Interprocess Communication

2 fundamentals mode of interprocess communication

- Shared memory
- Message Passing - easier to implement in distributed systems

Why shared memory over message passing

- Shared memory is faster since message passing requires system calls which requires context switching and kernel intervention
- In shared memory, sys calls are required to only establish a shared memory region, all accesses are treated as routine memory accesses
- IPC Methods
  - Pipes - known only to process and its descendents
  - Signals
- IPC provides 2 operations
  - Send(message)
  - Receive(message)

Consumer Producer Problem

Consumer and producer must be synchronized; consumer should not try to consume an item which is not yet been produced by the producer.

2 types of buffers

- Unbounded buffer
  - Producer never waits
  - Consumer waits if there is no buffer to consumer.
- Bounded buffer
  - Producer waits if the buffer is full
  - Consumer waits if there is no buffer to consumer.

Shared memory solution

```
#define BUFFER_SIZE 10
typedef struct{

}item;
```

```
item buffer[BUFFER_SIZE]; // circular array
int in = 0 /* points to next free position */, out = 0 /* points to the first full
position */;
```

PRODUCER

```
item next_produced;
while(true){
        // produce an item in next produced
        while(((in + 1) % BUFFER_SIZE) == 0)
        ; // do nothing
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

CONSUMER

```
while(true){
        while(in == out)
        ; // do nothing
        next_consumed  = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
}
```

This solution allows atmost `BUFFER_SIZE - 1` items in the buffer

ANOTHER SOLUTION: introduce a `count` variable
Problem: if `count = 5`, after execution of `count++` and `count--`, `count` could be 4, 5, or 6

RACE CONDITION

When two or more processes manipulate the **same data concurrently** and the outcome of the execution **depends on the particular order in which the access takes place**.

IPC in UNIX System V

get

- Each mechanism contains a **table** whose each entry contains a **numeric key**, which is its user chosen name.
- Each mechanism contains a `get()` sys call to **create a new entry or to retrieve an existing one**, and the parameters include the key and other flags

    - The kernel searches the table for an entry named by the key.
    - with the key `IPC_PRIVATE` it will return an unused entry
    - `IPC_CREAT` to create a new entry if one by the given key does not exists
    - Error notification if it already exists: `IPC_CREAT` and `IPC_EXCL`
- The kernel uses the following formula to find index into the table of data strcutures from the descriptor:

$$index = descriptor \mod (number\_of\_entries\_in\_table)$$

- When a process removes an entry:
    - The kernel increments the descriptor associated with it by the number of entries in the table.
    - The incremented descriptor becomes the new descriptor of entry of the next `get()` call
    - If $N = 100$ and we want to remove entry 201, the new descriptor becomes 301
    - Process that attempt to retrieve 201 receive an error, because it is no longer valid.
    - Descriptor values are recycled after some time.

control

- `control` sys call to query status of an entry, to set status info, or to remove entry from the system.
- When a process queries the status of an entry
    - The kernel verifies that is has the read permission
    - Then copies the data from the table entry to user address.
- To set paramters on an entry
    - The kernel verifies that userID of the process matches userID or the creator userID of the entry ot that the process is a superuser, same checking if it wanted to remove the entry.
    - Write permissions are not enough to set paramters.
    - The kernel copies the data into table entry

Permission structure

- Contains user ID
- groupID of the process that created the entry
- A user and groupID set by `control`
- A set of read-write permissions for user, group, and others.

Entry

Each entry also contains

- The process ID of the last process to update the entry
- The time of last access or update

System calls for messages

- `msgget` returns or possibly creates a message descriptor that designates a message queue for use in other system calls
- `msgctl` to set and return parameters associated with a message descriptor and an option to remove msg descriptors.
- `msgsnd` sends a message
- `msgrcv` receives a message

msgget

```
msgqid = msgget(key, flag);
```

`msgqid` is descriptor returned by the call
The kernel stores messages on a linked list (queue) per descriptor, and it uses `msgqid` as an index into an array of message queue headers (hash table with chaining).

In addition, the queue structure contains the following fields:

- Pointers to first and last messages on a linked list
- The number of messages and total number of data bytes on a linked list
- Max number of bytes of data that can be on the linked list
- The process IDs of the last processes to call `msgsnd` and `msgrcv`
- Time stamps of the last `msgsnd`, `msgrcv`, `msgctl` operations.

When a user calls to create a new descriptor:

- The kernel searches the array of queues to see if one exists with the given key.
- If one exists, it checks permissions and returns
- If one doesn't exists, the kernel allocates a new queue structure, intializes it, and returns an identifier to the user.

msgsnd

