Reprinted from the

Proceedings of the Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada

UML Simulator

Werner Almesberger[*]
`werner@almesberger.net`

Abstract

umlsim extends user-mode Linux (UML) with an event-driven simulation engine and other instrumentation needed for deterministically controlling the flow of time as seen by the UML kernel and applications running under it.

umlsim will be useful for a wide range of applications in research and kernel development, including simulations involving the networking code, regression tests, proof of race conditions, validation of configuration scripts, and also performance analysis.

This paper describes the design and implementation of umlsim, gives a brief overview of the scripting language, and shows a real-life usage example.

1 Introduction

Simulation is an effective means for examining properties of systems that are too complex, too volatile, too expensive, or simply too large to build and test in real life.

In the development of the Linux kernel, simulations only play a niche role, and are rarely used for more than helping in the design of individual components. Also for performance evaluation, there is broad reliance on benchmark suites, but little is done with simulations that would allow to pinpoint bottlenecks and regressions with much more accuracy.

umlsim provides an environment that allows the use of regular Linux kernel or application code in event-driven simulations. It consists of an extension of user-mode Linux (UML, [1]) to control the flow of time as seen by the UML kernel and applications running under it, and a simulation control system that acts like a debugger, and that is programmed in a C and Perl-like scripting language.

The key feature of umlsim is that—unlike most other simulators, which implement an abstract model of the system being simulated—it uses the original Linux kernel code, with only minor changes. This reduces the risk of creating simulations that differ in some important details from the original, avoids code forking, and generally shortens the process of designing and building a simulation.

The simulation environment is deterministic, i.e. running a simulation multiple times will produce exactly the same results, although one can of course also introduce real or pseudo randomness. This makes umlsim suitable for regression tests, and for exercising specific execution patterns that exhibit problems.

The project's home page is at `http://umlsim.sourceforge.net/`

One of the first uses of umlsim is to examine the behaviour of Linux TCP in gigabit networks [2], but it will also be useful for many other applications in research and kernel de-

[*]The work presented in this paper is conducted as part of the FAST project at the California Institute of Technology, `http://netlab.caltech.edu/FAST/`

velopment, including regression tests, examination of race conditions and other kernel bugs, validation of configuration scripts, and performance analysis.

This paper is intended for two different audiences: first, it aims to introduce the capabilities and concepts of umlsim to prospective users. Second, it gives other kernel developers an overview of the kernel changes, and describes mechanisms that could also be useful in other projects.

This introduction continues with the historical background and related work. Section 2 discusses overall design and implementation aspects, and section 3 describes the most important elements of the scripting language. A real-life simulation example is given in section 4. We conclude with a discussion of future uses and improvements.

1.1 History

The basic concept behind umlsim, namely to use original kernel and user space code in simulations, was already explored in the earlier tcng ("Traffic Control Next Generation" [3]) project.

The main component of tcng is a compiler that translates traffic control configurations from a high-level language to the low-level commands understood by the tc command-line utility. In that project, a simulator called tcsim is used to validate that these commands are formally correct, and that they also yield the desired behaviour. In particular, since the configuration process involves many inter-related parameters with poorly documented semantics, it happened quite frequently that the use of some parameters or constructs was mis-interpreted.

Simulators usually implement an abstracted model of the system they simulate. In the case of tcng, this approach could lead to a simulator that contains the same mis-interpretations as the program being tested, so both would happily agree on incorrect results.

To avoid this problem, tcsim reduces the amount of abstraction needed by building the simulation environment from portions of the original traffic control code of the kernel, and the tc configuration utility. The structure of tcsim is depicted on the left-hand side of Figure 1.

This approach also allows the use of powerful user-space debugging tools like ElectricFence [4] and valgrind [5] to find bugs in the original code.

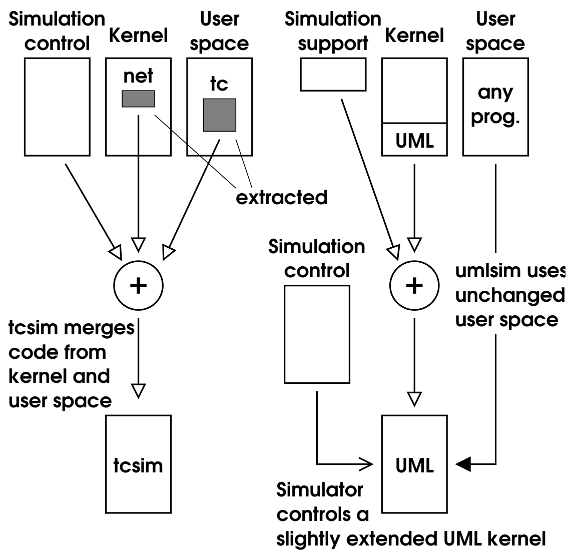
Simulation control
Kernel
net
User space
tc
extracted
Simulation support
Kernel
UML
User space
any prog.
tcsim merges code from kernel and user space
tcsim
Simulation control
umlsim uses unchanged user space
UML
Simulator controls a slightly extended UML kernel

Figure 1: tcsim uses a monolithic approach, with many dependencies on kernel and user space internals. umlsim is modular, and requires only very minor changes.

