Programming Assignment 6

- **This is an extra-credit assignment (3%). The due day: May 12.**
- **No credit for a program that does not compile or does not run.**

This assignment requires you to write a *server* program and two *client* programs (*Client-1* and *Client-2*) that use ***message queue*** and *message send/receive* for clients to request services and for the server to provide . Three programs should be written in C, and compiled and run in Athena that supports message queues.

1. The *Server* process creates a message queue using **`msgget()`**, and then waits for requests from clients. A request from a client is represented as a message containing a string of characters to be converted to its uppercase equivalent. The *Server* uses **`msgrcv()`** to receive a message, converts the string contained in the message body into its uppercase equivalent, composes a return message containing the uppercase string, sends it back to the client using **`msgsnd()`**, and then waits for the next request. Upon receiving a zero-sized message (message with empty body) from a client, the *Server* removes the message queue via the following call, and then quits:

   `msgctl``(mid, IPC_RMID, (struct msqid_ds *) 0);`

2. On the other hand, *Client-1* process gains access to the message queue created by the server using **`msgget()`**. It opens two input files, **`infile1`** and **`infile2`**, that are provided at the command line, reads the string from **`infile1`**, and then composes a message containing the string from **`infile1`**, and sends the message to the message queue using **`msgsnd()`** for processing by the server. Afterward, *Client-1* uses **`msgrcv()`** to receive the message containing the converted string from the *Server*, and displays the uppercase string to *stdout*. *Client-1* repeats the same process again for the string in **`infile2`**, and then quits.

3. The second client, *Client-2*, also gains access to the message queue through **`msgget()`**. It opens the input file **`infile3`** provided at the command line, repeats the same sequence of actions like *Client-1* to have the string converted by the *Server*, and displays the converted string to *stdout*. Afterward, *Client-2* composes a second message with empty body and then sends it to the *Server* in the following manner, initiating the *msgQ* terminating process.

   `msgsnd(mid, &msg, 0, 0)`

4. In both the *Server* and *Client-1*/*Client-2* programs, you need to include the following headers, constant, defined type, and some variables.

```
<stdio.h>, <unistd.h>, <stdlib.h>, <sys/types.h>, <sys/ipc.h>,
<sys/msg.h>, <string.h>, <ctype.h>, <fcntl.h>

#define   SERVER  1L
typedef  struct  {
   long         msg_to;
   long         msg_fm;
   char         buffer[BUFSIZ]
} MESSAGE;

int         mid;
key_t       key;
struct      msqid_ds   buf;
MESSAGE     msg;
```

   A message has a *header* (`msg_to`, `msg_fm`) and a *body* (`buffer`). "`msg_to`" and "`msg_fm`" are used to indicate the recipient and the sender of the message, respectively. When a client sends a message to the server, "`msg_to`" has the value of `SERVER`, and "`msg_fm`" has the value of the client's PID. It will be just the other way around when the server sends a message back to the client.

5. An IPC identifier is derived from a *key* value. There is *one-to-one* relationship between a key and an identifier for the **`msgget`** system call. Different processes using the same key will always get to the same IPC resource. A file-to-key interface, **`ftok`** (C library function) is the common method of having different processes obtain a correct key before they make an **`msgget`** system call. In the server program, use the following to create a message queue

```
key = ftok(".", 'z');
mid = msgget(key, IPC_CREAT|0660);
```

You can use **msgctl**(mid, IPC_STAT, &buf) to get the following message queue related information:

```
Struct   msqid_ds  {
  struct  ipc_perm    msg_perm;     // ipc permissions
  struct  msg         *msg_first;   // ptr to first msg
  struct  msg         *msg_last;    // ptr to last msg
  ulong               msg_cbytes;   // current # bytes on Q
  ulong               msg_qnum;     // # of msg in Q
  ulong               msg_qbytes;   // max # of bytes in Q
  pid_t               msg_lspid;    // pid of last msgsnd
  pid_t               msg_lrpid;    // pid of last msgrcv
    ……
};
```

The receive and send calls in the *Server* program look like the following:

```
msgrcv(mid, &msg, sizeof(msg), SERVER, 0);
msgsnd(mid, &msg, sizeof(msg.buffer),0). //Linux does not like "sizeof(msg)" in msgsnd
```

6. In the client programs, the same key should be used to gain access to the same message queue

```
key = ftok(".", 'z');
mid = msgget(key, 0);
```

The message send and receive calls are similar to those of the *Server*, except that in the receive call, the `msgtyp` value (fourth parameter) is "client_pid", rather than "SERVER".

The clients should get the initial string input from the input files (**infile1**, **infile2**, **infile3**) and then use it to initialize the message body (**read(fd, msg.buffer, BUFSIZ)**) before sending it to the message queue for processing by the server. For both *Client-1* and *Client-2*, **printf** the input string before sending it to the server. **printf** the processed string (uppercase) after receiving it back from the server.

7. You run the *Server* process in the background first (**&**) and then run *Client-1* process next and *Client-2* finally. Use "**ipcs**" to find out the message queue the server initially created and subsequently removed.

```
gcc   server.c  -o  server
gcc   client1.c  -o  client1
gcc   client2.c  -o  client2
ipcs
server&
ipcs
client1  infile1 infile2
client2  infile3
ipcs
```

Test the programs with your own data first, and then make sure that your programs work for the following strings in **infile1**, **infile2**, and **infile3**, respectively:

infile1: "processes can use message passing as an ipc mechanism to exchange information with each other."

infile2: "a message header indicates who is the sender and who is the recipient of the message."

infile3: "clients and servers use msg send and receive system calls to request and provide services."

8. Submission Requirements. Your programs must include adequate commenting (**points deduction for programs with inadequate comments**). Submit your three source files to Assignment 6 in SacCT.