REAL TIME SYSTEMS FOR AUTOMATION

# HI-LOW BUFFER

Assignment 2

## Problem Analysis

The hi-low buffer problem is very similar to the classic bounded buffer problem and can be seen as a direct extension to it. In fact, both problems require managing a shared store to keep it consistent (i.e., no race conditions occur between threads that concurrently access and edit the shared store). While in the classic problem the items stored are homogeneous, this is not the case of the hi-low buffer since elements belong to one of two classes (HIGH and LOW). These two classes add two different levels of priorities between threads that interact with the shared buffer. A correct implementation has to provide a way to prioritize some threads over others without losing concurrency.

## Monitor Design

Following the analysis of the problem detailed above, the main idea behind the design of the monitor for the hi-low buffer problem is to start from the classic bounded buffer monitor and then provide different queues to threads with different priorities. To be more precise, the implementation employs a pair of condition variables (one for each level of priority) for signaling blocked threads that the buffer is no longer full. Another pair of condition variables (one for each level of priority as above) notifies threads that a new item is available. This set of condition variables allows for maximum concurrency while keeping the implementation simple. To better explain why this is the case, let's consider for instance the implementation of the download operation. It starts by checking if the buffer contains any items of the desired priority. If none are found, it blocks the consumer thread on the condition variable that corresponds to the requested item priority. In this way, when the producer thread inserts a new item of the requested priority, only the consumer threads interested in items with a priority similar to the one of the just inserted item can be awakened. This is very important since it allows the implementation to be way more efficient at signaling. Once unblocked, the consumer thread can continue with the

download by taking the inserted item out of the buffer. It then finishes by signaling to the producer threads that a new item is available. However, signaling in this case should be done carefully. The problem mandates that producer threads which have been blocked while uploading high priority items have to be awakened before threads with low priority items. Also in this case, the paired condition variable provides an elegant and simple solution. In fact, since there exists one condition variable per item type we know for sure that threads uploading high priority items are blocked on the high priority condition variable. So, emitting a signal on this condition variable first is sufficient in order to give precedence to high priority threads. However, if there are no high priority threads waiting, that signal will be lost and the resulting implementation would be highly inefficient. To solve this problem, the implementation uses a counter to track how many threads are waiting on each condition variable so that it can skip emitting a signal if there are no threads currently waiting for it.