BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

SUPPORT OF RUN-TIME MONITORING OF PROCESSES IN ANACONDA FRAMEWORK

PODPORA PRO MONITORING PROCESŮ ZA BĚHU V PROSTŘEDÍ ANACONDA

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. MONIKA MUŽIKOVSKÁ**
AUTOR PRÁCE

SUPERVISOR — **Ing. ALEŠ SMRČKA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2019/2020

Master's Thesis Specification

Student: **Mužikovská Monika, Bc.**
Programme: Information Technology Field of study: Intelligent Systems
Title: **Support of Run-time Monitoring of Processes in ANaConDA Framework**
Category: Software analysis and testing
Assignment:

1. Study the ANaConDA framework for experiments with testing and dynamic analysis of concurrent C/C++ programs with noise-injection techniques.
2. Analyse the problematics of monitoring parallel processes which use basic synchronisation primitives.
3. Design the modification of ANaConDA framework such that it can be used for monitoring parallel processes. Focus on dynamic analysis of process synchronisation.
4. Implement the design extension in the ANaConDA framework.
5. Verify the functionality of implemented extension on suitable use cases. Focus on a support of synchronisation primitives used in student exercises in the course of Operating systems.

Recommended literature:

- FIEDOR Jan, MUŽIKOVSKÁ Monika, SMRČKA Aleš, VAŠÍČEK Ondřej a VOJNAR Tomáš. Advances in the ANaConDA Framework for Dynamic Analysis and Testing of Concurrent C/C++ Programs. In: *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York: Association for Computing, 2018, s. 356-359. ISBN 978-1-4503-5699-2.
- DIAS Ricardo J., FERREIRA Carla, FIEDOR Jan, LOURENCO Joao, SMRČKA Aleš, SOUSA Diogo J. a VOJNAR Tomáš. Verifying Concurrent Programs Using Contracts. In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Tokyo: Institute of Electrical and Electronics Engineers, 2017, s. 196-206. ISBN 978-1-5090-6032-0.

Requirements for the semestral defence:

- The first two items.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Smrčka Aleš, Ing., Ph.D.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2019
Submission deadline: June 3, 2020
Approval date: May 11, 2020

Abstract

This work extends ANaConDA framework for dynamic analysis of multi-threaded programs with support for multi-process monitoring. This thesis summarizes ANaConDA's approach to analysis and differences between threads and processes. The most important ones involve inter-process communication, separate logical address spaces, and synchronisation with general semaphores. The implemented extension provides API for inter-process communication via shared memory, monitors operations with shared memory in order to translate virtual addresses to their unique representation among processes, and monitors synchronisation operations with semaphores and provides information about them to analysers. The extension significantly simplifies the development of multi-process analysers. This is shown on implementation of two analysers for data race detection, AtomRace and FastTrack, which were, until now, available for multi-threaded programs only. The implementation of FastTrack algorithm uses happens-before relation for general semaphores which is also defined in this thesis. Proposed and implemented solutions were verified on a set of automatic tests and the two analysers were used for experiments on a set of students' projects. Experiments showed that ANaConDA framework is now able to detect concurrency-related errors in multi-process programs and, as such, provide support with implementation of large category of parallel programs.

Abstrakt

Tato práce rozšiřuje nástroj ANaConDA pro dynamickou analýzu vícevláknových programů o možnost analyzovat také programy víceprocesové. Část práce se soustředí na popis nástroje ANaConDA a mechanismů, které pro monitorování využívá, a na jejich nutné úpravy vzhledem k rozdílům procesů a vláken. Tyto zahrnují nutnost složitějších mechanismů pro meziprocesovou komunikaci, nutnost překládat logické adresy na jiný jednoznačný identifikátor a monitorování obecných semaforů. Rozšíření pro monitorování procesů tyto problémy řeší za vývojáře analyzátorů, čímž velmi zjednodušuje jejich vývoj. Užitečnost rozšíření je ukázána na implementaci dvou analyzátorů pro detekci souběhu (AtomRace a FastTrack), které bylo dosud možné využít pouze na vícevláknové programy. Implementace algoritmu FastTrack využívá happens-before relaci pro obecné semafory, která byla také definována jako součást této práce. Experimenty s analyzátory na studentských projektech ukázaly, že nástroj ANaConDA je nyní schopen detekovat paralelní chyby i ve víceprocesových programech a může tak pomoci při vývoji další skupiny paralelních programů.

Keywords

multi-process analysis, dynamic analysis, ANaConDA, synchronisation, happens-before relation, vector clocks, general semaphores, shared memory, virtual address, concurrency-related errors, data race, AtomRace, FastTrack

Klíčová slova

analýza procesů, dynamická analýza, ANaConDA, synchronizace, happens-before relace, vektorové hodiny, obecné semafory, sdílená paměť, logické adresy, paralelní chyby, souběh, AtomRace, FastTrack

Reference

MUŽIKOVSKÁ, Monika. *Support of Run-Time Monitoring of Processes in ANaConDA Framework*. Brno, 2020. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Aleš Smrčka, Ph.D.

Rozšířený abstrakt

S nárůstem paralelních programů také roste poptávka po nástrojích, které umí detekovat nedeterministické a těžce odhalitelné paralelní chyby. Jednou z úspěšně využívaných technik je dynamická analýza, která monitoruje běh programu a ze získaných informací vyvozuje závěry o případných chybách. Dynamická analýza je ale implementačně velmi náročná, a proto existují nástroje, které monitorování běhu programu poskytují. Detekci chyb potom provádí speciální algoritmy, které se implementují s využitím nástrojů ve formě analyzátorů.

Jedním z nástrojů pro dynamickou analýzu je ANaConDA, na jehož vývoji se podílí výzkumná skupina VeriFIT. Vývoj nástroje je podporován evropským projektem AQUAS, dosažené výsledky byly v roce 2018 oceněny na konferenci ISSTA a ANaConDA se s úspěchem využívá také v komerční sféře. Stejně jako jiné známé nástroje ale i on dosud podporoval pouze monitorování vícevláknových programů, přestože algoritmy pro detekci chyb mezi druhem paralelismu nerozlišují. Implementace analyzátorů je ale na druhu paralelismu závislá a vyžaduje podporu ze strany nástroje. Cílem této práce tedy bylo rozšířit nástroj ANaConDA o podporu monitorování procesů a o API, které by vývoj víceprocesových analyzátorů co nejvíce ulehčilo.

Velká část této práce se přirozeně zabývá rozdíly mezi vlákny a procesy. Pro dynamickou analýzu představují největší problém oddělené adresové prostory procesů a jiné způsoby synchronizace.

Samotné analyzátory jsou paralelní programy, které kopírují chování monitorovaného programu při vytváření vláken a procesů. Na detekci chyb se podílí všechna vlákna/procesy a musí tedy využívat určité formy komunikace. U vícevláknových analyzátorů se sdílená data ukládají do globálních proměnných, ale u víceprocesových programů je potřeba využívat např. sdílenou paměť. Sdílená data jsou často ukládána v dynamických STL kontejnerech a jejich ukládání do sdílené paměti je problematické. Jedním z výsledků této práce je tedy implementace API, které poskytuje sdílené datové typy včetně dynamických, jež je v analyzátorech možné využívat téměř totožným způsobem jako lokální proměnné. API přitom řeší problémy spojené s realokací a s nedostatkem volného prostoru ve sdílené paměti.

Dalším důsledkem oddělených adresových prostorů je nemožnost využívat logickou adresu jako jednoznačný identifikátor. Tohoto se doposud využívalo např. pro detekci souběhu, kdy se stejné paměťové místo ve více vláknech rozpoznalo na základě stejné logické adresy. U víceprocesových programů stejná logická adresa nemusí znamenat stejné místo v paměti a naopak. Navíc, ne všechny adresy, na které proces přistupuje, jsou sdílené, a mohou tak způsobit chybu. Druhým výsledkem této práce je návrh a implementace algoritmu, který logické adresy a jiné identifikátory z nich odvozené překládá na jednoznačný identifikátor, který je stejný pro všechny procesy. Zároveň pro danou logickou adresu rozpozná, zda je sdílená či nikoli. Tento algoritmus je založený na monitorování operací se sdílenou pamětí a rozsahu adres, který jí v daném procesu přísluší. K odvození jednoznačného identifikátoru se pak využívají informace o sdílené paměti a posun v rámci ní.

Pro detekci paralelních chyb je často nezbytné monitorovat synchronizaci. ANaConDA dosud podporovala monitorování především těch synchronizačních primitiv, která využívají vlákna, což jsou především zámky. Procesy se ale velmi často synchronizují s využitím obecných semaforů. Bylo tedy potřeba dodat podporu pro monitorování semaforů. Řada algoritmů také provádí extrapolaci pomocí happens-before relace a vektorových hodin, které reprezentují synchronizaci provedenou v monitorovaném programu. Aby bylo možné určité analyzátory implementovat také pro procesy, bylo třeba definovat happens-before relaci pro obecné semafory. Monitorování semaforů i tato nová definice byly implementovány

a úspěšně využity pro implementaci algoritmu FastTrack pro detekci souběhu. Druhým implementovaným algoritmem je AtomRace, taktéž pro detekci souběhu, který ale nemonitoruje synchronizaci. Oba nové analyzátory byly úspěšně využity pro experimenty nad studentskými projekty a prokázaly správnost návrhu a implementace rozšíření pro monitorování procesů.

Support of Run-Time Monitoring of Processes in ANaConDA Framework

Declaration

I hereby declare that this Master's thesis was prepared as an original author's work under the supervision of Ing. Aleš Smrčka, Ph.D. and Ing. Jan Fiedor, Ph.D. All relevant information sources, which were used during the preparation of this thesis, are properly cited and included in the list of references.

.....
Monika Mužikovská
May 29, 2020

Acknowledgements

I would like to thank Ing. Aleš Smrčka, Ph.D. and Ing. Jan Fiedor, Ph.D. for the constant guidance and help they provided me during the whole four years of my work on the ANaConDA framework.

Contents

Chapter 1

Introduction

Enhancement of computational performance has always been one of the non-negligible objectives in various sectors of information technologies. As advancements in hardware led to multi-core processors, clusters, and other distributed systems, algorithms took advantage of them and adapted parallelism. Nowadays, multi-threading or multi-processing is quite common in real-world programs. Despite their undeniable advantages, parallel programs are prone to concurrency-related errors. As these can be difficult to encounter, their detection and correction are often very demanding. As a result, tools and frameworks based on static or dynamic analysis are highly requested. The ANaConDA framework for dynamic analysis of multi-threaded C/C++ programs on the binary level is one of them.

ANaConDA is in the maintenance of VeriFIT research group which focuses on the analysis and verification of (not only) parallel programs. Currently, the framework is successfully used in a commercial sphere for multi-threaded analysis. For some time, there was an intention to extend it for multi-process analysis as well. The support for processes would help in the development of various applications, as similar tool currently does not exist. It will also be useful for students in the Operating Systems course, as one of their projects is a multi-process program in C.

The goal of this Master's thesis is to extend ANaConDA with necessary mechanisms for detection of concurrency-related errors in multi-process programs. The ANaConDA framework and dynamic analysis in general are described in Chapter 2. The differences between threads and processes, which necessitate certain changes, are summarized in Chapter 3. Solutions to presented issues are described in Chapter 4. The implementation of an extension and its usage are described in Chapter 5. Evaluation of the implementation and results of experiments with new multi-process analysers are presented in Chapter 6.

Chapter 2

Analysis of Parallel Programs

Parallel programs are prone to nondeterministic concurrency-related errors whose occurrences are dependent on threads or processes interleaving. Context-switching is strongly affected by an architecture executing the program and other processes using the computational resources. As a result, a developer or a tester usually encounters only a subset of possible program's executions and interleavings and common testing methods for serial programs are insufficient. Therefore, more complex approaches, such as static and dynamic analysis, are used for concurrency-related errors detection.

Static analysis gathers information from the program's source code without its execution. This method is one of the possible approaches for formal verification and, as such, it can be used to prove the program's correctness. To be able to do so, all relevant executions and interleavings need to be analysed which brings scalability issues. Static analysis usually implements various approximations to be able to analyse real-world programs, but these may dramatically increase a rate of false-positives. Despite these disadvantages, static analysis has been proven to be useful and is provided by a number of frameworks such as Coverity from Synopsys[1], SpotBugs[2] or tools from AbsInt[3]. VeriFIT currently uses FacebookInfer[4] and develops new plugins for parallel programs as well.

Dynamic analysis executes a program to detect errors. Similarly to common testing, it analyses only one execution. Unlike testing, dynamic analysis performs extrapolation which makes it possible to detect unwitnessed errors. Extrapolation in parallel programs is usually based on gathering information about synchronisation and using them to decide whether it prevents, for example, a data race from happening or not and a different interleaving would lead to error. The analysis is also able to more precisely locate the error and provide useful information for its correction. Unlike static analysis, dynamic analysis is not able to prove the program's correctness. However, dynamic analysis is usually easier to use and does not require a source code which can be an advantage in the commercial sphere.

Both types of analysis are quite demanding and usually provided by a tool or a framework. These often come with a set of analysers and support for the implementation of new ones. Analysers are implementations of algorithms for the detection of a certain type of errors. Some of the algorithms for concurrency-related errors are described in Section 2.2.

[1] https://www.synopsys.com/software-integrity.html
[2] https://spotbugs.github.io/
[3] https://www.absint.com/
[4] https://fbinfer.com/

2.1 State-of-the-Art in Dynamic Analysis of Concurrent Programs

As mentioned, gathering information about a monitored program is usually done by a framework which can monitor applications in a certain programming language. E.g. ConTest [5] and RoadRunner [10] are frameworks for dynamic analysis of multi-threaded programs in Java. Analysis of C/C++ programs is more demanding, as it needs to be done on the binary level. The ANaConDA framework is currently the only framework with binary level monitoring of multi-threaded programs that we are aware of. The most similar tool for dynamic analysis of C/C++ programs is Fjalar [12]. However, it does not concentrate on parallel programs and, as such, does not provide necessary information to analysers for their monitoring. Also, to the best of our knowledge, we are not aware of a tool for detection of concurrency-related errors in multi-process programs. To provide this functionality to developers, it needs to be implemented into an existing framework for multi-threaded analysis. For the following reasons, ANaConDA was a natural choice for multi-process extension:

- Although ANaConDA provides the best support for C/C++ programs, the binary level monitoring allows analysis of programs written in any compiled language.
- ANaConDA is continuously maintained in VeriFIT group. Ing. Jan Fiedor, Ph.D., who created ANaConDA as his PhD thesis, is also a member of the group.
- ANaConDA is successfully used in the commercial sphere.
- In 2018, ANaConDA received the Best Tool Demonstration award at ISSTA 2018[5] and its development is supported from AQUAS[6] project.

2.2 ANaConDA Framework

ANaConDA[7] (Adaptable Native-code Concurrency-focused Dynamic Analysis) is an open-source framework for dynamic analysis of multi-threaded C/C++ programs on binary level [6, 7]. Its main purpose is to provide support for easier development of new dynamic analysers for detection of concurrency-related errors. Developers may utilize ANaConDA on two levels:

- The framework informs analysers about events detected in a monitored program.
- Analysers may use a set of utilities provided by the framework. It includes functions and structures implementing operations widely used in concurrency-error-detection algorithms, such as data types that are independent on architecture or operating system and mechanisms for loading configuration files or obtaining backtrace information.

This section describes the current state of ANaConDA where both types of support are thread-oriented. The next chapter focuses on differences between threads and process as these determine the necessary changes to allow multi-process monitoring.

[5] https://conf.researchr.org/details/issta-2018/issta-2018-demos/6/Advances-in-the-ANaConDA-Framework-for-Dynamic-Analysis-and-Testing-of-Concurrent-C-C

[6] https://aquas-project.eu/

[7] http://www.fit.vutbr.cz/research/groups/verifit/tools/anaconda/

ANaConDA consists of three components:

- Pin tool – Manages program monitoring and dynamic instrumentation of binary file.
- ANaConDA framework core – Provides a higher abstraction over Pin tool, API for analysers with a set of utilities described above, and a noise insertion technique.
- Analysers – Usually implement algorithms for detection of concurrency-related errors.

2.2.1 Pin Tool

Intel Pin[8] [22] is a framework for dynamic instrumentation of binary programs. Instrumentation is a way of monitoring which events occur in the analysed program. Pin inserts its own code in the executable which allows him to:

1. collect important information, such as current register contents, and
2. execute user-defined functions when an important event occurs.

Fundamentals of Pin's Dynamic Instrumentation

Static instrumentation inserts monitoring code in the original executable before the analysis. With dynamic instrumentation, the monitoring code is inserted at the run-time and the original executable is not modified. Static instrumentation has lower overhead as dynamic instrumentation is performed each time anew. However, it cannot handle self-modifying or self-generating code. While dynamic instrumentation leads to slower analysis, the monitored program is not changed and can be used as usual simultaneously with the analysis [8]. Pin performs dynamic instrumentation in a way similar to just-in-time compilers. It attaches itself to the first instruction, generates a new code for the straight-line code sequence starting at this first instruction, and adds a mechanism for regaining control at the end of this sequence. Then, Pin transfers control to this generated sequence and regains it once the sequence is exited. After that, a new sequence is generated and the analysis continues. While generating code sequences, a user-defined code can be instrumented into the executable.

Development of Pintools

User-defined functions which are to be executed when a specific event occurs are called *callbacks* and defined in so-called *pintools*. Analysers for concurrency-related error detection may be developed directly as pintools. However, Pin's API is quite low levelled and complicated and the implementation would be overwhelming. To make this process as easy as possible, a special pintool ANaConDA was created.

Other Tools for Dynamic Instrumentation

Valgrind [25] is another tool for dynamic instrumentation. However, unlike Pin, it runs only on Linux platforms, and threads/processes in the analysed program are serialized. Due to these reasons, Pin was chosen as an underlying tool for ANaConDA at first. Currently, the framework is being prepared for Valgrind as well and users should be able to choose the tool they wish to use. This means that the extension for processes should be independent on instrumentation tool and use only ANaConDA's API.

[8] https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

2.2.2 ANaConDA Framework Core

ANaConDA framework is an interlayer between the Pin tool and analysers, and has several important functions in the dynamic analysis process:

1. Provides analysers with functions for callbacks registration. These functions are a high-level abstraction over the instrumentation of user-defined code. Callbacks are registered for specific concurrency-related events (e.g. mutex lock). These events can be caused by a library function (e.g. from `pthreads` library), system call, Windows or C++ implementation. Registration functions abstract particular implementation used in the monitored program.

2. Interprets some low-level information obtained from Pin tool (such as backtrace information, values of function's arguments and function's name) and provides them to analysers in a more user-friendly way.

3. Supports *noise-injection* [8] which is an important technique for dynamic analysis as it influences context switching and thread/process interleaving. This can lead to unexpected and unwitnessed program behaviour and cause an error which can be detected by an analyser. ANaConDA allows user to define parameters of noise, i.e. type, frequency and strength. Finding the ideal combination of parameters for a given program is often very demanding task, but it is not the focus of this thesis. However, one needs to be aware of this, as ANaConDA itself may encounter very rare interleavings and could also easily contain a concurrency error in the implementation of framework or analysers.

4. Provides an extensive API to simplify the development of new analysers. This includes structures for thread-local data, lockable objects, configuration files, logging mechanisms, synchronised print to standard output and much more.

Callbacks for Event-Driven Analysis

As callbacks are an essential part of dynamic analysis, they are also the most critic part of ANaConDA's adaptation for multi-process analysis. Currently, ANaConDA provides instrumentation for these events and for given callbacks retrieves information listed in the brackets:

- memory accessed for read, write or atomic update (thread's identifier, a virtual address of accessed memory space, size of accessed memory space, backtrace information about an accessed variable, location in source code),
- lock acquired or released (thread's and lock's identifier),
- monitor signal or wait (thread's and condition's identifier),
- thread forked, started, finished, joined (identifier of both threads when applicable),
- function entered, executed (thread's identifier, arguments and return value),
- transaction memory-related callbacks.

Each event is provided with a pair of callbacks, i.e. so-called *before* callback, which is executed before the detected operation (e.g. before a new thread is forked), and *after* callback, which is executed after the detected operation (e.g. after a thread successfully acquires a lock).

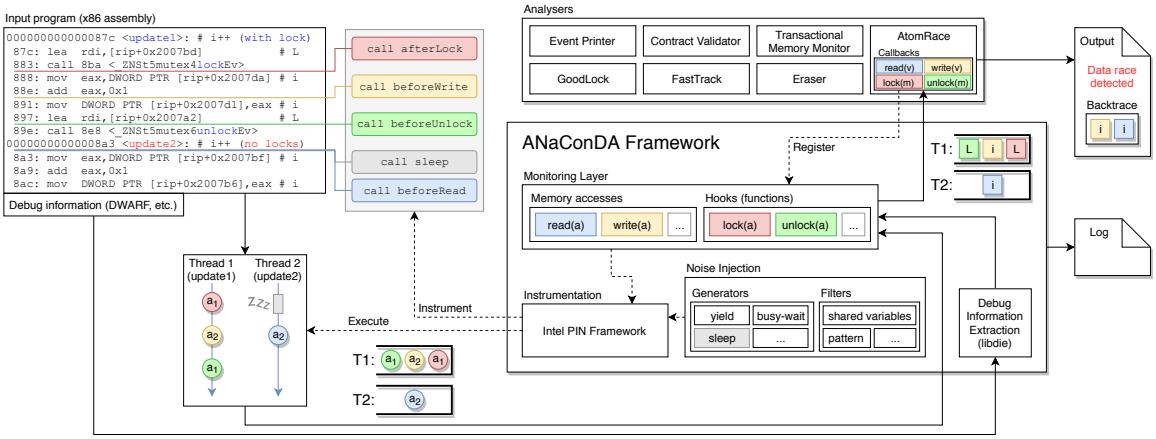


Figure 2.1: A scheme of ANaConDA framework. Source: [6]

2.2.3 Analysers

Analysers usually implement some algorithm for concurrency-related errors detection. This is still an active field of research and new algorithms and analysers are developed, e.g. contract validation [4]. This section summarizes analysers which are currently available in ANaConDA framework:

- **Event-Printer** – Monitors program behavior and prints information about detected events. It does not search for any errors.
- **Goodlock** – A deadlock detector [13]. It monitors operations with locks and creates a graph representation of them to detect a possible problem.
- **Analysers for contract validation** – Two analysers for detection of contract [4] and parametrized contract [23] violation. The algorithm uses a happens-before relation described in Section 2.3.
- **Analysers for data-race detection** – Currently Eraser [32], AtomRace [21], and FastTrack [9] are available. Each of these algorithms monitors memory accesses to the same address by different threads. AtomRace does not monitor any synchronisation, FastTrack uses happens-before relation and Eraser monitors locks. To use these algorithms for processes as well, ANaConDA should provide the necessary information about memory accesses and synchronisation performed by each process (this issue is addressed in depth in Chapter 3).
- **Other analysers** – ANaConDA also provides analyser for high level data race detection (HLDR-detector), Statistics-collector and TX-monitor for transaction memories.

