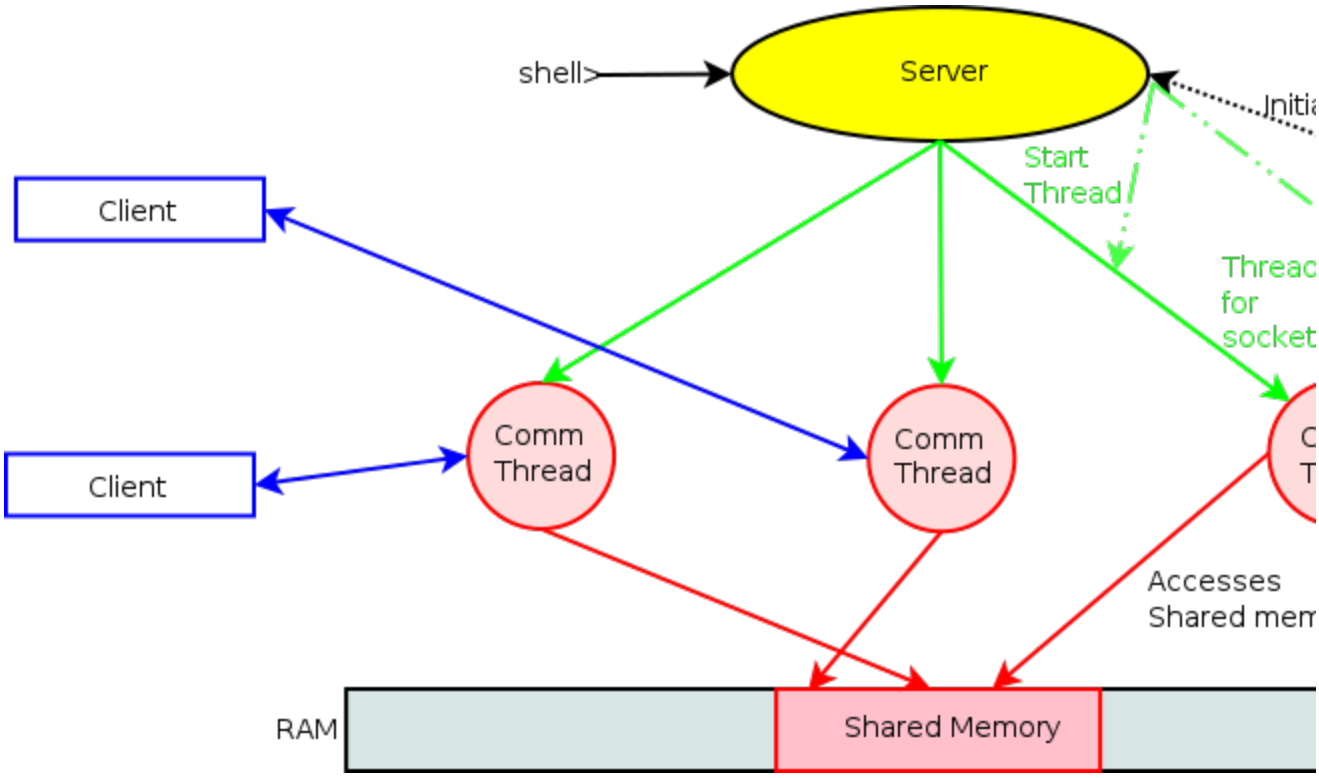
Program #3

DIPC - Distributed Interprocess Communication tool:
Due: 4/6/16.

The third programming assignment is to write a distributed interprocess communication tool. This can be thought of as a middleware shared memory system. The DIPC tool will provide shared memory for storage and interprocess communication through a socket interface. The shell will start a server process which will setup the shared memory, the required semaphores to control access to the shared memory and create the sockets. Since multiple processes will have access to the shared memory, the standard shared resource problems will arise. You must resolve the multiple access (or starvation and deadlock) issues that can occur. This program will allow a group of process to communicate with each other; each one connected as a client to a central server. Thus, the clients only need simple client socket code. A simple chat program would use something like this.

