

Problem Statement

This assignment has the following goals:

1. To solidify your understanding of IPC principles.
2. To develop greater appreciation for the different IPC mechanisms.
3. To gain hands-on experience using shared memory.
4. To gain hands-on experience using message queues.
5. To gain hands-on experience using signals.
6. To learn how to combine shared memory and message queues in order to implement a practical application where the sender process sends information to the receiver process.

Overview

In this assignment you will use your knowledge of shared memory and message queues in order to implement an application which synchronously transfers files between two processes.

You shall implement two related programs: a sender program and the receiver program as described below:

- sender: this program shall implement the process that sends files to the receiver process.

It shall perform the following sequence of steps:

1. The sender shall be invoked as `./sender file.txt` where sender is the name of the executable and file.txt is the name of the file to transfer.
2. The program shall then attach to the shared memory segment, and connect to the message queue both previously set up by the receiver.
3. Read a predefined number of bytes from the specified file, and store these bytes in the chunk of shared memory.
4. Send a message to the receiver (using a message queue). The message shall contain a field called size indicating how many bytes were read from the file.
5. Wait on the message queue to receive a message from the receiver confirming successful reception and saving of data to the file by the receiver.
6. Go back to step 3. Repeat until the whole file has been read.
7. When the end of the file is reached, send a message to the receiver with the size field set to 0. This will signal to the receiver that the sender will send no more.
8. Close the file, detach shared memory, and exit.

- receiver: this program shall implement the process that receives files from the sender process. It shall perform the following sequence of steps:

1. The program shall be invoked as `./recv` where recv is the name of the executable.
2. The program shall setup a chunk of shared memory and a message queue.
3. The program shall wait on a message queue to receive a message from the sender program. When the message is received, the message shall contain a field called

size denoting the number of bytes the sender has saved in the shared memory chunk.

4. If size is not 0, then the receiver reads size number of bytes from shared memory, saves them to the file (always called recvfile), sends message to the sender acknowledging successful reception and saving of data, and finally goes back to step 3.

5. Otherwise, if size field is 0, then the program closes the file, detaches the shared memory, deallocates shared memory and message queues, and exits.

•

When user presses Control-C in order to terminate the receiver, the receiver shall deallocate memory and the message queue and then exit. This can be implemented by setting up a signal handler for the SIGINT signal. Sample file illustrating how to do this have been provided (signaldemo.cpp).

Technical Details

The skeleton codes for sender and receiver can be found on Titanium. The files are as follows:

- sender.cpp: the skeleton code for the sender (see the TODO: comments in order to find out what to fill in)
- recv.cpp: the skeleton code for the receiver (see the TODO: comments in order to find out what to fill in).
- msg.h: the header file used by both the sender and the receiver

It contains the struct of the message relayed through message queues). The struct contains two fields:

- long mtype: represents the message type.
- int size: the number of bytes written to the shared memory.

In addition to the structure, msg.h defines macros representing two different message types:

- SENDER_DATA_TYPE: macro representing the message sent from sender to receiver. It's type is 1.