2.2.4 The Process of Analysis in ANaConDA Framework

Figure 2.1 shows the analysis of an input program using analyser AtomRace. The analyser registers callback functions for a few selected events (e.g. read, write, lock and unlock as shown in the figure). The registration is performed on two levels. The monitoring layer of ANaConDA framework performs registration for analyser and stores information

about callback functions which are to be executed in the analyser when the corresponding event occurs. For the same events, ANaConDA itself registers lower-level callback functions using Pin. Then, the Pin tool dynamically instruments and executes the binary file. During instrumentation, instructions to call registered callback functions (functions from the monitoring layer of the framework) are inserted before and after the corresponding event. During the execution of a monitored program, the inserted instructions execute callbacks in ANaConDA and provide them with low-level information. The framework then calls analyser's callback functions and provides them with preprocessed, higher-level information. Apart from callbacks, instrumentation can be used to insert a noise in the monitored program to cause rare interleavings. ANaConDA framework is also able to obtain debugging information, e.g. name of the function and location in the source code.

2.3 Happens-Before Relation

Algorithms for contract violation and FastTrack for data-race detection need to determine the order of events which occurred in different threads. This can be achieved by a so-called *happens-before* relation whose fundamentals were described by Lamport in [20]. The relation determines the order of events in a trace with regard to monitored synchronisation primitives, i.e. an event a in thread *t1* happened before event b in thread *t2* if the two threads were synchronised between them. Analysers using this relation perform *extrapolation* and can detect an error which did not happen in the current interleaving, but synchronisation does not prevent it from happening in a different run with different context switching.

Formally, a happens-before relation $\prec_{hb}$ on the set of events $\{e_1, \ldots, e_n\}$ in a trace $\tau = e_1 \ldots e_n$ is the smallest transitively-closed relation such that $e_j \prec_{hb} e_k$ holds when $j < k$ and one of the following conditions is satisfied:

1. Both events are performed by the same thread.
2. Event e_j releases the same lock e_k acquires.
3. Event e_j is a fork of a thread u in a thread t and e_k is executed by u or event e_k is a join of a thread u in a thread t and e_j is executed by u.

Events which are not related by a happens-before relation are considered to be concurrent and may cause concurrency-related errors [4].

2.3.1 Vector Clocks

Analysers in ANaConDA implement a happens-before relation using vector clocks described for FastTrack algorithm [9]. Let T be the set of all threads in the program. A vector clock $VC : T \to \mathbb{N}$ is a record of logical time of each thread $t \in T$ in the monitored program. The time is represented by a natural number. Each thread $t \in T$ holds its own vector clock VC_t. The value $VC_t(t)$ for a thread $t \in T$ is the current clock of the thread t. The value $VC_t(u)$ for a thread $u \in T, u \neq t$ is the last time when a thread t synchronised with a thread u. All events performed by a thread u which happened in a time lesser or equal to $VC_t(u)$ happened before ($\prec_{hb}$) current events performed by the thread t.

Vector clocks are partially-ordered ($\sqsubseteq$) with an associated operators join ($\sqcup$), minimal element ($\bot_V$), and a helper function for incrementing t-component of a vector clock (inc_t)

defined as follows:

$$V_1 \sqsubseteq V_2 \iff \forall t.\, V_1(t) \leq V_2(t) \tag{2.1}$$

$$V_1 \sqcup V_2 = \lambda t.\, max(V_1(t), V_2(t)) \tag{2.2}$$

$$\bot_V = \lambda t.\, 0 \tag{2.3}$$

$$inc_t(V) = \lambda u.\, \texttt{if}\ u = t\ \texttt{then}\ V(u) + 1\ \texttt{else}\ V(u) \tag{2.4}$$

Usage of vector clocks and given operators can be described as follows. After initialization of a vector clock for a thread t, each item $VC_t(u)$ for $u \neq t$ has value 0 (according to minimal element operator) and item $VC_t(t)$ has value 1 (current clock of thread t). When an important event (such as synchronisation with a lock or fork/join) happens in a thread, increment operator is used for time advance. Join operator represents a synchronisation of two threads and is used according to the happens-before relation described above:

- **Fork** synchronisation – when a new thread u is created in a thread t:

  $$VC'_u = VC_u \sqcup VC_t$$

  $$VC'_t = inc_t(VC_t)$$

  After this operation, the following holds:

  $$\forall v \in T : v \neq u \wedge v \neq t \implies VC'_u(v) = VC'_t(v)$$

  $$VC'_u(u) = 1$$

  $$VC'_t(u) = 0$$

  $$VC'_t(t) - 1 = VC'_u(t)$$

  This ensures that each event performed by thread t before the fork happened before any event in thread u.

- **Join** synchronisation – when a thread t is suspended and waits for a thread u to end its execution:

  $$VC'_t = VC_t \sqcup VC_u$$

  $$VC'_u = inc_u(VC_u)$$

  This ensures that each event performed by thread u before the join happened before any event in thread t after the join.

- **Lock** synchronisation – each lock L has its own vector clock VC_L. When a thread t releases a lock L:

  $$VC'_L = VC_t$$

  $$VC'_t = inc_t(VC_t)$$

  When a thread u acquires a lock L:

  $$VC'_u = VC_u \sqcup VC_L$$

  The lock's vector clock is used to forward vector clock of the releasing thread. This ensures that the acquiring thread constructs happens-before relation with correct thread.

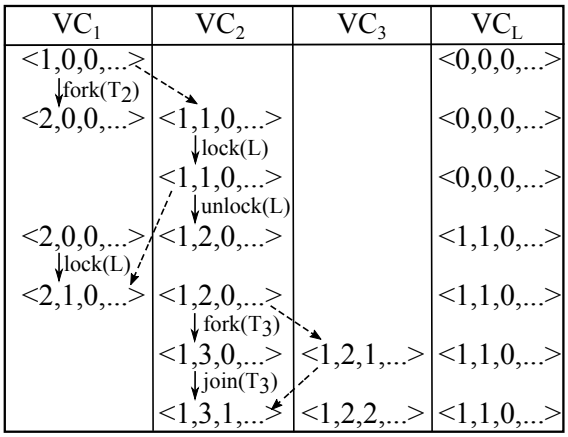


Figure 2.2: An example showing operations over vector clocks when synchronisation occurs. The dashed arrow represents happens-before relation.

Join operation ensures the transitivity of a happens-before relation as it updates information about each thread and not only the ones performing synchronisation. All operations described above are shown in Figure 2.2.

Happens-before relation is not specific for threads and can be used for processes as well. The problem is that processes often use semaphores for synchronisation. Because we are currently not aware of precise definition of a happens-before relation for synchronisation with semaphores, a possible solution was designed and is presented in Section 4.3.

Chapter 3

Differences in Analysis of Threads and Processes

This section summarizes differences between threads and processes and the impact of them on the dynamic analysis itself. A major part (everything except synchronisation) of this study was conducted prior to this Master's thesis in a course Project Practice.

3.1 Analysers

During monitoring of a multi-threaded program, there is an analyser thread for each thread of the program. Each analyser thread executes the same algorithm. The concurrency-related error is often detected by one thread thanks to shared information from other threads.

For now, multi-process programs will be monitored in the same way. For each process of the program, there will be an analyser process. This approach has a significant advantage as a majority of algorithms for threads should be easily applicable to processes as well. We do, however, assume, that there could be different approaches implemented in the future (e.g. for distributed programs or algorithms with one central process performing the analysis and other processes collecting information) and this is taken into account when designing extensions for processes monitoring.

As mentioned in Section 2.2, analysers implement algorithms for concurrency-related errors detection using callbacks. These are adjusted for multi-threaded analysis, i.e. they always provide thread identifier, and additional information (such as arguments of function, memory address, and lock identification) are often provided using virtual addresses. As identification and virtual addresses are two significant differences between threads and processes, they are described in more detail in Sections 3.2 and 3.4. However, almost every event represented by a callback is applicable for both threads and processes (memory access, lock operation, function entered or executed). Thread specific callbacks (fork, start, finish, join) which cannot be used for processes should be extended with similar process-related callbacks.

3.2 Identification of Threads and Processes

Threads have their identifier `THREADID`[1] [14] assigned by Pin. This ID is not unique as it is reused after the thread finalizes. When a unique identification is needed (e.g. for accessing thread local storage or for vector clocks) analysers usually assign their own identification.

Processes have their identifier `pid` assigned by an operating system. While this identification is also reused after the process finalizes, it can be assigned to any process running in the system and not only to monitored processes (as opposed to thread ID assigned by Pin). Linux systems reuse `pid` only after all other numbers had been assigned as operating system increments newly assigned identification until the maximum `pid` and then wraps around [19] (see `/proc/sys/kernel/pid_max` in `proc` manual[2]). Although the probability of two different **monitored** processes having the same `pid` is quite low, the `pid`'s uniqueness is assured neither.

As a thread identifier is important information for almost each callback in each algorithm, callbacks receive it by default in one of their arguments. However, analysers may also obtain it by using Pin's API function `PIN_GetTid()`[3]. Function `getpid()`, which is used by typical programs for obtaining process' `pid`, does not work in analysers. They should use Pin's API function `PIN_GetPid()`[4] instead, and in the future, the best approach will be to provide `pid` directly to all callbacks as well.

3.3 Local and Global Data

As analysers are multi-threaded/multi-process programs, they often need to share data, and developers should be aware of possible concurrency-related errors in an analyser itself. Data are shared in two ways:

1. among threads/processes of the analyser,
2. thread-local data among callbacks.

As an aspect in which the multi-threaded and multi-process programs differ the most is their address space, both types of sharing will be affected.

Threads, unlike processes, share their address space, thus both types of sharing can be accomplished by global variables. Data shared among threads are usually guarded by a lock to prevent data race. Thread-local data are usually stored in a Thread Local Storage (TLS) provided by Pin[5] [14] and accessed with unique thread identifier.

For a multi-process analysis, the situation is quite the opposite. Thanks to separate address spaces, global variables can be used for sharing process-local data among callbacks and no TLS is required. Developers only need to have in mind that when a child process is forked these global variables might need to be cleared and reinitialized. Sharing data among processes is much more challenging and dependent on the type of analysis. As mentioned above, ANaConDA will be extended for analysis of multi-process programs where all processes run on the same machine and may be analyzed in the same way as threads. For this purpose, data shared among analyser's processes can be stored in a shared memory.

[1] https://software.intel.com/sites/landingpage/pintool/docs/THREADID
[2] http://man7.org/linux/man-pages/man5/proc.5.html
[3] https://software.intel.com/sites/landingpage/pintool/docs/PIN_GetTid()
[4] https://software.intel.com/sites/landingpage/pintool/docs/PIN_GetPid()
[5] https://software.intel.com/sites/landingpage/pintool/docs/TLS

As this can be quite demanding, especially for dynamic structures as vectors or strings, a whole new API described in Section 4.1 was implemented in the ANaConDA framework during the Project Practice.

3.4 Virtual Addresses and Shared Memory

Separate address space also has its impact on callbacks for memory access which receive a virtual address of the affected memory. In a multi-threaded analysis, virtual address unambiguously identifies a memory space even among different threads and therefore can be used for data race detection. However, identical virtual addresses in different processes do not have to identify the same memory space and, on the other hand, the same memory space does not have to be identified by the same virtual addresses. An intuitive solution would be to translate the virtual address to corresponding physical address. However, this approach will not work as the physical address may change during the execution (e.g. due to reallocation). As a result, translation from a virtual address to some form of unambiguous identifier has to be provided by the ANaConDA itself. A solution based on shared memory monitoring is described in Section 4.2.

Another issue to consider is the fact that processes from one program (binary file) may use shared memory to communicate with other applications whose processes are not monitored in ANaConDA framework. Their memory accesses are naturally not detected, and an unnoticed concurrency error may occur. This problem cannot be solved by ANaConDA, as it is not possible to detect errors caused by non-monitored processes, but the user should be aware of it. Eventually, developers of analysers may offer two types of analysis:

- **Not-safe analysis** – An analyser will assume that all processes using the shared resources are monitored. If this assumption is not correct, it can cause false-negatives.
- **Safe analysis** – An analyser will assume that other non-monitored processes are using the shared resources as well, but they correctly use a given synchronisation mechanism. If any monitored process accesses a region in the shared memory without synchronisation, the user should be warned.

Both not-safe and safe analysis affect the logic and implementation of analyser's algorithm and cannot be directly provided by ANaConDA framework.

3.5 Combining Processes and Threads

Real-world programs do not have to be limited to processes or threads only but may combine both approaches to multiple processes with multiple threads. As analysers are directly affected, the effect of a fork of a multi-threaded process needs to be considered. Linux manual for `fork()` [18] states that child process is created with a single thread, i.e. the one calling `fork()`. However, the address space of the new process is a copy of an entire address space of its parent and includes resources which can be held by other threads. As they do not exist in the child process, the resources will never be released and the process may encounter various errors including the deadlock. Common programs can use a handler `pthread_atfork()`[6] to get the resources to a consistent state. Analysers for multi-process

[6] http://man7.org/linux/man-pages/man3/pthread_atfork.3.html

and multi-threaded programs should avoid these problems as well in a dedicated callback executed before `fork()`.

Further, only programs not combining threads and processes will be considered, as this type of analysis will be the next step after extending support to processes.

3.6 Synchronisation

Concurrency-related errors are caused by incorrect usage of synchronisation mechanisms in parallel programs. Thus, monitoring of synchronisation events is a crucial part of (not only) dynamic analysis. Basic synchronisation mechanism provided by operating systems and programming languages include:

- Locks – A lock or a mutex is a simple mechanism providing mutual exclusion when accessing a critical section. Before accessing a shared resource, a process or a thread needs to acquire the corresponding lock and release it after exiting the critical section.
- Semaphores – While the lock is only a boolean flag, semaphores are integer variables with operations `up` and `down` and do not necessarily provide mutual exclusion. Their usage is more general and often dependent on a particular application. Semaphores are described in more detail further in this section.
- Monitors – Monitors are higher-level synchronisation mechanism. They provide mutual exclusion to resources inside the monitor and allow threads and processes to wait until a particular condition holds true.

This enumeration is by no means exhaustive and other mechanisms such as read-copy-update [3] or barrier synchronisation can be used, but we will concentrate on the most common locks and semaphores. Both mechanisms can be used to synchronise threads and, when stored in shared memory, processes as well. Currently, only locks are supported in ANaConDA. For the purpose of multi-process monitoring, it is necessary to provide a new set of callbacks for operations over semaphores. Callbacks for locks need to be adjusted as well. Since lock's identification provided by the callback is derived from its virtual address, it will not work for multi-process analysis for the same reasons as callbacks for memory accesses. The same lock in different processes may have a different virtual address and different derived identification. Thus, the ID should be derived from the same unambiguous identifier mentioned in Section 3.4.

3.6.1 Semaphores

As mentioned, synchronisation can be provided by an operating system (e.g. mutexes on Windows[7]), programming language (e.g. `std::mutex` in C++) or `pthreads` library[8]. ANaConDA provides callbacks for operations over locks independently of the concrete implementation used in the monitored program. To preserve this approach, various implementations of semaphores will be considered and eventually supported in ANaConDA under the same callbacks. Namely POSIX[9] and Windows[10] semaphores and System V IPC[11] semaphore sets.

[7] https://docs.microsoft.com/en-us/windows/win32/sync/using-mutex-objects
[8] https://computing.llnl.gov/tutorials/pthreads/
[9] https://linux.die.net/man/7/sem_overview
[10] https://docs.microsoft.com/en-us/windows/win32/sync/semaphore-objects
[11] http://man7.org/linux/man-pages/man7/sysvipc.7.html

POSIX Semaphores

A POSIX semaphore [16] is an integer, whose value may or may not fall below zero depending on implementation. This value can be incremented by one using a function `sem_post()` and decremented by one using a function `sem_wait()` or its non-blocking alternative `sem_trywait()`.

A thread or a process performing `sem_wait()` operation over a semaphore whose value is currently less than or equal to zero will be blocked and queued in a queue associated with the semaphore. Its execution will continue only after another thread or process performs `sem_post()` operation. Depending on the implementation, the decrement may proceed and cause the semaphore's value to be negative. The value then represents a number of waiting threads/processes. When only values greater than or equal to zero are allowed, the semaphore's value will not be decremented immediately, but only after the thread/process is released from waiting. If a thread/process performs `sem_wait()` over a semaphore with a positive value, the decrement proceeds, the function returns, and a thread/process can continue in its execution. Function `sem_trywait()` has the same behaviour for semaphore's with positive value. However, when the decrement cannot proceed, the thread/process is not blocked and the function returns with an error.

A function `sem_post()` either increments semaphore's value (when no thread/process is blocked) or releases one waiting thread/process and allows it to finish its `sem_wait()` operation. The POSIX standard[12] [1] states that semaphore's queue is not guaranteed to be a FIFO because scheduler may choose threads/processes according to scheduling policies.

All of these functions take a virtual address of affected semaphore as an argument. A POSIX semaphore is one of two types, i.e. named or unnamed.

A named semaphore has its own string identifier which makes it possible to use the same semaphore in various independent and unrelated processes without need to store it in shared memory. The identifier is passed as an argument to a function `sem_open()` which opens an existing named semaphore or creates a new one if it does not exist. The function also takes an initial value of the semaphore which is used when a semaphore is created and ignored if it is opened. A semaphore is closed using a function `sem_close()` and removed from system using a function `sem_unlink()`. Example 3.1 shows a usage of a named semaphore.

An unnamed semaphore does not have an identifier and, as such, has to be stored in a memory location shared between all threads and processes which will use it. This can be a thread-shared global variable or a shared memory for processes. Before its first usage, the unnamed semaphore has to be initialized using `sem_init()` function by precisely one thread or process. Initialization sets semaphore's value and determines whether it will be shared among threads or processes. A semaphore is destroyed in the shared memory using function `sem_destroy()`. Example 3.2 shows a usage of an unnamed semaphore.

[12] https://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_post.html

```
int main() {
  /* Open an existing semaphore "/name" or create it with
     permissions 0644 and initial value 0 */
  sem_t* sem = sem_open("/name", O_CREAT, 0644, 0);

  pid_t pid = fork();

  if (fork == 0) {
    ...
    sem_post(sem);
    exit();
  }
  else {
    sem_wait(sem);
    ...
  }
  sem_close(sem);
  sem_unlink("/name");
}
```

Listing 3.1: An example showing usage of a named semaphore for determining order of execution of two processes.

```
int main() {
  // Shared memory creation
  sem_t* sem = (sem_t*) mmap(NULL, sizeof(sem_t), ... );

  // Initializing process-shared semaphore to value 0
  sem_init(sem, 1, 0);

  ...
  // Using semaphore
  ...

  sem_destroy(sem);
}
```

Listing 3.2: An example showing how unnamed semaphores are created, initialized, and destroyed.

System V Semaphores

System V semaphores [17] are more complicated but also more powerful than POSIX semaphores. The most significant differences are:

- Functions `semget()`, `semctl()`, `semop()` for initialization, controlling and operating semaphores use **a semaphore set**.
- System V allows controlling semaphore's permissions.
- Operations up and down may change semaphore's value by a different count than 1.

System V semaphore set is initialized in a way similar to POSIX named semaphores. Each set has its identifier associated with an integer key. The key is generated from a pathname using a function `ftok()`. A function `semget()` takes this key, information about a number of elements in a set, permissions, and control flags to open an existing set or create a new one. Instead of the key, a flag `IPC_PRIVATE` may be used. In such a case, new semaphore set is created. After the function successfully opens or creates a set, its unique identifier is returned. This `semid` is used by `semctl(), semop()` functions. In contrast to POSIX semaphores identified by a virtual address, this identifier is the same for all processes operating the same set. Another difference is that `semget()` does not initialize semaphore's value, and it needs to be done by `semctl()` function. An initialization of a semaphore set with one element is shown in Example 3.3 and with more elements in Example 3.4. `semctl()` function also supports a number of other commands such as `GETVAL` for obtaining semaphore's value, `GETPID` for obtaining an identifier of the process last performing operation on the semaphore, and `IPC_RMID` to destroy the semaphore set.

```
int main() {
  // Creating a key
  key_t key = ftok("path",'A');

  // Open or create a semaphore set with one element and R&W permissions
  int semid = semget(key, 1, 0666 | IPC_CREAT);

  // Initialize first ([0]) semaphore in a set to value 1
  int val = 1;
  semctl(semid, 0, SETVAL, val);
}
```

Listing 3.3: Initializing a semaphore set with one semaphore.

```
int main() {
  // Creating a key
  key_t key = ftok("path",'A');

  // Open or create a semaphore set with two elements and R&W permissions
  int semid = semget(key, 2, 0666 | IPC_CREAT);

  // Initialize both semaphores in a set to value 1
  unsigned short array[2] = {1, 1};
  semctl(semid, 0, SETALL, array);
}
```

Listing 3.4: Initializing a semaphore set with two semaphores.

As mentioned, System V semaphores are operated by a `semop()` function which makes it possible to decrement or increment semaphore's value by more than one. Furthermore, the third operation called *wait for zero* is possible. A type of operation is specified by an integer argument `sem_op`:

- If `sem_op` is a positive integer, the value is added to semaphore's value.

- If `sem_op` is zero and the semaphore's value is zero as well, `semop()` function returns immediately. Otherwise, a calling thread/process is suspended until the value becomes zero. When this happens, all threads/processes waiting for zero are released at once.
- If `sem_op` is a negative integer and the semaphore's value is greater than or equal to absolute value of `sem_op`, the absolute value is subtracted from semaphore's value, and function `semop()` returns immediately. Otherwise, a calling thread/process is suspended until the value becomes greater than or equal to the absolute value of `sem_op`. System V semaphores also do not guarantee FIFO order when releasing suspended threads/processes.

The `semop()` function may perform multiple operations on a particular semaphore set. Operations are specified in an array, and each one of them is performed on a single semaphore from the set. Operations are performed in array order and atomically, i.e. they are performed completely or not at all. Let's consider Example 3.5. Function `semop()` performs two operations over two different[13] semaphores in the set. Both of these are down operations decrementing semaphore's value by one. At first, a process decrements semaphore `setid[0]`. After this operation successfully finishes (the semaphore's value was greater than 0 or another process performed up operation and released waiting process), the function proceeds to operation down on semaphore `setid[1]`. If no error occurs, the process has successfully performed all operations and the function `semop()` successfully returns. When flag `IPC_NOWAIT` (described below) is not used, only abnormal situations such as permission errors can cause a failure of the operation. If such situation occurs, all previously performed operations in the array are annulled, and values of both semaphores remain unchanged. That is, after `semop()` function only two states are possible:

1. Both semaphores have been successfully decremented.
2. Neither of semaphores has been decremented.

```
int main() {
  // Get semaphore set setid with two semaphores
  struct sembuf sops[2];
  sops[0].sem_num = 0; /* Operate on semaphore setid[0] */
  sops[0].sem_op = -1; /* Perform down operation by 1 */
  sops[0].sem_flg = 0;

  sops[1].sem_num = 1; /* Operate on semaphore setid[1] */
  sops[1].sem_op = -1; /* Perform down operation by 1 */
  sops[1].sem_flg = 0;

  semop(setid, sops, 2); /* Perform both operations atomically */
}
```

Listing 3.5: Performing operations over multiple semaphores.

