Experiment #2
Multi-Processing Environment under Unix/Linux
Process Management for Real-Time Applications

1. Prerequisite

ENCS 538, C programming language, basics of inter-process communications under Unix/Linux.

2. Objectives

- To learn how to create and manage processes on a multi-tasking environment.
- To learn how to establish inter-task communication between the different tasks.
- To learn and implement different control algorithms in embedded systems using software and programming in a multi-tasking environment.

3. Background

The realization and control of embedded system needs some tools to manage multitasks and concurrent processes. This experiment aims to introduce some of these concepts using Linux as multi-tasking environment.

3.1 FIFOs

A **FIFO** ("First In, First Out") is sometimes known as a **named pipe**. It's like a **pipe**, except that it has a name and process pipes can only be shared among the parent process and its "children". Also, it is a one-way (half-duplex) flow of data. But unlike pipes, a FIFO has a pathname associated with it, allowing unrelated processes to access a single FIFO.

A FIFO is created by the mkfifo function.

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname , mode_t mode);
```

// Return: 0 if OK , -1 on error

The *pathname* is a normal Unix pathname , and this is the name of the FIFO.
The *mode* argument specifies the file permission bits, similar to the second argument to open.

The **mkfifo** function implies **O_CREAT | O_EXCL**. It creates a new FIFO or returns an error of **EEXIST** if the named FIFO already exists. If the creation of a new FIFO is not desired , call open instead of **mkfifo**. To open an existing FIFO or create a new FIFO if it does not already exist , call **mkfifio**, check for an error of **EEXIST**, and if this occurs, call **open** instead.

The **mkfifo** command also creates a FIFO. This can be used from shell scripts or from the command line.

Once a FIFO is created , it must be opened for reading or writing , using either the **open** function , or one of the standard I/O open functions such as **fopen**. A FIFO must be opened either read-only or write-Only. It must not be opened for read-write , because a FIFO is half duplex.

A **write** to pipe or FIFO always appends the data , and a read always returns what is at the beginning of the pipe of FIFO.

3.2 Message Queues

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. We'll call the message queue just a queue and its identifier a queue ID.

The Single UNIX Specification includes an alternate IPC message queue implementation in the message-passing option of its real-time extensions. We do not cover the real-time extensions in this text.

A new queue is created or an existing queue opened by **msgget**. New messages are added to the end of a queue by **msgsnd**. Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to **msgsnd** when the message is added to a queue. Messages are fetched from a queue by **msgrcv**. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

Each queue has the following **msqid_ds** structure associated with it:

```
struct msqid_ds {
  struct ipc_perm  msg_perm;
  msgqnum_t        msg_qnum;     /* # of messages on queue */
  msglen_t         msg_qbytes;   /* max # of bytes on queue */
  pid_t            msg_lspid;    /* pid of last msgsnd() */
  pid_t            msg_lrpid;    /* pid of last msgrcv() */
  time_t           msg_stime;    /* last-msgsnd() time */
  time_t           msg_rtime;    /* last-msgrcv() time */
  time_t           msg_ctime;    /* last-change time */
  .
  .
  .
};
```

This structure defines the current status of the queue. The members shown are the ones defined by the Single UNIX Specification. Implementations include additional fields not covered by the standard.

The first function normally called is **msgget** to either open an existing queue or create a new queue.

```
#include <sys/msg.h>

int msgget(key_t key, int flag);
```

Returns: message queue ID if OK, 1 on error

On success, **msgget** returns the non-negative queue ID. This value is then used with the other three message queue functions.

The **msgctl** function performs various operations on a queue.

```
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf );
```

Returns: 0 if OK, 1 on error

The **cmd** argument specifies the command to be performed on the queue specified by **msqid**.

IPC_STAT
Fetch the **msqid_ds** structure for this queue, storing it in the structure pointed to by *buf*.

IPC_SET
Copy the following fields from the structure pointed to by *buf* to the **msqid_ds** structure associated with this queue: msg*_perm.uid, msg_perm.gid, msg_perm.mode, and msg_qbytes.* This command can be executed only by a process whose effective user ID equals *msg_perm.cuid or msg_perm.uid* or by a process with superuser privileges. Only the superuser can increase the value of *msg_qbytes.*

IPC_RMID
Remove the message queue from the system and any data still on the queue. This removal is immediate. Any other process still using the message queue will get an error of **EIDRM** on its next attempted operation on the queue. This command can be executed only by a process whose effective user ID equals *msg_perm.cuid* or *msg_perm.uid* or by a process with superuser privileges.

Data is placed onto a message queue by calling **msgsnd**.

