**Machine Problem 9: More IPC Mechanisms**

**Due XX/XX/XXXX**

**Executive Summary**

The purpose of this machine problem is to teach you about different IPC mechanisms and how to use them.

In this machine problem, you will fill in the code for five classes that each implement the behavior of RequestChannel (the same class from MP7 and MP8, but converted into an interface class for this machine problem) using a different IPC mechanism: NamedPipeRequestChannel (practically equivalent to the original RequestChannel class), UnnamedPipeRequestChannel, MessageQueueRequestChannel, SharedMemoryRequestChannel, and SignalRequestChannel, where each class is named after the primary IPC mechanism it uses.

In addition to these five main IPC mechanisms, there are two others that you will learn about: Unix-domain sockets and process-shared semaphores. Unix-domain sockets will be used to pass file descriptors between processes inside the UnnamedPipeRequestChannel constructor, and process-shared semaphores (encapsulated by the ProcessSharedSemaphore class, which you will need to fill in) will be used in various places to help synchronize separate processes.

This may seem like a lot, but really all that is required is to fill in some methods using the right system calls and library functions. Most of the required code is given to you, and this handout contains man page references for all the system calls and library functions that will be needed as well as indications of how and where to use them. Also, detailed descriptions are provided of the various IPC mechanisms and their corresponding classes, from method semantics all the way down to implementation suggestions. As a result, there should very little tedious work or confusion involved in this machine problem.

**Table of Contents**

* Introduction
* Background
  + So, why learn any IPC mechanisms other than FIFOs?
  + Descriptions of the IPC mechanisms in this assignment
    1. Named Pipes, or FIFOs
    2. Unnamed Pipes (also “anonymous pipes,” but most often simply “pipes”)
    3. Message Queues
    4. Shared Memory
    5. Signals
    6. Unix-domain Sockets
    7. Process-shared Semaphores
       - Man page references
* The Assignment
  + Code
    1. NamedPipeRequestChannel
       - Man page references
       - Implementation Notes
    2. UnnamedPipeRequestChannel
       - Man page references
       - Method descriptions
    3. MessageQueueRequestChannel
       - Man page references
       - Implementation Notes
       - Method descriptions
    4. SharedMemoryRequestChannel
       - Man page references
       - Implementation Notes
       - Method descriptions
    5. SignalRequestChannel
       - Man page references
       - Implementation Notes
       - Method descriptions
  + Report
* What to Turn In
* Criteria for Success
  + Code Test Cases
  + Report

**Introduction**

The last two machine problems have made heavy use of the RequestChannel class, which was pre-written and given to you along with the dataserver code so that you could focus on learning synchronization techniques. In fact, it was possible to complete both MP7 and MP8 without even *looking* at either dataserver.cpp or reqchannel.cpp.

If you did look at them, you may have noticed that the RequestChannel class uses a mechanism called “named pipes” or “FIFOs” to communicate between the two sides of the channel. However, FIFOs are only one of several different IPC mechanisms, each of which have their own particular uses that make them suited to particular applications. In this machine problem, we’re going to expand our toolbox by learning about 4 “new” IPC mechanisms in addition to named pipes: unnamed pipes, message queues, shared memory, and signals.

**Background**

So, why learn IPC mechanisms other than FIFOs?

There are many IPC tasks that are FIFOs are unsuited for. For example, what if you just need to communicate between a process and its child and it seems like overkill to use a FIFO, or you just *really* want to avoid adding something to the filesystem (which may provide memory or security benefits in some cases)? In that case, unnamed pipes work much better. What if, rather than a byte stream, you need something that respects message boundaries, or message priority is a concern? In this case, message queues are more appropriate. What if, instead of or in addition to simply communicating, two processes need to share some user-defined object? Then that object will probably need to be constructed using shared memory. Finally, what if you need to communicate asynchronously with another process? Then signals can be very helpful.

Before we introduce the assignment, we will give a brief overview of each of these different IPC mechanisms, as well as Unix-domain sockets and process-shared semaphores. Please note that there is a man page for just about everything here, and they explain these things much better than we do ☺. The following information is provided as an introduction to the assignment, but when you are actually writing code it will serve you *much* better to rely on the man pages. Man page references for process-shared semaphores are provided in this section, but the references for all the other IPC mechanisms are in the Assignment section. However, you will see little references to them throughout the document.

Descriptions of the IPC mechanisms in this assignment

1. Named Pipes, or FIFOs

Pipes, generally speaking, are unidirectional byte-stream channels with a read end and a write end. Both ends have associated file descriptors, and if a process has opened both it must close one before using the other. Since individual pipes are only one-way channels, two-way or “full-duplex” communication using pipes requires that 2 pipes be created.

