
Interprocess Communication

Operating Systems

School of Data & Computer Science
Sun Yat-sen University

Lecture Notes: os_sysu@163.com
Instructor: Guoyang Cai
email: isscgymail@mail.sysu.edu.cn





■ Contents

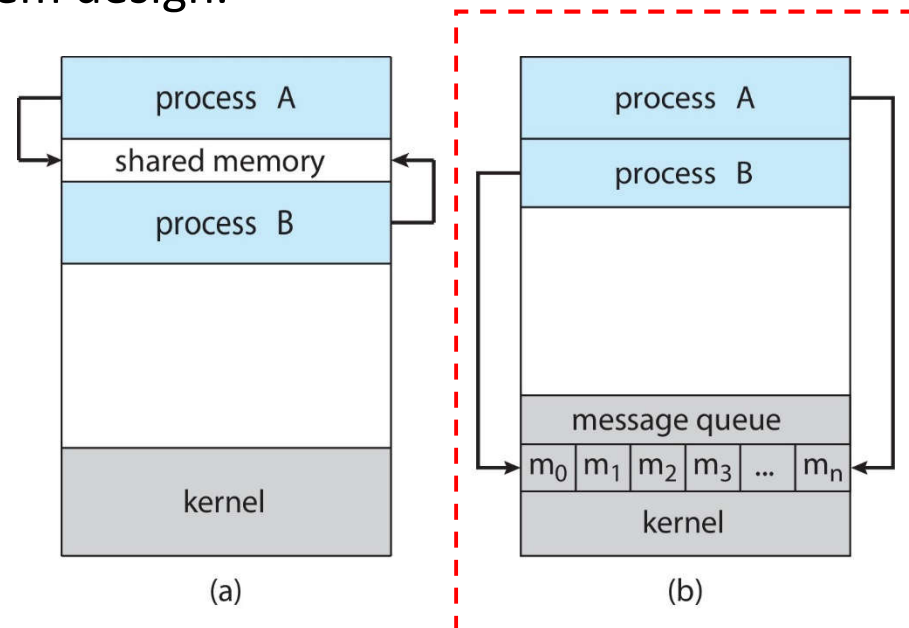
- Overview
- Shared-memory Systems
- Message-passing Systems
- Pipes
- Communications in Client-Server Systems
 - Sockets
 - Remote Procedure Calls

■ Message-passing Systems

- Message-passing is a mechanism provided for the cooperating processes to communicate with each other and to synchronize their actions without resorting to shared variables.
 - Communication takes place by means of messages exchanged between cooperating processes.
 - useful for exchanging smaller amounts of data
 - typically implemented using system calls and thus require the more time-consuming task of kernel intervention
 - easier to implement in a distributed system than shared memory
 - particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.

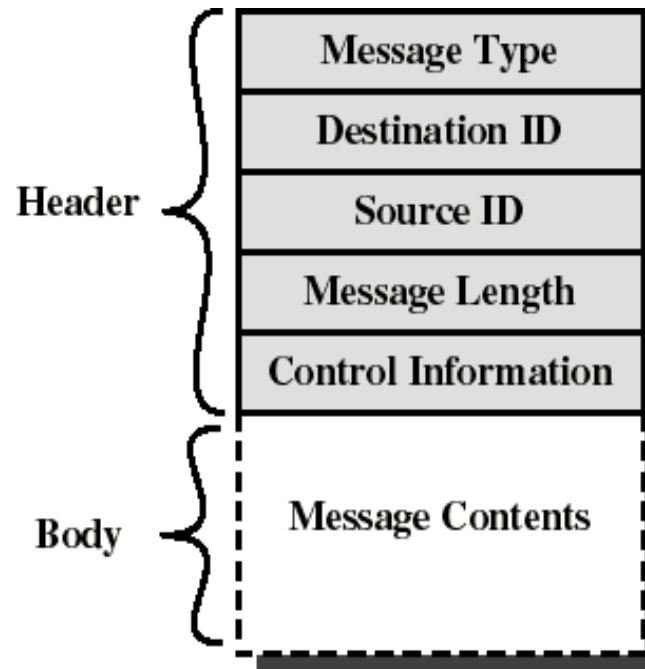
■ Message-passing Systems

- A message-passing facility provides at least two *primitives* (原语):
`send(destination, message)` or `send(message)`
`receive(source, message)` or `receive(message)`
- Message size is fixed or variable.
 - Variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler.
 - This is a common kind of tradeoff seen throughout operating system design.



