CPSC 351 Programming Assignment 1

By Joseph Luong, Jyo Jain, Albert Dang, Ben Cheng

Introduction

This software exemplifies an understanding of the following concepts:

1. IPC principles
2. shared memory
3. message queues
4. signals
5. shared memory and message queues used to implement a practical application where the sender process sends information to the receiver process

System Overview

The purpose of this software is to implement an application which synchronously transfers files between two processes, a sender program, and a receiver program.

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.

1. When user presses Control-C 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).

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 keyfile.txt where sender is the name of the executable and keyfile.txt is the name of the file to transfer.

2. The program attaches to the shared memory segment and connects 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.

Design Considerations

System Architecture

The program is split into four modules: 1) message queue/memory segment connection, 2) message buffer for storage and sending/receiving, 3) memory clean up, and 4) command line argument.

Cohesion is a measure that defines the degree of intra-dependability within elements of a module. The cohesion types in this program are functional and procedural cohesion. The elements of each sender module have functional cohesion because they contribute to a single well-defined purpose. The shared memory and message connection, message buffer, and command line modules all have procedural cohesion because the order in which each module performs a task affects the outcome. For example, if ftok() was called before fopen(), the user may encounter a non-existent file that prevents the use of files in programs.

Coupling is a measure that defines the level of inter-dependability among modules of a program. Both programs use all forms of coupling: content coupling, common coupling, control coupling, stamp coupling, and data coupling. The send buffer module is content coupled with the receive buffer module because both directly access and modify a shared text file. Similarly, both modules are also considered stamp coupled because they share a common data structure "message."

The send and receive modules are control coupled because their respective programs terminate message sending when their conditions have been met.

Finally, the memory cleanup modules in both the send and receive programs are data coupled because they detach and deallocate the memory space after message sending is complete.

Detailed System Design

Receiver

```
        a. init()
Enter RECV <FILE_NAME>
SEEK <FILE_NAME> in directory
If <FILE_NAME> found THEN
Create Key for File
SEEK Shared memory ID
if (key matched) THEN
Get Shared memory ID
else
print error message
exit
Attach to shared memory segment variable created
Attach to message queue segment
if nothing in memory segment
print error message
Else
print error message
Call mainLoop
Endif
        b. mainLoop()
OPEN a file for writing received message into
IF new file does not exist THEN
PRINT error message
EXIT
Save message size of file from function msgrcv
IF message equals 0 THEN
EXIT
WHILE message size does not equal 0
IF message size does not equal 0 THEN
WRITE message into the file we opened
SEEK if there is more to read/write
ELSE
CLOSE file
        c. cleanUp()
     Detach and destroy shared memory and message queue
     IF detach error
          Error message
          ENDIF
     IF destroy shared memory error
          Error message
          ENDIF
     IF destroy shared message queue error
          Error message
          ENDIF
```

2. **Psuedocode**

init(){

Open or Create file “keyfile.txt”;

```
Write "Hello World!" in keyfile.txt;
Close file;
Create key using keyfile.txt;
Allocate a piece of shared memory of size SHARED_MEMORY_CHUNK_SIZE;
Attach shared memory pointer to shared memory;
Create a message queue;
Call mainLoop();
}
mainLoop(){
Open new file for writing received message;
Output error if file not opening;
See if there is a message in the file;
If there is no message{
exit;
            }
While there are message to read/write{
if there are more messages to read/write{
write message into our new file;
check if there are more messages after the message we read;
}
else
close file;
}
}
cleanUp(){
            Detach from shared memory;
            if(memory address does not match shared address) { print error message }
            Destroy shared memory;
            if(ID does not match shared ID) { print error message }
            Destroy message queue;
            if(ID does not match shared ID) { print error message }
            }
```