I/O with pipes is done by using the pipe file descriptors in calls to read() and write(). The behavior of read and write can vary depending on how the pipes were created and opened, but there are a few of cases worth noting: 1) attempting to read from a pipe whose write end is not open will return end-of-file, 2) attempting to read from a pipe in which no data is available will block unless the pipe has been specified as non-blocking (then it fails with EAGAIN), 3) attempting to write to a pipe whose read end is not open will generate SIGPIPE, 4) attempting to write to a full pipe will block unless the pipe has been specified as non-blocking (then it fails with EAGAIN). Reads and writes are done using the underlying pipe mechanism, which resides in the kernel: it makes no difference semantically whether there is a file system entry corresponding to that mechanism, as there is with named pipes.

As briefly mentioned in the previous section, pipes provide a “byte stream” between the two ends, i.e. there are no message boundaries: one simply asks to read or write a given number of bytes. This is important to keep in mind, as it is a significant difference in semantics between pipes and other IPC mechanisms.

Now, named pipes are called “named” pipes because they have pathnames within the filesystem, which can be used to open() them like most other files. However, they are not ordinary files but *special* files and must be created with a call to mkfifo() before they can be open()-ed. When named pipes are no longer needed, their file descriptors can be closed using close() and the pipe itself removed using remove() (this gets rid of both the filesystem entry and the underlying pipe mechanism). All of this is visible inside the RequestChannel class from MP7 and MP8.

2. Unnamed Pipes (also “anonymous pipes,” but most often simply “pipes”)

Unnamed pipes differ from named pipes in that they do not exist within the file system. They can only be accessed directly by their file descriptors, and since they have no names open() cannot be used to make new file descriptors. As such it most common that unnamed pipes are used to communicate between parent and child processes, which automatically share all the file descriptors that are open at the time of the call to fork().

An unnamed pipe is created with a call to pipe(), which produces two file descriptors: one for the pipe’s read end and the other for its write end. Once the file descriptors have been set up, they are used the same way as for named pipes. Then when all file descriptors referring to the pipe have been closed, the underlying pipe structure is removed automatically (convenient, eh?).

Since using unnamed pipes requires direct access to file descriptors, they will require much more setup than the other IPC mechanisms: one process has to send the file descriptors to the other process, and ensure that they have been received, before the pipes can be used. This requires the use of Unix-domain sockets, which will be discussed later.

3. Message Queues

While pipes provide a byte stream between two processes, message queues allow for the exchange of…messages (whoah, that’s amazing!)…between processes. There are library functions (POSIX library, not STL) for message queue opening/creation, sending messages, receiving messages, closing the message queue, deleting the message queue, and modifying the message queue’s attributes. You may be able to use default attributes for this assignment, but those defaults vary by system. Visit man 7 mq\_overview for how to check and set default message queue attributes.

On most systems, message queues not only exist as entities within the file system but are all kept in the same directory. On Linux, this directory is /dev/mqueue (it may need to be manually mounted before becoming visible, see man 7 mq\_overview for details). Their contents cannot be accessed by command line, though one can get a brief summary of their contents from “cat [mqueue-pathname]”. Both the filesystem entry for a message queue, and the underlying structure itself, have what’s called “kernel persistence”: if not deleted by the appropriate system call during the execution of a process (mq\_unlink), they will remain until they are either manually deleted by command line (only possible on systems where message queues exist in the file system) or the kernel is restarted.

Messages can be given a priority when sent, and are received in a FIFO-by-priority order, i.e. messages of higher priority are always processed before messages of lower priority, but when priority is held equal older messages are always processed before newer messages. Additionally, sending and receiving messages to and from the same message queue is completely threadsafe. Thus, the message queue a very nice, ready-to-use structure with very little additional legwork needed from the programmer. Within the scope of this assignment, you’ll find that working with message queues is significantly easier than working with other IPC mechanisms.

4. Shared Memory

Up until now there have been IPC mechanisms to provide byte streams and message passing, but what if something a little more versatile is needed? Shared memory is exactly what it sounds like: a segment of memory that can be read and modified (depending on its configuration) by multiple different processes.

There are system calls that allow one to open/create a shared memory segment, set its size, map it into a process’s virtual address space, change its ownership and permissions, obtain statistics about it, close it, and delete it. However, you’ll notice that there are no system calls for reading and writing the shared memory segment. This is because the shared memory segment is semantically identical to any other memory segment, such as can be obtained from malloc, except for the IPC and synchronization considerations. One can read and modify it using memset, strcpy, memcpy, or just about any other memory-reading/writing operations.

Like message queues, shared memory segments exist (on some operating systems) as entities in the filesystem. In Linux, shared memory segments are kept in the directory /dev/shm, where they can be modified or deleted by command line.

Shared memory is used by mapping the segment into a process’s virtual address space, and it is that mapping which processes can then do operations on. Modifications made to individual mappings must be written to the underlying segment before they can be seen in other mappings. Such updates are sometimes done automatically, but unless they are forced (man 2 msync) there is no guarantee that updates will have been made by the next time a process accesses the segment. This introduces synchronization concerns that will be discussed in the Assignment section.