■ Message Format

- Header and Body
- Control Information
 - what to do if run out of buffer space
 - sequence numbers
 - priority
- Queuing discipline: usually FIFO but can also include priorities.



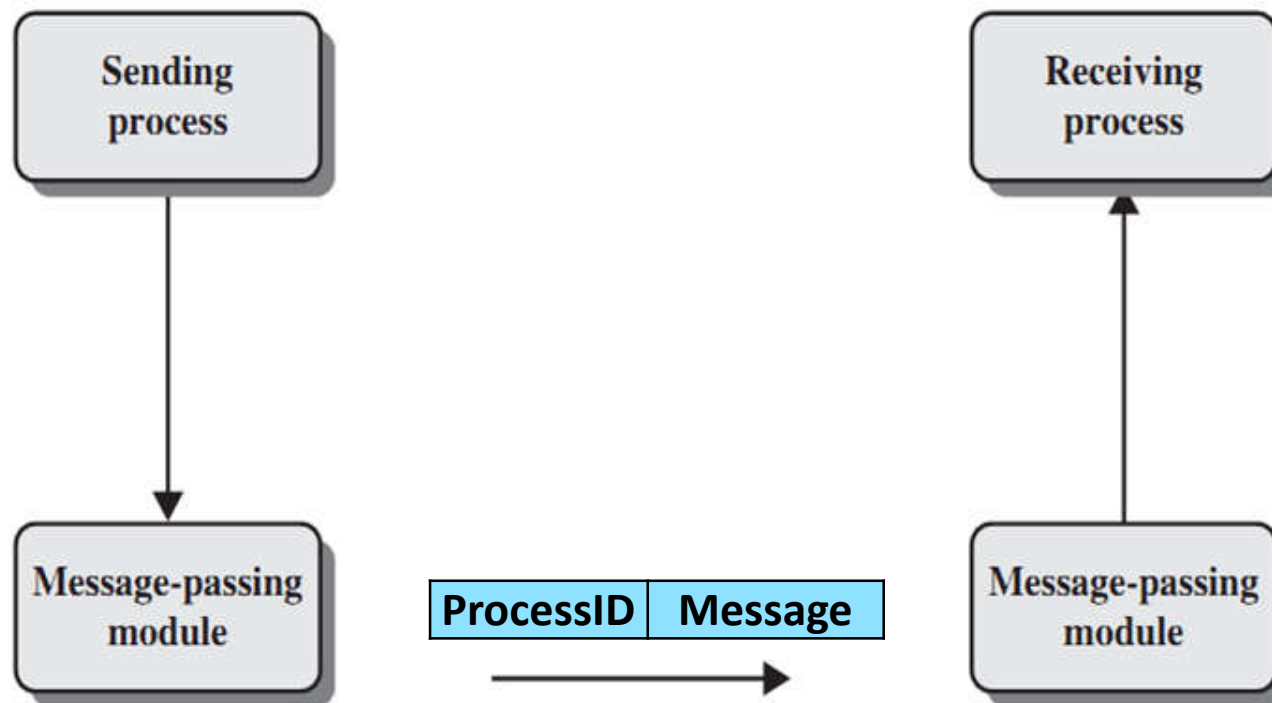


■ Message Format

■ Linux Message Structure

```
struct msg_st {  
    long int msg_type;  
    char mtext[TEXT_SIZE];  
};
```

■ Basic Message-passing Primitives



■ Logical Communication Link

- A *communication link* must exist between two processes if they want to communicate, send messages to and receive messages from each other.
- A communication link can be implemented in a variety of ways.
 - We are concerned here not with the link's physical implementation but rather with its logical implementation.
- Several methods for logically implementing a link and the send/receive operations:
 - *Direct* or *indirect* communication
 - *Synchronous* or *asynchronous* communication
 - *Automatic* or *explicit* buffering.

■ Direct Communication

- Processes of both sides explicitly *name* the recipient or sender of the communication symmetrically:

```
send(destination_process_name, message)
receive(source_process_name, message)
```

or asymmetrically: The receiver is not required to name the sender. The variable *id* is set to the name of the process message received from.

```
send(destination_process_name, message)
receive(id, message)
```

■ Properties

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.

