Design Document

作者 or 組員：陳育霖 B113040052

目標：Synchronization and Mailbox

新增或修改的檔案：

System Call: misc.c proto.h table.c callnr.h

Test: shell.c sender.c receiver.c shared_memory.h (minix 版本使用)

程式使用的函式：

putchar 將輸入的文字放入 buffer 中；getchar 將 buffer 中的文字輸出

fork 在 Spawn 中負責建立新 Process；waitpid 在 Waitpid 中用以等待子 Process

execvp 在 Spawn 中用來執行指令；usleep 避免導致顯示問題或 Race condition

strtok 將文字分割；open 開啟檔案；mmap 將地址 map 進變數中

shmget 建立共享記憶體；shmat 啟動記憶體功能；shmdt 關閉記憶體功能

右下方函式 -> 用以輸出 buffer 內容： va_list 宣告可變參數；

va_start 初始化可變參數；va_arg 從中獲取下個值；va_end 結束可變參數

檔案路徑如下：

misc.c /usr/src/minix/servers/pm/ 設定 system call 的 function

proto.h /usr/src/minix/servers/pm/ 設定 system call function 的格式

table.c /usr/src/minix/servers/pm/ 設定 system call 代碼對應的 function

callnr.h /usr/src/minix/include/minix/ 設定 system call 的代碼

編譯步驟如下：

編譯 system call

cd /usr/src/minix/servers/pm

make clean

make

編譯環境

cd /usr/src/releasetools

make clean

make hdboot

流程圖：

mmap 版本

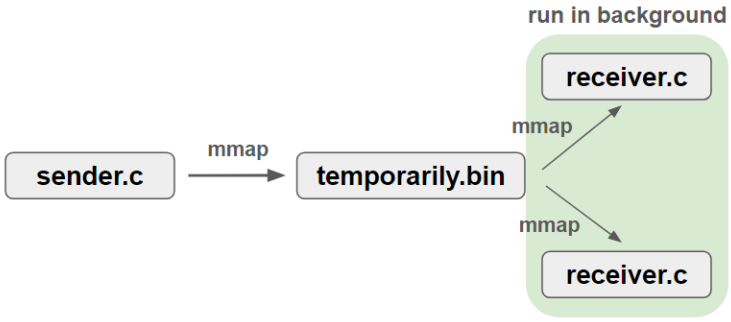

圖 1 傳送訊息流程圖

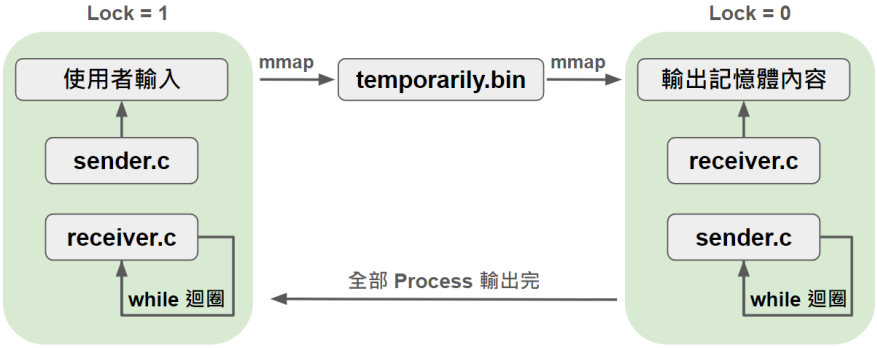

圖 2 狀態流程圖

shm 版本

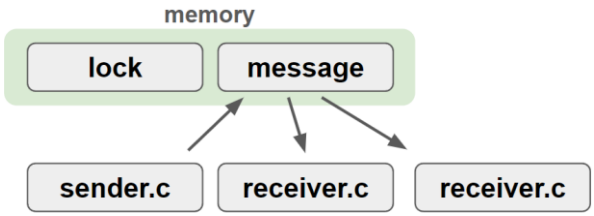

圖 3 傳送訊息流程圖

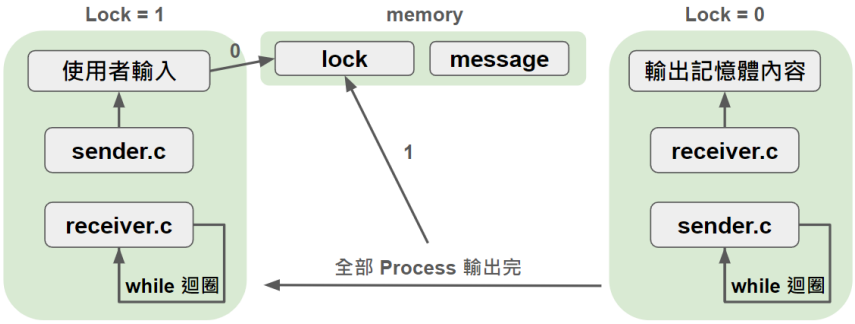

圖 4 狀態流程圖

使用的 Trap 功能

Spawn	創立子 Process 並執行使用者輸入的指令
Waitpid	等待子 Process 執行結束，並在結束後 return PID
Getchar	處理使用者輸入的資料，並將其存在 buffer 中
Putchar	將 buffer 中的內容輸出
ProcessFork	程式可以透過 ProcessFork 來建立 Process

