ECE 4220

Zachary Rump

2017-03-21

Lab 4 Report

Objectives

The primary objectives for Lab 4 were to learn about inter-process communication (IPC) and syncronization using pipes and semaphores.

Lab Description

The specific objectives of the Lab:

1. Learn about the differences between named and unnamed pipes.
2. Get more experience with real time tasks and syncronization.
3. Introduce the serial port on TS-7250.
4. Learn about communication between kernel modules and userspace.

Implementation

The implementation of this lab can be broken down into multiple components. The first component was provided as a binary, and simulates the device sending gps data. It consists of a test program (test_serial) that sends a different uint8_t over the TS-7250 serial port every 250ms. The second component is a kernel module running on the TS-7250. The kernel module creates a real-time fifo (under /dev/rtf) and initializes a real time task (75ms period) to check for a push button (B0) event. If the button is pressed, the real time task will get the current time and write the timestamp to the fifo. The third component, the main program, runs on the TS-7250, it reads data from the serial port and real time fifo, performs linear interpolation on the timestamps, and sends the data to a thread which writes all the information to stdout.

Flowcharts

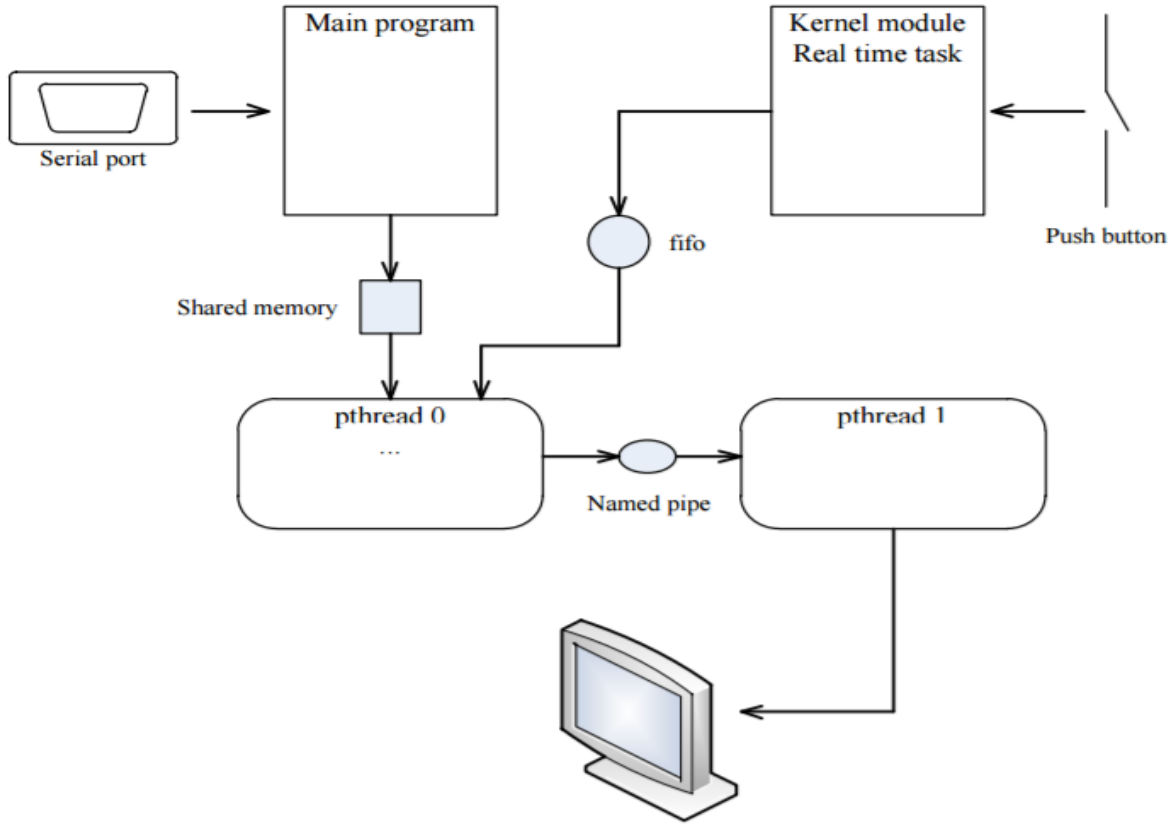

Figure 1. A basic flowchart of the program

Figure 1. Flowchart from the Lab 4 document.

Experiments and Results

The program as a whole was tested by testing each component individually in order. The serial communication was setup first, then the kernel module responding to the push button, then the dynamic threads reading from the fifo, and finally the thread to print to print the output.

Discussion

The main issue encountered in the lab was probably misunderstanding which functions could be used in kernel modules. Got stuck trying to use system calls like open,write() before getting redirected to the real-time fifos from the prelab. The other thing that caused trouble was figuring out how to handle multiple push button events with the interpolation. The TA helped realize how to use dynamic threads, and how to determine the necessary number of threads based on the period of the fifo button event and the period of the serial port data. I.e. 250ms/75ms = 4 threads max. Main thing I learned was probably the role of kernel modules and the different functions available to them.

