Advanced Software Development for Robotics Assignment set 2 — Timing in Robotic / Cyber-Physical Systems

Group10: Tongli Zhu(s3159396), Thijs Hof (s3099571)

Statement regarding use of AI tools

1. Code: In the Sigwait part, we used ChatGpt to help debugging. It helped us find out the cause of the problem because we unblocked the signal before sigwait and to understand the signaling mechanism.
2. Report: Section 2.2.1- particularity discussion, the ”reason” in the second paragraph was suggested by ChatGpt after we did not find the answer in the literature, it seems like a plausible reason to us.

Assignment 2.1: Timing of periodic non-realtime thread (Ubuntu)

2.1.1 Basic requirement of this task

1.Run the program twice: without and with extra processor load (from another program). Investigate the timing of the loop. Discuss any particularities. Does extra computational load significantly affect the timing performance?

We perform certain calculation tasks every 1ms in the Posix periodic thread, and compare the deviation between the Worst-Case Execution Time and the set Period (1ms) as shown in Figure 1.

Under the same computing load, the distribution of deviations with additional load (using Stress) is significantly more dispersed and the fluctuations become larger than the deviations without additional load. This indicates that under increased processor load, the system’s response to real-time tasks becomes less precise. And we observe that even when there is no load, the deviation can only be said to be relatively stable, but there is also some jitters. Especially after we conducted the 2.2 experiment, it better reflects the advantages of Xenomai.

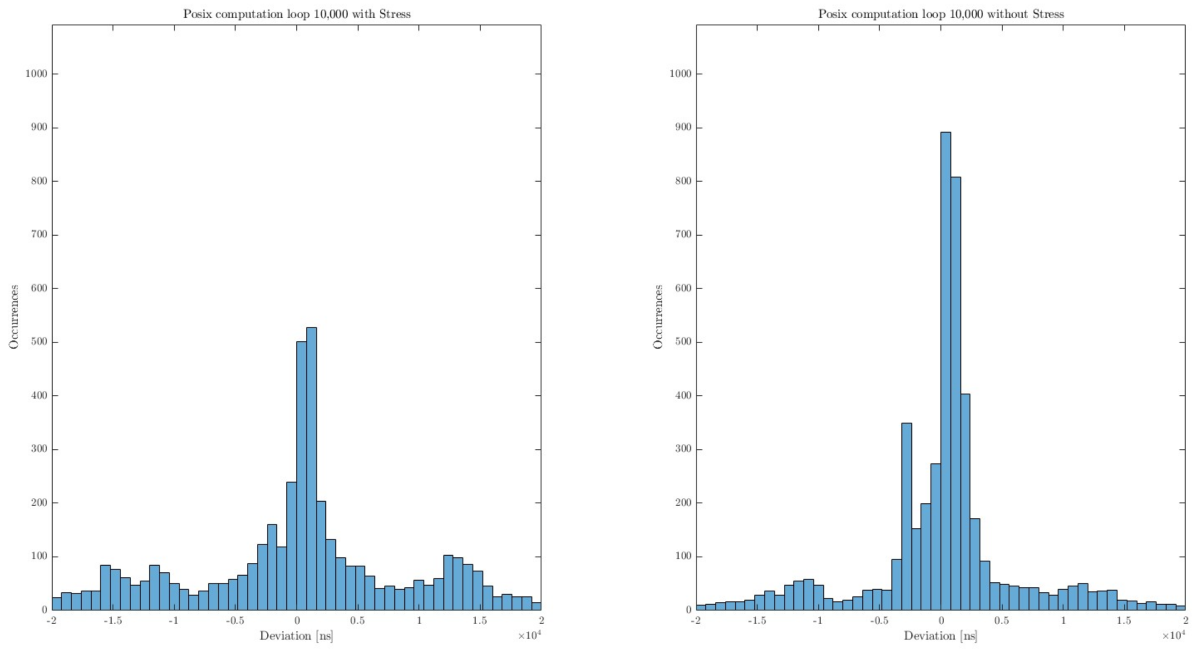


Figure 1: Comparison of the timing deviations of the additional processor load(left) and the normal computational load(right) with the same Posix frame

2.1.2 Questions

1.Explain what timing aspects we are interested in and why. Keep in mind that the course is on software development for robotics; a typical use case here is doing closed-loop control with a given sample time. Give at least two ways of representing the measured data. Discuss advantages and disadvantages of them.

