Bull. Nov. Comp. Center, Comp. Science, 23 (2005), 57–70

OS-independent detection of thread switches on uniprocessor

A.V. Mogilev

Abstract. Parallel programs often are non-deterministic in their nature, what greatly complicates testing, debugging, verifying and analyzing such programs. On a uniprocessor, interleaving actions of the system scheduler (thread switches) can be thought of as source of nondeterminism. The precise detection of these actions helps many tasks, especially the *schedule-based* execution replay — the technique for robust debugging of non-deterministic programs. Known solutions to detection of thread switches are either specific to system schedulers providing this information, or require modification of the scheduler. For the system scheduler in the OS Windows, all these solutions are inapplicable.

The paper presents an OS-independent algorithm for detection of thread switches in a multithreaded program running on a uniprocessor based on the primitive thread operations such as suspending a thread, accessing its context, etc. The correctness of the algorithm is proven on the low-level abstract model of multithreaded programs that is also presented in the article.

1. Introduction

The most common definition of program *determinism* is as follows: *a program is deterministic iff its result depends only on its input data, otherwise the program is non-deterministic.* However, this definition does not determine what is the program result and input data, as well as what is the program itself. So, the exact definition is only possible when the exact model of the program execution is specified. For real-world programs there are several interpretations of this definition, including *internal* and *external* non-determinism [10], etc.

No matter which definition is used, the problem of debugging is much harder for non-deterministic programs than for deterministic ones.

The problem of debugging of sequential deterministic programs is well studied — there are several scientific debugging methodologies [1], including *cyclic debugging*, *reverse execution*, etc.

However, adopting these techniques to concurrent non-deterministic programs is a big challenge. One of the problems is the requirement for a simple model of execution of a concurrent program that should be close to the real execution of the program on a real computer. Another major problem is instability of execution of non-deterministic programs: a repeated execution with the same input data can differ from the first one, so all information

collected during previous program executions becomes unreliable after the program restart.

As a solution to the problem, the technique of *execution replay* was proposed [5]. The first execution of a program is *traced*, and the collected information is reused during subsequent program runs to enforce exactly the same executions. In other words, a new deterministic version of a program is created that always repeats the traced execution, so called *replaying version*. Therefore, all techniques for debugging of deterministic programs can be applied to the replaying version of the program. Indeed, if a bug manifested itself in the traced execution, it will do so in all executions of the replaying version of the program, and all information collected during one of its executions remains valid in all subsequent executions.

The main problem why this technique is not widely used yet is that the amount of the information that should be traced is excessively huge in the common case. Collecting this information during the program execution may significantly affect this execution, so the behaviors of the traced and original programs can differ significantly.

However, for multithreaded programs executed on a uniprocessor the amount of the required information is small enough — it is sufficient to trace only the thread switches. This technique is used in so-called *schedule-based* replay systems [11].

Note that the problem of detection of thread switches should be solved for such systems. There are several different solutions known, but all of them utilize specific features of particular operating systems. Russinovich and Cogswell modified the kernel of Mach 3.0 operating system [11], adding the tracing code to the system scheduler. As a result, the system scheduler invokes user-defined callbacks whenever the switch occurred. In the same way, in the DejaVu replay system [3] for Jalapeño [2] Java virtual machine, the scheduler of the virtual machine is modified. The replay system described in [13] uses existing switch detection functionality provided by the kernel of VxWorks operating system.

As a result, these solutions can not be reused for other OSes, for example for OS Windows. Windows 2000 contained undocumented functions (`KeSetThreadSelectNotifyRoutine`, `KeSetSwapContextNotifyRoutine`) providing similar functionality for code of kernel-mode drivers, but they were removed from the later Windows versions.

An OS-independent algorithm presented in this paper does not modify the system scheduler, nor it relies that the scheduler provides such functionality. The algorithm detects the thread switches on a uniprocessor with the help of only the commonly available thread operations: suspending and restoring a thread, accessing thread contexts, synchronizing with critical sections.

The rest of the paper is organized as follows. Section 2 presents a model of a multithreaded program used in the algorithm description and in correctness theorems. Section 3 defines the notions of program instrumentations used later to describe the algorithm in Section 4. Next, Section 5 contains the correctness theorems and their proofs. Section 6 describes the implementation of this algorithm for OS Windows and some practical results, and, finally, Section 7 concludes the paper.

