Real-time Linux (RTAI) LinuX Real-time - LXRT

Introduction

LXRT provides a **userspace interface** to the facilities and **features of RTAI**. It provides a **symmetric API** that may be used by both real-time RTAI tasks and Linux processes. LXRT is unique to RTAI and is one of its most useful features. The main advantages of LXRT is that the developer can safely develop a real-time task in soft userspace using all the user-space debugging tools, and be protected against system crashes when the application crashes. Finally after developing a task can be easily moved into kernel space.

Description

The idea of LXRT is already quit old and has under gone a lots of changes. However the main goal always has been the ability to program real-time tasks in user space with the same API as in kernel space.

The basic steps of evolution were :

1. RTAI started has an real-time OS were **RTAI real-time tasks** run in the **real-time scheduler** and any normal linux process is run by the **linux scheduler** in the time left over.
2. First LXRT was making the RTAI API available in userspace. It resolved in the ability of linux processes run by the linux scheduler to use the RTAI API just as the RTAI real-time tasks. However these processes were still SOFT real-time because they are still scheduled by the Linux scheduler and not the real-time scheduler. Linux processes configured to be able to use the RTAI API are called **LXRT SOFT real-time processes**.
3. Finally the people of RTAI managed to get an userspace linux process having access to the RTAI API to be scheduled by the real-time scheduler. Such process is called a **LXRT HARD real-time process** because it is scheduled in hard real-time by the real-time scheduler. The performance of a LXRT HARD real-time process is slightly less than that of a RTAI real-time task.

How do they compare?

- LXRT HARD/SOFT real-time processes run in the user mode privilege level and are therefore not allowed to address any memory in the kernel space which is protected by the hardware by an running in higher privilege level. So when making mistakes during development one cannot by accident mess up the kernel's own memory space and there causing a system crash.
- LXRT SOFT real-time processes have next to the RTAI API access to the full linux API. So for instance one can build GUI's in a SOFT real-time process like in a normal linux process, and still be able to interact with other hard real-time linux process by using the IPC functionality of the RTAI API. The price you pay however is that you have to give up HARD real-time.
- LXRT HARD real-time processes also run in user space and also have access to the full linux API. However when they call a function from the linux API which does a linux syscall it will temporarily be set back to a LXRT SOFT real-time process until the syscall is resolved by the linux kernel. So calling syscalls in a LXRT HARD real-time process will degraded its performance to that of a SOFT real-time process. It is therefore better for clarity and design issues to don't use linux API calls in a LXRT HARD real-time process. If such a process would need access to the Linux API it can than

instead communicate through the RTAI API with a LXRT SOFT real-time process to let it do the linux system call in its behave.
After all it is pure nonsense to use a non hard real time operation from within hard real time processes. When designing your system you should seperate you hard real-time parts from your soft real-time parts and never mixed them!

- By just a single RTAI API call can a LXRT SOFT real-time process be transformed in LXRT HARD real-time process and vice versa. The switch from soft to hard is just switching from the linux scheduler to the real-time scheduler and vice versa.
- However both LXRT HARD/SOFT real-time processes can not be transformed into RTAI real-time tasks. Vice versa is also not possible.

What extra overhead does LXRT processes have compared to RTAI real-time tasks because they run in user space?

1. Additional scheduler latency. The real-time scheduler runs in **kernel** space and does the rescheduling of a LXRT process running in user space to another LXRT process running a user space. So for every switch of a LXRT process an user-kernel-user round trip must be made and the MMU must be switched.
2. Additional latencies on RTAI API calls in LXRT. Using a semaphore, for example, in kernel space will always be faster than in userspace. This is because some additional microseconds are lost because each LXRT call is transfered to the kernel and the result back to user space. (the RTAI analog of a linux system call)

Recent measurements on modest hardware (Intel PII 300Mhz) showed that an additional 3us overhead on top of the standard scheduler latency of 3us. So we can conclude :

For most applications this slightly extra overhead is worth the benefits of LXRT.
Therefore from now on we will use LXRT HARD/SOFT real-time processes in userspace instead of the RTAI real-time tasks.

The biggest advantage of course that we can develop in soft real-time in which a buggy implementation won't take down the whole system, and were we can easily kill any wild running processes without a reboot.