The difficult part in writing tcsim was to extract precisely the right amount of kernel code, and to make it fit in the simulation environment. In many cases, some small code modifications are needed to eliminate unwanted references to structure elements, variables, and functions not available in the simulator. All this makes the extraction procedure very sensitive to even the smallest changes.

tcsim was written for 2.4 kernels. Early in 2.5 development, it became clear that the networking code had changed sufficiently to require a major rewrite of the extraction process.

Another limitation of tcsim is that it only covers a very small part of the networking stack. For instance, it would be interesting to use TCP as a reactive traffic source.

The bottom line of the experience with tcsim is that, while using the original source also for the simulator works well, the process of extracting it causes problems and confines the simulator to only a small part of the system. So, why not avoid the extraction step at all, and use the entire kernel?

This is the approach chosen for umlsim, as shown on the right-hand side of Figure 1: instead of extracting the interesting bits from the kernel, it builds on UML, where all the work of making the Linux kernel run in user space has already been done, and adds a few functions for simulation control to it. User space is left completely unchanged.

1.2 Other simulators

Particularly in the area of networking, simulators are rather common tools. In many cases, they focus only on a very limited set of functions, such as a specific protocol. Among the more general simulators, the network simulator ns-2 [6] is certainly the one most widely known.

ns-2 consists of a modular simulation core written in C++, which is configured through scripts written in an extended version of the Tcl scripting language. The core provides network elements, protocol engines, and traffic generators.

umlsim also has a "core," but this core provides only very low-level primitives, and higher level functions are implemented by scripts. On the other hand, large subsystems, such as TCP, are simply reused without needing any special treatment in the simulator, and they behave in every detail like in a real system.

ns-2 is much faster than umlsim, and will probably always be, while umlsim is more general and can also be used for simulations involving other subsystems, instead of or in addition to networking.

2 Simulator design

umlsim consists of a simulation control process (we shall call it simply "the simulator"), and the UML systems that are being studied in the simulation. Besides UML systems, a simulation can also include other processes, e.g. to implement communication services. The general structure of a simulation system is shown in Figure 2.

The simulator executes a script in a C/Perl-like language. Scripts serve two purposes: (1) they define the simulation and control its execution, and (2) they provide the "glue" between the actual simulation and the processes used in it, and also between elements in these processes.

A simulation can choose how closely the simulator and the UML systems interact, i.e. the simulator may just watch a few variables and perform basic synchronization, but exercise no further control over execution, but it may as well intercept even the slightest activity, manipulate variables in the UML kernel, and even alter the flow of execution. Typically, umlsim controls UML at a very low level, but hides most of these interactions inside the simulator and behind library functions that provide a higher level of abstraction.

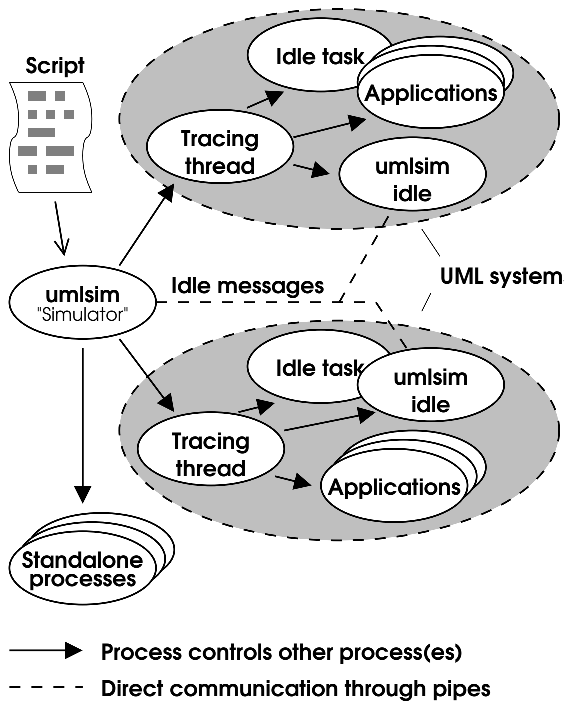

Figure 2: The simulator controls UML systems and other processes. Each UML system in turn consists of several processes.

The simulator basically acts like a debugger, and places breakpoints into the UML kernel. When the kernel is stopped, the simulator can read and change variables. The simulator can also call functions, make them return, etc.

In addition to this, the simulator exchanges time updates with the umlsim idle thread described in the next section directly through a pair of pipes.

Figure 3 shows the current structure of libraries. Work in this area of umlsim is still very much in progress.

2.1 Virtual time

It is frequently desirable to run simulations in a deterministic virtual time instead of real time. umlsim can accomplish this by adding code to the kernel that intercepts all functions reporting

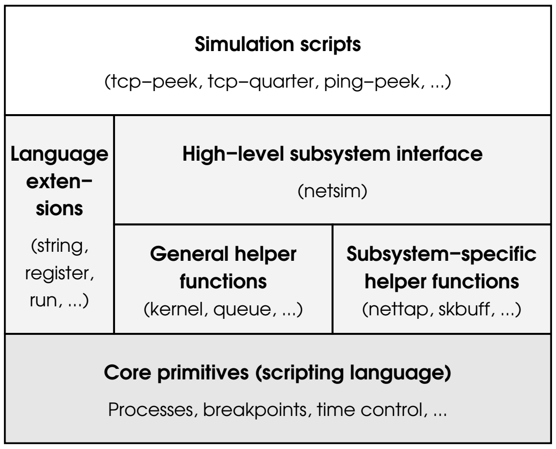
Simulation scripts
(tcp–peek, tcp–quarter, ping–peek, ...)
Language exten–sions
(string, register, run, ...)
High–level subsystem interface
(netsim)
General helper functions
(kernel, queue, ...)
Subsystem–specific helper functions
(nettap, skbuff, ...)
Core primitives (scripting language)
Processes, breakpoints, time control, ...

