     lastpointer:0..N-1;
     count:0..N;
    notempty,notfull:condition;
procedure append(x:portion);
     begin
          if count=N then notfull.wait;
          note 0<count<N;
          buffer[lastpointer]:=x;
          lastpointer:=lastpointer+1;
          count:=count+1;
          notempty.signal
     end append;
procedure remove(resultx:portion);
     begin
          if count=0 then notempty.wait;
          note 0<count<N;
          x:=buffer[lastpointer-count];
          notfull.signal
     end remove;
count:=0;lastpointer:=0;
end boundedbuffer;
```

Algorithm 4.0: Pseudo code solution of Producer/Consumer Bounded-Buffer problem

4.1.0 The *MonitoredBuffer* Structure

The boundedbuffer structure is named *MonitoredBuffer* in this example. The *MonitoredBuffer* contains the buffer array, *buffer,* of *CAPACITY* items. The field *size* defines the size of the buffer. The field *head* defines the head of the *FIFO* queue.

The *MonitoredBuffer* also has two unsigned variables *notFullCount* & *notEmptyCount* to count threads that are waiting on *notFull* condition variable and *notEmpty* condition variable respectively. The *MonitoredBuffer* structure is shown in Listing 4.0.

```
typedef struct _MonitoredBuffer{
    int buffer[CAPACITY];
    int size ;
    int head ;
    _(invariant 0 <= size && size <= CAPACITY)
    _(invariant 0 <= head && head < CAPACITY)
    unsigned notFullCount;
    unsigned notEmptyCount;
}MonitoredBuffer;
```

Listing 4.0: The MonitoredBuffer Structure

4.1.0.0 Invariant of *MonitoredBuffer*

The invariants of the *MonitoredBuffer* structure state that the size of the *MonitoredBuffer* and the value of field *head* to be in the range from *0* to *CAPACITY.*

$\_(invariant\ 0 <= size\ \&\&\ size <= CAPACITY)$

$\_(invariant\ 0 <= head\ \&\&\ head < CAPACITY)$

4.1.1 Implementation & Specification of *Condition* Semaphores

In order to maintain the synchronization, two condition semaphores are used:

- *NotEmptySemaphore*: Used by the consumer thread to suspend itself until the buffer is not empty.

- *NotFullSemaphore*: Used by the producer to suspend itself until the buffer is not full.

The semaphor's implementations are similar to the semaphore described in the previous chapter. However they differ in some ways as follows:

- Both of the semaphores have a pointer to the *MonitoredBuffer* object, *buff,* rather than a generic object pointer.

- The invariant of *notFullSemaphore* states that when the semaphore is 1 the size of the *buff* will be less than the CAPACITY.
- The invariant of *notEmptySemaphore* states that when the semaphore is 1 the size of the *buff* will be greater than zero.

The implementation of acquire methods (which is described at section 3.0.2) and the condition semaphores is based on test-and-set instruction. If multiple processes access the same memory, and at some moment if a process is performing a test-and-set, no other process can begin another test-and-set until the first process is done. This implementation of acquire methods and condition semaphores can be used in multi-CPU hardware but not in single process systems.

The *NotEmptySemaphore* & *NotFullSemaphore* structures are shown in Listing 4.1 & 4.2.

```
_(claimable) _(volatile_owns) typedef struct _NotEmptySemaphore {
    volatile int s;
    MonitoredBuffer * buff;
    _(invariant s==0 || s==1 )
    _(invariant s == 1 <==> \mine(buff))
    _(invariant s == 1 ==> buff->size > 0)
} NotEmptySemaphore;
```

Listing 4.1: The NotEmptySemaphore Structure

```
_(claimable) _(volatile_owns) typedef struct _NotFullSemaphore {
    volatile int s;
    MonitoredBuffer * buff;
    _(invariant s==0 || s==1 )
    _(invariant s == 1 <==> \mine(buff))
    _(invariant s == 1 ==> buff->size < CAPACITY)
}NotFullSemaphore;
```

Listing 4.2: The NotFullSemaphore Structure

4.1.1.0 Invariant of Condition Semaphores

The first invariant of the *condition* semaphores (*NotEmptySemaphore & NotFullSemaphore*) is the implementation variable *s*, can be either *0* or *1*.

$\_(invariants == 0 \mid\mid s == 1)$

The next invariant says that, if *s* is set to the value *1* the *MonitoredBuffer* object *buff*, is protected by the *condition semaphore* (*NotEmptySemaphore & NotFullSemaphore*) itself.

$\_(invariants == 1 \Longleftrightarrow \backslash mine(buff))$

The last invariant of the *condition semaphore* varies with its structure. The last invariant of *NotEmptySemaphore* ensures that whenever *s* is set to the default value (in this case 1), the *size* of the *buff* (which is owned by the *NotEmptySemaphore*) is greater than zero.

$\_(invariants == 1 \Rightarrow buff \rightarrow size > 0)$

However, the last invariant of *NotFullSemaphore* ensures that whenever *s* is set to the default value (in this case 1), the *size* of the *buff* (which is owned by the *NotFullSemaphore*) is less than the *CAPACITY*.

$\_(invariants == 1 \Rightarrow buff \rightarrow size < CAPACITY)$

4.1.1.1 Implementation & Specification of Condition Semaphore *Acquire()*

On *await* operation the *monitor* gives up the ownership of the *MonitoredBuffer* object, *buff* and the *condition* semaphore is given ownership of *buff*. To obtain ownership from the condition variable, its acquire function is used.

```
void notEmptySemaphoreAcquire(NotEmptySemaphore *sem _(ghost \claim c))
_(always c, sem->\consistent)
_(ensures \wrapped(sem->buff) && \fresh(sem->buff))
```

```
_(ensures sem->buff->size > 0)
{
    int stop = 0;
    do {
        _(atomic c, sem) {
            stop = InterlockedCompareExchange(&sem->s, 0, 1) == 1;
            _(ghost if (stop) sem->\owns -= sem->buff)
        }
    } while (!stop);
}
```

Listing 4.3: Acquire method of notEmptySemaphore

```
void notFullSemaphoreAcquire(NotFullSemaphore *sem _(ghost \claim c))
_(always c, sem->\consistent)
_(ensures \wrapped(sem->buff) && \fresh(sem->buff))
_(ensures sem->buff->size < CAPACITY)
{
    int stop = 0;
    do {
        _(atomic c, sem) {
            stop = InterlockedCompareExchange(&sem->s, 0, 1) == 1;
            _(ghost if (stop) sem->\owns -= sem->buff)
        }
    }while (!stop);
 }
```

Listing 4.4: Acquire method of notFullSemaphore

Listing 4.3 & Listing 4.4 represent the annotated implementations of the *acquire* functions for the *condition* semaphores (*NotEmptySemaphore* & *NotFullSemaphore* respectively).

Both of the functions are similar to the semaphore *acquire()* (which is given in Listing 3.2). However, the *notEmptySemaphoreAcquire* ensures the *size* of the *buff* to be greater than zero and the *notFullSemaphoreAcquire* ensures the *size* of the *buff* to be less than *CAPACITY.* The reason that these postconditions of both acquire functions verifies is that the invariant of each condition semaphore ensures that the appropriate assertion be true.

4.1.1.2 Implementation & Specification of Condition Semaphore *Release()*

On a *signal* operation, the *condition* variable gives up the ownership of the *buff* and the *monitor* variable owns the *buff.* Signalling threads use the condition variable's, release function to give ownership of buff to the semaphore.

