OS 2019 Homework1

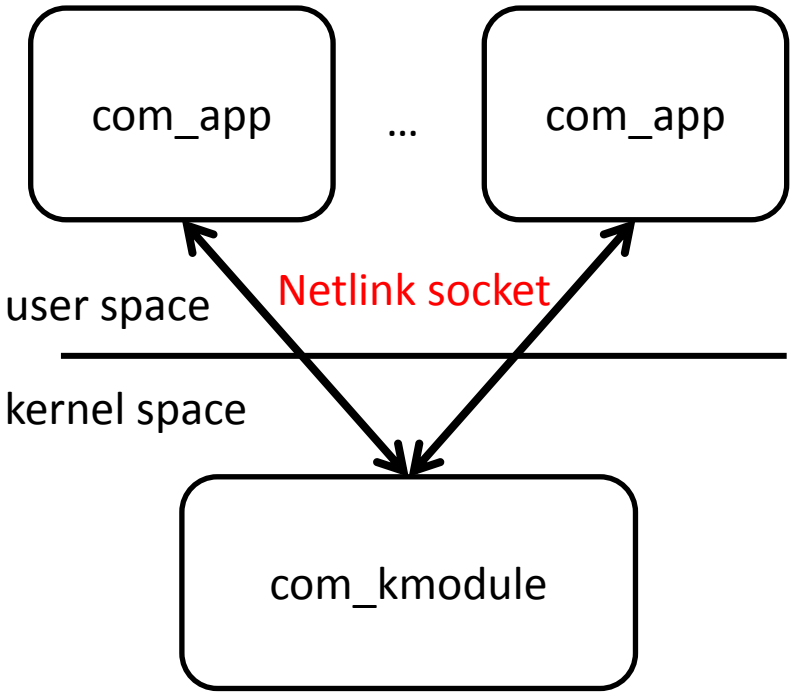
Process communication

(Due date: 2019/10/31 23:59)

Objectives

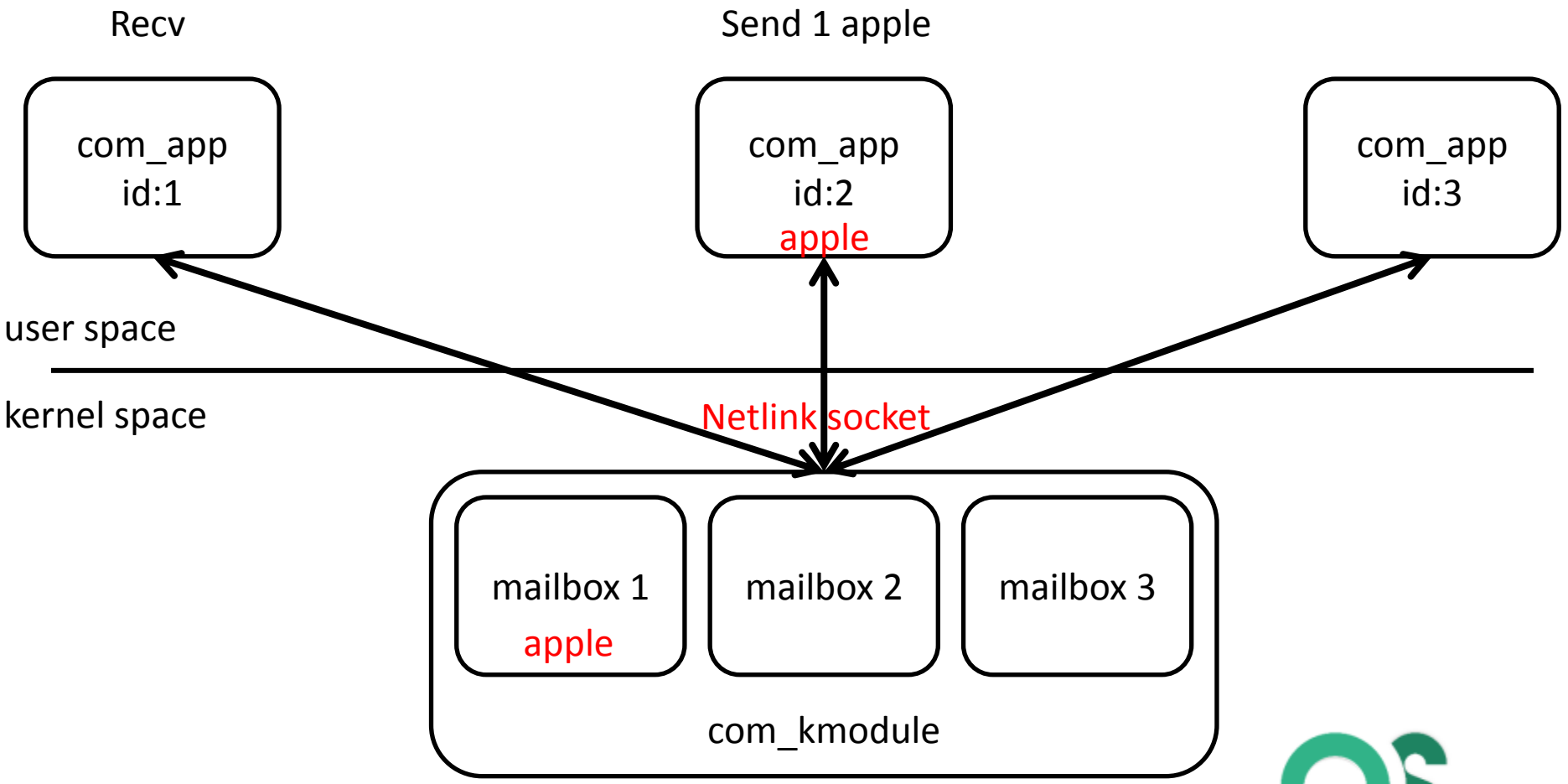
- Understand how to program a kernel module
- Understand how to transfer information between the kernel and the user space processes

