Dynamic Verification of Concurrency in Operating Systems

(Dynamiczna weryfikacja współbieżności w systemach operacyjnych)

Jakub Urbańczyk

Praca licencjacka

Promotorzy: mgr Krystian Bacławski
prof. Witold Charatonik

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

22 czerwca 2021

Abstract

As multicore hardware becomes increasingly powerful and prevalent, many programs heavily utilize advanced multi-threaded techniques to enhance performance. However, concurrent execution can introduce a series of new problems, such as data races and deadlocks. They are notoriously difficult to detect and resolve since they are only triggered under certain instruction interleavings. Many analysis techniques and tools for validating concurrent programs have been proposed, but they often fail to detect concurrency bugs in operating systems (OSes) due to the complex characteristics of OSes. Here I present an overview of selected dynamic detectors of concurrency bugs with an emphasis on those that are applicable in OS kernels. In addition, I detail the implementation within the Mimiker OS of some of the tools that I have created as part of this thesis.

Rosnąca wydajność i popularność maszyn wielordzeniowych sprawia, że coraz więcej programów zaczyna stosować różne techniki i algorytmy równoległe, by lepiej wykorzystać wiele rdzeni. Jednakże, przetważanie współbieżnie wiąże się z szeregiem nowych błędów takich jak wyścigi, czy zakleszczenia. Błędy te są wyjątkowo trudne do znalezienia dlatego, że pojawiają się jedynie przy konkretnym przeplocie instrukcji. Powstało wiele technik i narzędzi znajdujących błędy i weryfikujących poprawność programów współbieżnych, lecz większość z nich nie może być użyta do jąder systemów operacyjnych z powodu ich wyjątkowego stopnia złożoności. Poniższa praca stanowi niewyczerpujący przegląd narzędzi i technik znajdujących błędy w programach współbieżnych w sposób dynamiczny. Szczególny nacisk zostanie położony na te, które znajdują zastosowanie w jądrach systemów operacyjnych. Dodatkowo, w ramach poniższej pracy, niektóre z narzędzi zostały zaimplementowane w systemie operacyjnym Mimiker, co również zostanie dokładnie opisane.

Contents

1 Introduction **7**

2 Concurrency **9**

2.1 Models for Concurrent Programming 9

2.2 Memory Reordering 10

2.2.1 Compile-time Memory Ordering 10

2.2.2 Runtime Memory Ordering 12

2.2.3 Memory Consistency Models 13

2.2.4 Memory Barriers 16

2.3 Problems with Concurrency 17

2.3.1 Race Conditions 17

2.3.2 Atomicity 18

2.3.3 Deadlock 20

2.4 Concurrency in Operating Systems 22

2.5 Generic Ways of Detecting Bugs 24

2.5.1 Lockset Algorithm 24

2.5.2 Happens-before Algorithm 25

2.5.3 Fuzzing 28

3 Tools for Finding Concurrency Bugs **31**

3.1 Thread Sanitizer 31

3.1.1 Compiler Instrumentation 31

3.1.2 Shadow Memory 32

3.1.3 Run-time Library 34
3.1.4 The Kernel Thread Sanitizer 36
3.2 Kernel Concurrency Sanitizer 37
3.3 DataCollider 39
3.4 Lockdep 40

4 Implementation in the Mimiker OS 43
4.1 Kernel Thread Sanitizer 43
4.2 Kernel Concurrency Sanitizer 44
4.2.1 Implementation Details 44
4.2.2 Example Usage and Bug Report 47
4.2.3 Results and Future Improvements 49
4.3 Lock Dependency Correctness Validator 50
4.3.1 Implementation Details 50
4.3.2 Usage and Results 53

5 Conclusions 55
5.1 Contributions 55
5.2 Future Work 56

Bibliography 57

Chapter 1

Introduction

As the number of cores per processor is increasing, concurrent programming is becoming more popular and plays a key role in increasing computing performance. Writing concurrent programs, however, is not as easy as writing sequential ones. It is more complicated and bug-prone. Many operating systems (such as FreeBSD, Linux, and Windows) used to have a single lock, which was held by a thread while the kernel code was running. Nowadays, many of them support multithreading to improve performance. Compared to user-space programs, bugs in the kernel are more likely to hide from the programmer's eye and can have more severe consequences. This is due to a bigger amount of complicated synchronization primitives (dozens of different types of locks), interrupts, and shared hardware resources.

Many analysis techniques have been proposed to detect concurrency bugs. Here I consider only the tools and methods that use dynamic analysis – an analysis performed during executing the program. In this thesis, I also show how I have added two of them into the Mimiker kernel. Mimiker [22] is an open-source operating system developed at the University of Wrocław since 2015 for educational and research purposes. It is Unix-like and inspired mainly by the *BSD world.

The structure of this thesis is as follows: in Chapter 2, I describe concurrency in general. I present different types of concurrency bugs, why they occur, and how they can be detected or even prevented. Then I discuss how the concurrency differs in the kernels of operating systems from regular user-space programs. Finally, I present a few generic methods for detecting bugs.

Then, in Chapter 3, I describe in more detail some of the dynamic concurrency bugs detectors. Finally, in Chapter 4, I present my contribution which is adding two of the tools into the Mimiker, including example usage, implementation details, and the list of already found bugs.

Even though many terms and concepts are explained within the thesis, some basic knowledge of operating systems and computer architecture is recommended. If needed, please refer to [1] or [2].

Chapter 2

Concurrency

In this chapter, a brief introduction to concurrent computing is presented.

2.1 Models for Concurrent Programming

There are two representative approaches to concurrent programming: message passing and shared memory.

In the message-passing model, processes communicate with each other by sending and receiving messages through a communication channel, with no shared mutable state. Messages are generally required to be immutable – data is copied between the local memory of processes. In the synchronous message-passing model, the sender waits for the receiver to confirm communication. Alternatively, communication may be asynchronous. In this case, the sender continues immediately after sending a message which is then queued up for later delivery to the receiver. The message-passing approach can be used to communicate between different processes, cores, or in a cluster of servers.

In the shared-memory approach, multiple processors share access to a global memory space. Thus, the processors can efficiently exchange or share access to data. What is interesting about that paradigm is that the sender does not know who is going to read the data in the future. It can happen at any time, even while another processor is writing to that memory location. Hence, there is no guarantee that the data will not get corrupted. As a consequence, accesses have to be synchronized with each other. This can be done at the hardware (atomic operations, memory barriers) or software (locks, semaphores) level.

The shared-memory programming model has several advantages over the message-passing model. Primarily, it simplifies data exchange, as there is no imposed communication protocol. Therefore, in the context of operating systems shared-memory systems gained wide acceptance for both technical and commercial computing. In

this thesis, the shared-memory model will be used exclusively.

2.2 Memory Reordering

On most modern processors, memory operations are not executed in the order defined by the program. Although computer programs seem to be executed in the specified order, in reality, the machine can change the order of some low-level operations according to its needs. It is mostly due to performance reasons. In single-threaded environments, it is hidden to the programmer and the side effects of the program remain the same, so the behavior of the program is not modified. However, in multithreaded environments, this can lead to many problems.

2.2.1 Compile-time Memory Ordering

The compiler can reorder instructions, but only those that do not depend on each other. One instruction (line of code) is said to *depend* on another instruction if it uses the result from the latter one or if the latter one is a side effect earlier in the program order. The compiler is free to reorder instructions provided that no instructions are executed after any of their dependent ones.

Look at the following code written in the C language:

```
int X, Y;
void test()
{
    X = Y + 1;
    Y = 0;
}
```

High-level expressions define the ordering as follows: the memory store to the variable ‘Y‘ happens right after the store to ‘X‘. We can now compile it into assembly and check if the reordering has occurred. GCC 10.2 without any flags was used in this example.

```
test():
    ...
    mov     eax, DWORD PTR Y[rip]
    add     eax, 1
   ❶mov     DWORD PTR X[rip], eax
   ❷mov     DWORD PTR Y[rip], 0
    ...
```

The store ❶ to the memory associated with the variable ‘X‘ occurs before the store ❷ to ‘Y‘, so the ordering is preserved.

When optimizations are enabled using -O2 flag, the compilation results in the assembly as in the following listing:

```
test():
    ...
    mov     eax, DWORD PTR Y[rip]
   ❸mov     DWORD PTR Y[rip], 0
    add     eax, 1
   ❹mov     DWORD PTR X[rip], eax
    ...
```

This time, the compiler decided to reorder the store to ‘Y‘ ❸ before the store to ‘X‘ ❹. It is allowed to do so because the reordering has not broken any rules. A single-threaded program would work the same, as it would never know the difference.

The compiler reordering can be prevented by using a special C language directive (called also a compiler barrier). Such an instruction forbids the compiler to move memory accesses declared before the barrier behind it and vice-versa. GCC uses `__asm__ __volatile__("":::"memory")` for this purpose. The previous example can be modified to show the effect of this barrier.

```
int X, Y;
void test()
{
    X = Y + 1;
    __asm__ __volatile__("":::"memory");
    Y = 0;
}
```

This results in the assembly (again compiled with the -O2 flag):

```
test():
    ...
    mov     eax, DWORD PTR Y[rip]
    add     eax, 1
   ❺mov     DWORD PTR X[rip], eax
   ❻mov     DWORD PTR Y[rip], 0
    ...
```

As can be seen, the ordering is preserved. Now, '`X`' ❺ is stored before '`Y`' ❻.

2.2.2 Runtime Memory Ordering

Besides the compile-time memory reordering, a CPU can dynamically reorder accesses to memory during runtime. Many modern processors support issuing (and executing) multiple instructions every clock cycle. Even if an assembly instruction is explicitly placed after another, they can end up being issued and executed in parallel. Hardware can reorder instructions in any way it wants as long as the semantics of the code is not changed when compared to the execution in a single-threaded environment. This sort of instruction-scheduling freedom allows for a variety of optimizations to occur.

One example of reordering is via speculative execution. This is when the CPU starts executing a code that has not been reached yet, in the opportunistic chance that the results can be ready when that code is eventually reached. The most interesting case of speculative execution for us is data speculation. It is mostly because the speed of operation of a processor and memory differs by orders of magnitude. To reduce the negative impact of such a bottleneck, a CPU is allowed to schedule data to be loaded from memory earlier than is needed.

Consider the example code shown in Listing 1 which illustrates a producer-consumer interaction between processors.

```
// Thread 1                                  // Thread 2
data = 1                                   ❼if (ready == 1)
ready = 1                                     ❽data_copy = data;
```

Listing 1: A simple producer-consumer scenario. Initially `data = ready = data_copy = 0`.

As shown, the process **P1** stores some value under the variable '`data`' and then synchronizes with **P2** by setting the variable `ready`. After that, **P2** can safely copy the value of the variable. The intended behaviour of this interaction is that every read of **P2** can only happen after the variable `data` is set.

A CPU with speculative execution is free to start ❽ store before reading ❼ the '`flag`'. Clearly, this can lead to a data corruption and hence to a system failure.

Another example of reordering is by CPU caches. CPUs do not read/write directly to shared memory since, as mentioned earlier, it is relatively slow. Instead, each CPU core has its fast-access local memory called cache. Most memory operations are performed on a CPU's cache and eventually flushed to/refreshed from other caches in a process called cache coherency. There are several reasons why memory reordering might occur. For example, consider two cores accessing the same part of the memory. One of them reads from memory, another writes to it. Cache co-

herency protocols might force the reading core to wait until the writer flushes its local cached data back to the main memory so that the reader can read the most up-to-date information. The idling core might choose to run other memory instructions in advance, instead of wasting cycles doing nothing.

2.2.3 Memory Consistency Models

Before we begin our consideration of how to write correct and efficient multithreaded programs, we need a precise definition of the shared memory model that we will use.

