# An introduction to UVMs

## What is this?

UVM support is a unique RTAI/fusion feature, which allows running a nearly complete real-time system embodied into a single multi-threaded Linux process in user-space, hence the name, standing for "User-space Virtual Machine".

Running a real-time application inside a UVM offers a straighforward approach for porting from other RTOS environments to RTAI/fusion. For this reason, all emulators of traditional RTOS available with RTAI/fusion (e.g. VxWorks, pSOS+, VRTX, and uITRON) can be hosted by UVMs, although they were initially intended to run as kernel modules. The rest of this article will focus on such use of the UVM support.

First of all, one should keep in mind that there is a 1:1 relationship between every real-time thread created inside any UVM and a corresponding RTAI/fusion thread in kernel space (aka *shadow* thread), which means that UVM-based threads do provide stringent real-time guarantees, just like any real-time thread/task created through the native RTAI interface.

Since the whole real-time system runs in the context of a single multi-threaded Linux process context, the target code is sandboxed in its own address space, and can be debugged using common tools such as GDB, which makes the UVM support a solution of choice for conducting first order ports from other RTOS platforms to a GNU/Linux environment.

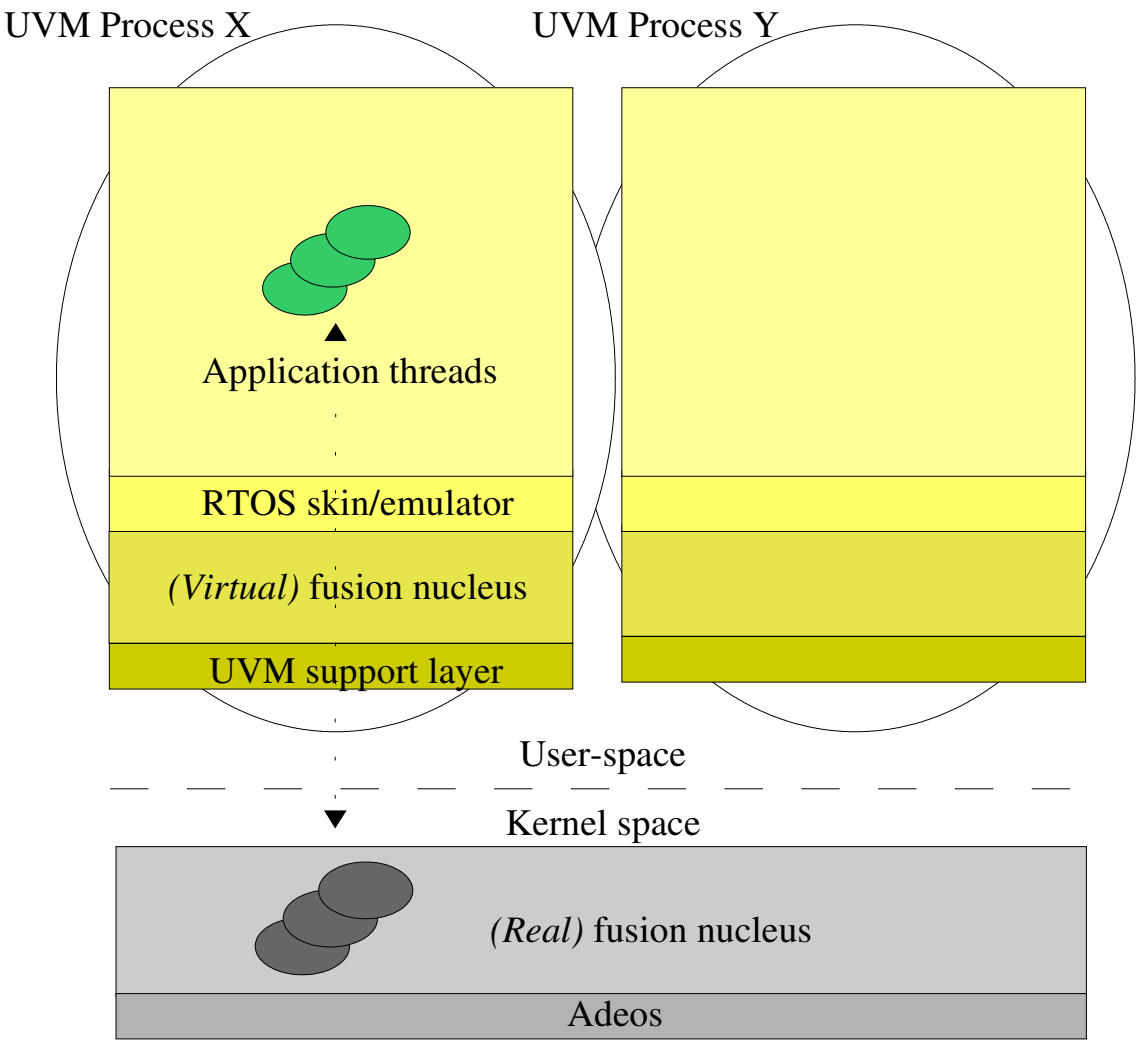
## How do UVMs work?