Sender

1. **Structured English**
    a. Init()
    b. Enter SEND <FILE_NAME>

```
SEEK  <FILE_NAME> in directory
IF <FILE_NAME> found THEN
        Create key for file
        SEEK shared memory ID
        IF key matched THEN
            Get shared memory ID
        ELSE
            PRINT error message
            END IF
            Attach to shared memory segment
            Attach to message queue
        IF no attachment
                Print error message
                    END IF

    ELSE
            PRINT error message
    Call SEND <FILE_NAME>
    ENDIF
```

c. Send()

```
    • Open file
      IF <FILE_NAME> not open THEN
                PRINT error message
                ENDIF
      DO Send Message
      WHILE Not end of file
      UNTIL
                Error reading from shared memory
                Error sending message to receiver
                Error sending message from receiver
      Empty message queue
      IF message queue empty THEN
      ENDIF
      Close file
```

e. cleanUp()

```
    i.  Detach from shared memory and message queue
        IF detach error
                Error message
                ENDIF
```

2. **Pseudo-code**
   a. Void function Init

```
void function init
{
create file to represent memory;
check if stream is good enough to work;
for (int i = 0; i < message count; i++)
{
Print message;
}
create key to shared memory segment;
get address of shared memory segment;
attach to shared memory;
attach to message queue;
}//end init
```

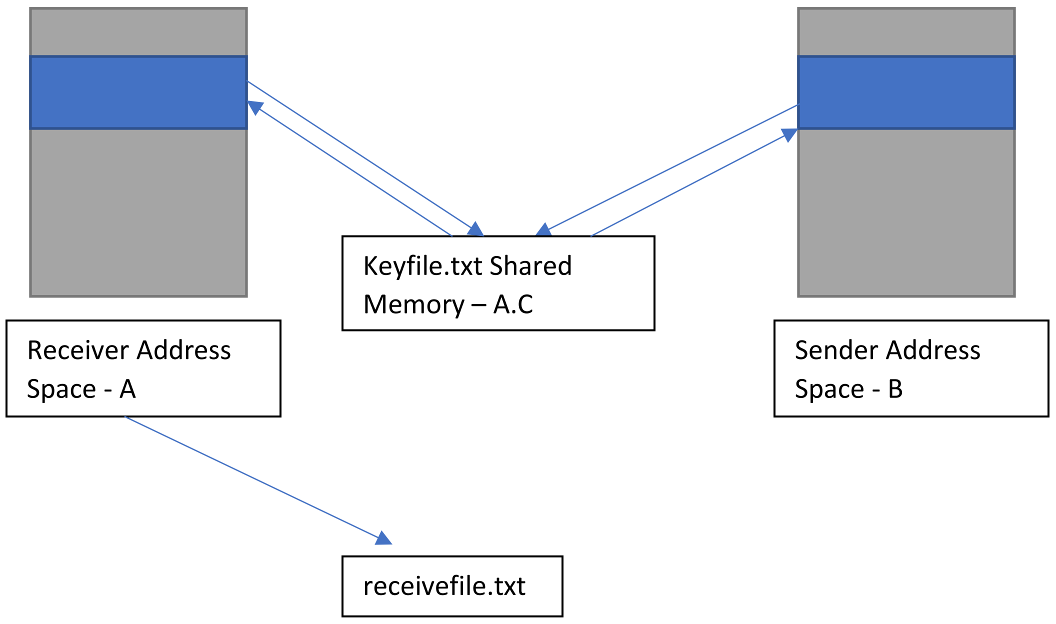
b. Void Send()

```
void send
{
Open shared file;
Create send message;
Create receive message;
Check if file is open;
if(no file) { report error message; }
while(not end of file)
{
read message;
if (message size last read < 0) { report error message;}
send message;
if(no more messages to append) { report send error message }
if(no more messages to store in buffer) { report receive error message }
message size=0; //let receiver no there are no more messages
if(no more messages to append)
{
report send error message;
}
close the file;
}
```

c. Cleanup()

```
void function cleanUp
{
Detach from shared memory;
if(memory address does not match shared address) { print error message }
}
```

3. **Data Flow Diagram**


Keyfile.txt Shared Memory – A.C
Receiver Address Space - A
Sender Address Space - B
receivefile.txt