In Periodic tasks the period time is fixed, it is taken as 1ms in this experiment. We are more concerned about Worst-Case Execution Time and Deadline (period end time). As shown in figure 2, if the end time of Worst-Case Execution Time is after deadline, it will cause problems and affect periodic tasks.

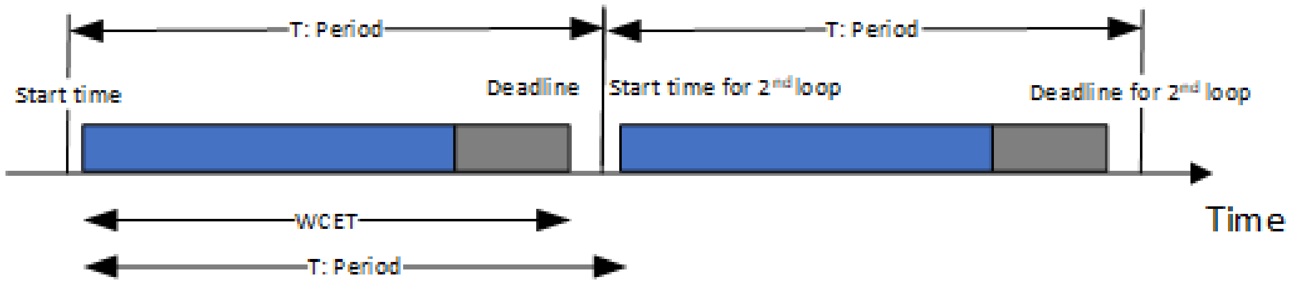


Figure 2: Periodic Tasks

The figure 3 shows the workflow of this experiment. We set that code blocks are executed periodically with a fixed sampling interval of 1ms. Use the following two methods for measurement:

1. **Measure the WCET of each loop:** By calculating $time2(i) - time1(i)$, that is, the time before and after the calculation work is performed within the loop, you can observe the time spent on each calculation work actual time. If the difference is less than 1ms, it means that the computational workload does not exceed the expected sampling interval. The advantage of this approach is that we can directly observe the actual time consumed by each execution. However, it doesn't tell us whether the next iteration of the loop will start on time.

2. **Measure the start time difference between consecutive loops:** By collecting the timestamp before each calculation starts and calculating the adjacent time difference $time1(i + 1) - time1(i)$, we can check whether the loop always starts every 1 ms as expected. The advantage of this method is that it ensures the constancy of the cycle time interval. So we chose this method in the experiment.

In practice, we first extract the time $time1$ and $time2$ through the timestamp, and perform a simple conversion to store them in the array. After the loop ends, we further calculate the time interval and write it to a TXT file. This avoids operations such as printing to the screen, which would interfere with real-time execution. For better visualization, we represent the deviation of the actual period from the desired period as the jitter. Of course, timestamps, time intervals, and deviations can also be called means of representing the measured data, depending on the visualization requirements.