Definition (Memory consistency model)**.** *A **memory consistency model** for a shared-memory multiprocessor specifies how memory behaves concerning reading and write operations from multiple processors. In other words, given a potentially concurrent program, a memory consistency model defines the possible values returned for each read in the program as well as the final values of each location memory [4].*

Sequential Consistency

A natural way to define a memory model for systems with multiple processors is to extend the semantics of sequential machines. Such a model can be defined formally. A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each processor appear in this sequence in the order specified by its program [3]. This model can be conceptually represented with multiple processors sharing the main memory. Every processor requests its memory operations in the same order as defined in the program. Operations are then served by the memory controller one at a time. Therefore, they seem to be executed atomically concerning other memory operations. Processors can be handled in any fair order. This results in an arbitrary interleaving of operations from different processors, but the order of operations from each processor remains untouched.

The example code from Listing 1 on sequential consistent multiprocessors is guaranteed to work as intended. Every read of **P2** can only happen after variable `a` is set.

Sequential consistency is the way how most programmers expect a concurrent program to behave. While conceptually intuitive, this model restricts many optimizations (like buffering and pipelining of memory accesses) that modern processors are capable of. So the reality is that to improve the performance (mainly regarding speed), a lot of optimizations are being performed underneath by the processor. Hence, many modern architectures *relax* the consistency model by dropping some of the restrictions and at the same time the guarantee of sequential consistency.

We can distinguish four types of memory operation orderings:

- W $\rightarrow$ R: write must be completed before subsequent read
- R $\rightarrow$ R: read must be completed before subsequent read
- R $\rightarrow$ W: read must be completed before subsequent write
- W $\rightarrow$ W: write must be completed before subsequent write

Sequential consistency maintains all four orderings, while relaxed memory models allow certain orderings to be violated.

Total Store Ordering

Total store ordering (TSO) is one of the strongest models amongst today's modern CPU implementations. It allows reads to move ahead of writes. In other words, the memory ordering W $\rightarrow$ R is abandoned, but only between different threads. A processor's reads and writes remain ordered.

To see how it affects a program execution, consider an example code shown in Listing 2.

```
// Thread 1
A = 1;
print(B);
```

```
// Thread 2
B = 1
print(A);
```

Listing 2: A code showing the TSO memory model. Initially `A = B = 0` and `print` procedure is assumed to be atomic.

On a machine with sequential consistency, both threads will never print 0 at the same time. There is no instruction interleaving that would lead to this scenario. However, in the total store ordering model, there is a possibility that we will see two zeros in the output. The write to variable `A` does not have to be propagated to the second thread immediately. The model does not guarantee that this will happen at all. Similarly with the second thread.

The reason why this model is so important comes from the abundance of processors using the TSO model. The most prominent representative is the x86/64 family of processors [10]. Although actual CPU implementations by Intel or AMD are not publicly available, the simplified model (x86-TSO model) can be illustrated as in Figure 2.1.

As illustrated, the hardware interacts with a subsystem consisting of shared memory, one write buffer per hardware thread, and a global lock indicating who (which CPU) has exclusive access to the memory at that time. When a thread wants to make read access, it firstly searches for a corresponding write buffered in the store buffer. If it was not found, the read is served from the shared memory.

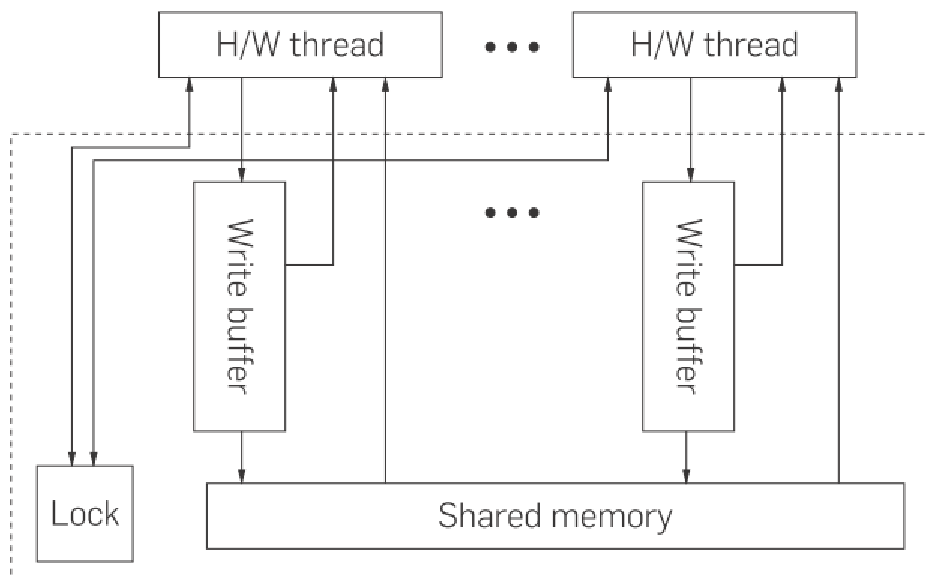

Figure 2.1: x86-TSO block diagram from [10].

Every write is not immediately visible to the other threads. It must be propagated from the buffers to the shared memory. A thread must hold the lock to flush all buffered writes to the shared memory.

Weak Memory Ordering

Weak memory ordering models [12] are significantly more relaxed. They allow the hardware to make a wider range of optimizations. Memory order relaxation has many advantages – it consumes less power, the hardware is less complex, and most importantly, it improves performance. However, as it is usually in computer science, advantages in one part of the system are traded for disadvantages in another. The biggest problem is that implementing concurrent data structures is considerably more difficult.

Models in this class do not have a guarantee that memory operations become visible to other processors in the same order as defined in the program. This allows the memory management unit to execute multiple reads and writes in any order (no matter when they were issued). Therefore, when referring to the previously mentioned four types of memory operation orderings, none of them are respected. This allows both read and write latency to be hidden by executing accesses optimally. In the TSO model, writes could not be executed before earlier reads, so the processor had to wait for a pending read to finish before doing anything else. Moreover, in contrast to the TSO model, the weak memory ordering does not guarantee at what order a write becomes visible to others. Each processor may see the write at a different time.

Several modern CPU architectures have a weak memory ordering model. The most popular is ARM architecture which is currently found in most of all smartphones and multicore embedded devices.

Each hardware thread can be conceptually represented as a single unit with its copy of memory as shown in Figure 2.2. A write operation by one thread may propagate to other threads in any order. The propagation, however, can be interleaved with writes to different addresses. Memory barriers and cache coherence algorithms can restrict this relaxation. These architectures provide various memory barriers (read-read, read-write, etc.), so nonblocking data structures can enforce the desired ordering of memory operations.

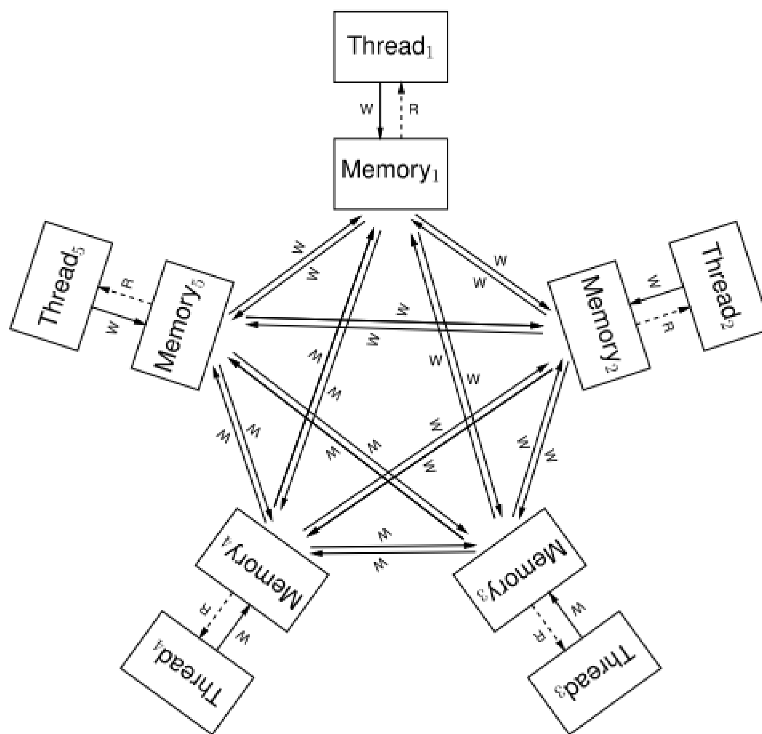

Figure 2.2: ARM storage subsystem. The illustration taken from [11].

2.2.4 Memory Barriers

Memory reordering drastically improves performance but also makes concurrent programming more difficult. Not to mention that it would be almost impossible to write correct multithreaded programs without control over reordering. Therefore, both compilers and processors provide methods to enforce an ordering constraint on memory operations. They are known as *memory barriers* (alternatively *membars*, *memory fences*, or *fence instructions*).

The exact semantics of a membars is hardware dependent and defined by the

memory consistency model. On architectures with relaxed memory models, there are usually many memory fences enforcing different types of ordering constraints. Since compilers can also reorder instructions it is necessary to have separate fence instructions that prevent this type of reordering. They are also referred to as *compiler barriers* and they are defined just on the programming language level, i.e., compiler barriers are architecture-independent.

Memory barriers are typically used in low-level code to implement complex synchronization primitives and lock-free data structures. In most cases, explicit use of memory barriers is not necessary. In high-level programming languages, such as Java, C#, or even C++, synchronization is done by primitives that are built on top of memory fences. In fact, many languages do not provide any instructions to enforce ordering constraints.

2.3 Problems with Concurrency

2.3.1 Race Conditions

One of the most common, dangerous, and elusive errors in multithreaded programs is a race condition.

Race conditions are serious programming errors that can lead to data integrity issues and abnormal program termination. Such errors can be very difficult to detect since they occur only under certain conditions, which are often difficult to repeat on purpose, even if these conditions are known - for example, a special sequence of switching threads.

It should be noted that there are two similar, but with different meanings, concepts. One of them is called *race condition* and refers to the non-determinism of execution of parallel processes. Another, *data race*, refers to a special case of non-determinism that occurs in multithreaded programs when threads do not work properly on shared memory.

Definition (Data race)**.** ***Data race*** *is a situation when two or more threads of execution can simultaneously access the same memory area (for example, a variable) and at least one of these accesses is a write.*

To get better performance, some multithreaded applications intentionally allow data races. Furthermore, sometimes unintentional data races are not considered harmful, for example, a data race on an integer variable used to calculate statistics.

Definition (Benign data race)**.** *A* ***benign data race*** *is an intentional data race whose existence does not affect the correctness of the program.*

In the following subsections, there are presented examples of the most frequent race done in C/C++ programs [6].

Simple Race

A rather classical example, where two threads concurrently increment the same integer variable. To increment a variable, the following sequence of operations have to take place: reading the variable from the memory, incrementing it, and writing it back to the memory. These instructions can interleave in any order. Hence, there is a chance that after executing this piece of code by both threads, variable `A` will be set to 1 (instead of 2).

```
// Thread 1
A++;
```

```
// Thread 2
A++;
```

Notification

Consider a scenario, where a plain variable is used to send notifications between threads. This is tricky because it may result in a data race on some platforms, while it may be perfectly correct on others. The correctness depends on the optimization level used in the compilation and the memory consistency model.

```
// Thread 1
bool flag = false;
while (!flag)
    sleep(1);
```

```
// Thread 2
flag = true;
```

Publishing Objects without Synchronization

One thread initializes a pointer variable (which was initially null) to a new instance of an object. At the same time, another thread waits while the object pointer has a null value. Without any synchronization, thread 2 can see that `obj` is non-null, although it has not been fully initialized yet.

```
// Thread 1
MyObject *obj_ptr = NULL;
...
obj_ptr = new MyObject();
```

```
// Thread 2
while (obj_ptr == NULL)
    yield();
obj->do_something();
```

2.3.2 Atomicity

The atomicity is a close concept to race conditions. *Atomic operations* are the base of almost all synchronization primitives.

Definition (Atomic operation)**.** ***Atomic operation*** *is a sequence of one or more statements that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation [1].*

Plain operations on primitive variables are not considered atomic in most programming languages. For example, a simple incrementation of a variable `counter++` in C language will be compiled to three distinct instructions (load, increment, and store):

```
mov     eax, DWORD PTR counter[rip]
add     eax, 1
mov     DWORD PTR counter[rip], eax
```

If this code is run concurrently, the instructions can be interleaved in all possible ways. In the worst case, the incrementation operation will not change the value of the variable.