Shared memory is much more unwieldy for message-passing than pipes or message queues. Within the scope of this assignment, it is more difficult to use than other IPC mechanisms. However, it is also a more powerful programming tool than the other IPC mechanisms, since it can be used to create shared objects and data structures that do more than pass messages.

5. Signals

Signals can be used for asynchronous IPC. For example, the key chord Ctrl-C, if typed while a process is running in a shell, sends SIGINT to (or “interrupts”) the running process, which (by default) tells that process to terminate regardless of what it had been doing. Signals also don’t depend on creation of any data structures outside of a process’s own address space.

Signals tend to have different meanings depending on how they are generated. For example, a bad memory access causes the infamous SIGSEGV to be generated (“Segmentation fault”). Setting an alarm or timer (using the right system call) causes SIGALRM to be generated. Dividing by zero causes SIGFPE (“Floating point exception”) to be generated. There are many different signals with many different meanings, and the interesting thing is that any of them can be generated synchronously using a system call (such as raise or kill).

Programmers can define the behavior for most signals by using system calls to establish a “signal handler,” which is a function that then gets called when a specific signal is delivered to a thread. Notice we said thread, not process: signals are usually sent to processes (except in the case that one signals a specific thread using its thread id), but are always delivered to threads. Different threads within the same process can define different signal handlers, and have different signal masks.

A signal mask is the set of the signals that are blocked by given thread. When new threads are created using pthread\_create(), they inherit the signal mask of the thread that created them. Likewise, child processes created via fork() inherit their parent’s signal mask. When a signal is blocked by a thread, but other threads within the same process are not blocking it, and the signal is sent to the process rather than a specific thread, then the signal is delivered to one of the threads that isn’t blocking it (chosen arbitrarily). If all threads are blocking the signal then the the signal is “pending” until a thread either unblocks it or uses a system call to accept it synchronously. Multiple signals can be pending for a given process (or “queued”), and they are delivered in the order they were sent in.

Signals have a wide variety of uses, especially for error handling. The man page (man 7 signal) describes several of them. Like shared memory, they are unwieldy for the purposes of this assignment but are extremely useful in practice.

6. Unix-domain Sockets

These are sockets of the AF\_UNIX (or equivalently, AF\_LOCAL) family, and they are used to communicate between processes on the same machine. Depending on how they are constructed they can be either named (existing as entities in the filesystem) or unnamed, and be of type SOCK\_STREAM, SOCK\_DGRAM, or (Linux-only) SOCK\_SEQPACKET.

Unix-domain sockets can be used to pass file descriptors and process credentials between processes, using the sendmsg and recvmsg system calls. Besides these and other special features, they are set up using mostly the same system calls as other sockets, such as bind, listen, accept, and connect. They also support many ordinary socket system calls, such as send() and recv(). See man 7 unix for details.

In the section about unnamed pipes, it was mentioned that the only way for two processes to use the same pipe is directly through the pipe’s file descriptors. For two processes that are not parent and child, one process can pass the file descriptors to the other through a Unix-domain socket. That is the approach that will be used for this machine problem.

7. Process-shared Semaphores

You are already familiar with the concept of semaphores from MP7 and MP8, and even had to write your own. However, that semaphore was only “process-local” or “thread-shared,” since it was allocated within the address space of a given process. What if two different processes needed to synchronize on the same semaphore, a “process-shared” semaphore?

One of the many uses of shared memory is that it allows the creation of process-shared semaphores. Now, one way to create a process-shared semaphore would be to simply allocate a semaphore inside a shared memory segment using the semaphore class from MP7 and MP8. This will work if you change the attributes passed to the different pthread\_\*\_init functions (and maybe change some other stuff, we don’t know since we haven’t tried it ourselves ☺). However, there are convenient library functions that eliminate the need to do all that work.

For this machine problem, part of what you will need to do is fill in the methods of the ProcessSharedSemaphore class. ProcessSharedSemaphore implements the familiar Semaphore interface but is distinct from the process-local semaphore from MP7 and MP8, which is preserved in the ThreadSharedSemaphore class. ProcessSharedSemaphore is used in many different places throughout this machine problem, so it is very important to finish it early. Don’t worry, it isn’t very difficult.

man 7 sem\_overview – <http://man7.org/linux/man-pages/man7/sem_overview.7.html>

man 3 sem\_open – <http://man7.org/linux/man-pages/man3/sem_open.3.html>

man 3 sem\_post – <http://man7.org/linux/man-pages/man3/sem_post.3.html>

man 3 sem\_wait – <http://man7.org/linux/man-pages/man3/sem_wait.3.html>

man 3 sem\_close – <http://man7.org/linux/man-pages/man3/sem_close.3.html>

man 3 sem\_unlink – <http://man7.org/linux/man-pages/man3/sem_unlink.3.html>

**The Assignment**

**Code**

**For this machine problem, you will implement the constructor and destructor, as well as cwrite and cread functions, of 5 new classes which correspond to the 5 major IPC mechanisms discussed in the Background section: NamedPipeRequestChannel, UnnamedPipeRequestChannel, MessageQueueRequestChannel, SharedMemoryRequestChannel, and SignalRequestChannel**. Each of these inherit from RequestChannel, which has been converted into an interface class. Before the code will work, you will also have to fill in the ProcessSharedSemaphore class.