2. Abstract model of a multithreaded program

This section presents a low-level abstract model of execution of a shared-code shared-memory parallel program in a well-known form of a transition system. It is designed close to the real execution of the program on a computer, so representation of the real program in this abstract form is quite straightforward. Unlike many popular models of parallel computing like Petri nets [9], CCS [6], timed automata [4] etc., this model does not require a program to be described in a specific model language. Representation of an existing multithreaded program written in a C-like imperative language is straightforward.

Let us define a memory M of a program P to be a finite set of cells whose values are from some common domain $\mathbb{D}$, and registers R to be a finite set of cells, also each within the domain. The memory is shared between all N_T threads (referred below by thread indexes: $t \in Threads = [1, N_T]$), while the values of registers are local to threads. Let us define the states of the memory and registers σ_M and σ_R as interpretations that give values for each cell in the memory and registers, correspondingly: $\sigma_M(m_i) \in \mathbb{D}$, $\sigma_R(r_i) \in \mathbb{D}$. Σ_M and Σ_R denote the sets of all possible states of memory and registers: $\Sigma_M = \{\sigma_M\} = \mathbb{D}^{N_M}$ and $\Sigma_R = \{\sigma_R\} = \mathbb{D}^{N_R}$.

The program code consists of atomic instructions placed at locations from the set $LOC = \{l_1, ..., l_{N_L}\}$. For the sake of simplicity, we define a special location l_{finish} common for all programs and suppose that a thread is finished when it reaches this location.

An activity state $a_t \in \mathbb{B}$ of a thread t determines whether instructions can be executed in the context of the thread. An activity state of the program $\bar{a} \in \Sigma_A = \mathbb{B}^{N_T}$ is formed from activity states of each thread. We define a *context* c_i of the thread T_i as a union of states of the registers and current location of the thread: $c_i \in C = \Sigma_R \times LOC$. The activity state, memory state, and contexts of each thread forms the global state of the program:

$$\sigma \in \Sigma = \Sigma_A \times \Sigma_M \times C^{N_T}.$$

In general, each instruction is a function on states of a program that represents one transition of the system. Each instruction may be executed in any thread. We denote the thread, where an instruction was executed, and

the context of this thread as *current thread* and *current context*, respectively. It is convenient to separate instructions into 3 kinds:

1) *computational* instructions $\alpha : \Sigma_M \times C \to \Sigma_M \times C$ that, given the current context and the state of the memory, produces new values for them;
2) *synchronizing* instructions $\beta : \Sigma_M \times \Sigma_A \times C \to \Sigma_M \times \Sigma_A \times C$ that additionally may change the activity states of all threads; and
3) *special* instructions $\gamma : \Sigma \to \Sigma$ - all other instructions (with no restrictions)

Below we suppose that an original program does not contain instructions of the third kind. However, such instructions are needed to represent our algorithm. Note that the execution of each instruction is deterministic and depends only on the preceding state; the only source of non-determinism is the order of execution.

At last we can define a transition system as a triple $Trans(P) =< \Sigma, \overset{P}{\to}, \Theta >$, where Σ is a set of states defined above, $\Theta \subset \Sigma$ is a subset of initial states, and $\overset{P}{\to}$ is a relation on states, constructed as follows: let $\sigma =< \bar{a}, \sigma_M, \bar{c} >$ and $\sigma_1 =< \bar{a}_1, \sigma_{M1}, \bar{c}_1 >$. Then,

$$\sigma \overset{P}{\to} \sigma_1 \tag{1}$$

iff

$$\exists i \; a_i = True, \; instr(l_i)(\sigma) = \sigma_1, \tag{2}$$

where $l_i = loc_i(\sigma)$ is the current location of the thread i in the state σ, and $instr(l_i)$ denotes an instruction in that location.

An execution e of the transition system is a sequence $\sigma_0\sigma_1...\sigma_N$, where $\sigma_0 \in \Theta$, $\forall i \; \sigma_i \overset{P}{\to} \sigma_{i+1}$ and σ_N is *terminating*, i.e. current locations of all threads are l_{finish} for this state. Never-terminating executions are not considered in this paper for the sake of simplicity.