Figure 3: Organization of libraries in umlsim.

or advancing time, and puts them under its own control. This code also introduces a umlsim-specific idle thread that yields to all other tasks, except the kernel's regular idle task.

Whenever the kernel is idle (i.e. no process is scheduled to run), the umlsim code in the kernel does one of the following:

- if a soft-interrupt is pending: it generates a timer interrupt, but does not advance time
- if a timer will expire within the next jiffy:[1] it generates a timer interrupt, and allows `do_timer` to advance time by one jiffy
- if the next timer will expire in the future: the UML kernel reports this back to the simulator, and waits for further instructions

Since the kernel always runs some timers (such as `neigh_periodic_timer` and `rt_check_expire`), umlsim does not need to handle the case of a timeout without further activity.

[1]The "jiffy" is the basic time unit in the kernel. One jiffy typically equals 1–10 ms.

The kernel can also become active when a device interrupt arrives. umlsim currently only handles network events. Instead of using signals (which correspond to interrupts in UML), it calls the functions invoked by interrupt handlers directly.

When all kernels in a simulation report that they are waiting for a timer, the simulator picks the earliest expiration time among these timers (or a timeout specified in the `wait` command, if it is earlier), sets the global simulation time to that value, and updates the local time in all kernels.

2.2 Running UML under a debugger

When running UML under a debugger or a similar program (such as strace), the tracing thread watches the debugger with `ptrace`, intercepts calls to functions like `ptrace` and `waitpid`, and emulates them or redirects them to the process currently executing the kernel. This part of UML is called the "`ptrace` proxy."

Unfortunately, this design allows only a single UML system per debugger, because a process can be watched with `ptrace` by at most one process at any given time.

In order to control multiple UML systems, umlsim forks a forwarder process for each such system. This process communicates with the main simulator through pipes, and executes the `ptrace` calls on its behalf. This is shown in Figure 4.

As an example, Figure 5 shows a simplified flow of control when the simulator performs a `ptrace` call on a UML system.

2.3 Debugging the kernel

There are several idiosyncrasies of kernel code and of the way gcc compiles it that need special attention in umlsim. This section describes some of them.

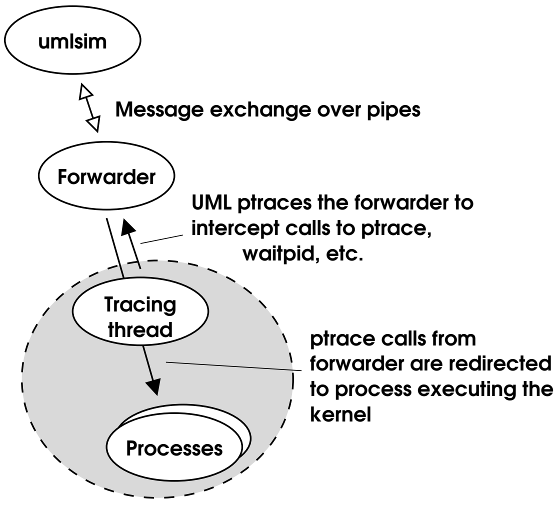
umlsim
Message exchange over pipes
Forwarder
UML ptraces the forwarder to intercept calls to ptrace, waitpid, etc.
Tracing thread
ptrace calls from forwarder are redirected to process executing the kernel
Processes

Figure 4: umlsim uses an intermediate forwarder process to "debug" the UML system.

Because the `jiffies` variable is defined in the linker, the debugging information generated by the compiler only contains its declaration, but not its location. umlsim therefore retrieves this information from the symbol table of the kernel executable, and augments the declaration with it.

Some functions use registers instead of the stack to pass arguments (e.g. those declared with `FASTCALL`). umlsim currently does not support or even recognize this.

The kernel makes heavy use of inline functions. One peculiarity of inline functions is that breakpoints in an inline function need to be repeated for each instance of this function. Furthermore, gcc rearranges and sometimes even removes labels (the ones used as targets for `goto` statements) when optimizing. umlsim introduces a mechanism called "reliable markers" that includes an explicit label in the function, which can then be used for breakpoints. Reliable markers also work in functions that

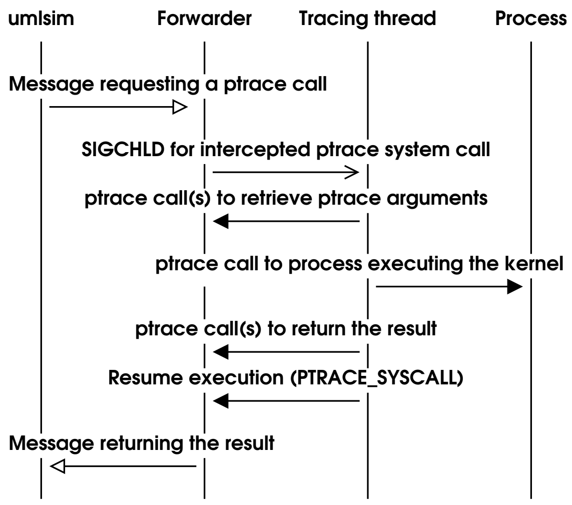
umlsim
Forwarder
Tracing thread
Process
Message requesting a ptrace call
SIGCHLD for intercepted ptrace system call
ptrace call(s) to retrieve ptrace arguments
ptrace call to process executing the kernel
ptrace call(s) to return the result
Resume execution (PTRACE_SYSCALL)
Message returning the result