```
void notEmptySemaphoreRelease(NotEmptySemaphore *sem _(ghost \claim c))
_(always c, sem->\consistent)
_(requires sem->buff->size > 0)
_(writes sem->buff)
_(requires \wrapped(sem->buff))
{
    _(atomic c, sem) {
        sem->s = 1;
        _(ghost sem->\owns += sem->buff)
    }
}
```

Listing 4.5: Release method of notEmptySemaphore

```
void notFullSempahoreRelease(NotFullSemaphore *sem _(ghost \claim c))
_(always c, sem->\consistent)
_(requires sem->buff->size < CAPACITY)
_(writes sem->buff)
_(requires \wrapped(sem->buff))
{
    _(atomic c, sem) {
        sem->s = 1;
        _(ghost sem->\owns += sem->buff)
    }
}
```

Listing 4.6: Release method of notFullSemaphore

Listing 4.5 & Listing 4.6 represent the annotated implementation of the *release* functions for the *condition* semaphores (*NotEmptySemaphore* & *NotFullSemaphore* respectively).

Both of the functions are similar to the semaphore *release()* (which is given in Listing 3.4). However, the *NotEmptySempahoreRelease* requires the *size* of the buffer to be greater than zero and the *NotFullSempahoreRelease* requires the *size* of the buffer to be less than *CAPACITY*.

4.1.2 The *BufferMonitor* Structure

A global structure *BufferMonitor*, is used to own other objects. Each *BufferMonitor* object contains a *MonitoredBuffer* object, *theBuffer,* a *Semaphore* object, *entrance,* a *NotFullSemaphore* object, *notFullQ,* and a *NotEmptySemaphore* object, *notEmptyQ.* *BufferMonitor* owns all the semaphore objects. The operations on the monitor include *enterMonitor* and *exitMonitor* (described in the Chapter 3), which wrap calls to *acquire* and *release* to obtain and relinquish ownership of the *buffer.*

The *BufferMonitor* structure is shown in Listing 4.7.

```
_(claimable) typedef struct _BufferMonitor
{
    MonitoredBuffer theBuffer;
    Semaphore entrance;

    _(invariant \mine(& notFullQ))
    _(invariant notFullQ.buff == &theBuffer)

    NotFullSemaphore notFullQ;
    NotEmptySemaphore notEmptyQ;

    _(invariant \mine(& notEmptyQ))
    _(invariant notEmptyQ.buff == &theBuffer)

    _(invariant \mine(& entrance))
    _(invariant entrance.protected_obj == &theBuffer)
}BufferMonitor;
```

Listing 4.7: The BufferMonitor Structure

4.1.2.0 Invariant of the *BufferMonitor* Structure

The invariants of *BufferMonitor* requires that the semaphores (*entrance*, *notEmptyQ* & *notFullQ*) be owned by the monitor. As a result, any claim that claims that the *BufferMonitor* object is consistent will also claim that all semaphores are consistent. The invariant also requires that semaphores (*entrance*, *notEmptyQ* & *notFullQ*) protect the *MonitoredBuffer* object.

_(*invariant* \mine(&*notFullQ*))

_(*invariant notFullQ.buff* == &*theBuffer*)

_(*invariant* \mine(&*notEmptyQ*))

_(*invariant notEmptyQ.buff* == &*theBuffer*)

_(*invariant* \mine(&*entrance*))

_(*invariant entrance.protected_obj* == &*theBuffer*)

4.1.2.1 Implementation of *await* & *signal*

Because of the additional clause in the invariant of *NotEmptySemaphore*, the condition $buff- > size > 0$ is the precondition of its release operation and a postcondition of its acquire function (for *NotFullSemaphore,* condition $buff- > size < CAPACITY$ is used).

These preconditions and postconditions are inherited by the monitor level operations from the semaphore level operations; i) *await (awaitNotEmptyCondition & awaitNotFullCondition)* and ii) *signal (signalNotEmptyCondition & signalNotEmpty-*

Condition). In the next section the implementation and specification of these operations are described.

Implementation & Specification of *await* Once a process is in the monitor, it may temporarily suspend itself on condition c by issuing *await(c).*

In this example two *await* functions are used in the monitor level for: i) *awaitNotEmptyCondition & ii) awaitNotFullCondition.* Both the *await* function are similiar. However, the *awaitNotEmptyCondition* function ensures $buff- > size > 0,$ and the *awaitNotFullCondition* ensures $buff- > size < CAPACITY.$

The await function requires that the buffer be *wrapped* as well as it also needs the write access to the buffer.

The function ensures the object (in this case the *buff*) which was owned by the condition semaphore, is *wrapped.*

When a thread enters *await*, it needs to wait to acquire the condition semaphore until the condition becomes true. As a result, it will leave the monitor and the *count* of waiting thread on the condition semaphore will be increased. Later, the *count* will be decreased when the condition semaphore is acquired.

The await functions are shown in Listing 4.8 and Listing 4.9.

```
void awaitNotEmptyCondition(BufferMonitor *monitor _(ghost \claim c))
_(always c, monitor->\consistent)
_(requires \wrapped(& monitor->theBuffer))
_(requires monitor->entrance.protected_obj != c)
_(writes & monitor->theBuffer)
_(ensures \wrapped(& monitor->theBuffer) )
_(ensures monitor->theBuffer.size > 0)
{
    _(unwrap & monitor->theBuffer);
    _(unchecked)monitor->theBuffer.notEmptyCount ++;
    _(wrap & monitor->theBuffer);
    release(& monitor->entrance _(ghost c));
    notEmptySemaphoreAcquire(&monitor->notEmptyQ _(ghost c));
    _(unwrap & monitor->theBuffer);
    _(unchecked)monitor->theBuffer.notEmptyCount --;
    _(wrap & monitor->theBuffer);
}
```

Listing 4.8: The awaitNotEmptyCondition method

```
void awaitNotFullCondition(BufferMonitor *monitor _(ghost \claim c))
_(always c, monitor->\consistent)
_(requires \wrapped(& monitor->theBuffer))
_(requires monitor->entrance.protected_obj != c)
_(writes & monitor->theBuffer)
_(ensures \wrapped(monitor->notFullQ.buff) )
_(ensures monitor->theBuffer.size < CAPACITY)
{
    _(unwrap & monitor->theBuffer);
    _(unchecked)monitor->theBuffer.notFullCount ++;
    _(wrap & monitor->theBuffer);
    release(& monitor->entrance _(ghost c));
    notFullSemaphoreAcquire(&monitor->notFullQ _(ghost c));
    _(unwrap & monitor->theBuffer);
    _(unchecked)monitor->theBuffer.notFullCount --;
    _(wrap & monitor->theBuffer);
}
```

Listing 4.9: The awaitNotFullCondition method

Implementation & Specification of *signal* When a process executing in the monitor detects a change in the condition variable, it gives signal using the *signal(c)* to the processes that are waiting in the condition queue.

In this example two signal functions (*signaltNotEmptyCondition & signalNotFullCondition*) are used. Both the *signal* functions are similiar. However, the *signaltNotEmptyCondition* requires $buff- > size > 0$, on the other hand the *signalNotFullCondition* requires $buff- > size < CAPACITY$.

The *signal* function also requires that the object owned by the condition semaphore is *wrapped*.

If there is any thread waiting in the condition semaphore (i.e. $count > 0$) then a waiting thread will be released from the condition semaphore and eventually acquires the *entrance* semaphore. If there is no thread waiting in the semaphore, the signal function will do nothing.

Thus the function ensures the *protected_ obj* owned by the *entrance* semaphore to be *wrapped*.

The signal functions are shown in Listing 4.10 & Listing 4.11.

```
void signalNotEmptyCondition(BufferMonitor *monitor _(ghost \claim c))
_(always c, monitor->\consistent)
_(requires \wrapped(& monitor->theBuffer) )
_(requires monitor->theBuffer.size > 0)
_(writes & monitor->theBuffer)
_(ensures \wrapped(& monitor->theBuffer))
{
    unsigned nEcount;
    _(unwrap & monitor->theBuffer);
    nEcount = monitor->theBuffer.notEmptyCount;
    _(wrap & monitor->theBuffer);

    if(nEcount > 0)
    {
        notEmptySemaphoreRelease(& monitor->notEmptyQ _(ghost c));
        acquire(& monitor->entrance _(ghost c));
    }
}
```

Listing 4.10: The signalNotEmptyCondition method

```
void signalNotFullCondition(BufferMonitor *monitor _(ghost \claim c))
_(always c, monitor->\consistent)
_(requires \wrapped(& monitor->theBuffer) )
_(requires monitor->theBuffer.size < CAPACITY)
_(writes monitor->entrance.protected_obj)
_(ensures \wrapped(& monitor->theBuffer))
{
    unsigned nFcount;
    _(unwrap & monitor->theBuffer);
    nFcount = monitor->theBuffer.notFullCount;
    _(wrap & monitor->theBuffer);

    if(nFcount > 0)
    {
        notFullSempahoreRelease(& monitor->notFullQ _(ghost c));
        acquire(& monitor->entrance _(ghost c));
    }
}
```

Listing 4.11: The signalNotFullCondition method

4.1.3 Implementation & Specification of *deposit* method

The *append* function in Algorithm 4.0 is implemented as *deposit* function in this example. The *deposit* thread puts one element to *theBuffer* when *notFull* condition is true (which implies the buffer is not full).

After updating *theBuffer*, the thread will notify the other threads waiting on the *NotEmptySemaphore* (as now the *notEmpty* condition is *true*). Up to one thread that was suspended on the *NotEmptySemaphore* can enter to the monitor. Finally the thread will leave by calling the monitor exit function (*exitMonitor* function described in Chapter 3).