A structure describing operation may also contain two flags, i.e. `IPC_NOWAIT` and `SEM_UNDO`. When `IPC_NOWAIT` is defined, operations which would cause a suspension of a thread or a process will not do so. Instead, an operation will fail (causing the failure of the whole function as well). Let's consider what will happen when precisely one operation

[13] It is allowed to perform multiple operations over the same semaphore as well.

from Example 3.5 has specified flag `IPC_NOWAIT`. As operations are performed in array order, the result depends on which operation from the two is non-blocking.

- Let's consider situation where `sops[0].sem_flg = IPC_NOWAIT`. Two results are possible.
  1. Semaphore's value is positive, the down operation successfully proceeds, and the following operation is executed. Process' suspension due to `sops[1]` **will not** cause function's failure. The flag affects only the particular operation for which it is specified.
  2. Semaphore's value is lesser than or equal to zero. As this would normally cause process' suspension, the operation fails and causes a failure of the whole `semop()` function.
- Let's consider situation where `sops[1].sem_flg = IPC_NOWAIT`. Again, two results are possible. Both of them are independent of the value of semaphore `setid[0]` used in the first operation `sops[0]`. Even if this operation causes process' suspension, it will not cause `semop()` to fail. After the first operation is successfully finished, operation `sops[1]` is performed.
  1. If the value of semaphore `setid[1]` is positive, the operation is successfully executed and function `semop()` successfully returns as well.
  2. If its value is zero or negative, the operation `sops[1]` fails causing the whole `semop()` function to fail. Operation `sops[0]` performed earlier needs to be annulled.

When `SEM_UNDO` is defined, the performed operation will be automatically annulled when a thread/process terminates. This is achieved thanks to a semaphore adjustment value `semadj`. The value is a per-process, per-semaphore integer which mirrors an opposite operation than the one performed on a semaphore. For example, when `sem_op` has value 3 and `SEM_UNDO` is set, the semaphore's value will be incremented by 3, and `semadj` will be decremented by 3. When a thread/process terminates, semaphore's value is adjusted according to `semadj` value. However, when a semaphore's value is set directly by the `semctl()` function using command `SETVAL` or `SETALL`, all adjustment values for the corresponding semaphore are cleared.

Windows Semaphores

Windows semaphores [27] combine both presented implementations. Just like POSIX semaphores, they can be either named or unnamed. The operation up is similar to the System V one as it can increment semaphore's value by more than one. And similarly to System V, where a thread or a process may wait for several semaphores from a set, it can wait for multiple synchronisation objects.

A semaphore is created or opened using a `CreateSemaphore()` function or a `CreateSemaphoreEx()` function which allows specifying access mask. Both named and unnamed semaphores are created with the same function, and the latter is achieved by using `NULL` instead of semaphore's name. After successfully creating or opening a semaphore, the function returns a corresponding handle which is used for operations on the semaphore. A handle is an abstraction over a system resource and a process' access-control list [28]. As a handle can be inherited, different processes can but do not have to use the same handle for the

same semaphore [29]. The same handle will be used for unnamed semaphores, as it is the only option for using such semaphore in more processes. To correctly identify operations over the same named semaphore in ANaConDA, a handler obtained by a callback should be translated to the semaphore's name.

The operation up is performed by `ReleaseSemaphore()` function which takes a handle and a count to add to semaphore's value. The operation down can be performed by any of the wait functions provided for synchronisation objects [31]. The two basic functions `WaitForSingleObject()` and `WaitForMultipleObjects()` wait for a semaphore or semaphores to get into a signalled state which holds when semaphore's value is greater than zero. Wait functions may have a timeout and return after it is elapsed. When a function returns because a semaphore was set to signalled state, its value is decremented by one. The documentation for Windows semaphores also states that FIFO queue cannot be assumed. A semaphore is destroyed when all threads/processes, which used it, closed a handle by `CloseHandle()` function. Example 3.6 shows usage of mentioned functions.

```
int main() {
  // Create named semaphore with default access rights, initially set to
  // zero and maximum count is set to 5
  HANDLE sem = CreateSemaphore(NULL, 0, 5, "SemaphoreName");

  // Increment semaphore's value by two
  LPLONG prev_count;
  ReleaseSemaphore(sem, 2, &prev_count);

  // Wait for a semaphore to get to a signaled state
  WaitForSingleObject(sem, INFINITE);

  CloseHandle(sem);
}
```

Listing 3.6: Using a Windows semaphore.

Chapter 4

Design of Processes Monitoring

This chapter describes the proposed solutions to presented issues with processes monitoring and approaches to their implementation in ANaConDA framework in C/C++. New API for interprocess communication and an algorithm for virtual address translation were designed (and the former also implemented) in the Project Practise. However, as they are an important part of extension for multi-process monitoring, they are also described in this thesis in full detail. A proposed extension of happens-before relation for semaphores and an algorithm for semaphore monitoring were fully designed during the Master's Thesis.

4.1 Interprocess Communication

The API for interprocess communication should provide an easy way of sharing data among processes. As mentioned, we currently consider only processes running on the same machine, thus able to use shared memory for communication. However, as ANaConDA may be extended for distributed processes in the future as well, the API was designed to be easily extendible for message passing and other communication mechanisms.

At first, the following requirements were stated:

1. An API should provide classes or structures which will represent data types of shared variables. An instance of the class will represent a variable which is shared among processes.
2. Data types:
   - 2.1. An API should provide common data types used in analysers. Namely:
     - 2.1.1. Basic data types: `bool, int, short, long, float, double`
     - 2.1.2. STL containers: `std::vector, std::string, std::map, std::deque`
     - 2.1.3. Synchronisation primitives: `PIN_Mutex`
   - 2.2. Classes and structures for shared data types should provide the same operators and methods as the data type they represent.
   - 2.3. A process of implementing new data type or adding another operator to an existing one should be as easy as possible.
3. Back-end implementation:
   - 3.1. Currently, an API should use shared memory.

3.2. It should be possible and easy enough to add another implementation (e.g. message passing). The developer should be able to choose the desired back-end implementation in the analyser.

3.3. Thus, usage of API should not be affected by the chosen back-end.

4. Implicit/explicit synchronisation:

   4.1. An API should provide atomic operations as well as unsynchronised operations over shared variables. The developer should be able to choose the preferred type in the analyser.

5. Write through/back:

   5.1. Currently, an API should provide so-called write through implementation. This means that changes performed over shared variable by one process are immediately propagated to all processes using this variable. E.g. when using shared memory, changes are directly performed over the variable in the shared memory.

   5.2. However, it should be easy to add so-called write back implementation, when changes over shared variable are stored locally and carried out only when another process needs them.

   5.3. The developer should be able to choose the preferred type of implementation in the analyser.

6. Parameters specifying back-end implementation (BI), a type of synchronisation (SI), and write through/back (WI) should be set in one place in an analyser. All variables of the same type will use the same settings.

7. As parameters affect both reading and writing, operations over shared variables need to be carried out by the class or structure provided by API. An analyser should never use these operations directly on a structure encapsulated inside the provider class as it could cause inconsistency or errors.

4.1.1 Using Shared Memory for Dynamic Structures

Provided shared data types include STL containers which are dynamic structures growing during execution. As shared memory has a fixed size, the following issues arise:

1. Detection of a write operation into a memory with insufficient free space.

2. An enlargement of the affected memory when such attempt has occurred.

Shared memory can be created and used directly by system calls or associated wrappers in `C` (these are described in more detail in Section 4.2). However, it would be complicated to solve the first presented issue as it would require a demanding implementation of a new allocator. For this reason, we have decided to use `Boost.Interprocess`[1] [11] which has a solution for both issues. It provides STL-like containers and allocators[2] suitable for shared memory. (Note: `Boost` allocators for shared memory cannot be used with STL

[1] ANaConDA uses Boost version 1.58.0.

[2] https://www.boost.org/doc/libs/1_58_0/doc/html/interprocess/allocators_containers.html

containers due to problems with pointers. This issue is described in more detail in `Boost` documentation[3].)

During write operations over a container, memory segments from shared memory are dynamically allocated. When an operation cannot proceed due to insufficient free space, an exception is thrown which allows API to detect and solve the problem. `Boost` also provides `managed_shared_memory` class which encapsulates memory segments and, more importantly, provides `grow()` method allowing the object's enlargement when the exception occurs. Another advantage of using `Boost` is Windows/Linux portability and it is also already used in ANaConDA quite commonly.

4.1.2 Reallocation of Shared Memory

The `grow()` method can cause a reallocation of affected shared memory resulting into invalidation of virtual addresses used by processes to access data stored in it. After reallocation, each process which has this memory mapped into its address space will have to remap it. The problem is, how should processes be informed that one of them has caused reallocation and remapping is necessary. Three various approaches were taken into consideration: using signals, implementing atomic operations over a shared memory, and accessing a shared memory through a guard.

Signals The process causing reallocation should send a signal to other processes using the same shared memory. This solution would bring distributed computing into analysers and further complications as well. A process accessing shared memory should be certain that it did not miss any signal about reallocation. This would be difficult to achieve as we cannot guarantee the maximal delay between sending the signal and receiving it by all processes. Thus, deciding whether the shared memory is accessible or not would be complicated. For this reason, solution using signals was not implemented.

Atomic operations All operations over a shared memory should be atomic, i.e. perform the following three steps: map shared memory into process's address space, perform the operation, unmap shared memory. The main **advantages** of this approach are that it would certainly solve the problem and would be easy to implement. However, the **disadvantages** outweigh them so much that this solution was not used. Firstly, it would serialize all operations (even reading) over the shared memory and all variables and containers stored in it. This issue could be partially solved by making separate shared memory for each container or variable and thus serializing operations only over one of them. However, it would still result in a significant downgrade in analysis performance as shared memory remapping would be unnecessary in most cases. Secondly, iterations over containers would be problematic. As methods `begin()` and `end()` are considered as separate operations, the container would be remapped between them, and iteration using `for` cycle would not be possible.

A guard This approach signalizes reallocation by setting a flag. It naturally cannot be part of the affected shared memory. Instead, a dedicated shared structure has to be created and used for accessing the shared variable. Each shared structure should be in its own shared memory so that its growth will not affect operations with other shared

[3] https://www.boost.org/doc/libs/1_58_0/doc/html/interprocess/containers_explained

variables. The main **advantage** of this approach is its generality and it also does not have disadvantages of the previous approaches.

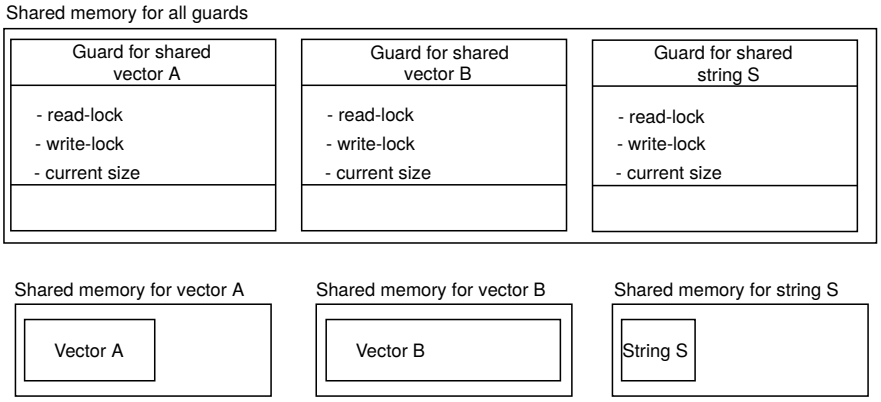


Figure 4.1: A scheme of shared memories when using the guard to solve reallocation problem.

As it is not guaranteed, that processes accessing a shared variable will be mutually excluded, the guard's structure needs to be more complicated than a flag. It will contain write-lock, read-lock, and information about the current size of the shared memory (see Figure 4.1). Locks do not provide synchronisation for operations over the shared variable but for accessing the shared memory itself. Before accessing a shared memory to perform an operation with shared variable, a process needs to take two steps:

1. Acquire read-lock in the guard.
2. Compare the current size of the shared memory with its local information about its size when last mapped. The difference in sizes signalizes that reallocation might have happened, and the process needs to remap the memory into its address space.

Then, an operation may be performed. When writing operation fails due to insufficient space, a process will need to ensure mutual exclusion by acquiring write-lock. Only then it can perform the `grow()` method and update information about the current size of shared memory.

4.1.3 Shared Memory with Variables of Fixed Size

Using guard for shared variables of fixed size is unnecessarily complicated as operations over them will not cause reallocation. For this reason, it is implemented for dynamic shared variables (containers) only. However, even if the shared memory contains only variables of a fixed size, it can have insufficient free space for constructing a new variable. A solution using so-called *banks* was designed and implemented. Each data type has its own shared memory. Its name is composed of:

1. the word `"Shared"`,
2. the name of data type (`"Int"`, `"Double"`, `"Bool"`, etc.),
3. the suffix which can be either empty or specified by the developer of an analyser when different channels for interprocess communication are needed.

Each shared memory is given a space of fixed size called a bank. When analyser requests new shared variable and there is not sufficient free space in the last bank, a new one of the same size is created, and the variable is constructed in it. It means that no reallocation is needed. This approach is shown in Figure 4.2.

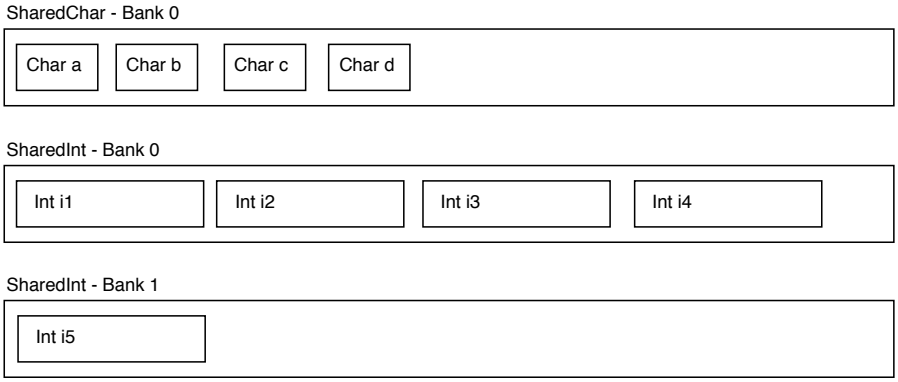


Figure 4.2: A scheme for constructing fix sized variables in banks of shared memory.

4.1.4 Support for API Parameters

To easily meet requirements for synchronization (SI), back-end (BI), and write through/back (WI) parameters, a factory design pattern and inheritance were considered as an ideal approach to implementation. Each class representing a particular shared data type would override operators from its parent. The developer of analyser would use the factory to get the exact implementation depending on specified data type and parameter. However, as API should provide containers for various data types, using templates is inevitable. As combining templates and virtual methods does not bring any advantages, this approach was not used. Instead, the implementation uses templates only. This approach has its advantages from the performance point of view, as templates are resolved by compiler whereas virtual methods are looked up during runtime.

When using templates, classes representing shared data types (e.g. SharedString, SharedVector) take three or four template arguments specifying data type and parameters SI, BI, WI. Operation over shared variables are implemented using methods from classes `Backend`, `Write`, and `Synchronisation`. These methods are templated and can be easily specified for different values of parameters. Pseudocode in Example 4.1 shows implementation of operation `push_back()` over shared vector. Implementation of methods `lock()` and `unlock()` depends on parameter SI. If the developer of analyser requests atomic operations, methods do lock and unlock a shared mutex. If the developer requests unsynchronised operations, these methods do nothing. However, this behaviour is specified for each value of SI argument in the corresponding class `Sync`, and implementation of `push_back()` operation in the shared vector is not affected. Similarly, class `Write` uses class `Backend`, thus takes both parameters WI and BI. Method `push_back()` is either forwarded to class `Backend` immediately (when write through is requested), or a local copy of the shared variable is updated and the operation is performed on this local copy only. Methods provided by class `Backend` perform operations over shared variables according to the chosen implementation of interprocess communication. That is, in the shared memory or by message passing. The final implementation is described in more detail in Chapter 5.

This approach satisfies requirements 3, 4, 5 and 6. Various values of parameters can be supported, adding support for new value of a parameter is easy enough as it requires only template specialisation, and the usage of API in an analyser is not dependent on the chosen parameters. The final implementation satisfies all requirements as it provides all required data types and analysers cannot directly use the encapsulated container.

```
void SharedVector<T>::push_back(T value)
{
  Sync<SI>::lock();
  Write<WI, BI, T>::push_back(value);
  Sync<SI>::unlock();
}

void Sync<Synchronised>::lock()
{
  mutex.lock();
}

void Sync<Unsynchronised>::lock()
{

}

void Sync<Synchronised>::unlock()
{
  mutex.unlock();
}

void Sync<Unsynchronised>::unlock()
{

}

void Write<Through, BI, T>::push_back(T value)
{
  Backend<BI, T>::push_back(value);
}

void Write<Back, BI, T>::push_back(T value)
{
  localVector.update(Backend<BI, T>::getCurrentState());
  localVector.push_back(value);
}
```

Listing 4.1: Illustration of an approach to implementation of shared API for various parameters.

4.1.5 Clearing Shared Memory

When using `managed_shared_memory` class, a named shared memory which outlives the analyser's execution is created. If it is not properly cleared and removed, the next execution would be affected by data already present in the memory. This cannot be done in destructors of structures representing shared variables, as they can be executed at the end of each callback and should not remove corresponding shared memory. Instead, an API should collect information about created shared memories and remove them once analysis finishes. ANaConDA framework provides `PLUGIN_FINISH_FUNCTION()` which is executed at the end of analysis for **each** process. However, it is important not to clear shared memory every time a process finishes, as others can still use it. Instead, the last finishing process should be detected, and only then the clearing function should proceed.

4.2 Virtual Address Translation

Callbacks for memory access do not distinguish whether the affected memory region is local to the process or shared. It provides a virtual address for both types. However, if the affected region is in a shared memory, the virtual address is not sufficient identifier and needs to be translated. This new 'address' will be derived from the given virtual address but will unambiguously identify a region in the shared memory. The same region will have to be represented by the same address in all processes accessing the shared memory. To satisfy this condition, the identifier needs to be derived from information about the affected shared memory and an offset represented by the virtual address. For this reason, ANaConDA will need to monitor all relevant system calls which create, change the size of, or destroy shared memory. This section summarizes system calls provided on Linux and Windows (although currently only Linux will be supported) and describes an algorithm for virtual address translation.

4.2.1 Shared Memory on Linux

Linux provides two types of shared memory, i.e. System V compatible and POSIX.

System V Shared Memory

Create/open shared memory System V shared memory [17] is identified by a key generated from a pathname using `ftok()` function. This key is used as an argument for `shmget()` function which creates new shared memory or opens already existing one and returns its unique identifier (`shmid`). Similarly to System V semaphores, a flag `IPC_PRIVATE` may be used. In such a case, new shared memory is created. However, it is not in any means private (as the name of the flag suggests) and can be accessed normally by all processes with correct permissions when using the same `shmid`. Another important argument is size, as it will determine the range of virtual addresses corresponding to this shared memory. When a new segment of shared memory is created, its size is equal to the given argument rounded up to a multiple of the page size. The final step is attaching shared memory identified by `shmid` to process's address space using `shmat()` function. This function returns a pointer to the beginning of a shared memory (virtual address of the beginning). The range of valid addresses accessing a shared memory is interval `[base, base+size)` where `base` is virtual address of the beginning and `size` is argument given to `shmget()` function. Although the program will not fail due to access to addresses outside this range but still in the same

page, it is not good programming practice thus it will not be considered as valid access to shared memory.

Resize shared memory For translation purposes, the actual physical size is not important. It is only required to know the range of virtual addresses. Therefore, system calls like `brk()` do not have to be monitored. System V does not provide any function which would change the range other than opening the shared memory with different size and attaching it to a different virtual address.

Detache shared memory A shared memory is detached from process' address space using `shmdt()` function. The range of virtual addresses used for accessing regions within it is invalidated. However, this function does not destroy the shared segment. For this purpose, `shmctl()` with command `IPC_RMD` needs to be called. Example 4.2 shows a typical usage of mentioned functions.

```
int main() {
  key_t key = ftok("/path/to/file", 'A');

  size = 1024;
  shmflg = 0644 | IPC_CREAT;
  int shmid = shmget(key, size, shmflg);

  char* data = shmat(shmid, NULL, 0);
  ...
  shmdt(data);
  shmctl(shmid, IPC_RMID, NULL);
}
```

Listing 4.2: Using System V shared memory.

System calls Translation algorithm will need information about shared memory identifier (`shmid`), its size, and the virtual address of its beginning. These are provided by `shmget()` and `shmat()` functions which are in fact wrappers around system calls. To avoid issues when monitoring programs providing their own implementation of similar wrappers, it would be better to monitor system calls directly. According to the source code of `shmget()`[4] and `shmat()`[5] functions, there are two system dependent system calls:

1. The operating system provides system calls `shmget()`, `shmat()` directly[6].
2. Otherwise, the `ipc()`[7] system call with corresponding arguments is used.

At the moment, ANaConDA provides callbacks for executed functions providing information about their arguments and return values, but similar funcionality for system calls is not available. However, Pin provides mechanism for system calls monitoring which will eventually be utilized in ANaConDA as well.

