HOWTO build a simple RT application

The POSIX API forms the basis of real-time applications running under PREEMPT_RT. For the real-time thread a POSIX thread is used (pthread). Every real-time application needs proper handling in several basic areas like scheduling, priority, memory locking and stack prefaulting.

Basic prerequisites

Three basic prerequisites are introduced in the next subsections, followed by a short example illustrating those aspects.

Scheduling and priority

The scheduling policy as well as the priority must be set by the application explicitly. There are two possibilities for this:

1. **Using `sched_setscheduler()`**
   This funcion needs to be called in the start routine of the pthread before calculating RT specific stuff.
2. **Using pthread attributes**
   The functions `pthread_attr_setschedpolicy()` and `pthread_attr_setschedparam()`offer the interfaces to set policy and priority. Furthermore scheduler inheritance needs to be set properly to PTHREAD_EXPLICIT_SCHED by using `pthread_attr_setinheritsched()`. This forces the new thread to use the policy and priority specified by the pthread attributes and not to use the inherit scheduling of the thread which created the real-time thread.
3. **Problems with pthread condition variables**
   Multithreaded applications which rely on glibc's libpthread are prone to unexpected latency delays since its condition variable implementation does not honor priority inheritance (bugzilla). Unfortunately glibc's DNS resolver and asynchronous I/O implementations depend in turn on these condition variables. librtpi is an alternative LGPL-licensed pthread implementation which supports priority inheritance, and whose API is as close to glibc's as possible. The alternative MUSL libc has a pthread condition variable implementation similar to glibc's.

Memory locking

See here

Stack for RT thread

See here

Capabilities: running the app with RT priority as a non-root user

Several of the Pthread APIs, like `mlockall()`, `pthread_attr_setschedpolicy()`, by default and convention require root in order to successfully get their work done. Thus, RT apps - which need to set an RT sched policy and priority - are often run via `sudo`.

There's a far better approach to this; `sudo` gives the process root capabilities. This interests hackers 😃. Instead, you should leverage the powerful POSIX **Capabilities** model! This way, the process (and threads) get _only_ the capabilities they require and nothing more. This follows the infosec best practice, the *principle of least privilege*.

Apps start out with no capabilities by default; also note that capabilities are a per-thread resource (essentially translating to bitmasks with the task structure,which is per-thread of course). Among the various capability bits, the man page on `capabilities(7)` shows that `CAP_SYS_NICE` is the appropriate capability to use in this circumstance; a snippet from the `capabilities(7)` man page reveals this:

… **CAP_SYS_NICE**

- Lower the process nice value (nice(2), setpriority(2)) and change the nice value for arbitrary processes;
- **set real-time scheduling policies for calling process, and set scheduling policies and priorities for arbitrary processes (sched_setscheduler(2), sched_setparam(2), sched_setattr(2));**
- set CPU affinity for arbitrary processes (sched_setaffinity(2));
- set I/O scheduling class and priority for arbitrary processes (ioprio_set(2));
- apply migrate_pages(2) to arbitrary processes and allow processes to be migrated to arbitrary nodes;
- apply move_pages(2) to arbitrary processes;
- use the MPOL_MF_MOVE_ALL flag with mbind(2) and move_pages(2). …

Ok, great, but how exactly is this capability bit to be set on the app?

1. One approach is to do so programatically, via the `capget()/capset()` system calls. (Note that's it's generally easier to use the libcap library wrappers, `cap_[g|s]et_proc(3)`: https://man7.org/linux/man-pages/man3/cap_get_proc.3.html. This man page even provides a small example of doing so).
2. Another easy way is to leverage systemd and run your app as a service; in the service unit, specify the capability (see the man page on systemd.exec(5); https://www.freedesktop.org/software/systemd/man/systemd.exec.html#Capabilities.
3. Perhaps the easiest way: via the `setcap(8)` utility (it's man page: https://man7.org/linux/man-pages/man8/setcap.8.html). The setcap/getcap are typically part of the libcap package. For example:

```
sudo setcap CAP_SYS_NICE+eip <your-app-binary-executable>
```

You could put this line in the app Makefile (or equivalent). (The `getcap(8)` utility can be used to verify that the 'dumb-capability' binary now has the `CAP_SYS_NICE` bit set)!

And you're all set to run it as non-root now, a much more secure approach.

Example

```
/*
 * POSIX Real Time Example
```

```
 * using a single pthread as RT thread
 */

#include <limits.h>
#include <pthread.h>
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

void *thread_func(void *data)
{
        /* Do RT specific stuff here */
        return NULL;
}

int main(int argc, char* argv[])
{
        struct sched_param param;
        pthread_attr_t attr;
        pthread_t thread;
        int ret;

        /* Lock memory */
        if(mlockall(MCL_CURRENT|MCL_FUTURE) == -1) {
                printf("mlockall failed: %m\n");
                exit(-2);
        }

        /* Initialize pthread attributes (default values) */
        ret = pthread_attr_init(&attr);
        if (ret) {
                printf("init pthread attributes failed\n");
                goto out;
        }

        /* Set a specific stack size  */
        ret = pthread_attr_setstacksize(&attr, PTHREAD_STACK_MIN);
        if (ret) {
            printf("pthread setstacksize failed\n");
            goto out;
        }

        /* Set scheduler policy and priority of pthread */
        ret = pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
        if (ret) {
                printf("pthread setschedpolicy failed\n");
                goto out;
        }
        param.sched_priority = 80;
        ret = pthread_attr_setschedparam(&attr, &param);
        if (ret) {
                printf("pthread setschedparam failed\n");
                goto out;
        }
        /* Use scheduling parameters of attr */
        ret = pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
        if (ret) {
                printf("pthread setinheritsched failed\n");
                goto out;
        }

        /* Create a pthread with specified attributes */
        ret = pthread_create(&thread, &attr, thread_func, NULL);
        if (ret) {
                printf("create pthread failed\n");
                goto out;
        }

        /* Join the thread and wait until it is done */
        ret = pthread_join(thread, NULL);
        if (ret)
                printf("join pthread failed: %m\n");

out:
        return ret;
}
```