Assignment2

Student ID:105062617
Name:林展逸

1. 目錄結構

以下是本次作業的目錄

```
HW2
|___containerClient
      |___rootfs
           |___mnt_echo_client.c
      |___config.json
|___containerServer
      |___rootfs
           |___ipc_echo_server.c
           |___msg_helper.h
     |___config.json
|___bridge.c
|___msg_helper.h
```

2. 執行

step1: 啟動 client 與 server

Server:

```
$ sudo runc run --pid-file /tmp/runc1.pid server
$ gcc  ipc_echo_server.c –o server
$ ./server
```

Client:

```
$ sudo runc run --pid-file /tmp/runc2.pid client
$ gcc mnt_echo_client.c –o client
$ ./client
```

step2: 啟動 bridge

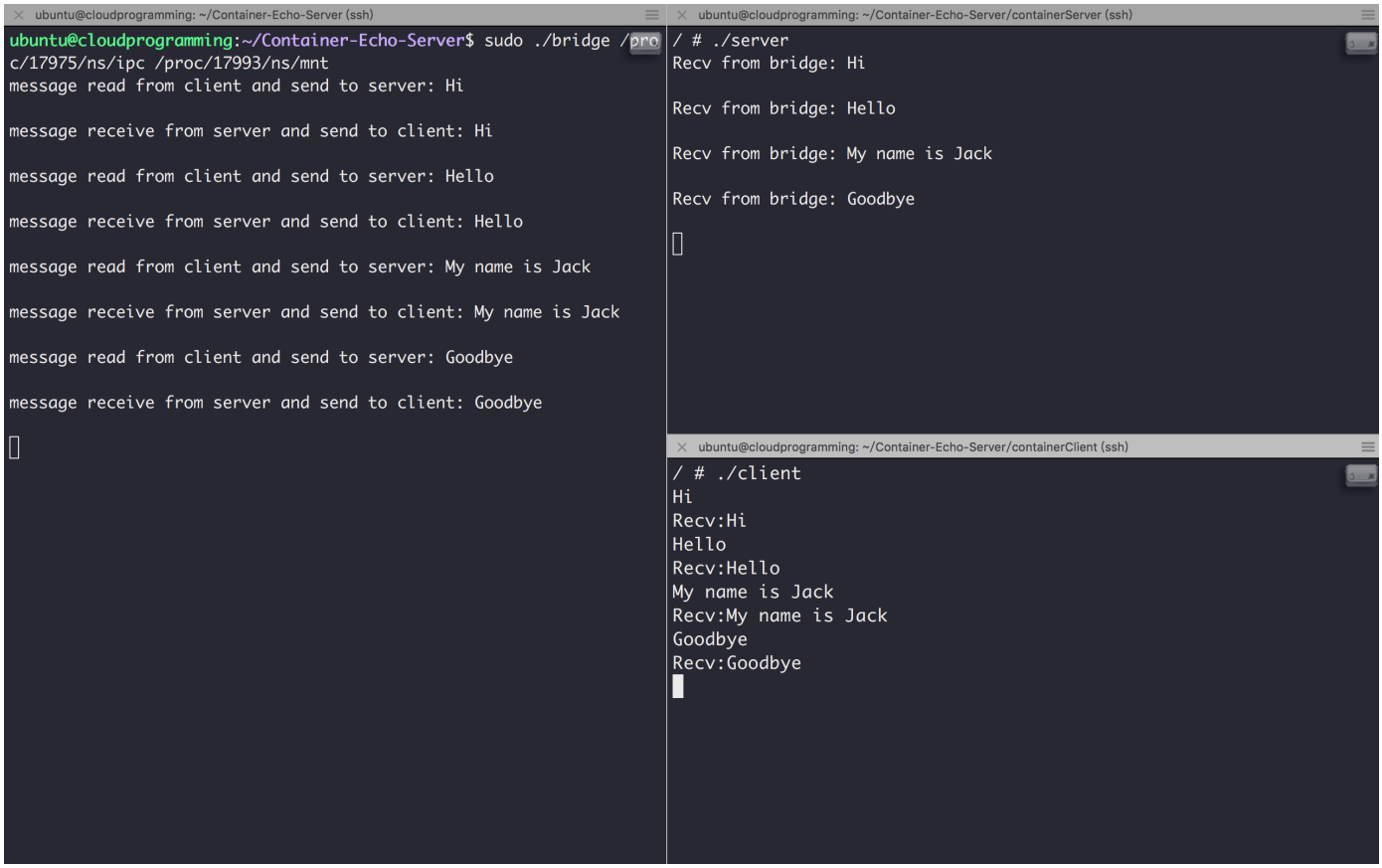
取得 pid:

```
$ cat /tmp/runc1(2).pid
```

bridge:

```
$ gcc bridge.c -o bridge
$ sudo ./bridge /proc/{containerServerPID}/ns/ipc  /proc/{containerClientPID}/ns/mnt
```

