9 - Realtime tasks in Linux

TSEA81 - Computer Engineering and Real-time Systems

This document is released - 2014-12-15

Author - Andreas Ehliar

Lecture - 9 - Realtime tasks in Linux

This lecture discusses programs with real-time requirements in Linux through a case study of a stepper motor controller. Note that there is a video version of this lecture available on the examiners youtube page[1] which you might be interested in looking at. (Some information which was hard to see during the lecture is easier to see in that video.)

[1] https://www.youtube.com/watch?v=uIXkvz1-weQ

Example application - Stepper driver

A stepper motor is controlled by sending a periodic pulse train to the inputs of the motor. Note that if this motor is controlled by a microcontroller, it is also necessary to use a driver chip as the current requirements of a stepper typically exceeds the output current capacity of microcontroller pins. In the example in this lecture we will consider a (widely and cheaply available) stepper called 28BYJ-48 with a ULN2003 based driver.

To control a stepper motor it is necessary to control general purpose I/O (GPIO) pins on the system we are using. In Linux this can be done by for example using the sysfs GPIO interface using the following method:

- Identify the GPIO pin number we are interested in (by reading the manual/datasheet/schematics/etc)
- Write the GPIO number to the file /sys/class/gpio/export (this causes a directory /sys/class/gpio/export/gpioX to appear (where X is the number of the I/O-pin)
- This causes directory contains a number of files which can be used to configure the GPIO pin. Notably the file direction (which can be used to set the pin as an input or output), and value, which can be used to either set the value of the pin, or read the current value of the pin.

A short function that performs these operations can be seen in the following code listing:

```
// Initialize a GPIO pin in Linux using the sysfs interface
FILE *init_gpio(int gpioport)
{
        // Export the pin to the GPIO directory
        FILE *fp = fopen("/sys/class/gpio/export","w");
        fprintf(fp,"%d",gpioport);
        fclose(fp);

        // Set the pin as an output
        char filename[256];
        sprintf(filename,"/sys/class/gpio/gpio%d/direction",gpioport);
        fp = fopen(filename,"w");
        if(!fp){
                panic("Could not open gpio file");
        }
        fprintf(fp,"out");
        fclose(fp);

        // Open the value file and return a pointer to it.
        sprintf(filename,"/sys/class/gpio/gpio%d/value",gpioport);
        fp = fopen(filename,"w");
        if(!fp){
                panic("Could not open gpio file");
        }
        return fp;
}

// Given a FP in the stepper struct, set the I/O pin
// to the specified value. Uses the sysfs GPIO interface.
void setiopin(FILE *fp, int val)
{
        fprintf(fp,"%d\n",val);
        fflush(fp);
}
```

The advantage of using this interface is that it is rather portable. However, it is not very fast as all writes to the physical I/O pin has to pass through the kernel. However, if very low latency, or high throughput is required it may be necessary to use memory mapped I/O instead. This can be done by opening the special file /dev/mem and mapping it into the memory space of the current process with the system call mmap. The main drawback of this approach is that it requires the program to run with the privileges necessary to access /dev/mem (normally root). (Unrestricted access to the physical

memory of the computer breaks all security models in all operating system.) Another drawback is that it is by necessity less portable as it requires detailed knowledge of the hardware the program is running on.

First try: main loop with usleep() calls (steppero.c)

In the first version of the stepper control program the main loop looks as follows:

```
// Demo program for running a stepper connected to the Raspberry PI
// platform.
int main(int argc, char **argv)
{
        int delay = 1000; // Note: delay in us here
        FILE *pin0 = init_gpio(14);
        FILE *pin1 = init_gpio(15);
        FILE *pin2 = init_gpio(17);
        FILE *pin3 = init_gpio(18);

        signal(SIGINT, dumptimestamps);

        while(1){
                usleep(delay); logtimestamp(); setiopin(pin0,1);
                usleep(delay); logtimestamp(); setiopin(pin3,0);
                usleep(delay); logtimestamp(); setiopin(pin1,1);
                usleep(delay); logtimestamp(); setiopin(pin0,0);
                usleep(delay); logtimestamp(); setiopin(pin2,1);
                usleep(delay); logtimestamp(); setiopin(pin1,0);
                usleep(delay); logtimestamp(); setiopin(pin3,1);
                usleep(delay); logtimestamp(); setiopin(pin2,0);
        }
}
```

The problem with this version is that the calls to usleep() only guarantees that the program will sleep for at least this amount of time. However, the program does not take into account that the other parts of the main loop (i.e. the call to logtimestamp() and setiopin() may take up a certain amount of time as well). This means that it is hard to create a portable program that will always output data to the stepper with a fixed frequency.