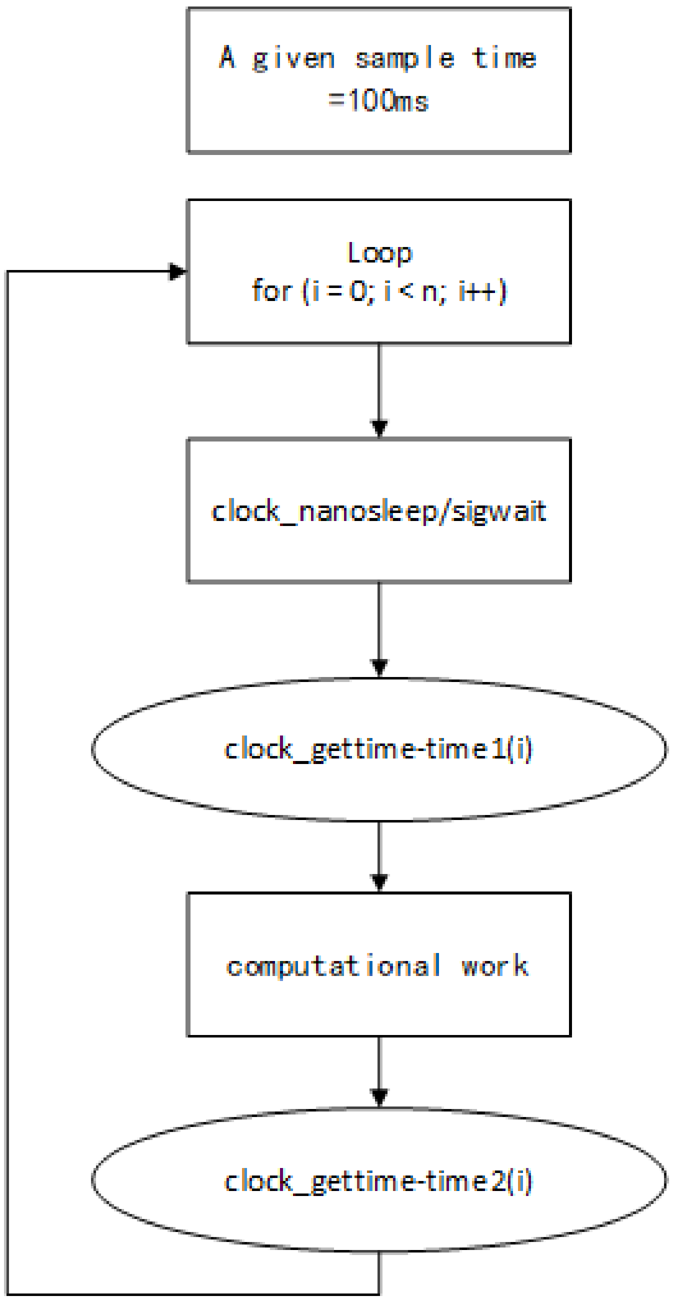


Figure 3: The workflow of representing the measured data

2.Explain why you have chosen the method you used (either `clock_nanosleep` or `timer_create/sigwait`). What are advantages and disadvantages of both methods?

We have tried both clock_nanosleep and sigwait.

The advantage of clock_nanosleep is that it allows the program to continue executing after a specified time. It is suitable for occasions where precise control of sleep time is required, that is, each sleep and wake-up of a thread is precise timetable.

But the shortcoming is that assuming a scenario, the starting time is 20ms, the thread wakes up every 1ms, and wakes up for the first time at 21ms. It takes 1.5ms to complete the calculation task, so it is completed at 22.5ms. At this time The 22ms of the second wake-up have been missed, so the thread will wake up immediately to execute the next loop. In this case, despite using clock_nanosleep, the execution of the task will still be affected by the execution time of the previous task, and the execution frequency cannot be accurately controlled. In addition, clock_nanosleep may return early when it receives a signal during sleep, so this situation should be considered manually in program.

The advantage of Sigwait is that it creates a timer. When the time set by the timer arrives, it will send a signal, and the sigwait function is used to block the thread before one or more signals arrive. When the signal arrives, sigwait will return, and the thread will unblock and continue executing subsequent code. This method ensures that signals are not lost if they are not processed in a timely manner, as they will be queued for processing.