You’re given some code to help with testing, and it is the same code that will be used for grading: client.cpp, RequestChannelTest.cpp, Dataserver.cpp, Dataserver.h, and dataserver\_main.cpp. The last four implement the familiar worker-channels-sending-requests-to-dataserver behavior from MP7 and MP8, and client.cpp simply calculates the average latency of 10000 calls to send\_request. All of them take command-line arguments to determine which of the \*RequestChannel classes to use for the control and worker channels, as well as other arguments that are explained in the help messages (-h flag).

You’re also given a functioning makefile and all the .cpp and .h files that you should need in order to complete the assignment, such that all you will need to do is fill in the functions correctly and make sure they work with the provided test programs. Many of the class declarations, including for RequestChannel and ProcessSharedSemaphore classes, are contained in sync\_lib.h, while the implementations will be in the .cpp files corresponding to the class. It will help you to read sync\_lib.h carefully before beginning to write code.

The primary measures of success will be that certain test cases (provided in the Criteria for Success section) run to completion, process as many requests as specified by the -n flag, and clean up all IPC structures that were allocated during execution. Checking the /dev/mqueue and /dev/shm directories, as well as checking for leftover fifos and Unix-domain socket files, will help you make sure that everything is working correctly.

Your code will be graded on the departmental Linux server, so be sure to test it there. You will find that the system calls and library functions referred to in this document, even though most of them are specified by POSIX, are more predictable and easier to use on Linux machines. It is also easier to check for proper cleanup (and hence to do grading) on Linux machines since they allow for shared memory and message queues to be accessed via command line.

Before we describe how the constructors, destructors, and cread and cwrite functions are going to work in the different \*RequestChannel classes, here are some man page references you will find immensely useful for debugging:

man 3 errno – <http://man7.org/linux/man-pages/man3/errno.3.html>

man 3 perror – <http://man7.org/linux/man-pages/man3/perror.3.html>

man 3 strerror – <http://man7.org/linux/man-pages/man3/strerror.3.html>

When writing error-handling code for a given system call or library function, be sure to read its man page very carefully to know how it reports errors and what those errors mean. For example, pthread\_\* functions return their error values instead of setting errno; write() fails with EPIPE if the file descriptor argument refers to a broken pipe, but only if SIGPIPE is blocked; and other such idiosyncrasies.

**NamedPipeRequestChannel**

man 7 pipe – <http://man7.org/linux/man-pages/man7/pipe.7.html>

man 7 fifo – <http://man7.org/linux/man-pages/man7/fifo.7.html>

man 3 mkfifo – <http://man7.org/linux/man-pages/man3/mkfifo.3.html>

man 3 remove – <http://man7.org/linux/man-pages/man3/remove.3.html>

man 2 read – <http://man7.org/linux/man-pages/man2/read.2.html>

man 2 write – <http://man7.org/linux/man-pages/man2/write.2.html>

man 2 open – <http://man7.org/linux/man-pages/man2/open.2.html>

man 2 close – <http://man7.org/linux/man-pages/man2/close.2.html>

Implementation Notes: All method implementations can be copy-pasted from the original RequestChannel code, but it would behoove you to read them thoroughly and understand what they do since it will help with writing the other RequestChannel subclasses. Enjoy the free points. ☺

**UnnamedPipeRequestChannel**

Using unnamed pipes:

man 7 pipe – <http://man7.org/linux/man-pages/man7/pipe.7.html>

man 2 pipe – <http://man7.org/linux/man-pages/man2/pipe.2.html>

man 2 read – <http://man7.org/linux/man-pages/man2/read.2.html>

man 2 write – <http://man7.org/linux/man-pages/man2/write.2.html>

man 2 close – <http://man7.org/linux/man-pages/man2/close.2.html>

Unix-domain sockets:

man 7 unix – <http://man7.org/linux/man-pages/man7/unix.7.html>

man 2 socket – <http://man7.org/linux/man-pages/man2/socket.2.html>

man 2 bind – <http://man7.org/linux/man-pages/man2/bind.2.html>

man 2 listen – <http://man7.org/linux/man-pages/man2/listen.2.html>

man 2 accept – <http://man7.org/linux/man-pages/man2/accept.2.html>

man 2 connect – <http://man7.org/linux/man-pages/man2/connect.2.html>

man 2 unlink – <http://man7.org/linux/man-pages/man2/unlink.2.html>

Passing file descriptors via Unix-domain socket:

man 3 cmsg – <http://man7.org/linux/man-pages/man3/cmsg.3.html>

man 2 sendmsg – <http://man7.org/linux/man-pages/man2/sendmsg.2.html>

man 2 recvmsg – <http://man7.org/linux/man-pages/man2/recvmsg.2.html>

Blocking SIGPIPE (if you want to):

man 7 signal – <http://man7.org/linux/man-pages/man7/signal.7.html>

man 3 sigsetops (includes things like sigemptyset and sigsetadd) – <http://man7.org/linux/man-pages/man3/sigsetops.3.html>

man 3 pthread\_sigmask – <http://man7.org/linux/man-pages/man3/pthread_sigmask.3.html>

* **Constructor** –
  1. Block SIGPIPE if you don’t want the program to crash inside cwrite if a pipe is broken (sigemptyset, sigaddset, pthread\_sigmask)
     1. If you do this, the call to write will fail and return EPIPE: make sure you handle it appropriately (errno)
        + This is often preferable to allowing SIGPIPE to kill the process, but either is acceptable for this machine problem.
  2. SERVER\_SIDE:
     1. Construct a Unix-domain socket (socket)
     2. bind the socket fd to a name in the filesystem corresponding to the name passed as the first argument to the constructor (bind)
        + If a socket with the same name already exists, remove it (unlink)
          - Will probably need to be done before bind is called
        + It may be necessary to use the global namespace for the call to bind, so it may look like if(**::**bind(…[args]…) < 0) { do stuff }, where the two colons are a prefix for the bind system call that specifies global namespace
     3. Call listen on the new socket (listen)
     4. Signal the client that the socket is ready for connections
        + We find that process-shared semaphores are a very easy way to do this
        + If the client attempts to call connect before the server calls listen, it will fail with ECONNREFUSED (connect, errno)
     5. Accept the client’s connection request (accept)
     6. Create the pipes (pipe)
     7. Initialize the msghdr structure that will be passed to sendmsg, placing the client’s pipe file descriptors (i.e. read end of one pipe, write end of the other pipe) in the data section (cmsg)
     8. Send the client’s file descriptors to the client (sendmsg)
     9. Wait for client to signal that it has received its file descriptors
        + Again, a process-shared semaphore will do this very nicely
     10. Close the server-side copies of the client’s file descriptors
  3. CLIENT\_SIDE:
     1. Construct a Unix-domain socket (socket)
     2. Wait for the server to signal that it’s listening on the socket
        + A process-shared semaphore will work, so long as the server is using the same one
     3. Connect to the server on the Unix-domain socket (connect)
     4. Construct a msghdr structure into which the file descriptors can be received (cmsg)
     5. Receive the server’s message, containing the file descriptors (recvmsg)
     6. Parse the message and extract the file descriptors (cmsg)
     7. Signal the server that the file descriptors have been received
  4. **NOTE FOR THE CONSTRUCTOR** – These instructions assume that the Unix-domain sockets are of type SOCK\_STREAM. You are free to use another socket type, such as SOCK\_DGRAM, if you can figure out how on your own.
* **Destructor** –
  1. Close the pipe file descriptors, and remove the Unix-domain socket if you haven’t done so already (it can be deleted as soon as the constructor is finished with it) (close, unlink)
  2. Make sure any semaphores that were used get properly cleaned up
* **cwrite** – write using the write file descriptor, return number of bytes written (code from NamedPipeRequestChannel::cwrite can be reused) (write)
* **cread** – read using the read file descriptor, return a string result (code from NamedPipeRequestChannel::cread can be reused) (read)

**MessageQueueRequestChannel**

man 7 mq\_overview – <http://man7.org/linux/man-pages/man7/mq_overview.7.html>

man 3 mq\_open – <http://man7.org/linux/man-pages/man3/mq_open.3.html>

man 3 mq\_send – <http://man7.org/linux/man-pages/man3/mq_send.3.html>

man 3 mq\_receive – <http://man7.org/linux/man-pages/man3/mq_receive.3.html>

man 3 mq\_close – <http://man7.org/linux/man-pages/man3/mq_close.3.html>

man 3 mq\_unlink – <http://man7.org/linux/man-pages/man3/mq_unlink.3.html>

Implementation Notes: The number of MessageQueueRequestChannels that can be created at run time is determined by the system limit on the number of message queues. However, you never need to know what that limit is so long as you know how to clean up and keep going when you hit it. Error-handling inside the constructor will help with this. (mq\_open, errno, mq\_overview)

You may notice a class called MessageQueueBoundedBuffer. Full-duplex communication is difficult to implement using only a single message queue, and a two-way request channel can be easily thought of as using two bounded buffers, so you may find it helpful to wrap the message queue-related functions inside the MessageQueueBoundedBuffer class. The MessageQueueRequestChannel class would then just construct two MessageQueueBoundedBuffer objects, one “read buffer” and one “write buffer,” and use their push\_back and retrieve\_front functions inside cwrite and cread. All the destruction operations could be done automatically through the MessageQueueBoundedBuffer destructor.