■ Direct Communication

- Disadvantage of direct communication
 - Direct communication uses a specific process identifier for source/destination. But it might be **impossible to specify the source ahead of time**.
 - Another disadvantage in both symmetric and asymmetric schemes is the **limited modularity** of the resulting process definitions.
 - Changing the identifier of a process may necessitate examining all other process definitions.
 - All references to the old identifier must be found, so that they can be modified to the new identifier.
 - In general, any such **hard-coding** (硬编码) techniques, where identifiers must be explicitly stated, are less desirable than techniques involving indirection, as described next.

■ Indirect Communication

- The messages are sent to and received from *mailboxes*, or *ports*.
- A mailbox can be viewed abstractly as an object that
 - Processes can place messages into, and
 - Processes can receive and remove messages from.
- Each mailbox has a unique identification.
 - E.g., POSIX message queues use an integer value to identify a mailbox. A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox ID.

`send(mailbox_A, message)`
`receive(mailbox_A, message)`

- Properties
 - A link is established between a pair of processes only if both members of the pair have a shared mailbox.
 - One link may be associated with more than two processes.
 - Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox.

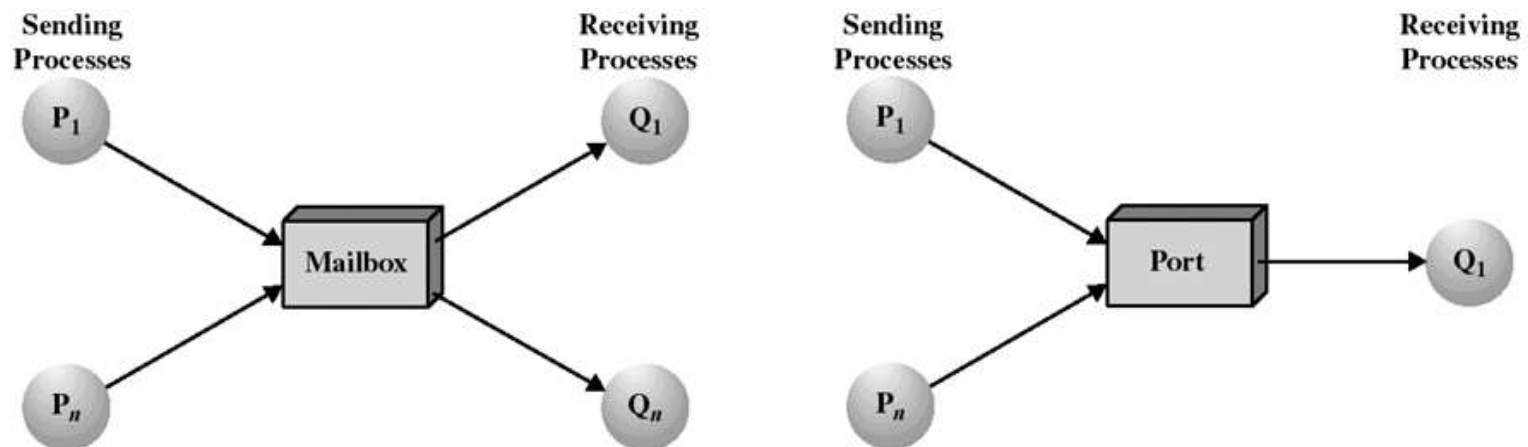
■ Indirect Communication

- Suppose that processes P_1 , P_2 , and P_3 all share mailbox A . Process P_1 sends a message to A , while both P_2 and P_3 execute a `receive()` (and *delete* the received message) from A . Which process will receive the message sent by P_1 ? The answer depends on which of the following methods we choose:
 - Allow a link to be associated with two processes at most.
 - Allow at most one process at a time to execute a `receive()` operation.
 - Allow the system to select arbitrarily which process will receive the message (that is, either P_2 or P_3 , but not both, will receive the message). The system may define an algorithm for selecting which process will receive the message.
 - E.g, *Round-Robin*, where processes take turns receiving messages.
 - The system may identify the receiver to the sender.
- A mailbox may be owned either by a process or by the operating system. The ownership and receiving privilege of a mailbox must be considered.

