Operating Systems 2INC0 Assignment 3

Zachary Kohnen (1655221) and Cansu Izat (1372416)

Eindhoven University of Technology

1 FIFO Design

1.1 Structure

We put all the important information about the FIFO in one structure **fifo_t**, and this structure contains the buffer, the used length and the next expected item. We implemented the FIFO using a singular continuous buffer rather than a circular buffer. We chose this kind of buffer as it is simpler to reason about. Even though the circular buffer would have a faster running time, with a small buffer size, the singular continuous buffer does not create a significant impact on the running time. We chose to have one mutex for the FIFO contents, and one read-write lock for each of the other fields. The use of read-write locks allows each producer thread to access the FIFO length and the next expected item concurrently — given that no thread has an active write lock on them.

1.2 Interface

In order to separate the FIFO implementation and the producer/consumer logic, we implemented two functions to push and pop items to/from the FIFO.

In our implementation of **fifo_push**, the function first checks whether the item being pushed is the next item expected. If the item being pushed is not the next item expected, the function returns **PUSH_NOT_NEXT** as defined in our **fifo_push_result_t** enum. The function the checks whether the buffer is full or not, and if the buffer is full, the function returns **PUSH_FULL** . If the previous checks passed, the item is added to the next open spot in the buffer, the length of the FIFO is incremented, and the expected next item is also incremented. If the function succeeds in the aforementioned steps, then it returns **PUSH_SUCCESS**.

In our implementation of **fifo_pop**, the function first checks if the FIFO is empty. If it is empty, then the function also checks if the next expected item is equal to the number of items. If this is the case, the function returns **POP_DONE** as defined in our **fifo_pop_result_t** enum. Otherwise, the function returns **POP_EMPTY**. If the buffer isn't empty, then the function pops the item from the FIFO, decrements the length of the FIFO, shifts every item in the FIFO to the left, and returns **POP_SUCCESS**.

2 Condition Variable Synchronization

2.1 Structure

We made two slightly different structures, one called **condvar_t**, and the other called **expecting_t**.**condvar_t** contains a simple boolean predicate, the condition variable and a mutex.**expecting_t** contains everything that **condvar_t** contains as well as an extra variable that details for which item this condvar depends on. We can use the extra variable in the **expecting_t** structure to ensure that this condvar is only signalled when the item it depends on is pushed into the FIFO.

2.2 Interface

For each of these two structures, we have implemented two functions, one that waits, and one that signals. The only difference between the interface of **condvar_t** and **expecting_t** is that the function **expecting_signal** ensures that the item that is expected is the same item that the condvar is waiting for a signal about. This implementation helps ensure that we do not signal a producer that would still be unable to push their item onto the FIFO.

2.3 Usage

The consumer only requires one **condvar_t**, to be notified when new items were pushed if the FIFO was empty when it previously attempted to pop from the FIFO. All producers can also share one **condvar_t** for any producer that received the **PUSH_FULL** result. Only one producer will ever wait on this condvar, and that is when this producer has the next item that is to be pushed when there becomes room in the FIFO. For every producer that receives **PUSH_NOT_NEXT**, they require their own **expecting_t** — which we store in an array, indexed by the producers' ID. By using the **expecting_t** structure, we can ensure they are woken only when they are able to be pushed onto the FIFO