Model-checking of a Multi-threaded Operating System*

Nicolas Marti † Reynald Affeldt ‡ Akinori Yonezawa †‡

†Department of Computer Science, University of Tokyo

‡Research Center for Information Security (RCIS), National Institute of Advanced Industrial Science and Technology (AIST)

Abstract

Model-checking has proved effective to verify low-level properties on isolated parts of operating systems such as scheduling algorithms or implementations of inter-process communications. In such situations, the relevant implementation is well-localized in the source code, and the modeling language usually lends itself very well to formal paraphrase. However, there are high-level properties of operating systems that require modeling of various parts of the implementation. For example, task isolation, the property that user threads cannot access kernel memory, requires modeling of thread management, memory management, hardware protection mechanisms, etc. In this paper, we show how to build in the Spin model-checker a model of a multi-threaded operating system (namely, Topsy) that covers most parts of the implementation, thus enabling verification of properties such as task isolation.

1 Introduction

Model-checking has proved effective to verify low-level properties on isolated parts of operating systems such as scheduling algorithms or implementations of inter-process communications (IPCs) [1–3]. In such situations, the relevant implementation is well-localized in the source code, and the modeling language usually lends itself very well to formal paraphrase.

However, there are high-level properties of operating systems that require modeling of various parts of the implementation. For example, *task isolation*, the property that user threads cannot access kernel memory [4], requires modeling of thread management, memory management, hardware protection mechanisms, etc. It is possible to break the verification of such high-level properties that span the whole source code into smaller verifications on well-localized parts of the source code. However, this approach naturally augments the number of specifications and it introduces the risk of making conflicting model assumptions. Ideally, one would prefer a single model, abstract enough to be refined at will, and that would lend itself easily to verification of several, possibly orthogonal properties.

In this paper, we show how to build in the Spin model-checker [5] a model of the Topsy operating system [6] that covers most parts of the implementation, thus enabling verification of high-level properties such as task isolation. The main difficulty of building such a global model is the trade-off between exhaustivity and tractability: too fine-grained abstractions would irremediably lead to state-space explosion. In our model, we provide abstractions to deal with several aspects of operating systems, such as scheduling, IPCs, memory management, hardware interface, and user applications. We show experimentally that these abstractions enable verifications of several, both low-level and high-level properties, such as memory protection and kernel-data consistencies for scheduling and message passing.

The operating system we deal with in this paper is Topsy v2 [6], a multi-threaded embedded operating system. Besides multi-threading, the kernel implements most classical features of general-purpose operating systems, such as memory allocation, dynamic thread creation and message-passing IPCs, making our approach reusable for other operating systems. As an embedded operating system, Topsy does not load dynamically user applications; the kernel and the user application are compiled and linked together statically (only one multi-threaded user application is supported).

This paper is organized as follows. In Sect. 2, we describe the Topsy operating system together with its Spin model. In Sect. 3, we discuss the specification and verification of several properties, including task isolation. In Sect. 4, we compare with related work,

*This work has been presented at the 23rd Workshop of the Japan Society for Software Science and Technology: `http://www.ipl.t.u-tokyo.ac.jp/jssst2006/`.

and finally conclude.

2 Model

We explain in detail our model in a modular way: multi-threading in Sect. 2.1, scheduling in Sect. 2.2, hardware in Sect. 2.3, memory in Sect. 2.4, IPCs and system calls in Sect. 2.5. For each part, we first give a general description with possibly specificities for Topsy, then we describe the model in Spin.

2.1 Multi-threading

Definitions A *thread* is a control-flow implemented by a piece of code and described by a data structure called *thread descriptor*. A thread descriptor contains an id that identifies uniquely the thread, a set of levels of privileges (to control resource access and scheduling), a context (the state of the thread, i.e., the values of the local variables and the last instruction executed), and status information (for scheduling and communication).

A multi-threaded operating system is the control-flow resulting of the interleaved execution of threads. A special thread, called the *interrupt handler*, manages the scheduling and the interaction between threads; it is activated in-between the execution of any other two threads.

Spin Model We model threads by Spin processes, whose interleaved execution is provided natively by Spin. In this setting, each thread is given a unique Spin process id, and its context is modeled by the state of the Spin process. A thread descriptor is represented by a Spin data-structure composed of: the Spin id (and therefore the associated context), the execution privilege, scheduling information and a message queue for communication:

```
typedef Thread_desc {
  byte contextPtr;
  bool privilege;
  SchedulerInfo schedInfo;
  MessageQueue msgQueue;
};
```

In order to control the interleaving, we use the special Spin process definition `provided`, that specifies a condition under which a process is executed or not. For example, the kernel service of Topsy in charge of IOs is modeled as follows:

```
proctype ioThread()
  provided (_curr_ctxt == _pid) { ... };
```

where `_curr_ctxt` is the Spin id of the currently running operating system thread, and `_pid` is the Spin id of the currently executing Spin process (this is a variable natively provided by Spin).