Figure 5: Simplified control flow when the simulator performs a `ptrace` call on a UML system.

are not inlined, and are used as follows:

```
void some_function(int a)
{
    int b = 10;

    MARKER(label_name,a,b);
...
```

Variables that may be accessed by the simulator when stopped at the location of the marker are listed after the label name. This makes sure that the variables in question have a memory location, that they are not cached in registers when passing the marker, and that no code accessing these variables gets moved across the marker. (E.g. in the example above, the compiler might otherwise try to move the initialization of `b` after the marker.)

Also, since many low-level service functions are declared `static inline`, they cannot be called directly. umlsim generates callable instances of the most common inline functions by including their definitions in a file compiled with `-fkeep-inline-functions`.

2.4 Kernel changes

The kernel changes required for umlsim are comparably minor, and most of them are in the area specific to the UML architecture.

umlsim requires the following changes in generic kernel code:

- `calibrate_delay` explicitly waits for a timer interrupt, which would never happen under umlsim, because at that time, the umlsim idle thread does not yet exist. Therefore, umlsim simply skips `calibrate_delay` when using virtual time, and sets `loops_per_jiffy` to one.
- functions are added to `timer.c` to retrieve the expiration time of the next timer.

umlsim replaces the following functions using the linker's `--wrap` mechanism:

- `do_gettimeofday` and `gettimeofday` return the simulation time instead of the system's real time.
- `setitimer` becomes a no-op, because umlsim generates all timer interrupts under its own control.
- a switch is added to control whether a timer interrupt invokes `do_timer`. This way, timer interrupts can be used to run soft-interrupts without advancing the jiffies count.
- `idle_sleep` leads to the timeout handling code of umlsim.

The timeout handling code decides which actions to take (e.g. to raise a timer interrupt if there are pending soft-interrupts), communicates with the simulator, and maintains the various "current time" variables.

The umlsim patches also add the reliable markers, and callable definitions of common inline functions, which are both described in the previous section.

2.5 Network simulation

When simulating network elements, umlsim builds on the infrastructure used by uml_switch, but the script intercepts the transmit function and replaces most of the UML-specific part of the stack. Figure 6 shows the key functions invoked when sending and receiving packets.

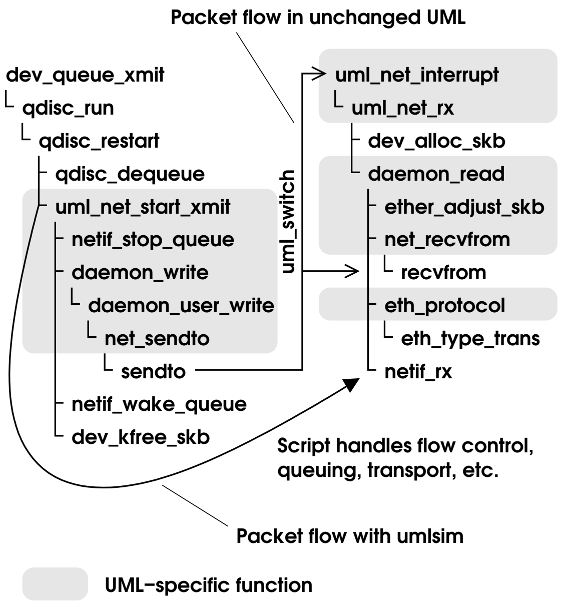

Figure 6: Call sequence and packet flow with and without umlsim (simplified).

With umlsim, the UML-specific networking part is only used for device setup, but all the transport and low-level packet manipulation functionality is provided directly by the simulation script.

umlsim currently only provides a single-link model, which will be extended and generalized in the near future. Figure 7 shows how the device interface and link are implemented. This code can be found in the files `include/netsim.umlsim` and `include/nettap.umlsim` of the umlsim distribution.

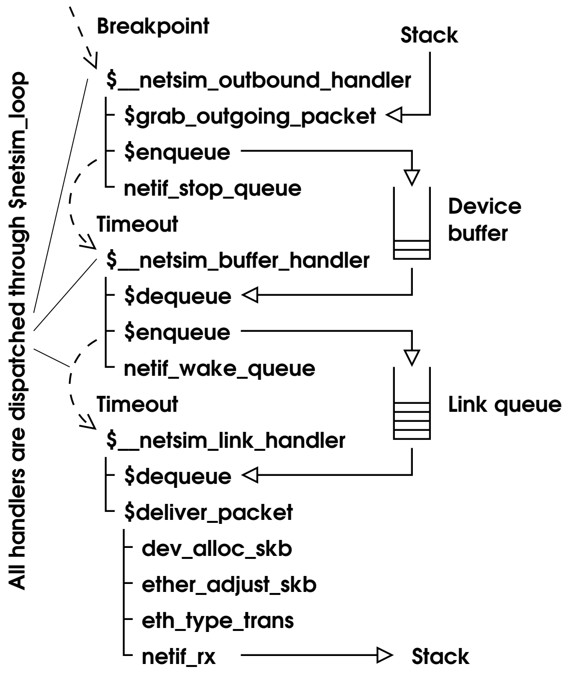

Figure 7: Control flow in script implementing network device and link.

