Thread Message Queue Project

You may discuss this Thread Message Queue Project specification with others to understand the problem. You may use the Internet for point-specific questions. For example, you may search the Internet for how to create threads, semaphores, and condition variables. You **may not** use the Internet to discover thread message queue code that is copied or mimicked into your solution. Our OSTEP textbook explains Posix threads, this document provides a link to a Posix threads tutorial, we have discussed threads in lectures, and this document provides insight into the problem. Our OSTEM textbook explains mutex locks, condition variables, semaphores, bounded buffer problems, etc. and we have studied them in class. Use this material to design and implement your own code.

Use the file `ThreadMsgQProjectAnswers.docx` to answer questions in this lab.

Introduction

Chapter 27 of OSTEP discusses the Pthreads API. The following link is another discussion on Pthreads - https://computing.llnl.gov/tutorials/pthreads/. Chapters 28-32 of OSTEP discuss concurrent programming techniques that you may find useful in this solution. In particular, study the Producer Consumer (Bounded Buffer) Problem, which is discussed in detail in Chapter 30. The solution in Chapter 30 uses condition variables. Chapter 31 shows a Producer Consumer solution with a semaphore, which I think is a easier to understand solution.

Message Queue API and Data Structure

A thread-safe message queue data structure and API provides the ability for threads to exchange messages in a first-in-first-out order. The following is the API you must design and implement.

- `struct msgq *msgq_init(int num_msgs);` - initializes a message queue and returns a pointer to a `struct msgq`. The parameter `num_msgs` is the maximum number of messages that may be in the message queue. The returned pointer is used in the other API functions.
- `int msgq_send(struct msgq *mq, char *msg);` - places `msg` on message queue `mq`. `mq` is returned from `msgq_init`. `msgq_send` must make a copy of `msg` on the heap. If `mq` has `num_msgs` in it; then `msgq_send` blocks until there is room for `msg`. `msgq_send` returns 1 for success and -1 for failure.
- `char *msgq_recv(struct msgq *mq);` - returns a message from `mq`. `mq` is returned from `msgq_init`. The returned message is on the heap. The function that receives the message can `free` it when it is no longer needed.
- `int msgq_len(struct msgq *mq);` - returns the number of messages on `mq`. `mq` is returned from `msgq_init`. Due to the nature of threads and interrupts, the length returned may be incorrect by the time it is used.

- `void msgq_show(struct msgq *mq);` - displays all of the messages in `mq` to `stdout`. `mq` is returned from `msgq_init`.

You must design and implement `struct msgq`, which must involve dynamically allocated memory. My solution includes a linked list of messages, where `send_msg` mallocs memory for the data structure and places it on the end of the linked list. The data structure has a `char*` pointer to the message. The heap memory for the message is allocated using `strdup`. `recv_msg` removes the head from the linked list and returns the `char*` pointer to the message.

Implementation Guidance

You should design your solution first. During the design phase, you must select your thread primitives. For example, I decided to use semaphores, but you could solve it with condition variables and mutexes.

semaphore.h and zemaphore.h

MacOS does not support unnamed semaphores that are provided by semaphore.h. I have provided zemaphore.h and zemaphore.c, which implements semaphores using condition variables. OSTEP Chapter 31 provides the same (or similar) code. The following is how I include `semaphore.h` in my `msgq.c` and `main.c` files. You are welcome to make this fancier by using the `#ifndef __zemaphore_h__` style programming.

```
#include "zemaphore.h"
#include "msgq.h"
```

Testing

I provide a main.c that does a few tests. You must implement the following tests.

- Show that `send_msg` blocks when the message queue contains `num_msgs`.
- Show that `recv_msg` blocks when the message queue is empty.
- Solve a producer consumer problem with your message queue and five threads. Two threads are producers generating messages and three threads are consumers consuming messages. Each producer must generate at least 50 messages. You shall use the same function code for the two producers. An argument passed is used to select the messages that are generated. The consumers shall save their consumed messages in static arrays - one for each consumer. You shall use the same function code for the three consumers. An argument passed is used to select the static array to store the messages. Main shall `wait` for the two producers to finish and then `wait` an additional five seconds, at which point the three consumers should be blocked on `msgq_recv` calls. Main shall print the messages received in each consumer. Ensure that all 100 messages are consumed. Experiment to determine which consumers consume which messages.

Submissions

Be sure your submission uses the `ThreadMsgQProjectAnswers.docx` to answer the questions so I can easily decipher your answers in your submission.

1. Submit your code, makefile, and screenshots of your testings.