But the shortcomings are similar to clock_nanosleep. Assume the same scenario as above, a signal arrives at 21ms, but due to the processing delay caused by the large amount of calculation in the loop, the second signal actually arrives at 22.5ms. Even if there is a delay in processing the first signal, the second signal is not lost but waits in the queue. When the first signal processing is completed, the program will process the second signal that should arrive in 22ms at the 22.5ms time. Although the signal will not be lost, real-time performance cannot be guaranteed.

We chose to use timer_create/sigwait for our program because in this way we can set a timer once, and it will keep running at the set interval. If we would have used clock_nanosleep the next wakeup time would need to be calculated for each period.

3.What is the difference between `CLOCK_MONOTONIC` and `CLOCK_REALTIME`? Which one is better for timing a firm real-time loop?

1. `CLOCK_REALTIME` represents the actual time (wall-clock time), which is based on international standard time and can be affected by system time settings, such as users manually changing the system time. If we use `CLOCK_REALTIME` for time measurement, any adjustments to the system time will affect the time measurement results.

2. `CLOCK_MONOTONIC` represents the time that has passed since the system started and will not be affected by the adjustment of the system time. It is only reset on system startup, which makes it ideal for measuring time intervals.

Take a typical example - daylight saving time. The time points were recorded using `CLOCK_REALTIME` and `CLOCK_MONOTONIC` respectively at 1:59:50 am before the daylight saving time adjustment, and then again at 3:00:10 after the daylight saving time adjustment. In fact, from 1:59:50 to 3:00:10, using `CLOCK_REALTIME` is 1 hour and 20 seconds. However, using `CLOCK_MONOTONIC` the interval is 20 seconds, which is also the actual interval. Hence, for a firm real-time loop, `CLOCK_MONOTONIC` is more suitable because it provides a stable time source that will not be affected by changes in system time.

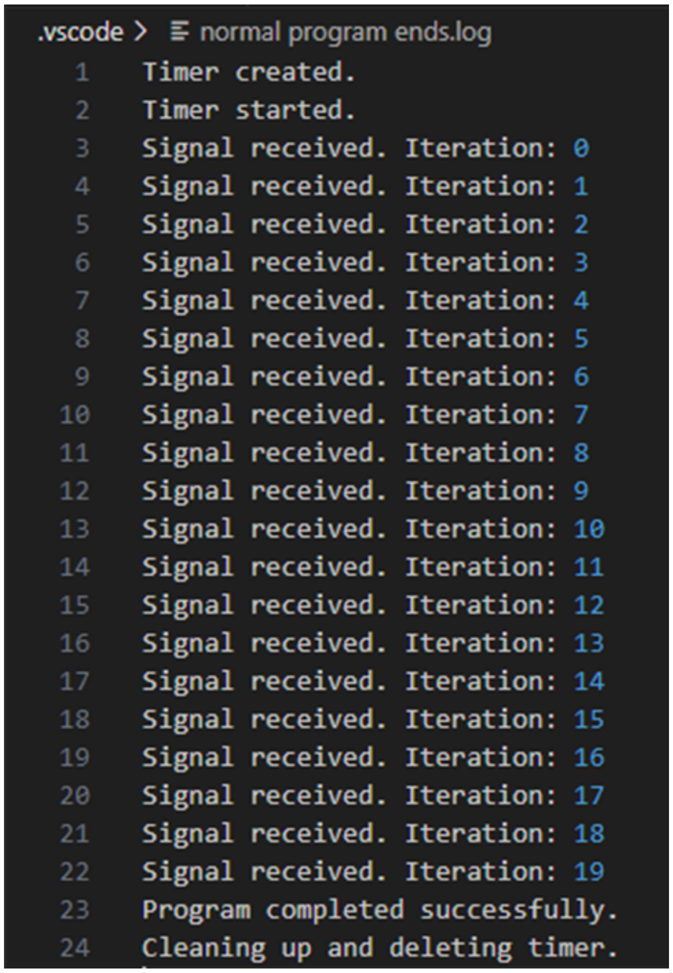
4. When using `timer_create/sigwait`, what happens with the timer you initialized when the program ends? And what happens when it crashes due to some error?