In the rest of the paper, the term 'state σ' is used to denote not only a tuple of the values of memory, activity state and contexts, but also a position in an execution e. The only exception is comparison of two states for equality — in this case only the values are compared.

Given a transition in (1) constructed with i from (2), we denote $CT(\sigma) \overset{def}{=} i$ and $CL(\sigma) \overset{def}{=} l_i$ for the current thread and location of the instruction whose execution corresponds to the transition. Note, that theoretically it is possible that several appropriate i values exist, but in this case $\sigma = \sigma_1$ and such transitions can be safely removed from the execution e.

A transition system may have many possible executions starting from the same state, because one of N_A transitions can be chosen from a state σ, where N_A is the number of currently active threads. A program is deterministic if $\forall\sigma_0 \in \Theta$ there is at most one execution starting with σ_0 state.

The set of all possible executions e forms the language of the given program $L(P)$.

Given an execution e we may determine the points of *thread switches*: a switch to a thread t is occurred at the i-th step ($0<i<N$) iff $CT(\sigma_{i-1}) \neq t$ and $CT(\sigma_i) = t$.

3. Instrumentation of a program

Instrumentation of a program is its modification that adds new functionality.

Formally, let us call a program P' to be an instrumentation of a program P, if its memory, code and set of threads are the extensions of the entities of the original program: $M' = M \cup \Delta M$, $LOC' = LOC \cup \Delta LOC$, $Threads' = Threads \cup \Delta Threads$. It is supposed that no instructions of the original program ($l \in LOC - - - \{l_{finish}\}$) can be executed in new threads. Below we refer to newly added entities as *instrumentation locations and instructions*, *instrumentation memory* and *instrumentation threads*, in contrast to *original locations, memory*, etc.

Consider an execution e' of an instrumented program and a single state

$$\sigma' =<< a_1, ..., a_{N'_T} >, \sigma_{M'}, < c'_1, ..., c'_{N'_T} >>$$

in this execution. A truncation of the execution of the instrumented program $Trunc(e')$ is the corresponding execution of the original program that is constructed in 3 steps.

1. All instrumentation threads are removed from the activity state and from the set of contexts, instrumentation variables are removed from the memory states:

   $$Tr_1(\sigma) \stackrel{def}{=} << a_1, ..., a_{N_T} >, \sigma_M, < c'_1, ..., c'_{N_T} >>,$$

   where N_T is the number of threads and M is the memory in the original program P.

2. If, in the state σ', the current location of a thread i is the instrumentation location, then the context c'_i is replaced by the context $c_i(\sigma'_1)$ of the nearest next (in the execution e') state σ'_1 such that $loc_i(\sigma'_1) \in LOC$,

i.e. the next original instruction executed in the thread i is used[1]. Let us denote the found context c_i:

$$Tr_2(\sigma) \stackrel{def}{=} < ..., < c_1, ..., c_{N_T} >> .$$

3. If, in the sequence of the states resulting from step 2, a state is equal to the preceding one, then it is removed:

$$Trunc(\sigma_0\sigma_1...\sigma_N) \stackrel{def}{=} Tr_2(\sigma_{i_0})Tr_2(\sigma_{i_1})...Tr_2(\sigma_{i_K}).$$

An instrumentation is *correct*, if the language formed by truncations of all possible executions of the instrumented program is equal to the language of the original program: $Trunc(L(P')) = L(P)$. In other words, all possible executions are still possible after the instrumentation, and no new original executions become possible.

Note, however, that the model does not determine the probability of choosing a given execution e among other possible executions starting from the same initial state. So, the correctness of the instrumentation does not imply its *accuracy* that can be defined as the approximate equality of probabilities for choosing the execution e in the original program and any of $\{e' \mid Trunc(e') = e\}$ in the instrumented program. Suppose that in a real program there are two possible executions and the first one is almost always taken. If, after the instrumentation, this execution will almost never be taken, then this instrumentation is definitely inaccurate and useless, although it may be correct.

4. Algorithm

The basic idea of the algorithm is the use of the induction principle for the detection.