Overview



1. Several com_apps send registration request to register with kernel module via the socket
2. After receiving registration request, com_kmodule creates a mailbox for each com_app and returns ack message
3. Mailboxes are used to manage message data from com_apps
4. com_app can communicate with each other through com_kmodule

Overview



Two types of user space application

- There are two types of com_app, queued com_app and unqueued com_app.
- The detail is explained in next two pages

Queued application

- It is like that there is a FIFO queue in kernel module. When the queue is empty, the module does not provide any message data to the application. When the queue is not empty and the application wants to receive the message data, then the module provides the application with the oldest message data and removes this message data from the queue.
- If new message data arrives in kernel module and the queue is not full, this new message data is stored in the queue. If new message data arrives and queue is full, this message data is lost.(Our homework can store at most 3 message data)

Unqueued application

- Unqueued application does not use the FIFO mechanism. The application does not consume the message data in kernel module during reception of message data – instead, a message data may be read multiple times by an application.
- If no message data is sent for the application after the mailbox is created, the module does not provide any message data to the application.
- Message data is overwritten by newly arrived message data.

Requirement – user space application

- Execution method:
  - ./com_app [id] [type]
  - id must be a positive integer (1~1000)
  - type must be “queued” or “unqueued”
  - For example, “./com_app 2 unqueued”
- Application must send a registration request and message data to the module by netlink socket
  - *netlink_socket = socket(AF_NETLINK, SOCK_RAW, ...);*
- Registration request:
  - Each application should send a registration request to kernel module
  - Format: “Registration. id=[id], type=[type]”
  - For example, “Registration. id=2, type=unqueued”

Requirement – user space application

- Ack message:
  - After sending registration request, kernel module will return a ack message. The ack message will be “Success” or “Fail”
  - You should print the ack message on the terminal
  - If it is “Success”: wait for the user’s input to communicate
  - If it is “Fail”: terminate this process
- Communication:
  - User can input commands to send or receive message data multiple times after registration
  - Application should send a command to module and print the message that is returned by kernel module

Requirement – user space application

- Send message data:
  - The input command: “Send [id] [message data]”
    - id: the process to which you want to send
    - message data: any string(may include spaces)
  - The command sent to kernel: the same as input
  - The message returned by kernel: “Success” or “Fail”
- Receive message data:
  - The input command: “Recv”
  - The command sent to kernel: “Recv [id]”
    - id: the registration id
  - The message returned by kernel: [message data] or “Fail”