```
void deposit(long value, BufferMonitor *monitor _(ghost \claim c))
_(always c, monitor->\consistent)
{
    int size, head;
    enterMonitor(monitor _(ghost c));
    if(monitor->theBuffer.size == CAPACITY)
    {
        awaitNotFullCondition(monitor _(ghost c));
        _(assert monitor->theBuffer.size < CAPACITY) ;
    }
    _(unwrap &monitor->theBuffer);
    size = monitor->theBuffer.size ;
    head = monitor->theBuffer.head ;
    monitor->theBuffer.buffer[(head+size) % CAPACITY] = value ;
    monitor->theBuffer.size += 1;
    _(wrap &monitor->theBuffer);
    #ifndef VERIFY
        printf ("deposit...\n");
    #endif
    _(assert monitor->theBuffer.size > 0);
    signalNotEmptyCondition(monitor _(ghost c));
    exitMonitor(monitor _(ghost c));
}
```

Listing 4.12: The deposit method

The *deposit* function is shown in Listing 4.12.

The specification part of the *deposit* says that the *monitor* is *consistent* .

4.1.4 Implementation & Specification of *fetch* method

The *remove* function in Algorithm 4.0, is implemented as *fetch* function in this example. The *fetch* thread removes one element from *theBuffer* when *notEmpty* condition is true (which implies the buffer is not empty).

After updating *theBuffer* the thread will notify the other threads waiting to the *NotFullSemaphore* (as now the *notFull* condition is *true*). Up to one thread that was suspended on the *NotFullSemaphore* can enter to the monitor. However, if there is no thread waiting on the *NotFullSemaphore*, the thread waiting to *enter* the monitor, can proceed. Finally the thread will leave by calling the monitor *exit* function (*exitMonitor* function described in Chapter 3).

```
int fetch(BufferMonitor *monitor _(ghost \claim c))
_(always c, monitor->\consistent)
{
    long result;
    enterMonitor(monitor _(ghost c));
    if(monitor->theBuffer.size == 0)
    {
        awaitNotEmptyCondition(monitor _(ghost c));
        _(assert monitor->theBuffer.size > 0);
    }
    _(unwrap & monitor->theBuffer);
    result = monitor->theBuffer.buffer[monitor->theBuffer.head];
    monitor->theBuffer.head = (monitor->theBuffer.head+1) % CAPACITY ;
    monitor->theBuffer.size -= 1;
    _(wrap & monitor->theBuffer);
    #ifndef VERIFY
        printf ("fetch...\n");
    #endif
    _(assert monitor->theBuffer.size < CAPACITY);
    signalNotFullCondition(monitor _(ghost c));
    exitMonitor(monitor _(ghost c));
    return result;
}
```

Listing 4.13: The fetch method

The *fetch* function is shown in Listing 4.13. The specification part of the *fetch* says that, the *monitor* is *consistent* .

4.1.5 Implementation & Specification of *Producer* Method

The specification of the *producer* thread is similiar to the *deposit* thread where the *producer* produces item and deposits in the buffer. If the buffer is *full,* it waits until the buffer is *not-full* and then deposits the item.

```
void Producer (int value, BufferMonitor * monitor _(ghost \claim c))
_(always c, monitor->\consistent)
{
    deposit(value, monitor _(ghost c));
}
```

Listing 4.14: The Producer method

Listing 4.14 shows the implementation & specification of *producer* thread.

4.1.6 Implementation & Specification of *Consumer* Method

The specification of the *consumer* thread is similiar to the *fetch* thread where the *consumer* fetches the item from the buffer. If the buffer is *empty,* it waits until the buffer is *not-empty* and then fetches the item.

```
int Consumer (BufferMonitor *monitor _(ghost \claim c))
_(always c, monitor->\consistent)
{
    int item;
    item = fetch(monitor _(ghost c));
    return item;
}
```

Listing 4.15: The Consumer method

Listing 4.15 shows the implementation & specification of *consumer* thread.

4.1.7 Validation of proposed methodology

The Verification technology (*VCC*) that is used in this thesis is quite new and to some extent it has its own limitation. It does not ensure liveness properties. However, to verify concurrent programs it is important to test in order to demonstrate that the threads do not get stuck. Testing is a good way to determine if, at least for the cases that were tested, the implementation behaviour is acceptable.

For the testing purpose, an oracle function is used. This section describes the test procedure as well as the test results of *producer/consumer bounded/buffer* example.

4.1.7.0 *Pthreads* in *Producer/Consumer Bounded/Buffer* Example

Testing concurrent programs is always challenging compared to sequential programs, as tests for concurrent programs are themselves concurrent programs. Moreover, failure in concurrent programs are nondeterministic due to its unpredictability and repeatablity. Race conditions, deadlocks, data races etc. are the common unexpected situations that might occur in concurrent programs. In order to test the *producer/consumer boundedbuffer* solution, an oracle function was written using multiple threads (multiple producers and multiple consumers) communicating in between through the protected monitored buffer.

The *pthread_create()* function only allows one argument to pass to the starting thread. In the *producer/consumer boundedbuffer* example, the producer thread takes two parameters: i) pointer to the *BufferMonitor* object and ii) pointer to the *claim* object.

To pass these two parameters, the *ThreadData* structure is implemented which contains a pointer to the *BufferMonitor* object *monitor*. It also contains a *claim* object, *claim* as well. The variable *k* is used as starting index for producer and *n* is used to represent the numbers to produce or consume.

The invariant of *ThreadData* object states that *claim* claims the *monitor*.

The *ThreadData* structure is shown in Listing 4.16.

```
typedef struct _thread_data{
    BufferMonitor * monitor;
    int k; // Starting index for producer; the first number to produce.
    int n; // How many numbers to produce or consume.
    _(ghost \claim claim;)
    _(invariant \claims_object(claim, monitor))
}ThreadData ;
```

Listing 4.16: The ThreadData Structure

Later the pointer to *ThreadData* structure is passed to the starting thread by the *pthread_create()* function.

To start the producer and consumer thread, *start_producer()* and *start_consumer()*

functions are used respectively. The specification of these two methods requires the *ThreadData* object *param* to be *wrapped* and also maintains the consistency of the *BufferMonitor* object (using the claim pointer *claim*). The *start_producer()* and *start_ consumer()* routines are shown in Listing 4.17 and Listing 4.18 respectively.

```
void *start_producer (void *param)
_(requires \wrapped(((ThreadData *) param)))
_(always ((ThreadData *) param)->claim, ((ThreadData *) param)->monitor↩
↪->\consistent)
{
    int n = ((ThreadData *) param)->n;
    int k = ((ThreadData *) param)->k;
    BufferMonitor *monitor = ((ThreadData *) param)->monitor ;
    int i ;
    for (i=k ; i < _(unchecked)(n+k); i++)
    {
        #ifndef VERIFY
            Sleep(rand() / RAND_DIVISOR);
        #endif
        Producer(i, monitor _(ghost ((ThreadData *) param)->claim));
    }
    return NULL;
}
```

Listing 4.17: Start Routine of Producer

