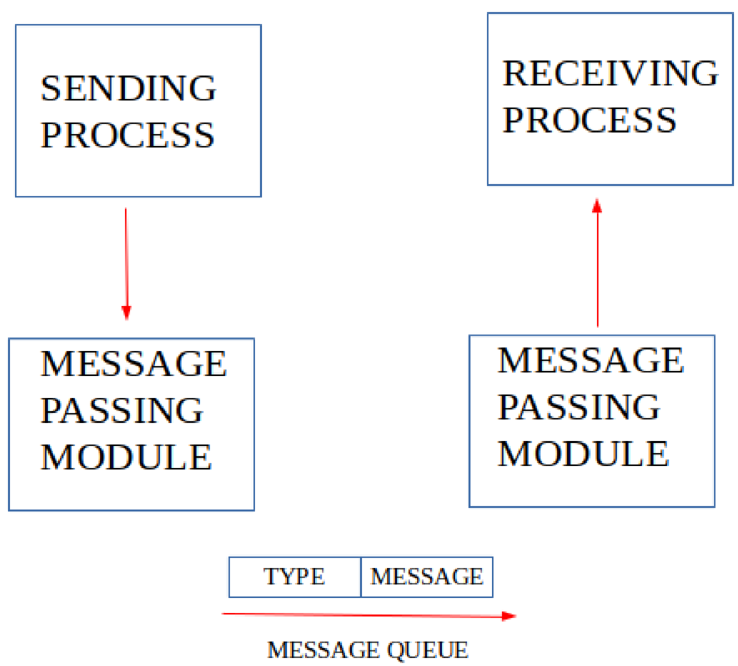
Message Queues

What are message queues in OS? IPC

- A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier
- 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 do not have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field
- All processes can exchange information through access to a common system message queue

Message queues

- The sending process places a message (via some (OS) message-passing module) onto a queue which can be read by another process
- Each message is given an identification or type so that processes can select the appropriate message
- Process must share a common key in order to gain access to the queue in the first place



Sys calls used in message queues

- **ftok()**: is used to generate a unique key
- **msgget()**: either returns the message queue identifier for a newly created message queue or returns the identifiers for a queue which exists with the same key value
- **msgsnd()**: Data is placed on to a message queue by calling msgsnd()
- **msgrcv()**: messages are retrieved from a queue
- **msgctl()**: It performs various operations on a queue. Generally it is use to destroy message queue

Message queue for writer process

```c
// C Program for Message Queue (Writer Process)
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MAX 10

// structure for message queue
struct mesg_buffer {
  long mesg_type;
  char mesg_text[100];
} message;

int main()
{
  key_t key;
  int msgid;

  // ftok to generate unique key
  key = ftok("progfile", 65);

  // msgget creates a message queue
  // and returns identifier
  msgid = msgget(key, 0666 | IPC_CREAT);
  message.mesg_type = 1;

  printf("Write Data : ");
  fgets(message.mesg_text,MAX,stdin);

  // msgsnd to send message
  msgsnd(msgid, &message, sizeof(message), 0);

  // display the message
  printf("Data send is : %s \n", message.mesg_text);

  return 0;
}
```

Message queue for reader process

```c
// C Program for Message Queue (Reader Process)
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// structure for message queue
struct mesg_buffer {
  long mesg_type;
  char mesg_text[100];
} message;

int main()
{
  key_t key;
  int msgid;

  // ftok to generate unique key
  key = ftok("progfile", 65);

  // msgget creates a message queue
  // and returns identifier
  msgid = msgget(key, 0666 | IPC_CREAT);

  // msgrcv to receive message
  msgrcv(msgid, &message, sizeof(message), 1, 0);

  // display the message
  printf("Data Received is : %s \n",
          message.mesg_text);

  // to destroy the message queue
  msgctl(msgid, IPC_RMID, NULL);

  return 0;
}
```

ftok() filetokey

- In C, the ftok() function generates an interprocess communication (IPC) key based on a path and ID
- The key can be used in subsequent calls to msgget(), semget(), and shmget() to obtain IPC identifiers
- The ftok() function returns the same key value for all paths that name the same file, when called with the same id value
- If a different id value is given, or a different file is given, a different key is returned
- The ftok function creates a sort of identifier to be used with the System V IPC functions (semget, shmget, msgget)

The routine “ftok()” is literally “file to key”, and identifies the key space by the volume ID and inode number of the file within that volume ID space

Message Queues

- These are structured ways for processes to send and receive messages, which can be of different types and sizes
- Messages are usually stored in a queue data structure, and processes can operate independently and handle messages at their own pace
- Message queues can behave in a FIFO manner, but they are also flexible enough to retrieve byte chunks out of order
- We can control the geometry of a message queue, setting ceilings on the number of messages and the size of each message
- A process can also determine the status of a message queue, including how many messages are on the queue, the maximums and flags that have been set, and the number of processes blocking to send or receive

Pipes

- These are unidirectional communication channels that transfer a stream of bytes from one process to another
- Pipes are typically used for simple data exchange between a producer process (writing end) and a consumer process (reading end)
- They rely on a sequential, byte-oriented data flow, and have strict FIFO behavior, meaning the first byte written is the first byte read, and so on
- Pipes are a good choice if you need to be able to write messages of varying lengths
- One major drawback of pipes is that their state is unknown

Difference between message queues and pipes

- Message queues are not limited by process hierarchy. They can be used between related processes (parent-child) or unrelated processes
- Pipes are commonly used between a parent process and its child, with the parent process typically creating the pipe and forking the child

Advantage of message queues over pipes

- Message queues are more versatile and suitable for structured communication between processes
- Pipes are simpler and are often used for direct data streaming between related processes