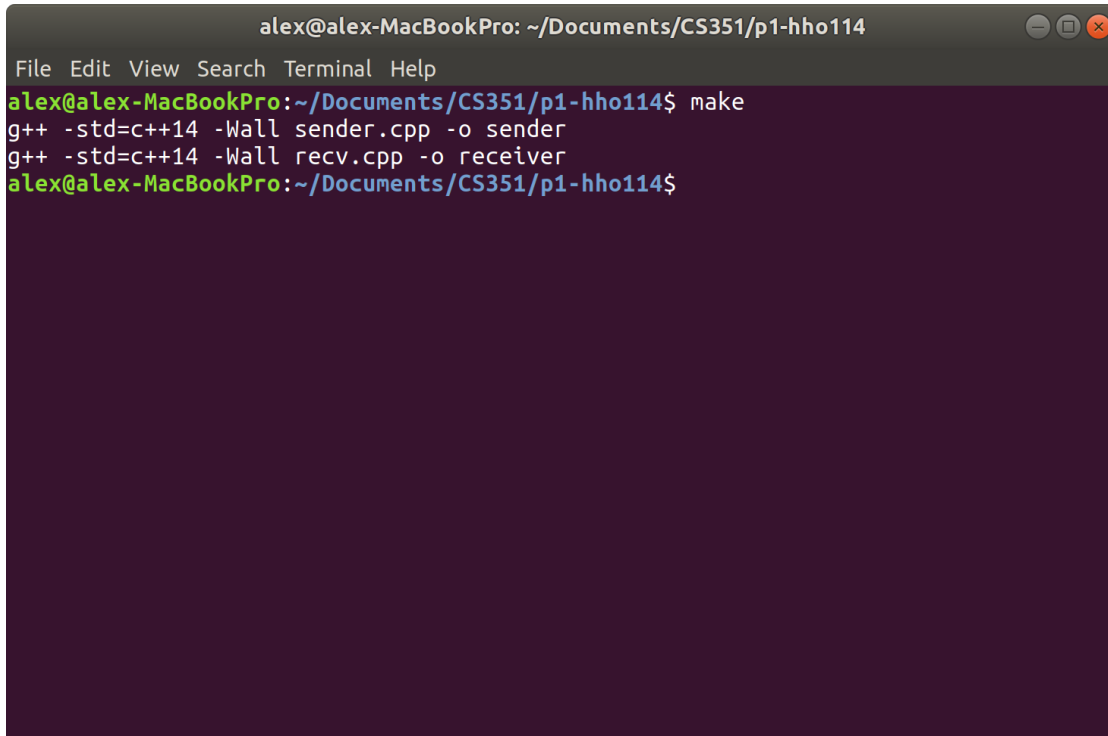
–

RECV_DONE_TYPE: macro representing the message sent from receiver to the sender acknowledging successful reception and saving of data.

- NOTE: both message types have the same structure. The difference is how the mtype field is set. Also, the messages of type RECV_DONE_TYPE do not make use of the size field.
- Makefile: enables you to build both sender and receiver by simply typing make at the command line.
- signaldemo.cpp: a program illustrating how to install a signal handler for SIGSTP signal sent to the process when user presses Control-C.

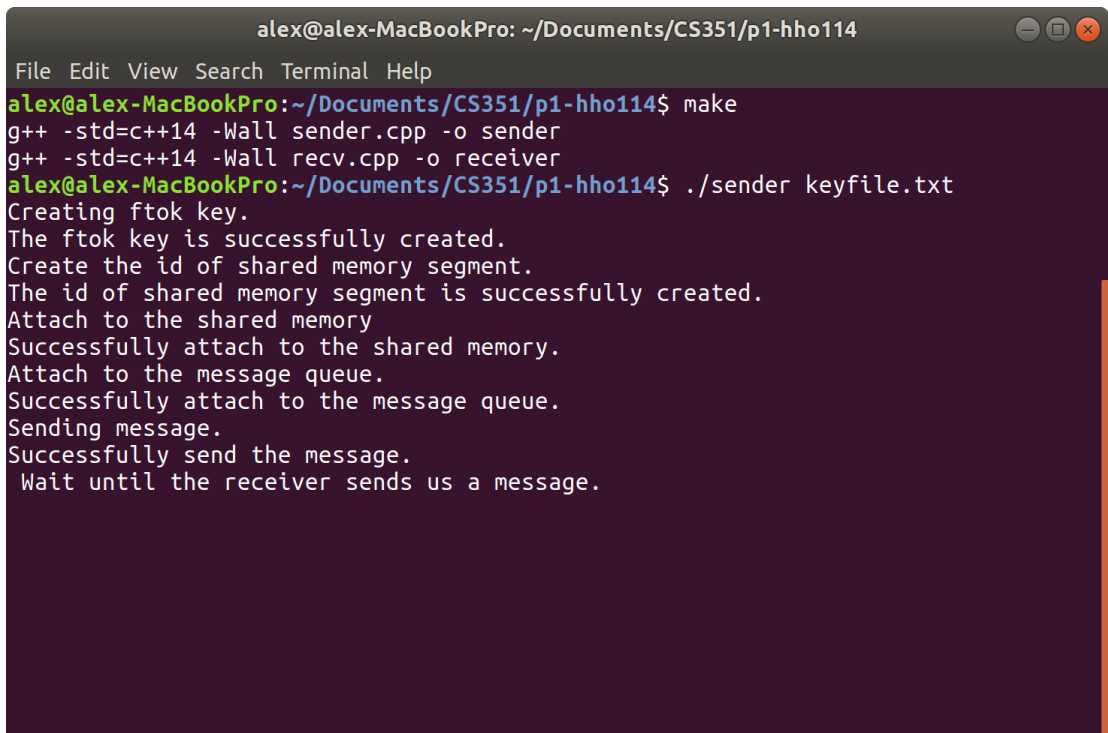
How to use your program

Type “make” to compile the sender.cpp and recv.cpp



```
alex@alex-MacBookPro: ~/Documents/CS351/p1-hho114
File Edit View Search Terminal Help
alex@alex-MacBookPro:~/Documents/CS351/p1-hho114$ make
g++ -std=c++14 -Wall sender.cpp -o sender
g++ -std=c++14 -Wall recv.cpp -o receiver
alex@alex-MacBookPro:~/Documents/CS351/p1-hho114$
```