Finally, you should note a signal handler is installed so that we dump the data collected by logtimestamp() to a file when the program is interrupted by a ctrl-c.

Second try: main loop with clock_nanosleep() calls (stepper1.c)

To make sure that we output data with the desired frequency we will implement a new function, sleep_until(). This function will take a struct timespec and a delay value (given in nanoseconds). It will then increment the time in the timespec by the desired number of nanoseconds and go to sleep until that point in time occurs. The implementation of sleep_until() is shown below:

```
// Adds "delay" nanoseconds to timespecs and sleeps until that time
static void sleep_until(struct timespec *ts, int delay)
{

ts->tv_nsec += delay;
if(ts->tv_nsec >= 1000*1000*1000){
ts->tv_nsec -= 1000*1000*1000;
ts->tv_sec++;
}
clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, ts,  NULL);
}
```

This means that sleep_until() will compensate for spurious variations in the running time of our main loop as long as the run-time of logtimestamp() and setiopin() is not excessive. The clock_nanosleep() function is used to perform the sleep operation with the CLOCK_MONOTONIC time source. This is a time source which is guaranteed to always increase, even if the date and time of the computer is modified for some reason. The TIMER_ABSTIME flag is used to ensure that we sleep to the specified time value. (If TIMER_ABSTIME is not specified, clock_nanosleep() will sleep for the specified amount of time instead, similar to how usleep() works.)

The second version of the stepper is a modified version of the previous version, where the calls to usleep() has been replaced by calls to clock_nanosleep(). It is also necessary to read the current time of the CLOCK_MONOTONIC time source at the start of the program so that sleep_until() has a starting reference point. The main loop of stepper1.c can be seen below:

```
// Demo program for running a stepper connected to
// the Raspberry PI platform.
int main(int argc, char **argv)
{
struct timespec ts;
unsigned int delay = 1000*1000; // Note: Delay in ns
FILE *pin0 = init_gpio(14);
```

```
FILE *pin1 = init_gpio(15);
FILE *pin2 = init_gpio(17);
FILE *pin3 = init_gpio(18);

signal(SIGINT, dumptimestamps);
clock_gettime(CLOCK_MONOTONIC, &ts);

while(1){
sleep_until(&ts,delay); logtimestamp(); setiopin(pin0,1);
sleep_until(&ts,delay); logtimestamp(); setiopin(pin3,0);
sleep_until(&ts,delay); logtimestamp(); setiopin(pin1,1);
sleep_until(&ts,delay); logtimestamp(); setiopin(pin0,0);
sleep_until(&ts,delay); logtimestamp(); setiopin(pin2,1);
sleep_until(&ts,delay); logtimestamp(); setiopin(pin1,0);
sleep_until(&ts,delay); logtimestamp(); setiopin(pin3,1);
sleep_until(&ts,delay); logtimestamp(); setiopin(pin2,0);
}
}
```

Third try: Ensure that a real-time scheduling class is used (stepper2.c)

When running stepper1.c we will typically get fairly deterministic performance, as long as no other processes are running at the same time on the Linux system. However, as soon as another process is running, the wait between two calls to setiopin() will have considerable variability. The reason for this is that stepper1.c is running as a normal process with no special privileges.

This can be seen by for example logging in repeatedly on the Linux machine and observing the movement of the stepper. It will appear to stutter for a brief moment whenever we login to the machine via SSH. (And the data collected by logtimestamp() will confirm this behavior.)

