CS238 Project Report:

Message Passing over Shared Memory: The msgshm238 Library

Janus Varmarken
`jvarmark@uci.edu`

Maruf Zaber
`mzaber@uci.edu`

June 2018

Abstract

In operating system design, there are two fundamental techniques for sharing information between processes, namely shared memory and message passing. Shared memory offers high performance, but at the cost of added complexity, as the application developer must handle synchronization and deal with relative pointers. In contrast, message passing offers simplicity in terms of a simple API and removes the need for synchronization, but this comes at the cost of reduced performance. We describe the implementation of `msgshm238`, a software library that seeks to combine the best of both worlds, i.e., the performance of shared memory and the simplicity of message passing, by implementing message passing over shared memory. The code for `msgshm238` is made publicly available at the following URL: `https://github.com/jvmk/msgshm238`.

1 Background

In operating system design, there are two fundamental techniques for sharing information between processes, namely shared memory and message passing [4, pp. 116]. In shared memory, the same memory segment is made part of the address spaces of two (or multiple) processes. The processes then exchange information by writing to, and reading from, this shared memory segment. The advantage of this technique is its speed. Once established and attached[1], the shared memory can be manipulated by the cooperating processes just like ordinary memory, and hence does not require (expensive) interaction with the kernel. On the downside, shared memory imposes more work on the application developer as the application code must perform synchronization to ensure data consistency in the shared memory segment. In message passing, the processes communicate by exchanging messages over some kind of communication link. The strength of message passing is its simplicity. There is no need for synchronization, and the API is simpler and can be reduced to just two functions: `send(message)` and `recv(message)`. However, message passing is generally slower than shared memory as it involves more copying of data (to/from send/receive buffers) and requires kernel interaction as it is typically implemented using system calls. Therefore, it is generally mostly useful when dealing with small amounts of data.

2 Objective

In this project, we seek to combine the best of both techniques—i.e. the simplicity of message passing and the performance of shared memory—by implementing message passing over shared memory. More precisely, the application should be left with the illusion that inter-process communication (IPC) is achieved through message passing by exposing a simplified API similar to the one mentioned above, and the API should handle the operations that create, attach, and synchronize the access to the shared memory segment under the hood.

Conceptually, the send and receive functions of this envisioned API can be considered wrapper functions for a series of other function calls. However, care must be taken if the performance of shared memory is to be retained. For example, as the performance of shared memory stems from the fact that it does not involve interaction with the kernel, the envisioned API cannot be implemented as system calls, but must be implemented in user space. A naïve implementation of a user space version of `send(message)` might use `shmget()` to locate an existing shared memory region for the intended receiver. However, such an implementation would put the performance back on par with regular message passing as `shmget()` is a system call. The API implementation must therefore internally keep track of (cache pointers to) the shared memory regions that have been established (or attached to) by the calling process such that these can be located and reused without the need for a system call. In addition, this must be complemented by an internal procedure that performs cleanup (detaches unused shared memory regions) based on some policy (e.g., if the time since last use has exceeded a threshold).

3 Design

We implement message passing over shared memory as a user space library, `msgshm238`, comprised of a header file, `msgshm238.h`, and a corresponding implementation file, `msgshm238.c`. `msgshm238.h` defines the external interface that client code uses. In line with the discussion in Section 1, the interface is simple. It consists of a data type and three functions:

- A struct, `msg`, that models a message sent between two processes. `msg` encapsulates message metadata (the sender's and the receiver's process IDs) and the actual payload of the message.

[1]Set up and teardown of the shared memory region requires system calls.

- A function, `void send(char* payload, int receiverId)`, that sends the character array `payload` to the process with a process ID equal to `receiverId`.
- A function, `msg* recv(int senderId)`, that attempts to read a pending message sent by a process with process ID equal to `senderId`. If there are no pending messages, the function returns `NULL`.