```
#include <sys/msg.h>

  int msgsnd(int msqid, const void *ptr, size_t nbytes , int flag);
```

Returns: 0 if OK, 1 on error

As we mentioned earlier, each message is composed of a positive long integer type field, a non-negative length (*nbytes*), and the actual data bytes (corresponding to the length). Messages are always placed at the end of the queue.

The ***ptr*** argument points to a long integer that contains the positive integer message type, and it is immediately followed by the message data. (There is no message data if nbytes is 0.) If the largest message we send is 512 bytes, we can define the following structure:

```
struct mymesg {
     long  mtype;      /* positive message type */
     char  mtext[512]; /* message data, of length nbytes */
   };
```

A *flag* value of **IPC_NOWAIT** can be specified. specifying **IPC_NOWAIT** causes **msgsnd** to return immediately with an error of **EAGAIN**. If **IPC_NOWAIT** is not specified, we are blocked until there is room for the message, the queue is removed from the system, or a signal is caught and the signal handler returns.

Messages are retrieved from a queue by **msgrcv**.

```
#include <sys/msg.h>

ssize_t msgrcv(int msqid, void *ptr, size_t nbytes
  , long type, int flag);
```

Returns: size of data portion of message if OK, 1 on error

As with ***msgsnd***, the *ptr* argument points to a long integer (where the message type of the returned message is stored) followed by a data buffer for the actual message data. **nbytes** specifies the size of the data buffer.

The *type* argument lets us specify which message we want.

type == 0 : The first message on the queue is returned.

type > 0 : The first message on the queue whose message type equals type is returned.

type < 0 : The first message on the queue whose message type is the lowest value less than or equal to the absolute value of type is returned.

A nonzero type is used to read the messages in an order other than first in, first out. For example, the type could be a priority value if the application assigns priorities to the messages. Another use of this field is to contain the process ID of the client if a single message queue is being used by multiple clients and a single server (as long as a process ID fits in a long integer).

4. Procedure

4.1 Part A: Named Pipes : FIFOs

1. Execute man **mknod** and man **mkfifo** and check how these commands can create a FIFO.
2. We intend to implement a client-server communication technique using FIFOs. Both the client and server applications run on the same platform. Briefly, the steps taken by the processes involved are as follows:
   - Server generates the public FIFO (available to all participating client processes).
   - Client process generates its own private FIFO.
   - Client prompts for, and receives, a shell command from the user.
   - Client writes the name of its private FIFO and the shell command to the public FIFO.
   - Server reads the public FIFO and obtains the private FIFO name and the shell command.
   - Server uses a popen-pclose sequence to execute the shell command. The output of the shell command is sent back to the client via the private FIFO.
   - Client displays the output of the command.
   - If the user types the command quit at the client's prompt, the client process exits. The server process continues to run indefinitely waiting for requests from the clients. It can be stopped or killed by the user.

3. Type the following program which creates the server part. Name the file server fifo.c:

```
/*
* The server process
*/
#include "local_fifo.h" void main(void)
{
        int     n, done, dummyfifo, publicfifo, privatefifo;
        struct message msg; FILE     *fin;
        static char     buffer[PIPE_BUF];

        /* Generate the public FIFO */
        mknod(PUBLIC, S_IFIFO | 0666, 0);

        /* Open the public FIFO for reading and writing */
        if ( (publicfifo = open(PUBLIC, O_RDONLY)) == -1 ||
        (dummyfifo     = open(PUBLIC, O_WRONLY | O_NDELAY)) == -1 )
        {
                perror(PUBLIC);
                exit(1);
        }

        /* message can be read from the PUBLIC pipe */
        while ( read(publicfifo, (char *) &msg, sizeof(msg)) > 0 ) {
                n = done = 0; /* clear counters / flags */
                do {
                    if ( (privatefifo = open(msg.fifo_name, O_WRONLY | O_NDELAY)) ==
                -1 )
                        sleep(3);
                    else {
```

