Lab 4

Inter-process communication and synchronization, Dynamic Threads and RT Tasks

Objective

In this lab, students will learn about inter-process synchronization and communication. In order to achieve the objective, students are required to:

- Create a real time periodic task to check push button events and collect time stamps.
- Use named pipes to communicate between separate processes.
- Use semaphores as a form of synchronization between threads.
- Use simple pipes to communicate between different threads.
- Receive data from a separate process and match events with the received data (using a GPS device could be an application of this).

Prelab

Provide brief descriptions of the following semaphore functions in user space: sem_init, sem_wait, sem_post, sem_destroy. Don't forget to include the flowchart or pseudo code for the assignment.

Scenario

This lab simulates the situation when we want to find the exact location of a real time event. Imagine that there is a camera mounted on a helicopter, which takes snapshots (the events) at different times. A GPS device provides location data every **250ms**. In order to find the exact location of each snapshot, we need to interpolate two GPS data values, one before and one after the event happens.

Lab Procedure

Figure-1 shows the basic flow of this lab assignment. A user space program (*Process-1*) needs to be created that consists of a main loop that receives data through a named pipe (N_pipe1) from an independent process (GPS_device – sending simulated GPS data), collects a time-stamp and saves both pieces of information in a buffer. Two other *pthreads* should also be created within *Process-1*. The job of the first pthread (Pthread0) is to read data (the time-stamp of the real time event) from another named pipe (N_pipe2), and to store the GPS data (provided by the main loop) before and after the real time event. The job of the second pthread (Pthread1 – the printing thread) is to receive the data (two GPS data pieces and their time-stamps, the time-stamp of the real-time event and an estimated GPS value) and print out the final results on the screen.

You need to create a second process (*Process-2*). It will initiate a real time task to check the real time events, collect time-stamps and send the data through N_pipe2.

The overall flow of the program is as follows (more details will be given further down):

1. The main thread in *Process-1* will loop, receiving GPS data through N_pipe1 and collecting time-stamps.

2. The GPS data received and the corresponding time-stamps should be saved in a shared memory buffer.

3. The real-time task in *Process-2* waits for button 1 on the auxiliary board to be pushed. When that is detected, a time-stamp is collected and it is passed to Pthread0 in *Process-1* through N_pipe2.

4. Pthread0 reads the time-stamp from N_pipe2. For each time-stamp received, a *child thread* will be dynamically created. The *child threads* need to collect the GPS data and time-stamps right before and right after the real-time event, estimate the GPS value for the real-time event through interpolation, and then send the final data through a simple pipe to Pthread1.