When reaching the breakpoint `uml_net_start_xmit`, umlsim retrieves the packet, calculates the queuing delay, and stores it in an internal queue. If the device queue is full, the script calls the flow-control function `netif_stop_queue`.

When the packet is due for sending, it is dequeued and put into the link queue, from which it emerges after the transfer delay. umlsim then invokes basically the same functions as the original code, and finally pushes the packet to the stack by calling `netif_rx`.

3 The scripting language

The scripting language is mainly based on C and Perl, but it also borrows concepts from the Bourne shell, Pascal, and LISP. This section gives a brief overview of the most important concepts, and differences to similar languages.

umlsim passes scripts through the C preprocessor, so the usual comment handling, macro capabilities, and include files are available.

3.1 Variables and functions in the simulator

The names of variables in the simulator always begin with a dollar sign, like in Perl. Also like in Perl, variables can be used without prior declaration, their value can be of any type, and the type can be change.

An uninitialized variable has the so-called *undefined value*. The undefined value can also be used explicitly, with the construct `undef`, and one can test whether an expression yields the undefined value with `defined` *expression*.

The scripting language, like C, uses lexical scoping, i.e. the visibility of a variable is determined by its location in the program, but not by the sequence of function calls that leads to a specific access.

By default, variables are visible within the enclosing function, but not in other functions. This can be changed by either declaring them `local`, which creates a new, uninitialized instance that is visible in the current block and any blocks inside it, or by declaring them `global`, which creates a new instance in the current function, which is visible also in functions defined inside this function. Example:

```
$a = 5;
{
    local $a = 0;
    $a++;
    {
        $a++;
        printf("inner %d\n",$a);
    }
    printf("middle %d\n",$a);
}
printf("outer %d\n",$a);
```

yields

```
inner 2
middle 2
outer 5
```

The goal of these slightly unusual scoping rules is to avoid explicit declarations as much as possible, but also to avoid the problem of functions accidentally altering global variables, which is common in other scripting languages.[2]

Functions are anonymous, similar to lambda expressions in LISP. In order to reference a function by name, the function has to be stored in a variable. Example:

```
global $gcd = function ($a,$b)
{
    if ($a == $b) return $a;
    return $a > $b ?
      $gcd($a-$b,$b) : $gcd($a,$b-$a);
};

print $gcd(300,90);
```

The scripting language also supports associative arrays. Indices can be integers, pointers, strings, processes, or breakpoints. Elements can be of any type, including arrays. Examples:

[2]Only time—and users—will tell whether this is indeed an improvement over more traditional scoping rules. Users preferring to declare all their variables can set the `-Wundeclared` option to enable warnings when trying to access undeclared variables.

```
$a[0] = 3;
$a["string"] = $a;
print $a["string"][0];
```

3.2 Printing and files

The scripting language has two output statements: the C-like `printf`, and the "smart" and somewhat Perl-like `print`.

`print` accepts a list of items to print, appends a newline after the last item, and it pretty-prints structured types. Example:

```
$proc = uml("linux");
print "xtime = ",xtime;
```

yields

```
xtime = {
    tv_sec = 0 (0x0)
    tv_nsec = 0 (0x0)
}
```

`print` outputs integers as decimal and as hexadecimal numbers, enumeration type members by name, strings and signed character arrays as text strings, and arrays of unsigned characters as a hexdump. Like in Perl, a separator between printed arguments can be introduced by setting the special variable `$,`.

To send output to a file, the file first has to be opened with the `open` function, which has a file name argument like Perl's `open`, but returns a file handle. Then, the file handle can be used as the first argument of `print` or `printf`. Example:

```
$file = open(">tmp");
print $file,"example data";
printf($file,"answer = %d\n",42);
close($file);
```

Data can be read from files with the `read` function, but this is rarely used.

3.3 Control statements

`if-else`, `while` (with `break` and `continue`), and `for` work exactly like in C. There is no `do-while` loop, because `while` can be used in its stead.[3]

`switch-case` is similar to C, with the difference that variable expressions can be used for case labels.

There is no `goto`.

3.4 Processes

Simple programs are started with the function `run`, and UML systems are started with the function `uml`. Both functions return a handle that identifies the process. They also set the "magic" variable `$$` to this value. `$$` always identifies the *current* process, i.e. the process that has most recently been created or stopped, and that is currently being manipulated. `$$` can be changed by the simulator (when a different process becomes current) and by the script (if one wants another process to be current).

`run` and `uml` also support some basic IO-redirection, e.g. `run("/bin/date", ">/tmp/xyz")`

After `run` or `uml`, the process is in the *starting* state, but not yet running. A starting or *stopped* process is run with the `continue` statement.[4]

The `wait` statement is used to make the simulator wait for the next event (process termination, breakpoint, timeout, etc.). If the event is related to a process, `wait` sets `$$` to this process. Example:

[3]. . . and because the way programs are represented internally makes `do-while` somewhat difficult to express. It may be added at a later time.