Requirement – kernel module

- Registration
  - When module receives a registration request, module should check whether the id has been used
    - If the id has not been used, module should allocate a mailbox to manage later communication and return "Success"
    - If the id has been used, module should return "Fail"
    - New registration after communication should be allowed
  - Data structure in kernel:

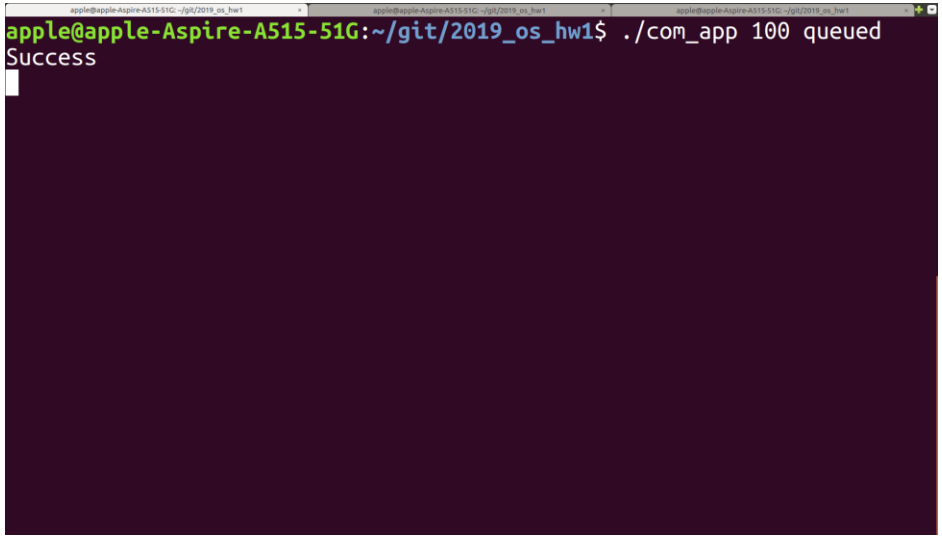
```
struct mailbox
{
    //0: unqueued
    //1: queued
    unsigned char type;
    int msg_data_count;
    struct msg_data *msg_data_head;
    struct msg_data *msg_data_tail;
};
```

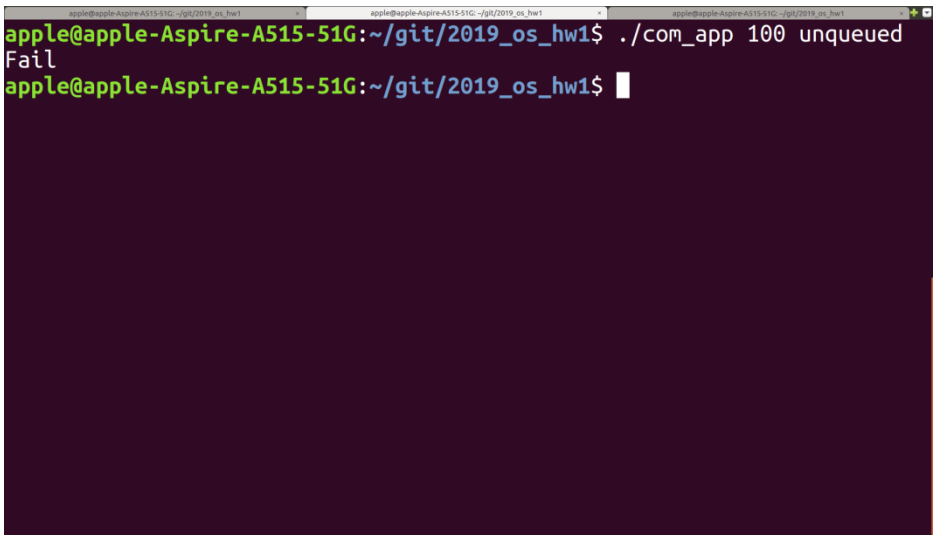
```
struct msg_data
{
    char buf[256];
    struct msg_data *next;
};
```

