Processes (Part II)

Amir H. Payberah
amir@sics.se

Amirkabir University of Technology
(Tehran Polytechnic)

Inter-Process Communication (IPC)

Cooperating Processes

- Processes within a system may be independent or cooperating.

Cooperating Processes

- Processes within a system may be independent or cooperating.
  - Independent process cannot affect or be affected by the execution of another process.

Cooperating Processes

- Processes within a system may be independent or cooperating.
  - Independent process cannot affect or be affected by the execution of another process.
  - Cooperating process can affect or be affected by other processes.

Cooperating Processes

- Processes within a system may be independent or cooperating.
  - Independent process cannot affect or be affected by the execution of another process.
  - Cooperating process can affect or be affected by other processes.

- Reasons for cooperating processes: information sharing, computation speedup, ...

Cooperating Processes

- Processes within a system may be independent or cooperating.
  - Independent process cannot affect or be affected by the execution of another process.
  - Cooperating process can affect or be affected by other processes.

- Reasons for cooperating processes: information sharing, computation speedup, ...

- Producer-Consumer model: producer process produces information that is consumed by a consumer process.

Inter-Process Communication

- Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information.

Inter-Process Communication

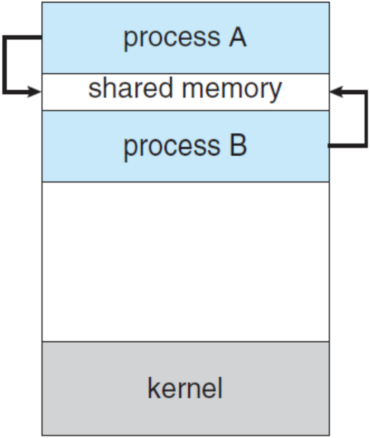
- Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information.

- Two models of IPC
  - Shared memory
  - Message passing

Shared Memory

Shared Memory (1/3)

- An area of memory shared among the processes that wish to communicate.



Shared Memory (2/3)

- It is resides in the address space of the process creating the shared-memory segment.
- Other processes must attach it to their address space for communication.

Shared Memory (3/3)

- But, OS prevents one process from accessing another process's memory.

Shared Memory (3/3)

- But, OS prevents one process from accessing another process's memory.
- Shared memory requires that two or more processes agree to remove this restriction.

Shared Memory (3/3)

- But, OS prevents one process from accessing another process's memory.
- Shared memory requires that two or more processes agree to remove this restriction.
- The communication is under the control of the users processes not the OS.
  - Synchronization

Shared Memory - Producer-Consumer Model (1/3)

- Defining the buffer.
  - Unbounded buffer: no practical limit on the size of the buffer.
  - Bounded buffer: a fixed buffer size.

```
#define BUFFER_SIZE 10
typedef struct {
  ...
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Shared Memory - Producer-Consumer Model (2/3)

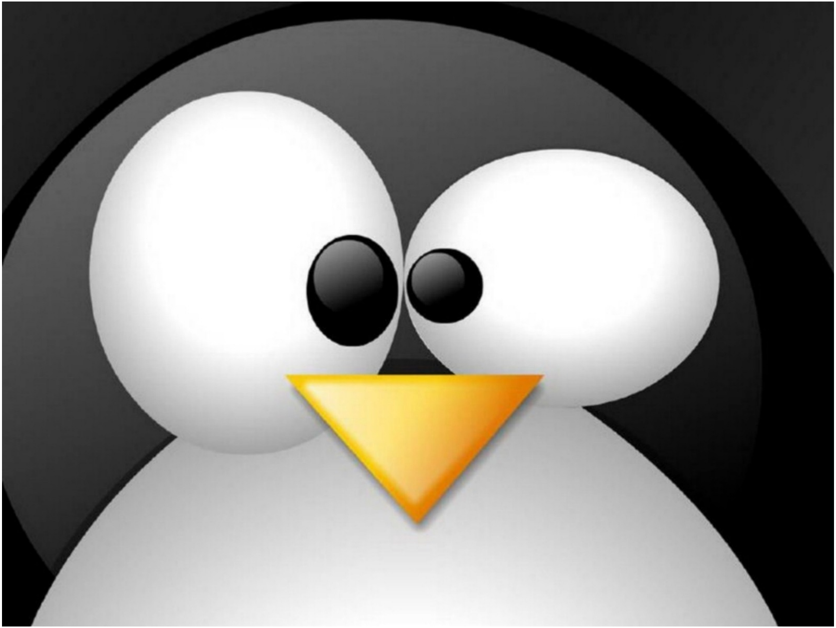
- Producer