Some programming languages allow annotating variables as atomic so all operations on that variable will be indivisible. In the C++ language, one can define an atomic variable simply by using one of the atomic types: `atomic_bool`, `atomic_int`, etc.

Using atomic variables ensures that simple data races cannot happen. However, this synchronization method will not be enough in case of complicated data structures and algorithms. More complex synchronization primitives have to be built on top of that.

Another way to avoid race conditions is to ensure *mutual exclusion* to shared memory, file, or a device. In other words, if a process is using a shared resource with this property, the other processes are excluded from doing the same thing. A segment of code with the mutual exclusion property we define as a *critical section*.

The differences between atomic operations and critical sections are subtle: the former are those operations that finish the computations as a whole (no interruption is allowed). In the critical section, however, the only restriction is that it cannot be executed concurrently by more than one thread. Thus, every atomic operation is indeed a critical section, but not the other way around.

The most common synchronization primitive is *mutex*. It can be used to realize a critical section.

Definition (Mutex)**.** *A* ***mutex****, short for "mutual exclusion", is an object that is in one of two states: unlocked or locked (taken). If mutex M is unlocked, any thread T can perform a lock operation. If the mutex is taken, then it can be unlocked only by the owner (the thread that previously acquired the mutex). When thread T' tries to take mutex M, which is already locked by another thread T,T' is blocked until it is possible to acquire mutex M.*

This definition will be useful later in this thesis.

2.3.3 Deadlock

In this subsection, we will review another class of concurrency bugs – deadlocks. Firstly, we start with a short description of the underlying principles of deadlock. Then, the common approaches to dealing with them are presented.

Definition (Deadlock)**.** ***Deadlock*** *is the permanent blocking of a set of processes, in which each process is waiting for an event that only another process in the set can cause.*

The blocked process continue to wait forever because all other processes are waiting and thus none of them can trigger an event that could wake up any of these processes. These events include having exclusive access to some resource (devices, data records, files), holding a lock, etc.

Note that, not every resource may cause a deadlock. As an example, let's assume that there are two threads and two resources: a printer and memory. In the beginning, the first thread owns the memory and the second one uses the printer. Then, the first thread requests the printer, but since it is already used, it has to wait until the printer has been released. At the same time, the second thread wants to use memory. Normally, this would lead to a deadlock, but memory has a unique property – it can be preempted. The operating system can pause the first thread, copy its memory to auxiliary storage, and then temporarily lend that resource to another thread. Thus, the progress of program execution will not be stopped. The conclusion is that not every resource can cause a deadlock.

It has been proved [2] that four conditions must be met for a deadlock to be possible:

1. **Mutual Exclusion**. Only one process may use a resource at a time. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and wait**. A process can hold allocated resources while waiting to acquire additional resources.
3. **No preemption**. No resource can be forcibly removed from a process holding it.
4. **Circular wait**. There must be a circular list of two or more processes, each of which is waiting for a resource held by the next member of the chain.

Referring to the previous example, a deadlock could not happen, because the third condition was not fulfilled.

Deadlocks can be naturally viewed in terms of a directed graph called a system *resource allocation graph* [1]. The graph has two kinds of nodes: processes and resources. We put an edge from a resource node to a process node if the resource has just been acquired by the process. If a resource has been requested by a process but not granted yet, we create a graph edge directed from the process to the resource. Resource allocation graphs are a popular way to think about deadlocks, because of a plethora of mathematical theorems and tools used in graph theory. These graphs are often utilized in many algorithms used for deadlock detection and prevention.

Figure 2.3 shows an example deadlock. Process P1 owns R2 and requests R1, while P2 holds R1 but waits for R2. For the sake of simplicity, we assume that there can be only one instance of a resource type. A cycle in the graph means that there is a deadlock. The processes involved in the deadlock are in the cycle because for each process its predecessor is waiting for a resource held by that process.

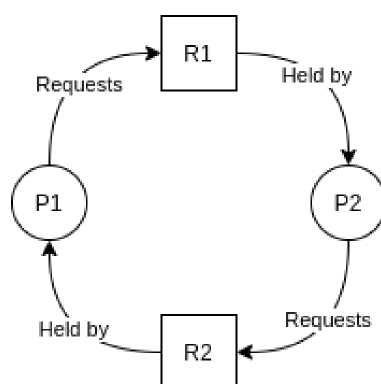

Figure 2.3: An example of resource allocation graph

There are three main methods for dealing with deadlocks:

1. Avoidance and prevention by using some protocol.
2. Let a deadlock occur, detect it, and then recover.
3. Ignore the problem altogether.

If any of the four aforementioned conditions is never satisfied then deadlocks are effectively prevented from happening. For example, if we want to eliminate the circular wait, we can introduce a rule that a thread is allowed to hold only a single resource at any moment. Another way is to provide an ordering of all resources. The rule would be now as follows: the process can request any resources it wants to, but the requests have to be issued in the correct order. It can be shown that in this situation, the resource allocation graph will never have cycles, and thus it is impossible for a deadlock to occur.

In another technique, the system does not make any attempts to prevent deadlocks from occurring, but instead, it lets them do so. This method bases on detecting

when this happens and then taking some action to recover the system after the deadlock. One of the most straightforward and uncomplicated algorithms for detection deadlocks is based on resource allocation graphs. Such a graph can be dynamically constructed by observing the execution of each thread. If this graph contains any cycles, a deadlock exists. Any graph traversal algorithm can be used to find a cycle. After the deadlock-detection algorithm succeeds and a deadlock is detected, various actions can be taken.

The simplest way to handle it is to report the error and kill the program. There are also more sophisticated techniques, but none of them are especially attractive. Sometimes, it may be possible to take a resource away from the currently holding processes and give it to another process. However, the ability to do that is highly dependent on the nature of the resource. This method is rather theoretical since in most cases recovering this way is difficult or even impossible. The system can also be recovered through rollbacks. In this case, the processes have to be checkpointed regularly. Process checkpointing is saving its state (context, local variables, etc.) so that it can be restarted later. When a deadlock occurs, the process that owns the needed resource must be rolled back a point in time before it requested that resource. As the result, the resource can be now assigned to another process, because the process that previously held the resource is reset to an earlier moment when it did not have it.

Although it may seem unacceptable, ignoring deadlocks is the solution used by most programs. If deadlocks occur infrequently and the system crashes more often due to other failures, it may not be worth the effort to do anything else. This method is cheaper to use than any other, because for one simple reason – other methods must be used constantly throughout the execution of the program. Furthermore, most programs cannot simply afford to have a significant performance loss to eliminate deadlocks.

2.4 Concurrency in Operating Systems

Traditional Unix kernels had a single lock [5], which was held by a thread while kernel code was running, so no other kernel code could interrupt that thread. Therefore, on single-processor machines, concurrency-related bugs were not a problem. That approach offered simplicity, but it had no longer worked on systems with an increasing number of processors. As multicore processors became more prevalent and user-space applications used many threads, many kernels have been transformed in a way that many more things are going concurrently. It has also, however, significantly complicated the task of kernel programming. Programmers cannot now assume that their code will be executed by a single thread. They must now take into account the concurrency in their designs and be aware of the side effects of the synchronization methods they use.

In modern operating systems, there are many sources of concurrency and, thus, possible race conditions. The concurrency comes from:

1. multiple user-space processes;
2. SMP systems[1]: multiple processors are executing the same piece of kernel code;
3. kernel preemption: any thread running in kernel space can be preempted by another process with higher priority. This introduces concurrency even on uniprocessor machines;
4. interrupt handlers: if hardware triggers an interrupt, the currently running thread is preempted by the interrupt handler.

To guarantee correctness, a synchronization between all possible sources of concurrency must be imposed. Note that different critical sections have different properties, so the kernel provides many primitives for different needs. In most of the operating systems, there are so many locking primitives that it is officially deemed as a bug [19].

Generally, one can distinguish two categories of synchronization primitives in the kernel: low-level (e.g., atomic operations, memory barriers, spinlocks, disabling interrupts) and high-level (mutexes, readers-writer locks, conditional variables, semaphores). The implementations of the former ones are largely architecture-specific and they are the base for high-level primitives.

Most of the high-level primitives work in a similar way to their user-space counterparts. However, there are two main differences between them:

- The kernel-space implementation of a synchronization primitive is more difficult – in the user space most of the lock logic is delegated to the kernel through system calls.
- When using kernel locks, a lot of errors can be made that are not user-space errors. For example, it is an error if a thread with a mutex goes to an unbounded sleep[2] state.

Race conditions occur as a result of shared access to resources, so a good rule of thumb is to avoid shared resources whenever possible. In the kernel, however,

[1] Symmetric multiprocessing (SMP) is a parallel computer architecture in which multiple processors share the memory and other resources of one computer.

[2] In a bounded sleep the only resource a thread needs is CPU time for the owner of the lock that has been just acquired. So we know that another thread will release the lock soon and the waiting thread will be able to proceed with its execution. In an unbounded sleep, a thread waits for an external event, so there is no guarantee that it will be ever awakened.

such sharing is often more required than in regular user-space applications. Hardware resources are, by nature, shared. This is another thing that makes concurrent programming in the kernel more difficult.

2.5 Generic Ways of Detecting Bugs

Numerous race detection tools have been proposed and built. Most of them, however, are based on the same ideas. In this section, we discuss the generic approaches and implementation techniques.

2.5.1 Lockset Algorithm

The Lockset Algorithm is built on the assumption that each variable that can be used by multiple threads must have a corresponding mutex synchronizing accesses to it. The idea is that if two threads access the same unlocked variable, then there is a possible data race. This algorithm was developed for the dynamic detector Eraser, which was used to detect data races in several types of programs, ranging from the AltaVista Web search engine to introductory programming exercises written by undergraduates [13].

The Lockset algorithm follows one simple locking rule that every shared variable must be protected by some mutex, that is, the lock is held by any thread whenever it accesses the variable. The algorithm verifies if the program follows this rule by monitoring all reads and writes. In the beginning, the algorithm has no way of knowing which locks are intended to protect which variables. The idea is to observe the execution of the program and dynamically infer the protection relation.

The Lockset Algorithm works as follows. Each shared variable V is matched with a set of mutexes (lockset) $LS(V)$, which were taken each time this variable was accessed by possibly different threads and different parts of code within the same thread. The original meaning of $LS(V)$ for each variable is specified as the set of all possible mutexes protecting the variable. Each time thread T accesses shared variable V, the $LS(V)$ is updated to the intersection of the old value of $LS(V)$ and the set of mutexes that T currently holds. If the set $LS(V)$ becomes empty, it means that there is not a single mutex that would synchronize accesses to V. In this case, an error message is displayed. The pseudocode for the algorithm is shown in Listing 3.

```
For each variable V, initialize LS(V) to the set of all locks.
On each access to V by thread T:
  set LS(V) := the intersection of LS(V) and locks_held(T);
  if LS(V) = {}, then issue a warning.
```

Listing 3: Pseudocode for the Lockset algorithm.

The Lockset algorithm does not allow false negatives, that is, if a race condition occurs during the execution of the multithreaded code, it will be reported by the detector. If some mutex $M \in LS(V)$, then either it was held during each access to V, or there were no accesses to V yet. This means that if the Lockset algorithm did not detect a race condition on the variable V, then this variable was either not accessed, or at least one mutex was locked during all accesses. Thus, a race condition is, by definition, not possible.

Unfortunately, the Lockset algorithm has false positives, i.e., if the detector reports a message about a found data race, this does not mean that an error is possible. In particular, false positives can occur when a program uses synchronization tools other than mutexes (semaphores, atomic variables, reader/writer locks, etc.). This version of the algorithm also is not aware of benign data races.

The main limitation of the lockset approach is that it does not detect data races, but it verifies whether a quite specific locking discipline is violated. Unfortunately, many programs (and in particular kernel code) use other sophisticated synchronization methods, which are more complex than simple locks. Therefore, this detector cannot be applied to every program.