Making a SOFT real-time linux process

To make the real-time behavior of a linux process better we can do two things :

1. Change its linux **schedulings policy** from SCHED_OTHER to SCHED_FIFO

   The Linux scheduler offers three different scheduling policies, one for normal processes and two for real-time applications. A static priority value sched_priority is assigned to each process and this value can be changed only via system calls. Conceptually, the scheduler maintains a list of runnable processes for each possible sched_priority value, and sched_priority can have a value in the range 0 to 99. In order to determine the process that runs next, the Linux scheduler looks for the non-empty list with the highest static priority and takes the process at the head of this list. The scheduling policy determines for each process, where it will be inserted into the list of processes with equal static priority and how it will move inside this list.

   SCHED_OTHER is the default universal time-sharing scheduler policy used by most processes, SCHED_FIFO and SCHED_RR are intended for special time-critical applications that need precise

control over the way in which runnable processes are selected for execution. Processes scheduled with SCHED_OTHER must be assigned the static priority 0, processes scheduled under SCHED_FIFO or SCHED_RR can have a static priority in the range 1 to 99. The system calls sched_get_priority_min() and sched_get_priority_max() can be used to find out the valid priority range for a scheduling policy in a portable way on all POSIX.1b conforming systems.

All scheduling is preemptive: If a process with a higher static priority gets ready to run, the current process will be preempted and returned into its wait list. The scheduling policy only determines the ordering within the list of runnable processes with equal static priority.

When making a process schedule with SCHED_FIFO it will be always run before any normal linux task scheduled with SCHED_OTHER

The linux function call we use for this is :

```
int sched_setscheduler(pid_t pid, int policy, const struct sched_param *p);
```

However we rarely use the above command because there exist a combined call to sched_setscheduler() and rt_task_init() which we use instead :

```
RT_TASK * rt_task_init_schmod(unsigned long name, int priority, int
stack_size, int max_msg_size, int policy, int cpus_allowed)
```

2. To **free** a linux process **from paging** to disk by locking the process memory. The linux function call we use for this is :

```
mlockall( MCL_CURRENT | MCL_FUTURE );
```

Making a LXRT SOFT real-time process

A LXRT SOFT real-time process is nothing else than the above soft linux process which Linux's task structure is extended by linking it to RTAI. This is done by calling **rt_task_init**. So for every thread we need to cal rt_task_init once.

Every RT_TASK in LXRT needs a `"Name"` to which it can be referenced from kernel and userspace. This name may be no longer than 6 alphanumeric characters and must be unique. Having names is an easy way to identify tasks and communication primitives when the program is running.
If you want automatic generation of `"Name"` use the function `rt_get_name(0)` to get a unique, valid, thread safe, name. You can always check yourself with `rt_get_addr("Name")` to see if a name is already taken.

NOTE :*You MUST make your thread an RTAI task from the moment you use in that thread any RTAI call. Examples are* `pthread_create_rt,rt_sem_init,...` *with the notable exception of* `rt_task_init()` RTAI/LXRT is an extension to your posix threads API, It is only logical that if you want to use these extensions, you need to register the thread with RTAI through `rt_task_init`.

making a LXRT SOFT process a LXRT HARD process

```
rt_make_hard_real_time();
```

making a LXRT HARD process a LXRT SOFT process

```
rt_make_soft_real_time();
```

Small API differences

In user space LXRT promises to have the same api as in kernel space. However some of the data structures in the API are stored in kernel space for usage for both API calls from both user as kernel space. It is however impossible to create such kernel space data structure from user space. Therefore the initializing of such structure goes with a named initialization and returned is the pointer to the data structure. The name is used as a key to the structure which is easy to remember by humans and can be used from both user space and kernel space because it is a value and not a pointer to some memory location. However it is important to remark again that the returned pointer of the structure cannot be address in user space because it points to kernel space!

E.g. for initializing a real-time task structure.
In **user space** you use :

```
RT_TASK* rt_task_init(unsigned long name, int priority, int stack_size, int
max_msg_size);
```

NOTE: normally you would used the rt_task_init_schmod(..) function in user space which is mentioned in the previous section..
In **kernel space** you use :

```
RT_TASK* rt_task_struct;
int rt_task_init(struct rt_task_struct *task, int priority, int stack_size, int
max_msg_size);
```

The same differences apply for *rt_sem_int*, *rt_mbx_int*, and *rt_cond_init* .