```
void *start_consumer (void *param)
_(requires \wrapped(((ThreadData *) param)))
_(always ((ThreadData *) param)->claim, ((ThreadData *) param)->monitor↩
↪->\consistent)
{
    int n = ((ThreadData *) param)->n;
    BufferMonitor *monitor = ((ThreadData *) param)->monitor ;
    int i ;
    for (i = 0; i < n; i++)
    {
        int index ;
        #ifndef VERIFY
            Sleep(rand() / RAND_DIVISOR);
        #endif
        index = Consumer(monitor _(ghost ((ThreadData *) param)->claim));
        #ifndef VERIFY
            a[index]++;
        #endif
    }
    return NULL;
}
```

Listing 4.18: Start Routine of Consumer

4.1.7.1 Comments on tests:

According to the function a large array a of N long items is created. Initially all the items are assigned to 0. The test procedure executes as follows:

- Each producer produces a chunk of integers in array a. E.g. producer 0 produces 0 to $k-1$, producer 1 produces k to $2k-1$, ... , producer $P-1$ produces $(P-1)k$ to $N-1$, where $k = N/P$ and P is the number of producers.
- Each consumer consumes j integers where $j = N/C$ and C is the number of consumers (if C does not divide N, the last consumer should only consume $N-(C-1)j$ integers).
- On consuming an integer i, the consumer increments $a[i]$
- After the completion of all threads, $a[i]$ should be 1 for all i (the variable *allOnes* is used for this purpose).

The solution has given a promising result for large amount of data and multiple threads as shown in the Table 4.0.

The first three rows shows the test results where no delay is defined. The code is tested using multiple threads (upto 20 threads). The last three rows are showing the results of the following tests,

Test #	**N:**	**# P:**	**# C:**	**allOnes:**	**Delay**
1	$10,000$	4	1	1	–
2	$10,000$	10	1	1	–
3	$10,000$	15	5	1	–
4	$100,000$	15	5	1	No delay
5	$100,000$	15	5	1	Delay in fetch
6	$100,000$	15	5	1	Delay in deposit

Table 4.0: Test Results of Producer-Consumer Code

- No delays at all.
- Delays only in the fetch routine.
- Delays only in the deposit routine.

4.2 Summary

In this chapter the implementation and verification of *semaphores* & *monitors* with *condition* variable are presented. The verification is done using the *VCC* tool. Using the implementation, the *producer-consumer bounded-buffer* has been verified. The code is also tested using pthreads to test implementation behavior of the code is acceptable.

Chapter 5

Conclusion and Future Research

5.0 Summary and Conclusions

With the advent of modern concurrent programming, verification has become more important in order to ensure concurrency and software reliability. The design issues of concurrency are (defined by *William Stallings)* such as i) communication among processes, ii) sharing of and competing for resources, iii) synchronization of the activities of multiple processes and iv) allocation of multiple processes as well as allocation of processor time to processes [Stallings, 1992]. Mutual exclusion algorithms are always used to avoid the simultaneous access of a common resource. The methods of monitors are executed with mutual exclusion.

This thesis is dedicated to developing a general approach to monitors specification and verification which can be used for solving synchronization problems in operating systems and other concurrent systems. Specifications are given at the level of C code using the annotation language of Microsoft's Verifier for Concurrent C (VCC). VCC takes the annotated C program and tries to prove that the program meets these specifications.

In addition, the implementation and verification of semaphores and monitors without condition variables are developed in this thesis. Later the implementation and verification of semaphores and monitors with condition variables are developed.

Using the proposed monitor specification code, the specified solution of producer-consumer synchronization problem has also been verified in this thesis.

5.1 Original Contributions

In this thesis an attempt has been made to address of building higher-level abstractions on top of the low-level verification capabilities of *VCC*. This section summarizes the original contributions of the thesis.

- The *semaphore* verification code is implemented and specified along with its *acquire* and *release* method which are used by the processes to acquire and release the resource respectively. The data invariants are also specified in the code.

- The *monitor* verification code is implemented and specified along with its *enterMonitor* and *exitMonitor* routines which are used to enter the monitor and exit the monitor respectively. The prosposed implementation is demonstrated using *timeOfDay* example.

- To handle the conditional delay of the program while acquiring or releasing the resource, two condition semaphores (*NotFullSemaphore* and *NotEmpty-*

Semaphore) are implemented.

- Using the implemented & specified monitor algorithm, the *producer/consumer bounded-buffer* problem is verified.

- The given solution is also tested using an oracle function with multiple threads.

5.2 Recommendations for Future Research

Recommendations for future work are as follows:

- Initially the code was intended to have a single class representing condition variables and attach additional meaning by using a ghost field that would be a pointer to a boolean function that encodes the condition. However VCC currently lacks pointers to pure functions (functions that have no side effects) and so calls to pointers to functions can not be used in assertions such as invariants, and pre- and postconditions. Thus the current design uses separate await and separate signal methods for each condition variable. If in the future pointers to pure functions are allowed, then the implementation of a single class of condition variables should be investigated.

- The *queue* semaphore (*NotFullSemaphore* and *NotEmptySemaphore*) is implemented using an implementation variable. The separate *acquire* and *release* routines are implemented instead of using the common semaphore *acquire* and *release* (although their implementation is same). Re-design of a *queue* semaphore using the *semaphore* class (instead of using the implementation variable) should be considered in future. However, this design also depends on the availability of pointers to pure functions.

References

[Abrial, 1996] J.-R. Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, New York, NY, USA, 1996.

[Alkassar *et al.*, 2008] Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert W. Schirmer, and Artem Starostin. The Verisoft Approach to Systems Verification. In *Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments*, VSTTE '08, pages 209–224, Berlin, Heidelberg, 2008. Springer-Verlag.

[Barnett *et al.*, 2004] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.

[Barnett *et al.*, 2006] Mike Barnett, Bor yuh Evan Chang, Robert Deline, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, volume 4111 of Lecture Notes in Computer Science*, pages 364–387. Springer, 2006.

[Barney, 2011] Blaise Barney. Posix threads programming, 2011. Tutorials, Lawrence Livermore National Laboratory.

[Barry, 2004] Richard Barry. The freertos project. FreeRtos website, http://www.freertos.org/, 2004.

[Ben-Ari, 2006] M. Ben-Ari. *Principles of Concurrent and Distributed Programming (2nd Edition) (Prentice-Hall International Series in Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Bevier *et al.*, 1989] William R. Bevier, Warren A. Hunt, Jr., J. Strother Moore, and William D. Young. An approach to systems verification. *J. Autom. Reason.*, 5:411–428, November 1989.

[Bevier, 1989] W. R. Bevier. Kit: A study in operating system verification. *IEEE Trans. Softw. Eng.*, 15:1382–1396, November 1989.

[Boyer and Moore, 1988] Robert S. Boyer and J. Strother Moore. *A computational logic handbook*. Academic Press Professional, Inc., San Diego, CA, USA, 1988.

[Bucher *et al.*, 2003] Roberto Bucher, Lorenzo Dozio, Daniele Gasperini, Hannes Mayer, Paolo Mantegazza, Pierangelo Masarati, Michael Neuhauser, Michael Racciu, David Schleef, and Peter Soetens. RTAI-The Realtime Application Interface for Linux from DIAPM. RTAI website, 2003.

[Cohen *et al.*, 2009] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Proceedings of the 22nd Interna-*

tional Conference on Theorem Proving in Higher Order Logics, TPHOLs '09, pages 23–42, Berlin, Heidelberg, 2009. Springer-Verlag.

[De Moura and Bjorner, 2008] Leonardo De Moura and Nikolaj Bjorner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[Dijkstra, 1971] E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

[Elphinstone *et al.*, 2007] Kevin Elphinstone, Gerwin Klein, Philip Derrin, Timothy Roscoe, and Gernot Heiser. Towards a practical, verified kernel. In *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 20:1–20:6, Berkeley, CA, USA, 2007. USENIX Association.

[Haigh and Young, 1987] J. Thomas Haigh and William D. Young. Extending the noninterference version of MLS for SAT. *IEEE Trans. Softw. Eng.*, 13:141–150, February 1987.

[Hansen, 1973] Per Brinch Hansen. *Operating system principles*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1973.

[Hillebrand and Leinenbach, 2009] Mark A. Hillebrand and Dirk C. Leinenbach. Formal verification of a reader-writer lock implementation in C. *Electron. Notes Theor. Comput. Sci.*, 254:123–141, October 2009.

[Hoare, 1974] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[Hohmuth and Härtig, 2001] Michael Hohmuth and Hermann Härtig. Pragmatic nonblocking synchronization for real-time systems, 2001.

[Hohmuth *et al.*, 2002a] Michael Hohmuth, Hendrik Tews, and Shane G. Stephens. Applying source-code verification to a microkernel: the vfiasco project. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, EW 10, pages 165–169, New York, NY, USA, 2002a. ACM.

[Kaufmann *et al.*, 2000] Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[Klein, 2009] Gerwin Klein. Operating system verification – An overview. volume Vol. 34, pages 27–69. Sadhana, 2009.

[Liedtke, 1995] J. Liedtke. On micro-kernel construction. *SIGOPS Oper. Syst. Rev.*, 29:237–250, December 1995.

[Lipton and Snyder, 1977] R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *J. ACM*, 24:455–464, July 1977.

[McCauley and Drongowski, 1979] E. J. McCauley and P. J. Drongowski. Ksos–the design of a secure operating system. volume Vol. 48, pages AFIPS Press 345–353. 1979 National Computer Conference, 1979.

[Neumann and Feiertag, 2003] Peter G. Neumann and Richard J. Feiertag. Psos revisited. In *Proceedings of the 19th Annual Computer Security Applications Conference*, ACSAC '03, Washington, DC, USA, 2003. IEEE Computer Society.

[Nipkow *et al.*, 2002] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic.* Springer-Verlag, Berlin, Heidelberg, 2002.

[Norvell, 2009] Theodore S. Norvell. Behavioural specifications. Course Notes, 2009. Memorial University of Newfoundland and Labrador, http://www.engr.mun.ca/∼theo/Courses/acce/pub/acce-notes-bg-0.pdf.

[Norvell, 2010] Theodore S. Norvell. Correctness of computing system. Course Notes, 2010. Memorial University of Newfoundland and Labrador, http://www.engr.mun.ca/∼theo/Courses/acce/pub/toc-ch-0.pdf.

[Owicki and Lamport, 1982] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4:455–495, July 1982.

[Owre *et al.*, 1996] Sam Owre, S. Rajan, John M. Rushby, Natarajan Shankar, and Mandayam K. Srivas. Pvs: Combining specification, proof checking, and model checking. In *Proceedings of the 8th International Conference on Computer Aided Verification*, CAV '96, pages 411–414, London, UK, 1996. Springer-Verlag.

[Peyton Jones, 2003] Simon Peyton Jones. Haskell 98 language and libraries: The revised report. Kluwer Academic Publishers, 2003.

[Reason, 1990] James Reason. *Human Error.* Cambridge [England] ; New York : Cambridge University Press, 1990. xv, 302 p., 1990.

[Robinson and Levitt, 1977] Lawrence Robinson and Karl N. Levitt. Proof techniques for hierarchically structured programs. *Commun. ACM*, 20:271–283, April 1977.

[Saydjari *et al.*, 1987] O. Saydjari, J. Beckman, and J. Leaman. Locking computers securely. pages 129–141. 10th National Computer Security Conference, 1987.

[Schulte *et al.*, 2010] Wolfram Schulte, Ernie Cohen, and Stephan Tobies. Verifying concurrent C programs with VCC, working draft, version 0.1, july 21, 2010.

[Shapiro and Weber, 2000] Jonathan S. Shapiro and Samuel Weber. Verifying the EROS confinement mechanism, 2000.

[Shapiro *et al.*, 1999] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. *SIGOPS Oper. Syst. Rev.*, 33:170–185, December 1999.

[Shapiro, 2008] Jonathan S Shapiro. Coyotos website, 2008. http://www.coyotos.org/. Link visited July 2011.

[Stallings, 1992] William Stallings. *Operating systems.* Macmillan Publishing Co., Inc., Indianapolis, IN, USA, 1992.

[Walker *et al.*, 1979] Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. Specification and verification of the UCLA unix security kernel (extended abstract).

In *Proceedings of the seventh ACM symposium on Operating systems principles*, SOSP '79, pages 64–65, New York, NY, USA, 1979. ACM.

Appendix A

An Appendix

In this thesis semaphores are implemented as a binary semaphores for monitors without condition variables. The implementation is shown in Appendix A.1. Monitor is simply an object protected by an entrance semaphore. In Appendix A.2 the implementation of TimeOfDay example is given, which is an example of monitor without condition variables. Later, in Appendix A.3 the implementation of Producer/Consumer Bounded Buffer example is presented which is an example of monitor with condition variables; notFullSemaphore (shown in Appendix A.4) & notEmptySemaphore (shown in Appendix A.5)

A.0 InterlockedCompareExchange Implementation