5. Pthread1 then receives the data through the simple pipe and prints out all the GPS values and time-stamps.

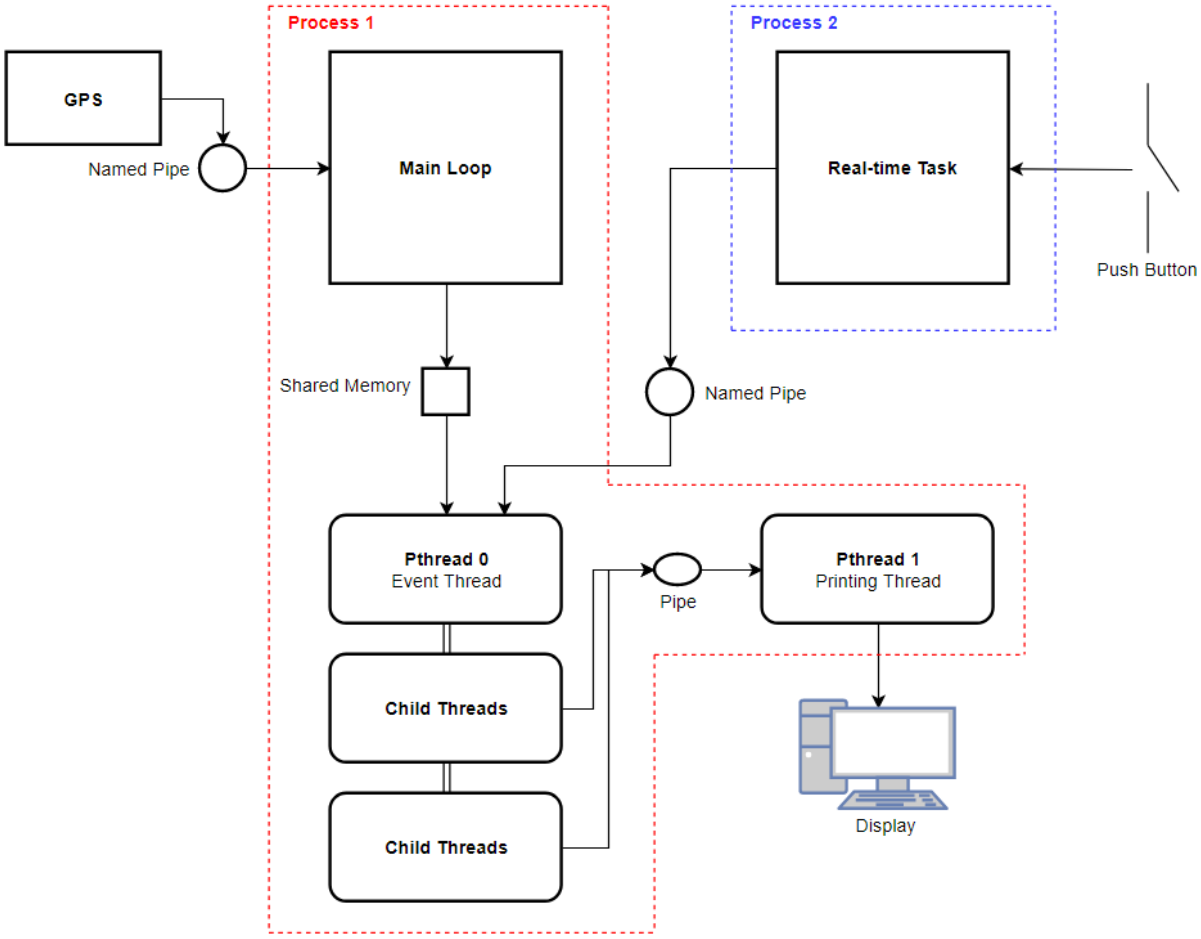

Figure 1: A basic overview of the lab assignment

This lab is broken down into **four parts** (and you will have two weeks to complete them).

Part 1: Main Loop and GPS data

In this part, you need to create the main loop of *Process-1*. The GPS_device executable file is provided. This program is the one that sends the simulated GPS data through a named pipe named N_pipe1 – one 8 bit integer value (unsigned char) every *250 ms*. **Note:** the GPS_device program will create N_pipe1 under **/tmp/**. Run the program using **sudo**.

First, make sure that the GPS data communication is working properly by running both GPS_device and a simple program that opens N_pipe1, reads and displays what is received. Next, get a time-stamp for each GPS data received. **Hint:** open separate terminals to run each of the programs.

The main loop of *Process-1* will continuously get the GPS data and time-stamps (what you did above), and it will place those pieces of information in a common buffer. Pthread0 of *Process-1* needs to be created, and it should have access to the buffer. Print the data to the screen from Pthread0, just to make sure that you are getting them correctly (printing here will eventually be removed).

Part 2: RT Events and Named Pipe 2

A second named pipe (N_pipe2) needs to be created for communication between *Process-1* and *Process-2*. *Process-2* will initiate a periodic real time task, with a period of *60 ms*. This task should check if button 1 (*GPIO port 16*) has been pushed (this will be referred to as the real-time or push button event). If a real-time event is detected, the task should collect a time-stamp and write it to N_pipe2.

As in Lab 3, you will need to install a kernel module and link a static library to your program, so you can use the check_button and clear_button functions. The files needed are the same: **libece4220lab3.a** (or **libece4220lab3_cpp.a**), **ece4220lab3.h**, and the module **ece4220lab3.ko**. You will have access to a second module, named **ece4220lab4.ko**, which detects when the GPIO port value is high (that is, when the button is pushed), rather than when there is a transition from low to high (when the button is being pushed). This allows continuous real-time events to be detected when the button is left pushed. The TA will give you details about the experiments that you will need to conduct using both kernel modules.

Note: Don't forget to configure the corresponding GPIO port as input, and to enable the pull down resistor.

Part 3: Pthread0, Dynamic Pthreads, and Interpolation

In this part, you need to modify the Pthread0 that you created in Part 1. The purpose of this thread is to receive data from *Process-2* through N_pipe2 and to read the latest GPS data and time-stamp from the common buffer, which is being updated by the main loop.

The final goal is to estimate the GPS value corresponding to the real-time (push button) events (their time-stamps are coming through the N_pipe2). Five pieces of information are needed for that estimation: the real-time event's time-stamp, the "previous" and the "next" GPS data, and their corresponding time-stamps. Using those pieces of

information, you could interpolate the GPS value of the real-time events. To get the "next" GPS data, Pthread0 may need to wait a while for it. This brings up an issue...

What happens if the messages are **not 1:1**? In other words, what happens if there is more than one push button event between two consecutive GPS events? Your program should be able to handle those multiple push button events (between consecutive GPS events). This is where the dynamic (child) threads come into play. Every time a time-stamp comes through N_pipe2, a new thread needs to be created (in Pthread0) to handle that particular event. That way, Pthread0 can go right back to checking the named pipe, while the child thread waits for the "next" GPS data.

Once the "next" GPS data comes, the *child threads* should have access to the five pieces of information needed. They can then do the interpolation to estimate the GPS value of the real-time events that they are in charge of. At last, they can display the final output required for each push button event: the time-stamp of the "previous" GPS event and its value, the push button event's time-stamp and its estimated GPS value, and the time-stamp of the "after" GPS event and its value.

Part 4: (Optional, for 10 Extra Credit points): Simple Pipe and Printing Thread

A printing thread (Pthread1) should be created to display the final output for each push button event. That is, the dynamic threads will do the interpolation, but they will not print the results themselves. Instead, they will send the six final pieces of information described above through a simple pipe to Pthread1. Once the data is received, Pthread1 will print them.

The TA will give you more details in the lab session.

NOTE 1: Before running your user space programs, make sure that the kernel module has been installed.

NOTE 2: In Figure-1, only a couple of dynamic (child) threads are shown. You might need more *pthreads*, *buffers*, *semaphores*, etc. in order to complete the assignment.

Experiments and Post Lab Questions

1. Given the specified periods for the RT task and the incoming GPS data, up to how many dynamic threads do you expect to be created between two consecutive GPS events? Make sure that your program can handle all events. Test using both kernel modules. Try pushing the button multiple times really fast, and try keeping the button pushed for a few seconds. Report your observations.

2. Imagine that you want to do the interpolation with more GPS points in order to have more accurate result. What would be changed in your program? What kind of buffer would you need? Why? You don't need to implement this.

3. When your program reads from the pipes, is that a blocking operation or non-blocking operation? How do you know?