[4]This `continue` has the process handle as argument, in parentheses, and therefore differs syntactically from the `continue` control statement. If continuing the current process, the parentheses can be left empty.

```
$proc =
  run("/bin/echo","hello world");
continue();
wait();
print $$ == $proc;
```

yields[5]

```
hello world
1 (0x1)
```

3.5 Breakpoints, functions, and timeouts

Breakpoints can be placed at function entry, at the point to which the current function returns, at labels, and at the reliable markers described in section 2.3. Breakpoints are set with the `break` function, which returns a handle that identifies the breakpoint.

In the example below, we set breakpoint `$b1` at the entry of the `main` function in the current process, breakpoint `$b2` at the label or reliable marker `label` inside the main function, and breakpoint `$b3` at the location to which the current function returns.

```
$b1 = break(main);
$b2 = break(main.label);
$b3 = break(return);
```

When reaching a breakpoint, umlsim sets `$$` to the process in which the breakpoint is located, and `$!` to the breakpoint handle.

When calling a function in the process, also a breakpoint is generated. This breakpoint is triggered when the function returns. The return value of the function is stored in the special variable `$?`. Example:

[5]After warning that `/bin/echo` has neither symbols nor debugging information, so there is very little umlsim can do with this process.

```
$b = call fn(1,2,3);
continue();
...
wait();
if ($! == $b)
    printf("result = %d\n",$?);
```

This example shows an *asynchronous* call, because other breakpoints, timeouts, or events in other processes can be handled before the function returns. If this flexibility is not needed, one can use the simpler *synchronous* form, which does not change `$!` or `$?`. Example:

```
printf("result = %d\n",fn(1,2,3));
```

Breakpoints can be removed implicitly, by destroying all references to them, or explicitly with `delete(`*breakpoint*`);`

A script can not only call functions in a process, but it can also make a function return. For example, `$__netsim_outbound_handler` in Figure 7 forces `uml_net_start_xmit` to return, without executing any code of that function, with `$$.return 0;`.

Besides terminating or reaching a breakpoint, a process may also stop with a timeout. Timeouts are specified with a time argument to `wait`. When the specified absolute time is reached, `wait` sets `$$` to the undefined value, and the "current time" variable `$@` to the timeout, rounded up to the next nanosecond. Example:

```
wait(10.2);
  /* wait until t = 10.2 seconds */
if (!defined $$) print $@;
```

If more than one timeout can occur at a given time (e.g. packets arriving within the same nanosecond at different points in the simulation), `wait($@)` must be called after handling each event, so that breakpoints reached when handling a timeout can be processed before handling further timeouts. An example for using `wait($@)` can be found in the event loop at the end of `include/netsim.umlsim` in the umlsim distribution.

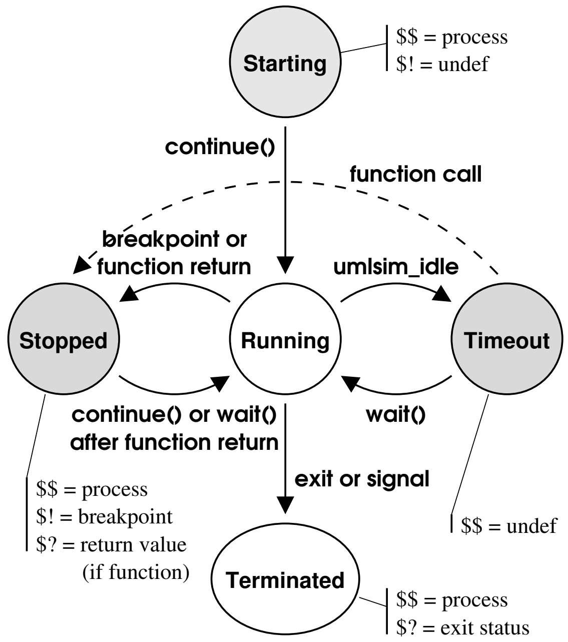

Figure 8: User-visible process states. "Function call' and "return" refer to asynchronous function calls.

Figure 8 summarizes the process states described in this section. States shown in grey allow manipulations of the process, such as the creation of new breakpoints, access to variables, or function calls. A function call from *timeout* puts the process in a state equivalent to *stopped*, but it does not affect any of the special variables.

3.6 Data in a process

umlsim scripts can directly read and write variables in a process, follow pointers, select struct or union members, and so on.

The basic operation is to access a variable. In many cases, simply specifying the variable's name is enough, e.g. given the example program below, `foo` retrieves the value 42.

```
static int foo = 42;

int main(void)
{
    static int bar = 5;

    MARKER(stop_here,bar);
    return bar;
}
```

Accessing `bar` is more complicated. If the program has not been started yet, umlsim looks for variables only in the global scope. To access a variable local to a function, it has to be qualified with the function name, i.e. `main.bar`.

If the program is stopped at the label `main.stop_here`, umlsim searches the local scope first, so just `bar` is sufficient.

Variables, functions, and labels can also be qualified with the process and the compilation unit. Compilation units are in double quotes. Examples:

```
$b1 = break($proc.main);
$b2 = break("fs/ext2/super.c".
  parse_options);
"drivers/net/tun.c".debug = 1;
"tun.c".debug = 0;
```

Since distinct processes may use the same name for different types, also struct or union tags can be qualified, e.g. `struct $proc_a.sk_buff`.