[4] https://code.woboq.org/userspace/glibc/sysdeps/unix/sysv/linux/shmget.c.html
[5] https://code.woboq.org/userspace/glibc/sysdeps/unix/sysv/linux/shmat.c.html
[6] http://man7.org/linux/man-pages/man2/shmat.2.html
[7] http://man7.org/linux/man-pages/man2/ipc.2.html

POSIX Shared Memory

Create/open shared memory POSIX approach is file-based [15]. Shared memory is a special memory-mapped file created/opened by the `shm_open()`[8] function. This is in fact a wrapper around `open()` system call. Shared memory files are created in a dedicated `/dev/shm` directory and identified by a path-like string `"/name"`. `shm_open()` function returns a file descriptor which is passed to `mmap()` function that maps the file into process' virtual address space. `mmap()` also takes several other important arguments:

1. Size – Determines the range of virtual addresses assigned to the mapped file (needs to be lesser than or equal to shared memory's size). POSIX shared memory has the same behaviour as System V. Access to addresses outside valid range but still in the same page will not cause failure.
2. Flags – Three flags are important for translation purposes:
   2.1. `MAP_PRIVATE` – The memory mapping will be private. Thus, it is not shared memory, and ANaConDA does not need to monitor it.
   2.2. `MAP_SHARED` without `MAP_ANONYMOUS` – The mapped file is identified by a provided file descriptor which was obtained using the file's pathname.
   2.3. `MAP_SHARED` with `MAP_ANONYMOUS` – The mapping is not backed by any file, the file descriptor is ignored, and the shared memory is available only to children processes forked after this operation as they need to use the same virtual addresses to access it.
3. Offset – The mapping starts at this position in the file.

Resize shared memory Unlike System V, POSIX provides `mremap()` function which expands or shrinks an existing mapping and, as such, changes the range of corresponding virtual addresses.

Detache shared memory The memory mapping of specified address range is deleted using `munmap()` function. The POSIX standard [1] states[9] that entire pages containing any part of the address range given to `munmap()` function are removed, and access to any of those pages causes SIGSEGV signal. However, unmapping does not remove the shared object itself. This is done by `shm_unlink()`. Example 4.3 shows a typical usage of mentioned functions.

4.2.2 Shared Memory on Windows

Shared memory on Windows [30] is very similar to POSIX and is based on file mapping.

Create/open shared memory In order to use shared memory, processes need to obtain a handle. It can be either associated with a file or backed by the system paging file (similar to `MAP_ANONYMOUS` flag in POSIX). In both cases, the file mapping (and an associated handle) is created using function `CreateFileMapping()`. When a file-associated mapping is used, a process may obtain a handle of the object which was previously created by

[8] https://code.woboq.org/userspace/glibc/sysdeps/posix/shm_open.c.html
[9] https://pubs.opengroup.org/onlinepubs/9699919799/functions/munmap.html

a different process using function `OpenFileMapping()`. The unnamed memory-mapping, which is not backed by any file, cannot be opened, but the handle may be inherited by children processes.

The mapping of a file into a process' address space is done using function `MapViewOfFile()` which takes the handle, size of memory mapping, and offset as an argument. The function returns a pointer to the beginning of the mapped shared memory.

```
int main() {
  // Open or create shared memory "/somename" with read-write permissions
  const char* name = "/somename";
  int oflag = O_CREAT | O_RDWR;
  mode_t mode = 0666;
  int shm_fd = shm_open(name, oflag, mode);

  // Set shared memory's size to 1024B
  size_t size = 1024;
  ftruncate(shm_fd, size);

  // Map the shared memory file to virtual address space
  int prot = PROT_WRITE;
  int flags = MAP_SHARED;
  off_t offset = 0;
  void* ptr = mmap(NULL, size, prot, flags, shm_fd, offset);

  ...

  munmap(ptr, size);
  shm_unlink(name);
}
```

Listing 4.3: Using POSIX shared memory.

Resize shared memory Windows does not provide any function to resize shared memory mapping and, as such, change the range of valid virtual addresses.

Detache shared memory The mapping of a file into a process' address space is destroyed using function `UnmapViewOfFile()`. The whole range of addresses previously used for accessing the shared memory is invalidated. The memory mapped objects itself is closed using function `CloseHandle()`.

4.2.3 Token Representation of Shared Memory

While monitoring operations with shared memory, ANaConDA will keep a vector of so-called tokens. Each token represents a particular shared memory which is accessible from the process' address space. The token has to contain information about the range of virtual addresses assigned to this shared memory and necessary information for the shared memory identification among processes. Provided this vector of tokens, the translation Algorithm 1 is quite straightforward.

Algorithm 1 Translation algorithm.

Input: Virtual address a, a vector vc of tokens t.

Output: If a identifies a segment in a shared memory, algorithm returns segment's identifier which is identical for all processes.

1: **if** $\exists t \in vc : a \in t.range()$ **then**
2: $h = t.hash()$
3: $offset = (a - t.base()) + t.offset()$
4: return $h + offset$
5: **end if**

Line 2 is a critical step for translation. The hash has to be created from an identifier of the shared memory which distinguishes shared memories from each other and, at the same time, identifies the same memory among different processes. The particular form of identifier is dependent on the type of shared memory. System V shared memory is identified by `shmid`. The algorithm does not need to distinguish `IPC_PRIVATE` flag in System V as it also identifies shared memory by `shmid`. POSIX shared memory is identified by a path-like identifier given to `shm_open()`. Anonymous mapping does not have any identifier. However, as all processes using the anonymous mapping do use the same virtual addresses for accessing the shared memory, no translation is needed. Windows shared memory is identified either by a file's pathname or by a handle in case of unnamed mapping.

Line 3 computes an offset from the beginning of the shared memory. It uses the virtual address of the beginning ($t.base()$) and also $t.offset()$. This value is always 0 for System V shared memory but for POSIX and Windows shared memory may be other than zero for two reasons:

- An offset other than zero was passed to `mmap()` or `MapViewOfFile()` function.
- The `munmap()` function was used to unmap a range of addresses in the middle of the original range as shown in Figure 4.3. The blue parts will be represented by two separate tokens with different base addresses 20 and 80. However, when a process accesses e.g. virtual address 100, the translated representation should take into consideration the base address 20 in the form of $t.offset()$. The calculation would be $offset = (100 - 80) + 60 = 80$ which is indeed correct with regard to base address 20.

As mentioned in the previous section, `munmap()` removes and invalidates entire pages and not only given address range. The manual page for the function states that the base address of the unmapped range needs to be a multiple of the page size as well. The correct size of the invalidated address range can be determined by rounding up the size to a multiple of the page size.

4.3 Happens-Before Relation for Semaphores

The happens-before relation described earlier in Section 2.3 is currently defined for locks (mutexes) only. These are similar to binary semaphores (with maximal value 1). However, for the happens-before relation definition, another condition is usually required, i.e. that lock can be released only by the thread or process currently holding it[10]. In terms of binary

[10]Not all implementations require this condition. E.g. `pthreads` mutexes do but `PIN_Mutex` does not.

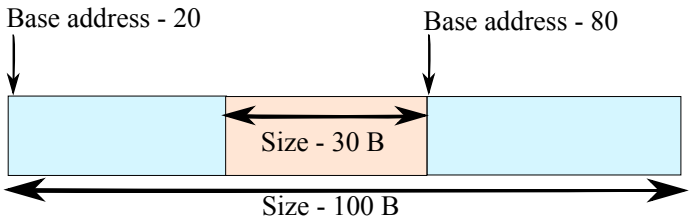


Figure 4.3: A demonstration of possible `munmap()` result. The former segment of mapped shared memory contained all three parts and had 100B in total. After `munmap()`, the orange part of 30B was removed from virtual address space. Addresses used in the figure are only illustrative as real addresses in real systems are page aligned.

semaphore, this means that a thread or a process performing down operation should be the one performing up operation. Neither of these conditions applies to semaphores which makes their usage more general and happens-before relation definition and monitoring more complicated. As we are currently not aware of any algorithm for construction of happens-before relation when using semaphores, it was proposed during the work on this thesis and will be described in this section. For now, only operations changing the semaphore's value by 1 are considered. Also, the relation will be described for multi-process programs because they commonly use semaphores, but it applies to multi-threaded programs as well.

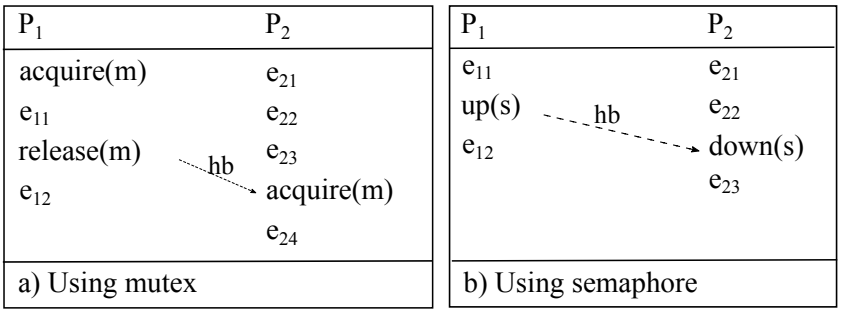


Figure 4.4: Forming a happens-before relation between two processes using a) mutexes or b) semaphores.

4.3.1 Forming the Relation

Figure 4.4 shows that the happens-before relation between two events in different threads or processes is formed when synchronisation occurs. When using a mutex, the relation forms when a process acquires mutex previously released by another process. Theoretically, happens-before relation for semaphores is very similar. As shown in the figure, it should be formed between two processes performing up and down operation over the same semaphore as well. The problem is when multiple processes are using the same semaphore.

In the following explanation, the down operation will be replaced by the `wait` and `continue`. The `wait` phase occurs when a process performs down operation over a semaphore with value 0 and is suspended as a result. The `continue` phase means that the process decremented semaphore's value and successfully returned from the down operation. This can happen either when the process performed down operation over a semaphore with a value greater than zero or when it was suspended and released by another process. As

they read and change semaphore's value which is shared, the `wait` phase, `continue` phase, and up operation are all performed in a critical section of the semaphore. If the process is suspended at the end of `wait` phase, it needs to leave the critical section and enter it again at the beginning of `continue` phase.

P_1	P_2	P_3	P_4
wait(s)	e_{21}	e_{31}	e_{41}
	wait(s)	e_{32}	e_{42}
		up(s)	e_{43}
		e_{33}	up(s)
	continue(s)		
continue(s)			

Figure 4.5: A trace of a program with 4 processes which use the same semaphore s. Processes P_1 and P_2 are queued in a waiting queue and released by processes P_3 and P_4.

Let's consider the situation shown in Figure 4.5. Processes P_1 and P_2 perform down operation over the semaphore s with value zero, and both are suspended. Process P_3 performs up operation and releases one queued process. However, the scheduler does not assign computational time to this process, and process P_4 performs up operation instead. Then, process P_2 may continue, and, after that, process P_1 as well. The problem is to determine the processes between which the happens-before relation is formed. Two questions need to be considered:

1. Which process will be released?
2. When a process performs up operation, should it form the relation with all waiting processes or with the released one only?

Ad 1. As mentioned in Section 3.6, the queue of waiting processes does not have to be FIFO. The released process is chosen by a scheduler. There are two ways of how the relation should reflect this:

1. Cover all possibilities – Let's consider that the happens-before relation forms between processes p and q where p performed up operation and released q from waiting. As process P_3 may release process P_1 or P_2, the relation covering all possibilities is:

$$\begin{aligned}&((P_3.up \prec_{hb} P_1.continue) \wedge (P_4.up \prec_{hb} P_2.continue)) \vee \\ &((P_3.up \prec_{hb} P_2.continue) \wedge (P_4.up \prec_{hb} P_1.continue))\end{aligned} \tag{4.1}$$

   This approach would probably be used in model checking or other methods which need to be sound. However, dynamic analysis is not sound, and this approach would, not only, be difficult to implement but would also considerably degrade the analysis performance. For these reasons, the second approach was used.

2. Cover only the witnessed situation – Dynamic analysis monitors a particular execution where precisely one of the possible situations occurs. Happens-before relation will

reflect this situation only. When considering trace in Figure 4.5 and situation when process P_3 released process P_2, the formula describing happens-before relation is:

$$(P_3.up \prec_{hb} P_2.continue) \wedge (P_4.up \prec_{hb} P_1.continue) \tag{4.2}$$

The second approach conforms to the usage of happens-before relation in dynamic analysis. Let's consider the situation in Figure 4.6. The dynamic analysis covers only the witnessed situation where the blue section happened-before the red section. However, different context switching could change their order as they are not synchronised. Choosing to cover only one possible situation is very similar.

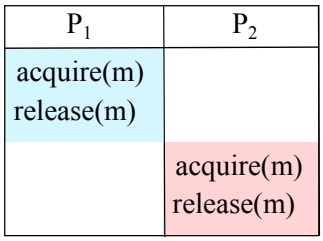


Figure 4.6: A trace of a program where context switching determined the order of two unsynchronised blocks.

Ad 2. Considering that process P_2 was released first, two approaches are possible:

1. Forming the relation with all suspended processes – As the $P_3.up$ occurred while P_1 and P_2 were suspended, and $P_4.up$ occurred after P_2 was no longer in the waiting queue, the formula would be:

$$(P_3.up \prec_{hb} P_1.continue) \wedge (P_3.up \prec_{hb} P_2.continue) \wedge (P_4.up \prec_{hb} P_1.continue) \tag{4.3}$$

2. Forming the relation with the released process only:

$$(P_3.up \prec_{hb} P_2.continue) \wedge (P_4.up \prec_{hb} P_1.continue) \tag{4.4}$$

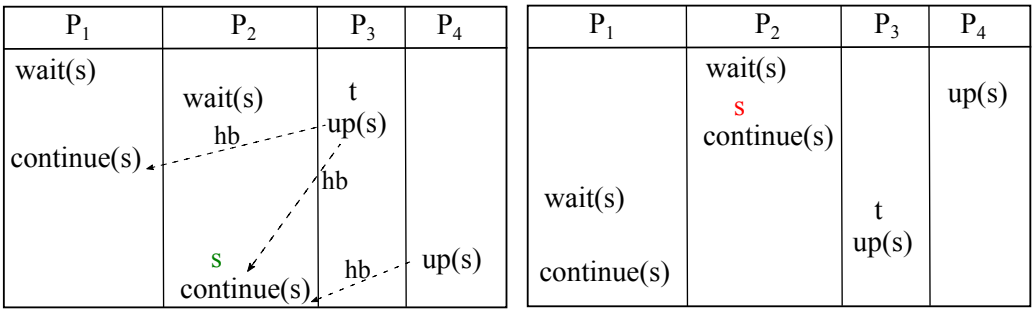


Figure 4.7: A trace of a program where process P_3 performs operation t which cannot be preceded by operation s performed by process P_2. In the left trace, the condition holds. However, it is not assured by synchronisation, as shown on the right side. Forming the happens-before relation between process P_3 and P_2 would not be correct.

Again, the second approach was used. The first approach implies that one semaphore can be used to synchronise two pairs of processes which is not true. Let's consider the situation in Figure 4.7. Process P_3 is performing operation t which cannot be preceded by operation s performed by process P_2. However, the synchronisation does not assure this order as shown on the right side of the figure. When using the first approach, this error will not be detected.

P_1	P_2	P_3	P_4
e_{11} e_{12} e_{13} e_{14} e_{15} continue(s)	e_{21} e_{22} e_{23} e_{24} continue(s)	e_{31} e_{32} up(s) e_{33}	e_{41} e_{42} e_{43} up(s)

Figure 4.8: A trace of the same program as in the Figure 4.5 but up operations are performed before down operations making them proceed without waiting.

Above, only situations when an up operation occurs while the waiting queue is not empty were considered. However, even if the down operation proceeds immediately and the process is not suspended, it should also form the relation with corresponding up. Let's consider situation in Figure 4.8. If the semaphore's initial value was zero, operation up performed by process P_3 allowed operation down in process P_2 to proceed. Similarly, up operation in process P_4 affected operation down in process P_1.

The definition of happens-before relation presented in 2.3 can be extended with another two conditions for relation between two events. Formally, event $e_j \in \tau$ happened-before event $e_k \in \tau$ ($e_j \prec_{hb} e_k$) when $j < k$ and one of the following holds:

1. Event e_k is `continue` phase of previously suspended process p, and event e_j is up operation which released process p.
2. Event e_k is `continue` phase of non-blocking down operation on semaphore s, and event e_j is up operation on semaphore s which:
   2.1. did not release any suspended process and
   2.2. when considering a restriction of a trace τ to trace $\{e_i, \ldots, e_l\}$ where $\forall e_n \in \{e_i, \ldots, e_l\} : e_n \in \tau \wedge ((e_n$ is operation up on semaphore $s \wedge n < j) \vee (e_n$ is `continue` phase on semaphore $s \wedge n < k))$, operation e_j in a trace $\{e_i, \ldots, e_l, e_j\}$ would increment the value of semaphore s from 0 to 1.

The second condition specifies which up operation made the down operation proceed without waiting. Intuitively, it is an operation such that, if it does not happen, the down operation will be blocking. In other words, if the operation happens just before the down operation, it changes semaphore's value from 0 to 1. As we are looking for the first operation up which satisfies this condition, the trace is restricted only to operations up which occurred in the trace before the event e_j. Let's consider the following three examples:

1. Semaphore s is initialized to value 0 and $\tau = \{P_1.up, P_2.up, P_3.continue, P_4.continue\}$. When looking for the corresponding up operation for event $P_3.continue$, the restriction of the trace is either empty set (when $e_j = P_1.up$) or $\{P_1.up\}$ (when $e_j = P_2.up$).

The second situation does not satisfy condition as $P_2.up$ changes semaphore's value from 1 to 2. Thus, $P_1.up \prec_{hb} P_3.continue$. $P_1.up$ is not corresponding up operation for $P_4.continue$, because the restriction of a trace would be $\{P_3.continue\}$ and operation $P_1.up$ would release suspended process P_3. When considering $P_2.up$, the restriction is $\{P_1.up, P_3.continue\}$ and in such a case the $P_2.up$ would increment semaphore's value to 1. Thus, $P_2.up \prec_{hb} P_4.continue$.

2. Semaphore s is initialized to value 1 and $\tau = \{P_1.continue, P_2.up, P_3.continue\}$. Event $P_1.continue$ does not form happens-before relation with any up operation as its proceeding was caused by the initial value of the semaphore and no synchronisation was necessary. The event $P_3.continue$ forms relation with event $P_2.up$ as it satisfies the second condition.

3. Semaphore s is initialized to value 0 and $\tau = \{P_1.wait, P_2.up, P_1.continue, P_3.up, P_4.continue\}$. As $P_2.up$ releases process P_1 from waiting, the happens-before relation $P_2.up \prec_{hb} P_1.continue$ is formed. Event $P_4.continue$ cannot form relation with event $P_2.up$ because it released a suspended process. Event $P_3.up$ satisfies the second conditions, thus $P_3.up \prec_{hb} P_4.continue$.

When applying these conditions to the traces in Figures 4.5 and 4.8, the formula describing happens-before relation in both cases is:

$$(P_3.up \prec_{hb} P_2.continue) \wedge (P_4.up \prec_{hb} P_1.continue)$$

The proposed approach and very similar definition[11] are also presented in [2]. However, this article considers only blocking down operations (condition 1.) and does not propose an algorithm for relation forming during dynamic analysis.

4.3.2 Dynamic Analysis of the Relation

The happens-before relation will be formed in the `continue` phase of the process performing down operation. This means that a corresponding up operation needs to be found. It is either the up which released process from the waiting queue or which made it possible to proceed without waiting. Both of these issues can be solved by using a queue of previously occurred up operations. The idea is that when a process performs up operation over the semaphore s, a corresponding note will be added to the associated FIFO queue. When a process performs `continue` over semaphore s, it will find the first up suitable for happens-before relation in the queue.

The note about up operation should contain current vector clock of the process and information about the current waiting queue. The vector clock is needed for the formation of happens-before relation. It is the same approach as described in Section 2.3.1 where a lock contains information about current vector clock of a thread which released it. The information about the current waiting queue is necessary for finding a corresponding up. Let's consider the situation in Figure 4.9. If the queue of up operations will not contain information about the current waiting queue, the `continue` phase of the process P_2 will form a relation with the up operation of P_3, as it will be the first item in the queue. But this is not correct, as the up operation occurred even before P_2 performed down operation,

[11] The only difference is that event e_k is not the `continue` phase directly but event following the `continue` phase. This difference does not affect the usage of happens-before relation for detection of concurrency-related errors.

thus it could not be released by it. If a note about up operation will contain information about the current waiting queue, process P_2 forms a relation with the first meaningful up, i.e. the one where either P_2 was in the waiting queue or the queue was empty.

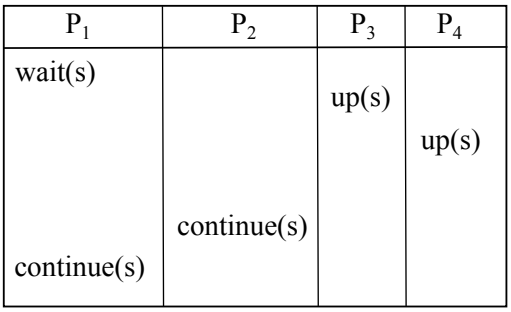


Figure 4.9: A trace showing the necessity of information about current waiting queue to correctly form a happens-before relation.

Figure 4.9 shows another problem with dynamic analysis, i.e. processes do not need to perform `continue` phase in the same order as they were released from waiting. Up operation performed by process P_3 surely released process P_1. However, ANaConDA is not able to detect this and sees the `continue` phase only when the released process is assigned computational time. As shown in the figure, released processes do not necessarily get the computational time in the same order as they were released. This does not cause problems for situations where only one process is in the waiting queue (e.g. Figure 4.9) thanks to information about the queue in the notes about up operations. But it can appear as a problem for the situation in Figure 4.5. There, the happens-before relation can be either $((P_3.up \prec_{hb} P_1.continue) \wedge (P_4.up \prec_{hb} P_2.continue))$ (when process P_3 releases process P_1) or $((P_3.up \prec_{hb} P_2.continue) \wedge (P_4.up \prec_{hb} P_1.continue))$ (when process P_3 releases process P_2). However, even if ANaConDA cannot detect which process was actually released, the information about order of `continue` phases is sufficient. If we will assume, that the process first performing `continue` phase is the one who was released first, the happens-before relation will still be correct. Considering the example in Figure 4.5, if process P_3 actually released process P_1 and process P_4 released process P_2, happens-before relation will be $((P_3.up \prec_{hb} P_2.continue) \wedge (P_4.up \prec_{hb} P_1.continue))$. Even though this does not reflect the particular situation which happend, it is still one of the possible outcomes of this synchronisation (see Equation 4.1).

Initialization needs to be considered as well. As the initial value v of a semaphore may be greater than zero, the first v down operations should not form happens-before relation with any up operation. Thus, the queue should contain v items, e.g. with pid 0, to reflect the initial value. This will also solve the second problem with looking for the corresponding up operation which allowed operation down to proceed without waiting. For the first v downs, no up was necessary, but each other continue had to be preceded by the first up in the queue.