The solution to this problem is to run the program in a real-time scheduling class as opposed to the normal scheduling class in Linux. This is done by calling pthread_setschedparam() as seen below:

```
// Set our thread to real time priority
struct sched_param sp;
sp.sched_priority = 1; // Must be > 0. Threads with a higher
// value in the priority field will be schedueld before
// threads with a lower number.  Linux supports up to 99, but
        // the POSIX standard only guarantees up to 32 different
// priority levels.
```

```
// Note: If you are not using pthreads, use the
// sched_setscheduler call instead.
if(pthread_setschedparam(pthread_self(), SCHED_FIFO, &sp)){
fprintf(stderr,"WARNING: Failed to set stepper thread"
"to real-time priority\n");
}
```

(If you don't want to use pthreads in your application you could call sched_setscheduler() instead.)

Fourth try: Ensure that our memory is locked (stepper3.c)

By moving our process into a real-time scheduling class we will ensure that other processes cannot interfere with us by using up too much CPU time. However, there are no guarantees that they cannot interfere with us in other ways. More specifically, our program is not protected from the memory usage by other processes. This can be seen when running a program which allocates as much memory as possible, such as the following program:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char **argv)
{
int i;
for(i=0; i < 500; i++){
char *tmp;
int j;
tmp = malloc(1024*1024);
if(!tmp){
printf("Cannot allocate memory\n");
exit(0);
}

// Ensure the pages are actually allocated!
for(j=0; j < 1024*1024; j+=1024){
tmp[j] = 1;
}
printf("Allocated memory (%d MiB)\n",i);
}
return 0;
}
```

When this program is run, the stepper program will run as normal until eatmem allocates too much memory. At this point the stepper program, or parts of it, are likely to be paged out to the swap area, leading to severly reduced performance where the stepper simply stops for a certain amount of time.

(Although it is not guaranteed whether this will happen before Linux kills eatmem.c due to the out of memory condition.)

Note that in Linux it is typically not enough to merely allocate memory using malloc, it is also necessary to modify the pages we have allocated to ensure that Linux knows that we are actually going to use this memory.

This feature is called "memory overcommit" and is included in Linux because many programs allocate memory which they are never going to use. As such it is wasteful to actually reserve this memory before it is actually used. However, in an embedded system or real-time system this behavior can be turned off or modified (see the overcommit accounting documentation in Linux[2] for more information).

[2] https://www.kernel.org/doc/Documentation/vm/overcommit-accounting

This can be solved by locking the pages of the stepper program in memory by using the mlockall() function as follows:

```
// Lock memory to ensure no swapping is done.
if(mlockall(MCL_FUTURE|MCL_CURRENT)){
        fprintf(stderr,"WARNING: Failed to lock memory\n");
}
```

This ensures that all current and future memory allocations are guaranteed to be locked into memory and should never be swapped out to the swap area or otherwise made unavailable.

Linux is not a true real-time operating system

While the method described above will be sufficient for most use cases, it should be noted that Linux is not designed as a real-time operating system from the beginning. As such, Linux do not provide any guarantees about the actual performance of a real-time task. There are however various methods by which the performance of Linux can be made more deterministic. One of the most interesting methods is to use the PREEMPT_RT patch[3]. Among other things, this reduces the latency of interrupt handlers by moving almost all of the work traditionally performed by interrupt handlers into kernel threads.

[3] https://rt.wiki.kernel.org/

Interacting with a real-time thread from a low-priority thread

In many cases it is necessary to interact with a real-time thread from a lower priority thread. This is a dangerous situation which, if not done carefully, can lead to non-deterministic real-time performance. Consider for example the stepper control program, where it is desired that the user should be able to change the delay value (in order to change the speed of the stepper).

It is fairly obvious that the following piece of code will not work correctly as the call to scanf() will block the main thread until the user inputs a new delay vaule.