```
msgsnd(msgqid, msg, count, flag);
```

`msgqid` is the descriptor returned by `msgget`
`msg` is a pointer to message

`count` is the size of the data array
`flag` specifies the action the kernel should take if runs out of internal buffer.

The kernel checks that

- The sending process has write permission for the message descriptor
- The message length does not exceed the system limit
- The message queue does not contain too many bytes
- The message type is a positive integer

If all checks success, the kernel

- Allocates space for the message from a message map
- Copies the data from user space
- **Allocates a message header**
- Puts it on the end of the linked list of message header for the message queue.
- Sets the message header to point to the message data, and updates various statistics field in the queue header.
- Awakens the processes waiting for a message to arrive on the queue
- If the number of bytes on queue > queue's limit, the process sleeps until other messages are removed from the queue or returns an error with `IPC_NOWAIT` flag.

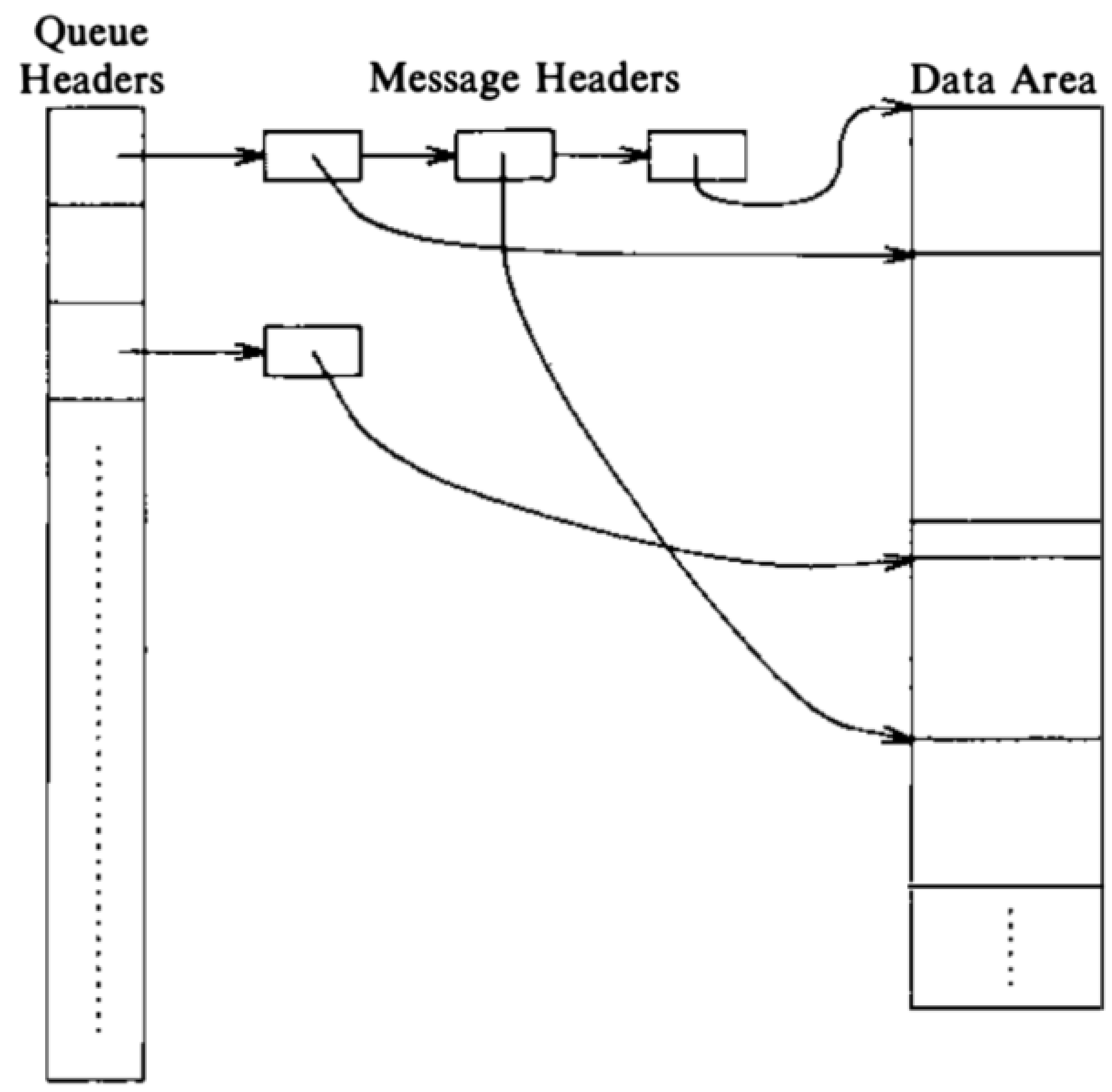

Figure 11.5. Data Structures for Messages

msgrcv

```
count = msgrcv(id, msg, maxcount, type, flag);
```

`id` = message descriptor
`msg` = address of user structure to contain the received message
`maxcount` = size of data array in msg
`type` = message type

- The kernel checks that the user has necessary **access rights to the message queue**.

- If the message type is 0, the kernel finds the first message on the linked list'
- If the msg size $\leq$ size user requested, the kernel copies the message data to the user data structure and adjusts its internal structures appropiately
  - Decrements the count of messages on the queue
  - The number of data bytes on the queue
  - Sets the receiving time and receiving process ID
  - Adjusts the linked list
  - Awaken the process who were waiting for room on the queue.
- Else, return an error
- If the process ignores size constraints (`MSG_NOERROR`), however, the kernel truncates the message, returns the requested number of bytes, and removes an entire message from the list.

`type > 0` returns the first message of given type
`type < 0` finds the lowest type of all messages on queue, and returns the first message of that lowest type

For example: message types are 3,1 and 2 and `type = -2`, the kernel returns the first message of type 1.

If no messages on the queue satisfy the recieve request, the process sleeps unless the process has specified the `IPC_NOWAIT` flag.

msgctl

```
msgctl(id, cmd, mstatbuf);
```

`cmd` specifies the type of command
`mstatbuf` contains the control parameters (user data structure) or results of a query

A process can query:

- The status of a message descriptor,
- Set its status
- remove a message descriptor