Algorithms 2, 3 and 4 describe the formation of happens-before relation using vector-clocks in corresponding callbacks. Each algorithm uses a representation of a semaphore with the queue for up operations. This queue is created during semaphore's initialization (Algorithm 2). If its initial value is greater than zero, a corresponding number of items with pid 0 are added to the queue. During up operation (Algorithm 3), process' pid, its current vector clock, and current waiting queue are added to the queue of up operations,

and process' logical time is incremented. The happens-before relation is formed during the `continue` phase (Algorithm 4) with the first up in the queue which occurred when the waiting queue was empty or contained the process' pid (lines 1 and 2). If the pid of up is 0, it was not an up operation but an initial value of the semaphore and the relation is not formed. Otherwise, the vector clock of the process is updated using join operation as described earlier in 2.3.1. After that, the note about the up operation is removed from the queue (line 6). The following section describes problems with semaphores' monitoring and obtaining necessary information about them. As a result, the current waiting queue of the semaphore provided to callbacks is not the actual queue hold in the operating system, but its representation created in ANaConDA. As ANaConDA cannot decide, which process will be released by up operation, it is removed from the representation of the queue during its `continue` phase. This means that all up operations which occurred before this phase contain the process' pid in the information about the waiting queue (although the process was not actually in the queue because previous up released it). Thus, after a process finds the corresponding up operation in the queue $s.q$ for happens-before relation, it also needs to remove itself from waiting queues saved in notes about following up operations (lines 10–14). All three algorithms are demonstrated on a particular traces in Figures 4.10, 4.11, 4.12 and 4.13.

Algorithm 2 Algorithm for initialization.

Input: Semaphore s, initial value v.

Output: Queue of up operations $s.q$ where its length is equal to v.

1: create empty queue $s.q$
2: **while** $v > 0$ **do**
3: $pid = 0$
4: $vc = VectorClock()$
5: $waiting\_queue = []$
6: $s.q.add(pid, vc, waiting\_queue)$
7: $v = v - 1$
8: **end while**

Algorithm 3 Algorithm for up operation.

Input: Semaphore s, process' pid, current vector clock vc of the process, current waiting queue $waiting\_queue$ of the semaphore.

Output: Updated semaphore's queue $s.q$ containing information about the occured up operation.

1: $s.q.add(pid, vc, waiting\_queue)$
2: $vc.increment(pid)$

4.4 Semaphore Monitoring

Currently, the happens-before relation monitoring and vector clocks' implementation is up to analysers and is not provided by ANaConDA framework. For consistency, the design proposed in the previous section will also be implemented in analysers. However, ANaConDA framework should provide callbacks with all necessary information about synchronisation operations for (not only) the relation monitoring. These include identifiers of the process,

Algorithm 4 Algorithm for continue phase.

Input: Semaphore s, process' pid, current vector clock vc of the process, current waiting queue $waiting\_queue$ of the semaphore.

Output: Updated vector clock vc of the process pid and updated queue of up operations $s.q$.

```
1: for up ∈ s.q do
2:   if up.waiting_queue.empty() ∨ pid ∈ up.waiting_queue then
3:     if up.pid ≠ 0 then
4:       vc.join(up.vc)
5:     end if
6:     s.q.remove(up)
7:     break
8:   end if
9: end for
10: for up ∈ s.q do
11:   if pid ∈ up.waiting_queue then
12:     up.waiting_queue.remove(pid)
13:   end if
14: end for
```

thread and semaphore, semaphore's value, count of waiting processes, and a value which will be added to or subtracted from the semaphore's value. Analysers will be provided with callback pairs (before, after) for the following operations: init, up, down. These callbacks correspond to callbacks provided for locks and they will not distinguish the particular implementation of semaphores (currently POSIX or System V, eventually also Windows). ANaConDA itself will register callbacks for particular system calls or functions, handle information and provide them to analysers in the same way for both types of semaphores.

4.4.1 Obtaining Information About Semaphores

As operations over semaphores do not provide information about their value and count of waiting processes, ANaConDA will have to obtain it in the corresponding callback. Semaphore's value can be found out by using `sem_getvalue()` (POSIX) or `semctl()` (System V) function. As a process of the analyser and monitored process do not share their address space, the function has to be called in the application context which leads to significant performance downgrade. For this reason, we have decided to create a structure representing the semaphore and mirroring its value in the ANaConDA. Thus, ANaConDA will also mirror the operations changing semaphore's value and provide the information to callbacks. Callbacks will receive:

- `sem` – A pointer to object representing the semaphore. It contains semaphore's identifier (virtual or translated address for POSIX semaphores, identifier of set and semaphore's order in the set for System V semaphores), a flag determining whether the semaphore is shared and the identifier is translated or not and the identifier is a virtual address, semaphore's current value, and the current count of waiting processes. The current values change between before-event and after-event callbacks.

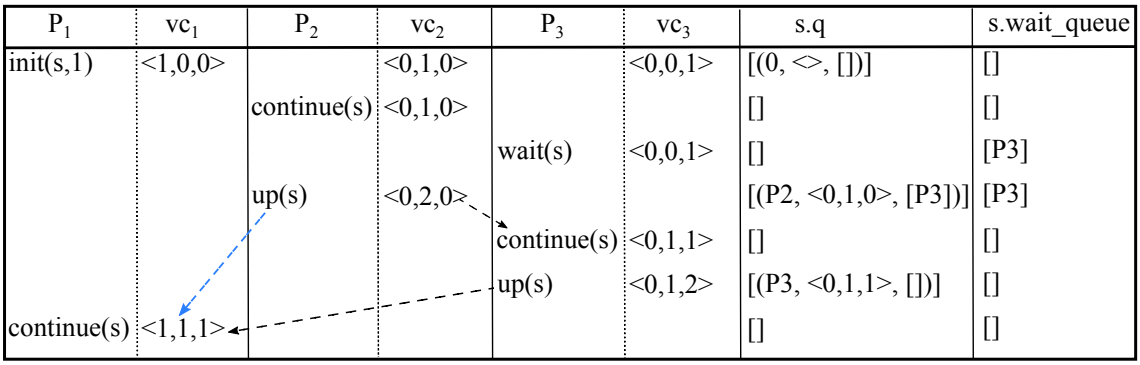


Figure 4.10: Three processes use the same semaphore. Corresponding vector clocks and up queue are listed as well. Their values represent the situation **after** performing the algorithm for the operation which occurred in the monitored program. Arrows show happens-before relation which is formed using join operation over vector clocks. The blue arrow demonstrates the transitivity of the relation.

P_1	vc_1	P_2	vc_2	P_3	vc_3	P_4	vc_4	s.q	s.wait_q
init(s,0)	<1,0,0,0>		<0,1,0,0>		<0,0,1,0>		<0,0,0,1>	[]	[]
wait(s)	<1,0,0,0>							[]	[P1]
		wait(s)	<0,1,0,0>					[]	[P1,P2]
				up(s)	<0,0,2,0>			[(P3, <0,0,1,0>, [P1,P2])]	[P1,P2]
						up(s)	<0,0,0,2>	[(P3, <0,0,1,0>, [P1,P2]), (P4, <0,0,0,1>, [P1,P2])]	[P1,P2]
		cont.(s)	<0,1,1,0>					[(P4, <0,0,0,1>, [P1])]	[P1]
cont.(s)	<1,0,0,1>							[]	[]

Figure 4.11: This trace is the same as in Figure 4.5. It demonstrates that algorithms described above form the happens-before relation as proposed in previous section. I.e. $P_3.up \prec_{hb} P_2.continue \wedge P_4.up \prec_{hb} P_1.continue$ (see Equation 4.4).

- `value` – Semaphore's value **before** the operation changed its value. Both callbacks (before and after) will receive the same value[12].
- `waiting` – Similarly to value, this count represents the number of suspended processes **before** the operation down suspends the process receiving callback or wakes up any suspended process.
- `change` – A number added to or subtracted from the semaphore's value.

To correctly mirror the semaphore's value, the triplet *[before-event callback – event – after-event callback]* needs to be atomic. If it could be interleaved with the execution of other processes performing operations with the same semaphore, the values provided to callbacks might not be the same as the semaphore's value at the moment of monitored operation. To ensure atomicity, both callbacks need to be inserted into the critical section of operations. As up and down functions change the value of a shared variable, the change itself is guarded by a low-level synchronisation mechanism such as futex.

[12]Current value is stored in the object representing semaphore and can be obtained as well.

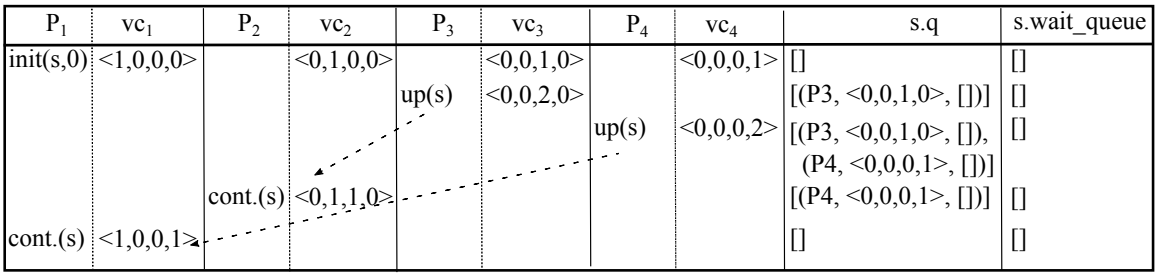


Figure 4.12: This trace is the same as in Figure 4.8 where two up operations occur before down operations thus making them proceed without blocking. However, the happens-before relation is still the same as in Equation 4.4

.

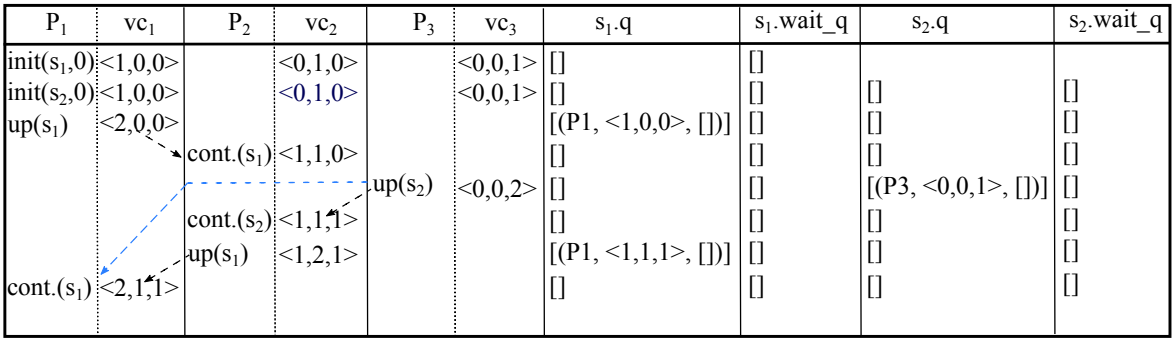


Figure 4.13: Three processes use two semaphores (s_1, s_2) for synchronisation. Process P_2 synchronises with process P_1 at first, then with process P_3 via different semaphore. When it synchronises with the process P_1 again, the transitivity of happens-before relation (blue arrow) correctly shows that processes P_1 and P_3 are synchronised as well (even though via different processes and semaphores).

As mentioned, ANaConDA needs to register callbacks for functions specific to each implementation of semaphores. As the usage of POSIX and System V semaphores differs, the next two sections summarize how each of these types will be monitored.

4.4.2 Monitoring of POSIX Semaphores

Initialization A semaphore's representation in ANaConDA needs to be created upon its initialization in the monitored program. Function for initialization is dependent on the semaphore's type. Example 4.4 shows pseudocode of callback for `sem_init()` function for unnamed semaphores. This callback is executed in ANaConDA framework and is used for initialization of semaphore's representation. The representation is created either in heap or in shared memory depending on argument `pshared` (as described in Section 3.6). Semaphore's identifier is address (either virtual or translated). Analyser's before-initialization and after-initialization callbacks receive information about semaphore mentioned above.

Example 4.5 shows pseudocode of callback for `sem_open()` function for named semaphores. Opposed to unnamed semaphores, this function does not take semaphore as an argument but returns it. It means that before-initialization callback cannot provide information about this semaphore. As we aim to preserve callback pairs, analyser's callback

for before-initialization will receive special value representing unknown semaphore. When function `sem_open()` returns, new semaphore was created or an existing one was opened. To determine which situation occurred, ANaConDA will try to find the semaphore's representation. If a different process already created it, no changes are needed. If the representation is not found, it needs to be created. The semaphore's value is obtained using `sem_getvalue()` function. As execution of `sem_open()` function in two different processes may interleave, the value provided to the function as an argument cannot be used. It is impossible to determine, which process actually created the semaphore and which value was used for its initialization.

Operation up Example 4.6 shows pseudocode of callback handling operation up over a semaphore. This callback is executed for `sem_post()` function. Although it is not possible to change the value of POSIX semaphore by different count than 1, the same callback will be used for System V semaphores as well thus argument `change` is necessary. The implementation of the callback is quite straightforward. Semaphore's value is incremented by given value and analyser's callback receives its state **before** operation. The count of suspended processes does not change as ANaConDA is not able to determine how many processes will be released (if `change` is greater than 1).

```
before_sem_init(sem, pshared, value) {
  if (pshared == 0) { // Semaphore is shared between threads only
    Semaphore* s = new Semaphore();
    s->id = sem;
    s->value = value;
    s->waiting = 0
  }
  else { // Semaphore is shared among processes
    Semaphore* s = new SharedMemorySemaphore(translate(sem));
    s->id = translate(sem);
    s->value = value;
    s->waiting = 0;
  }

  save(s); // Save semaphore for after_sem_init callback
  // Call callback in analyser and provide process', thread's identifier,
  // semaphore's representation, and its current value
  analyser_before_init_callback(pid, tid, s, s->value);
}

after_sem_init() {
  analyser_after_init_callback(pid, tid, s, s->value);
}
```

Listing 4.4: Initialization of representation of unnamed semaphore.

```
before_sem_open(name) {
  save(name); // Save name for after_sem_open callback
  // Call before init callback in analyser for unknown semaphore
  analyser_before_init_callback(pid, tid, UNKNOWN, 0);
}

// Argument sem is return value of sem_open() function
after_sem_open(sem) {
  Semaphore* s = search_shared_memory(name);

  if (s == NULL) { // Semaphore's representation has not been created yet
    s = new SharedMemorySemaphore(name);
    s->id = name;
    s->value = get_value(s);
    s->waiting = 0;
  }

  analyser_after_init_callback(pid, tid, s, s->value);
}
```

Listing 4.5: Initialization of representation of named semaphore.

```
before_up(sem, change) {
  Semaphore* s = find_representation_of_semaphore(sem);

  // Save semaphore's state before perfoming operation up
  value = s->current_value;
  waiting = s->current_waiting;
  save(s, value, waiting, change); // Save information for after_up callback

  s->current_value += change; // Perform operation up

  // Call analyser's callback with information about semaphore before
  // operation was performed
  analyser_before_up_callback(pid, tid, s, value, waiting, change);
}

after_up() {
  // Call analyser's callback with information about semaphore before
  // operation was performed
  analyser_after_up_callback(pid, tid, s, value, waiting, change);
}
```

Listing 4.6: Change of information in semaphore's representation upon operation up.

Operation down Example 4.7 shows pseudocode of callback handling operation down over a semaphore which is very similar to operation up. The before-down callback needs to determine whether the process will be suspended or not and update information accordingly. If the process will be suspended, the before-callback only increments the count of

suspended processes but semaphore's value will change in the after-down callback (which is, in fact, the `continue` phase). If the process will not be suspended, the semaphore's value may be changed either immediately or in the after-down callback[13].

```
before_down(sem, change) {
  Semaphore* s = find_representation_of_semaphore(sem);

  // Save semaphore's state before perfoming operation down
  value = s->current_value;
  waiting = s->current_waiting;
  suspended = (change > s->current_value);

  // Save information for after_down callback
  save(s, value, waiting, suspended, change);

  if (suspended)
    s->current_waiting += 1;
  else
    s->current_value -= change;

  // Call analyser's callback with information about semaphore before
  // operation was performed
  analyser_before_down_callback(pid, tid, s, value, waiting, change);
}

after_down() {
  if (suspended) {
    s->current_waiting -= 1;
    s->current_value -= change;
  }

  // Call analyser's callback with information about semaphore before
  // operation was performed
  analyser_after_down_callback(pid, tid, s, value, waiting, change);
}
```

Listing 4.7: Change of information in semaphore's representation upon operation down.

4.4.3 Monitoring of System V Semaphores

The main difference between System V and POSIX semaphores is that `semctl()` and `semop()` functions may perform operations on the whole set of semaphores (thus multiple operations on different semaphores at once). The problem is, what information should analysers receive. Two approaches are possible:

1. Create special callbacks for operations over multiple semaphores.

[13] As both callbacks will be inserted in semaphore's critical section, no other process can perform operation over the same semaphore until both callbacks are finished.

2. Divide the operation into suboperations and call analyser's callback for each suboperation separately.

Both approaches have their advantages and disadvantages. Calling one callback for the whole operation is more accurate. However, analysers would still need to handle each suboperation separately, and it would complicate their implementation. When an operation is divided into multiple callbacks, information about the connection between suboperations is lost. However, as analysers which are currently available in ANaConDA framework are not able to use this information and happens-before relation does not need either, the second approach was used. If a need to monitor these operations as a whole arises in the future, it can be either done in analysers by monitoring functions `semctl(), semop()` directly or added to ANaConDA framework later.

Dividing one operation into multiple callbacks brings another issue with interleaving of before-event and after-event callbacks. Until now, when an event occurred in a thread of monitored program and caused execution of before-event callback, the same thread could only execute after-event callback for the same event. Executing multiple before-event callbacks without closing the event with after-event callback first was possible only when the event was an execution of a recursive function. However, executing callbacks for each suboperation separately will result in the same behaviour. Suboperations will cause execution of multiple before-event callbacks. This will not cause problems for happens-before relation monitoring, but analyser's developer should be aware of this possible situation.

Initialization POSIX combines semaphore's creation and initialization into one function. System V semaphores do it separately. Semaphore's representation is created in callback for `semget()` function (see Example 4.8). Implementation of callbacks for this function is very similar to creating a representation of POSIX named semaphore as a semaphore may already exist and the function only opens it. However, there are three differences:

1. The function creates/opens a semaphore set with `nsems` semaphores which is an argument of the function. As operations over the set will be divided into suboperations, each semaphore of the set will have its own representation as well.

2. Callback does not need to obtain semaphore's value. If the function created a new set, it will need to be initialized using `semctl()` function later.

3. As it is not semaphore's initialization, no analyser's callbacks are executed.

System V semaphores are initialized using `semctl()` function with command `SETVAL` or `SETALL`. In a callback, representations of correspondings semaphores are initialized as well and analyser's callbacks are called for each initialized semaphore. The problem is that these commands are not limited to initialization only. They can change the value of previously initialized semaphore and behave as up or down operations. These situations need to be distinguished in the callback. If semaphore was previously initialized, the difference between its current value and a value which was given as an argument to `semctl()` function is counted. A corresponding callback for operation up (Example 4.6) or down (Example 4.7) is executed and the counted value is used as an argument `change`.

```
before_semget(nsems) {
  save(nsems); // Save size of the set for after_semget callback
}

// Argument setid is return value of semget() function
after_semget(setid) {
  for (i = 0; i < nsems; ++i) {
    Semaphore* s = search_shared_memory(setid, i);

    if (s == NULL) { // Semaphore's representation has not been created yet
      s = new SharedMemorySemaphore(setid, i);
      s->id = setid, i;
    }
  }
}
```

Listing 4.8: Initialization of representation of System V semaphore.

Operations The callbacks updating semaphore's representation are the same as presented for POSIX semaphores. The only difference is dividing operation into suboperations (see Example 4.9).

Operation *wait for zero* is System V specific. It is a unique type of synchronisation which cannot be covered by callbacks for up or down operation. Also, processes waiting for zero are queued in a dedicated queue, and all of them are released at once (when the semaphore's value becomes zero). As analysers currently available in ANaConDA framework are not able to use this information and definition of happens-before relation does not consider it either, callbacks for wait for zero operation are currently not supported. The support can be added later in the future if a real example using this synchronisation occurs.

Another System V specific features are `SEM_UNDO` and `IPC_NOWAIT` flags. The former changes semaphore's value once a process finishes. Each process needs to store its local `semadj` variable (see Section 4.4.3) with the value which will be added to semaphore once the process finishes its execution. When this happens, it will be handled similarly as `SETVAL` and `SETALL` commands and appropriate callbacks from analyser will be executed. The `IPC_NOWAIT` flag is more complicated, as it can cause a failure of the whole operation and annul already performed suboperations. If analysers would support this behaviour, they would need to store information about performed operations to be able to annul them when a failure occurs. This approach is very complicated and algorithms and analysers for concurrency-related errors detection usually assume a certain level of program's correctness (such as no-failing synchronisation). For this reason, the flag will currently not be supported and analysers will not be able to annul failed operations. However, it is planned that ANaConDA will be extended with a callback for failed synchronisation. It would make possible to implement an analyser checking whether all synchronisation successfully proceeds and, as such, an assumption for a more complicated algorithm is satisfied.

```
before_semop(setid, sops[], nsops) {
  for (i = 0; i < nsops; ++i) {
    op = sops[i];
    sem = setid, op.sem_num;
    value = op.sem_op;

    if (value < 0)
      before_down(sem, value*-1);
    else if (value > 0)
      before_up(sem, value);
    else
      before_zero(sem);
  }

  save(setid, sops[], nsops); // Save information for after_semop callback
}
```

Listing 4.9: Dividing operation into suboperations. Callback `after_semop()` is not shown as it would be very similar to the `before_semop()` but calling corresponding `after_*` callbacks instead.

Chapter 5

Implementation of Extension for Process Monitoring

This chapter describes the implementation of the proposed design into ANaConDA framework. Namely, shared API which provides shared data types for analysers and solves problems with reallocation, translation algorithm for shared memory monitoring, and monitoring of semaphores are now provided to analysers. For each extension, a new analyser was implemented which demonstrates the usage of new API and is used for automatic tests. Algorithms AtomRace and FastTrack for data race detection were implemented as new multi-process analysers and the latter one uses happens-before relation for general semaphores.

Technologies used for implementation The extension for process monitoring was implemented into ANaConDA framework version 0.4 which is based on Pin tool version 2.14. The implementation is written in the same language as ANaConDA framework, that is C++11, using Boost library version 1.58.0.

5.1 Callbacks for Process Monitoring

ANaConDA framework was extended with callbacks for process-related events, that is fork and termination of a process. Analysers are provided with three registration functions for callbacks related to fork event: `PROCESS_BeforeFork()`, `PROCESS_AfterForkParent()` and `PROCESS_AfterForkChild()`. All of them receive pid of the corresponding process and also an identifier of the thread, as, in the future, combination of process and threads will be considered as well. ANaConDA registers corresponding callbacks for fork event via low-level registration functions provided by Pin tool.

Analysers are provided with one function for process' termination and that is `PROCESS_ProcessFinished()`. This callback receives pid of the finishing process and also pid and thread id of the process/thread which called system call `wait()` or `waitpid()`. These system calls are monitored because Pin tool does not provide a low-level callback for process' termination.