We wrote a script where we registered a cleanup function with `atexit` to be called when the program exits normally. This function is responsible for performing cleanup tasks such as deleting timers and closing log files. At the same time, we introduced logging into the script to track the running status and key events of the program, and simulated a crash caused by an error by performing an operation of dividing 1 by 0 when the loop reached the fifth time.

The results of running the program show as figure4 and figure5:

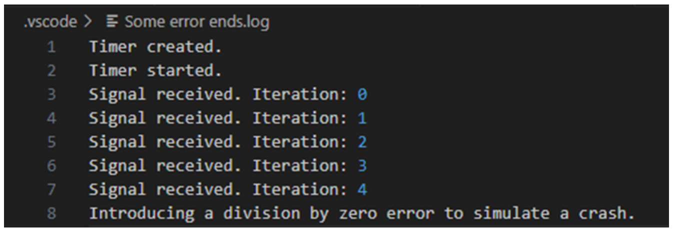
When the program ends normally: The program ends normally after all iterations are completed, and calls the cleanup function registered by `atexit` to delete the timer, which is implemented by calling `timer_delete`.

When the program crashes with an error: If the program crashes due to a runtime error (such as dividing by zero), it will not exit gracefully and the cleanup function registered by `atexit` will not be called. Therefore, the timer is not explicitly deleted. The log file records all events from the start of the program to the point of crash, including a message that heralds an impending crash. After a crash, there will be no new log entries, especially about timer deletion. Although the timer seems to remain in the system, the process is still terminated (albeit abnormally), and the operating system will still recycle timer, memory and other resources. The log when the program ends normally is shown in figure 4. The log in case of an error is shown in 5.



```
Timer created.
Timer started.
Signal received. Iteration: 0
Signal received. Iteration: 1
Signal received. Iteration: 2
Signal received. Iteration: 3
Signal received. Iteration: 4
Signal received. Iteration: 5
Signal received. Iteration: 6
Signal received. Iteration: 7
Signal received. Iteration: 8
Signal received. Iteration: 9
Signal received. Iteration: 10
Signal received. Iteration: 11
Signal received. Iteration: 12
Signal received. Iteration: 13
Signal received. Iteration: 14
Signal received. Iteration: 15
Signal received. Iteration: 16
Signal received. Iteration: 17
Signal received. Iteration: 18
Signal received. Iteration: 19
Program completed successfully.
Cleaning up and deleting timer.
```

Figure 4: The log when the program ends normally



```
Timer created.
Timer started.
Signal received. Iteration: 0
Signal received. Iteration: 1
Signal received. Iteration: 2
Signal received. Iteration: 3
Signal received. Iteration: 4
Introducing a division by zero error to simulate a crash.
```

Figure 5: The log when the program crashes with an error

5.Is it a good idea to use the approach used in this subassignment for closed-loop control of a robotic system? Why (not)?

If the closed-loop control of robotic systems can be ensured that the task processing time is much shorter than the control cycle, and the system's real-time requirements are not extremely strict, these methods can be considered. However, in situations where real-time requirements are very high, more specialized real-time methods such as Xenomai may be needed to ensure the accuracy and timeliness of control. The reason can be seen in the shortcomings part of the answer to the second question.

Assignment 2.2: Timing of periodic realtime thread (Xenomai)

2.2.1 Basic requirement of this task

Investigate the timing of the loop, that is, display the jitter in a diagram. Discuss any particularities. Does extra computational load significantly affect the timing performance?

We migrated the program to Xenomai. It can be seen from the experimental result diagram that under the same computing load, the jitter is very small, and almost all deviations are concentrated near zero, which shows that the periodicity (real-time) is very stable. The jitter of the POSIX calculation cycle is more obvious, the distribution of deviations is wider, and the timing fluctuations are larger.