Example : Hello world!

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sched.h>


#include <rtai_lxrt.h>
#include <rtai_sem.h>
#include <rtai_msg.h>


static int thread0;

static void *fun0(void *arg)
{
    RT_TASK *task;

    task = rt_task_init_schmod(nam2num("TASK0"), 0, 0, 0, SCHED_FIFO, 0xF);
    mlockall(MCL_CURRENT | MCL_FUTURE);
    // makes task hard real time (default: soft)
    // uncomment the next line when developing : develop in soft real time mode
    //rt_make_hard_real_time();
```

```
    rt_printk("Hello World!\n");

    // makes task soft real time
    rt_make_soft_real_time();

    // clean task
    rt_task_delete(task);
    return 0;
}


int main(void)
{
    RT_TASK *task;

    // make main thread LXRT soft real-time
    task = rt_task_init_schmod(nam2num("MYTASK"), 9, 0, 0, SCHED_FIFO, 0xF);
    mlockall(MCL_CURRENT | MCL_FUTURE);

    // start real-time timer and scheduler
    rt_set_oneshot_mode();
    start_rt_timer(0);

    // create a linux thread
    thread0 = rt_thread_create(fun0, NULL, 10000);

    // wait for end of program
    printf("TYPE <ENTER> TO TERMINATE\n");
    getchar();

    // cleanup stuff
    stop_rt_timer();
    return 0;
}
```

Experiments

Use editor to edit and save the helloworld ! example as ***hello.c***
Use editor to edit and save the following Makefile

```
TARGET = hello
SRCS = hello.c

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

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

OBJECTS = $(SRCS:.c=.o)
CC = $(shell rtai-config --cc)
LXRT_CFLAGS = $(shell rtai-config --lxrt-cflags)
LXRT_LDFLAGS = $(shell rtai-config --lxrt-ldflags)

all: $(TARGET)

%.o: %.c
	$(CC) -c  $(LXRT_CFLAGS) $<
```

```
$(TARGET) :  $(OBJECTS)
	$(CC)  -o $(TARGET) $(LXRT_LDFLAGS) -llxrt $(OBJECTS)

clean:
	rm -f *.o  *~ $(TARGET)

.PHONY: clean
```

Since the make file above need to execute the program ***rtai-config***, which can be found at ***/usr/realtime/bin*** We have to add that directory to the PATH environment variable before building the hello.c by using the command:

PATH=/usr/realtime/bin:$PATH <ENTER>

export PATH <ENTER>

we can check if the new directory is added by using command

echo $PATH <ENTER>

We can now run make command:

make <ENTER>

Now we should have the hello.c compiled, so we should have the executatble file called ***hello***

Next, we have to add path to share library to the environment variable called LD_LIBRARY_PATH

LD_LIBRARY_PATH=/usr/realtime/lib:$LD_LIBRARY_PATH <ENTER>

export LD_LIBRARY_PATH

before executing the ***hello*** program, make sure that you have inserted the ***rtai_hal.ko*** and ***rtai_lxrt.ko*** module. If they are already inserted, we can execute the program by typing:

./hello <ENTER>

You have to open another terminal window for executing ***dmesg*** in order to see the result.

```
root@slax:~/Desktop/Lab3# ls
Makefile  hello.c
root@slax:~/Desktop/Lab3# PATH=/usr/realtime/bin:$PATH
root@slax:~/Desktop/Lab3# export PATH
root@slax:~/Desktop/Lab3# echo $PATH
/usr/realtime/bin:/usr/local/sbin:/usr/sbin:/sbin:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/usr/ga
s:/opt/kde/bin:/usr/lib/qt/bin
root@slax:~/Desktop/Lab3# make
gcc -c  -I. -I/usr/realtime/include -O2 -I/usr/src/linux//include -Wall -Wstrict-prototypes -pipe hel
.c
gcc  -o hello -L/usr/realtime/lib -lpthread -llxrt hello.o
root@slax:~/Desktop/Lab3# ls
Makefile  hello*  hello.c  hello.o
root@slax:~/Desktop/Lab3# LD_LIBRAY_PATH=/usr/realtime/lib/:$LD_LIBRARY_PATH
root@slax:~/Desktop/Lab3# export LD_LIBRARY_PATH
```