Apart from process related events, callbacks for semaphore operations are now available. I.e. ANaConDA provides registration functions `SEM_BeforeInit()`, `SEM_AfterInit()`, `SEM_BeforeDown()`, `SEM_AfterDown()`, `SEM_BeforeUp()` and `SEM_AfterUp()`. Callbacks for semaphore's initialization receive process' pid, thread's id, a pointer to an object repre-

senting the semaphore and the semaphore's value. As mentioned in Section 4.4.2, before-init callback may receive unknown semaphore. Callbacks for operations receive a count of waiting processes and change of semaphore's value in addition to the information provided to initialization callbacks. Semaphore's value and count of waiting processes represent semaphore's state **before** operation is executed. The extraction of information provided to analysers is implemented according to pseudocodes described in Sections 4.4.2 and 4.4.3.

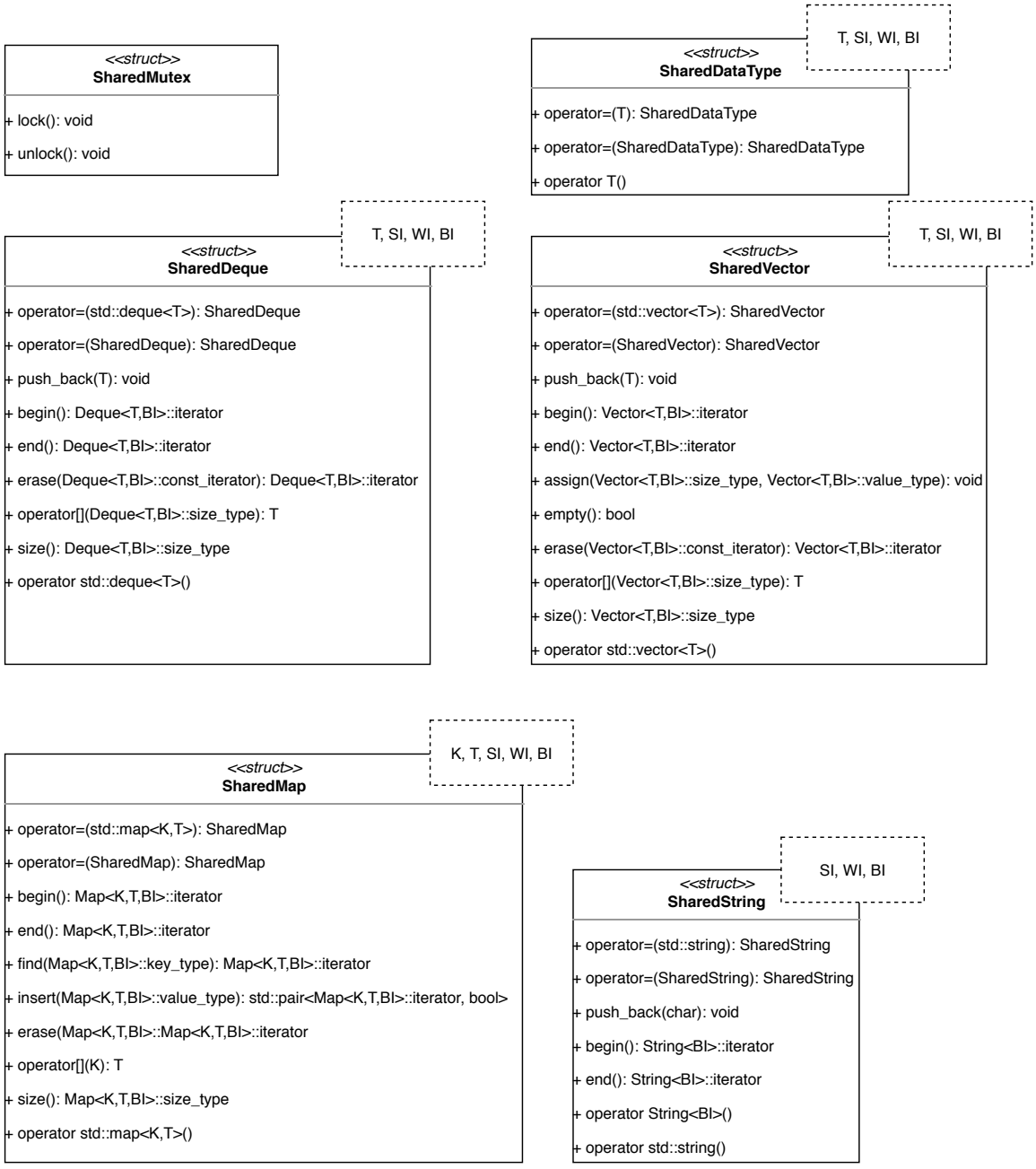


Figure 5.1: Structures representing shared data types which can be used in analysers for interprocess communication.

5.2 Shared API

Figure 5.1 shows shared data types and operations currently provided by the API. The supported data types are the fix sized ones (structure `SharedDataType`), strings, vectors, maps, deques and mutexes. As a different shared container may be required in the future, there is a manual in Appendix F for its implementation.

Shared mutex is `PIN_Mutex` stored in a shared memory thus available for process' synchronisation. This mutex is, in fact, a binary semaphore and does not require to be released by the process who is currently holding it.

Shared containers implement a subset of operations which are provided by their STL equivalent. Implemented operations should be sufficient for most analysers but if a need for another operation arise, there is a manual in Appendix E describing necessary steps for its implementation.

Template arguments All structures for shared data types take at least three template arguments specifying synchronisation (SI), write back/through (WI) and backend (BI). Structures for fix sized data types, vectors and deques also take argument T specifying the data type of elements. The shared map also needs the data type of key (K).

Data types Operations provided by shared containers are performed on a particular container whose data type is dependent on backend implementation[1]. As shared data types shown in Figure 5.1 should be independent on backend implementation, the data type of encapsulated container is represented by a corresponding templated structure (e.g. `Deque<T, BI>`). Currently, all of these templated structures are provided with partial template specialisation for shared memory as it is the only supported backend for now. If a new backend will be implemented in the future, a corresponding specialisations will need to be added as well. However, the usage of API in analysers will not be affected.

Implementation of parameters Parameters for synchronisation, write through/back and backend are implemented by corresponding classes. The general idea was described in Section 4.1 and shown in Example 4.1. Figure 5.2 shows relationship among these classes. More details about implementation are described in Appendix B.

5.2.1 Usage of the API

As structures representing shared data types demand many template arguments, it is easier to define macros or user-defined data types:

```
typedef SharedDataType<int,SI_SYNC,WI_WRITE_THROUGH,BI_SHARED_MEMORY> SInt;
```

This approach also ensures that SI, WI and BI parameters are all defined in one place for all shared variables of the type. Analyser's developer may easily change them, and the usage of shared variables will remain intact.

Constructors for shared data types take one mandatory and one optional argument. The mandatory one is the name of the shared variable, the second is the name of the shared memory in case various channels for communication are needed. Name of the variable is important as it is its unique identifier among all processes. If the developer does not wish

[1] Containers for shared memory need a special allocator and other backend implementations will have different requirements as well.

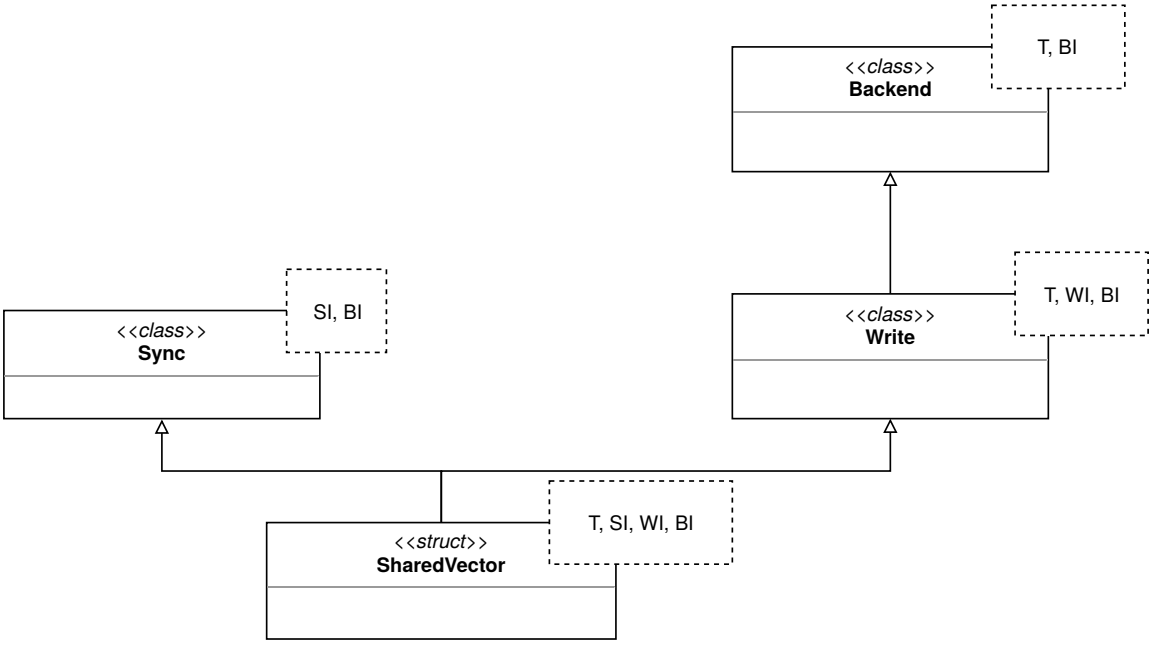


Figure 5.2: A scheme of inheritance used for the implementation of shared data types (there shared vector) and of parameters for synchronisation, write through/back and backend.

to specify the name of shared memory and the variable name should be derived from an instance's name, a macro simplifying construction may be used:

```
#define CREATE_SINT(x) SInt x(#x)
```

Then, a shared variable may be used almost as a typical local variable:

```
CREATE_SINT(a);
a = 5;
int b = 10;
a = b; ...

CREATE_SVECTOR_DOUBLE(v);
v.push_back(1.0);
for (auto &item: v) ...
```

There are two requirements on analysers using shared API. It is necessary to call:

- `UTILS_SetParentID(PIN_GetPid())` in `PLUGIN_INIT_FUNCTION()` in the analyser and
- `UTILS_ClearSharedMemory()` in `PLUGIN_FINISH_FUNCTION()` in the analyser.

These two functions ensure that created shared memories are correctly removed when the last process finishes its execution.

Proc-Analyser

To demonstrate usage of the shared API, new analyser called `Proc-Analyser` was implemented. It does not detect any concurrency-related errors but uses all currently supported

shared data types and operations. It is also used for automatic test for the shared API. Tests are described in Section 6.1.

5.2.2 User-Defined Structures in Shared Memory

As shared API provides support for data types of fixed size and containers, user-defined structures and classes can be shared among processes only if their size is fixed and not changed during the program's execution. Then, the structure can be given as a template argument for `SharedDataType` or for some container. Furthermore, it is important to keep in mind that shared data types provided by ANaConDA framework are meant to be **local**. These structures handle access of one process to a shared variable and, as such, e.g. an instance of `SharedVector` cannot be stored in a shared memory.

Luckily, each user-defined structure can be modified to be fix sized and still take advantage of shared data types provided by ANaConDA. Let's consider structure in Example 5.1.

```
structure Struct {
  std::string str;
  std::vector<int> vc;
  int i;
}
```

Listing 5.1: Dynamic user-defined structure.

If instances of structure `Struct` will be shared among processes, data types of containers in the structure cannot be modified to `SharedString` and `SharedVector`. Instead, the structure will contain names of shared variables. Naturally, names cannot be stored as strings. Thus, their meaningful names are hashed and the hash itself is used as a name. Furthermore, these names need to be unique for each instance of `SharedStruct`. The particular implementation of a user-defined type suitable for sharing among processes depends on its usage. If each process has one instance, the names of shared variables can include pid of the process. A possible solution is shown in Example 5.2. The original structure contains two containers and one numeric attribute. Containers were changed to names of corresponding shared variables but the fix sized attribute does not need to be modified. The example also shows constructor of the structure where unique names of containers are derived from process' pid but it can also be a random number if necessary. Operations over the shared structure remain almost the same, but variables for shared containers need to be declared. The modified structure can be provided as a data type for shared structures provided by shared API. A scheme of shared memory when storing an instance of this structure is shown in Figure 5.3. Shared string and shared vector are instances of corresponding classes from shared API. Thus, they are associated with a corresponding guard which solves problems with reallocation as described in Section 4.1.2.

```
structure SharedStruct {
  std::size_t str_name;
  std::size_t vc_name;
  int i;

  SharedStruct(pid_t pid) {
    str_name = std::hash<std::string>{}
      ("SharedStruct_str"+std::to_string(pid));
```

```
    vc_name = std::hash<std::string>{}
      ("SharedStruct_vc"std::to_string(pid));
  }

  void operation() {
    SharedString str(str_name);
    SharedVector vc(vc_name);
    ...
  }
}

typedef SharedDataType
  <SharedStruct,SI_SYNC,WI_WRITE_THROUGH,BI_SHARED_MEMORY> SSharedStruct;
```

Listing 5.2: Implementation of user-defined structure which is suitable for sharing among processes.

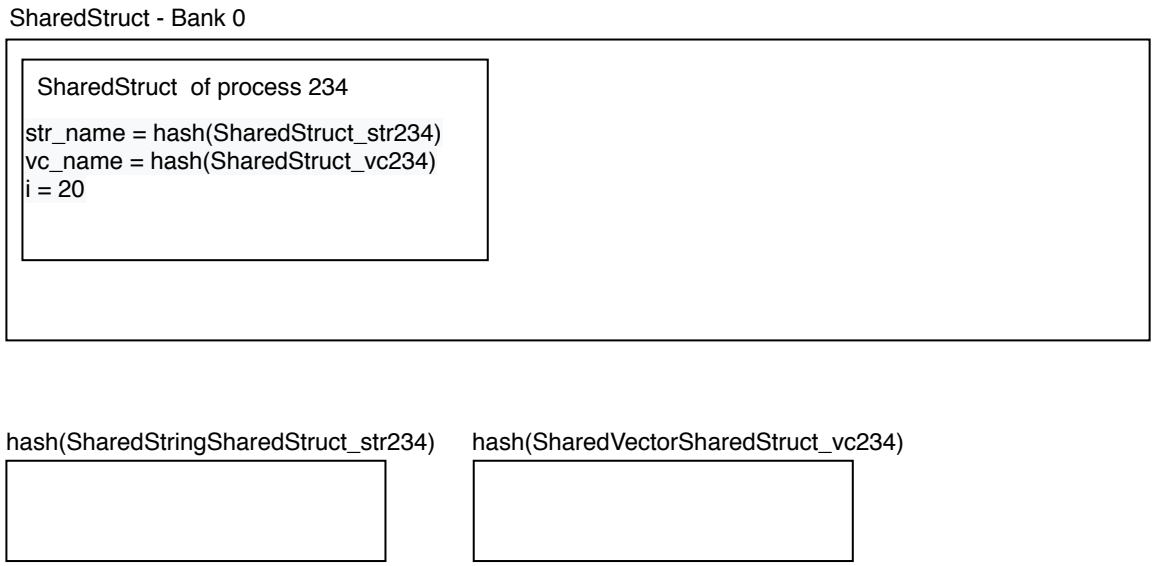


Figure 5.3: A scheme of shared memory when storing an instance of the user-defined structure from Example 5.2.

5.2.3 Clearing Shared Memories in Case of Error

If the analysis does not end correctly and, as such, does not execute the `PLUGIN_FINISH_FUNCTION()`, shared memories will remain in the system. This can happen quite easily when developing new analyser or analysing a faulty program. As persistent shared memories severely affect any following analysis, a tool called `shm_cleaner` was implemented. During analysis, each created shared memory is logged. The tool uses `shm_unlink()` to clear logged shared memories as they are created using `boost` which uses POSIX shared memory.

5.3 Monitoring of Shared Memory

The algorithm for translation of virtual addresses described in Section 4.2 is based on monitoring operations with shared memory in the analysed program and using them to create and manage so-called tokens. Currently, System V and POSIX shared memory is supported. Tokens are managed in callbacks for the following system calls: `shmget()`, `shmat()`, `shmdt()`, `shm_open()`, `mmap()`, `munmap()` and `mremap()`.

5.3.1 Tokens

Figure 5.4 shows structure `MemoryToken` which represents a shared memory. The structure is local for a process and contains the information described in Section 4.2:

- `type` – Helper information about the type of a shared memory (it distinguishes System V, POSIX anonymous and POSIX file-based shared memory).
- `key` – The unique identifier of a shared memory. Its type is dependent on the type of shared memory. Identifiers for each type were described in Section 4.2.
- `base` and `size` – These attributes determine the range of valid virtual addresses corresponding to the shared memory. Data type `ADDRINT` is provided by PIN and used in ANaConDA framework whenever an address needs to be stored. Its size is dependent on the type of architecture (32/64 bit).
- `offset` – Either the offset argument provided to `mmap()` function or an offset created due to unmapping a range of addresses in the middle of the original range (see Section 4.2).
- `translated_base` – A hash derived from the identifier ($t.hash()$ in Algorithm 1) which represents the base address of the shared memory which is identical for all processes.

Method `createHash()` creates `translated_base` from `type` and `key`. Method `translate()` performs lines 2 and 3 from Algorithm 1, thus computes the translated address and returns it. If the type of a shared memory is POSIX anonymous, no translation is needed and the returned address is the same as the one given to the function as an argument. All tokens are stored in a singleton `SharedMemoryMonitor`. The usage of translation in analysers is described in the following section.

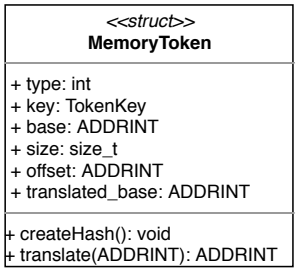


Figure 5.4: A structure representing a shared memory.

5.3.2 Usage of the Translation Algorithm

Analysers are provided with two functions:

- `UTILS_MonitorSharedMemory()` – This function needs to be called in `PLUGIN_INIT_FUNCTION()` if the analyser uses address' translation. The function registers callbacks for shared memory monitoring.
- `UTILS_IsShared()` – This function is used for translation. It takes two parameters where the first one is a virtual address and the second one is an output parameter containing the translated address if the virtual address accesses a shared memory. The function returns `bool` indicating whether the virtual address belongs to a shared memory.

```
Map Access;

beforeAccess(thread t, address a, mode m) {
  if (Access[a] == null) {
   Access[a] = (t, m);
  }
  else if (m == write ||
  Access[a].m == write)
   RACE DETECTED
}

afterAccess(thread t, address a, mode m) {
  if (Access[a].t == t)
    Access[a] = null;
}
```

Listing 5.3: Pseudocode of AtomRace.

5.3.3 AtomRace for Processes

One of the built-in analysers in ANaConDA framework is AtomRace [21] for data race detection in multi-threaded programs. This algorithm is very simple, does not produce false-positives and its modification for multi-process programs requires shared API and translation algorithm. Due to these reasons, it is an ideal analyser to be implemented with support for monitoring of processes.

AtomRace monitors memory accesses and detects data race according to its definition. Thus, data race is detected if two or more threads access the same memory segment and at least one access is for writing. The analyser does not perform extrapolation and, as such, does not need to monitor any synchronisation. The error is detected only if it really occurs. The implementation in ANaConDA framework uses before-event and after-event callbacks and a map which is shared among all threads and holds information about currently accessed addresses. Pseudocode of the algorithm is shown in Example 5.3. In the callback `beforeAccess`, a thread checks whether the same address was not already accessed by a different thread. If not, the thread stores information about its own access in the shared map. If the map already contains a record of access and at least one access is for writing, data race is detected. Callback `afterAccess` signalizes that the thread is not accessing the

address anymore, thus a corresponding record about the access is removed from the shared map.

Using the extension for multi-process monitoring which is now provided by ANaConDA framework, modification of the analyser is quite simple. For multi-threaded analysis, `Map` is in fact a global variable of type `std::map`. For multi-process analysis, a structure `SharedMap` provided by shared API is used. The second change in the implementation is that `address a` provided to the callback is a virtual address. Implementation for multi-threaded programs uses it directly. Implementation for processes needs to call function `UTILS_IsShared()` and if the address is shared, the rest of the code in the callbacks needs to use the translated address.

Proc-AtomRace was implemented and used for experiments described in Section 6.2. The actual implementation is more complicated because the record stored in the shared map is a dynamic user-defined structure. This is caused by storing debugging information about the accessed variable. The problem was solved using the approach described in Section 5.2.2.

5.4 Monitoring Semaphores in Analysers

Figure 5.5 shows structure `Semaphore` which represents a semaphore used in monitored program. This structure is provided to analyser's callbacks and describes semaphore's current state. Instances of this structure are stored in a shared memory. Attributes describing semaphore include:

- `type` – The type of semaphore. Possible values are POSIX named, POSIX unnamed or System V semaphore.
- `id` – Semaphore's identifier. The structure which describes it contains various attributes and not all of them are used for all types of semaphores:
  - `valid` – A flag determining whether the semaphore is known. Invalid semaphore represents unknown one as described in Section 4.4.2. Analyser's callbacks can easily check semaphore's validity using method `isValid()`.
  - `sem_id` – A unique identifier of the semaphore. For POSIX unnamed semaphores, it is either virtual address or translated address. The translated address is used when the semaphore is stored in a shared memory. The address' type is distinguished by attribute `virt`. For POSIX named semaphore, it is hash of the semaphore's name. As string is a dynamic structure whose size is not known in the compile-time, the string itself can be accessed using a shared variable with the name given by `sem_id`. If the semaphore represents one semaphore of a System V semaphore set, the `sem_id` is hash of `setid` and `semnum`.

    Attribute `sem_id` is used for all types od semaphores to allow their uniform usage. If the particular identifier used in monitored program is needed, it can be obtained using corresponding attributes.
- `current` – The current state of the semaphore which changes between before-event and after-event callbacks. The state is given by semaphore's value and a count of waiting processes. Attribute `init` is used for System V semaphores to determine whether `semop()` function with command `SETVAL` initializes semaphore or behaves as up or down operation.