You are not required to use the MessageQueueBoundedBuffer class, though, and whether or not you do will not affect grading. It just might make writing the code easier.

* **Constructor** –
  1. Construct two message queues. The server will read from one and write to the other, the client will read from the one the server writes to and write to the one the server reads from, similar to full-duplex communication using pipes. (mq\_overview, mq\_open)
  2. …that’s it.
* **Destructor** –
  1. Close the message queue descriptor, and delete the underlying message queue (mq\_overview, mq\_unlink)
* **cwrite** – send a message to the appropriate message queue (mq\_send)
* **cread** – retrieve a message from the appropriate message queue (mq\_receive)

**SharedMemoryRequestChannel**

man 7 shm\_overview – <http://man7.org/linux/man-pages/man7/shm_overview.7.html>

Used over the lifetime of a shared memory segment:

man 3 shm\_open – <http://man7.org/linux/man-pages/man3/shm_open.3.html>

man 3 shm\_unlink – <http://man7.org/linux/man-pages/man3/shm_unlink.3.html>

man 2 ftruncate – <http://man7.org/linux/man-pages/man2/ftruncate.2.html>

man 2 mmap – <http://man7.org/linux/man-pages/man2/mmap.2.html>

man 2 close – <http://man7.org/linux/man-pages/man2/close.2.html>

man 2 munmap – <http://man7.org/linux/man-pages/man2/munmap.2.html>

Can be helpful when reading from/writing to a shared memory segment:

man 3 memset – <http://man7.org/linux/man-pages/man3/memset.3.html>

man 3 memcpy – <http://man7.org/linux/man-pages/man3/memcpy.3.html>

man 3 strcpy – <http://man7.org/linux/man-pages/man3/strcpy.3.html>

Essential for synchronizing updates to the shared memory segment:

man 2 msync – <http://man7.org/linux/man-pages/man2/msync.2.html>

Implementation Notes: Notice that the previously discussed RequestChannel implementations construct some pre-defined data structure for communication. However, for a shared memory segment to be used for full-duplex communication, some structure has to be “imposed” on it. The structure you choose will determine how large the shared memory segment(s) needs to be. To save you some trouble we will here describe one possible implementation, but please note that it is only one of many possibilities.

We mentioned the possibility of using bounded-buffer semantics with MessageQueueRequestChannel. That can also be done here with SharedMemoryRequestChannel, through the SharedMemoryBoundedBuffer class. There would be two SharedMemoryBoundedBuffers (and thus two shared memory segments, yes that is allowed), where each side would have a read buffer and a write buffer and use push\_back() and retrieve\_front() inside cwrite() and cread().

A bounded buffer in shared memory can be simply an array of memory blocks (you pick the block size that works for you), which keeps track of the “ends” like a queue does, i.e. reads are done using a read index that represents the front of the queue, and writes are done using a write index that represents the back of the queue. When a read or write is finished, the appropriate index is incremented by one, and if an index points past the end of the queue (this can be checked by comparing the index to the maximum allowed size, in blocks, of the bounded buffer) it is reset to zero. After a read is performed, the block pointed to by the read index is zeroed out (perhaps by memset) to represent it being removed from the queue (as if “pop\_front” had been called).

In order for both the client and server to have the same indices, the read and write indices must be kept (in this implementation) in the shared memory segment along with all the messages. The shared memory segment also has to be protected by semaphores in the same manner as the ThreadSharedBoundedBuffer (which preserves the bounded buffer class from MP7 and MP8) class protects its underlying std::queue.

