4.

Review heap_mq.c and posix_mq.c. First, re-write the VxWorks code so that it uses RT-Linux Pthreads (FIFO) instead of VxWorks tasks, and then write a brief paragraph describing how these two message queue applications are similar and how they are different. You may find the following Linux POSIX demo code useful, but make sure you not only read and port the code, but that you build it, load it, and execute it to make sure you understand how both applications work and prove that you got the POSIX message queue features working in Linux on your Altera DE1-SoC or Jetson. Message queues are often suggested as a way to avoid MUTEX priority inversion. Would you agree that this really circumvents the problem of unbounded inversion? If so why, if not, why not?

ANSWER

The two codes, heap_mq.c and posix_mq.c were reviewed. They were understood, re-written so that they use RT-Linux Pthreads (FIFO) instead of VxWorks tasks, built, loaded and executed and a clarity was obtained on how both the applications work.

SIMILARITIES

Both functions:

- Give the receiver a higher priority than the sender.
- Ensure a safe way for the threads or processes to communicate with each other.
- Use tasks to run the sender and receiver functions.
- Use the posix queue for storing the messages with the function calls mq_open, mq_receive and mq_send.

DIFFERENCES

The first major difference between the two implementations is that heap_mq.c creates and opens the message queue before running the sender and receiver functions. The posix_mq.c code opens and creates the message queue in the receiver function and it is again opened in the sender function.

The other major difference is that heap_mq.c uses dynamically allocated memory to be used as the buffer, while posix_mq.c uses stack memory.

In message queue, the message transferred is the actual data and in the heap queue, the data transferred are actually pointers to messages. The pointers are set to point to a buffer allocated by the sender, and the pointer received is used to access and process the buffer. Hence in the heap queue, the queue is a buffer of pointers.

Heap_mq.c also implements the sender and receiver functions as infinite loops that must be closed by deleting the tasks, while posix_mq.c only runs through the sending and receiving functions once.

In message queue, it is important to ensure that the read and write to the buffer is atomic using some semaphore or mutex. Such a restriction is not necessary in case of heap queue.

For message queue the size of the message queue always remains constant which is determined when the queue is opened, but in case of the heap queue, the sender allocates the buffer and the receiver de-allocates to avoid exhaustion of the associated buffer heap.

Posix_mq

********** **Code Execution Screenshot and Brief Explanation** **********

Heap_mq

********** **Code Execution Screenshot and Brief Explanation** **********

********** **Implementation in Jetson – Failure – Explanation** **********

Message Queues – The solution to Unbounded MUTEX Priority Inversion?

Priority inversion is an issue when MUTEXes and semaphores are used, which do resource blocking in order to modify global data. Removing MUTEXes and semaphores is not feasible and advisable since global data is invariably required for communication between threads. An alternative to this would be to use Message Queues – they prevent the problem of unbounded priority inversion and provide a method for establishment of communication between threads and processes without locking resources. However, normal queues cannot always ensure the absence of priority inversion. POSIX message queues, to ensure that priority inversion does not take place, have a feature of queueing and servicing messages according to priority in a timely fashion. Message queues ensure that dequeue of messages takes place such that the one with the highest priority gets dequeued first. Thus, priority can be given to the messages in a way that priority inversion does not occur. The higher priority task, waiting on the lower priority task, waits for a deterministic amount of time, thus avoiding unbounded priority inversion. The enqueue and dequeue operations are thread-safe.