We acknowledge that the interface *could* be made slightly cleaner by changing `send()` to take a single parameter of type `msg`—in fact, this is how it was specified in an earlier version [2]. However, we chose to change it in favor of performance: if `send()` was to take a `msg` as argument, the payload would first have to be *copied* into that message (as it is stored in the `msg` struct as a `char[]`) and then copied *again* when that `msg` is copied into the shared memory region (henceforth abbreviated as SMR) as part of the implementation of `send()`. In contrast, by having `send()` take a pointer to the payload, we avoid one copy as the payload can be copied directly into the `msg` that is created in the SMR.

4 Implementation

4.1 Tracking Existing Shared Memory Regions

As described in Section 2, the library must cache references to existing SMRs in order to avoid an `shm_open()` system call every time it tries to access an already existing SMR. While this could be achieved by simply storing a pointer to the memory location[2] that designates the start of the SMR in the view of the local process, we opt for a slightly more sophisticated design so as to allow a single process to concurrently maintain communication channels with multiple processes[3]. Specifically, we maintain a hash table (we utilize Troy D. Hanson's and Arthur O'Dwyer's `uthash` [1] for this purpose) that maps an SMR identifier to an instance of the `shm_dict_entry` struct, which contains metadata for the SMR, namely its identifier and the memory location that designates the start of the SMR in the view of the local process. The naming (identifier) scheme for SMRs must be deterministic such that process A and process B will attempt to create (or attach to) the same SMR when they want to communicate. We use a naming scheme that is the concatenation of a slash[4], the smallest process ID of the two communicating processes, an underscore, and finally the largest process ID of the two communicating processes (e.g., the name of the SMR used for communication between process 42 and process 43 is "`/42_43`").

The hash table is queried at the beginning of each call to `send()` and `recv()`. If a match is found, these functions delegate the logic that inserts/retrieves a message into/from the SMR to two other functions, namely `put_msg()` and `fetch_msg()`, respectively. If there is no match, a call is made to `create_shared_mem_segment()` which *attempts* to create the SMR. Specifically, the SMR may in fact already exist if the remote process has already invoked `send()` or `recv()`, but this will not be reflected in the local process' hash table since each process executes its own "instance" of the `msgshm238` library and thereby has its own, separate hash table. The code in `create_shared_mem_segment()` therefore first attempts to create the SMR by means of a call to `shm_open()`. If the return value of `shm_open()` indicates that the SMR already exists, another call to `shm_open()` (with different arguments) is made, which instead attaches to the existing SMR. Finally, a new `shm_dict_entry` representing the created (or existing) SMR is created and inserted into the local process' hash table. We acknowledge that the need for two calls to `shm_open()` may not be necessary if there is a way to indicate in the first call that it should simply attach to the SMR if it already exists. We believe this might be possible by omitting the `O_EXCL` flag, but we did not have time to investigate this, so we leave this as a possible future optimization.

[2]Returned by the `mmap()` call that mamps the SMR into the local address space of the process that creates or attaches to an SMR.

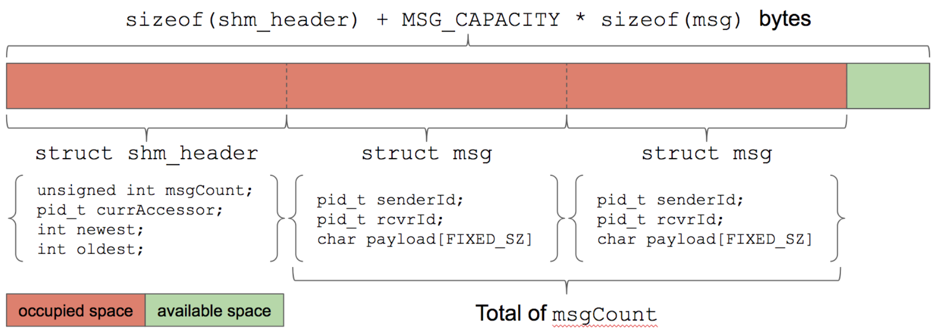
[3]We only support point-to-point channels and not one-to-many communication over a single channel, i.e., an SMR is only shared between two processes, but a process may partake in multiple SMRs, each of with is "connected to" a different endpoint (other process).