Although Spin has a native feature to spawn a process, it does not allow to end it dynamically: a process must reach its last statement to terminate. In order to model termination of user threads (kernel threads are never killed), we add a clause to the `provided` condition that allows for a killed process to execute to its end:

```
proctype uThread()
  provided (_curr_ctxt==_pid || _killed[_pid])
  { ... };
```

2.2 Scheduling

Definitions Scheduling is the operation by which the interrupt handler chooses the next thread to be run. This decision is based on priorities associated with threads. There is a wide variety of algorithms for this purpose.

The Topsy kernel implements a priority-based round-robin scheduling, by which the highest-priority ready-thread is always chosen. The scheduler uses queues to store the threads according to their status (`RUNNING`, `READY`, or `BLOCKED`). In addition, threads in the `READY` queue are sorted by their priority: "kernel" > "user" > "idle" (for a special idle thread executed when no thread is ready).

Spin Model The algorithm implemented in the model obeys the same priority rules as in Topsy. For this purpose, we use the scheduling status and priority that are stored into the `schedInfo` field of thread descriptors. The scheduling decision is stored in a global variable `_curr_id` that indexes a thread descriptor, from which one can retrieve the corresponding context (a Spin id).

There is a small difference with the original algorithm: we use a traversal of thread descriptors instead of queues, so that the scheduling is not fair anymore. This is not a problem for the properties we verify in this paper, because they are not related to the scheduling policy. However, in order to verify fairness properties, one would need to re-implement the scheduler with queues.

2.3 Hardware

Definitions The hardware consists essentially of a processor that provides execution of a single thread and accesses to resources such as segments of memory. To control the accesses to resources, the processor provides privileges (usually, operating systems privileges map the hardware privileges). In particular, the execution privilege level of the currently running process is always kept as a part of the context (usually in a register).

To enable interleaving of executions, the processor provides *interrupts*: a mechanism that puts a value into a special register and triggers switching of threads. Usually, the hardware includes an internal clock that can be used by the operating system to switch a thread after a predetermined execution period.

Spin Model To provide exclusive execution, we model the program-counter register of the processor by the global variable `_curr_ctxt`, that contains the Spin id of the current thread. In order to keep track of the current privilege in our model, there is a global variable called `CPU_MODE`; it is set by the context switch to the execution privilege level of the running thread.

To switch from one process to another, we use the `provided` clause: the execution of a given thread is triggered by changing the value of `_curr_ctxt` to the appropriate Spin id.

To model interrupts, we use a special channel to store the nature of the interrupt before context switching. For example, any thread can raise a software interrupt by sending a software interrupt message on this special channel and switch. In constrast, IOs interrupt are raised by external Spin process (representing some piece of hardware). In particular, the internal clock is model by a Spin process that nondeterministically send a time interrupt and switches.

2.4 Memory

Definitions For security reasons, processors allow to partition memory into independent regions associated with a memory access privilege level. When a thread attempts a memory access, the processor compares its execution privilege with the access privilege of the corresponding region. If the execution privilege is equal or higher, the access is granted, otherwise the processor stops the execution and raises an interrupt.

To use its memory efficiently, a kernel usually appeals to dynamic memory allocation. For example, dynamic memory allocation is used for dynamic creation and destruction of threads. Typically, such an allocator maintains a partition of free and allocated blocks inside the kernel data memory.

Since Topsy uses only one multi-threaded user application, the memory is split in two regions to separate the kernel from the user application.

Spin Model We model the memory access mechanism by (1) an array that associates each region with its privilege level and (2) a set of macros that models memory accesses. At each memory access, these macros check the current mode processor (`CPU_MODE`). If the access is allowed, the Spin assignments are executed, otherwise a memory-fault interrupt is raised and the context is switched.