```
                    fin  = popen(msg.cmd_line,  "r");  /*  execute  the  cmd  */
                   write(privatefifo,  "\n",  1);  /*  keep  output  pretty  */

                  while  ( (n  = read(fileno(fin),  buffer,  PIPE_BUF))  > 0  )
            {
                write(privatefifo,  buffer,  n);  /*  to  private  fifo  */
                memset(buffer,  0x0,  PIPE_BUF);  /*  clear  it out  */
             }
            pclose(fin);
            close(privatefifo);
            done  = 1;
          }
        }  while  ( ++n < 5  &&   ! done);

           if ( ! done  )
           write(fileno(stderr),  "\nSERVER  NEVER  accessed private  fifo\n",  48);
        }
}
```

4. Type the following program which creates the client part. Name the file **client fifo.c:**

```
/*
 *  The  client  process
 */

#include  "local_fifo.h" void

main(void)
{
   int                       n,  privatefifo,  publicfifo;
   static  char              buffer[PIPE_BUF];
   struct  message  msg;

   /*Make  the  name  for  the  private  fifo  */
   sprintf(msg.fifo_name,  "/tmp/fifo%d",  getpid());

   /*  Generate  the  private  FIFO  */
   if ( mknod(msg.fifo_name,  S_IFIFO  | 0666,  0)  < 0  ) {
      perror(msg.fifo_name);
      exit(1);
   }

   /*  Open  the  public  FIFO  for  writing  */
   if ( (publicfifo  = open(PUBLIC,  O_WRONLY))  == -1  ) {
      perror(PUBLIC);
      exit(2);
   }

   while  ( 1  ) {  /*  FOREVER  */
      write(fileno(stdout),  "\ncmd>",  6);  /*  prompt  */
```

```
        memset(msg.cmd_line, 0x0, B_SIZ); /* clear first */
        n = read(fileno(stdin), msg.cmd_line, B_SIZ); /* Get cmd */

        if ( ! strncmp("quit", msg.cmd_line, n - 1) ) /* EXIT ? */
          break;

        write(publicfifo, (char *) &msg, sizeof(msg));

        /* Open private fifo to read returned command output */
        if ( (privatefifo = open(msg.fifo_name, O_RDONLY)) == -1 ) {
          perror("msg.fifo_name");
          exit(3);
        }

        /* Read private FIFO and display on standard error */
        while ( (n = read(privatefifo, buffer, PIPE_BUF)) > 0 )
          write(fileno(stderr), buffer, n);

        close(privatefifo);
      }

      close(publicfifo);
      unlink(msg.fifo_name); /* Remove */
    }
```

5. Type the following header file which is common between the server and client processes. Name the file local **fifo.h:**

```
/*
  * local header file for fifo client-server
  */
#ifndef       LOCAL_F_H_
#define       LOCAL_F_H_

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <limits.h>
#include <stdlib.h>

#define PUBLIC "/tmp/PUBLIC"
#define B_SIZ (PIPE_BUF / 2)
struct message {
    char       fifo_name[B_SIZ];
    char       cmd_line[B_SIZ];
};
#endif
```

6. Compile the server part to create the program **server fifo**. Run it in the background in one of the shell terminals.
7. Compile the client part to create the program **client fifo**. Run multiple clients in different shell terminals.
8. Run different commands in the client processes, such as **ls, pwd** and check the output.
9. If you run a command that doesn't exist in one of the client terminals, check the message that you get in the server terminal.

TODO:

It will be given during the lab based on material covered in this section.

4.2 Part B: Inter-process communication

1. Execute a man **ipcs** and discover what options can be used with it. Special attention should be given to the **-m, -s** and **-q** options that print information about the active shared memory segments, active semaphores and active message queues respectively.

2. Execute the command **ipcs** and notice how many message queues, semaphores and shared memory segments are currently in use. Notice the different columns that are printed on the screen.

3. To remove a message queue, semaphore or shared memory segment, use the command **ipcrm** with the options **-q, -s** and **-m** respectively. Execute man **ipcrm** and discover the other options.

Message Queues

1. Execute man **msgget**, man **msgsnd**, man **msgrcv**, man **msgctl** and check what these functions do.
2. We intend to implement a client-server communication technique using Message Queues. Both the client and server applications run on the same platform. Briefly, the steps taken by the processes involved are as follows:

    - Client invokes the server process.
    - Client prompts for, and receives, a string from the user.
    - Client issues a message to the server containing the string.
    - Server reads the message, converts the string to upper case and sends it back to the client through a message.
    - Client displays the string received from the server.

3. Type the following program which creates the server part. Name the file **server queue.c**:

```
/*
 * SERVER-receives messages from clients
 */
#include "local_queue.h"

int main(int argc, char *argv[ ])
{
   int        mid, n;
```

```
    MESSAGE  msg;
    void         process_msg(char  *,  int);

    if (argc  !=  3)  {
       fprintf(stderr,  "Usage:  %s  msq_id  &\n",  argv[0]);
       return  1;
    }

   mid  = atoi(argv[1]);

   while  (1)  {
      if ((n  = msgrcv(mid,  &msg,  BUFSIZ,  SERVER,  0))  == -1  ) {
         perror("Server:  msgrcv");
         return  2;
      }

      else  if (n  == 0)
         break;

      process_msg(msg.buffer,  strlen(msg.buffer));
      msg.msg_to  = msg.msg_fm;
      msg.msg_fm  = SERVER;
      n += sizeof(msg.msg_fm);

      if (msgsnd(mid,  &msg,  n,  0)  == -1  ) {
         perror("Server:  msgsnd");
         return  3;
      }
   }
   msgctl(mid,  IPC_RMID,  (struct  msqid_ds  *)  0  );
   exit(0);
}

/*
  * Convert  lowercase  alphabetics  to  uppercase
  */
void  process_msg(char  *b,  int  len)
{
   int  i;

        for  (i = 0;  i < len;  ++i)
           if (isalpha(*(b  + i)))
         *(b  + i) = toupper(*(b  + i));
}
```

4. Type the following program which creates the client part. Name the file **client queue.c**:

```
/*
  *   CLIENT  ...  sends messages  to  the  server
  */
```

```
#include "local_queue.h"

int main( )
{
   key_t            key; pid_t
                    cli_pid; int
                    mid, n;
   MESSAGE          msg;
   static char m_key[10];
   cli_pid = getpid( );

   if ((key = ftok(".", SEED)) == -1) {
      perror("Client: key generation"); return 1;
   }
   if ((mid = msgget(key, 0 )) == -1 ) {
      mid = msgget(key,IPC_CREAT | 0660);

      switch (fork()) {
      case -1:
         perror("Client: fork");
         return 2;

      case 0:
         sprintf(m_key, "%d", mid);
         execlp("./server", "server", m_key, "&", 0);
         perror("Client: exec");
         return 3;
      }
   }

   while (1) {
      msg.msg_to = SERVER; msg.msg_fm =
      cli_pid; write(fileno(stdout), "cmd> ", 6);
      memset(msg.buffer, 0x0, BUFSIZ);

      if ( (n = read(fileno(stdin), msg.buffer, BUFSIZ)) == 0 )
         break;

      n += sizeof(msg.msg_fm);

      if (msgsnd(mid, &msg, n, 0) == -1 ) {
         perror("Client: msgsend");
         return 4;
      }

      if( (n = msgrcv(mid, &msg, BUFSIZ, cli_pid, 0)) != -1 )
         write(fileno(stdout), msg.buffer, n);
   }
   msgsnd(mid, &msg, 0, 0);
   return 0;
 }
```

5. Type the following header file which is common between the server and client processes. Name the file **local queue.h**:

```
/*
 * Common header file for Message Queue Example
 */

#ifndef   LOCAL_Q_H_
#define   LOCAL_Q_H_

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <ctype.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>

#define SEED     'g'/* seed for ftok */
#define SERVER 1L /* message for the server */

typedef struct {
   long        msg_to; /* Placed in the queue for */
   long        msg_fm; /* Placed in the queue by     */
   char        buffer[BUFSIZ];
} MESSAGE;

#endif
```

6. Compile the server and the client files to create the programs server queue and client queue respectively.

7. Run the client process and type strings at the prompt. Check the returned string from the server.

8. If you stop the client using Ctrl-C, re-start the client and type strings, nothing happens. Explain why.

TODO:

It will be given during the lab based on material covered in this section.