2.5.2 Happens-before Algorithm

The main idea of the algorithm is to observe the program execution as a sequence of events. We distinguish two types of events: memory access events and synchronization events. Originally, back in 1978, Leslie Lamport established a framework with two types synchronization events: *signal* and *wait* [7]. *Signal* event corresponds to sending a message by one process and *wait* is waiting for the same message by another process. However, this can be generalized to cover any kind of synchronization primitives [8].

Let us consider a pair of operations that synchronize one thread with another and operates on the same data structure. We are going to name the operations in this pair as either *release* or *acquire*. Alternately, we can say that they *have release (or acquire) semantics.*

If we give release semantics to message send and acquire semantics to message receive, we get the old definition of synchronization events. A thread sending a

message synchronizes with the thread (or threads) that receives the message. The release-acquire definition can be used for many synchronization primitives, ranging from CPU message buses ensuring that their caches are coherent, locks preventing concurrent accesses to the same data structure, ending with abstract concepts with complex semantics.

Happens-before algorithms use partial ordering relation *happens-before* to check the simultaneity of events.

Definition (Happens-before)**.** *Event A* ***happens-before*** *event B ($A \prec B$), if at least one of the following statements is true:*

- *A and B were executed by the same thread of execution and A happened before B;*
- *A is an event with release semantics on some synchronization primitive P, and B is an event with acquire semantics on the same primitive P;*
- *There are events A' and B' such that, $A \preceq A' \prec B' \preceq B$ (transitivity property).*

Definition. *Two distinct events A and B are said to be* ***concurrent*** *if $A \not\prec B$ and $B \not\prec A$.*

Figure 2.4 illustrates this definition. In three threads of execution T_1, T_2 and T_3 there are events $E_1, E_2, \ldots, E_7$. In addition, there are two synchronization primitives P_1 and P_2.

According to the definition:

- $E_1 \prec E_4$, as E_1 happened before E_4 in one thread of execution;
- $Release(P_1) \prec Acquire(P_1)$ and $Release(P_2) \prec Acquire(P_2)$ since these are signal and wait events on the same primitive;
- $E_1 \prec E_7$ by the property of transitivity;
- $E_4 \not\prec E_2, E_2 \not\prec E_3$, since the events are not ordered.

To see that the definitions also apply to more abstract concepts, consider a scenario [8] where two threads access a global variable `text`:

```
// thread 1
text = "foo";
thread_create(&t);                      // thread 2
                                        print(text);
                                        text = "bar";
thread_join(t, NULL);
print(text);
```

Procedure `thread_create()` has release semantics and thus synchronizes with the beginning of the second thread (which has acquire semantics). Hence, everything that was written before the thread was created can be then safely read. Exiting from thread 2 has release semantics and is synchronized with `thread_join()` (which has acquire semantics). Thus, thread 1 can safely read the variable with the guarantee that it will not cause a data race.

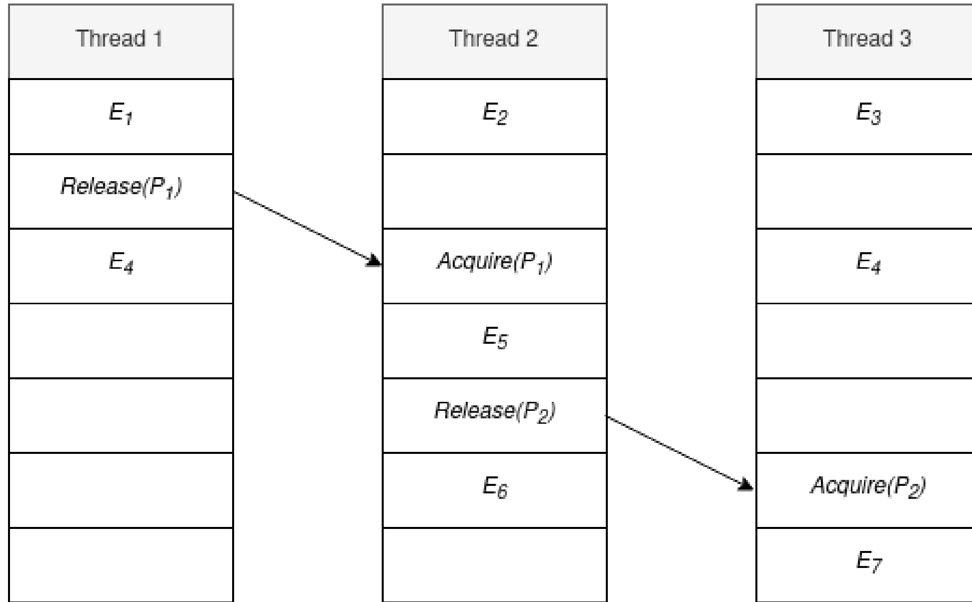

Figure 2.4: Illustration of the happens-before relation

Detectors based on this algorithm track memory accesses and the use of synchronization primitives during program execution. The detector builds the happens-before relation defined above on the set of events that have occurred. If there are two concurrent accesses to the same memory address and at least one of them is a write, a race condition message is reported.

There are several ways to implement the calculation of the happens-before relation on a set of events that have occurred. One of them is building a directed graph of events. Edges are drawn between successively occurring events of individual threads, as well as between pairs of release-acquire events. To check the happens-before relation for two separate events, a graph search algorithm can be used. Unfortunately, traversing the graph to check each memory access results in immense performance overhead.

There are other algorithms for calculating the happens-before relation. They are equivalent to traversing the event graph. Most often, more efficient algorithms based on *vector clock* are used.

Definition (Vector clock)**.** ***Vector clock*** V *is a mapping from set of all threads* τ *to a set of non-negative integers* $V : \tau \to \mathbb{Z}_+$.

Each thread in the program T *is mapped to its vector clock* V_T*. In addition,*

each synchronization primitive P also has its vector clock V_P. The quantity $V_T(T)$ is called the thread's own time.

Vector clock values are defined as follows:

- *When thread T is created, its vector clock is set to*

$$V_T(t) = \begin{cases} 1, & t = T \\ 0, & t \neq T \end{cases}$$

- *When a memory access event happened in thread T, the thread's own time is incremented:*

$$V'_T(t) = \begin{cases} V_T(t) + 1, & t = T \\ V_T(t), & t \neq T \end{cases}$$

- *When a new synchronization primitive P is created, its vector clock stores zero for all threads*

$$V_P(t) = 0, \text{ for all } t \in \tau$$

- *When a release operation of a synchronization primitive P is executed by thread T, the vector clock of this primitive is updated as follows*

$$V'_P(t) = \max(V_P(t), V_T(t)), \text{ for all } t \in \tau$$

  and then thread's own clock is incremented $V'_T(T) = V_T(T) + 1$

- *When a acquire operation of a synchronization primitive P is executed by thread T, the vector clock of the thread is updated as follows*

$$V'_T(t) = \max(V_P(t), V_T(t)), \text{ for all } t \in \tau$$

For threads T and T' vector clock element $V_T(T')$ corresponds to the thread's T' time at the moment when the thread T synchronized with it for the last time.

Theorem ([16])**.** *Let the events A and B happened in threads T_1 and T_2 respectively, while V_{T_1} and V_{T_2} are their vector clocks at the time of occurrence of events. Then:*

$$A \prec B \iff V_{T_1}(T_1) < V_{T_2}(T_1)$$

2.5.3 Fuzzing

Fuzzing is an automatic way of testing the program by feeding it with a series of inputs and observing how it behaves. The fuzzing process starts with data that is manually provided or randomly generated. If the program crashes or hangs, it means that a new error is found and the vulnerable input is saved for future verification. After that, the fuzzer proceeds with other data.

To maximize the number of founds bugs, many fuzzers add annotations to the binary of the tested program to trace the code flow. This allows detecting how the provided input changes the target's behavior. Based on that, the fuzzer can choose inputs such that more branches of the program are reached. Many fuzzers employ sophisticated methods for generating inputs like genetic algorithms [27]. Therefore in combination with code instrumentation, it discovers many interesting test cases that trigger previously unexplored states of the program.

There are many advantages of fuzzing over other testing methods:

- Fuzzing process is almost completely automated.
- It constantly verifies different parts of the program as opposed to e.g., unit testing, where executing a test suite always has the same result.
- Fuzzing is scalable – the code can be tested simultaneously by many machines.
- Fuzzers can be combined with other dynamic verification (such as data race detectors) to find a range of different types of bugs.

Nowadays, many modern programs (including operating systems, browsers, and cryptography libraries) are being constantly fuzzed to find new bugs and vulnerabilities that could be possibly used in an attack.

Chapter 3

Tools for Finding Concurrency Bugs

This chapter contains a non-exhaustive list of various tools used for detecting concurrency bugs described in Chapter 2. The overview focuses on those that are applicable both for user-space programs and modern operating systems like: Linux, NetBSD, FreeBSD, and Windows.

3.1 Thread Sanitizer

The Thread Sanitizer (TSAN) [6] is a dynamic data-race detector created by Google. It is a part of the set of sanitizers – tools performing dynamic analysis of C and C++ programs. The Thread Sanitizer detector has two parts: one is a compiler component and the other is a run-time library component. The compiler component implements the instrumentation for memory accesses and the run-time library contains annotations for used synchronization primitives and the implementation of the algorithm. During program execution, the program monitors memory accesses and usage of synchronization primitives and passes them for processing. The algorithm is based on the happens-before relation and uses the concept of a vector clock.

3.1.1 Compiler Instrumentation

The functionality of dynamic detectors bases on observing events occurring in the program. They have to intercept some instructions like memory accesses and prepend them with an additional code. The process of introducing additional code into a program without changing its main functionality is called *code instrumentation*. Instrumentation is performed either at compile-time, or before the execution of the program, or during the execution, and is called compiler, static and dynamic in-

strumentation respectively. For user applications, well-known examples of systems using dynamic instrumentation are Valgrind [9] and Pin [14]. Gcov, a test coverage tool, and GCC are popular examples of compiler instrumentation.

In the TSAN, the compiler instruments every memory access in the program unless it can be proven to be race-free or redundant. An example of race-free accesses are reads from constant and global variables. Memory accesses are simply prepended with a function call `__tsan_{read, write}N(address)`, where `N` denotes the size of the accessed variable. Further, function entry and exit are instrumented with `__tsan_func_entry(caller_pc)` and `__tsan_func_exit`. The compiler also adds instrumentation for specialized memory accesses like atomic variables, but we will not go into detail.

As an example consider the following code:

```
void foo(int *p) {
  *p = 42;
}
```

The compiler changes it to:

```
void foo(int *p) {
  __tsan_func_entry(__builtin_return_address(0));

  __tsan_write4(p);
  *p = 42;

  __tsan_func_exit();
}
```

Note that the compiler only inserts function calls and does not provide their implementation. It is the run-time library's job to implement them. The TSAN contains parts that depend on the CPU architecture, memory consistency model, and synchronization methods available in the system, so the run-time library must be created independently for each use case.

3.1.2 Shadow Memory

Many detectors store additional data associated with each memory location in the application. It can contain, for example, information about the availability of this memory cell for reading or writing, or information about the last few accesses to this memory cell. The memory that stores this additional data is called *shadow memory*.

To access additional data for a given application-memory location, the detector must calculate the address of the corresponding shadow-memory location. The simpler the procedure for calculating this address, the higher the detector performance. In other words, for the effective operation of the detector, it is required that the address of the shadow memory can be obtained by a simple procedure, knowing the address of the analyzed memory cell.

One of the simplest options for storing shadow memory is to use a continuous portion of the address space, into which the entire address space is mapped using compression and translation. This version of the shadow memory device is used in the Address Sanitizer and in the Thread Sanitizer detectors. Here we describe the Thread Sanitizer. The integration of the other sanitizer in the Mimiker OS is a subject of a separate project [15].

The Thread Sanitizer uses shadow memory to store additional information about the program's main memory. For each 8-byte memory location, the detector stores 32 bytes of shadow memory. These 32 bytes are 4 cells (called *Shadow Words*) of 8 bytes, each of which describes one of the last calls to the corresponding memory cell.

The shadow memory is allocated every time the program allocates memory using the memory allocator. *Shadow Words* are addressed using a direct address mapping (so no memory accesses are required to compute the shadow address). The i-th *Shadow Word* location is calculated using a formula: $8 \cdot i + 32 \cdot (\text{address}/8) + \text{offset}$. The offset is chosen for each architecture depending on where the shadow addresses reside within the virtual address space.