Consider an imaginary instruction *guard* that registers a thread switch (from a known previous thread to the current one), inserts another `guard` instruction just prior to the current location of the previous thread and changes the current location of the previous thread to point to that `guard`. Let us define that a thread is in the *detection-aware state* if its current location points to the `guard` instruction. This means that the switch to this thread will be registered.

If the starting locations of all threads except the first one point to `guard` instructions, then the problem of detection is solved. Indeed, consider a pair of thread switches occurred:

[1]Note that such a context always exists, because all threads finish at the predefined location $l_{finish} \in LOC$.

```
//without guard instructions:
thread 1: i1                  i4
            |               |
thread 2:     i2 - ... - i3

//with guard instructions:
thread 1: i1                        guard - i4
            |                      |
thread 2:    guard - i2 - ... - i3
```

In this case, at each moment of execution, all threads except the current one point to `guard` instructions, so all switches are detected. As can be easily proven, such instrumentation is correct, since the execution of the original instructions is affected in no way.

Of course, the problem is that there is no such a useful instruction in real instruction sets. Its functionality can be implemented by a sequence of real instructions, but such implementation is not easy, because this code sequence itself can be interrupted by the thread switches. Therefore, some non-trivial synchronization is required to keep the correctness. The proposed code sequence is presented below in the form of a C-like function with macros that can be implemented in the assembly language. Also some system functions common to most OSes are used, their exact names vary in different operating systems. We suppose that the readers of this paper are familiar with the concepts of critical sections and basic operations on threads, so no additional comments for the calls of functions `enter_critical_section`, `leave_critical_section`, `suspend_thread` and `resume_thread` are required.

Below we refer to $curThread$ as to the number of the current thread, to $prevThread$ as to the value of the variable `prevThread` defined below, and to $EIP(i)$ as to the current location of the thread i.

```
const int MAX_THREADS = ...; //some constant that limits the
                             //number of threads
int prevThread; //contains the number of the 'previous' thread
int storedEip[MAX_THREADS]; //array contains per-thread data

// starting point for all threads, also set as current EIP after
// processed thread switches
void onSwitch() {
    L0: //next location is the address of the function
    //no code is executed before saving all registers
    //registers are saved to the stack
    MACRO_save_registers;
    //the number of the current thread is obtained in
    //the implementation-defined manner
```

```
    i = MACRO_current_thread_num;
    enter_critical_section(critSection);
    L1:
    suspend_thread(prevThread);
    L2:
    //next call obtains the EIP(prevThread)
    storedEip[prevThread] = get_thread_eip(prevThread);
    //the value of EIP in context changed, so the previous thread
    //will continue execution from the onSwitch function
    set_thread_eip(prevThread, &onSwitch);
    resume_thread(prevThread);
    L3:
    //registers the switch to this thread in user-defined way
    registerSwitchTo(i);
    prevThread = i;
    //prepares the jump to the original EIP
    MACRO_prepare_jump(storedEip[i]);

    leave_critical_section(critSection);
    L4:
    //restore all registers from the stack
    MACRO_restore_registers;
    //jumps to the prepared address, uses no registers
    MACRO_jump;
    L5:
}
```

In terms of this algorithm, a thread i is in a *detection-aware state* iff $EIP(i) \in [L0, L2)$.

The instrumentation required by the algorithm is described below:

- The cells corresponding to global variables `prevThread` and `storedEip` are added to the memory;
- The function `onSwitch()` is added to the code;
- The initialization code for new variables is added to the starting code of the first thread:

```
void init() {
    int i;
    for(i = 0; i < MAX_THREADS; i++) {
        //in our model, the starting locations of all threads
        //are known. In real implementation it may be obtained
        //dynamically at the moment of the thread creation.
        storedEip[i] = MACRO_start_of_thread_i;
    }
    prevThread = 0;
}
```

- The starting locations of all threads except the initial one are changed to `&onSwitch()`.

Informally, the synchronization can be explained as follows. The critical section `critSection` helps if execution of the `onSwitch()` code is interrupted by the switch to any thread in the detection-aware state. But the critical section cannot help if the switch to the previous thread happens before its context is modified. That is why additional synchronization with the `suspend/resume` functions is needed.