```
#include <vcc.h>

_(atomic_inline) int InterlockedCompareExchange(volatile long
*Destination, long Exchange, long Comparand) {
        if (*Destination == Comparand) {
            *Destination = Exchange;
            return Comparand;
        } else {
            return *Destination;
        }
    }
```

Listing A.0: InterLockedCompareExchange.h

A.1 Semaphore Implementation

```
#ifdef VERIFY
    #define CAPACITY 5
#endif
_(claimable) _(volatile_owns) typedef struct _Semaphore {
    volatile int s;
    _(ghost \object protected_obj;)
    _(invariant s==0 || s==1 )
    _(invariant s == 1 <==> \mine(protected_obj))
}Semaphore;

void initializeSemaphore(Semaphore *sem _(ghost \object obj))
_(writes \span(sem))
_(writes obj)
_(requires \wrapped(obj))
_(ensures \wrapped(sem) && sem->protected_obj == obj)
_(ensures sem->s == 1)
{
    sem->s = 1;
    _(ghost {
        sem->protected_obj = obj;
        sem->\owns = {obj};
    _(wrap sem)
    })
}
void acquire(Semaphore *sem _(ghost \claim c))
_(always c, sem->\consistent)
_(ensures \wrapped(sem->protected_obj) && \fresh(sem->protected_obj))
{
    int stop = 0;
    do {
        _(atomic c, sem) {
            stop = InterlockedCompareExchange(&sem->s, 0, 1) == 1;
            _(ghost if (stop) sem->\owns -= sem->protected_obj)
        }
    }while (!stop);
}

void release(Semaphore *sem _(ghost \claim c))
_(always c, sem->\consistent)
_(requires sem->protected_obj != c)
_(writes sem->protected_obj)
_(requires \wrapped(sem->protected_obj))
{
    _(atomic c, sem) {
        sem->s = 1;
        _(ghost sem->\owns += sem->protected_obj)
    }
}
```

Listing A.1: semaphore.h

A.2 TimeOfDay Implementation

```
#include "vcc.h"
#include <stdlib.h>
#include<time.h>
#ifdef VERIFY
    #include "InterLockedCompareExchange.h"
#else
    #include <Windows.h>
    #include <pthread.h>
    #include <stdio.h>
#endif

#include "semaphore.h"
#define NUM_OF_TICK_THREADS 100
#define NUM_OF_GET_THREADS 100
#define NUM_OF_TICKS 100000
#define NUM_OF_GETS 100000
#define RAND_DIVISOR 100000000


#ifndef VERIFY
    pthread_t tick_threads[NUM_OF_TICK_THREADS];
    pthread_t get_threads[NUM_OF_GET_THREADS];
```

```
#endif
typedef struct _TimeOfDay{
    int hr;
    int min;
    int sec;
    _(invariant 0 <= sec && sec < 60
            && 0 <= min && min < 60
            && 0 <= hr && hr < 24 )

}Time;

_(claimable) typedef struct _Time_Monitor
{ Time t ;
    Semaphore entrance;
    _(invariant \mine(&entrance))
    _(invariant entrance.protected_obj == &t)
}TimeMonitor;

void enterMonitor(TimeMonitor *monitor _(ghost \claim c))
_(always c, (& monitor->entrance)->\consistent)
_(ensures \wrapped(monitor->entrance.protected_obj) )
_(ensures \fresh(monitor->entrance.protected_obj) )
{
    acquire(& monitor->entrance _(ghost c));
}

void exitMonitor(TimeMonitor *monitor _(ghost \claim c))
_(always c, (& monitor->entrance)->\consistent)
_(requires \wrapped(monitor->entrance.protected_obj))
_(requires monitor->entrance.protected_obj != c)
_(writes monitor->entrance.protected_obj)
{
    release( & monitor->entrance _(ghost c));
}

void tick(TimeMonitor *monitor _(ghost \claim c))
_(always c, monitor->\consistent)
{
    enterMonitor(monitor _(ghost c));
    _(unwrap &monitor->t);
        monitor->t.sec += 1;
        monitor->t.min += monitor->t.sec/60;
        monitor->t.hr += monitor->t.min/60;
        monitor->t.sec = monitor->t.sec % 60;
        monitor->t.min = monitor->t.min % 60;
        monitor->t.hr = monitor->t.hr % 24;
    _(wrap &monitor->t);
    exitMonitor(monitor _(ghost c));
}

void get(int time[], TimeMonitor *monitor _(ghost \claim c))
_(writes \array_range(time,3) )
_(always c, monitor->\consistent)
{
        enterMonitor(monitor _(ghost c));
        _(unwrap &monitor->t);
                time[0] = monitor->t.sec;
                time[1] = monitor->t.min;
                time[2] = monitor->t.hr;
        _(wrap &monitor->t);
        exitMonitor(monitor _(ghost c));
}
#ifndef VERIFY
static volatile bool go = 0 ;
#endif

typedef struct _thread_data{
    TimeMonitor * monitor;
    int count;
    bool spin;
    _(ghost \claim claim;)
    _(invariant \claims_object(claim, monitor))
}ThreadData ;