Each of the shadow memory cells consists of several parameters for accessing memory. It includes the identifier of the thread that accessed the memory, its own time (the time used in vector clocks) during access, the offset from the beginning of the 8-byte memory location, corresponding to the address of the access, the size of the access, and a flag indicating whether the access is a write or read.

Figure 3.1 shows an example of an application memory address space and how it is mapped to a special shadow region. Each entry of the shadow map consists of 4 *Shadow Words*, so the TSAN can save the information about up to 4 accesses to every aligned 8-byte word of the application memory. Since we only consider the virtual address space, it is not a problem to have a memory layout as presented below – the actual application memory can be mapped into any addresses. Note that, for the sake of brevity, some parameters like the size of an access are omitted.

Each time a memory location is accessed, the corresponding shadow memory locations are properly updated. However, this does not mean that the four most recent calls to the program memory location are stored in the four shadow memory locations. The procedure for updating the contents of shadow memory cells is described below. The cell size of 8 bytes allows for efficient manipulation of the shadow

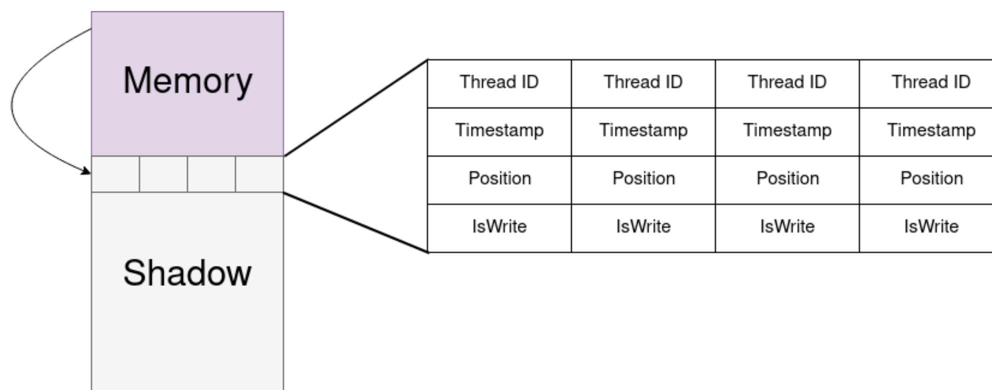

Figure 3.1: The virtual address space mapping.

memory cell with a single write or read operation.

3.1.3 Run-time Library

For the detector to work, the implementation of the algorithm must be included in the application under test. In addition, the shadow memory has to be allocated and managed. The instrumentation alone is not enough for these purposes. The program must also embed a run-time library. It contains new functions and replaces the implementation of existing ones.

The TSAN tracks the usage of all synchronization primitives by using so-called annotations. Annotations are special calls to detector functions. They are used just to inform the detector about the synchronization event, so they do not affect the operation of the synchronization primitives. There are a lot of synchronization primitives used in C/C++ (mutexes, spinlocks, rwlocks, semaphores, memory access barriers, etc.) and they all need to be annotated. The absence of annotations or incorrect annotations for at least one of the primitives will lead to false positives of the detector. Functions whose calls are added as annotations to the implementation of the synchronization primitives, perform operations on the vector clocks of threads and the vector clocks of the primitives themselves. The operations are the same as the one given in the description of the happens-before algorithm.

Annotation of memory barriers is especially difficult. This is because the memory barrier itself does not perform any synchronization. However, one barrier can change the semantics of several preceding or subsequent atomic memory accesses. To handle the synchronization performed by barriers, in addition to the usual vector clocks, each thread also has a couple of vector clocks: release vector clocks and acquire vector clock. The main idea is that a sequentially executed atomic operation and a memory access barrier behave in the same way as the atomic operation with release or acquire semantics. Consider an example – let the thread sequentially execute the write barrier and atomic write with relaxed semantics. This sequence of actions is equivalent to an atomic write operation with release semantics. The

semantics of a memory access barrier is release if it is a write barrier and acquire if it is a read barrier. For details on how the additional vector clocks should be updated, refer to [6].

The run-time library of the TSAN also consists of the code handling memory accesses. Before each memory access, the detector checks for the possibility of a race condition between this access and one of the previous calls to the same memory cell. To do this, the detector first checks to see if the memory section overlaps with the 8-byte cell that was accessed. If this condition is met, then further checks are performed following the definition of a race condition. It is checked if the accesses occurred in different threads, and if at least one of them is a write operation and if the accesses occurred without synchronization. Synchronization check is done using vector clocks. If a race condition is possible by definition, then an error message is printed.

Note that it is impractical to check the possibility of a data race between the current memory access and all calls that have occurred to this cell during the entire time of the program operation until the current moment. It would consume tremendous amounts of memory. Checking for possibilities of data races is only carried out for calls whose description is stored in the corresponding Shadow Words. At the next memory access, the contents of the shadow memory must be updated. The new Shadow Words are inserted in the place of an empty Shadow Word or in place of Shadow Words that *happened-before* the new one. If no place for insertion is found, a random Shadow Word is evicted.

The tool that only tells if a data race happened without any additional information would not be too useful, so an informative message about bugs has to be prepared. Retrieving the stack trace of one of the threads involved in the data race is easy – the thread that accessed as the second one can simply dump its context. The problem is with the stack traces and other information of previous accesses involved in a data race. It possibly could happen a relatively long time ago. The solution is to keep a per-thread cyclic buffer of events like memory accesses (for the exact top stack trace frame) or function entry/exit (for stack traces). With that, it is possible to replay the trace to get a stack trace for interesting memory access. Due to the size limitation of the buffer, information is lost after some time.

Listing 4 shows an error report printed by the TSAN when it detects a race condition. The report contains a lot of information that can help to find and fix the error.

```
==================
WARNING: ThreadSanitizer: data race (pid=26327)
  Write of size 4 at 0x7f89554701d0 by thread T1:
    #0 Thread1(void*) simple_race.cc:8 (exe+0x000000006e66)

  Previous write of size 4 at 0x7f89554701d0 by thread T2:
    #0 Thread2(void*) simple_race.cc:13 (exe+0x000000006ed6)

  Thread T1 (tid=26328, running) created at:
    #0 pthread_create tsan_interceptors.cc:683 (exe+0x00000001108b)
    #1 main simple_race.cc:19 (exe+0x000000006f39)

  Thread T2 (tid=26329, running) created at:
    #0 pthread_create tsan_interceptors.cc:683 (exe+0x00000001108b)
    #1 main simple_race.cc:20 (exe+0x000000006f63)
==================
ThreadSanitizer: reported 1 warnings
```

Listing 4: Report from the Thread Sanitizer. Taken from [18].

3.1.4 The Kernel Thread Sanitizer

The Thread Sanitizer was created for user-space programs. Despite that, the algorithm has been adapted to work in the Linux kernel [17]. The high-level idea stays the same, but due to the complexity of operating system kernels, the run-time library is completely rewritten.

The first issue comes from the fact, that the Linux kernel uses much more synchronization primitives. Each subsystem can have its own locking rules with highly customized synchronization primitives. Hence, the KTSAN is more prone to false positives, since it is harder to correctly annotate each of these primitives. Another problem with the Linux kernel is that there are a lot of benign data races. They are not considered harmful, so there is no motivation and a real need to fix them or even annotate them correctly. As a result, these data races generate many uninformative reports, so reports about real data races can be missed.

The complete code of the kernel cannot be sanitized. For instance, let us consider a scheduler, a part of the kernel, which periodically switches code execution on each processor core to one of the pending threads. Normally, the KTSAN assumes that synchronization can only happen with the usage of well-defined synchronization primitives. When switching threads, the scheduler uses different synchronization methods that have to be ignored by the KTSAN. Otherwise, threads that are sequentially started by the scheduler on the same core would seem to be synchronized. In addition, the detector also ignores memory accesses that occur inside the

scheduler. Ignoring these events is also explained by the fact that it is unclear to which thread these events are attributed because they occur at the moment when threads are switched. Thus, this makes it extremely difficult to find data races in the scheduler.

There is also another important difference between user space programs and the Linux kernel. Typically, there are no more than a few dozen threads in a regular program. The Linux kernel can have up to a thousand threads under the normal load. In addition, there are hundreds of thousands of synchronization primitives and for each of them, we need to keep vector clocks (each of the length of the number of threads). No synchronization primitive can be omitted since it would lead to a false positive. As the result, the KTSAN requires a lot of physical memory just for vector clocks. The original implementation of the KTSAN in the Linux kernel consumes about 16 GB of memory [17], which makes the algorithm difficult to use.

3.2 Kernel Concurrency Sanitizer

KCSAN [20] is an alternative data-race detector based on the same compiler instrumentation as the TSAN, but with a different run-time library. It trades precision for performance and memory consumption (as it does not use shadow memory). Unlike other sanitizers developed by Google, the KCSAN was designed solely for use in an operating system kernel. It was integrated into the Linux kernel in November 2020 and since then it is used extensively. The KCSAN has already found over 100 errors [24], many of which have been already fixed.

KCSAN detects races by using a *watchpoint*-based sampling approach. The idea is to sample a small number of memory accesses and check if a data race occurred on these addresses. The sanitizer does it by inserting code breakpoints at randomly chosen memory access instructions. A breakpoint is a mechanism to mark a place in the code so that a running program will temporarily stop there. They are primarily used for debugging purposes. Breakpoints can be triggered by various conditions such as executing a specific code branch or modification of a specific location in an area of memory. The latter kind of breakpoint is referred to as a watchpoint. During the interruption, the programmer or another program can verify whether the watched process is functioning as expected.

The algorithm relies on observing that two accesses happen concurrently. Namely, KCSAN sets up a watchpoint for some memory address and then waits until it gets triggered. More precisely, for each *plain* memory access (that is, without any annotations):

1. Check if a matching watchpoint exists. A data race is encountered if the watchpoint was found and at least one access is a write.

2. If no corresponding watchpoint exists, check when the last watchpoint was created. If it was created adequately long ago, set up a new watchpoint and wait for a small delay.
3. Also read the data before and once after delay. If the values do not match, a data race of an unknown origin occurred.

The algorithm has to distinguish different types of memory accesses. For example, variables marked as atomic can be accessed without locking. Hence, some memory accesses have to be skipped, as it would lead to false positives. KCSAN would report an occurrence of a data race, but in fact, it did not happen.

Note the third step in the algorithm. The watched variable could be changed while the thread was waiting. It could only happen in a situation where an uninstrumented part of the code made access and changed the value of the variable.

Watchpoints can be implemented either in hardware or in software. KCSAN, with the help of code instrumentation, uses the latter option. The main advantages of using software watchpoints are portability (as they do not rely on hardware) and greater convenience. Creating a watchpoint is as simple as adding another entry to the list of watchpoints. The single entry uses an encoding that stores an access address, size, and type (read or write) in one machine word.

KCSAN was created as a better alternative to the Kernel Thread Sanitizer and thus it has many advantages over its predecessor [25]:

1. Memory overhead: The overall memory overhead is only a few MiB. All that KCSAN requires is a small array to encode watchpoint information, which is rather negligible – there are at most several dozens of watchpoints. KTSAN required an immense amount of shadow memory.
2. Performance overhead: KCSAN does not perform any heavy computations during memory accesses.
3. Annotation overhead: Minimal annotations are required outside the run-time library. In contrast to KTSAN, KCSAN does not need to be aware of synchronization primitives used in the kernel. Hence, the implementation of the sanitizer is self-contained and thus is easier to maintain.

As often happens in computer science, trade-offs have to be made. In this case, the performance boost is traded for precision. There are two big disadvantages of this algorithm:

1. KCSAN is not aware of the memory model. It does not know whether a CPU has reordered memory accesses. Therefore, it cannot detect some subtle bugs, such as a missing memory barrier. This will result in false negatives (some data races may be missed).

2. The accuracy of the analysis is much worse because only some accesses are monitored. The run-time library has to be fast, so the simplest sampling strategy is used – a watchpoint is set up every n-th memory access. Again, this results in possible false negatives.