* **Constructor** – **(if using SharedMemoryBoundedBuffer, this is how its constructor will work)**
  1. Create semaphores to be used with the shared memory segment (sem\_open)
  2. Create/open shared memory segment (shm\_overview, shm\_open)
  3. Set the size of the segment (ftruncate)
     1. On Linux, it’s okay for ftruncate to be called more than once for the same segment, so client and server can both do so without requiring any additional logic.
     2. On the other hand, if you try running this on OS X then ftruncate will fail with EINVAL if called more than once for the same segment.
  4. Map the segment to the process’s virtual address space (mmap)
  5. Close the shared memory file descriptor returned by shm\_open, since it is no longer needed
  6. Initialize the pointers that will be used by cread and cwrite (i.e. the pointers to the mapping of the shared memory segment)
* **Destructor** –
  1. Remove the virtual address space mapping (munmap)
  2. Delete the underlying shared memory segment (shm\_unlink)
  3. Close and remove any semaphores that were created (sem\_close, sem\_unlink)
* **cwrite** **- (also semantics for SharedMemoryBoundedBuffer::push\_back)**
  1. If using bounded buffer implementation, obtain empty and access semaphores in that order (sem\_wait)
  2. Pinpoint location of next section available for writing (using write index in the bounded buffer implementation)
  3. Write message to the memory block, store length for return value (memcpy, strcpy)
  4. Update memory location for next write (in the bounded buffer implementation, simply increment/reset the write index)
  5. **Force changes to be written to the underlying shared memory segment** (msync)
     1. It will probably be best to use the MS\_SYNC and MS\_INVALIDATE flags.
  6. If using bounded buffer implementation, release access and full semaphores in that order
  7. Return the number of bytes written in step 3
* **cread – (also semantics for SharedMemoryBoundedBuffer::retrieve\_front)**
  1. If using bounded buffer implementation, obtain full and access semaphores in that order (sem\_wait)
  2. Pinpoint location of next message to be read (using read index in the bounded buffer implementation)
  3. Read message from memory, store in a string that will be the return value of this function (strcpy)
  4. Zero-out the message block that was just read from (memset)
  5. Update memory location of next message to be read (in the bounded buffer implementation, simply increment/reset the read index)
  6. **Force changes to be written to the underlying shared memory segment** (msync)
     1. Again, the MS\_SYNC and MS\_INVALIDATE flags help ensure correctness.
  7. If using bounded buffer implementation, release the access and empty semaphores in that order
  8. Return the string constructed in step 3

**SignalRequestChannel**

man 7 signal – <http://man7.org/linux/man-pages/man7/signal.7.html>

man 3 sigqueue – <http://man7.org/linux/man-pages/man3/sigqueue.3.html>

man 3 sigsetops – <http://man7.org/linux/man-pages/man3/sigsetops.3.html>

man 3 pthread\_sigmask – <http://man7.org/linux/man-pages/man3/pthread_sigmask.3.html>

man 2 sigwaitinfo – <http://man7.org/linux/man-pages/man2/sigwaitinfo.2.html>

man 2 getpid – <http://man7.org/linux/man-pages/man2/getpid.2.html>

Implementation Notes: There are two difficulties with using signals for a request channel:

1. In order to support multiple different request channels, multiple different signals have to be used. For example, if multiple request channels in the same process are all listening for SIGALRM then there’s no way to guarantee that the signal is delivered to the correct one.
2. We have not yet introduced how data can be sent along with a signal.

In Linux, real-time signals can be used along with the sigqueue and sigwaitinfo system calls to solve both of these problems.

As mentioned earlier, each signal has a unique value (i.e. SIGALRM = 14, SIGKILL = 9). Real-time signals have values SIGRTMIN through SIGRTMAX. Each client-server pair of SignalRequestChannels must share a distinct real-time signal number in order not to interfere with other SignalRequestChannels. We found that an effective solution was to use a SharedMemoryBoundedBuffer (described in the SharedMemoryRequestChannel implementation notes) and when it is first constructed to populate it with all the available real-time signal numbers (as well as any additional signal numbers you want to use). This can all be done inside the SignalRequestChannel constructor. Then, each server-side SignalRequestChannel takes a number from the buffer, and sends it along with its process id to the client (it is acceptable to use one of the other request channel classes for this, so long as it gets cleaned up properly), receives the client’s process id, and uses the signal number and client process id for calls to cread and cwrite. Finally, in the server-side destructor the channel signal number is returned to the buffer of signal numbers.

Now that we have a unique signal number for each request channel pair, how to we transfer data via signal? The answer is, by using the sigqueue and sigwaitinfo system calls inside cwrite and cread respectively. The man pages for those system calls describe how this works. But that’s only a partial solution, since typically pointers are only valid within the processes own address space (this will make sense after you read the sigqueue man page), as the memory it points to doesn’t get transferred with it. So, we’re confined to sending integers. This part of the implementation is so hacked that we won’t expect you to code it yourselves: you are given two functions, string\_to\_sig\_msg\_value() and sig\_msg\_value\_to\_string(), which respectively encode and decode the requests that the dataserver understands. Inside cwrite, call string\_to\_sig\_msg\_value() on the request and use the result as the sigval argument to sigqueue. Inside cread, call sig\_msg\_value\_to\_string() on the data provided by sigwaitinfo and return the result. But you say, how will we know which person a request was for? The answer is, that has always been handled at the application level since the dataserver only sees “data” requests regardless of “who” the requests are for, so it doesn’t affect the assignment.