```
while(1){
sleep_until(&ts,delay); logtimestamp(); setiopin(pin0,1);
sleep_until(&ts,delay); logtimestamp(); setiopin(pin3,0);
sleep_until(&ts,delay); logtimestamp(); setiopin(pin1,1);
sleep_until(&ts,delay); logtimestamp(); setiopin(pin0,0);
sleep_until(&ts,delay); logtimestamp(); setiopin(pin2,1);
sleep_until(&ts,delay); logtimestamp(); setiopin(pin1,0);
sleep_until(&ts,delay); logtimestamp(); setiopin(pin3,1);
sleep_until(&ts,delay); logtimestamp(); setiopin(pin2,0);

printf("Enter new delay value: ");
scanf("%d",&delay);
}
```

(In addition, the call to printf() may exhibit non-deterministic behavior, depending on what the stdout file descriptor is connected to.)

One solution to this is to use so-called non-blocking I/O. Another, more general solution is to move all user interface tasks to another thread which does not run with real-time priority.

In this situation it is necessary to protect the data shared between the real-time task and the user interface task with for example semaphores (or locks). However, it is undesirable that the real-time task should be blocked by the low priority task when the user interface task is updating the shared data. One way to avoid this is to use the sem_trywait() in the real-time task which will return immediately, regardless of whether it managed to decrement the semaphore or not. However, this function will return 0 if it manages to decrement the semaphore.

In the real-time thread, the following code can be included, where the step variable points to a structure which, among other things, contains the new delay value written by the user interface task:

```
// Check for commands from user interface thread
```

```
        if(!sem_trywait(&step->sem)){
                target_delay = step->target_delay;
                // Note: Do not decrement step->sem here.
                // The user interface is responsible for
                // incrementing step->sem when it has
                // a command for the real-time stepper thread
                if(step->doexit){
                        // Set the I/O pins to 0 to ensure that
                        // the stepper do not draw power when
                        // being idle.
                        setiopin(step,0,0);
                        setiopin(step,1,0);
                        setiopin(step,2,0);
                        setiopin(step,3,0);
                        pthread_exit(0);
                }else if(step->getrpm){
                        step->getrpm = 0;
                        step->num_steps = num_steps;
                }

                // Acknowledge command:
                sem_post(&step->rt_sem);
        }
```

If the doexit flag is set,the thread should exit and if the getrpm flag is set, the stepper returns the current number of steps to the user interface task.

The user interface thread, capable of controlling two different steppers, looks like the following:

```
void *user_interface_thread(void *arg)
{
        struct stepper *step = (struct stepper *) arg;
        while(1){
                char buf[80];
                printf("Enter command (q, d or p): ");
                fgets(buf,79,stdin);
                if(buf[0] == 'q'){
                        // Exit
                        step[0].doexit = 1;
                        step[1].doexit = 1;
                        sem_post(&step[0].sem); // Notify RT thread
                        sem_post(&step[1].sem); // Notify RT thread
                        // No need to wait for acknowledgment before exit...
                        pthread_exit(0);
```

```
                }else if(buf[0] == 'd'){
                        int delay;
                        int stepno;
                        if(sscanf(buf,"d %d %d", &stepno, &delay) == 2){
                                step[stepno].target_delay = delay;
                                sem_post(&step[stepno].sem);
                                sem_wait(&step[stepno].rt_sem);
                        }
                }else if(buf[0] == 'p'){
                        step[0].getrpm = 1;
                        step[1].getrpm = 1;
                        sem_post(&step[0].sem);
                        sem_post(&step[1].sem);
                        sem_wait(&step[0].rt_sem);
                        sem_wait(&step[1].rt_sem);
                        printf("Current stepval for stepper 0 is %d\n",step[0].num_steps);
                        printf("Current stepval for stepper 1 is %d\n",step[1].num_steps);
                }
        }
}
```

In essence, the user interface owns the shared data until it notifies the appropriate real-time thread of a change by using sem_post on the semaphore. After this point it may not modify the shared data until the real-time thread acknowledges the change via the rt_sem semaphore. This is a variant of the symmetric synchronization protocol discussed in Lecture 3 - *Task synchronization*.

Source code availability

The source code of stepper0 - stepper3 and eatmem is available for download here under the X11 license:

- stepper0.c[4]
- stepper1.c[5]
- stepper2.c[6]
- stepper3.c[7]
- stepper.c[8] [
- eatmem.c[9]

[4] lectures/linux_realtime/stepper0.c

[5] lectures/linux_realtime/stepper1.c

[6] lectures/linux_realtime/stepper2.c

[7] lectures/linux_realtime/stepper3.c

[8] lectures/linux_realtime/stepper.c

[9] lectures/linux_realtime/eatmem.c