After a data race is detected, an informative report is created. In comparison to KTSAN, retrieving stack traces of threads involved in a data race is easy. When KCSAN finds a data race through fired watchpoint, it has all needed information about both threads. We know that their next instruction is accessing conflicting addresses. One of them has just made access to the variable and the second thread (the one that set up the watchpoint) is waiting. Thus, all needed information can be simply read from their current context.

3.3 DataCollider

DataCollider [21] is another lightweight and effective tool for detecting data races during the execution of the program. DataCollider is unaware of synchronization protocols (such as locking methods) used to protect shared memory accesses. This tool was implemented for the Windows 7 kernel and has found many erroneous data races.

Similar to KCSAN, DataCollider samples a small number of memory accesses at run-time. It chooses memory accesses at random and then inserts code breakpoints there. When a breakpoint is triggered, DataCollider starts detecting data races on that particular memory access for a short time. It uses two strategies at once to do so. Both of them are very similar to the strategies used by KCSAN. Namely, DataCollider sets a hardware breakpoint to trap conflicting accesses by other threads. However, this method will miss conflicting writes performed by either hardware devices or processors accessing the memory location through a different virtual address. To verify if a data race occurred, the value of the memory is read before the breakpoint was created and once after the delay. If a value is changed, we know that there was a conflicting write, and hence a data race.

What distinguishes DataCollider from KCSAN is the sampling algorithm for data-race detection. DataCollider's approach is based on performing static sampling, which works as follows. Firstly, DataCollider generates a sampling set consisting of all program locations that access memory by disassembling a program binary. Then, it performs a simple static analysis to determine which of the instructions are guaranteed to be race-free (e.g. thread-local variables) and removes them from the sampling set.

Initially, DataCollider chooses a small number of program locations from the sampling set. Then breakpoints are set at these locations. If a breakpoint is triggered, DataCollider performs its regular data-race detection procedure, and then

another program location is chosen and a new breakpoint is set at that location.

Even though software watchpoints are flexible and portable, they are slower than their hardware counterparts. Many modern hardware architectures provide a solution to trap the processor when it reads or writes to a particular memory location. As DataCollider was designed exclusively for the Windows 7 kernel (it runs only on the x86 architecture), the authors decided to use hardware breakpoints to effectively monitor possibly conflicting accesses to the currently sampled memory locations.

3.4 Lockdep

This tool differs from the previously described ones in the way that it does not search for data races. The lock dependency correctness validator (lockdep) [26] [28], however, discovers possible deadlock risks. The tool observes a running program and proves (with some assumptions) that none of the locking patterns could cause a deadlock. The validator was originally designed and implemented for the Linux kernel, but because of its simplicity, it can be successfully used in other programs (not only operating systems).

Note that data races and deadlocks can be viewed as complementary concurrency bugs. Data races are caused by an insufficient or inaccurate usage of synchronization primitives. On the other end of the spectrum, we have deadlocks. They usually occur when many locks are used in the program without a proper locking strategy. Therefore, lockdep can be combined with other tools searching for data races to create a comprehensive suite of concurrency bug detectors.

Lockdep does not operate upon a single lock instance, but upon a *lock-class*. We define a class of locks as a group of locks that follow the same locking rules. The motivation is that there are usually thousands of locks, but many of them are treated exactly in the same way, so there is no point in tracking each of them individually. For instance, let us consider the *vnode*[1] structure. There is a lock protecting all fields in this structure. Therefore, this lock is one class, while each *vnode* lock is an instantiation of that lock class.

The validator keeps track of the dependencies between different lock classes while the program is running. In other words, it tries to inference the ordering between locks by observing situations where a thread is attempting to acquire a lock while holding another. To do so, lockdep maintains two lists for each lock class L. One of them (called *before* list) contains all lock classes which have ever been held when a lock of class L is acquired. Therefore, it comprises lock classes that might be acquired before L. This can be conceptually viewed as a list of incoming

[1] *vnode* is the representation of a file in UNIX operating systems. For more details please refer to [1], Chapter 12.

edges in the dependency graph. The other list (called the *after* list) holds all lock classes acquired while L is held. Respectively this is the list of outgoing edges. The ordering of how other locks have to be taken relative to L is uniquely determined by these two lists.

Whenever a lock of class L is acquired, the validator checks if any of the already held locks are on the *after* list of lock L. All locks on that list can only be taken after acquiring L. Hence, if any such are found, it means that the lock ordering rules are violated, so possibly it can result in a deadlock. The validator code also merges the after list of L with the before lists of the currently held locks and searches for ordering violations anywhere within that chain. This can be done with any graph traversal algorithm. If no errors are reported, the lists are updated and the program continues its execution.

Using lock classes instead of individual locks increases the performance, but also imposes a few restrictions about how the code should do locking:

- Invariance of locking rules over time – the locking rules for a given lock cannot change over the lock's lifetime. Even in correct locking protocols, the validator detects data race risk if the locking rules depend on the state of the object. For example, when an object has different locking rules for creation and destruction.
- Common locking rules for the same objects – the second restriction is that all locks from one lock class must follow the same locking rules. Lockdep is not able to spot differences in locking rules between each object. The problem is illustrated by an example code in Listing 5. There is a tree and each of its nodes has private data. The programmer decided that the root node must acquire locks in a different order. The code cannot cause a deadlock, but lockdep would report it as an error.

Traversing the whole dependency graph on every lock taken would impose a massive amount of run-time overhead. The worst-case complexity of checking is $O(N^2)$ (it is equal to traversing the full graph), so lockdep would have to perform thousands of checks for every lock taken with just a few hundred lock classes. The observation is that many of the checks are just redundant. Each locking scenario, that is, a unique sequence of locks taken after each other, has to be verified only once. To keep track of already verified scenarios, each thread can maintain a stack of currently held locks and calculate the hash value of this stack. When the validator verifies the locking scenario for the first time, it puts the hash value into a hash table. If the same sequence of locks occurs again later on and the hash value is already in the hash table, the locking scenario does not have to be validated again.

The key point behind lockdep is that it can find deadlock risks without having to make the system crash. The validator verifies all scenarios in which locks are acquired in the order as observed in the program. Therefore, even though a particular

deadlock might only happen as the result of unfortunate timing, the validator has a good chance of catching it.

Lockdep is giving false positives if the assumptions about the locking are not met. However, false negatives are not possible. If lockdep does not report an error, we are certain that no unfortunate combination or interleaving of instructions can cause a deadlock.

```
struct node_data {
    ...
    mutex_t lock;
};

struct tree_node {
    ...
    struct node_data *data;
    mutex_t lock;
};

void lock_tree_node_and_storage(struct tree_node *A) {
    if (is_root(A)) {
        mutex_lock(A->lock);
        mutex_lock(A->data->lock);
    } else {
        mutex_lock(A->data->lock);
        mutex_lock(A->lock);
    }
}
```

Listing 5: An example code showing different locking rules for the same object.

Chapter 4

Implementation in the Mimiker OS

Mimiker [22] is an open-source operating system developed at the University of Wrocław since 2015 for educational and research purposes. It is inspired by the Unix family, and especially by the *BSD operating systems. As the codebase is steadily getting bigger, concurrency errors are more likely to occur. More and more synchronization primitives are being added, the number of concurrently running processes is increasing, and many complex locking protocols are introduced. This gives rise to the need for automatic verification of concurrency with reliable bug detection. It should not only find bugs already present in the kernel but more importantly, prevent many more from being added in the future. It is worth mentioning that there is already one bug detector integrated into the Mimiker – the Kernel Address Sanitizer (KASAN) [15].

As part of my contribution to the Mimiker OS, I have been investigating possibilities of dynamic verification of concurrency in the system for the last year. This chapter contains a summary of the work done within the kernel. I will describe the implementation of the tools, faced challenges, and mention found errors. I will also discuss possible improvements and future work in detecting concurrency bugs.

4.1 Kernel Thread Sanitizer

The Kernel Thread Sanitizer was the first tool that I tried to implement for the Mimiker OS. Compared to other tools, it offered the biggest accuracy in finding errors. The attempt, however, was unsuccessful. There are a few reasons why I decided to abandon the plan of implementing this tool. The arguments of other Mimiker's contributors were also very useful to make this decision.

First, it would take a substantial amount of time to implement this tool. Besides

the run-time library, many other changes throughout the kernel would have to be made:

1. Adding shadow memory requires many modifications in the virtual memory subsystem. Although shadow memory is already implemented for the KASAN, it differs significantly from the one used in KTSAN and cannot be reused.
2. Some parts of the KTSAN code are machine-dependent, so each supported architecture by Mimiker has to be handled individually.
3. Each synchronization primitive in the Mimiker OS has to be annotated. The annotations, however, would not be the same everywhere, since they are highly dependent on the semantics of the primitive.

The above-mentioned difficulties, however, were not insurmountable. The second and more important problem was code maintainability. Since the Mimiker OS is an academic project, most of the contributors are students and thus, on average, they are involved in the project for two or fewer semesters.

It would not be a problem if the KTSAN was self-contained and independent of the rest of the code. However, the implementation of the KTSAN relies on a code that is constantly modified. New synchronization primitives are regularly added and the existing ones are modified. The implementation of the sanitizer would have to be subsequently adjusted after all these changes. Omission of any primitive would make the KTSAN completely unreliable. Therefore, a long-term maintainer of the KTSAN that understands the code and can constantly fix newly made bugs is necessary. Unfortunately, due to the nature of the Mimiker OS project, this requirement cannot be met.

4.2 Kernel Concurrency Sanitizer

4.2.1 Implementation Details

The implementation consists of a run-time library and a series of changes in the build system. In total, approximately 400 new lines of code were added to the kernel. All changed files can be found online at Mimiker's GitHub repository [23]. Please note that the implementation is partially based on the sanitizer within the Linux kernel. All the work done has already been merged into the mainline of the Mimiker OS.

Changes in the Build System

To integrate the KCSAN with the kernel, the build system had to be modified. Mimiker uses GNU Make for compiling and linking the source code. The build

system comprises a set of rules describing how different parts of the kernel should be built. For example, different rules apply either for kernel code or for user code.

New rules added to the build system have two purposes. Firstly, all files that should be sanitized receive additional compiler flags enabling code instrumentation. Secondly, all kernel files are now compiled with an additional flag indicating if the sanitizer is enabled. This allows us to use the preprocessor's `#if` directive to conditionally compile a part of the code if KCSAN is turned on.

The build system also supports choosing a target architecture. The Mimiker OS can be built on two architectures: MIPS32 and ARM64. The latter, however, has been recently added to the kernel and, as of now, misses a few functionalities. Therefore, KCSAN is turned on only for MIPS32, but porting it to another architecture will not be difficult.

All memory access within the kernel is instrumented except in a few cases. The KCSAN run-time library cannot be instrumented, as it would lead to an undesirable recursion. Code that is run at the boot time and during the kernel initialization is also excluded from the sanitization. It is executed on a single thread, so there is no point in detecting data races at that moment anyway.

Run-time Library

The flowchart of the implementation of KCSAN is presented in Figure 4.1. We will briefly describe the implementation of each step of the algorithm.

The entry point to the run-time library (a point where the execution of KCSAN's algorithms begins) is in instrumentation functions. The compiler puts a special function call in front of every memory access. These functions are in the form `__tsan_{read, write}N(address)`, where `N` denotes the size of the accessed variable. One may notice that they are the same as in the KTSAN. The compiler is indifferent to how the instrumentation is used. It just puts function calls in proper places. It is the run-time library's job is to provide the implementation of these functions.

Each memory access instruction is augmented with a function call that depends not only on the size but also on the type of access. Operations on atomic variables are ignored, as they are not considered harmful. There is a global lock that is acquired when the run-time library is entered. This allows us to avoid data races in the library itself.