In order to model a memory allocator that manipulates several types of data structures, we provide a macro that creates an instance of a specialized memory allocator for a given data-structure:

```
#define HL(type,size,data,used,hmlock,
  hmInit,hmAlloc,hmFree)
type data[size];
bool used[size];
chan hmlock = lock;
inline hmInit(i) { ... };
inline hmAlloc(return) { ... };
inline hmFree(return) { ... };
};
```

For illustration, the memory allocator of the kernel is instantiated by:

```
HL(Thread_desc, MEMBLOCK_N, mem, used, hmlock,
   hmInit, hmAlloc, hmFree)
```

The effect of the macro expansion above is to build an array `mem` of thread descriptors, an array `used` to keep track of which thread descriptors are free or allocated, and a set of functions `hmInit`, `hmAlloc` and `hmFree`. The initialization function `hmInit` sets all the data-structures to `Free`. The allocation function `hmAlloc` tries to find a free data-structure, declares it `Allocated` and returns its index. The deallocation function `hmFree` sets a data-structure of a given index to `Free`.

2.5 Kernel Services

2.5.1 IPCs

Definitions IPCs are a message passing mechanism that allows threads to communicate with each other. To use this mechanism, a thread sets its register to appropriate values (id of the receiver/sender, address of a buffer where the body of the message is stored or have to be stored), and then raises a software interrupt. This interrupt switches the context of the thread with the context of the interrupt handler. The latter routes the message, makes a scheduling decision and finally restores the running thread by switching the context.

Spin Model A global channel is used to pass arguments from a thread to the interrupt handler, whereas the response is sent back to the thread through a private channel whose pointer is passed as the `reply` argument of the message receiving and sending functions:

```
inline recvmsg(from, smsg, reply) { ... };
inline sndmsg(to, smsg, reply) { ... };
```

These two functions make use of the `msgQueue` field of the thread descriptor to store messages and communication status information. Let us explain in more details their implementations.

When a thread wants to receive a message, the interrupt handler looks into its message queue. If an adequate message is present, it is dequeued and sent to the thread, otherwise the thread is blocked and declared waiting for a message, and the IPC arguments are saved. Concretely, the interrupt handler sets the thread to a status called `WAITING` (`msgPendingStatus` field of `msgQueue`), saves the expected sender id (`threadIdPending` field) and the pointer of the channel where to send back the message (`msgPendingPtr` field).

When a thread wants to send a message, the interrupt handler tries to find the receiver. If it is not an existing thread, the IPC fails, otherwise the interrupt handler checks whether the receiving thread is waiting for this message. If this is the case, the message is sent directly to the thread which is unblocked, otherwise it is inserted into the thread message queue `msgQueue`.

2.5.2 System Calls

Definitions In Topsy, system calls (such as threads creation and destruction) are provided by kernel threads. More precisely, a thread makes a system call by sending a request to the appropriate kernel threads via an IPC message, whose body contains the name of the system call and its arguments. In the same way, the result is sent back into a message.

Spin Model The kernel threads are defined as Spin processes, and therefore are managed like the other threads. Yet, they belong to the kernel code and hence can manipulate directly its data. These threads are implemented as infinite loops which receive and parse a message, execute the system call code, and send back a response.

Our model implements the thread manager (responsible for creation and deletion of threads), the IO manager (that acts as a directory for IO drivers), and the network manager (network IO driver) but not the memory management kernel thread (whose purpose is to manage pages of virtual memory, but Topsy v2 uses a flat memory model and there is no implementation yet).

3 Experiments

We present several verifications done on the Topsy model described in the previous section: "status correctness" is a low-level property that only deals with the scheduler, "status consistency" and "reply consistency" are high-level properties that deal with both the scheduler and the IPCs, and "task isolation" is a high-level property that deals with the execution privilege and memory management.

3.1 A Generic Test Program

Verifications are done using a test program. This test program is an echo server that is generic in the sense that it uses all the kernel services provided by the model of the Topsy thread manager and IO manager (see Sect. 2.5.2).

The main thread of the echo server repeatedly does the following: it tries to create a child-thread using the Topsy thread manager and waits for a message by which the child-thread indicates it is about to terminate. The child-thread does the following. When it starts, it tries to open a network connection by sending a request to the Topsy IO manager; the response it gets is the id of a kernel thread managing the network. Via this kernel thread, the child-thread eventually receives a network packet, sends it back (the echo service), and sends a closing message to the IO manager. Then, it sends a message to its father indicating it will make an exit system-call.