ThreadData tickData[NUM_OF_TICK_THREADS];
ThreadData getData[NUM_OF_GET_THREADS];

void *start_tick (void *param)
_(requires \wrapped(((ThreadData *) param)))
_(always ((ThreadData *) param)->claim, ((ThreadData *) param)->monitor↩
↪->\consistent)
{
    int i ;
    TimeMonitor *monitor = ((ThreadData *) param)->monitor ;
    #ifndef VERIFY
        while( ! go ) {}
    #endif
    for (i=0 ; i < NUM_OF_TICKS ; i++)
    {
        tick(monitor _(ghost ((ThreadData *) param)->claim));
    }
    return 0 ;
}

void *start_get (TimeMonitor *monitor _(ghost \claim c))
_(always c, monitor->\consistent )
{
    int time[3],i;
    #ifndef VERIFY
        while( ! go ) {}
    #endif
    for (i=0 ; i < NUM_OF_GETS ; i++)
    _(writes \array_range(time,3) )
    {
        get(time, monitor _(ghost c) );
        if( time[0] >= 60 || time[1] >= 60 || time[2] >= 24 )
        {
            #ifndef VERIFY
                printf("Get␣FAILED") ;
            #endif
        }
    }
}

void *start_get1 (void *param)
_(requires \wrapped(((ThreadData *) param)))
_(always ((ThreadData *) param)->claim, ((ThreadData *) param)->monitor↩
↪->\consistent)
{
    start_get1( ((ThreadData *) param)->monitor _(ghost ((ThreadData *) ↩
↪param)->claim)) ;
    return 0 ;
}
TimeMonitor monitor;
int main()
_(writes \universe())
```

```
_(requires \program_entry_point())
{
    int i,t, rc,countSum;
    int tickCount, getCount;
    #ifndef VERIFY
        int endTime ;
        int startTime = clock() ;
    #endif
    _(ghost \claim c, c1;)

    monitor.t.hr = 0;
    monitor.t.min = 0;
    monitor.t.sec = 0;
    _(wrap & monitor.t);

    initializeSemaphore(& monitor.entrance _(ghost & monitor.t));
    _(ghost (&monitor)->\owns += &monitor.entrance;)
    _(assert monitor.entrance.protected_obj == & monitor.t);
    _(assert & monitor.entrance \in \domain(& monitor.entrance))
    _(wrap &monitor);
    _(ghost c = \make_claim({&monitor}, (&monitor)->\consistent);)

    #ifndef VERIFY
        t = 0;
        tickCount = 0;
        countSum = 0;
        printf("Tick␣threads:␣%d.␣Ticks␣per␣thread:␣%d.␣Get␣threads:␣%d↩
↪.␣Gets␣per␣thread:␣%d\n",
                NUM_OF_TICK_THREADS, NUM_OF_TICKS, ↩
↪NUM_OF_GET_THREADS, NUM_OF_GETS);
        printf("Creating␣Tick␣thread\n");
        for(t=0; t<NUM_OF_TICK_THREADS; t++){
            tickData[t].monitor = &monitor ;
            _(ghost tickData[t].claim = c ;)
            _(wrap & tickData[t]);
            rc = pthread_create(&tick_threads[t], NULL, start_tick, (void *) &↩
↪tickData[t]);
            tickCount++;
            printf("tick␣%d␣is␣created␣\n", t);
            if (rc){
                printf("ERROR;␣return␣code␣from␣pthread_create()␣is␣%d\n↩
↪", rc);
```

```
            exit(-1);
        }
    }

    for(t=0; t<NUM_OF_GET_THREADS; t++){
        getData[t].monitor = &monitor ;
        _(ghost getData[t].claim = c ;)
        _(wrap & getData[t]);
        rc = pthread_create(&get_threads[t], NULL, start_get, (void *) & getData[t]);
            printf("get␣%d␣is␣created\n", t);
        if (rc){
            printf("ERROR;␣return␣code␣from␣pthread_create()␣is␣%d\n", rc);
            exit(-1);
        }
    }

    go = 1 ;

    for(t=0; t<NUM_OF_TICK_THREADS; t++){
        pthread_join(tick_threads[t], NULL);
        countSum += NUM_OF_TICKS ;
    }

    for(t=0; t<NUM_OF_GET_THREADS; t++){
        pthread_join(get_threads[t], NULL);
    }
    printf("t.hr:␣%d␣==␣%d\n␣", monitor.t.hr, (countSum/3600)%24 );
    printf("t.min:␣%d␣==␣%d\n␣", monitor.t.min, (countSum%3600)/60 );
    printf("t.sec:␣%d␣==␣%d\n␣", monitor.t.sec, countSum%60 );
    endTime = clock() ;
    printf("time␣is␣%d␣s␣", (endTime-startTime)/CLOCKS_PER_SEC );
    #endif
    return 1;
}
/*
C:\Saimon\vcc\Prod-Cons\Prod-Cons>vcc /2/it/smoke testTime.c
Verification of _Semaphore#adm succeeded.
Verification of _TimeOfDay#adm succeeded.
Verification of _Time_Monitor#adm succeeded.
Verification of _thread_data#adm succeeded.
Verification of _wctime succeeded.
Verification of _wctime_s succeeded.
Verification of difftime succeeded.
Verification of ctime succeeded.
Verification of ctime_s succeeded.
Verification of gmtime succeeded.
Verification of gmtime_s succeeded.
Verification of localtime succeeded.
Verification of localtime_s succeeded.
Verification of mktime succeeded.
Verification of _mkgmtime succeeded.
Verification of time succeeded.
Verification of initializeSemaphore succeeded.
Verification of acquire succeeded.
Verification of release succeeded.
Verification of enterMonitor succeeded.
Verification of exitMonitor succeeded.
Verification of tick succeeded.
Verification of get succeeded.
Verification of start_tick succeeded.
Verification of start_get1 succeeded.
Verification of start_get succeeded.
Verification of main succeeded.
*/
```

Listing A.2: time.c

A.3 Producer/Consumer Implementation

```
#include "vcc.h"
#include <stdlib.h>
#ifdef VERIFY
    #include "InterLockedCompareExchange.h"
#else
    #include <Windows.h>
    #include <pthread.h>
    #include <stdio.h>
#endif
#include "semaphore.h"
#include "notFullSemaphore.h"
#include "notEmptySemaphore.h"
```

```
#define CAPACITY 5
#define N 1000
#define NUM_OF_PROD_THREADS 4
#define NUM_OF_CONS_THREADS 1
#define RAND_DIVISOR 100000000
int a[N];

#ifndef VERIFY
    pthread_t prod_threads[NUM_OF_PROD_THREADS];
    pthread_t cons_threads[NUM_OF_CONS_THREADS];
#endif

typedef struct _MonitoredBuffer{
    int buffer[CAPACITY];
    int size ;
    int head ;
    _(invariant 0 <= size && size <= CAPACITY)
    _(invariant 0 <= head && head < CAPACITY)
    unsigned notFullCount;
    unsigned notEmptyCount;
}MonitoredBuffer;

_(claimable) typedef struct _BufferMonitor
{
    MonitoredBuffer theBuffer;
    Semaphore entrance;

    _(invariant \mine(& notFullQ))
    _(invariant notFullQ.buff == &theBuffer)

    NotFullSemaphore notFullQ;
    NotEmptySemaphore notEmptyQ;

    _(invariant \mine(& notEmptyQ))
    _(invariant notEmptyQ.buff == &theBuffer)

    _(invariant \mine(& entrance))
    _(invariant entrance.protected_obj == &theBuffer)