5. Correctness theorems

The section contains the proofs of two theorems that the instrumentation is correct and that all required switches are detected. The proofs use the following lemmas.

Lemma 1. *At any moment of execution either of these four conditions is true:*

1) $curThread$=$prevThread$; $EIP(curThread) \notin [L0, L4)$*; all other non-finished threads have* $EIP \in [L0, L1)$*;*

2) $curThread \neq prevThread$; $EIP[prevThread] \notin [L0, L4)$*; among all other non-finished threads, one has* $EIP \in [L0, L2)$ *and others have* $EIP \in [L0, L1)$*;*

3) $EIP(prevThread) \notin [L0, L4)$*;* $EIP(curThread) \in [L2, L3)$*; the thread* $prevThread$ *is inactive (suspended); all other non-finished threads have* $EIP \in [L0, L1)$*;*

4) $EIP(curThread) \in [L3, L4)$*; all other non-finished threads have* $EIP \in [L0, L1)$*.*

Proof. Proof of the statement by induction is quite obvious: at the very beginning of the program, condition 1 is true, $prevThread = 0$.

The only way for condition 1 to become false is the thread switch, so condition 2 becomes true. The conditions may be thought of as the states of the finite automata representing the actions performed for a thread switch:

$$1 \leftrightarrows 2 \rightarrow 3 \rightarrow 4 \rightarrow 1 \ldots \ .$$

It is easy to see that no other transitions can occur.

There are two major corollaries of this lemma:

Lemma 2. *At any moment of execution, exactly one active thread have* $EIP \notin [L0, L2)$*.*

Proof. Direct corollary of Lemma 1.

Lemma 3. *The context of a thread i can not be changed, and the value of* `storedEip[i]` *cannot be modified from other threads while* $EIP(i) \in [L0, L4)$.

Proof. One of the presumed restrictions to the original program is the absence of instructions modifying the contexts of other threads (instructions of the third kind), so the context and `storedEip` value can be changed only by the instrumentation instructions (locations in the range $[L2, L3)$). As can be seen from the code of `onSwitch()` function, only the context of the thread `prevThread` and the `storedEip[prevThread]` value can be modified. So, we should check the case of $EIP(prevThread) \in [L0, L4) \wedge EIP(curThread) \in [L2, L3)$. Due to Lemma 1, such a case is impossible. Indeed, if either of conditions 1-3 is true, then $EIP(prevThread) \notin [L0, L4)$. If condition 4 is true, then $EIP(curThread) \notin [L2, L3)$.

Theorem 1. *The instrumentation defined in the previous section is correct.*

Proof. Note that the original memory is not affected by the instrumentation instructions, only the registers can be modified.

Part 1. Given an execution $e \in L(P)$, let us construct the execution $(e' \mid Trunc(e') = e)$ and show that $e' \in L(P')$. The simplest one can be constructed by, first, expanding the memory states to hold the values of instrumentation variables, and, second, adding the state transitions, corresponding to the straightforward execution of the initialization code, to the very beginning of the execution, and of the `onSwitch` function code to all states where thread switches are occurred. It is easy to check that the constructed execution satisfies the requirements.

Part 2. Given an execution $e' \in L(P')$, we show that $Trunc(e') \in L(P)$. Let us prove that, for any two subsequent states $\sigma_i \xrightarrow{P'} \sigma_{i+1}$ in e', either $Tr_2(\sigma_i) = Tr_2(\sigma_{i+1})$, or $Tr_2(\sigma_i) \xrightarrow{P} Tr_2(\sigma_{i+1})$. Denote $CT(\sigma_i)$ by t.

Consider two cases: $CL(\sigma_i) \in LOC$ and $CL(\sigma_i) \notin LOC$.

Let $CL(\sigma_i) \in LOC$. In this case $loc_t(\sigma_{i+1}) \in LOC$, since the original instructions in our algorithm can transfer control only to original locations. By construction of Tr_2, the context of the thread t is the only context changed between $Tr_2(\sigma_i)$ and $Tr_2(\sigma_{i+1})$, therefore execution of the instruction at the location $CL(\sigma_i)$ in the original program P would have the same results: $Tr_2(\sigma_i) \xrightarrow{P} Tr_2(\sigma_{i+1})$.