```
item next_produced;

while (true) {
  // produce an item in next produced
  while (((in + 1) % BUFFER_SIZE) == out) {
      // do nothing
  }

  buffer[in] = next_produced;
  in = (in + 1) % BUFFER_SIZE;
}
```

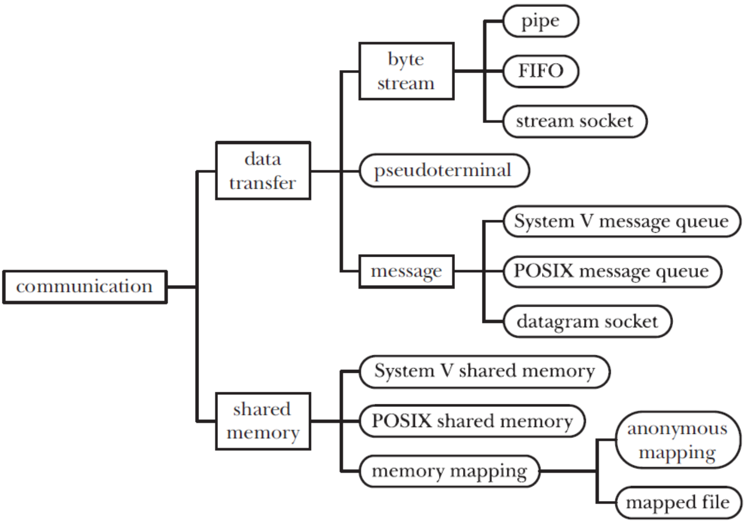
Shared Memory - Producer-Consumer Model (3/3)

- Consumer

```
item next_consumed;

while (true) {
  while (in == out) {
    // do nothing
  }

  next_consumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  // consume the item in next consumed
}
```



A Taxonomy of Linux IPC Facilities



[Michael Kerrisk, The Linux Programming Interface, No Starch Press, 2010]

System V vs. POSIX IPC

- System V (System Five): one of the first commercial versions of the Unix OS.
- POSIX (Portable Operating System Interface): a family of standards for maintaining compatibility between OSs.

System V vs. POSIX IPC

- System V (System Five): one of the first commercial versions of the Unix OS.
- POSIX (Portable Operating System Interface): a family of standards for maintaining compatibility between OSs.
- Both have the same basic IPC tools, but they offer a slightly different interfaces to those tools.

System V vs. POSIX IPC

- System V (System Five): one of the first commercial versions of the Unix OS.
- POSIX (Portable Operating System Interface): a family of standards for maintaining compatibility between OSs.
- Both have the same basic IPC tools, but they offer a slightly different interfaces to those tools.
- System V is fully supported on all Linux kernels.

POSIX Shared Memory

- To use a POSIX shared memory object, we perform two steps:
  1. Use the `shm_open()` function to open an object with a specified name.
  2. Pass the file descriptor obtained in the previous step in a call to `mmap()` that maps the shared memory object into the process's virtual address space.

Creating Shared Memory Objects

- shm_open() creates and opens a new shared memory object or opens an existing object.
- mmap() creates a new mapping in the virtual address space of the calling process.

```
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>

int shm_open(const char *name, int oflag, mode_t mode);

void *mmap(void *addr, size_t length, int prot, int flags, int fd,
   off_t offset);
```

Removing Shared Memory Objects

- shm_unlink() removes a shared memory object.

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

int shm_unlink(const char *name);
```

Producer-Consumer via Shared Memory (1/2)

- Producer

```
int SIZE = 4096;
char *my_shm = "/tmp/myshm";
char *write_msg = "hello";
char *addr;
int fd;

// create the shared memory object
fd = shm_open(my_shm, O_CREATE | O_RDWR, 0666);

// configuare the size of the shared memory object
ftruncate(fd, SIZE);

// memory map to the shared memory object
addr = mmap(NULL, SIZE, PROT_WRITE, MAP_SHARED, fd, 0);

// write to the shared object
sprintf(addr, "%s", write_msg);
```

Producer-Consumer via Shared Memory (2/2)

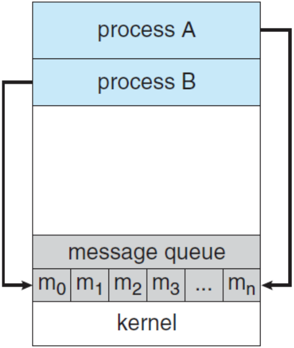
- Consumer

```
int SIZE = 4096;
char *my_shm = "/tmp/myshm";
char *addr;
int fd;

// open the shared memory object
fd = shm_open(my_shm, O_RDONLY, 0666);

// memory map to the shared memory object
addr = mmap(NULL, SIZE, PROT_READ, MAP_SHARED, fd, 0);

// read from to the shared object
printf("%s", (char *)addr);

// remove the shared memory object
shm_unlink("my_shm");
```

Message Passing

Message Passing (1/2)

- Processes communicate with each other without resorting to shared variables.
- Useful in a distributed environment: processes on different computers.