Post Lab Questions

- Imagine that you want to do the interpolation with more GPS points in order to have a more accurate result. What would be changed? What kind of buffer would you need? Why?
  - Possibly would increase the period of the task responding to the push button (gps event), and increase the size of the buffer, and the number of dynamic threads that are available to respond.

- When your user space program reads from the FIFO, is this a blocking operation or non-blocking operation? How do you know?
  - This is a blocking operation because the program doesn't continue executing until there is data to read.

Code

Note this is code incomplete. See the actual source code submission.

Main Program

```
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>
#include <fcntl.h>
#include <pthread.h>
#include "serial_ece4220.h"

#define DATA_MAX 256
#define FIFO_ID 1

typedef struct gps_st {
uint8_t data;
struct timeval tv;
} gps_t;

pthread_t tid;

// Common buffer for data and time
int i=0;
gps_t data_buffer[DATA_MAX];

void read_fifo(void *args) {
struct timeval tv_buf;
int fd = open("/dev/rtf/1", O_RDWR);
if(-1 == fd)
{
return;
}
while(1) {
int count = read(fd, &tv_buf, sizeof(tv_buf));
if(count < 0)
{
return;
}
else
{
// spin up thread on each button press to wait for the next data
printf("Button pressed\n");
printf("Kernel Event Time: %u.%ld \n", (unsigned int)tv_buf.tv_sec, tv_buf.tv_usec);
printf("Last GPS Event: Data: %d, Time: %u.%ld \n", data_buffer[i-1].data, (unsigned int)data_buffer[i
}
```

```
}
}

int main(int argc, char **argv) {
// Create thread to read from fifo (kernel)
pthread_create(&tid, NULL, (void *)read_fifo, NULL); //(void *)
// Attempt to open serial port
int port_id = serial_open(0, 0, 5);
unsigned char buf=0;
ssize_t num_bytes=0;
// Endlessly loop and read from serial port
while(1)
{
// Read into array at pos(i);
for(i=0; i<DATA_MAX; i++)
{
struct timeval *tv = &(data_buffer[i].tv);
num_bytes = read(port_id, &buf, 1);
if(-1 == num_bytes)
{
// Error read()
break;
}
data_buffer[i].data = buf;
// Get timestamp
int ret = gettimeofday(tv, NULL);
if(-1 == ret)
{
// Error gettime()
break;
}
//printf("Data: %d, Time: %u.%ld \n", data_buffer[i].data, (unsigned int)data_buffer[i].tv.tv_sec, dat
//fflush(stdout);
}
}
serial_close(port_id);
return 0;
}
```

Kernel Module

```
#ifndef MODULE
#define MODULE
#endif

#ifndef __KERNEL__
#define __KERNEL__
#endif

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/time.h>
#include <unistd.h>
```

```
#include <rtai.h>
#include <rtai_sched.h>
#include <rtai_fifos.h>

MODULE_LICENSE("GPL");

#define NUM_PERIODS 1000
#define FIFO_ID 1
RTIME period;
RT_TASK t1;
unsigned long *BasePtr, *PBDR, *PBDDR;  // pointers for port B DR/DDR

void rt_process(void *args) {
// Open named pipe (get fd) -- write only
// Opens for writing will block until a process opens it for reading
// Unless specify O_NONBLOCK or O_NDELAY
while(1)
{
//http://www.cs.uml.edu/~fredm/courses/91.308/files/pipes.html
struct timeval tv;
// Detect button press on PORT B0
int button_status = (*PBDR & (1 << 0));
if(0 == button_status)
{
printk(KERN_INFO "Button pressed!\n");
// Button pressed -> get time
 do_gettimeofday(&tv);
// Attempt to write timeval struct to fifo
int ret = rtf_put(FIFO_ID, &tv, sizeof(tv));
if(ret < 0)
{
printk(KERN_INFO "Error writing to rtfifo\n");
}
}
rt_task_wait_period();
}
}

int init_module(void) {
// Attempt to map file descriptor
BasePtr = (unsigned long *) __ioremap(0x80840000, 4096, 0);
if(NULL == BasePtr)
{
printk(KERN_INFO "Unable to map memory space\n");
return -1;
}

int ret = rtf_create(FIFO_ID, sizeof(struct timeval));

// Configure PORTB registers
PBDR = BasePtr + 1;
PBDDR = BasePtr + 5;
// Set push button as input
```

```
// button is PORTB0
*PBDDR &= ~(1 << 0);

// Start realtime timer
rt_set_periodic_mode();
// Task should 'go off' every 75ms
// 1 ms = 1000000 ns
// 75 ms = 75000000
// 1 ms
period = start_rt_timer(nano2count(1000000));

// Initialize rt task
rt_task_init(&t1, (void *)rt_process, 0, 256, 0, 0, 0);
rt_task_make_periodic(&t1, 0*period, 75*period);
rt_task_resume(&t1);

printk(KERN_INFO "MODULE INSTALLED\n");
return 0;
}

void cleanup_module(void)
{
rt_task_delete(&t1);
stop_rt_timer();

printk(KERN_INFO "MODULE REMOVED\n");
return;
}
```