Type definitions with `typedef` differ from C in that umlsim cannot usefully distinguish at parse time between typedef names and other identifiers. Therefore, typedef names are always prefixed with the keyword `typedef`, e.g. `typedef pte_addr_t`. Like all other names, they can be qualified. For convenience, the C99 standard integer types `uint32_t`, `int8_t`, etc. are predefined.

Conflicts between C identifiers and keywords of the scripting language (e.g. `printf`) can be resolved by escaping the word with a backslash when the C identifier is meant, e.g. `\printf`.

A peculiarity in the way umlsim handles data are array copies: when accessing an object of array type, the entire array is copied. To obtain a pointer to the array, the & operator must be used. Example:

C program fragment:

```
int a[10];
int b[10];
```

umlsim script:

```
$array = a;
b = a; /* memcpy equivalent */
$ptr = &a;
```

This can also be used in type casts. E.g. the following construct copies the content of a network packet:

```
$pkt = (unsigned char [skb->len])
  skb->data;
```

4 Simulation example

In this section, we use umlsim to demonstrate a bug in Linux TCP, and to show the effect of a possible fix. The problem in question, which was first observed on a simulator by Cheng Jin, is that Linux TCP[6] decreases the congestion window (*cwnd*, TCP's estimate of how many packets can be "in flight" for a given connection) too much if there are multiple packet losses in a single round-trip time.

[6]Most if not all 2.4 and 2.5 kernels are affected. At the time of writing, this bug still exists in the mainstream kernel. The entire discussion can be found at [7].

When a packet is lost, TCP assumes that this was due to congestion, and reduces *cwnd* by half. However, if multiple losses occur within a single round-trip time, they should be treated only like a single loss. Linux TCP does not do this, and may reduce *cwnd* to as low as a quarter of the original value. This causes TCP to send data a little slower than it would be allowed to.

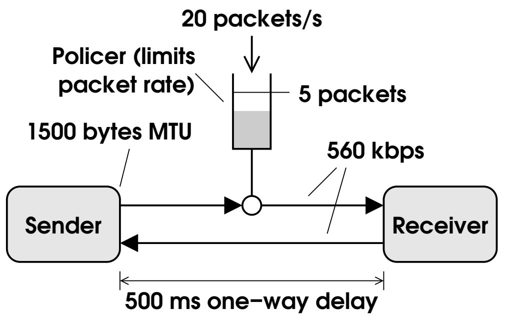

Figure 9: Network setup used in the simulation.

Figure 9 shows the network configuration used in the simulation: the TCP sender and receiver are connected by a single link with a round-trip time of one second. The maximum throughput is rate-limited to twenty packets per second. We simulate the transfer of a 1 MB file.

The left-hand side of Figure 10 shows the transfer with an unchanged 2.5.66 kernel. `snd_cwnd` is the congestion window, in segments. `snd_ssthresh` marks the point where TCP switches between "slow start" and "congestion avoidance" mode. `snd_cwnd` should not fall below `snd_ssthresh`. `snd_una` is the number of bytes that have been acknowledged by the receiver. `packets lost` is the cumulative number of packets dropped by the rate limiter.

For the second simulation, we use the same

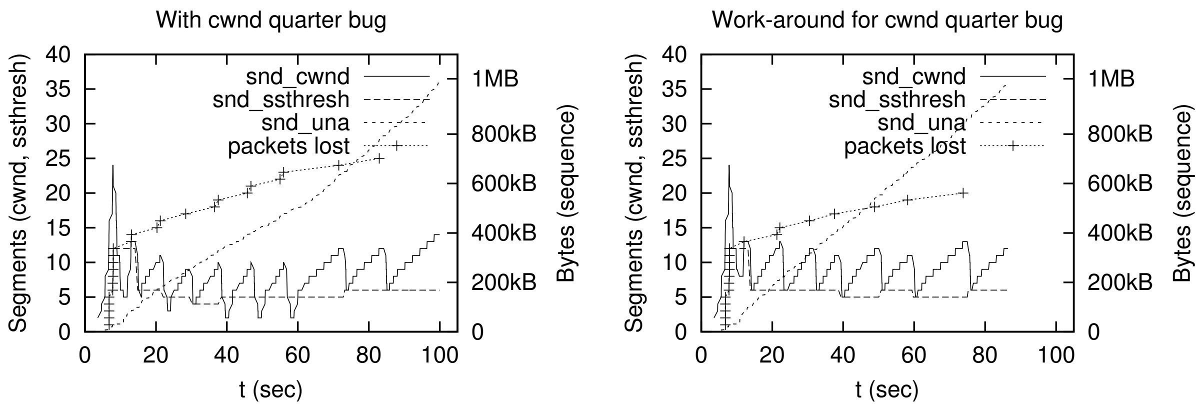

Figure 10: Simulated transfer with and without the "cwnd quarter" bug.

kernel, but set a breakpoint at the beginning of `tcp_cwnd_down`, and execute a replacement in the script instead of the original function. This replacement implements a fix that keeps *cwnd* from being lowered too far.

With this work-around in place, `snd_cwnd` never falls below `snd_ssthresh`, and the transfer finishes considerably earlier than in the buggy version.