Use of the system will be:

1. The user will start the server on the command line specifying number of mailboxes, size of a mail box, the communication port and the size of a communication packet.
2. A client program will connect to the server and get handed off to a thread to manage the communication.
3. The communication thread will access the shared memory in a safe manner. It will move data back and forth between the shared memory and the client process (via a socket).
4. When finished, the user will issue a server shutdown.

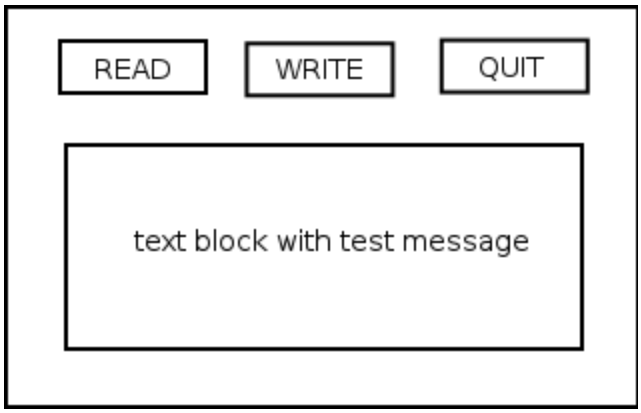


The communication thread will be both reading and writing to the shared memory. Since there can be multiple threads running, there can be multiple readers and writers. You will allow more than one reader but only one writer in a block. You should use pthreads for the thread library. You may use the shared memory aspect of pthreads or you can use the System V IPC Shared Memory - your design decision.

Demonstration of your solution:

To demonstrate you will need to produce some clients that can show the system is working. I have not specified this part and you are free to address this as you wish. However, some have asked for suggestions. Here are a couple of ideas. Now that does not mean you have to do it this way.

- Suggested client (version 1): Modify the Python client code below to read in a command from the user ("read", "write"). If "write" then accept a block of text from the user with max block size of the packet size. This can be implemented in a single prompt to the user or two prompts to the user. Send the message. If it is a write, then the communication thread will write the block of text. If it is a read, the thread will read and return the data to the python client.
- Suggested client (version 2): Produce a Python-Tk window that has a small text window and some buttons. A read button, write button and quit button. These will send and receive text in the little window.



- The point is that you are writing something to test your code. I plan to use my own clients to test your code.

Required elements:

- DIPC initialization
  - Usage: dipc <number of mailboxes> <size of mailbox in kbytes> <port> <size of packet in kbytes>
  - Starts the dipc server
  - Create k blocks
    - k = number of mailboxes requested
    - block size = size of the mailbox
  - Create the socket server which can hand
  - Returns (exits) with error message if server init fails, otherwise runs in the background like a daemon.
- DIPC removal
  - Usage: dipcrm <port>
  - Clean up semaphores and shared memory if required.
  - Will kill server process (stop the socket accept loop).
- DIPC server
  - Will listen for connections.
  - Will accept connections and hand them off to a thread.
  - Will exit when it gets the exit command from dipcdel.
  - Will establish the shared memory component.
- DIPC Communication thread
  - Interact with client over socket setup by server.
  - Interact with shared memory and solving the critical section problem (no starvation, no deadlocks).
- DIPC datagram
  - First line will have a "r" for read, "w" for write, or "q" for quit and the mailbox number. A line is text and then newline - correct?
  - Note that "q" will kill the connection to the specific client, not kill the server.
  - Following the first line is data (void for "q").