■ Indirect Communication

■ Mailboxes vs. Ports

- A mailbox can be **private** to one sender/receiver pair.
- The same mailbox can be **shared** among several senders and receivers:
 - The OS may then allow the use of message types (for selection).
- A Port is a mailbox associated with **one receiver and multiple senders**.
 - used for client/server applications - the receiver is the server



■ Indirect Communication

■ Mailboxes vs. Ports

■ Ownership of mailboxes

- OS creates a mailbox on behalf of a process (which becomes the owner).
- The mailbox is destroyed at the owner's request or when the owner terminates.

■ Ownership of ports

- A port is usually created and own by the **receiving process**.
- The port is destroyed when the receiver terminates.

■ Synchronization in Message-passing Systems

- There are different design options for implementing the two primitives `send()` and `receive()`.
- Message passing may be either *blocking* or *non-blocking*. Blocking is considered synchronous, and non-blocking is considered asynchronous.
 - *Blocking send*
 - The sending process blocks until the message is received by the receiving process or by the mailbox.
 - *Blocking receive*
 - The receiver blocks until a message is available.
 - *Non-blocking send*
 - The sending process sends the message and resumes.
 - *Non-blocking receive*
 - The receiver retrieves a valid message or a null.

■ Synchronization in Message-passing Systems

- For the sender
 - It is more natural **not to be blocked** after issuing send:
 - The sender can send several messages to multiple destinations.
 - Senders usually expect acknowledgment of message receipt.
 - in case receiver fails
- For the receiver
 - It is more natural **to be blocked** after issuing receive request.
 - The receiver usually needs the information before proceeding, but it could be blocked indefinitely (无限期堵塞) if sender process fails before send.

■ Synchronization in Message-passing Systems

- There are really three combinations here that make sense:
 - Blocking send, Blocking receive
 - have a rendezvous (汇聚点) between the sender and the receiver
 - both are blocked until the message is received
 - occurs when the communication link is unbuffered (no message queue)
 - provides tight synchronization (*rendezvous*).
 - Non-blocking send, Non-blocking receive
 - used in purpose
 - Non-blocking send, Blocking receive
 - Server process provides services/resources to other processes. It will need the expected information before proceeding.
 - most popular
- We will discuss the synchronization problem in detail later.

■ Buffering

- Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:
 - Zero capacity
 - a message system with no buffering
 - The link cannot have any messages waiting in it.
 - Sender must wait until the recipient receives the message (rendezvous).
 - Bounded capacity
 - The queue has a finite length n .
 - Sender must wait if link full.
 - Unbounded capacity
 - The queue's length is potentially infinite.
 - Sender never waits.



■ Linux: Message-passing

■ Linux IPCs Limits

- The kernel level limits can be redefined in /etc/sysctl.conf

```
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test$ ipcs -l

----- Messages Limits -----
max queues system wide = 32000
max size of message (bytes) = 8192
default max size of queue (bytes) = 16384

----- Shared Memory Limits -----
max number of segments = 4096
max seg size (kbytes) = 18014398509465599
max total shared memory (kbytes) = 18014398509481980
min seg size (bytes) = 1

----- Semaphore Limits -----
max number of arrays = 32000
max semaphores per array = 32000
max semaphores system wide = 1024000000
max ops per semop call = 500
semaphore max value = 32767

isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test$
```



■ Linux: Message-passing

■ Message structure

```
struct msg_struct {
    long int msg_type;
    char mtext[TEXT_SIZE];
};

struct msqid_ds { /* linux/msg.h */
    struct ipc_perm msg_perm;
    time_t msg_stime; /* Last msgsnd time */
    time_t msg_rtime; /* Last msgrcv time */
    time_t msg_ctime; /* Last change time */
    msgqnum_t msg_qnum; /* Current number of messages in queue */
    msglen_t msg_qbytes; /* Maximum number of bytes allowed in queue */
    pid_t msg_lspid; /* PID of last msgsnd */
    pid_t msg_lrpid; /* PID of last msgrcv */
};
```

■ Linux: Message-passing

■ Functions

```
key_t ftok(char *pathname, char proj_id)
```

```
int msgget(key_t key, int msgflg)
```

```
int msgsnd(int msqid, void *msgp, size_t msgsz, int msgflg)
```

```
ssize_t msgrcv(int msqid, const void *msgp, size_t msgsz, long msgtyp,  
int msgflg)
```

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

■ Linux: Message-passing

- Sending & Receiving process illustrating Linux message-passing API.