The UVM support builds upon the ability of most of the RTAI/fusion core (i.e. the nucleus and the various real-time interfaces) to execute over different contexts seamlessly. For instance, most of the nucleus code can either run over Adeos in kernel space - which is the usual configuration -, or on top of the event-driven simulation engine shipped with the RTAI/fusion distribution under the *sim/* directory. The context-dependent portions providing system-level resources, such as the CPU-dependent bits, are handled inside a normalized system layer which underlies the nucleus, onto which the real-time interfaces (aka *skins*) are eventually stacked. Incidentally, this is also what makes RTAI/fusion a highly portable system, since only Adeos and the well-defined machine-dependent portions of the system interface layer need to be ported to a new architecture.

In the same way, the UVM support is built over a special system interface layer which maps requests (e.g. actual real-time thread creation, hardware timer management) issued by a *virtual* nucleus image, embedded into the virtual machine along with the real-time application, to system calls processed by the *real* nucleus running in kernel space. Albeit this additional indirection, the approach remains remarkably efficient because most of the work induced by calling a real-time service from within the virtual machine is actually

performed inside the UVM itself, only leaving simple and limited actions to be done by the *real* nucleus in kernel space; most of the time, such action is about performing some actual context switch the *virtual* nucleus has decided for its own threads. Thanks to the 1:1 relationship which exists between *virtual* nucleus threads and *real* nucleus ones, the actions to perform this way are always straightforward and unambiguous.

Let's see how this translates into the RTAI/fusion architecture:




As you can see from the figure above, multiple UVM processes can co-exist concurrently on the same machine, all being eventually controlled by the central real-time nucleus in kernel space. Additionally, UVMs don't need any real-time API module (e.g. RTAI/fusion's VxWorks skin) to be present in kernel space, since each of them already embeds a user-space copy of the needed API module.

Another point of interest is about the representation of real-time threads in the figure above. If you are familiar with RTAI/fusion's concept of real-time threads in user-space,

you already know that each of them is underlaid by a regular Linux task, whose control is shared between the RTAI/fusion nucleus and the Linux kernel, depending on the thread's current execution mode, i.e. primary or secondary. As illustrated, each real-time thread started by the application lives in the UVM's address space, and has a system-wide kernel space counterpart. This counterpart is called a *shadow* thread in RTAI/fusion's jargon; it should be noted that such shadow entity is not actually a full-blown context with its own stack and set of CPU registers, but rather a placeholder for the underlying Linux task in user-space in RTAI/fusion's scheduling queues. In other words, UVMs do not suffer the overhead of a client/agent scheme, since only one stack context and CPU register set exists for each of the application threads it is hosting.

Going into deeper details about RTAI/fusion's scheme to bring ultra-low and predictable scheduling latency to user-space threads is beyond the scope of this article, but to sum up, one should keep in mind that UVMs threads created by RTOS emulators, and real-time tasks created by the native RTAI interface, are underlaid by the very same core support in kernel space; therefore, UVM threads are handled as deterministically as any other RTAI thread, and can also operate either in primary mode for reaching worst-case latencies in the lowest micro-second range, or in the secondary mode to access the full extent of regular Linux services seamlessly. By convention, emulators of traditional RTOS never call regular Linux services, thus will keep the calling threads in the highest deterministic mode when performing application requests.

# Under the hood

## *Initialization process*

At initialization time, the UVM support layer first maps the *main()* context of the new process to a real-time shadow thread in kernel space, then creates an internal timer thread, also mapped to a real-time shadow context, which will be in charge of simulating a periodic time source for the whole UVM process. Only the periodic timer service is provided by UVMs to applications, since it's the most relevant timing policy for emulators of traditional RTOS.