Solution method:

- Start with the socket code you had in dsh or use the code below. You will need to modify this to get the connection handled by a thread. Take the code after the connection accepted line and place it in a function. You can hand this function to a thread. The accept line should be in a loop to open multiple connections. You can use the python code below to check the connections.

```
/*
    C socket server example
*/

#include<stdio.h>
#include<string.h>    //strlen
#include<sys/socket.h>
#include<arpa/inet.h> //inet_addr
#include<unistd.h>    //write

int main(int argc , char *argv[])
{
    int socket_desc , client_sock , c , read_size;
    struct sockaddr_in server , client;
    char client_message[2000];

    //Create socket
    socket_desc = socket(AF_INET , SOCK_STREAM , 0);
    if (socket_desc == -1)
    {
        printf("Could not create socket");
    }
    puts("Socket created");

    //Prepare the sockaddr_in structure
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons( 50007 );

    //Bind
    if( bind(socket_desc,(struct sockaddr *)&server , sizeof(server)) < 0)
    {
        //print the error message
        perror("bind failed. Error");
        return 1;
    }
    puts("bind done");

    //Listen
    listen(socket_desc , 3);

    //Accept and incoming connection
    puts("Waiting for incoming connections...");
    c = sizeof(struct sockaddr_in);

    //accept connection from an incoming client
    client_sock = accept(socket_desc, (struct sockaddr *)&client, (socklen_t*)&c);
    if (client_sock < 0)
    {
        perror("accept failed");
        return 1;
    }
    puts("Connection accepted");

    //Receive a message from client
    while( (read_size = recv(client_sock , client_message , 2000 , 0)) > 0 )
    {
        //Send the message back to client
        write(client_sock , client_message , strlen(client_message));
    }

    if(read_size == 0)
    {
        puts("Client disconnected");
        fflush(stdout);
```

```
    }
    else if(read_size == -1)
    {
        perror("recv failed");
    }

    return 0;
}
```

- You can also find many examples of threaded echo servers: http://www.binarytides.com/server-client-example-c-sockets-linux/
- You can try using telnet to talk to the server above: telnet localhost 50007 or write your own:
- A simple client (Python) is provided below. This can interact with the echo server above.

```
from socket import *
HOST = 'localhost'
PORT = 50007
s = socket ( AF_INET , SOCK_STREAM )
s.connect((HOST, PORT))
while True :
  message = raw_input("> ")
  if message == 'q': break
  if message != '':
    s.send(message)
    data = s.recv(1024)
    print 'Received' , repr(data)
s.close()
```

- In the server, dynamically allocate a block of memory and modify your communication thread function to access this. Read and write to this block.
- Add the synchronization functionality. Start with the reader-writer algorithms. You will protect the critical section with a semaphore - but this introduces starvation issues.
- You will need to have some queues or other approaches to make sure starvation is not a problem.
- A timeout (or something) is necessary to make sure deadlock is not a problem.
- You may write this assignment in C/C++/Python/Go/Haskell.

Testing:

- Using the python clients, get several of them sending messages back and forth.
- Again, try to break the code.
- Just because human interaction is so slow and the chance of actual synchronization problems is very very low, this does not mean that there are no problems. Testing asynchronous process interaction is tricky and subtle.

Notes and hints:

- The exit from the server will be tricky. The accept() function is blocking and waits until a client connects. Existing clients cannot easily interrupt this blocking call. The standard way is to address this problem is to have a client change the loop condition (with the "q") and issue a second client connection so the accept() block is released and one can exit the server.
- To get daemon functionality you can fork and detach or possibly use the daemon() function.

Submission contents:

- Documentation
  - Format: PDF, lower case filename, filename = prog3.pdf
- Main programs: dipc, dipcrm .
- Any required external functions.
- Any include files you use (other than the standard ones).
- The Makefile to build the program.

Grading approach :

- copy from the submit directory to my grading directory
- tar -xzf prog3.tgz
- cd prog3
- make
- GRADE
- assign a grade
- cd ..
- rm -rf prog3

Submission method: You will use the submit page to submit your program. The files need to be tarred and gzipped prior to submission. Please empty the directory prior to tar/zip. [You will tar up the directory containing the files: `tar -czf prog3.tgz prog3 .`]