Requirement – kernel module

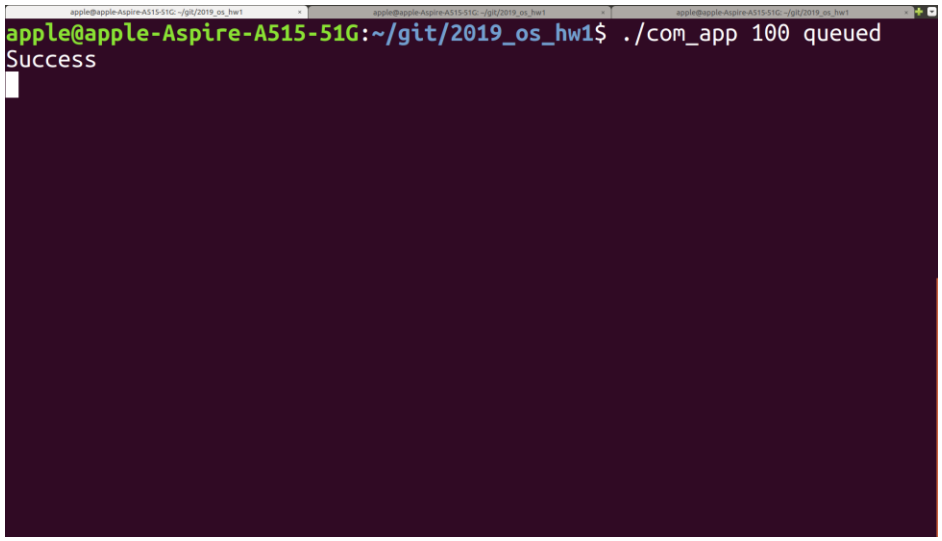
- Handle the send command: “Send [id] [message data]”
  - If the id is not exist: return “Fail”
  - If the length of message data more than 255 bytes: return “Fail”
  - If the type of the application is queued application and the queue is full: return “Fail”
  - Otherwise, return “Success” and store the message data
- Handle the receive command: “Recv [id]”
  - If the id is not exist: return “Fail”
  - If there is no message data to receive: return “Fail”
  - Otherwise, return the message data
- The message data should be handled follow the rules in page 5 and 6