*/proc/rtai/sched* should display both threads respectively under the names *uvm-root* and *uvm-timer* when a UVM is running, and */proc/rtai/timer* should display the currently active timer setup.

## *System-wide priorities*

The UVM root thread is given the minimal SCHED_FIFO priority (*sched_get_priority_min(SCHED_FIFO)*), whilst the UVM timer has the latter priority + 2. All threads/tasks subsequently created by the embedded RTOS emulator upon request from the application will also belong to the SCHED_FIFO scheduling class, all with the minimal SCHED_FIFO priority + 1. All these priorities are meant to be system-wide, and are only known by the UVM support layer; this must not be confused with the priority levels defined by the application for the threads/tasks it creates, and which are enforced locally by the scheduler of the *virtual* nucleus. In other words, each UVM thread

has two priority levels, one applicable inside the UVM with respect to other threads running inside the same virtual machine, and a second one which has a system-wide scope and is used to schedule the given thread among all competing threads known by the *real* RTAI/fusion nucleus and the Linux kernel altogether.

## *Application startup*

Once the UVM's internal threads have been spawned and the timer has started ticking at the proper frequency, the UVM control layer calls the application entry point conventionally named *root_thread_init()*. This routine should exist within the application code, and contain all the necessary bootstrap code to initialize it properly, e.g.:

**mydemo.c:**

```
int root_thread_init (void)

{
    /* Create a message queue. */
    message_qid = msgQCreate(16,sizeof(char *),MSG_Q_FIFO);

    /* Spawn two real-time tasks to exchange messages
       through the just created queue. */

    consumer_tid = taskSpawn("ConsumerTask",
                        CONSUMER_TASK_PRI,
                        0,
                        CONSUMER_STACK_SIZE,
                        (FUNCPTR)&consumer_task,
                        0,0,0,0,0,0,0,0,0,0);

    producer_tid = taskSpawn("ProducerTask",
                        PRODUCER_TASK_PRI,
                        0,
                        PRODUCER_STACK_SIZE,
                        (FUNCPTR)&producer_task,
                        0,0,0,0,0,0,0,0,0,0);
    return 0; /* Success */
}
```

## *Typical system call dynamics with UVMs*

Let's say that our VxWorks application attempts to acquire a resource on behalf of one of its threads, from a fully depleted semaphore using the *semTake()* service provided by the embedded VxWorks emulator. Here is the typical flow of events which would occur:

- In the UVM space, the *semTake()* service implemented by the RTOS emulator notices that the caller should wait, link it to some internal wait queue, and calls the *virtual* nucleus to suspend it;

- The *virtual* nucleus in UVM space proceeds to blocking the caller, then updates its scheduling state searching for the next thread to run, and eventually requests the system layer to switch out the current thread, and switch in the next ready-to-run thread;
- The UVM system layer translates the request coming from the *virtual* nucleus to another simple one, palatable for the *real* nucleus in kernel space, i.e. resume the incoming thread which is ready-to-run at the UVM's level, and suspend the blocked thread, all in a system-wide manner.
- The real nucleus in kernel space receives the previous request through its internal syscall interface dedicated to the UVM management, then proceeds as requested. As a consequence of this, the real-time thread the *virtual* nucleus has elected for running, in order to replace the one just blocked on the VxWorks semaphore, resumes from its previous suspension, as expected.

# Building UVMs

## *RTAI configuration*

As illustrated before, each UVM embeds the following components into a single, multi-threaded Linux process:

- A copy of the standard RTAI/fusion's real-time core which will act as the *virtual* nucleus for the UVM;
- The UVM support layer which connects the embedded *virtual* nucleus to the *real* one running in kernel space for managing real-time threads;
- A copy of any RTOS skin or real-time interface needed by the application;
- The application code itself.

The first two components are automatically built when the *User-space VM support* from the *APIs* section has been enabled at configuration time for RTAI/fusion (CONFIG_RTAI_OPT_UVM=y). This feature also requires the real-time support in user-space to be compiled in, which is the default (CONFIG_RTAI_OPT_FUSION=y). Then, for each RTOS skin available with RTAI/fusion which has been selected into the *APIs* configuration section, and which supports UVM hosting, a user-space library belonging to the third kind of components will be produced. E.g. After installation */usr/realtime/lib/libvxworks.a* will contain the UVM-embeddable version of the VxWorks skin, */usr/realtime/lib/libpsos.a* will contain the one for the pSOS+ skin, and so on.

