SYSTEM PROGRAMMING

Module 7

EGCI 252
SYSTEM PROGRAMMING

Computer Engineering Department
Faculty of Engineering
Mahidol University

Message Queues

- A Message Queue is a linked list of message structures stored inside the kernel's memory space and accessible by multiple processes
- Synchronization is provided automatically by the kernel
- New messages are added at the end of the queue
- Each message structure has a long message type
- Messages may be obtained from the queue either in a FIFO manner (default) or by requesting a specific type of message (based on message type)

Message Structs

- Each message structure must start with a long message type:

```
struct msg
{
        long msg_type;
        char cdata[512];  /* rest of message */
        int idata;
        float fdata;
};
```

Message Queue Limits

- Each message queue is limited in terms of both the maximum number of messages it can contain and the maximum number of bytes it may contain
- New messages cannot be added if either limit is hit (new writes will normally block)
- On linux, these limits are defined as (in /usr/include/linux/msg.h):
  - MSGMAX 8192 /* max bytes in a message */
  - MSBMNB 16384 /* max bytes in a queue */

Obtaining a Message Queue

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

- The **key** parameter is either a non-zero identifier for the queue to be created or the value IPC_PRIVATE, which guarantees that a new queue is created.
- The **msgflg** parameter is the read-write permissions for the queue OR'd with one of two flags:
  - IPC_CREAT will create a new queue or return an existing one
  - IPC_EXCL added will force the creation of a new queue, or return an error

Writing to a Message Queue

int **msgsnd**(int **msqid**, const void * **msg_ptr**,
size_t **msg_size**, int **msgflags**);

- **msgqid** is the id returned from the msgget call
- **msg_ptr** is a pointer to the message structure
- **msg_size** is the size of that structure
- **msgflags** defines what happens when the queue is full, and can be set to the following:
  - IPC_NOWAIT (non-blocking, return −1 immediately if queue is full)

Reading from a Message Queue

int **msgrcv**(int **msqid**, const void * **msg_ptr**,
size_t **msg_size**, long **msgtype**, int **msgflags**);

- **msgqid** is the id returned from the msgget call
- **msg_ptr** is a pointer to the message structure
- **msg_size** is the size of that structure
- **msgtype** is set to:
  - = 0 first message available in FIFO stack
  - > 0 first message on queue whose type equals type
  - < 0 first message on queue whose type is the lowest value less than or equal to the absolute value of msgtype
- **msgflags** defines what happens when the queue is empty, and can be set to the following:
  - IPC_NOWAIT (non-blocking, return −1 immediately if queue is empty)

Message Queue Control

```
struct msqid_ds {
        struct ipc_perm msg_perm;     /* Ownership and permissions */
        time_t msg_stime;             /* Time of last msgsnd() */
        time_t msg_rtime;             /* Time of last msgrcv() */
        time_t msg_ctime;             /* Time of last change */
        unsigned long __msg_cbytes;   /* Number of bytes in queue */
        msgqnum_t msg_qnum;           /* Number of messages in queue */
        msglen_t msg_qbytes;          /* Maximum bytes in queue */
        pid_t msg_lspid;              /* PID of last msgsnd() */
        pid_t msg_lrpid;              /* PID of last msgrcv() */
};
```

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

- **cmd** can be one of:
  - IPC_RMID destroy the queue specified by msqid
  - IPC_SET set the uid, gid, mode, and qbytes for the queue
  - IPC_STAT get the current msqid_ds struct for the queue

Message Queue - Example

```
//Sender.c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct a_msg
{
  long int msg_type;
  char data[BUFSIZ];
};
```

```
int main(void)
{
  int running = 1, msgID
  struct a_msg a_msg;
  char buffer[BUFSIZ];

  msgID = msgget((key_t) 1234,
                 0666 |IPC_CREAT);
  if (msgID == -1)
  {
    fprintf(stderr, "msgget failed\n");
    exit(EXIT_FAILURE);
  }
```

```
while (running)
{
    printf("Enter data: ");
    fgets(buffer, BUFSIZ, stdin);
    a_msg.msg_type = 1;
    strcpy(a_msg.data, buffer);
    if (msgsnd(msgID, (void *) &a_msg,
                  BUFSIZ,  0) == -1)
    {
      fprintf(stderr, "msgsnd failed\n");
      exit(EXIT_FAILURE);
    }
    if (strncmp(buffer, "end", 3) == 0)
      running = 0;
  }

  exit(EXIT_SUCCESS);
}
```

Message Queue – Example (Cont.)

```
//Receiver.c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

```
struct a_msg
{
  long int msg_type;
  char data[BUFSIZ];
};
```

```
int main(void)
{
  int running = 1, msgID
  struct a_msg a_msg;
  long int rxcv_msg_type = 0;

  msgID = msgget((key_t) 1234,
                          0666 |
    IPC_CREAT);
 if (msgID == -1)
 {
    fprintf(stderr, "msgget failed\n");
    exit(EXIT_FAILURE);
 }
```

```
while (running)
{
    if (msgrcv(msgID, (void *) &a_msg,
        BUFSIZ, rcv_msg_type, 0) == -1)
    {
      fprintf(stderr, "msgrcv failed\n");
      exit(EXIT_FAILURE);
    }
      printf("Received Message: %s",
              a_msg.data);
      if (strncmp(a_msg.data, "end", 3) == 0)
       running = 0;
  }
 if (msgctl(msgID, IPC_RMID, 0) == -1)
 {   fprintf(stderr, "msgctl failed\n");
   exit(EXIT_FAILURE);   }

  exit(EXIT_SUCCESS);
}
```

Assignment

- Write a simple chat program using a message queue
- Requirements:
  - The program's name must be "qchat.c"
  - The program takes one command line argument (i.e., 1 or 2 to indicated the type of messages)
  - Create a message queue with the key value of 11235
  - Must be able to concurrently send and receive any messages between two "qchat" processes.
  - Use the word "end chat" as a command to end the chat program

End of Module