[4]The slash is required in order to comply with the POSIX naming convetions.

4.2 Indexing the Shared Memory Region

Since each process maps the SMR into its own virtual address space, an *absolute* pointer stored in the SMR by process A will only point to the correct location in the view of process B if the two processes (by chance) select the same mapping (base address) for the SMR. Needless to say, this is highly unlike to happen and therefore cannot be relied upon. Instead, the SMR must be indexed using *relative* pointers (as explained in [3]). To this end, we place an instance of the `shm_header` struct at the front of the SMR (see Figure 1 for an overview of how the data is structured in the SMR). This struct keeps count of the number of pending (unread) messages currently residing in the SMR and maintains indices for the newest (most recently added) and oldest (least recently added) message. These indices are used for calculating the offset when a `msg` is inserted into, or read from, the SMR. When inserting a message, the message is written to the memory segment starting at the base address plus the calculated offset and ending at the base address plus the offset plus `sizeof(msg)`. Similarly, when reading a message, the contents of the memory segment that starts at the base address plus the offset and extends another `sizeof(msg)` bytes is interpreted as a `msg`.

Figure 1: Layout of data stored in the SMR. The size of the SMR is specified in terms of the number of messages it should allow room for (`MSG_CAPACITY`).



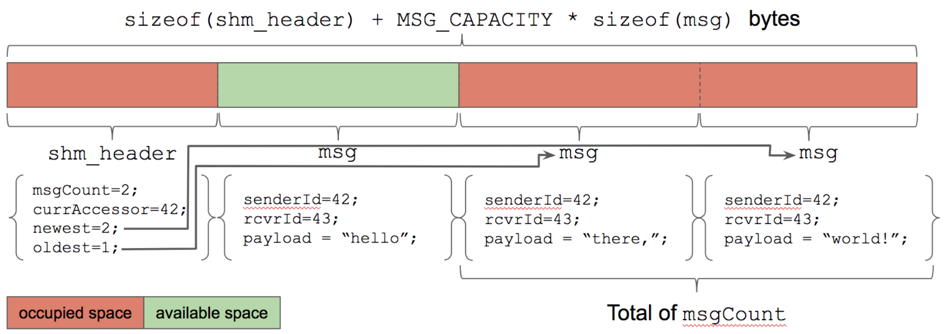
We show how the offsets are computed in listing 1. Notice that the calculation of the offset for insertions uses a modulus operation in order to “wrap around” such that old messages at the front of the SMR are overwritten when `MSG_CAPACITY` (the size of the SMR in terms of the number of messages it can contain) messages have been inserted into the SMR. However, the `put_msg()` function that performs the insertion does not blindly overwrite old messages. Instead, it returns an error code if the SMR is filled with pending messages[5] (i.e., if `shm_header->msgCount==MSG_CAPACITY`). Figure 2 shows an example state of an SMR with a message capacity of three. The message at index zero has already been received (read) as evident from the state of the `shm_header` (`msgCount` has a value of two and `oldest` indicates that the segment of pending messages start at index one), but still *physically* resides in the SMR. The message at index zero will therefore be overwritten by the invocation of the `put_msg()` function as part of the next call to `send()`.

```c
// The shm_header resides at the very front of the SMR, so cast whatever is there to an shm_header.
// Note: shm_ptr refers to an shm_dict_entry that holds the metadata for the SMR.
// Its addr field is the base address of the SMR.
shm_header* header = (shm_header*)shm_ptr->addr;
// Offset when inserting a message
size_t insert_offs = sizeof(shm_header) + sizeof(msg) * ((header->newest + 1) % MSG_CAPACITY);
// Offset when reading a message
size_t read_offs = sizeof(shm_header) + sizeof(msg) * (header->oldest);
```

Listing 1: Computing the offset when inserting into, and reading from, the SMR.