## *Compilation and link*

All recent RTAI versions provide the *rtai-config* script, which sends back various information upon request, like the valid compilation and link flags to pass to the GCC toolchain when building applications for various execution contexts (i.e. kernel, user-space, simulation), and RTAI/fusion is no exception to this. The Makefile frag below is enough to build a demo application called *mydemo*, from a single source file named

*mydemo.c*:

**Makefile:**

```
prefix := $(shell rtai-config --prefix)

ifeq ($(prefix),)
$(error Please add <rtai-install-path>/bin to your PATH variable)
endif

UVM_CFLAGS  := $(shell rtai-config --uvm-cflags)
UVM_LDFLAGS := $(shell rtai-config --uvm-ldflags) -lvxworks
UVM_TARGETS := mydemo

all: $(UVM_TARGETS)

$(UVM_TARGETS): $(UVM_TARGETS:%=%.c)
	$(CC) -o $@ $< $(UVM_CFLAGS) $(UVM_LDFLAGS)

clean:
	$(RM) -f $(UVM_TARGETS)
```

As you can see, most of the cryptic and cumbersome work is done by the *rtai-config* script, which is asked to return the correct compilation and link flags to produce a UVM executable.

# Running UVMs

Since a UVM is embodied into a regular Linux process, running it is just about launching a plain Linux executable, after having loaded the required kernel support, like this:

```
# insmod /usr/realtime/modules/rtai_hal.ko
# insmod /usr/realtime/modules/rtai_nucleus.ko
```

Because real-time threads are underlaid by native POSIX threads undergoing the SCHED_FIFO scheduling policy, super-user privileges will be required.

The UVM can be passed parameters using the shell environment strings, which largely depends on the application and the RTOS emulator used. The following parameter is common to all existing emulators of traditional RTOS:

tick_hz=<*periodic-tick-duration*>

The value is given in HZ, and usually defaults to 1000. It specifies the frequency of the UVM timer; in other words, in the default case, the UVM's internal time source will be programmed to tick each millisecond. Specifying a 250 us tick on the command line can be achieved this way:

```
# tick_hz=4000 ./my-uvm-based-app
```

or

# export tick_hz=4000; ./my-uvm-based-app

Please refer to the specific RTOS skin documentation for other parameters.

# Debugging code within UVMs

One word: GDB. RTAI/fusion has been designed to work as symbiotically as possible with the Linux kernel, and as such, standard *ptracing* and signal supports are kept available for use with real-time threads in user-space, like the ones created by UVMs. Internally, the threads will be switched automatically to secondary mode, i.e. under the control of the Linux kernel, when their respective code is stepped by GDB, or a breakpoint is hit. RTAI/fusion guarantees that they will be switched back automatically to primary mode as soon as it is needed.

# Any example code?

Yes, there is some. Search for a *demos/* sub-directory into the various existing RTOS emulator packages. When one exists, you can find there some simple but still runnable demos of the UVM support embedding the emulator code, along with the proper Makefile one can use to build it. Currently, some demonstration code exists for RTAI/fusion's VxWorks and pSOS+ emulators, respectively under the *skins/vxworks/demos* and *skins/psos+/demos* directories.

# Conclusion

The UVM support is actually born out of the author's laziness, with respect to writing several interface libraries, which should have implemented a user-level wrapper for each and every system call exported by each existing emulator of traditional RTOS in kernel space. Crafting the UVM support was actually much simpler and likely less boring than having to code up to eighty system call wrappers for some emulators like the VxWorks or pSOS+ skins... Although this interface library exists for the native RTAI interface and allows user-space applications to invoke kernel-based RTAI services, just like the ***glibc*** provides wrappers for each of the Linux kernel syscalls, on the other hand, emulators of traditional RTOS currently need the UVM support to be used in user-space.

This said, the UVM support is remarkably efficient latency-wise, since it is based on the same kernel support than are the native RTAI tasks in user-space, thus guaranteeing worst-case latencies in the lowest micro-second range. In the same way, the emulated RTOS tasks can also invoke regular Linux services, migrating seamlessly between the primary and secondary modes as needed. This should dramatically ease the porting effort, especially for portions of the original application which could accommodate with the possibly increased latency induced by using some of the native *glibc* services.

*rpm@xenomai.org*