Now, let $CL(\sigma_i) \notin LOC$, i.e. one of the instrumentation instruction is executed. The contexts of the thread t found in the truncation step 2 for states σ_i and σ_{i+1} are equal, so the only non-trivial case is the execution of a special `set_thread_eip` instruction, which changes the $EIP(prevThread)$ to $L0$ (=`&onSwitch()`), while storing the original value in `storedEip[prevThread]`. Denote $prevThread$ by p. Let us prove that further execution in the thread

p results in changing the location to the stored EIP value prior to execution of any original instructions in that thread. That would mean that the contexts of the thread p found in the truncation step 2 for states σ_i and σ_{i+1} are also equal, so $Tr_2(\sigma_i) = Tr_2(\sigma_{i+1})$.

Due to Lemma 3, the context of the thread p and the value of $storedEip[p]$ can not be changed from other threads until the execution proceeds to the location $L4$. To this moment, the correct EIP value would be set for use by the corresponding `MACRO_jump` instruction. Even if the thread switches would occur before `MACRO_jump` is executed, the value used by `MACRO_jump` is not overwritten, as it is stored on the stack. The `onSwitch` can be treated as the implicitly recursive function — an implicit recursive call potentially can occur prior to execution of any instruction in the range $[L4, L5)$. As we consider only terminated executions, the call chain is finite, so the last `MACRO_jump` from the chain results in the transfer of control to the initially stored EIP.

Theorem 2. *All original thread switches are detected by the algorithm. Formally, given an execution e' of the instrumented program P', for any two consecutive executions of the original instructions occurred in different threads that are (optionally) separated only by the instrumentation instructions*

$$\forall i, j: \; CL(\sigma_i) \in LOC \wedge CL(\sigma_j) \in LOC \wedge CT(\sigma_i) = t \wedge CT(\sigma_j) \neq t \wedge (\forall i{<}k{<}j \; CL(\sigma_j) \notin LOC)),$$

there is at least one execution of the instruction `registerSwitchTo(t)` between them, where $t = CT(\sigma_j)$, and if there are several `registerSwitchTo()` executed, then the last executed registers switch to the thread t

$$\exists i{<}d{<}j \;\; (CL(\sigma_d) = l_{detect} \wedge CT(\sigma_d) = t \wedge (\forall d{<}g{<}j \; CL(\sigma_g) \neq l_{detect})),$$

where l_{detect} stands for the location of the instruction `registerSwitchTo()`.

Proof. Due to Lemma 2, at each moment of execution, all threads except one are in the detection-aware state, which means that a switch to thread t will cause execution of `onSwitch()` from some point in the range [L0, L2). Before execution of any original instruction in the context of these threads, all instructions in the range [L2, L5) should be executed, including `registerSwitchTo(t)`. From the conditions of the theorem, $CT(\sigma_i) \neq CT(\sigma_j)$, therefore at least one `registerSwitchTo(t)` is executed. All that should be proven is that, if there are several `registerSwitchTo()` executed, then the last registered switch is the switch to t. Suppose the contrary, i.e. that the last '`registerSwitchTo()`' is executed in a thread differing from t. Then, another thread switch to t shall occur. Due to Lemma 2, the thread t is in the detection-aware state at that moment, therefore another `registerSwitchTo()` is executed as well, which is impossible.

6. Implementation notes and practical results

The proposed algorithm was successfully implemented for the Windows operating system family. The main implementation notes follow.

The first implementation used the functions `GetThreadContext` and `SetThreadContext` from Windows API [8] for implementing `get_thread_eip` and `set_thread_eip`, respectively. While the implementation was correct in terms of the algorithm and instrumentation, it was not *accurate*, as these API functions imply additional thread switches, because they use the mechanism of *asynchronous procedure calls (APC)* [12]. As a result, the set of thread switches occurred in the instrumented programs differed very much from the one in the original program. However, due to these additional thread switches, this implementation was a good test for correctness of synchronization in the algorithm.

Later, the kernel-mode driver providing `get_thread_eip` and `set_thread_eip` functionality was implemented [7]. It has not only fixed the problem with additional thread switches, but also reduced the time of execution of `onSwitch` thanks to combining get/set functions into a single function `change_thread_eip` (which reduces the number of required user-mode $\leftrightarrow$ kernel-mode switches).