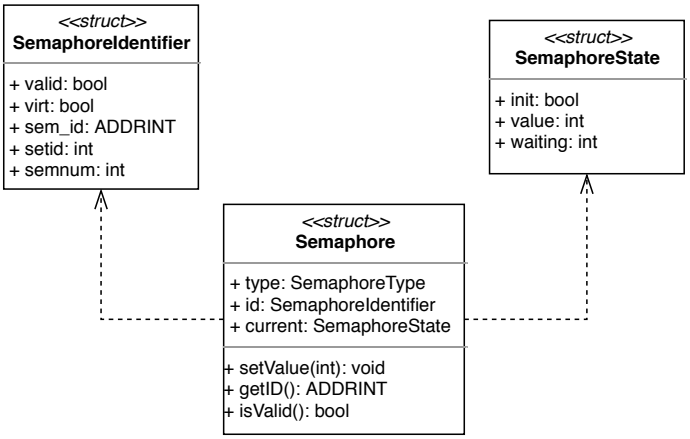


Figure 5.5: Structures representing a semaphore.

5.4.1 Implementation of Happens-Before Relation

As mentioned, vector clocks for happens-before relation monitoring are currently implemented in analysers which use them (e.g. analysers for contract validation or FastTrack). All of these implementations are currently thread-oriented and do not consider semaphores. To preserve this approach, a solution for happens-before relation for semaphores was not implemented directly to ANaConDA framework but a new analyser Semaphore-Analyser was implemented instead. This analyser does not detect any concurrency-related errors. It is used for automatic tests for monitoring of semaphores and happens-before relation and serves as an example of its implementation. The happens-before relation is implemented using vector clocks which are updated when a fork-join synchronisation (a new process is created or a process finishes its execution) or synchronisation using semaphores occurs.

Vector Clocks for Processes

The structure for vector clock's representation was modified from Jan Fiedor's implementation. A vector clock for threads is a wrapper around `std::vector` with operations `init()`, `increment()`, `join()` and with a method `hb()` which for a given thread and action represented by a vector clock decides whether the action happened-before the action represented by `this` vector clock. Vector clocks are stored in global variables as threads need to access vector clocks of different threads or locks. For processes, vector clocks will need to be stored in a shared memory. Thus, they were modified using the approach described in Section 5.2.2. The difference between vector clocks for threads and processes is shown in Example 5.4.

One problem with the implementation of vector clocks for threads and processes is their unique identifier. Thread's identifier assigned by PIN cannot be used (see Section 3.2). Thus, when a new thread is created, it is assigned a unique identifier by an analyser. The uniqueness of pid assigned by operation system cannot be assured either, but the main problem with pids is that higher numbers would cause large vectors with unnecessary elements for non-existing processes. For this reason, processes are assigned unique identifier starting from number 0.

Fork-Join Synchronisation

The fork is handled in two callbacks separately. At first, the child process initializes its vector clock and updates it according to rules described in Section 2.3.1. Parent process only increments its logical time. It is important to synchronise these two callbacks so that the child process updates its vector clock **before** parent process increments its logical time.

When a process finishes its execution, corresponding vector clocks are updated according to rules described in Section 2.3.1.

```
struct VectorClock_threads {
  std::vector<int> vc;

  void increment(Thread tid) {
    ++vc[tid];
  }
  ...
}

typedef SharedVector
  <int, SI_SYNC, WI_WRITE_THROUGH, BI_SHARED_MEMORY> Container;

struct VectorClock_processes {
  std::size_t container_name;

  void increment(Process pid) {
    Container vc(std::to_string(container_name));

    ++vc[pid];
  }
  ...
}
```

Listing 5.4: A difference between the implementation of a vector clock for threads and processes.

Synchronisation with Semaphores

The happens-before relation is formed in after-initialization, before-down, after-down and after-up callbacks. Callbacks implement Algorithms 2, 3 and 4. The before-down callback is necessary to decide whether the process will be suspended and if so a waiting queue of the semaphore is updated.

The three algorithms use a queue of up operations over semaphore s and a waiting queue of semaphore s. Neither of these is provided by ANaConDA and they are implemented in the analyser itself. The queue of waiting processes is, in fact, a shared vector of pids. The queue of up operations is shared deque where each element is a triplet `(pid, vector clock, name of the shared waiting queue)`. The triplet is of fixed size thus can be used as a data type for the deque. Each semaphore has associated up queue whose name is derived from semaphore's identifier. The only complication in implementation is that a record about up operation (the triplet) needs to store a state of the waiting queue at the

moment when the up operation was executed. Thus, the name of the shared waiting queue needs to be unique for each up operation.

5.4.2 FastTrack for Processes

Semaphore-Analyser monitors happens-before relation but does not detect any errors. To utilize this implementation for data race detection, new analyser Proc-FastTrack implementing FastTrack algorithm for processes was created. The implementation is based on pseudocode described in [9].

FastTrack Algorithm

In contrast to AtomRace, FastTrack does not detect data races which actually occurred but monitors synchronisation and decides whether accesses to a shared variable are synchronised or not. The original algorithm for multi-threaded programs maintains a vector clock for each thread and lock and updates it when synchronisation occurs according to rules described in Section 2.3.1.

The detection of unsynchronised accesses to a shared variable is based on a principle designed for algorithm DJIT+ [26]. The DJIT+ algorithm maintains two vector clocks (R_x, W_x) for each variable x. Value $R_x(t)$ for any thread t represents the logical time of the last read from variable x by thread t. Similarly, value $W_X(t)$ for any thread t represents the logical time of last write to variable x by thread t. A read access by thread u with current vector clock VC_u is synchronised if the last write of each thread happened-before the current read, i.e. $W_x \sqsubseteq VC_u$. A write access by thread u with current vector clock VC_u is synchronised if the last access (read or write) of each thread happened-before the current write, i.e. $W_x \sqsubseteq VC_u$ and $R_x \sqsubseteq VC_u$.

FastTrack algorithm enhances the performance of DJIT+ by replacing vector clocks R_x and W_x by so-called epochs. An epoch, denoted $c@t$, is a pair where t is a thread which last accessed the shared variable and c is the time of this last access. This simplification is based on the fact that if data race has not been detected yet, all writes and reads were synchronised and ordered by happens-before relation. Thus, it is sufficient to keep information only about the last read and write. The only exception is that reads do not need to be synchronised. A situation when multiple unsynchronised threads read from a variable is called shared reading and, in such a case, the whole vector clock is necessary. FastTrack algorithm distinguishes these situations.

Modification for Processes

Implementation of FastTrack algorithm for processes differs from a multi-threaded version in three ways. Firstly, it monitors semaphores and fork-join events on processes. The vector clocks are updated according to algorithms described in Section 4.3.2. Secondly, translated addresses of variables are used as their unique identifier, and variables which are not shared are not monitored at all. Lastly, shared API is used for interprocess communication. Thus, analyser Proc-FastTrack uses all parts of the extension for multi-process monitoring in ANaConDA framework.

Chapter 6

Evaluation

To verify the functionality of implemented extension, a set of automatic test cases is provided. To verify that proposed design can be used for concurrency-related errors detection in multi-process programs, analysers Proc-AtomRace and Proc-FastTrack were used for experiments on students' projects. Both tests and experiments are described in this chapter.

6.1 Tests

ANaConDA framework already provides a set of automatic tests for various features and analysers. It was extended with over 40 test cases for process monitoring. All tests can be found in directory `tests/framework/monitoring`. Each test in ANaConDA framework needs to be executed with a particular analyser. For this reason, analysers which do not detect any errors but only print information about detected events were implemented.

Test for shared API Test `shared_api` uses Proc-Analyser which uses all shared structures and operations provided by shared API. The analyser was implemented directly for this test and assumes that the monitored program has two processes. Once the child process is forked, both processes in analyser use shared API to communicate. At first, parent process initializes shared structures and writes to them. Then, child process prints current values in shared structures and rewrites them. Lastly, parent process prints current values in shared structures. The test compares printed values with expected result and, as such, verifies that all operations over shared structures work and that both independent processes can see changes in these structures. The analyser also serves as an example of usage of shared API.

Tests for address translation Monitoring of shared memory and translation of virtual addresses is tested using analyser Shared-Memory-Monitor. Again, this analyser does not detect any errors but monitors accesses to memory and if the address is shared, prints its translated representation. Test cases verify whether accesses to the same memory segments result into the same translated address. Tests are divided into two categories according to the type of a shared memory. Table 6.1 shows test cases for POSIX shared memory and covered features. These include:

- Private – The test creates a memory mapping with flag `MAP_PRIVATE`. Such memory is not shared and accesses to it should not be detected by Shared-Memory-Monitor.

- Anonymous – Shared memory was mapped with flag `MAP_ANONYMOUS`. If the feature is not covered in a test case, then a file descriptor was provided to `mmap()`.
- Inherited pointer – The mapping was created before children processes were forked and all processes access the shared memory using the same virtual addresses. If the feature is not covered in a test case, a mapping is created in each process separately.
- Offset – An offset other than zero was provided to `mmap()`.
- Unmap – Part of shared memory is unmapped using `munmap()` or `mremap()`. If a whole range of addresses is unmapped, a process creates the same mapping again and accesses it again. The value *Beginning* means that a process unmaps a range of addresses from the beginning of the whole range and accesses other addresses. Similarly, the value *End* means that a range of addresses at the end of the whole range is unmapped and the value *Middle* means that the test case unmaps a page in the middle of the address range.

Table 6.2 shows test cases for System V shared memory. As System V does not provide offset and unmap equivalent, all features can be covered by fewer tests:

- `IPC_PRIVATE` – `shmget()` function is not provided with a key but with a flag `IPC_PRIVATE` instead. Multiple shared memories are created. One of them is accessed by all processes using the same virtual address. Other shared memories are created after fork and accessed by one process only (hence Yes/No in Inherited pointer column). The point of this test case is to verify that different shared memories created with `IPC_PRIVATE` flag are provided with different translated addresses.
- Inherited pointer – The same meaning as for POSIX shared memory.

Table 6.1: Overview of test cases for address translation using POSIX shared memory and features that are covered by them.

Test case	Private	Anonymous	Inh. pointer	Offset	Unmap
`shared_memory_posix1`	Yes	Yes	Yes	No	No
`shared_memory_posix2`	No	No	Yes	No	No
`shared_memory_posix3`	No	No	No	Yes	No
`shared_memory_posix4`	No	No	No	No	Whole
`shared_memory_posix5`	No	No	No	No	Beginning
`shared_memory_posix6`	No	No	No	No	End
`shared_memory_posix7`	No	No	No	No	Middle
`shared_memory_posix8`	No	No	No	Yes	Middle
`shared_memory_posix9`	No	No	No	Yes	`mremap()`

Tests for semaphores monitoring Tests for semaphores monitoring verify whether the information provided to analyser's callbacks correctly represent semaphore's state in the monitored program. Analyser Semaphore-Analyser which is used for the construction of happens-before relation also prints information about synchronisation events on semaphores and, as such, may be used for these tests as well. Again, test cases are divided into three

Table 6.2: Overview of test cases for address translation using System V shared memory and features that are covered by them.

Test case	IPC_PRIVATE	Inherited pointer
shared_memory_system_v1	Yes	Yes/No
shared_memory_system_v2	No	Yes
shared_memory_system_v3	No	No

categories according to the type of semaphore, i.e. POSIX named (Table 6.3) and unnamed (Table 6.4) semaphores and System V semaphores (Table 6.5). Features include:

- Threads – The test case is multi-threaded and a semaphore is used by multiple threads.
- Processes – The test case is multi-processed and a semaphore is used by multiple processes.
- Suspended – A thread/process performs down operation and is suspended.
- Shared – Parameter `pshared` given to `sem_init()`. If Shared is Yes, the semaphore is shared among processes. Otherwise, it is shared among threads only.
- Set – A number of semaphores in the set.
- Operations (Op.) – A function used to perform operations over semaphores.
- Atomic – Determines whether `semop()` function is used to perform multiple operations at once.
- Change – A value added to or subtracted from semaphore's value.

All test cases perform up and down operations.

The problem with tests for semaphores monitoring and happens-before relation is that they need to be deterministic. This means that operations need to be performed in a certain order. Determinism is ensured by different synchronisation. Tests for semaphores monitoring use fork-join synchronisation and `sleep()` where different synchronisation is not possible (e.g. when a process needs to perform down operation and be suspended first, as a suspended process cannot release a lock or end its execution).

Table 6.3: Overview of test cases for monitoring of POSIX named semaphores and features that are covered by them.

Test case	Threads	Processes	Suspended
semaphore_posix_named1	No	No	No
semaphore_posix_named2	Yes	No	No
semaphore_posix_named3	No	Yes	No
semaphore_posix_named4	No	Yes	Yes

Table 6.4: Overview of test cases for monitoring of POSIX unnamed semaphores and features that are covered by them.

Test case	Shared	Threads	Processes	Suspended
`semaphore_posix_unnamed1`	No	No	No	No
`semaphore_posix_unnamed2`	No	Yes	No	No
`semaphore_posix_unnamed3`	Yes	No	No	No
`semaphore_posix_unnamed4`	Yes	Yes	No	No
`semaphore_posix_unnamed5`	Yes	No	Yes	No
`semaphore_posix_unnamed6`	Yes	No	Yes	Yes

Table 6.5: Overview of test cases for monitoring of System V semaphores and features that are covered by them.

Test case	Set	Op.	Atomic	Change	Processes	Susp.
`semaphore_system_v1`	1	`semop()`	No	1	No	No
`semaphore_system_v2`	2	`semop()`	No	1	No	No
`semaphore_system_v3`	2	`semctl()`	No	1	No	No
`semaphore_system_v4`	1	`semop()`	No	2	No	No
`semaphore_system_v5`	2	`semop()`	Yes	1	No	No
`semaphore_system_v6`	2	`semop()`	Yes	1	Yes	Yes

Tests for happens-before relation Creation of happens-before relation implemented in Semaphore-Analyser is tested on four scenarios which were previously used to explain a proposed design. Namely, situations in Figures 4.10, 4.11. 4.12 and 4.13. Each situation is implemented using all three types of semaphores, thus there are 12 test cases in total. Determinism is ensured by `sleep()` and locks, however, there is no way how to make the situation in Figure 4.11 truly deterministic. The outcome depends on the order in which the processes are released. There are two possible results and the test checks whether the output of analyser is one of them.

6.2 Experiments

The extension for monitoring of processes was used for the implementation of two analysers which can detect data races. Proc-AtomRace does not produce false-positives but does not monitor synchronisation. Proc-FastTrack monitors synchronisation. If the happens-before relation correctly represents all synchronisation used in the monitored program, the analyser should not produce false-positives either.

A web page for ANaConDA framework[1] provides a simple bank example containing data race. Each thread of the program maintains its own account and increments it in a loop by a random sum. There is also a shared variable which represents a sum of all bank accounts. When a thread increments its own account, it also increments this shared variable by the same number. At the end of the program, the shared variable and an actual sum of all accounts are compared. If the sums do not match, a data race occurred. The access

[1] http://www.fit.vutbr.cz/research/groups/verifit/tools/anaconda/

to the shared variable is either synchronised or unsynchronised depending on a parameter. As such, it is a good example for analysers' verification.

The original Jan Fiedor's implementation was modified to use processes and semaphores for experiments with multi-process analysers. Proc-AtomRace correctly analysed the example. If the implementation uses synchronised access, no data race was detected. If the implementation uses unsynchronised access and a data race occured, it was correctly detected by analyser.

There is a problem with Proc-FastTrack as ANaConDA currently does not provide atomic callbacks for synchronisation operations as described in Section 4.4.1. For now, they are monitored as common functions and the triplet *[before-event callback – event – after-event callback]* is not atomic. The implementation where callbacks are inserted into semaphore's critical section is currently worked on by Jan Fiedor. It will be supported in ANaConDA framework but not in time for this thesis as the implementation is complicated. The beginning of critical section is not clearly defined and is strongly dependable on the architecture. When the triplet is not atomic, the semaphore's state provided to analysers does not necessarily represent the semaphore's real state in the monitored program. That is because the operation might be performed over a semaphore with different value, thus result into different outcome. The problem is caused by multiple processes operating the same semaphore at once. To be able to perform experiments with Proc-FastTrack, the implementation of semaphores monitoring was modified to insert noise in `beforeDown()` callback. This increases chances that the process is the only one operating the semaphore. This modification worked and I was able to correctly analyse bank example without any false-positives. Results of experiments were the same as for Proc-AtomRace. When the example did not use synchronisation, data race was correctly detected. When the example used synchronisation, no data race was detected and Proc-FastTrack did not produce any false-positives. This would not be possible if the representation of synchronisation using happens-before relation would not be correct with regard to synchronisation used in the monitored program.

6.2.1 Experiments with Students' Projects

As mentioned at the beginning of this thesis, the extension for multi-process monitoring can be helpful for students of Operating Systems course. To prove this hypothesis, the two analysers were used on a set of 19 anonymized students' projects which received maximal possible rating as common tests did not detect any error. Students use various combinations of types of semaphores and shared memory. All projects were successfully analysed by both analysers and a data race was detected in two of them by both analysers[2]. Examples 6.1 and 6.2 show excerpts of projects which contain detected data races. Both of these concurrency-related errors are caused by a different fault. In Example 6.1, the student means to check return value of function `fork()` but uses wrong variable `child_pid[i]` instead of `tmp_pid`. As this variable is shared and accessed without synchronisation, data race was correctly detected. Similar fault is in Example 6.2, where student means to check return value of function `shmat()` but accesses the shared memory instead. Again, the access is unsynchronised and the data race was correctly detected.

[2]Proc-FastTrack produced false-positives on one project due to non-atomic triplet *[before-event callback – event – after-event callback]*.

```
int tmp_pid = fork();
if (child_pid[i] < 0) {
    sem_post(sem.kill);
    print_error("Child fork error\n");
    break;
}
```

Listing 6.1: A snippet of student's project containing data race.

```
shmA = (int*)shmat (shmID, NULL, 0);
if (*shmA == -1)
    return -1;
```

Listing 6.2: A snippet of student's project containing data race.

Furthermore, two of these 19 projects do not correctly finish all children processes before the parent process ends its execution. For this reason, simple analysers which monitor correct usage of synchronisation primitives and processes' termination could be also useful.

As we plan to provide ANaConDA framework with extension for monitoring of processes and analysers for data race detection in multi-process programs to students in the future, similar errors could be avoided[3]. Analysers take advantage of ANaConDA's ability to extract debugging information and, as such, are able to provide useful information about detected errors, e.g.:

```
Data race on memory address 0x7f36815de004 detected.
Process 11159 read from <unknown>
  accessed at line 145 in file /home/monika/proc-benchmark/all/x8/proj2.c
Process 11159 written to <unknown>
  accessed at line 220 in file /home/monika/proc-benchmark/all/x8/proj2.c
```

Thus, the analysis should be easy to use even for a student who is not familiar with dynamic analysis.

[3] Also, teachers evaluating these projects will easier detect any remaining errors.

Chapter 7

Conclusion

This thesis aimed to extend ANaConDA framework with support for monitoring of processes. This work summarizes how the framework performs multi-threaded analysis and how the differences between processes and threads affect the monitoring and analysers in general. Several problems, such as communication in analysers, virtual addresses and synchronisation with general semaphores have been described. Solutions for all of these problems were designed, implemented and verified on a set of automatic tests. Furthermore, the extension was used for the implementation of two analysers (Proc-AtomRace and Proc-FastTrack) for data race detection which were, until now, available for multi-threaded programs only. The implementation of Proc-FastTrack was possible thanks to the definition of happens-before relation for general semaphores which was also proposed as a part of this thesis. These analysers were used for experiments on real programs which were not implemented with the intention to be analysed. Experiments proved that the definition of happens-before relation correctly reflects synchronisation in monitored program and that the extension for process monitoring works and is able to detect concurrency-related errors in multi-process programs. As such, ANaConDA framework, which was already a valuable helper with the development of multi-threaded programs even in the commercial sphere, can be utilized for a wider range of programs. Results of this work were presented on student conference Excel@FIT [24]. It was awarded a Prize of Jiří Kunovský, an award of expert committee and it was also rewarded by two enterprise partners Honeywell and Red Hat.

As the support for processes demands a quite extensive implementation, not all features have been implemented. Lists of supported and unsupported features are presented in Appendices C and D. The implementation in this thesis was aimed at students' projects from the course of Operating systems. One of the directions for future work would be to implement the rest of the features so that the extension could be used for any type of program.

There are several possible directions for future development. Namely, extending support on programs combining threads and processes, distributed systems and systems using message passing instead of shared memory, or generalizing happens-before relation definition on operations which change semaphore's value by a different count than 1. Also, new analysers for processes need to be implemented or derived from existing multi-threaded implementations.

Bibliography

[1] IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7. *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*. 2018, p. 1–3951.

[2] Agarwal, R. and Stoller, S. D. Run-Time Detection of Potential Deadlocks for Programs with Locks, Semaphores, and Condition Variables. In: *Proceedings of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging*. New York, NY, USA: Association for Computing Machinery, 2006, p. 51–60. PADTAD '06. DOI: 10.1145/1147403.1147413. ISBN 1595934146.

[3] Desnoyers, M., McKenney, P. E., Stern, A. S., Dagenais, M. R. and Walpole, J. User-Level Implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems*. IEEE Press. 2012, vol. 23, no. 2, p. 375–382. DOI: 10.1109/TPDS.2011.159. ISSN 1045-9219.

[4] Dias, R. J., Ferreira, C., Fiedor, J., Lourenço, J. M., Smrčka, A. et al. Verifying Concurrent Programs Using Contracts. In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Institute of Electrical and Electronics Engineers, 2017, p. 196–206. DOI: 10.1109/ICST.2017.25. ISBN 978-1-5090-6031-3.

[5] Edelstein, O., Farchi, E., Goldin, E., Nir, Y., Ratsaby, G. et al. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience*. Chichester, UK: John Wiley & Sons, Ltd. 2003, vol. 15, 3-5, p. 485–499. DOI: 10.1002/cpe.654. ISSN 1532-0626.

[6] Fiedor, J., Mužikovská, M., Smrčka, A., Vašíček, O. and Vojnar, T. Advances in the ANaConDA Framework for Dynamic Analysis and Testing of Concurrent C/C++ Programs. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2018, p. 356–359. ISSTA 2018. DOI: 10.1145/3213846.3229505. ISBN 9781450356992.

[7] Fiedor, J. and Vojnar, T. ANaConDA: A Framework for Analysing Multi-threaded C/C++ Programs on the Binary Level. In: *Proc. of 3rd International Conference on Runtime Verification—RV'12*. Berlin, Heidelberg: Springer, 2012, 7687 of Lecture Notes in Computer Science, p. 35–41. DOI: 10.1007/978-3-642-35632-2_5. ISBN 9783642356315.

[8] Fiedor, J. and Vojnar, T. Noise-based Testing and Analysis of Multi-threaded C/C++ Programs on the Binary Level. In: *Proceedings of the 2012 Workshop on*

Parallel and Distributed Systems: Testing, Analysis, and Debugging. New York, NY, USA: Association for Computing Machinery, 2012, p. 36–46. PADTAD 2012. DOI: 10.1145/2338967.2336813. ISBN 9781450314565.