The next step is to find a watchpoint corresponding to the accessed address. Active watchpoints, which are encoded in one machine word each, are stored in the global hash table with open addressing. A watchpoint only holds information about the memory access: address, size, and a flag denoting whether it was read or write. Note that the address of memory access has the same length as the machine

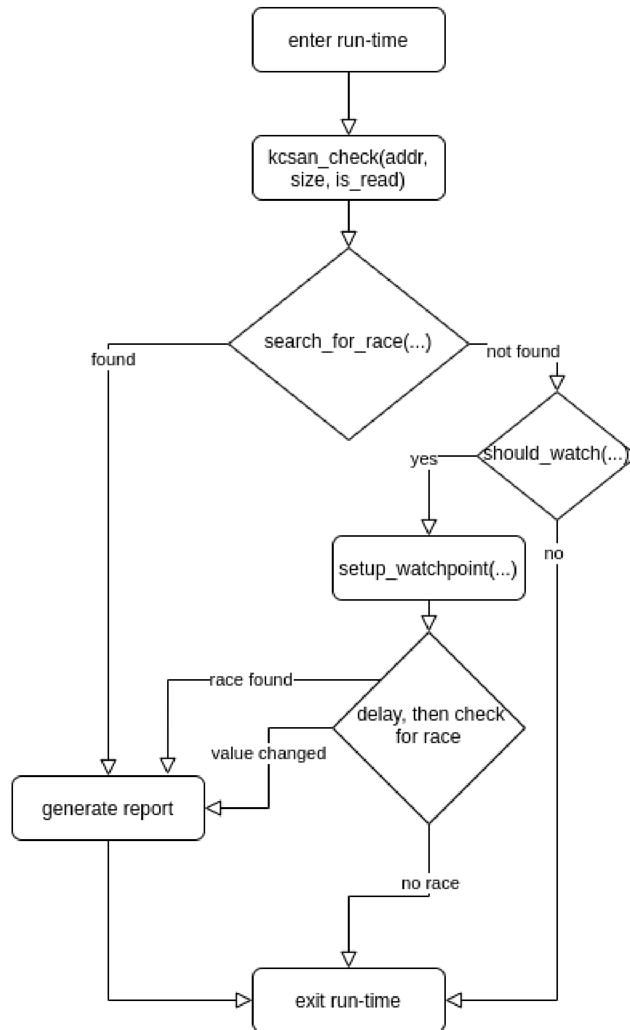

Figure 4.1: Flowchart of the KCSAN implementation

word. Therefore, to fit extra data, compression has to be performed. In the MIPS architecture, addresses in the kernel space always have the three most significant bits set. Therefore, there are free 3 bits left for use – 2 bits for the logarithm of the size of access and 1 bit for the aforementioned flag.

If the corresponding watchpoint is found, an error about the data race is generated. In the other case, the algorithm decides whether a new watchpoint should be created. The detection efficiency of the KCSAN is highly dependent on the way how the variables are chosen to be watched. Some analysis could be performed to watch only those that are vulnerable to data races. However, such an analysis would be computationally expensive and would significantly slow down the program. Note that it would have to be run on every memory access. Therefore, a much simpler strategy is used – every n-th memory access is watched. In the Mimiker OS, n is equal to 500 and was chosen experimentally to create an optimal performance-precision balance.

After the watchpoint is created, the thread waits for a fixed amount of time. It is done by yielding – deprioritizing the current running CPU task and allowing another task to run. In the Mimiker OS, there is a function named `thread_yield()` that instructs the scheduler to switch voluntarily to another thread. It is important to ensure that preemption is enabled before the watchpoint is created. Otherwise, the waiting would be futile, as other threads would be prevented from ever getting the CPU (Mimiker runs on one physical CPU).

The watched variable could be changed while the thread was waiting. It could be done by an uninstrumented access or by a DMA transfer. In such a case, we do not know the other thread involved in the data race, but it does not stop us from reporting it as an error of a different kind.

4.2.2 Example Usage and Bug Report

Running the Mimiker OS with the Kernel Concurrency Sanitizer does not require any additional effort. If everything is already set up (toolchain, required packages, etc.), building the kernel with KCSAN enabled is as simple as running `make KCSAN=1` (instead of `make`) command.

To see how KCSAN reports data races, I have added a kernel unit test (shown in Listing 6) that contains a model example of a data race. Two concurrent threads are increasing the same variable without any synchronization.

```
static int buggy_variable;

static void buggy_thread(void *p) {
  for (int i = 0; i < 10000; i++) {
    buggy_variable++;
  }
}

static void buggy_thread2(void *p) {
  for (int i = 0; i < 10000; i++) {
    buggy_variable++;
  }
}
```

Listing 6: Example code containing a data race.

Just after the test suite is launched, an error is reported to the system log. The message shown in Listing 7 is simplified for the sake of brevity.

```
| ===========KernelConcurrencySanitizer===========             |
| * found data race on the variable 0xc01da07c                 |
| * write of size 4                                            |
| * you can find the second thread using gdb                   |
| ============================================                 |
```

Listing 7: Example KCSAN report.

The message alone is not very informative, since it gives almost no information about the location where the data race happened. However, the test suite can be launched with the debugger, which can be used to see the exact location of the error. The debugger, `gdb`, automatically pauses the execution of the kernel after an error message is reported.

Listing 8 shows a result of running `backtrace` command within the debugger. The backtrace (also called stack trace) is a report of the active stack frames of threads at the moment of the data race. We can read that `test-thread-race-3/7` made an access to variable `buggy_variable` in `buggy_thread2` function (frame 9, file `sys/tests/thread_race.c`, line 21). It can be also seen that another thread attempted to make a write to the same variable in function `buggy_thread` (frame 4, file `sys/tests/thread_race.c`, line 15).

```
>>> backtrace for thread{test-thread-race-1/5}
...
#2  0xc010d628 in kcsan_check () at sys/kern/kcsan.c:193
#3  0xc010d718 in __tsan_write4 (ptr=ptr@entry=0xc01da07c
↪   <buggy_variable>) at sys/kern/kcsan.c:212
#4  0xc01687bc in buggy_thread () at sys/tests/thread_race.c:15
#5  0xc01289f0 in thread_self () at sys/kern/thread.c:140

>>> backtrace for thread{test-thread-race-3/7}
...
#5  0xc010d2bc in delay () at sys/kern/kcsan.c:69
#6  setup_watchpoint () at sys/kern/kcsan.c:160
#7  0xc010d5a8 in kcsan_check () at sys/kern/kcsan.c:191
#8  0xc010d6f0 in __tsan_read4 (ptr=ptr@entry=0xc01da07c
↪   <buggy_variable>) at sys/kern/kcsan.c:212
#9  0xc01687f4 in buggy_thread2 () at sys/tests/thread_race.c:21
#10 0xc01289f0 in thread_self () at sys/kern/thread.c:140
```

Listing 8: Example `gdb` backtrace (simplified) containing the error's location.

This data race reporting differs from the one in the original KCSAN imple-

mentation in the Linux kernel [29]. They combine all useful information (data race location, backtraces, etc.) in one report that is printed to a user. This method is better for a large project which is run in many environments by thousands of developers. It would be inconvenient to run the debugger for each test run because it would take too much time to manually debug every data race.

However, in the Mimiker OS, each developer uses the same standardized development environment. The number of reports is also smaller by several orders of magnitude. Thus, it is not a problem to occasionally use the debugger after a data race is reported. The biggest advantage of this approach is that our implementation of KCSAN does not contain superfluous code preparing extensive reports.

4.2.3 Results and Future Improvements

Overhead

The slow-down is highly dependent on the parameters used in the KCSAN. The bigger the waiting time is, the kernel will run slower. One can also adjust the number of memory operations that should be skipped before another watchpoint is created. A smaller value of this parameter results in more thorough race detection. On the other hand, a larger value improves the system performance, but more races are likely to be missed.

The implementation of dedicated benchmarking tools for Mimiker is still a work in progress, so there is no reliable method of measuring the performance overhead of KCSAN. However, the test suite consists of over 100 tests, and launching them can be used to approximate the overall system performance.

The parameters were chosen so that the kernel is not slowed down to an extent that it is unusable. The average time of running all tests is 12 seconds without KCSAN and 39 seconds with KCSAN enabled, which gives a 325% slowdown. The memory overhead is negligible, as the run-time library stores only one hash table and a few variables.

Found bugs

Here is a list of bugs that the Kernel Concurrency Sanitizer has reported so far:

- Unprotected access to a shared variable in the kernel log subsystem[30].
- Data race in the virtual memory subsystem[33].

Future improvements

Currently, there is no support for running the Mimiker's test suite concurrently. Tests are sequentially run one at a time, so a data race cannot occur, and thus KCSAN will not find essential bugs. However, concurrency occurs in a few situations, such as when a new process is forking to another one. To make KCSAN more useful two things need to be done. Firstly, the testing infrastructure should be changed to allow running multiple tests at the same time. Secondly, new tests should be added to increase the code coverage. The goal is to have many threads concurrently executing different parts of the kernel.

4.3 Lock Dependency Correctness Validator

4.3.1 Implementation Details

The implementation I have created as part of my work is significantly simpler and shorter than the original implementation in the Linux kernel. It mainly comes from the fact that the Mimiker OS does not have such an enormous codebase as Linux has, so there are not that many special cases that need to be taken care of. All the work done has already been merged into the mainline of the Mimiker OS. All referenced files and the code can be found online at Mimiker's GitHub repository [23].

Modifications in the Existing Code

The validator is architecture-independent. In other terms, it does not use specific properties of the architecture that the kernel is running on (in contrast to KCSAN). Lockdep can be enabled at compile-time by setting a specific flag in the Mimiker's build system. Therefore, the kernel's source code is left unmodified if the validator is turned off.

Lock classes are implemented by assigning a specific key to every lock in the system. For each lock that is declared statically (for example, `all_proc_mtx`, which is used to protect the list of all processes in the system), its key is equal to its address in the memory. Locks that are allocated dynamically cannot be tracked that way. Each lock allocated in this way will have a different address, but all locks associated with a specific structure should have the same key so that they could be recognized as one lock class. A solution to this problem is discussed below.

Listing 9 shows a structure that is added to every lock object. It is used as a generic representation of a lock type. This structure is initialized when a new instance of a lock is created. The lock key uniquely identifies a class, so storing the pointer to the class is not necessary. It is used, however, to speed up the process

of finding the class for a given key. Once the class is found it does not have to be searched for again.

```
typedef struct lock_class_mapping {
  lock_class_key_t *key;
  const char *name;
  lock_class_t *lock_class;
} lock_class_mapping_t;
```

Listing 9: A `lock_class_mapping_t` structure.

Keys for dynamically allocated locks are assigned by redefining the initialization functions of the synchronization primitives. Listing 10 shows a macro that is used to initialize a mutex. For each place in the code where the mutex is going to be initialized, this macro creates a static variable[1] (`__key`). Then the address of that static variable is used as the key identifying the type of the lock. The key observation is that locks initialized in the same place in the code tend to follow the same locking rules.

```
#define mtx_init(lock, attr)                    \
  {                                             \
    static lock_class_key_t __key;              \
    _mtx_init(lock, attr, #lock, &__key);       \
  }
```

Listing 10: Overridden function initializing a mutex.

There is the validator's interface – a set of public functions that the kernel can call to interact with lockdep. It basically contains a function to initialize lockdep's internal data structures and two generic functions called on every locking event:

```
void lockdep_init(void);
void lockdep_acquire(lock_class_mapping_t *lock);
void lockdep_release(lock_class_mapping_t *lock);
```

Listing 11: The validator's interface, a fragment of `include/sys/lockdep.h`.

The other parts of the kernel use only `lockdep_acquire/lockdep_release` function to inform the validator about locking events. They are called just after/before a lock is acquired/released (as shown in Listing 12). It is important to

[1] Static variables in the C language can be placed anywhere in the code and they retain the value even after they exit the scope. They point to the same memory address and they are initialized once at the first access.

underline the fact that these functions are only used in the code of synchronization primitives. Lockdep remains transparent to actual lock users.

```
void _mtx_lock(mtx_t *m, const void *waitpt) {
   ...
#if LOCKDEP
  lockdep_acquire(&m->m_lockmap);
#endif

  ...
}
```

Listing 12: A fragment of _mtx_lock function, from file include/sys/mutex.h, that acquires the mutex.

Note that for the sake of brevity, we only present the modifications done to the implementation of mutexes. In the Mimiker OS, there is another lock type – spinlocks[2], which had to be changed similarly. Introducing lockdep's support for a new synchronization primitive will only require adding annotations inside some of the primitive's functions.