}BufferMonitor;

void awaitNotFullCondition(BufferMonitor *monitor _(ghost \claim c))
```

```
_(always c, monitor->\consistent)
_(requires \wrapped(& monitor->theBuffer))
_(requires monitor->entrance.protected_obj != c)
_(writes & monitor->theBuffer)
_(ensures \wrapped(monitor->notFullQ.buff) )
_(ensures monitor->theBuffer.size < CAPACITY)
{
    _(unwrap & monitor->theBuffer);
    _(unchecked)monitor->theBuffer.notFullCount ++;
    _(wrap & monitor->theBuffer);
    release(& monitor->entrance _(ghost c));
    notFullSemaphoreAcquire(&monitor->notFullQ _(ghost c));
    _(unwrap & monitor->theBuffer);
    _(unchecked)monitor->theBuffer.notFullCount --;
    _(wrap & monitor->theBuffer);
}

void signalNotFullCondition(BufferMonitor *monitor _(ghost \claim c))
_(always c, monitor->\consistent)
_(requires \wrapped(& monitor->theBuffer) )
_(requires monitor->theBuffer.size < CAPACITY)
_(writes monitor->entrance.protected_obj)
_(ensures \wrapped(& monitor->theBuffer))
{
    unsigned nFcount;
    _(unwrap & monitor->theBuffer);
    nFcount = monitor->theBuffer.notFullCount;
    _(wrap & monitor->theBuffer);

    if(nFcount > 0)
    {
        notFullSempahoreRelease(& monitor->notFullQ _(ghost c));
        acquire(& monitor->entrance _(ghost c));
    }
}

void awaitNotEmptyCondition(BufferMonitor *monitor _(ghost \claim c))
_(always c, monitor->\consistent)
_(requires \wrapped(& monitor->theBuffer))
_(requires monitor->entrance.protected_obj != c)
_(writes & monitor->theBuffer)
_(ensures \wrapped(& monitor->theBuffer) )
```

```
_(ensures monitor->theBuffer.size > 0)
{
    _(unwrap & monitor->theBuffer);
    _(unchecked)monitor->theBuffer.notEmptyCount ++;
    _(wrap & monitor->theBuffer);
    release(& monitor->entrance _(ghost c));
    notEmptySemaphoreAcquire(&monitor->notEmptyQ _(ghost c));
    _(unwrap & monitor->theBuffer);
    _(unchecked)monitor->theBuffer.notEmptyCount --;
    _(wrap & monitor->theBuffer);
}

void signalNotEmptyCondition(BufferMonitor *monitor _(ghost \claim c))
_(always c, monitor->\consistent)
_(requires \wrapped(& monitor->theBuffer) )
_(requires monitor->theBuffer.size > 0)
_(writes & monitor->theBuffer)
_(ensures \wrapped(& monitor->theBuffer))
{
    unsigned nEcount;
    _(unwrap & monitor->theBuffer);
    nEcount = monitor->theBuffer.notEmptyCount;
    _(wrap & monitor->theBuffer);

    if(nEcount > 0)
    {
        notEmptySemaphoreRelease(& monitor->notEmptyQ _(ghost c));
        acquire(& monitor->entrance _(ghost c));
    }
}

void enterMonitor(BufferMonitor *monitor _(ghost \claim c))
_(always c, (& monitor->entrance)->\consistent)
_(ensures \wrapped(monitor->entrance.protected_obj) )
_(ensures \fresh(monitor->entrance.protected_obj) )
{
    acquire(& monitor->entrance _(ghost c));
}

void exitMonitor(BufferMonitor *monitor _(ghost \claim c))
_(always c, (& monitor->entrance)->\consistent)
_(requires \wrapped(monitor->entrance.protected_obj))
```

```
_(requires monitor->entrance.protected_obj != c)
_(writes monitor->entrance.protected_obj)
{
    release(& monitor->entrance _(ghost c));
}

void deposit(long value, BufferMonitor *monitor _(ghost \claim c))
_(always c, monitor->\consistent)
{
    int size, head;
    enterMonitor(monitor _(ghost c));
    if(monitor->theBuffer.size == CAPACITY)
    {
        awaitNotFullCondition(monitor _(ghost c));
        _(assert monitor->theBuffer.size < CAPACITY) ;
    }
    _(unwrap &monitor->theBuffer);
    size = monitor->theBuffer.size ;
    head = monitor->theBuffer.head ;
    monitor->theBuffer.buffer[(head+size) % CAPACITY] = value ;
    monitor->theBuffer.size += 1;
    _(wrap &monitor->theBuffer);
    #ifndef VERIFY
        printf ("deposit...\n");
    #endif
    _(assert monitor->theBuffer.size > 0);
    signalNotEmptyCondition(monitor _(ghost c));
    exitMonitor(monitor _(ghost c));
}

int fetch(BufferMonitor *monitor _(ghost \claim c))
_(always c, monitor->\consistent)
{
    long result;
    enterMonitor(monitor _(ghost c));
    if(monitor->theBuffer.size == 0)
    {
        awaitNotEmptyCondition(monitor _(ghost c));
        _(assert monitor->theBuffer.size > 0);
    }
    _(unwrap & monitor->theBuffer);
    result = monitor->theBuffer.buffer[monitor->theBuffer.head];
    monitor->theBuffer.head = (monitor->theBuffer.head+1) % CAPACITY ;
    monitor->theBuffer.size -= 1;
    _(wrap & monitor->theBuffer);
    #ifndef VERIFY
        printf ("fetch...\n");
    #endif
    _(assert monitor->theBuffer.size < CAPACITY);
    signalNotFullCondition(monitor _(ghost c));
    exitMonitor(monitor _(ghost c));
    return result;
}

void Producer (int value, BufferMonitor * monitor _(ghost \claim c))
_(always c, monitor->\consistent)
{
    deposit(value, monitor _(ghost c));
}

int Consumer (BufferMonitor *monitor _(ghost \claim c))
_(always c, monitor->\consistent)
{
    int item;
    item = fetch(monitor _(ghost c));
    return item;
}

typedef struct _thread_data{
    BufferMonitor * monitor;
    int k; // Starting index for producer; the first number to produce.
    int n; // How many numbers to produce or consume.
    _(ghost \claim claim;)
    _(invariant \claims_object(claim, monitor))
}ThreadData ;

ThreadData producerData[NUM_OF_PROD_THREADS];
ThreadData consumerData[NUM_OF_CONS_THREADS];

void *start_producer (void *param)
_(requires \wrapped(((ThreadData *) param)))
_(always ((ThreadData *) param)->claim, ((ThreadData *) param)->monitor↩
↪->\consistent)
{
    int n = ((ThreadData *) param)->n;
    int k = ((ThreadData *) param)->k;
    BufferMonitor *monitor = ((ThreadData *) param)->monitor ;
    int i ;
    for (i=k ; i < _(unchecked)(n+k); i++)
    {
        #ifndef VERIFY
            Sleep(rand() / RAND_DIVISOR);
        #endif
        Producer(i, monitor _(ghost ((ThreadData *) param)->claim));
    }
    return NULL;
}

void *start_consumer (void *param)
_(requires \wrapped(((ThreadData *) param)))
_(always ((ThreadData *) param)->claim, ((ThreadData *) param)->monitor↩
↪->\consistent)
{
    int n = ((ThreadData *) param)->n;
    BufferMonitor *monitor = ((ThreadData *) param)->monitor ;
    int i ;
    for (i = 0; i < n; i++)
    {
        int index ;
        #ifndef VERIFY
            Sleep(rand() / RAND_DIVISOR);
        #endif
        index = Consumer(monitor _(ghost ((ThreadData *) param)->claim));
        #ifndef VERIFY
            a[index]++;
        #endif
    }
    return NULL;
}

BufferMonitor monitor;

int main()
_(writes \universe())
_(requires \program_entry_point())
{
```

```
int i,t, rc, allOnes;