[9] Flanagan, C. and Freund, S. FastTrack: efficient and precise dynamic race detection. *Communications of the ACM*. New York, NY, USA: Association for Computing Machinery. 2010, vol. 53, no. 11, p. 93–101. DOI: 10.1145/1839676.1839699. ISSN 0001-0782.

[10] Flanagan, C. and Freund, S. N. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In: *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. New York, NY, USA: Association for Computing Machinery, 2010, p. 1–8. PASTE '10. DOI: 10.1145/1806672.1806674. ISBN 9781450300827.

[11] Gaztanaga, I. *Boost.Interprocess – Boost 1.58.0 Library Documentation*, april 2015. Available at: https://www.boost.org/doc/libs/1_58_0/doc/html/interprocess.html.

[12] Guo, P. J. *A Scalable Mixed-Level Approach to Dynamic Analysis of C and C++ Programs*. Cambridge, USA, 2006. Master's thesis. MIT.

[13] Havelund, K. Using Runtime Analysis to Guide Model Checking of Java Programs. In: *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*. Berlin, Heidelberg: Springer, 2000, p. 245–264. ISBN 3540410309.

[14] Intel Corporation. *Pin 2.14 User Guide*, may 2018. Available at: https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html/index.html.

[15] Kerrisk, M. *Shm_overview(7) Linux Programmer's Manual*, 5.06th ed., december 2016.

[16] Kerrisk, M. *Sem_overview(7) Linux Programmer's Manual*, 5.06th ed., may 2017.

[17] Kerrisk, M. *Sysvipc(7) Linux Programmer's Manual*, 5.06th ed., april 2020.

[18] Kerrisk, M. and Eckhardt, D. *Fork(2) Linux Programmer's Manual*, 5.06th ed., september 2017.

[19] Kerrisk, M., Quinlan, D., Cox, A., Neuffer, M. and Brouwer, A. *Proc(5) Linux Programmer's Manual*, 5.06th ed., april 2020.

[20] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery. 1978, vol. 21, no. 7, p. 558–565. DOI: 10.1145/359545.359563. ISSN 0001-0782.

[21] Letko, Z., Vojnar, T. and Křena, B. AtomRace: Data Race and Atomicity Violation Detector and Healer. In: *Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*. New York, NY, USA: Association for Computing Machinery, 2008, p. 7:1–7:10. PADTAD '08. DOI: 10.1145/1390841.1390848. ISBN 9781605580524.

[22] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A. et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *ACM SIGPLAN Notices*. New York, NY, USA: Association for Computing Machinery. june 2005, vol. 40, no. 6, p. 190–200. DOI: 10.1145/1064978.1065034. ISSN 0362-1340.

[23] Mužikovská, M. *Rozšíření frameworku ANaConDA pro podporu kontraktů s parametry a jejich omezeními*. Brno, CZ, 2018. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology.

[24] Mužikovská, M. *Detekce paralelních chyb ve víceprocesových programech: Excel@FIT, Student conference*. 2020.

[25] Nethercote, N. and Seward, J. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2007, vol. 42, no. 6, p. 89–100. PLDI '07. DOI: 10.1145/1250734.1250746. ISBN 9781595936332.

[26] Pozniansky, E. and Schuster, A. Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs. In: *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: Association for Computing Machinery, 2003, p. 179–190. PPoPP '03. DOI: 10.1145/781498.781529. ISBN 1581135882.

[27] Satran, M., Batchelor, D. and Schofield, M. *Semaphore Objects – Microsoft Docs*, may 2018. Available at: https://docs.microsoft.com/en-us/windows/win32/sync/semaphore-objects.

[28] Satran, M., Jacobs, M. and Schofield, M. *About Handles and Objects – Microsoft Docs*, may 2018. Available at: https://docs.microsoft.com/en-us/windows/win32/sysinfo/about-handles-and-objects.

[29] Satran, M., Jacobs, M., Schofield, M. and Batchelor, D. *Handle Inheritance – Microsoft Docs*, may 2018. Available at: https://docs.microsoft.com/en-us/windows/win32/sysinfo/handle-inheritance.

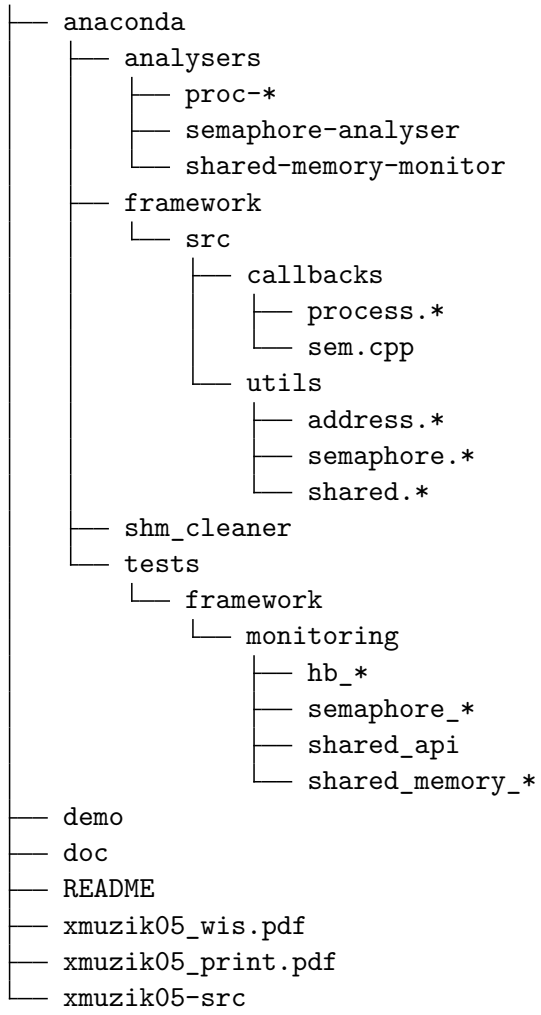
[30] Satran, M., Schofield, M. and Batchelor, D. *Sharing Files and Memory – Microsoft Docs*, may 2018. Available at: https://docs.microsoft.com/en-us/windows/win32/memory/sharing-files-and-memory.

[31] Satran, M., Schofield, M. and Batchelor, D. *Wait Functions – Microsoft Docs*, may 2018. Available at: https://docs.microsoft.com/en-us/windows/win32/sync/wait-functions.

[32] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P. and Anderson, T. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. In: *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*. New York, NY, USA: Association for Computing Machinery, 1997, p. 27–37. SOSP '97. DOI: 10.1145/268998.266641. ISBN 0897919165.

Appendix A

Storage medium



`anaconda` contains source codes for ANaConDA framework, the extension for processes, new analysers for processes, test cases and `shm_cleaner`. The list above shows source codes implemented for this thesis. `demo` contains source codes of bank example and scripts for its execution. `doc` is HTML documentation generated by Doxygen. `xmuzik05_*.pdf` is the text of this thesis and `xmuzik05-src` contains its source codes in LaTeX.

Appendix B

Details of Implementation of Shared API

Figure B.1 shows class diagram of class `Sync`. This templated class provides methods `lock()` and `unlock()` which are used in shared data types to provide atomic or non-synchronised operations. The particular implementation for each value of the parameter SI is specified in template specialisations of methods.

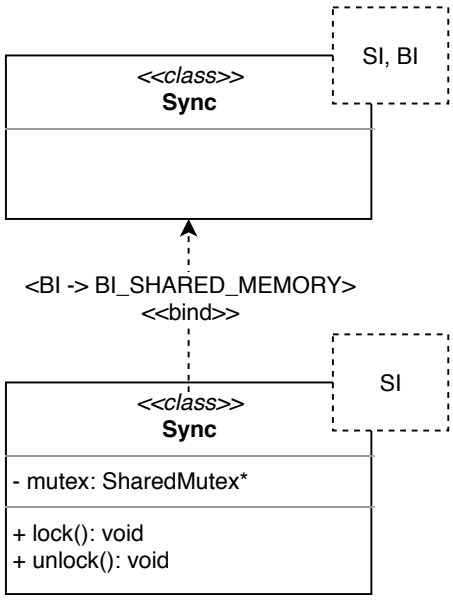


Figure B.1: Diagram of class `Sync` implementing methods according to parameter SI. The synchronisation is dependent on the backend implementation as well.

Figure B.2 shows class diagram of class `Write`. Currently, only write through is supported and corresponding methods are defined in partially specialised classes. The variety of methods depends on the type of shared variable. For fix sized data types, only methods for setting value and getting value are implemented. The figure is simplified, the class is partially specialised for each type of supported container and provides methods for all supported operations over the shared data type. In case of write through, all methods call a corresponding method from `Backend` class and return the result. Also, a templated

data type for a container is used as it depends on parameter BI and the class `Write` is independent of the backend implementation.

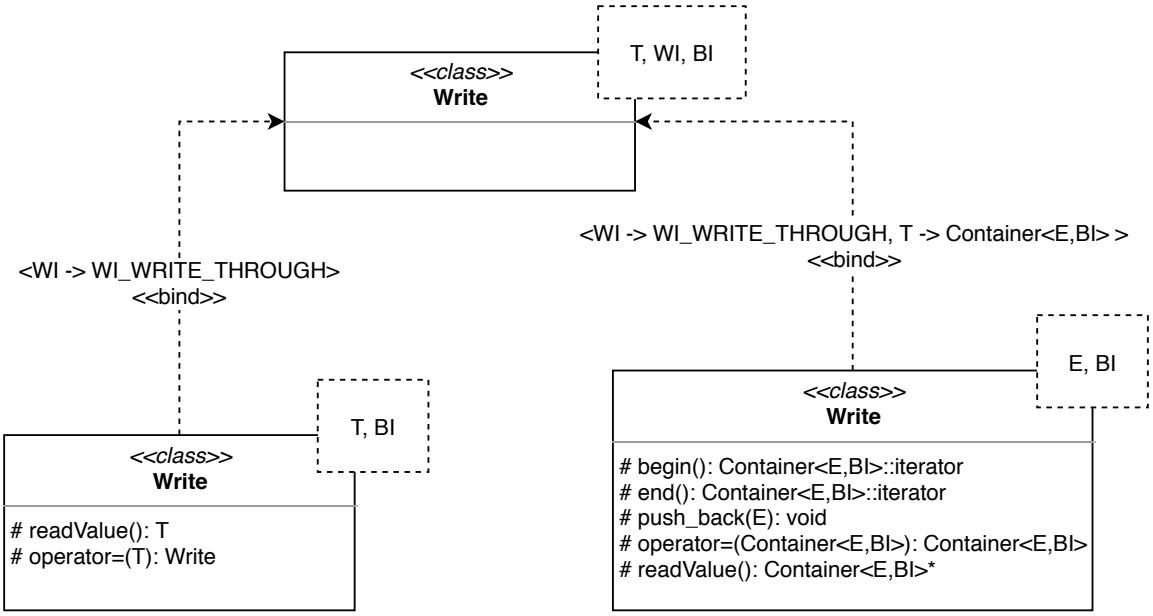


Figure B.2: Diagram of class `Write` implementing methods according to parameter WI.

The majority of implementation is in the class `Backend` which is shown in Figure B.3. Again, the implementation differs according to data type of shared variable. For data types of fixed size, a pointer to variable in shared memory is stored. The variable is created in banks as described in Section 4.1.3. Specialisations for containers are more complicated. Firstly, the data type of container is not templated and the class uses containers suitable for shared memory. Again, the figure is simplified and the class is partially specialised for each type of supported containers. The class holds following information:

- `shmem_container` – Shared memory with the container.
- `shmem_size` – Last seen size of the shared memory with the container.
- `realloc_manager` – The guard for accessing shared memory with the container as described in Section 4.1.2.
- `container` – A pointer to the container in shared memory.

To simplify implementation of operations over shared variables, friend functions are implemented and used. These include:

- `containerConstructor()` – Opens or constructs shared container and finds or constructs corresponding `ReallocationManager` (the guard).
- `containerUpdate()` – Handles possible reallocation by comparing local information about the size of shared memory (attribute `shmem_size`) and the actual size stored in `ReallocationManager`. If they differ, reopens shared memory with container and remaps it into process' address space.
- `containerGrow()` – Enlarges shared memory with the container.

- `containerOperation()` and `containerOperationVoid()` – These function encapsulate a given operation over shared container into steps which are necessary to handle possible reallocation and inssuficient space in shared memory. Firstly, a readlock from `ReallocationManager` is acquired which ensures that no process is currently enlarging shared memory with the container. Then, `containerUpdate()` is called to handle possible reallocation. Finally, the operation is performed. If it fails due to insufficient space, a writelock is acquired, `containerGrow()` used to enlarge shared memory and the whole cycle is repeated.

Using these friend functions, the implementation of operations over shared containers is very simple:

```
ShmemString::iterator begin()
{
  ShmemString::iterator(ShmemString::* op)() = &ShmemString::begin;
  return containerOperation<ShmemString, iterator> (this, op);
}
```

Thus, they can be easily extended with new operations which are currently not supported and the developer does not need to keep in mind problems with reallocation and their solution.

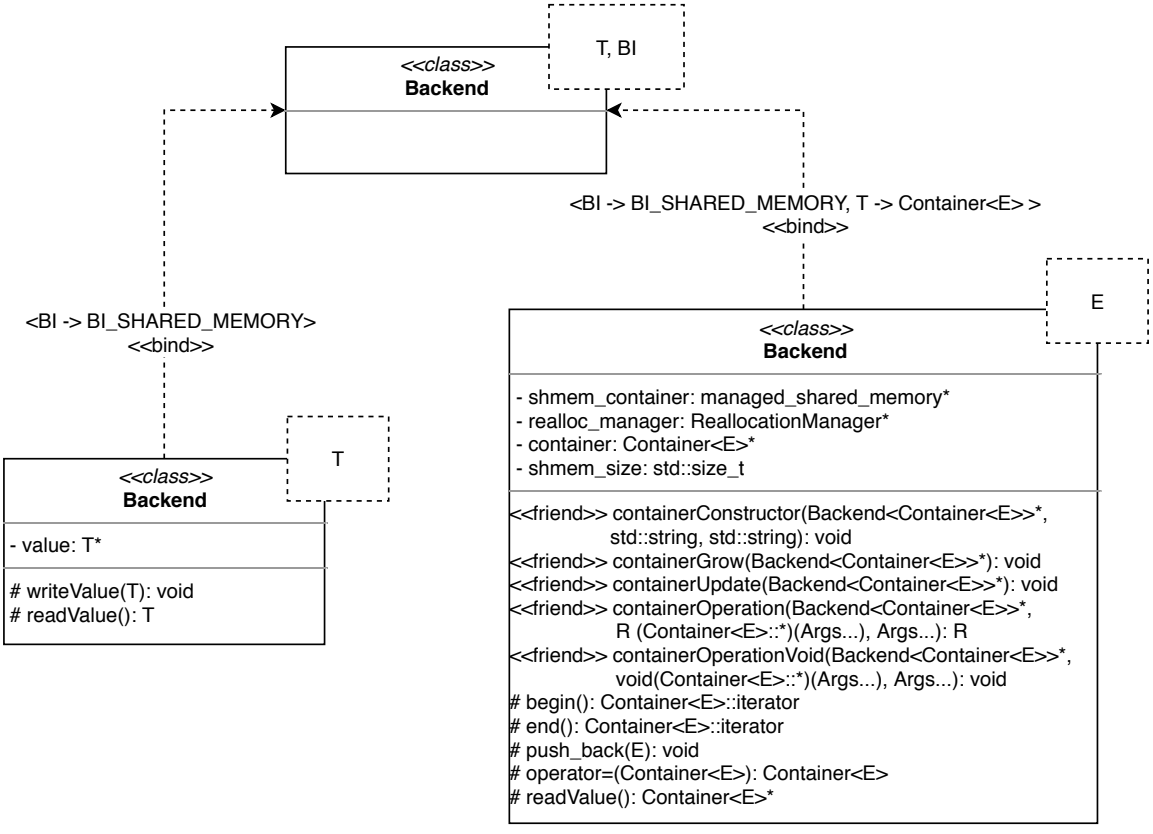


Figure B.3: Diagram of class `Backend` implementing methods according to parameter BI.

Appendix C

Supported Features

Callbacks:

- `SEM_BeforeInit(pid_t pid, THREADID tid, Semaphore* sem, int val)`
- `SEM_AfterInit(pid_t pid, THREADID tid, Semaphore* sem, int val)`
- `SEM_BeforeDown(pid_t pid, THREADID tid, Semaphore* sem, int val, int waiting_count, int change)`
- `SEM_BeforeUp(pid_t pid, THREADID tid, Semaphore* sem, int val, int waiting_count, int change)`
- `SEM_AfterDown(pid_t pid, THREADID tid, Semaphore* sem, int val, int waiting_count, int change)`
- `SEM_AfterUp(pid_t pid, THREADID tid, Semaphore* sem, int val, int waiting_count, int change)`
- `PROCESS_BeforeFork(pid_t pid, THREADID tid)`
- `PROCESS_AfterForkParent(pid_t pid, THREADID tid)`
- `PROCESS_AfterForkChild(pid_t pid, THREADID tid)`
- `PROCESS_ProcessFinished(pid_t pid, THREADID tid, pid_t epid)`

API for interprocess communication:

- Shared memory backend.
- Write through.
- Implicit and explicit synchronisation.
- Data types of fixed size and STL containers with operations shown in Figure 5.1.

Virtual address translation:

- Shared memory on Linux (POSIX and System V).

Happens-before relation:

- Operation which change semaphore's value by 1.

Semaphore monitoring:

- Semaphores on Linux (POSIX and System V semaphores).
- `semop()` function performing multiple operations at once.

Appendix D

Unsupported Features

Callbacks:

- Existing callbacks are not provided with pid of the process. It can be obtained using function `PIN_GetPid()`.
- Identifier of the lock is not translated.
- Callbacks for system calls. Currently, system calls are monitored as functions which is not ideal. Support for monitoring of system calls directly is currently being worked on.

API for interprocess communication:

- Dynamic user-defined structures.
- Not all STL containers and not all operations are supported.
- Different types of backend implementation.
- Write back.
- ANaConDA framework has a problem with `munmap()` function which is used when shared memory is enlarged. For this reason, the grow method currently does not work. However, this issue is currently worked on. The support for enlargement of shared memory is implemented and once ANaConDA will be able to execute this function properly, it will work.

Virtual address translation:

- Shared memory on Windows.

Happens-before relation:

- Operation which change semaphore's value by a different count than 1.

Semaphore monitoring:

- Callbacks for synchronisation are not in the semaphore's critical section.
- Semaphores on Windows.

- `sem_trywait()`
- Operation *wait for zero* on System V semaphores.
- Flags `IPC_NOWAIT` and `SEM_UNDO` for System V semaphores.
- Generic callback for a failure of synchronisation operation.

Appendix E

Manual for Extending Shared Container with New Operation

Let's consider that `SharedVector` does not provide method `push_back()`. In order to add support for it, an implementation of the operation needs to be provided for corresponding template specialisations of classes `SharedVector`, `Write` and `Backend`:

1. Define operation in the structure `SharedVector`. The operation will be performed by the class `Write` and needs to be encapsulated in synchronisation operations of the class `Sync`:

   ```
   void push_back(T value) {
     Sync<SI>::lock();
     Write<Vector<T, BI>, WI, BI>::push_back(value);
     Sync<SI>::unlock();
   }
   ```

2. Define operation in a template specialisation of class `Write` for write through and data type `Vector<T, BI>`. For parameter write through, the method only forwards operation on `Backend` class:

   ```
   void push_back(T value) {
     Backend<VectorType, BI>::push_back(value);
   }
   ```

3. Define operation in a template specialisation of class `Backend` for shared memory and data type `ShmemVector`. The implementation should use a friend function `containerOperationVoid()` or `containerOperation()`. Both of these functions are templated and need to be provided with a pointer to the operation and its arguments:

   ```
   void push_back(T value) {
     void(ShmemVector<T>::* op)(T&) = &ShmemVector<T>::push_back;
     containerOperationVoid<ShmemVector<T>, T&> (this, op, value);
   }
   ```

Appendix F

Manual for Extending Shared API with New Container

Let's consider that class `SharedDeque` is not provided. In order to add support for new container, the following classes or template specialisations need to be provided:

1. Add a structure which represents container deque but the particular data type is dependent on the type of backend implementation:

   ```
   template < typename T, BackendInfo BI >
   struct Deque {};
   ```

2. Add a template specialisation of this class for all currently supported backend implementations:

   ```
   template < typename T >
   struct Deque<T, BI_SHARED_MEMORY> {typedef ShmemDeque<T> type;};
   ```

   Data type `ShmemDeque` is `deque` provided by `boost` with an allocator suitable for shared memory which is already provided by shared API:

   ```
   template <typename T>
   using ShmemDeque = ip::deque<T, ShmemAllocator<T>>;
   ```

3. Add templated structure `SharedDeque`:

   ```
   template < typename T, SyncInfo SI, WriteInfo WI, BackendInfo BI >
   struct SharedDeque : Sync <SI>, Write <Deque<T, BI>, WI, BI> {
   };
   ```

   A scheme of inheritance in Figure 5.2 needs to be preserved and, as such, the new container class needs to inherit classes `Sync` and `Write`. The first template argument of class `Write` is a data type of shared variable, thus type `Deque<T, BI>` is provided.

4. Class `SharedDeque` needs to be provided with constructor that takes one mandatory and one optional argument. The mandatory argument is name of the shared variable, the optional one is name of communication channel. Names are provided to parent classes. Constructors of parent classes take the same arguments and the following name convenction should be preserved:

```
SharedDeque(std::string variable, std::string ipc = "") :
  Sync <SI> (variable, std::string("SyncDeque") + ipc + variable),
  Write<Deque<T, BI>, WI, BI>
    (variable, std::string("SharedDeque") + ipc){}
```

5. Add template specialisation of class `Write` for container `Deque<T, BI>`:

   ```
   template < typename T, BackendInfo BI >
   class Write<Deque<T, BI>, WI_WRITE_THROUGH, BI> :
     Backend <typename Deque<T, BI>::type, BI> {
   };
   ```

   Again, this class needs to preserve a scheme of inheritance and, as such, inherit class `Backend` with corresponding data type. The constructor initializes parent class:

   ```
   Write(std::string variable, std::string ipc) :
       Backend<typename Deque<T, BI>::type, BI>(variable, ipc)
   {}
   ```

6. Add template specialisation of class `Backend` for shared memory and data type `ShmemDeque`. To make this definition as easy as possible, a helper macro `BACKEND_CONTAINER` was defined and should be used:

   ```
   template < typename T >
   class Backend<ShmemDeque<T>, BI_SHARED_MEMORY> {
     BACKEND_CONTAINER(ShmemDeque<T>,T);
   }.
   ```

   The macro needs to be provided with the container's type and with the type of elements.

Operations over new shared container can be implemented according to manual in Appendix E.