RODOS

Tutorial REAME Core

Version: 2.0
Date: Sept. 2019
Author: S. Montenegro

1. How to begin

First compile the RODOS core. Go to the RODOS root directory and execute:

```
% source setenvs.sh
% rodos-lib.sh linuxMC
```

Then go to the Tutorials.

use

```
% rodos-executable.sh linuxMC <list of sources>
```

to compile and

```
% tst
```

Control-C to terminate (Control programmes normally do not terminate... never)

If you want to terminate the execution automatically, add the programm
terminate-test.cpp
to you source code list then it will be terminated after 20 seconds.
eg.

```
% rodos-executable.sh linuxMC timewaitfor.cpp terminate-test.cpp
```

to execute – e.g.

```
% rodos-executable.sh linuxMC timewaitfor.cpp
% tst
```

Or to combine two applications:

```
% rodos-executable.sh linuxMC timewaitfor.cpp timewaitbeats.cpp
% tst
```

For each example programme please first read and understand the code, then compile and execute and see if it acts as expected. Then modify and continue trying.

Try to combine different programs together.

Useful to know:

Time unit = nanoseconds, stored in 64 bits (`int64_t`).

Threads implement co-operative switch (yield) and pre-emption.

Priorities begin with 1 (lowest) up to 2^31, stored in long.

Compile and execute the following programmes. Be aware of the prints before your application begins. You will see there the configuration of the system and the list of loaded modules. Then take a look to the sources and try to modify them to add more functionality.

We have very few macros and only for operations which are used very often. We recommend to use such macros: (to find examples use for example %grep PROTECT_WITH *)

`NOW()`	Returns the current time (nanoseconds counter)
`AT(time)`	Because very often we have to wait a time point, AT is a short cut for suspendCallerUntil(time)

`TIME_LOOP(firstExecution, Period) {`
`  …`
`}`
	Almost each control loop has a start time and a period. This macro provides this loop with no end.

`PROTECT_WITH_SEMAPHORE(semaphore) {`
`  …`
`}`
	Critical blocks which shall be protected by semaphores, have almost always such structure. This macro uses the given semaphore to protect the region in { … }

`PRIORITY_CEILING {`
`  …`
`}`
	To deactivate scheduling for a very short time to protect very short critical section.

2. Programmes

2.1. Threads and Time

timewaitfor.cpp

This is a thread which waits for a time point. It loops and waits 2 seconds in each loop.

Check to see if the time difference of each execution is more than 2 seconds, because of the time consumed in the thread execution.

```
% rodos-executable.sh linuxMC timewaitfor.cpp
```

timewaitbeats.cpp

Periodic threads. It loops and prints in beats of 2 seconds. The beat shall have a minimal delay and no shifts as in timewaitfor.cpp

```
% rodos-executable.sh linuxMC timewaitbeats.cpp
```

Now try to combine both together

```
% rodos-executable.sh linuxMC timewaitbeats.cpp
timewaitfor.cpp
```

Observe the time shift.

For exact beats use `waitUntilNextBeat()`.

timewaitbeats_shortcut.cpp

A shortcut for timebeats using macros.

Compare with timewaitbeats

```
% rodos-executable.sh linuxMC timewaitbeats.cpp
```

timeevents.cpp:

Reaction to pre-defined time points without using threads. Instead it uses event handlers.

The event handler can react to internal events, interrupts and time events.

```
% rodos-executable.sh linuxMC timeevents.cpp
```

Combine it with other threads

```
% rodos-executable.sh linuxMC timewaitfor.cpp
timeevents.cpp
```

timeevent_resumer01.cpp & timeevent_resumer02.cpp

timeevent_resumer.cpp shows how a time event resumes a waiting thread. This other thread waits in a time-loop. Despite extra resumes, the beat does not get lost.

utctest.cpp

Shows how to use UTC time (time 0 is 1th of jan 2000 at 0:00) and calendar time.

(Calendar time is useless for internal computations but nice to communicate with people)

timepoints.cpp

Used to manage a (small) log file of time points and print them. Usable for debug and time tuning.

2.2. Priorities

preemptiontest.cpp

Two threads, one with a high priority which is executed very shortly every 3 seconds and one with a low priority which is executed constantly. The high priority thread (printing “*”) shall assume the CPU when it needs it even if the low priority thread (Printing “.”) does not suspend or yield.