Therefore, it can be proved that the additional computing load has a significant impact on the timing performance of the POSIX environment, while the EVL environment has better real-time performance and can handle the computing load more stably.

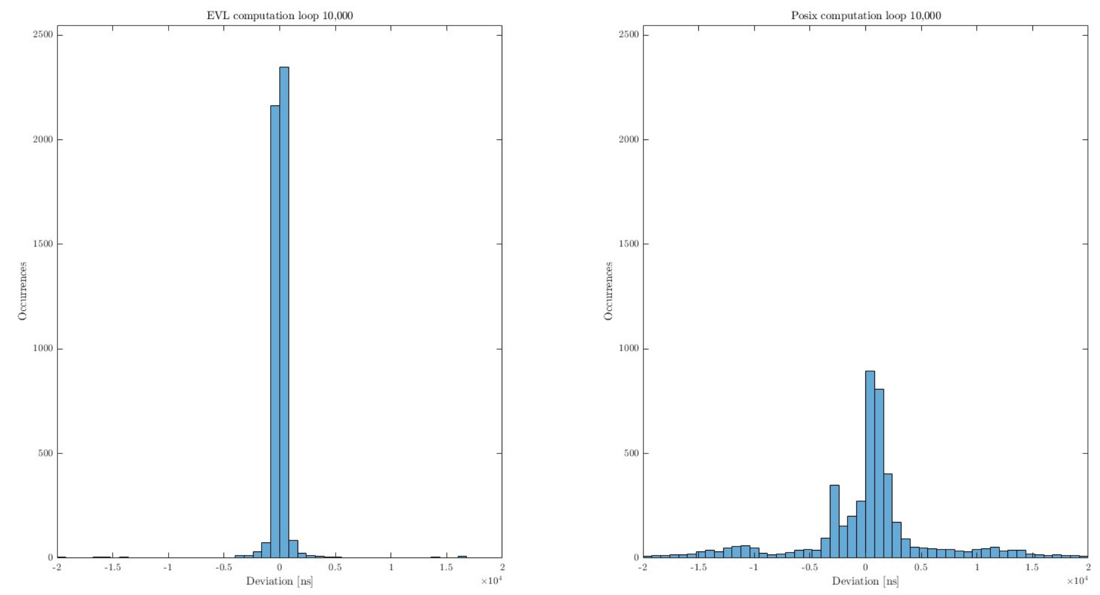
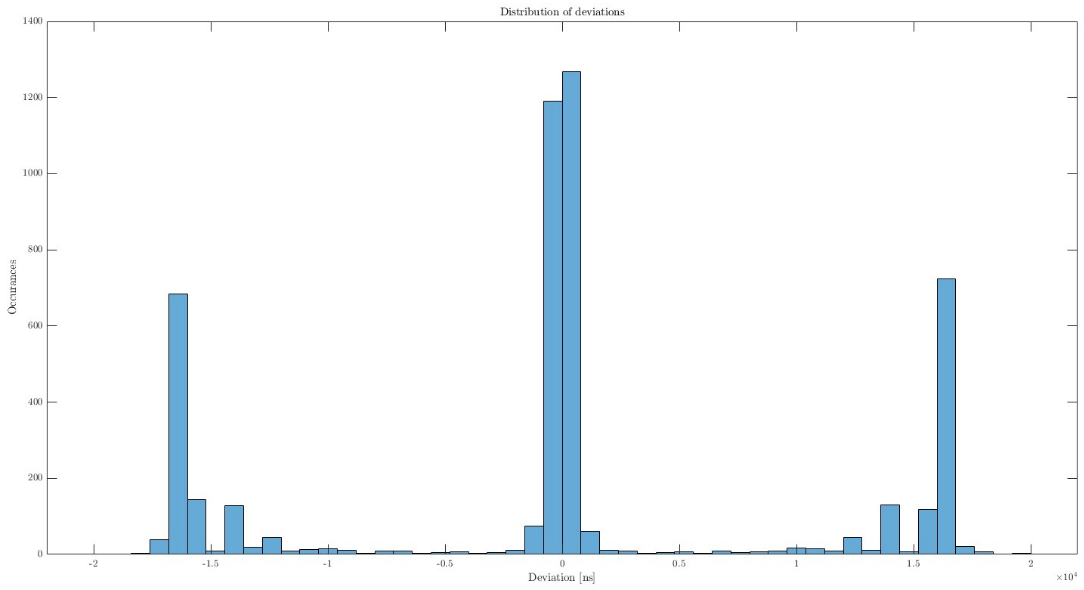