Message Passing (2/2)

- Two operations: `send(message)` and `receive(message)`.

- If processes `p` and `q` wish to communicate, they need to:
  - Establish a communication link between them.
  - Exchange messages via `send` and `receive`.

Message Passing (2/2)

- Two operations: `send(message)` and `receive(message)`.

- If processes `p` and `q` wish to communicate, they need to:
  - Establish a communication link between them.
  - Exchange messages via `send` and `receive`.

- Implementation of communication link:
  - Physical links, e.g., shared memory, hardware bus, network
  - Logical links

Logical Links

- Naming: direct or indirect communication
- Synchronization: synchronous or asynchronous communication
- Buffering: automatic or explicit buffering

Naming (1/3)

- With direct communication, processes must name each other explicitly:
  - `send(p, message)`: sends a message to process `p`.
  - `receive(q, message)`: receives a message from process `q`.

Naming (1/3)

- With direct communication, processes must name each other explicitly:
  - `send(p, message)`: sends a message to process `p`.
  - `receive(q, message)`: receives a message from process `q`.

- Properties of communication link:
  - A link is associated with exactly two processes.
  - Between each pair of processes, there exists exactly one link.

Naming (2/3)

- With indirect communication the messages are sent to and received from mailboxes or ports.
  - `send(A, message)`: sends a message to mailbox `A`.
  - `receive(A, message)`: receives a message from mailbox `A`.

Naming (2/3)

- With indirect communication the messages are sent to and received from mailboxes or ports.
  - `send(A, message)`: sends a message to mailbox `A`.
  - `receive(A, message)`: receives a message from mailbox `A`.

- Properties of communication link:
  - A link is established only if processes share a common mailbox.
  - A link may be associated with more than two processes.
  - Each pair of processes may share several communication links.

Naming (3/3)

- Mailbox sharing
  - p_1, p_2, and p_3 share mailbox A.
  - p_1 sends; p_2 and p_3 receive.
  - Who gets the message?

Naming (3/3)

- Mailbox sharing
  - p_1, p_2, and p_3 share mailbox A.
  - p_1 sends; p_2 and p_3 receive.
  - Who gets the message?

- Solutions
  - Allow a link to be associated with at most two processes.
  - Allow only one process at a time to execute a receive operation.
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization (1/2)

- Message passing may be either blocking or non-blocking.

Synchronization (1/2)

- Message passing may be either blocking or non-blocking.

- Blocking is considered synchronous.
  - Blocking send: the sender is blocked until the message is received.
  - Blocking receive: the receiver is blocked until a message is available.

Synchronization (1/2)

- Message passing may be either blocking or non-blocking.
- Blocking is considered synchronous.
  - Blocking send: the sender is blocked until the message is received.
  - Blocking receive: the receiver is blocked until a message is available.
- Non-blocking is considered asynchronous.
  - Non-blocking send: the sender sends the message and continue.
  - Non-blocking receive: the receiver receives a valid message, or null message.

Synchronization (2/2)