[5]There is a slight flaw in our design here. The error code is not forwarded to become a return value of `send()` as `send()` was defined to have no return value (see Section 3).

Figure 2: Example state of an SMR with MSG_CAPACITY=3. The message at index 0 has already been read as evident from the state of the shm_header, but still *physically* resides in the SMR until it is overwritten by a new message.



4.3 Synchronizing Access to the Shared Memory Segment

The variables in shm_header that hold information about the number of pending messages and the indices of the newest and oldest message are modified during each call to send() and recv(). As these variables are used for determining the offset in the shared memory when writing (send()) or reading (recv()) a message, there is hence a need for synchronizing send() and recv() calls to prevent race conditions in accesses to the shm_header variables from causing a message to be written to (or read from) the wrong position in the shared memory segment (e.g., thereby causing a message that has not been read to be overwritten).

Mutexes and semaphores are not an option for this purpose as their use incur system calls. Instead, we implement a spin-lock that relies on hardware support for compare-and-swap (CAS) as shown in listing 2. The CAS-lock operates on the pIdOfCurrent field of shm_header. pIdOfCurrent holds the pid_t of the process that is currently granted exclusive access to modify the SMR, or a designated negative value (SHM_SEGMENT_UNLOCKED) if the lock is available.

```
const pid_t SHM_SEGMENT_UNLOCKED = -1; //Lock available if shm_header->pIdOfCurrent==SHM_SEGMENT_UNLOCKED
//...
void lock_shm(shm_dict_entry* shm_metadata) {
    int expected = SHM_SEGMENT_UNLOCKED;
    shm_header* header = (shm_header *)shm_metadata->addr; // Read header at front of shm.
    get_invoker_pid(); // Refresh cache with invokers pid if necessary.
    while(!atomic_compare_exchange_weak(&(header->pIdOfCurrent), &expected, invoker_pid)) {
        expected = SHM_SEGMENT_UNLOCKED;
    }
}

void unlock_shm(shm_dict_entry* shm_metadata) {
    int expected = invoker_pid; // shm_header->pIdOfCurrent should be pid of lock-holder (the invoker)
    shm_header* header = (shm_header *)shm_metadata->addr; // Read header at front of shm.
    while(!atomic_compare_exchange_weak(&(header->pIdOfCurrent), &expected, SHM_SEGMENT_UNLOCKED)) {
        expected = invoker_pid;
    }
}
```

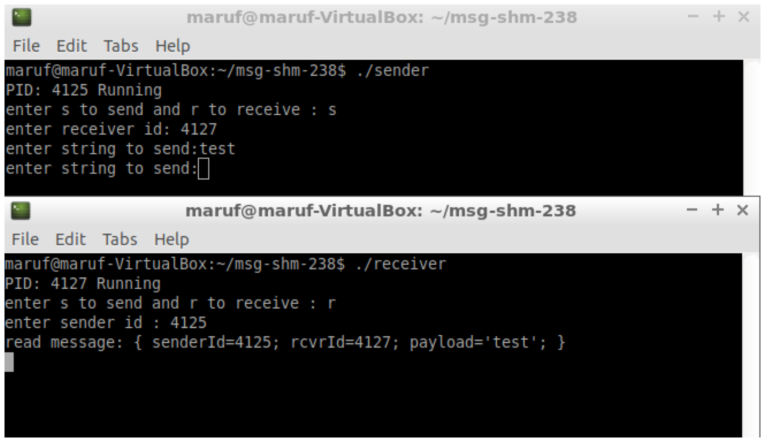
Listing 2: CAS-based spin-lock in msgshm238.c

4.4 Freeing Resources: Closing the Shared Memory Region

When closing the SMR, care should be taken as the messages that are currently in the buffer might not have been read yet. Therefore, when closing the SMR, the calling process locks the SMR and checks whether msg_count in shm_header is zero or not. If it is zero, it implies that there are no pending messages in the buffer. Therefore, it calls shm_unlink() with the identifier for the SMR and frees the space. Upon success it returns a success code to the calling process.