As a result, the execution of `onSwitch` function (measured on several typical multithreaded programs) takes $< 0.1\%$ of the total execution time. The faster `onSwitch` is executed, the lesser is the probability of the thread switch during its execution, and the more accurate the algorithm is. Currently, this probability is so small that turning off the synchronization does not result in the algorithm failure, as no thread switches actually occurred inside `onSwitch` for real programs. However, theoretically it is unsafe, as there is no minimal time in Windows that a thread is guaranteed to work without being interrupted by a thread switch.

The determination of the current thread number in `onSwitch` is implemented with the use of dynamically created per-thread functions `onSwitch<i>`:

```
// void onSwitch<i>():
    push i;        //i is a constant here
    jmp  onSwitch; //common onSwitch will take 'i' as a parameter
```

To adopt the algorithm for this implementation, the address of the corresponding `onSwitch<i>` should be used on EIP substitution for the thread i, all other argumentations remain valid.

Since, in contrast to our simplified model, the threads in real programs are created dynamically, the Windows API `CreateThread` function is intercepted, and the real starting address of the thread i is replaced with the address of `onSwitch<i>` on the fly.

7. Conclusion and future work

The paper presented the following results:

- A simple low-level formal model of multithreaded programs that is close to a real model of programs executed on a uniprocessor is presented;
- In terms of this model, the formal definition of arbitrary program instrumentation is given, along with the definition of the instrumentation correctness;
- An OS-independent algorithm for detection of thread switches on a uniprocessor is described in a form of the instrumentation of a program;
- The formal proof of the algorithm correctness is given;
- The implementation of the algorithm for OS Windows is briefly described.

Future work will be an extension of the presented model with calls to *external functions* whose code is not known and can not be instrumented, such as the system functions.

References

[1] Agrawal H. Towards Automatic Debugging of Computer Programs. — PhD thes., Purdue University, SERC, August 1991.

[2] Alpern B. et al. The Jalapeño virtual machine // IBM Systems J. — 2000. — Vol. 39, N 1. — P. 211–238.

[3] Alpern B. et al. A perturbation-free replay platform for cross-optimized multithreaded applications // Proc. of the 15th Internat. Parallel & Distributed Processing Sympos. (IPDPS-01), Los Alamitos, CA, April 23–27 2001. — IEEE Computer Society, 2001. — P. 23–23.

[4] Alur R., Dill D. L. A theory of timed automata // Theor. Comput. Sci. — 1994. — Vol. 126, N 2. — P. 183–235.

[5] LeBlanc T., Mellor-Crummey J. Debugging parallel programs with instant replay // IEEE Trans. on Computers. — 1987. — Vol. 36, N 4. — P. 471–482.

[6] Milner R. A calculus on communicating systems // Lect. Notes in Comput. Sci. — 1980. — Vol. 92.

[7] Mogilev A. A functional extension of Windows OS for monitoring streams switching // Microsoft Technologies in Informatics and Programming. — Novosibirsk, 2005. — P. 30–31 (In Russian).

[8] The Microsoft Developer Network. — http://msdn.microsoft.com/.

[9] Petri C. A. Kommunikation mit Automaten. — PhD thesis, Univ. Bonn, West Germany, 1962.

[10] Ronsse M., De Bosschere K., de Kergommeaux J. Ch. Execution replay and debugging // Proceedings of the Fourth International Workshop on Automated Debugging (AADEBUG2000) / Ed. by M. Ducasse and M. Brugge. — Munchen: TUM/IRISA, 2000. — P. 5–18.

[11] Russinovich M., Cogswell B. Replay for concurrent non-deterministic shared-memory applications // Proc. of the ACM SIGPLAN '96 Conf. on Programming Language Design and Implementation, Philadelphia, Pennsylvania, 21–24, May 1996. — P. 258–266.

[12] Solomon D. A., Russinovich M. E. Inside Microsoft Windows 2000. — Third edition, 2000.

[13] Sundmark D. et al. Replay Debugging of complex real-time systems: experiences from two industrial case studies // Proc. of the Fifth Internat. Workshop on Automated Debugging (AADEBUG 2003), November, 17, 2003.