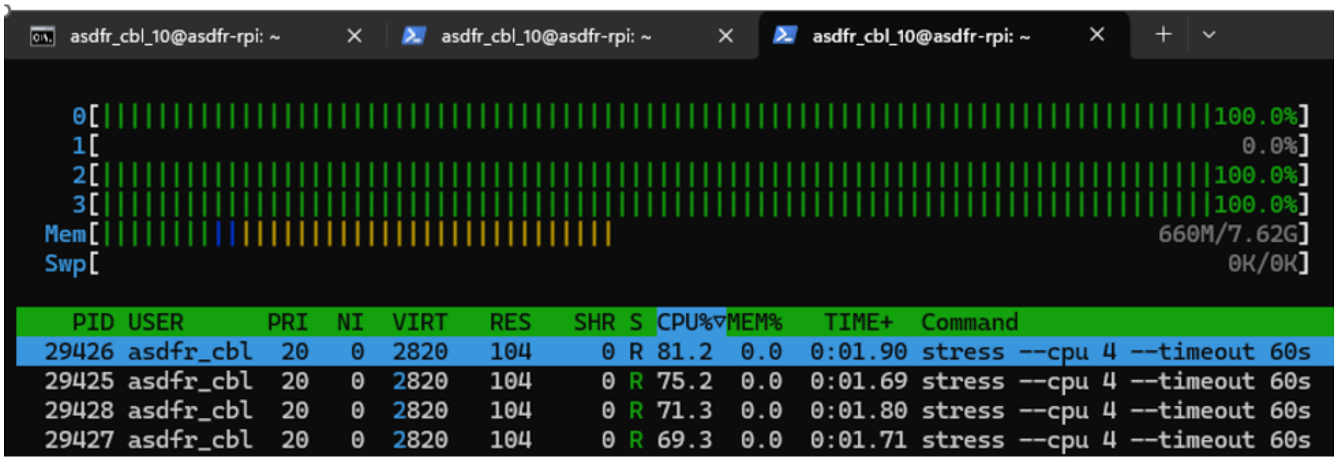
Figure 6: Comparison of the timing deviations of EVL(left) and Posix(right) with the same computational load

Particularity Discuss:

When we use the stress command, it only seems to affects the non-real-time parts of Linux, namely cores 0, 2 and 3. the Real-time tasks run on the reserved core 1 and which is not affected by stress.

But we found one situation that when we applied the current CPU load of cores 0, 2, and 3 to 100%, the real-time task on core1 also encountered delays, as shown in the figure 7. This might be because system resources such as memory bandwidth, I/O bus, cache, etc. are shared, if non-real-time tasks generate extreme memory access, even real-time tasks running on dedicated cores may be indirectly affected. (This reason was suggested by ChatGpt, we did not find the answer in the literature. It does seem like a plausible explanation to us.)