- Producer-consumer model

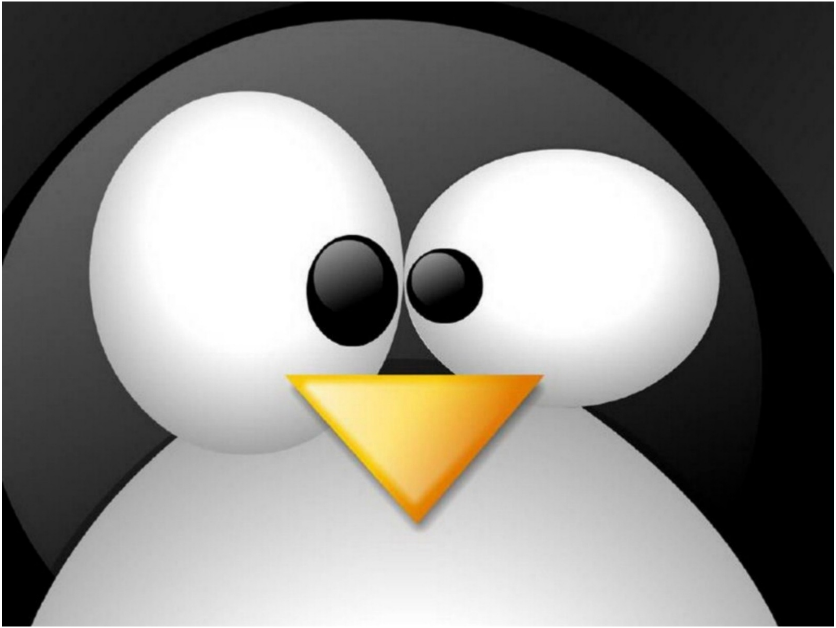
```
// producer
// message: next_produced;
while (true) {
  // produce an item in next produced
  send(next_produced);
}

// consumer
// message: next_consumed;
while (true) {
  receive(next_consumed);
  // consume the item in next consumed
}
```

Buffering

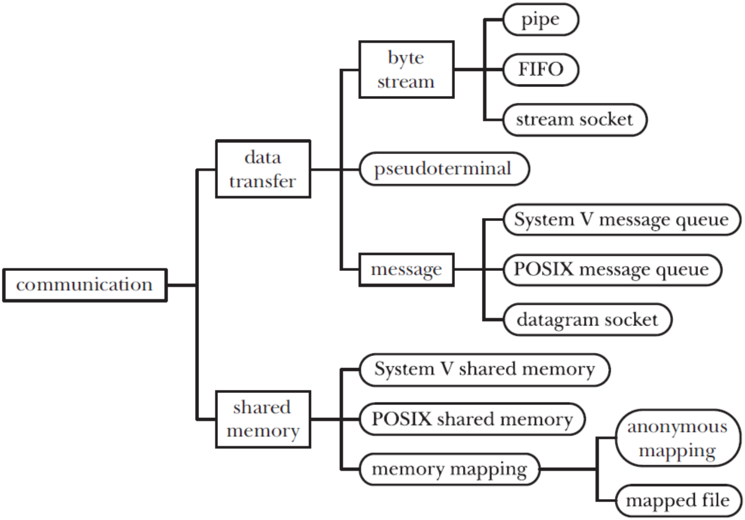
- Queue of the messages attached to the link.

Buffering

- Queue of the messages attached to the link.
- Implemented in one of three ways:
  - Zero capacity: no messages are queued on a link. Sender must wait for receiver.
  - Bounded capacity: finite length of n messages Sender must wait if link full.
  - Unbounded capacity: infinite length. Sender never waits.



A Taxonomy of Linux IPC Facilities



[Michael Kerrisk, The Linux Programming Interface, No Starch Press, 2010]

Data Message vs. Data Stream (1/2)

- Message oriented protocols send data in distinct chunks or groups.
  - The receiver of data can determine where one message ends and another begins.

Data Message vs. Data Stream (1/2)

- Message oriented protocols send data in distinct chunks or groups.
  - The receiver of data can determine where one message ends and another begins.

- Stream protocols send a continuous flow of data.

Data Message vs. Data Stream (2/2)

- Example with mobile phones: text messages would be a message oriented protocol, and a phone call is stream oriented.

Data Message vs. Data Stream (2/2)

- Example with mobile phones: text messages would be a message oriented protocol, and a phone call is stream oriented.

- UDP is a message oriented protocol, and TCP is a stream oriented protocol.

Message Passing IPC Facilities

- Data stream:
  - Pipe
  - FIFO (named pipe)
  - Stream socket
- Data message:
  - Message queue
  - Datagram socket

Pipe

Pipe

- Pipes allow two processes to communicate in standard producer-consumer fashion.
  - The producer writes to the write-end, and the consumer reads from the read-end.

Pipe

- Pipes allow two processes to communicate in standard producer-consumer fashion.
  - The producer writes to the write-end, and the consumer reads from the read-end.
- Pipes are unidirectional, allowing only one-way communication.

Pipe

- Pipes allow two processes to communicate in standard producer-consumer fashion.
  - The producer writes to the write-end, and the consumer reads from the read-end.

- Pipes are unidirectional, allowing only one-way communication.

- Require parent-child relationship between communicating processes.