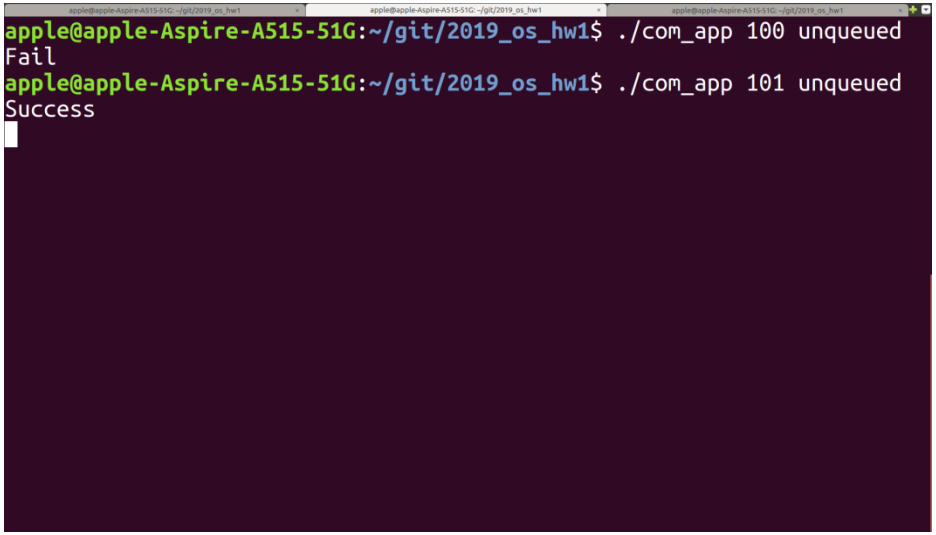
Example: id has been used



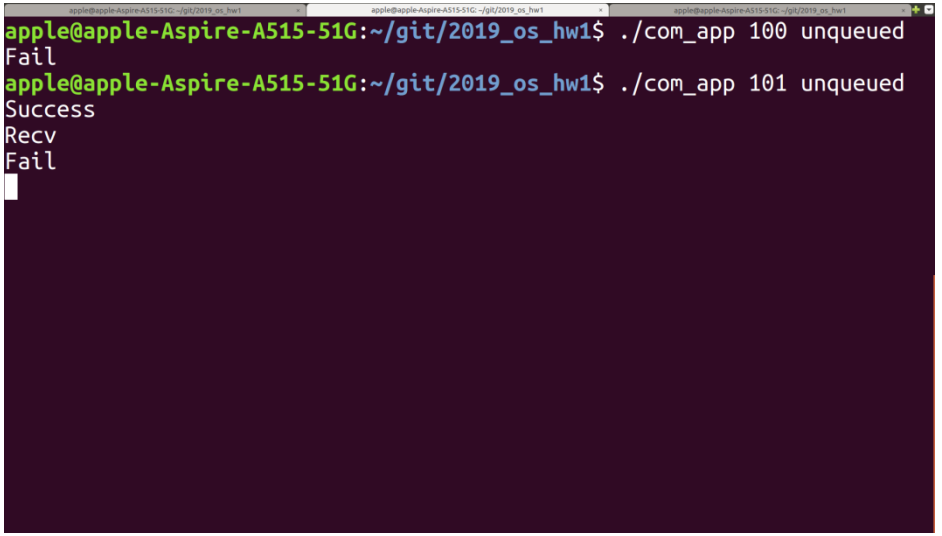


Example: use different id

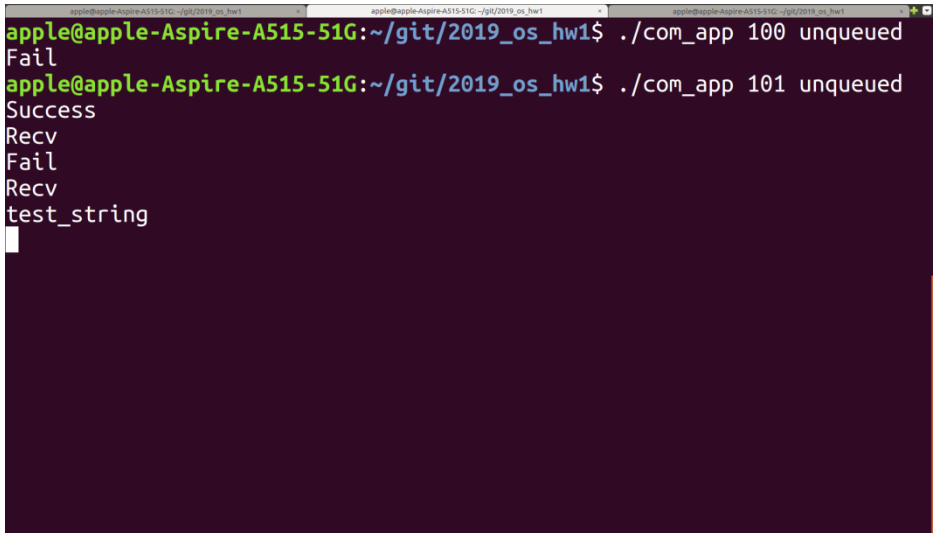




Example: there is no message to receive



Example: successfully send and receive



Example: queued application receive

```
apple@apple-Aspire-A515-51G:~/git/2019_os_hw1$ ./com_app 101 unqueued
Success
Send 100 test1
Success
Send 100 test2
Success
Send 100 test3
Success
Send 100 test4
Fail
```

```
apple@apple-Aspire-A515-51G:~/git/2019_os_hw1$ ./com_app 100 queued
Success
Recv
test1
Recv
test2
Recv
test3
Recv
Fail
```

Example: unqueued application receive

```
apple@apple-Aspire-A515-51G:~/git/2019_os_hw1$ ./com_app 108 queued
Success
Send 107 test1
Success
Send 107 test2
Success
Send 107 test3
Success
Send 107 test4
Success
```

```
apple@apple-Aspire-A515-51G:~/git/2019_os_hw1$ ./com_app 107 unqueued
Success
Recv
test4
Recv
test4
```