5 Future work

As a complex but relatively young project, umlsim still has shortcomings in many areas. This section discusses some of the problems, and outlines approaches for solving them.

Future work on umlsim will primarily focus on the needs of network simulations, and in particular the analysis of TCP performance.

5.1 Functionality

Networking simulations are essentially limited to a single-link scenario at the time of writing. Work is under way for providing building blocks that allow the construction of arbitrary network topologies.

Another issue all but ignored so far is portability to architectures with other byte order or word size than ia32. Also support for multiprocessing is absent so far.

The simulator has currently no direct control over processes running in the user space under a UML kernel. It would be useful if simulations could treat such processes like ordinary processes, i.e. by launching them with a simple command, by placing breakpoints, etc.

It would be interesting to explore the possibility of using umlsim to reconstruct the internal state of the kernel, based on traces obtained from "live" systems. For example, this could be used to explain anomalies in network activity captured with tcpdump. The open issue here is how quickly unavoidable time differences and events not recorded in the trace (such as soft-interrupt execution after a hardware interrupt) will cause the simulation to diverge from the original system, and how such errors can be compensated.

5.2 Usability

umlsim today is clearly a hacker's toy. Most users will want high-level components when implementing their simulations, and the scripting language could also use some minor cleanup.

Dark corners of the language include the cast operator, pointers to data in the simulator, inconsistencies in the syntax (e.g. normally, `$$.`*thing* is equivalent to just *thing*, but `return` is very different from `$$.return`), and subtle differences in the semantics of array indices and `case` expressions.

To be useful outside the kernel hacker community, umlsim needs libraries with application-oriented building blocks that provide a convenient level of abstraction. At the time of writing, such a library is slowly emerging for networking, with the main focus on TCP.

Beyond libraries, also preprocessors that translate simpler application-oriented languages, like the one used by tcsim, to umlsim may be useful.

One of the most important aspects of simulations is the visualization of results. While it is desirable to retain a maximum of flexibility, examples for data formats, and visualization packages for common tasks will help users to obtain results more rapidly.

Also, as befits a hacker's toy, documentation is incoherent and spotty.

5.3 Performance

At the time of writing, umlsim is rather slow. While some optimization work has been done to reduce startup time and to accelerate some lookup operations, and more recently also to accelerate the communication between the simulator and the UML processes, several areas remain where major speed improvement are possible.

`ptrace` is a rather inefficient means for accessing process memory. It would be better if the simulator entirely bypassed the tracing thread when reading or changing variables, and accessed the address space of the UML processes directly.

Also the performance of UML itself is the object of on-going work [8]. In particular, the so-called "skas mode" ("skas" stands for "Separate Kernel Address Space") has been added recently, to accelerate context switches of processes under UML [9]. By following these changes, umlsim will permit UML to run faster, which in turn will benefit overall system performance, and may perhaps also itself be able to access UML systems more efficiently.

Last but not least, several algorithms and data structures inside the simulator are rather inefficient, and will have to be improved for larger simulations. For example, associative arrays just store their elements in a linear list. Also, results of identifier lookups could be cached.

6 Conclusion

umlsim provides the infrastructure for turning the (UML) Linux kernel into a versatile event-driven simulator, that can be customized using a scripting language most programmers will find easy to learn.

The next challenges in the project will be to bring performance closer to that of comparable simulators, to improve overall usability, to apply umlsim to concrete problems, and to use experience gained from such real-life applications to further improve the simulator.

References

[1] Dike, Jeff *et al. The User-mode Linux Kernel Home Page*, `http://user-mode-linux.sourceforge.net/`

[2] C. Jin, D. Wei, S.H. Low, G. Buhrmaster, J. Bunn, D.H. Choe, R.L.A. Cottrell, J.C. Doyle, W. Feng, O. Martin, H. Newman, F. Paganini, S. Ravot, S. Singh. *FAST TCP: From Theory to Experiments*, Submitted for publication in IEEE Communications Magazine, Internet Technology Series, April 1, 2003. `http://netlab.caltech.edu/pub/papers/fast-030401.pdf`

[3] Almesberger, Werner. *Linux Traffic Control—Next Generation*, Proceedings of the 9th International Linux System Technology Conference (Linux-Kongress 2002), pp. 95–103, September 2002. `http://www.linux-kongress.org/2002/papers/lk2002-almesberger.html`

[4] Perens, Bruce. *Electric Fence*, `ftp://ftp.perens.com/pub/ElectricFence/`

[5] Seward, Julian; Nethercote, Nick. *Valgrind, an open-source memory debugger for x86-GNU/Linux*, `http://developer.kde.org/~sewardj/`

[6] *The Network Simulator – ns-2*, `http://www.isi.edu/nsnam/ns/`

[7] Almesberger, Werner; Jin, Cheng; Kuznetsov, Alexey. *snd_cwnd drawn and quartered*, Discussion thread on the netdev mailing list, December 25, 2002. `http://marc.theaimsgroup.com/?t=104078129500001`

[8] Dike, Jeff. *Making Linux Safe for Virtual Machines*, Proceedings of Ottawa Linux Symposium 2002, pp. 107–116, June 2002. `http://www.linux.org.uk/~ajh/ols2002_proceedings.pdf.gz`

[9] Dike, Jeff. *skas mode*, `http://user-mode-linux.sourceforge.net/skas.html`