Pipe

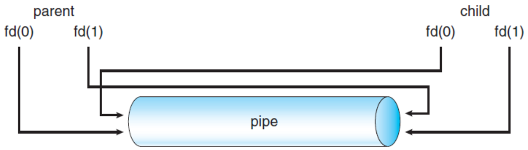
- Pipes allow two processes to communicate in standard producer-consumer fashion.
  - The producer writes to the write-end, and the consumer reads from the read-end.
- Pipes are unidirectional, allowing only one-way communication.
- Require parent-child relationship between communicating processes.

```
ls | wc -l
```

Creating a Pipe

- The pipe() system call creates a new pipe.
- It returns two open file descriptors in the array fd: fd[0] to read from the pipe, and fd[1] to write to the pipe.

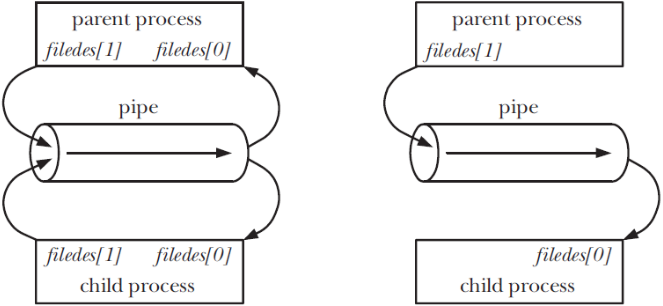
```
#include <unistd.h>

int pipe(int fd[2]);
```



Parent-Child Communication Through a Pipe (1/2)

```
int BUFFER_SIZE = 25;
char write_msg[BUFFER_SIZE] = "hello";
char read_msg[BUFFER_SIZE];
int fd[2];

pipe(fd); // Create the pipe

switch (fork()) {
  case -1: // fork error
    break;
  case 0: // Child
    close(fd[1]); // Close unused write end
    read(fd[0], read_msg, BUFFER_SIZE);
    printf("read %s", read_msg);
    break;
  default: // Parent
    close(fd[0]) // Close unused read end
    write(fd[1], write_msg, strlen(write_msg) + 1);
    break;
}
```

Parent-Child Communication Through a Pipe (2/2)



[Michael Kerrisk, The Linux Programming Interface, No Starch Press, 2010]

FIFO

FIFO (Named Pipe)

- FIFO is similar to a pipe, but it has a name within the file system and is opened in the same way as a regular file.

FIFO (Named Pipe)

- FIFO is similar to a pipe, but it has a name within the file system and is opened in the same way as a regular file.
- Communication is bidirectional.

FIFO (Named Pipe)

- FIFO is similar to a pipe, but it has a name within the file system and is opened in the same way as a regular file.
- Communication is bidirectional.
- No parent-child relationship is necessary.

FIFO (Named Pipe)

- FIFO is similar to a pipe, but it has a name within the file system and is opened in the same way as a regular file.
- Communication is bidirectional.
- No parent-child relationship is necessary.
- Several processes can use a FIFO for communication.

Creating a FIFO

- The `mkfifo()` function creates a new FIFO.

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

int mkfifo(const char *pathname, mode_t mode);
```

Producer-Consumer via FIFO (1/2)

- Producer

```
char *my_fifo = "/tmp/myfifo";
char *write_msg = "hello";
int fd;

// Create the FIFO (named pipe)
mkfifo(my_fifo, 0666);

// Write "hello" to the FIFO
fd = open(my_fifo, O_WRONLY);
write(fd, write_msg, strlen(write_msg));
close(fd);

// Remove the FIFO
unlink(my_fifo);
```

Producer-Consumer via FIFO (2/2)

- Consumer

```
int MAX_SIZE = 100;
char *my_fifo = "/tmp/myfifo";
char buf[MAX_SIZE];
int fd;

// Open the FIFO
fd = open(my_fifo, O_RDONLY);

// Read the message from the FIFO
read(fd, buf, MAX_SIZE);
printf("Received: %s\n", buf);

// Close the FIFO
close(fd);
```

Message Queue

Message Queue

- Message queues allows processes to exchange data in the form of messages.

Message Queue

- Message queues allows processes to exchange data in the form of messages.
- In message queue the consumer receives whole messages, as written by the producer.
  - It is not possible to read part of a message and leave the remainder in the queue, or to read multiple messages at a time.
  - In pipes, the consumer can read an arbitrary number of bytes at a time, irrespective of the size of data blocks written by the producer.

Creating a Message Queue

- `mq_open()` creates a new message queue or opens an existing queue.