3.2 Status Correctness

The status correctness property states that the kernel always restores the thread that has been scheduled. Put formally, "whenever a thread is executing (`threadrun` assertion below), its scheduling status is `RUNNING` (`runningthreadcurr` assertion)":

```
#define threadrun
  (_curr_ctxt == (mem[curr_id].contextPtr) &&
  _syst_run)

#define runningthreadcurr (used[curr_id] &&
  mem[curr_id].schedInfo.status == RUNNING)

[](threadrun -> runningthreadcurr)
```

3.3 Status Consistency

This property states that a thread waiting for a message can never be scheduled. Put formally, "if a thread is waiting for a message (`waitingthread` assertion below), its scheduling status must be `BLOCKED` (`blockedthread` assertion)":

```
#define waitingthread (used[ut_init_id] &&
  mem[ut_init_id].msgQueue.msgPendingStatus ==
    WAITING)

#define blockedthread (used[ut_init_id] &&
  mem[ut_init_id].schedInfo.status == BLOCKED)

[]((waitingthread && threadrun) ->
  blockedthread)
```

In this specification, `threadrun` ensures that this is not the interrupt handler that is executing. This is essential to distinguish this situation because the interrupt handler precisely may break this property when updating the thread status.

The Topsy thread id of the user thread is hardwired (`ut_init_id` variable). This is not a limitation of our approach because in Spin it is always possible to construct by hand a never-claim with an implicit quantification over a range of thread ids. However, this is not directly expressible in LTL.

3.4 Reply Consistency

The reply consistency property states that the return channel for an IPC is not changed until the thread is unblocked and the expected message is sent. Put formally, "if a thread is waiting for a message, the value of its return channel (field `msgPendingPtr` below) does not change until the thread is `NOT_WAITING`":

```
#define notwaitingthread (used[ut_init_id] &&
  mem[ut_init_id].msgQueue.msgPendingStatus ==
    NOT_WAITING)

#define pendingPtrval (used[ut_init_id] &&
  mem[ut_init_id].msgQueue.msgPendingPtr ==
    _reply_chan[ut_init_id])

[](waitingthread ->
  (pendingPtrval U notwaitingthread))
```

3.5 Task Isolation

The task isolation property is important for operating system verification because it implies that only the kernel can change its data. For a multi-threaded OS such as Topsy (only one user application), the task isolation means that whenever a user thread is running, it must not have a privilege level that grants him access to the memory of the kernel. Put formally, "whenever a user thread is running (`userthreadrun` assertion below), the current privilege does not allow access to kernel memory (`kernelaccess` assertion)":

```
#define kernelaccess
  ((CPU_MODE == segment[0]) || kernelmode)

#define userthreadrun
  (_curr_ctxt == (mem[curr_id].contextPtr) &&
  _user_thread[curr_id] && _syst_run)

[](userthreadrun -> !kernelaccess)
```

3.6 Results

Despite its completeness, the size of the model is reasonable: 770 lines of code for Topsy and 80 lines for the echo server (the whole Spin development is available online [7])

We measure[1] resource consumption in function of the number of child-threads created by the echo server for the properties of the previous sections.

The verifications of "status correctness", "status consistency" and "task isolation" use almost the same amount of ressource, linear in both space and time (that is why we merge these three graphs into only one in Fig. 1).

In contrast, the verification of "reply consistency" uses more ressource. The space consumption is still linear, but around three times larger because state formulas of the LTL formula contain more information. The time consumption is quadratic because the two temporal modalities nested in the LTL formula require an additional nested traversal of the state-space.

4 Related Work

Contrary to our work, all Spin-based verification of operating systems focus on only one property. In consequence, the proposed models are specification-oriented and does not enable verification of high-level properties.

In [1], the inter-task communication facility of the RUBIS micro-kernel is modeled to build test programs. Verifications check properties such as consistency of flags or validity of the status, similarly to "status correctness" and "status consistency" we checked in Sect. 3.

In [2], the Fluke kernel is modeled by a set of macros and used in several test programs. These programs are decorated with assertions checking the return values of kernel functions. Although the model spans a wide part of the kernel, it does not include any modeling of the hardware.

In [3], the VFiasco IPCs are modeled to verify the communication mechanism of the kernel. The approach is to translate directly the source code in Spin. The modeled scenario is composed of two communicating threads, modeled as Spin processes. All the tests focus on the consistency of flags used by the communication mechanism.

[1]Experiments done on an Opteron (64-bit) 2.4GHz machine with 16GB of RAM.

5 Conclusion

In this paper, we have model-cheked several properties of a multi-threaded operating system. The originality of our model is to enable verification of high-level properties, i.e., properties whose verification requires modeling of several parts of the system (not only the operating system but also its underlying hardware). In particular, this model made it possible to verify the task isolation propery of the Topsy operating system. Despite the richness of the model, our experiments showed that resource consumption during verification is reasonable enough to be performed on a standard computer.