See the application definition, which now uses an application ID (2000) to use the application discovery protocol with other applications (see ceiler).

```
% rodos-executable.sh linuxMC preemptiontest.cpp
% tst
```

ceiler.cpp

This is a demonstrator of priority ceiling to keep the CPU for a critical block. It will search for pre-emption application (using the application ID 2000). If not found it reports a warning.

Try:

```
% rodos-executable.sh linuxMC ceiler.cpp
% tst
% rodos-executable.sh linuxMC preemptiontest.cpp
ceiler.cpp
% tst
% rodos-executable.sh linuxMC preemptiontest.cpp
timeevents.cpp
% tst
```

Even if ceiler has a low priority it keeps the CPU when it is inside of the critical block in PRIORITY_CEILING (...). It will not be interrupted by other threads when it is inside of this block.

But time events and other events can not be blocked. They always run.

2.3. Local Communication

Local communication uses shared memory. To provide consistency in a concurrent system some means of synchronisation has to be used or else objects which can guarantee concurrent access without blocking threads. These objects are the means of communication between eventhandlers, and between eventhandlers and threads, because there we cannot suspend events.

commbufftest.cpp

Is a double buffer with only one writer and only one reader. Both can work concurrently.

The writer may write at any time. The reader gets the newest consistent data (eg. The last complete written record)

commbufftest-struct.cpp

Like commbufftest.cpp, but using more complex structures than just integers.

fifotest.cpp

For synchronous communication from one single writer to one single reader. Neither will be suspended. Writing to a full FiFo has no effect and returns 0. Reading from an empty FiFo returns 0.

2.4. Thread synchronisation

Thread synchronisation deals with suspending and resuming threads to synchronise their activities. To avoid concurrent access of critical sections semaphores must be used. To synchronise threads on data transfer use SyncFiFos.

syncfifotest.cpp

A sender and a receiver which wait until data is ready.

semaphoretest.cpp

Many threads try to access a protected area. Only one is allowed to enter at a time.

semaphore_worm.cpp

A (semi) graphical demonstration of semaphores. Several threads move with random speeds each a letter in a screen area (30x30 Characters). The right part of the screen is protected by a semaphore. Only one thread may move its letter in the right part, all other which want to enter have to wait until the one which occupies the semaphores exits the protected area.

Note: The terminal size has to be at least 30x30 characters (check the lower limit).

2.5. Events/Interrupts

With events you may react to interrupts like timers and signals from devices. You may need this when programming I/O drivers.

resumefromevent.cpp

A thread can be suspended and will wait until another thread that knows him resumes him. For example an event handler.

Following examples for interrupts: See tutorials/70-network-and-io/30-interrupts

externalevents01.cpp

It reacts to external interrupts. In the Linux simulation we have only one external interrupt and it is simulated by the Linux command “kill” or “killall”. Using kill we can generate an signal number 15 (Linux stuff).

Use two Linux windows. In one compile and execute externalevents01.cpp. Execute tst. In the other window call

```
% killall tst
```

externalevents02.cpp

Like externalevents01.cpp, but it uses the default event::handle, which searches for a thread waiting for this signal and resumes it.

2.6. Test programmes

yieldtime.cpp

Example how to use co-operative threads. It measure the time for thread switches in

nanoseconds.

```
% rodos-executable.sh linuxMC yieldtime.cpp
```

Now add some more threads and see the difference

```
% rodos-executable.sh linuxMC yieldtime.cpp
yieldtime_extra.cpp
```

or try just

```
% rodos-executable.sh linuxMC yieldtime_extra.cpp
```

3. Summary

File name	Content
ceiler.cpp	Use with preemptiontest.cpp, thread communication
gets-async-minimal.cpp	Receives data from interrupt
gets-async.cpp	Receives data
getsnowiat.cpp	Also receives data
helloworld-minimalistic.cpp	Prints ‘Hello World!’
preemption*	Several examples about pre-emption
resumefromevent.cpp	Thread gets resumed after short time
resumer.cpp	Similar to above
timeevent-resumer*	Thread gets resumed after some time

All other file names should be self explaining.