5 Evaluation

We tested the library by creating a small program that exercises the `msgshm238` library by using it to send and receive messages between two communicating processes. Upon execution, the process can set itself as "Sender" or "Receiver". The Sender process is prompted to input the process ID of the process that it wants to send a string to. Similarly, the receiver process is prompted to input the process ID from which it wishes to receive a string. The Sender can also close the SMR by inputting the string "close". Upon successful closure, it gets a success notification. A sample execution screen-shot is shown in Figure 3.


maruf@maruf-VirtualBox: ~/msg-shm-238
File Edit Tabs Help

```
maruf@maruf-VirtualBox:~/msg-shm-238$ ./sender
PID: 4125 Running
enter s to send and r to receive : s
enter receiver id: 4127
enter string to send:test
enter string to send:
```

maruf@maruf-VirtualBox: ~/msg-shm-238
File Edit Tabs Help

```
maruf@maruf-VirtualBox:~/msg-shm-238$ ./receiver
PID: 4127 Running
enter s to send and r to receive : r
enter sender id : 4125
read message: { senderId=4125; rcvrId=4127; payload='test'; }
```

Figure 3: Screenshot showing the execution of a simple program that exercises the `msgshm238` library.

6 Code

The source code of `msgshm238` is made available here: `https://github.uci.edu/mzaber/msg-shm-238`.

The execution commands for Ubuntu are as follows:

```
gcc -o receiver playground.c msgshm238.o -lrt
gcc -o sender playground.c msgshm238.o -lrt
./sender
./receiver
```

Listing 3: Executing the sample program that exercises the `msgshm238` library

Then insert either 'r' or 's', depending on whether you want to send or receive.

7 Conclusion

We described the implementation of `msgshm238`, a user space library that provides IPC implemented as message passing over shared memory. The approach seeks to provide the application developer with the best of both worlds, namely the simplicity of message passing, and the performance of shared memory. We achieve the former by providing a simplistic API consisting of three primitives (`send()`, `recv()`, and `close_mem()`), and by internally handling synchronization. We achieve the latter by caching pointers to existing shared memory regions, thereby avoiding unnecessary system calls. We acknowledge that performance will not be *exactly* equal to "regular" shared memory IPC as there is a need for copying content to and from the shared memory region when performing message passing over shared memory. Nevertheless, we argue that msgshm238 offers a reasonable middle ground between ultimate performance at the cost of complexity, and ultimate simplicity at the cost of performance.

8 Division of Labor

- Project proposal: joint responsibility.
- Author of midterm report: Janus Varmarken.
- Authors of present report:
  - Janus Varmarken: Abstract, Conclusion, and Section 3 to Section 4.3, inclusive.
  - Maruf Zaber: Section 4.4 to Section 6, inclusive.
  - Sections 1 and 2 were reused from the project proposal (joint responsibility).
- Architect of mechanism for tracking existing SMRs: Janus Varmarken.
- Architect of mechanism for indexing the SMR: Janus Varmarken.
- Architect of synchronization mechanism: Janus Varmarken.
- Architect of cleanup (closing the SMR): Maruf Zaber.
- Architect of sample program that exercises the library: Maruf Zaber.
- Architect of serialization code[6]: Maruf Zaber.

[6]Not used in the current solution presented here, but may be relevant if the solution is to be extended to seamlessly integrate communication with both local as well as *distributed* processes. See serialize.h.

References

[1] Troy D. Hanson and Arthur O'Dwyer. uthash: a hash table for C structures. [Online; accessed 2018-05-18].

[2] Janus Varmarken and Maruf Zaber. CS238 Midterm Report. May 2018.

[3] Michael Kerrisk. Linux/UNIX System Programming: POSIX Shared Memory. `http://man7.org/training/download/posix_shm_slides.pdf`, February 2015. [Online; accessed 2018-06-15].

[4] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley, 8th edition, 2010. International Student Version.