asdfr_cbl_10@asdfr-rpi: ~
asdfr_cbl_10@asdfr-rpi: ~
asdfr_cbl_10@asdfr-rpi: ~

```
    0[||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||100.0%]
    1[                                                                              0.0%]
    2[||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||100.0%]
    3[||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||100.0%]
  Mem[||||||||||||||||||||||||||                                           660M/7.62G]
  Swp[                                                                          0K/0K]

    PID USER      PRI  NI  VIRT   RES   SHR S CPU%▽MEM%   TIME+  Command
  29426 asdfr_cbl  20   0  2820   104     0 R 81.2  0.0  0:01.90 stress --cpu 4 --timeout 60s
  29425 asdfr_cbl  20   0  2820   104     0 R 75.2  0.0  0:01.69 stress --cpu 4 --timeout 60s
  29428 asdfr_cbl  20   0  2820   104     0 R 71.3  0.0  0:01.80 stress --cpu 4 --timeout 60s
  29427 asdfr_cbl  20   0  2820   104     0 R 69.3  0.0  0:01.71 stress --cpu 4 --timeout 60s
```

Figure 7: Particularity discuss

2.2.2 Questions

1.For each command from POSIX, EVL, and Xenomai 4 utilized in our program, a detailed explanation is provided below. Each description is articulated in a manner to illustrate the command's functionality and its necessity within a real-time programming context. Furthermore, when a command encompasses arguments or parameters that significantly impact the real-time behavior—for instance, the scheduling type `SCHED_X` or the clock type `CLOCK_X`—a thorough discussion is presented. This discussion aims to elucidate why the selected option is deemed optimal for enhancing real-time performance.

1. evl_attach_self: This function is used to attach the calling thread to the EVL core. It ensures that the thread runs under the control of the EVL core, which is central to real-time control.
2. evl_new_timer: This function is used to create a new timer in EVL. The timer is used for precise control of task execution time. The parameter used is EVL_CLOCK_MONOTONIC, which is not affected by system time changes, ensuring continuity and consistency.
3. evl_set_timer: This function is used to set the trigger time and interval of the timer, which is the foundation for implementing periodic real-time tasks.
4. oob_read: This function is used in EVL to wait for timer events, blocking the thread until the next timer trigger, similar to sigwait in POSIX, to trigger periodic tasks.
5. evl_read_clock: This function is used to read the time of the EVL clock, to be used for calculating the execution time and interval of tasks, used to detect the real-time performance of periodic tasks.
6. (POSIX)pthread_create: This is used to create a new thread. Because EVL does not actually create threads, POSIX is needed to create a real-time thread, to be used in combination with EVL functions.
7. (POSIX)pthread_attr_setschedpolicy and pthread_attr_setschedparam: These functions are used to set the thread's scheduling policy (SCHED_FIFO) and priority. The SCHED_FIFO policy ensures that higher-priority threads always run before lower-priority threads. The prioity is set to the highest available priority for SCHED_FIFO. But if there is only one thread in the experiment, and there are no other competing threads for core1, the effect of this policy and priority will not be apparent.
8. sudo taskset -c 1 ./executable file: We use this command to run the executable on core 1 because this core is reserved for real-time tasks.

2.Compare the timing performance of this program with the program from Section 2.1. What can you conclude on timing with respect to real-time capability?

The program using real-time frameworks such as EVL/Xenomai should exhibit better timing performance compared to the programs in Section 2.1. Because it provides a stricter time control mechanism and higher task scheduling priority, it shows higher anti-interference ability and reduces the jitter of calculations. We have proved that in 2.2.1.

3.Is it a good idea to use the approach used in this subassignment for closed-loop control of a robotic system? Why (not)?

Using a real-time framework (such as EVL or Xenomai) is a good idea for closed-loop control of robotic systems because these systems are able to provide the necessary real-time performance to ensure that control instructions are executed promptly and accurately. More specifically, real-time tasks run on specific RT-kernel parts, 'normal' tasks and RT tasks are separated, so the crash of standard OS does not affect RT tasks. But this mode needs careful division of tasks over 'normal' and RT part.

Besides that, if the non-real-time Linux system can already meet the real-time requirements of the application in some cases, there is no need to migrate to a more complex real-time framework.