The number of worker channels you can create will be limited by the number of real-time signals available. Your code will be expected to be able to handle only as many as SIGRTMAX – SIGRTMIN + 1 threads when using SignalRequestChannels, and we recommend you not try to exceed that limit. In that situation, if you’re using a SharedMemoryBoundedBuffer to keep track of the signal numbers, your code will probably hang and you will not get bonus points for fixing it.

* **Constructor – (note that signals numbers may NOT be passed as an additional argument to the constructor)**
  1. Block all signal numbers that will be used by SignalRequestChannels to avoid unwanted interrupts (sigemptyset, sigaddset, pthread\_sigmask)
  2. SERVER\_SIDE
     1. Access pool of available signal numbers
        + If it is being created for the first time, add all signal numbers to it that will be used by the SignalRequestChannels
        + If using SharedMemoryBoundedBuffer, you can set a flag inside its constructor if the shared memory segment was newly created and check it inside the SignalRequestChannel constructor (shm\_open)
     2. Take a number from the pool
     3. Send process id and signal number to the client (getpid)
        + It is acceptable to use one of the other \*RequestChannel subclasses for this
          - If you do, make sure it gets cleaned up when the constructor is finished with it
     4. Receive and store the client’s process id for use in cwrite
  3. CLIENT\_SIDE
     1. Wait to for server to send its process id and signal number
     2. Store both for use in cwrite
     3. Send client process id to server
* **Destructor –**
  1. SERVER\_SIDE
     1. Return signal number to pool
        + Make sure that any data structures use to represent the pool are eventually cleaned up
  2. CLIENT\_SIDE
     1. The client doesn’t need to do anything inside its destructor, since only the server side needs to know about the signal number pool.
        + However, it may be good to do so anyway for redundancy.
* **cwrite –** 
  1. Declare a sigval variable
  2. Call string\_to\_sig\_msg\_value on the string argument that was given to cwrite, store the return value in the sival\_int field of the sigval from step 1.
  3. Send the sigval to the partner using the channel signal number and the partner’s process id (sigqueue)
  4. The return value for SignalRequestChannel::cwrite is somewhat arbitrary, one option is to return the value that was sent to the partner
* **cread –**
  1. Construct a sigset\_t whose only member is the channel signal number (sigemptyset, sigaddset)
  2. Declare a siginfo\_t
  3. Call sigwaitinfo using the sigset\_t from step1 and siginfo\_t from step 2 (sigwaitinfo)
  4. Call sig\_msg\_value\_to\_string on the si\_int field of the siginfo\_t, return the result

**Report**

1. Use the provided client.cpp file to gather timing data for each of the \*RequestChannel classes.
   * Present a performance comparison of the different IPC mechanisms based on this data, and attempt to provide an explanation for any differences and similarities.
     + You may need to run the program many times for each RequestChannel subclass before patterns begin to emerge.
   * Describe the machine the data was gathered on.
   * Please suppress unnecessary output messages while gathering data.
2. Use the provided RequestChannelTest.cpp file to gather performance data for each of the 5 IPC mechanisms from this machine problem, varying the number of threads while holding everything else constant. Include separate graphs of your data for each IPC mechanism.
   * What are some of the limits encountered by each class, either due to the specific implementation or to operating system limitations, and how does the program behave when it encounters them?
   * If these limits caused errors in your program, how did you handle them?
   * Again, please suppress unnecessary output messages while gathering data.
3. Please describe briefly how you implemented each of the 5 \*RequestChannel classes.
   * This also helps us know how to check for proper cleanup during grading.
4. If compiling your code requires anything besides the makefile (by the way you shouldn’t need to modify the makefile, but may do so if it turns out to be necessary for your implementation), please provide instructions here.

**What to Turn In**

Turn in a single zip file containing the report, and all the files necessary to compile and run your code on the departmental Linux server.

**Criteria for Success**

* Code Test Cases (run on departmental Linux server):
  + 1. ./reqchannel\_test –v 2 –n 10000 –w 40 –r 0
       1. Final totals must be correct, and no fifo special files leftover
    2. ./reqchannel\_test –v 2 –n 10000 –w 40 –r 1
       1. Final totals correct, and no Unix-domain socket special files leftover
    3. ./reqchannel\_test –v 2 –n 10000 –w 40 –r 2
       1. Final totals correct, and no message queue objects leftover in the filesystem (can check with ls /dev/mqueue)
    4. ./reqchannel\_test –v 2 –n 10000 –w 40 –r 3
       1. Final totals must be correct, and no shared memory segments leftover in the filesystem (can check with ls /dev/shm)
    5. ./reqchannel\_test –v 2 –n 10000 –w 10 –r 4
       1. Final totals must be correct, proper cleanup depends on implementation
    6. Any class that uses process-shared semaphores must properly clean them up as well (can check with ls /dev/shm/sem.\*)
* Report:
  1. Should be complete, readable, and reasonable