```
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag, ...
  /* mode_t mode, struct mq_attr *attr */);
```

Closing a Message Queue

- mq_close() closes the message queue descriptor mqdes.

```
#include <mqueue.h>

int mq_close(mqd_t mqdes);
```

- mq_unlink() removes the message queue identified by name.

```
#include <mqueue.h>

int mq_unlink(const char *name);
```

Message Queue Attributes

- Specifies attributes of a message queue.

```
struct mq_attr {
  long mq_flags; // Message queue description flags
  long mq_maxmsg; // Maximum number of messages on queue
  long mq_msgsize; // Maximum message size (in bytes)
  long mq_curmsgs; // Number of messages currently in queue
};
```

Retrieving and Modifying a Message Queue Attributes

- mq_getattr() returns a mq_attr structure of the message queue.

```
#include <mqueue.h>

int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
```

- mq_setattr() sets attributes of the message queue.

```
#include <mqueue.h>

int mq_setattr(mqd_t mqdes, const struct mq_attr *newattr,
  struct mq_attr *oldattr);
```

Sending and Receiving Messages

- mq_send() adds the message msg_ptr to the message queue.

```
#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
  unsigned int msg_prio);
```

- mq_receive() removes the oldest message from the message queue.

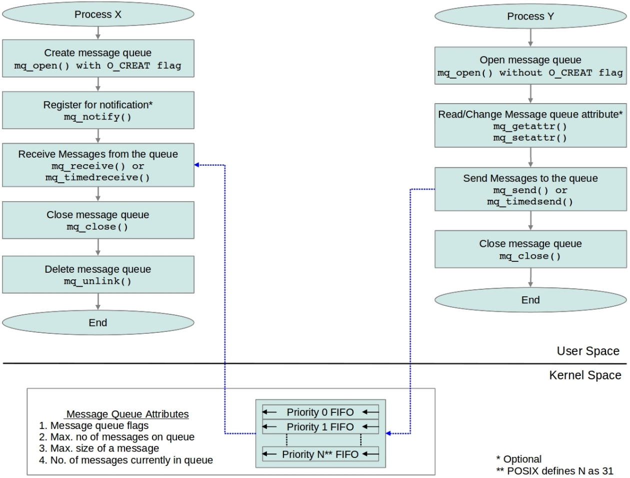
```
#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
  unsigned int *msg_prio);
```

Message Notification

- A way for asynchronous communications.
- A process can request a notification of message arrival and then performs other tasks until it is notified.
- The `mq_notify()` registers the calling process to receive a notification when a message arrives on the empty queue.

```
#include <mqueue.h>

int mq_notify(mqd_t mqdes, const struct sigevent *notification);
```

Linux Message Queue Pattern



[http://www.linuxpedia.org/index.php?title=Linux_POSIX_Message_Queue]

Producer-Consumer via Message Queue (1/2)

- Producer

```
char *my_mq = "/mymq";
char *write_msg = "hello";
mqd_t mqd;

// Open an existing message queue
mqd = mq_open(my_mq, O_WRONLY);

// Write "hello" to the message queue
mq_send(mqd, write_msg, strlen(write_msg), 0);

// Close the message queue
mq_close(mqd);
```

Producer-Consumer via Message Queue (2/2)

- Consumer

```
#define MQ_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
int MAX_SIZE = 100;
int MAX_NUM_MSG = 10;
char *my_mq = "/mymq";
char buf[MAX_SIZE];
mqd_t mqd;
struct mq_attr attr;

// Form the queue attributes
attr.mq_maxmsg = MAX_NUM_MSG;
attr.mq_msgsize = MAX_SIZE;

// Create message queue
mqd = mq_open(my_mq, O_RDONLY | O_CREAT, MQ_MODE, &attr);

// Read the message from the message queue
mq_receive(mqd, buf, MAX_NUM_MSG, NULL);
printf("Message: %s\n", buf);

// Close the message queue
mq_close(mqd);
```

Summary

Summary

- Inter-Process Communication: shared memory vs. message passing

Summary

- Inter-Process Communication: shared memory vs. message passing

Summary

- Inter-Process Communication: shared memory vs. message passing
- Message passing: data messages vs. stream

Summary

- Inter-Process Communication: shared memory vs. message passing
- Message passing: data messages vs. stream
- Data stream: pipe, FIFO

Summary

- Inter-Process Communication: shared memory vs. message passing
- Message passing: data messages vs. stream
- Data stream: pipe, FIFO
- Data message: message queue

Questions?