Daniel Benson (djbenson)
Michael Racine (mrracine)
CS3013

Project 4 README

Top Directory:

- Message_LKM (folder)
- Test (folder)
- README.pdf

Message_LKM:

- Mailbox_LKM.c
- mailbox.h
- Makefile

Test:

- mailbox.c
- makefile
- testmailboxX.c (X = 1-7)
- testoutputs.txt

*Note: The mailbox.h in the Message_LKM subdirectory is what we used when compiling the LKM...the one provided did not contain the “asmlinkage” type at the beginning of function declarations and definitions.

To begin our implementation of the Kernel Messaging System, we intercepted the 3 system calls cs3013_syscall1, cs3013_syscall2, and cs3013syscall3 to instead call the functions SendMsg, RcvMsg, and ManageMailbox, respectively. We created a mailbox struct which held key information for a mailbox, such as the PID it belongs to, the number of messages within it, its stop boolean, a reference counter (for the number of mailboxes waiting on it), its array of messages, a spinlock, and a locking waitqueue. Each message is implemented as a struct, which contains the string, the length of the message, and the sender of the message. Our hashtable is implemented as another struct which contains fields such as the current allocated size of the table, the actual number of mailboxes within its array, the array of mailboxes, and a main spinlock which controls access in changing elements within the hashtable. We also implemented functions for inserting into the hashtable, removing from the hashtable, and retrieving a mailbox from the hashtable.

SendMsg will insert a message into the desired mailbox by calling insertMsg, which creates a new message based on the string provided and inserts it into the mailbox provided in the destination parameter. Each message is allocated from a slab called message_cache. Once the message is allocated, its fields are populated, and added to the mailbox's message pointer array. Before creating the message, we have several statements checking to make sure the message does not exceed the maximum size, whether the stopped variable is true for the destination mailbox, and whether the mailbox is full. In the case that the mailbox is full and the block variable is set to true, insertMsg will wait until space becomes available via the mailbox's waitqueue. If the block variable is false, then the process will return with the MAILBOX_FULL error. If the destination mailbox is currently stopped, we do not send the message and instead return a MAILBOX_STOPPED error code. We also check if getBox returns NULL, in which case we create a new mailbox, since one has not been created already. At the end of

insertMsg, we check if there are no messages in the mailbox and its reference counter is greater than 1, in which case we wake up all processes and/or threads waiting on the mailbox.

RcvMsg will receive and remove the first message (the oldest) queued from the current mailbox by calling removeMsg. This checks whether the mailbox is empty, whether the mailbox is NULL, whether the mailbox is stopped, and the block variable. If the mailbox is empty and the block variable is false, then it returns with a MAILBOX_EMPTY error. If it is empty and the block variable is true, then it will wait until a message is put into the mailbox and the process is therefore put onto the waitqueue. If the mailbox is stopped, as long as there is a message in it, removeMsg will remove the message from the mailbox and copy its contents into user space. If it is stopped and empty, we return a MAILBOX_STOPPED error code. Otherwise, removeMsg removes the message normally and copies the string and the message length into user space. Finally, if the mailbox was full, and there are mailboxes waiting to send to it, the mailbox will wake up all the processes and allow the first in the queue to deliver.

ManageMailbox simply sets the stop variable and counts the number of messages in the mailbox. It does so atomically using the spin locks to protect the mailbox critical region while changing its fields.

To prevent more than one task from accessing a mailbox simultaneously, we implemented a locking waitqueue. After completing a majority of the project, we realized that they should be placed outside of the mailbox (like in a hashentry) so that the lock is not still set when the mailbox is removed. However, time did not permit us to change this. Our implementation still works for correctly handling race conditions amongst mailboxes when sending and receiving messages since each mailbox still has its own waitqueue and lock. We also have a main spinlock implemented in the hashtable struct that protects critical sections of code, such as inserting into the hashtable, so that two processes or threads are not accessing the same index simultaneously.

Ideally, when deleting a mailbox, the waitqueue for the mailbox will be locked, and the mailbox will be flushed. Then the space the mailbox occupies on the slab will be freed, and the waitqueue can still be unlocked since it is not contained within the mailbox. This is done by intercepting the exit and exit_group system calls. Although we intercepted both calls, we only modify exit_group since it exits a whole process rather than just a single thread. In this new call, we call the function doExit, which essentially removes the current process' mailbox before exiting. After removing the mailbox, the original system calls are called and should return normally. Since we ran into unforeseen obstacles with removing mailboxes, this call does not actually remove the mailbox correctly, hence why some lines are commented out. Also, when the module is unloaded, all mailboxes and messages within them are freed from the slab so that the two slabs can then be destroyed cleanly and there will be no remnants of the “caches”. We have code that is commented out in the module unload function that conceptually does this; however, when this code is actually run, it segfaults and crashes the kernel for reasons unknown to us. Additionally, our LKM may be unsafe to run multiple times without unloading since we are not able to successfully free up memory that is used by the hashtable, mailboxes, and messages.

As you can see from testmailbox1 and testmailbox2, our mailbox successfully sends and receives messages. Testmailbox3 also works since it successfully stops a mailbox, and then receives all messages from it. Then, it tries to send to a stopped mailbox and fails, which means a success in our case. Testmailbox4 also tests the stopping of mailboxes more and succeeds for our messaging system. Since we did not correctly implement the hashentry struct, testmailbox5, testmailbox6, and testmailbox7 do not completely work. We modified the first four test programs since we noticed they did not correctly test our LKM. For instance, in the first two test programs, the parent process does not

wait for the child to send a message. We also added sleep statements to allow each process time for sending and receiving a message. We also changed testmailbox3.c to attempt to send to a stopped mailbox, which is was not correctly testing before. As stated above, we ran into errors when trying to delete a mailbox via the system calls exit and exit_group as well as deleting mailboxes when the module unloads.

When working on this project, we decided to use git to manage our source code and prepare for a professional situation. The files are also stored online in Michael's Github account, which you can access here:

https://www.github.com/mracine/KernelMailbox