以下為執行結果



*註：為方便分辨 input 與 echo 回來的訊息，故 client 收到 echo 回來的訊息會加上 Recv 字樣。

3. 實作

在本次作業中，Client 採用 mnt 方法與 bridge 溝通; Server 採用 ipc 方法與 bridge 溝通。Bridge 則是在一開始透過參數取得兩個 container 的 pid 後，透過 setns 方式將自己的 mnt namespace 設定與 client 一樣，而 ipc namespace 與 Server 一樣。

以下介紹使用到之 systemcall

ipc 部分

(1). msgget

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

根據給定的 key 回傳 message queue 的 identifier。

msgflg 可以給定 IPC_CREAT 或是 IPC_CREAT|IPC_EXCL。前者表示若 message queue 存在即返回其 identifier 或是不存在則重新創建。後者表示，若不存在則重新創建但若存在則回傳 error。

(2). msgsnd

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

在前一個 function 取得 id 後，即可將 message 透過此 system call 寫入該 message queue

特別注意的是，msg.type 只有在 recv 時候可以設定成 0，在 send 時設定成 0 會造成錯誤。

(3). msgrcv

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

透過取得同一個 key 的 message queue 之 id，我們可以用此 system call 取得 sender 寫入的 message。

mnt 部分

透過 Linux 中的 inotify 機制，我們有能力去監控檔案的一系列改變事件。Inotify 將檔案的變化設定為 10 中：

IN_ACCESS:File was read from.
IN_MODIFY:File was written to.
IN_ATTRIB:File's metadata (inode or xattr) was changed.
IN_CLOSE_WRITE:File was closed (and was open for writing).
IN_CLOSE_NOWRITE:File was closed (and was not open for writing).
IN_OPEN:File was opened.
IN_MOVED_FROM:File was moved away from watch.
IN_MOVED_TO:File was moved to watch.
IN_DELETE File:was deleted.
IN_DELETE_SELF:The watch itself was deleted.

(1). inotify_init

初始化 inotify instance 並回傳一個 file descriptor. 該 file descriptor 關聯一個 inotify event queue.

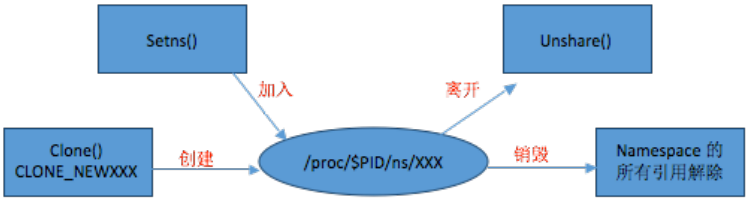
(2). inotify_add_watch

```
int inotify_add_watch(int fd, const char *pathname, uint32_t mask);
```

pathname 指定要監控的 file 其目錄。fd 為 inotify_init 取得的，最後一個參數讓使用者可以設定要監控得事件類型，或是也可以全部監控。此 system call 回傳一個 unique watch descriptor。

namespace 部分

先來介紹 namespace，namespace 是一種純軟體的隔離，在同一個 namespace 下的 processes 彼此可以看到的對方的資源進而可以互相通訊。



來自網路上的示意圖。

```
/ # ls -al /proc/self/ns/
total 0
dr-x--x--x    2 root     root             0 Apr 15 02:28 .
dr-xr-xr-x    8 root     root             0 Apr 15 02:28 ..
lrwxrwxrwx    1 root     root             0 Apr 15 02:28 ipc -> ipc:[4026532283]
lrwxrwxrwx    1 root     root             0 Apr 15 02:28 mnt -> mnt:[4026532281]
lrwxrwxrwx    1 root     root             0 Apr 15 02:28 net -> net:[4026532461]
lrwxrwxrwx    1 root     root             0 Apr 15 02:28 pid -> pid:[4026532284]
lrwxrwxrwx    1 root     root             0 Apr 15 02:28 uts -> uts:[4026532282]
```

一個 process 下面，每一個部分在一個 namespace，例如 ipc 便是在 ipc namespace，mnt 便是指 mount namespace。

Container 即是透過此種技術來達到隔離的目的。此作業中便是透過以下 setns function 來讓 bridge 進入 client 與 server 兩個 container 的 namespace 中，與兩個 container 通訊。

(1). Setns

```
int setns(int fd, int nstype);
```

第一個參數傳入打開/proc/{containerServerPID}/ns/ipc（例）得到的 fd，第二個參數表示要加入的 namespace 類型。這個 fuction 或將此 process 加入到該 namespace 底下。

以上就是簡單介紹此次作業的實作。

4. 心得

由於本身研究方向包含 OpenStack 與 Kubernetes/Docker。因此經常使用到 Container 技術，但是由於比較著重在上層管理編排的角度，故還未如此深入得了解 Container 實作方法。透過此次作業，不管對於 namespace 概念，或是 runc 手動啟動 Container 之方法還有 Container 網路細節都有很好認識。了解底層之實踐，對於未來研究也有很好得幫助。