_(ghost \claim c;)

for (i = 0; i < N; i++)
    a[i] = 0;

for (t = 0; t < CAPACITY; t++)
    monitor.theBuffer.buffer[t] = -1;
monitor.theBuffer.size = 0;
monitor.theBuffer.head = 0;
monitor.theBuffer.notFullCount = 0 ;
monitor.theBuffer.notEmptyCount = 0 ;
_(wrap & monitor.theBuffer);

notFullSemaphoreInitialize(&monitor.notFullQ, & monitor.theBuffer);
_(ghost (& monitor)->\owns += & monitor.notFullQ;)
_(assert monitor.notFullQ.buff == &monitor.theBuffer);
_(assert & monitor.notFullQ \in \domain(& monitor.notFullQ));

notEmptySemaphoreInitialize(&monitor.notEmptyQ, & monitor.theBuffer);
_(ghost (& monitor)->\owns += & monitor.notEmptyQ;)
_(assert monitor.notEmptyQ.buff == &monitor.theBuffer);
_(assert & monitor.notEmptyQ \in \domain(& monitor.notEmptyQ));

initializeSemaphore(& monitor.entrance _(ghost & monitor.theBuffer));
_(ghost (&monitor)->\owns += &monitor.entrance;)
_(assert monitor.entrance.protected_obj == & monitor.theBuffer);
_(assert & monitor.entrance \in \domain(& monitor.entrance))

_(wrap &monitor);
_(ghost c = \make_claim({&monitor}, (&monitor)->\consistent);)

#ifndef VERIFY
    t = 0;
    for(t=0; t<NUM_OF_PROD_THREADS; t++){
        producerData[t].monitor = &monitor ;
        _(ghost producerData[t].claim = c ;)
        producerData[t].k = t * (N/NUM_OF_PROD_THREADS) ;
        if (t == (NUM_OF_PROD_THREADS -1))
            producerData[t].n = (N - ((N/NUM_OF_PROD_THREADS) * (NUM_OF_PROD_THREADS-1)));
        else
            producerData[t].n = N/NUM_OF_PROD_THREADS;
        _(wrap & producerData[t]);
        rc = pthread_create(&prod_threads[t], NULL, start_producer, (void *) &producerData[t]);
        printf("producer␣%d␣is␣created␣\n", t);
        if (rc){
            printf("ERROR;␣return␣code␣from␣pthread_create()␣is␣%d\n", rc);
            exit(-1);
        }
    }

    for(t=0; t<NUM_OF_CONS_THREADS; t++){
        consumerData[t].monitor = &monitor ;
        _(ghost consumerData[t].claim = c ;)
        if (t == (NUM_OF_CONS_THREADS -1))
            consumerData[t].n = (N - ((N/NUM_OF_CONS_THREADS) * (NUM_OF_CONS_THREADS-1)));
        else
            consumerData[t].n = N/NUM_OF_CONS_THREADS;
            _(wrap & consumerData[t]);
            rc = pthread_create(&cons_threads[t], NULL, start_consumer, (void *) & consumerData[t]);
            printf("consumer␣%d␣is␣created\n", t);
        if (rc){
            printf("ERROR;␣return␣code␣from␣pthread_create()␣is␣%d\n", rc);
            exit(-1);
        }
    }

    for(t=0; t<NUM_OF_PROD_THREADS; t++){
        pthread_join(prod_threads[t], NULL);
        printf("producer␣%d␣done\n", t);
    }

    for(t=0; t<NUM_OF_CONS_THREADS; t++){
        pthread_join(cons_threads[t], NULL);
        printf("consumer␣%d␣done\n", t);
    }

```

```
        allOnes = 1;
        for (i = 0; i< N; i++)
        {
            allOnes *= a[i];
            if(a[i] != 1)
            printf("a[%d]=␣%d␣",i,a[i]);
        }
        printf("\nallOnes␣=␣%d\n", allOnes);
        return 1;

    #endif
}

/*
Verification of _Semaphore#adm succeeded.
Verification of _NotFullSemaphore#adm succeeded.
Verification of _MonitoredBuffer#adm succeeded.
Verification of _NotEmptySemaphore#adm succeeded.
Verification of _BufferMonitor#adm succeeded.
Verification of _thread_data#adm succeeded.
Verification of initializeSemaphore succeeded.
Verification of acquire succeeded.
Verification of release succeeded.
Verification of notFullSemaphoreInitialize succeeded.
Verification of notFullSemaphoreAcquire succeeded.
Verification of notFullSempahoreRelease succeeded.
Verification of notEmptySemaphoreInitialize succeeded.
Verification of notEmptySemaphoreAcquire succeeded.
Verification of notEmptySemaphoreRelease succeeded.
Verification of awaitNotFullCondition succeeded.
Verification of signalNotFullCondition succeeded.
Verification of awaitNotEmptyCondition succeeded.
Verification of signalNotEmptyCondition succeeded.
Verification of enterMonitor succeeded.
Verification of exitMonitor succeeded.
Verification of deposit succeeded.
Verification of fetch succeeded.
Verification of Producer succeeded.
Verification of Consumer succeeded.
Verification of start_producer succeeded.
Verification of start_consumer succeeded.
Verification of main succeeded.
```

```
*/
```

Listing A.3: prod-cons.c

A.4 NotFullSemaphore Implementation

```
#ifdef VERIFY
    #define CAPACITY 5
#endif
typedef struct _MonitoredBuffer MonitoredBuffer;

_(claimable) _(volatile_owns) typedef struct _NotFullSemaphore {
    volatile int s;
    MonitoredBuffer * buff;
    _(invariant s==0 || s==1 )
    _(invariant s == 1 <==> \mine(buff))
    _(invariant s == 1 ==> buff−>size < CAPACITY)
}NotFullSemaphore;

void notFullSemaphoreInitialize(NotFullSemaphore *sem, MonitoredBuffer * buffer)
_(writes \span(sem))
_(ensures \wrapped(sem))
_(ensures sem−>s == 0)
_(ensures sem−>buff == buffer)
{
    sem−>s = 0;
    sem−>buff = buffer;
    _(ghost {
        sem−>\owns = {};
        _(wrap sem)
    })
}

void notFullSemaphoreAcquire(NotFullSemaphore *sem _(ghost \claim c))
_(always c, sem−>\consistent)
_(ensures \wrapped(sem−>buff) && \fresh(sem−>buff))
_(ensures sem−>buff−>size < CAPACITY)
{
    int stop = 0;
    do {
        _(atomic c, sem) {
            stop = InterlockedCompareExchange(&sem−>s, 0, 1) == 1;
            _(ghost if (stop) sem->\owns -= sem->buff)
        }
    }while (!stop);
 }

void notFullSempahoreRelease(NotFullSemaphore *sem _(ghost \claim c))
_(always c, sem->\consistent)
_(requires sem->buff->size < CAPACITY)
_(writes sem->buff)
_(requires \wrapped(sem->buff))
{
    _(atomic c, sem) {
        sem->s = 1;
        _(ghost sem->\owns += sem->buff)
    }
}
```

Listing A.4: notFullSemaphore.h

A.5 NotEmptySemaphore Implementation

```
#ifdef VERIFY
    #define CAPACITY 5
#endif
typedef struct _MonitoredBuffer MonitoredBuffer ;

_(claimable) _(volatile_owns) typedef struct _NotEmptySemaphore {
    volatile int s;
    MonitoredBuffer * buff;
    _(invariant s==0 || s==1 )
    _(invariant s == 1 <==> \mine(buff))
    _(invariant s == 1 ==> buff->size > 0)
} NotEmptySemaphore;

void notEmptySemaphoreInitialize(NotEmptySemaphore *sem, MonitoredBuffer * ←↩
↪buffer)
_(writes \span(sem))
_(ensures \wrapped(sem))
_(ensures sem->s == 0)
_(ensures sem->buff == buffer)
{
    sem->s = 0;
    sem->buff = buffer;
    _(ghost {
        sem->\owns = {};
        _(wrap sem)
    })
}

void notEmptySemaphoreAcquire(NotEmptySemaphore *sem _(ghost \claim c))
_(always c, sem->\consistent)
_(ensures \wrapped(sem->buff) && \fresh(sem->buff))
_(ensures sem->buff->size > 0)
{
    int stop = 0;
    do {
        _(atomic c, sem) {
            stop = InterlockedCompareExchange(&sem->s, 0, 1) == 1;
            _(ghost if (stop) sem->\owns -= sem->buff)
        }
    } while (!stop);
}

void notEmptySemaphoreRelease(NotEmptySemaphore *sem _(ghost \claim c))
_(always c, sem->\consistent)
_(requires sem->buff->size > 0)
_(writes sem->buff)
_(requires \wrapped(sem->buff))
{
    _(atomic c, sem) {
        sem->s = 1;
        _(ghost sem->\owns += sem->buff)
    }
}
```

Listing A.5: notEmptySemaphore.h