References

[1] G. Duval and J. Julliand. Modeling and Verification of the RUBIS μ-kernel with SPIN. In *1st SPIN Workshop (SPIN 1995).*

[2] P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, and G. Back. Formal Methods: A Practical Tool for OS Implementors. In *6th Workshop on Hot Topics in Operating Systems (HotOS-VI 1997).*

[3] Endrawaty. Verification of the Fiasco IPC Implementation. Master Thesis. Desden University of Technology. 2005.

[4] W. R. Bevier. A Verified Operating System Kernel. Ph. D. Thesis. University of Texas at Austin. 1987.

[5] G. J. Holzmann. The Spin Model Checker. Addison Wesley. 2003.

[6] L. Ruf, C. Jeker, B. Lutz, and B. Plattner. Topsy v3: A NodeOS For Network Processors. In *2nd International Workshop on Active Network Technologies and Applications (ANTA 2003).*

[7] R. Affeldt and N. Marti. Spin model of Topsy available at: `http://staff.aist.go.jp/reynald.affeldt/seplog/`.

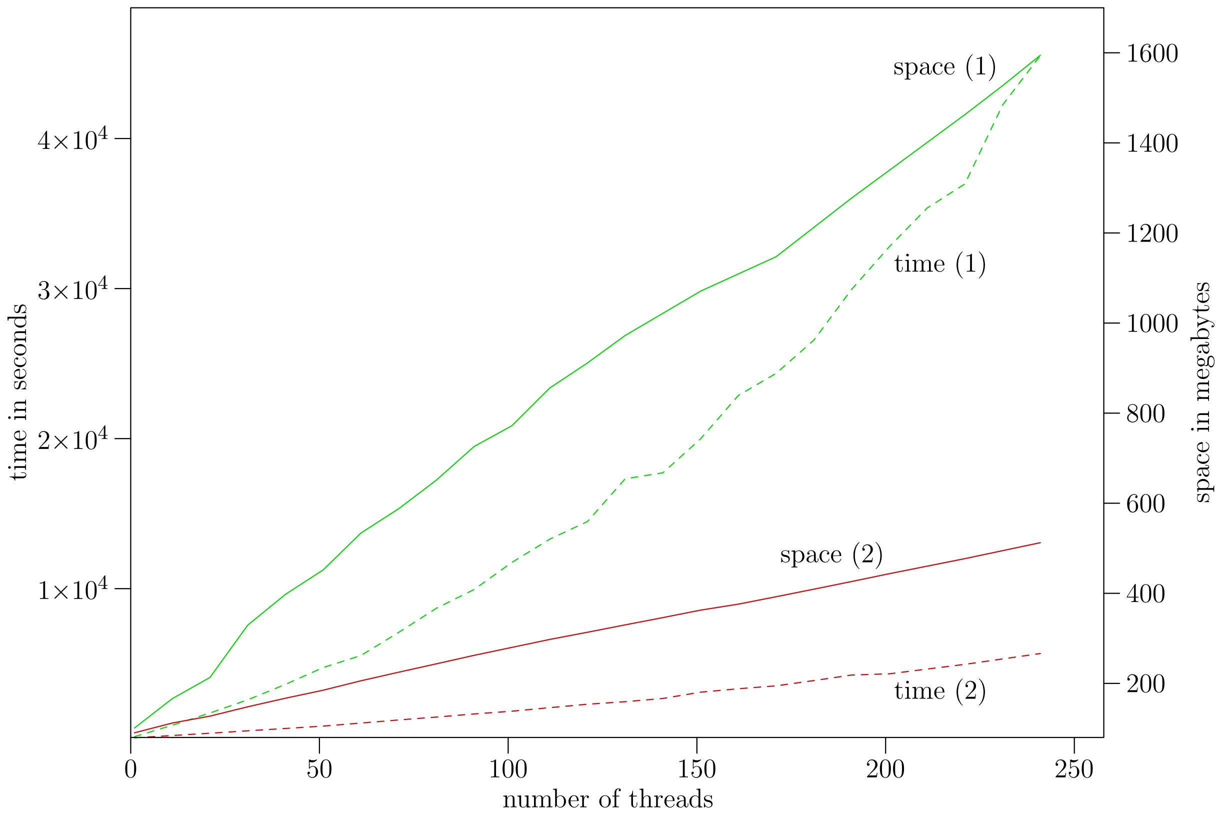

Figure 1: Resource consumption (memory space and execution time). Graphs (1) are for the “reply consistency” property. The results for the properties “status correctness”, “status consistency”, and “task isolation” are merged in graphs (2).