Run-time Library

The main component of the run-time library is lock classes. They are registered on the first acquisition of a lock of a particular lock class. Then all following instances with the same key will be mapped to the same class. Lock classes are stored in a hash table and a hash code is calculated from the lock key.

Each lock class represents a node in the lock dependency graph. For each node, we store the outgoing edges, so in terms of locks, it means that we keep a list of lock classes that were acquired after the lock class associated with the graph node (in section 3.4 we called this list *after*). A cycle in the dependency graph indicates a violation of the lock order, so we run a breadth-first search algorithm to verify the lock ordering. If the algorithm visits any vertex twice (and thus finds a cycle), an error is reported.

Each thread maintains a stack of currently held locks. When a lock is acquired, we add it to the *after* list of the top element of the stack. If the list did not contain that lock class before, we run the graph traversal algorithm to verify whether the newly added edge created a cycle in the lock dependency graph. When a lock is released, we simply remove it from the stack of held locks.

[2] A spinlock is a type of lock that causes a thread trying to acquire it actively wait and *spin* in the loop until the lock is available.

If the validator finds out that the lock ordering is violated, the kernel execution is stopped and the report about the error is displayed. The message contains short information about lock classes involved in a potential deadlock.

4.3.2 Usage and Results

Lockdep can be enabled by building the kernel with the `LOCKDEP` flag, that is running `make LOCKDEP=1` command in the source code directory. When enabled, the validator will automatically track the usage of synchronization primitives and inform about potential deadlocks.

Currently, there are exactly 40 lock classes in the Mimiker OS. The validator searches for a cycle in the dependency graph every time an edge is added. Therefore, we can limit the number of runs of the graph traversal algorithm to roughly 1600. This amortized performance overhead is negligible. Memory consumed by lockdep is also insignificant. For these reasons, the validator is enabled by default when the test suite is run.

Figure 4.2 shows a small part of the lock dependency graph. Nodes are labeled with the names of lock classes. Vertices without edges are omitted in this figure for the sake of brevity, so this subgraph is not a representative sample of the whole graph. The average degree of a vertex in the original graph is smaller.

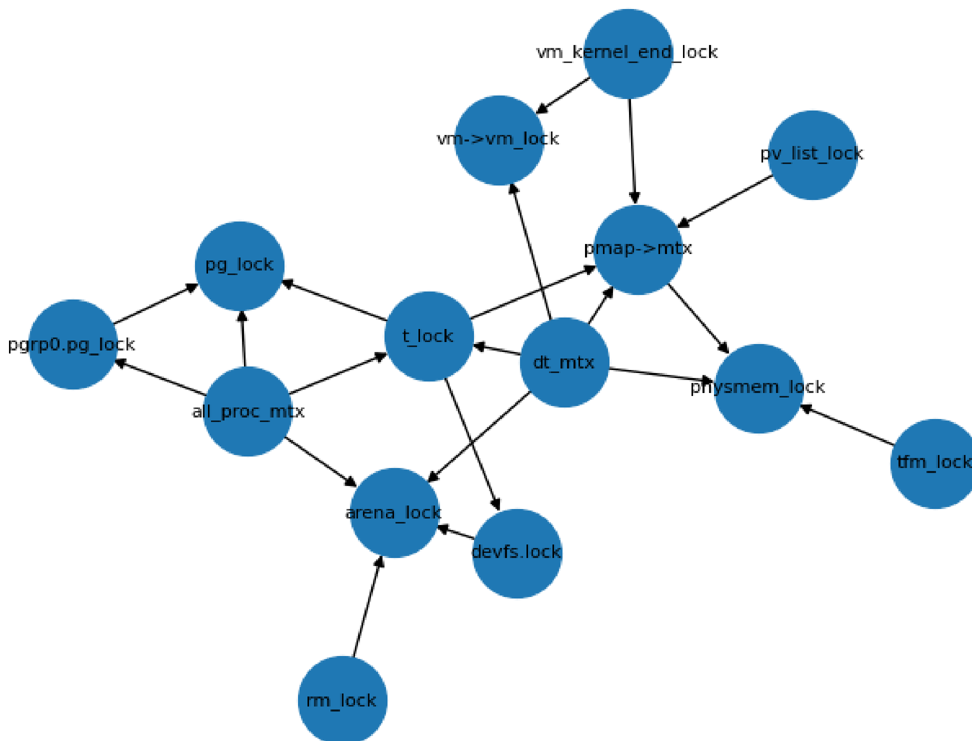

Figure 4.2: A part of the lock dependency graph.

Lockdep has already found two bugs [31][32]. In each of them, the ordering between a pair of locks was violated.

To find more bugs, more locking scenarios must be considered. This can be achieved by adding new tests to increase branch coverage[3]. In contrast to KCSAN, running the same test suite more than once will not yield more bug reports. The lock dependency graph will remain the same no matter in what order the tests are run.

[3]Branch coverage is a testing method, where we ensure that each of the possible branches of the program is executed at least once.

Chapter 5

Conclusions

Concurrent systems are hard to design, mainly because of the difficulties of detecting and fixing concurrency bugs. The growing software complexity of operating system kernels keeps exacerbating the situation. Errors can have different consequences ranging from harmless inaccuracies in calculations to severe security threats.

In this thesis, I have briefly discussed concurrency bugs: why they occur, what can be done about them, and how they differ in kernels. I have also described a few methods and tools for detecting concurrency errors. As part of this thesis, I have implemented the KCSAN and lockdep in the Mimiker OS. It has already proved to be valuable by finding a few errors. Since the tools are integrated into Mimiker's workflow, more bugs are expected to be found in the future. Each new code change is required to pass all tests with the KCSAN and lockdep enabled.

5.1 Contributions

Here is a detailed summary of my code contributions to the Mimiker OS, as part of the implementation efforts described in this thesis:

- Rewriting the toolchain[1] build system – the code instrumentation used in the KCSAN is a new feature and compilers have only recently added support for it (GCC 11.1 was released on 27th April 2021). Therefore, we had to update Mimiker's toolchain. It turned out that the toolchain build system (mainly a set of scripts used to build the toolchain) was so obsolete that I had to rewrite it completely.

- Adapting the kernel to the code instrumentation – enabling the instrumentation in the whole kernel resulted in a crash at boot-time. Some parts of the code

[1] A toolchain is the set of tools that compiles source code into executable. It includes a compiler, a linker, and run-time libraries. The toolchain is configured and built to work with a specific platform and CPU architecture.

have to be excluded from the instrumentation, i.e., the code of the KCSAN itself (otherwise it would result in an unbounded recursion), the scheduler, the early stages of the boot process, and many smaller functions.

- Implementation of the KCSAN – mostly contained in the file `sys/kern/kcsan.c`. Also, some small modifications in the build system have to be made.
- Implementation of lockdep – the main algorithm is in the file `sys/kern/lockdep.c`. I have also added annotations in the synchronization primitives used in the kernel.
- Fixing found bugs – many bugs were discovered during the work, not only those found by the detectors. To name a few[2]: `#958`, `#969`, `#957`.

5.2 Future Work

Future work mainly includes enhancing Mimiker's test suite so that it tests different parts of the kernel concurrently. However, it is not as simple as running multiple tests at a time. First of all, some parts of the kernel, such as the virtual file system, contain many concurrency bugs. It is not like that there are a few data races – some locking strategies have serious shortcomings in their design. Therefore, there is a need to correctly rewrite several subsystems with concurrency in mind.

The testing infrastructure itself should be improved as well. Currently, the Mimiker OS lacks thorough tests to many subsystems, so this would be a good place to start. Afterward, support for running multiple tests at a time could be added. Additionally, a fuzzer with other bug detectors can be added to perform automated testing.

[2]These are the numbers denoting pull requests that fixed particular bugs. All pull requests can be viewed here `https://github.com/cahirwpz/mimiker/pull`.

Bibliography

[1] Operating Systems: Internals and Design Principles, William Stallings, Pearson Education, 2015.

[2] Operating System Concepts Essentials, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, John Wiley & Sons, Inc. 2011.

[3] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers, C-28(9):690–691, September 1979.

[4] Memory Consistency Models for Shared-Memory Multiprocessors. Kourosh Gharachorloo, Stanford University, December 1995.

[5] Linux Device Drivers. 3rd Edition. Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman. O'Reilly Media, 2005.

[6] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In WBIA '09, pages 62–71. ACM, 2009.

[7] Leslie Lamport. Time, Clocks and the Ordering of Events in a Distributed System. Communications of the ACM 21, 7 (July 1978), 558-565. Reprinted in several collections, including Distributed Computing: Concepts and Implementations, McEntire et al., ed. IEEE Press, 1984.

[8] LWN (Linux Weekly New), An introduction to lockless algorithms, `https://lwn.net/Articles/844224/`

[9] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, pages 89–100, New York, NY, USA, 2007. ACM

[10] x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. Communications of the ACM, 53(7):89–97, 2010.

[11] A Tutorial Introduction to the ARM and POWER Relaxed Memory Models. Luc Maranget, Susmit Sarkar, and Peter Sewell. Technical report, October 2012.

[12] Memory Models for C/C++ Programmers. Manuel Pöter, Jesper Larsson Träff. arXiv:1803.04432v1 [cs.DC].

[13] Eraser: A Dynamic Data Race Detector for Multithreaded Programs, Michael Burrows, Greg Nelson, ACM Transactions on Computer Systems, Vol. 15, No. 4, November 1997.

[14] Pin: Building Customized Program Analysis Toolswith Dynamic Instrumentation. Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, Kim Hazelwood, Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. - PLDI '05. - New York, NY, USA: ACM, 2005. - P. 190-200.

[15] Integrating the Kernel Address Sanitizer into the Mimiker Operating System (Master's Thesis), Julian Pszczołowski, University of Wrocław, 2020.

[16] Andrey Konovalov. "KernelThreadSanitizer (KTSAN): a data race detector for the Linux kernel", 2015, `github.com/google/ktsan/wiki#implementation`.

[17] Kernel Thread Sanitizer (KTSAN), `https://github.com/google/ktsan/wiki`

[18] Thread Sanitizer Documentation, `https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual`

[19] FreeBSD 13.0 Kernel Developer's Manual, locking(9)

[20] Marco Elver, Paul E. McKenney, Dmitry Vyukov, Andrey Konovalov, Alexander Potapenko, Kostya Serebryany, (...), and Luc Maranget. Concurrency bugs should fear the big bad data-race detector. Linux Weekly News (LWN), 2020.

[21] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective Data-Race Detection for the Kernel, OSDI 2010.

[22] The Mimiker Project, `https://mimiker.ii.uni.wroc.pl`

[23] The Mimiker's GitHub repository, `https://github.com/cahirwpz/mimiker`.

[24] google/ktsan repository, Upstream Fixes of Data Races found by KCSAN, `https://github.com/google/ktsan/wiki/KCSAN`

[25] The Kernel Concurrency Sanitizer (KCSAN) – The Linux Kernel documentation, `https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html`

[26] Runtime locking correctness validator – The Linux Kernel documentation, `https://www.kernel.org/doc/html/v5.6/locking/lockdep-design.html`

[27] American fuzzy loop, `https://lcamtuf.coredump.cx/afl/`

[28] LWN (Linux Weekly New), The kernel lock validator, `https://lwn.net/Articles/185666/`

[29] The Linux kernel 5.12 source code, file `kernel/kcsan/core.c`, https://elixir.bootlin.com/linux/v5.12/source/kernel/kcsan/core.c

[30] cahirwpz/mimiker repository pull request #957: *klog: fix race*, https://github.com/cahirwpz/mimiker/pull/957

[31] cahirwpz/mimiker repository issue #1027: *No ordering between locks p_lock and t_lock*, https://github.com/cahirwpz/mimiker/issues/1027

[32] cahirwpz/mimiker repository issue #968: *Deadlock between physmem_lock and pv_list_lock*, https://github.com/cahirwpz/mimiker/issues/968

[33] cahirwpz/mimiker repository issue #1156: *Data race on vm_page_t::flags*, https://github.com/cahirwpz/mimiker/issues/1156