Type “./sender keyfile.txt” to run sender process which sends the file and the message to the receiver.



```
alex@alex-MacBookPro: ~/Documents/CS351/p1-hho114
File Edit View Search Terminal Help
alex@alex-MacBookPro:~/Documents/CS351/p1-hho114$ make
g++ -std=c++14 -Wall sender.cpp -o sender
g++ -std=c++14 -Wall recv.cpp -o receiver
alex@alex-MacBookPro:~/Documents/CS351/p1-hho114$ ./sender keyfile.txt
Creating ftok key.
The ftok key is successfully created.
Create the id of shared memory segment.
The id of shared memory segment is successfully created.
Attach to the shared memory
Successfully attach to the shared memory.
Attach to the message queue.
Successfully attach to the message queue.
Sending message.
Successfully send the message.
Wait until the receiver sends us a message.
```

Open second terminal and type “./receiver” to run receiver process.

```
alex@alex-MacBookPro: ~/Documents/CS351/p1-hho114
File Edit View Search Terminal Help
alex@alex-MacBookPro:~/Documents/CS351/p1-hho114$ ./receiver
Creating ftok key.
ftok key is successfully created.
Create the id of shared memory segment.
The id of shared memory segment is successfully created.
Attach to the shared memory
Successfully attach to the shared memory.
Attach to the message queue.
Successfully attach to the message queue.
Read new message.
Successfully read a new message.
Ready to read the next file chunk.
Send an empty message back.
Successfully send empty message back.
Read new message.
Successfully read a new message.
Detach from shared memory.
Successfully detach from shared memory.
Deallocate the shared memory chunk.
Successfully deallocate the shared memory chunk.
Deallocate the message queue.
Successfully deallocate the message queue.
*****Successfully Inter Process Communication*****
alex@alex-MacBookPro:~/Documents/CS351/p1-hho114$
```

Switch back to previous terminal, check the sender process finish detach from memory.

```
alex@alex-MacBookPro: ~/Documents/CS351/p1-hho114
File Edit View Search Terminal Help
alex@alex-MacBookPro:~/Documents/CS351/p1-hho114$ ./sender keyfile.txt
Creating ftok key.
The ftok key is successfully created.
Create the id of shared memory segment.
The id of shared memory segment is successfully created.
Attach to the shared memory
Successfully attach to the shared memory.
Attach to the message queue.
Successfully attach to the message queue.
Sending message.
Successfully send the message.
Wait until the receiver sends us a message.
Successfully receive the message.
Sending the Empty message.
Successfully send the empty message.
Detach from shared memory.
Successfully detach from shared memory.
alex@alex-MacBookPro:~/Documents/CS351/p1-hho114$
```

Design of your program

Create file name keyfile.txt include the string “Hello World”.

Sender.cpp

- Create **key** by using **ftok** and **keyfile.txt**
- Get the **smhid** of the shared memory segment. The size of the segment must be **SHARED_MEMORY_CHUNK_SIZE**
- Attach **sharedMemPtr** to the shared memory
- Get the message queue id by using **msqid**
- Store the IDs and the pointer to the shared memory region in the corresponding parameters
- Send a message to the receiver telling him that the data is ready (message of type **SENDER_DATA_TYPE**)
- Wait until the receiver sends us a message of type **RECV_DONE_TYPE** telling us that he finished saving the memory chunk.
- Once we have finished sending the file. Lets tell the receiver that we have nothing more to send. We will do this by sending a message of type **SENDER_DATA_TYPE** with size field set to 0.
- Detach **sharedMemPtr** from shared memory

Receiver.cpp

- Create **key** by using **ftok** and **keyfile.txt**
- Get the **smhid** of the shared memory segment. The size of the segment must be **SHARED_MEMORY_CHUNK_SIZE**
- Attach **sharedMemPtr** to the shared memory
- Get the message queue id by using **msqid**
- Store the IDs and the pointer to the shared memory region in the corresponding parameters
- Receiving the message
- Save the shared memory to file
- Tell the sender that we are ready for the next file chunk. I.e. send a message of type **RECV_DONE_TYPE**
- Detach from shared memory segment, and deallocate shared memory and message queue (i.e. call **cleanup**)