- Algorithm 9-0: msgdata.h

```
#define TEXT_SIZE 512
/* considering
----- Messages Limits -----
max queues system wide = 32000
max size of message (bytes) = 8192
default max size of queue (bytes) = 16384
-----

The size of message is set to be 512, the total number of messages is 16384/512 = 32
If we take the max size 8192, the number would be 16384/8192 = 2. It is not reasonable
*/

/* message structure */
struct msg_struct {
    long int msg_type;
    char mtext[TEXT_SIZE]; /* binary data */
};

#define PERM S_IRUSR|S_IWUSR|IPC_CREAT

#define ERR_EXIT(m) \
    do { \
        perror(m); \
        exit(EXIT_FAILURE); \
    } while(0)
```

■ Linux: Message-passing

■ Sending & Receiving process illustrating Linux message-passing API.

■ Algorithm 9-1: msgsnd.c (1)

```
int main(int argc, char *argv[])
{
    struct msg_struct data;

    long int msg_type;
    char buffer[TEXT_SIZE], pathname[80];
    int msqid, ret, count = 0;
    key_t key;
    FILE *fp;
    struct stat fileattr;

    if(argc < 2) {
        printf("Usage: ./a.out pathname\n");
        return EXIT_FAILURE;
    }
    strcpy(pathname, argv[1]);

    if(stat(pathname, &fileattr) == -1) {
        ret = creat(pathname, O_RDWR);
        if (ret == -1) {
            ERR_EXIT("creat()");
        }
        printf("shared file object created\n");
    }
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/msg.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "alg.9-0-msgdata.h"
```

■ Linux: Message-passing

- Sending & Receiving process illustrating Linux message-passing API.

- Algorithm 9-1: msgsnd.c (2)

```
key = ftok(pathname, 0x27); /* project_id can be any nonzero integer */
if(key < 0) {
    ERR_EXIT("ftok()");
}

printf("\nIPC key = 0x%x\n", key);

msqid = msgget((key_t)key, 0666 | IPC_CREAT);
if(msqid == -1) {
    ERR_EXIT("msgget()");
}

fp = fopen("./msgsnd.txt", "rb");
if(!fp) {
    ERR_EXIT("source data file: ./msgsnd.txt fopen()");
}

struct msqid_ds msqattr;
ret = msgctl(msqid, IPC_STAT, &msqattr);
printf("number of messages remained = %ld, empty slots = %ld\n", msqattr.msg_qnum,
16384/TEXT_SIZE-msqattr.msg_qnum);
printf("Blocking Sending ... \n");
```


■ Linux: Message-passing

- Sending & Receiving process illustrating Linux message-passing API.

- Algorithm 9-1: msgsnd.c (3)

```
while (!feof(fp)) {
    ret = fscanf(fp, "%ld %s", &msg_type, buffer);
    if (ret == EOF) break;
    printf("%ld %s\n", msg_type, buffer);

    data.msg_type = msg_type;
    strcpy(data.mtext, buffer);

    ret = msgsnd(msqid, (void *)&data, TEXT_SIZE, 0);
    /* 0: blocking send, waiting when msg queue is full */
    if (ret == -1) {
        ERR_EXIT("msgsnd()");
    }
    count++;
}

printf("number of sent messages = %d\n", count);

fclose(fp);
system("ipcs -q");
exit(EXIT_SUCCESS);
}
```



```
iisscgy@ubuntu:/mnt/os-2020$ gcc alg.9-1-msgsnd.c
iisscgy@ubuntu:/mnt/os-2020$ ./a.out /home/iisscgy/myshm

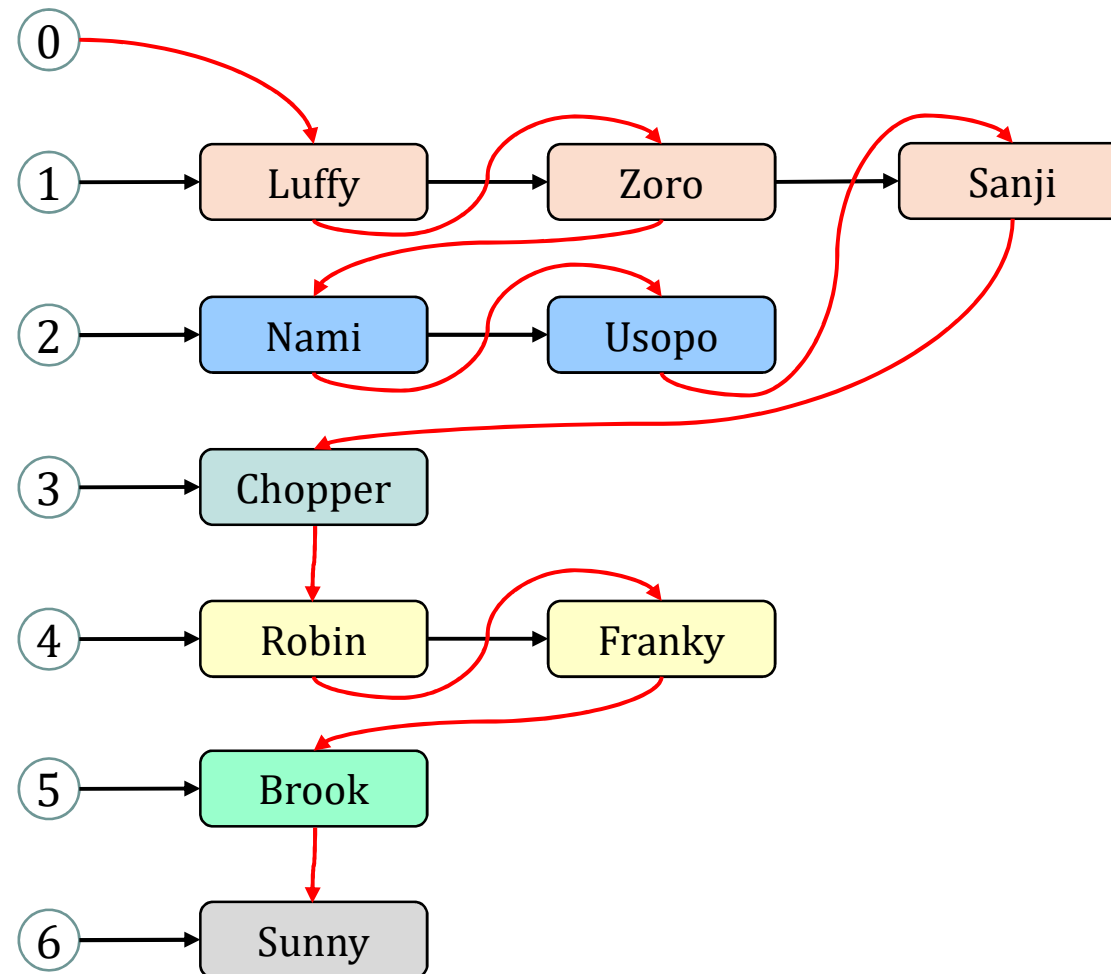
IPC key = 0x27011c6c
number of messages remained = 0, empty slots = 32
Blocking Sending ...
1 Luffy
1 Zoro
2 Nami
2 Usopo
1 Sanji
3 Chopper
4 Robin
4 Franky
5 Brook
6 Sunny
number of sent messages = 10

----- Message Queues -----
key          msqid      owner      perms      used-bytes   messages
0x27011c9e 1          iisscgy    666        0            0
0x27011c6c 4          iisscgy    666        5120         10

iisscgy@ubuntu:/mnt/os-2020$
```

Linux: Message-passing

- Sending & Receiving process illustrating Linux message-passing API.
- The message queues.



■ Linux: Message-passing

- Sending & Receiving process illustrating Linux message-passing API.

- Algorithm 9-2: msgrcv.c (1)

```
/* Usage: ./b.out pathname msg_type */
int main(int argc, char *argv[])
{
    key_t key;
    struct stat fileattr;
    char pathname[80];
    int msqid, ret, count = 0;
    struct msg_struct data;
    long int msgtype = 0;    /* 0 - type of any messages */

    if(argc < 2) {
        printf("Usage: ./b.out pathname msg_type\n");
        return EXIT_FAILURE;
    }
    strcpy(pathname, argv[1]);
    if(stat(pathname, &fileattr) == -1) {
        ERR_EXIT("shared file object stat error");
    }
    if((key = ftok(pathname, 0x27)) < 0) {
        ERR_EXIT("ftok()");
    }
    printf("\nIPC key = 0x%x\n", key);
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/msg.h>
#include <sys/stat.h>

#include "alg.9-0-msgdata.h"
```

■ Linux: Message-passing

- Sending & Receiving process illustrating Linux message-passing API.

- Algorithm 9-2: msgrcv.c (2)

```
msqid = msgget((key_t)key, 0666); /* do not create a new msg queue */
if(msqid == -1) {
    ERR_EXIT("msgget()");
}

if(argc < 3)
    msgtype = 0;
else {
    msgtype = atol(argv[2]);
    if (msgtype < 0)
        msgtype = 0;
} /* determin msgtype (class number) */
printf("Selected message type = %ld\n", msgtype);

while (1) {
    ret = msgrcv(msqid, (void *)&data, TEXT_SIZE, msgtype, IPC_NOWAIT);
    /* Non_blocking receive */
    if(ret == -1) { /* end of this msgtype */
        printf("number of received messages = %d\n", count);
        break;
    }

    printf("%ld %s\n", data.msg_type, data.mtext);
    count++;
}
```

■ Linux: Message-passing

- Sending & Receiving process illustrating Linux message-passing API.

- Algorithm 9-2: msgrcv.c (3)

```
struct msqid_ds msqattr;
ret = msgctl(msqid, IPC_STAT, &msqattr);
printf("number of messages remaining = %ld\n", msqattr.msg_qnum);

if(msqattr.msg_qnum == 0) {
    printf("do you want to delete this msg queue?(y/n)");
    if (getchar() == 'y') {
        if(msgctl(msqid, IPC_RMID, 0) == -1)
            perror("msgctl(IPC_RMID)");
    }
}

system("ipcs -q");
exit(EXIT_SUCCESS);
}
```

■ Linux: Message-passing

- Sending & Receiving process illustrating Linux message-passing API.
 - Algorithm 9-2: msgrcv.c (3)

```
struct msqid_ds msqattr;
```

```
isscgy@ubuntu:/mnt/os-2020$ gcc -o b.out alg.9-2-msgrcv.c
isscgy@ubuntu:/mnt/os-2020$ ./b.out /home/isscgy/mysh
shared file object stat error: No such file or directory
isscgy@ubuntu:/mnt/os-2020$ ./b.out /home/isscgy/myshm 2
```

```
IPC key = 0x27011c6c
```

```
Selected message type = 2
```

```
2 Nami
```

```
2 Usopo
```

```
number of received messages = 2
```

```
number of messages remaining = 8
```

```
----- Message Queues -----
```

key	msqid	owner	perms	used-bytes	messages
0x27011c9e	1	isscgy	666	0	0
0x27011c6c	5	isscgy	666	4096	8

```
isscgy@ubuntu:/mnt/os-2020$
```



Linux: Message-passing

- Sending & Receiving process illustrating Linux message-passing API.

```
isscgy@ubuntu:/mnt/os-2020$ ./b.out /home/isscgy/myshm 0

IPC key = 0x27011c6c
Selected message type = 0
1 Luffy
1 Zoro
1 Sanji
3 Chopper
4 Robin
4 Franky
5 Brook
6 Sunny
number of received messages = 8
number of messages remaining = 0
do you want to delete this msg queue?(y/n)y

----- Message Queues -----
key          msqid      owner      perms      used-bytes   messages
0x27011c9e 1          isscgy     666         0             0
[redacted]
isscgy@ubuntu:/mnt/os-2020$
```


■ POSIX: Message-passing

```
#include <mqueue.h>
```

■ Open, Close and Unlink

```
mqd_t mq_open(const char *name, int oflag, mode_t mode, struct  
mq_attr *attr ); /* return the mqdes, or -1 if failed */
```

```
mqd_t mqID;
```

```
mqID = mq_open("/anonymQueue", O_RDWR | O_CREAT, 0666, NULL);
```

```
mqd_t mq_close(mqd_t mqdes);
```

```
mqd_t mq_unlink(const char *name); /* return -1 if failed */
```

■ Send and Receive

```
mqd_t mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,  
unsigned msg_prio); /* return 0, or -1 if failed */
```

```
mq_send(mqID, msg, sizeof(msg), i)
```

```
mqd_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,  
unsigned *msg_prio); /* return the number of char received, or -1  
if failed */
```

```
mq_attr mqAttr;
```

```
mq_getattr(mqID, &mqAttr);
```

```
mq_receive(mqID, buf, mqAttr.mq_msgsize, NULL)
```

■ Windows XP: Message-passing

■ Local Procedure Calls in Windows XP