功能介紹：

Mailbox 是單向的，一個 Process 負責發送訊息，其他 Process 負責接收訊息。

製作過程面臨的問題：

1. 在設計 mailbox 的記憶體時，直接使用 mmap 的話，在同個程式不同 Process 時，可以共用記憶體內容，但換成不同程式的不同 Process 時，每個程式無法知道其他程式的記憶體位置。
2. 沒有使用互斥鎖的話，接收方會一直輸出內容，使畫面混亂；而發送方會不知道接收方是否收到訊息，以及每個 Process 是否收到後都輸出完成。
3. 在設計 shell 時，putchar 無法將變數加在字串中，例如：("Process%d", ID)。

處理同步問題：

mmap 版本

為了使不同程式之間可以互相溝通，我使用.bin 檔(二進位檔案)來當作一個訊息傳遞的媒介，將 mmap 的位置指向該檔案，進而達到共享記憶體內容的功能，並使用一個 bin 檔來傳送 lock 的資訊，同時解決同步問題。

shm 版本

在 shared_memory.h 中設定 key，不同的程式使用相同的 key 來設定記憶體，這樣程式間就可以使用相同記憶體位置來傳送資訊，並在該記憶體中設定 lock 變數來處理同步問題。

預防重複讀取：

使用 lock 來記錄。當 Process 接收到消息，並順利輸出後，lock 的數字會從 0 變成 1，這樣迴圈再次跑到判斷時，就會鎖住，避免重複讀取與輸出。當全部訊息都輸出後，發送訊息的 Process 的鎖被解開時，會提示使用者輸入數字，當使用者輸入完數字後，lock 的數字會從 1 變成 0，此時，接收訊息的 Process 才可再次輸出新的內容。

測試結果：

Input & output　輸入:　lab^H^Hscd^H^H -t　//　<CONTROL-H>是 Backspace

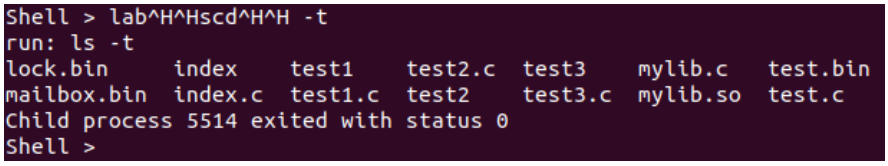

圖 5 輸入輸出測試

Testing (test1 為發送者、test2 為接收者)

接收者只會輸出，不須輸入，因此讓其在後台執行；而發送者需一直輸入，因此需使用 Waitpid 等待。

```
Shell > ./test2 &
run: ./test2 &
Shell > ./test2 &
run: ./test2 &
Shell > ./test1
run: ./test1
Input something: 123
Process 1: 123
Process 0: 123
Input something: Hello World!
Process 0: Hello World!
Process 1: Hello World!
Input something:
```

圖 6　Mailbox 測試(2 child Process)

```
Shell > ./test2 &
run: ./test2 &
Shell > ./test2 &
run: ./test2 &
Shell > ./test2 &
run: ./test2 &
Shell > ./test2 &
run: ./test2 &
Shell > ./test1
run: ./test1
Input something: 456
Process 0: 456
Process 2: 456
Process 1: 456
Process 3: 456
Input something: 你好
Process 3: 你好
Process 1: 你好
Process 2: 你好
Process 0: 你好
Input something:
```

圖 7　Mailbox 測試(4 child Process)

檔案修改的內容(System call):

callnr.h

```
#define PM_SPAWN        (PM_BASE + 48)
#define PM_WAITPID      (PM_BASE + 49)
#define PM_GETCHAR      (PM_BASE + 50)
#define PM_PUTCHAR      (PM_BASE + 51)
#define PM_PROFORK      (PM_BASE + 52)
#define NR_PM_CALLS     53
```

table.c

```
CALL(PM_SPAWN)      =     do_spawn,   //  在最後一行加上
CALL(PM_WAITPID)    =     do_waitpid,
CALL(PM_GETCHAR)    =     do_getchar,
CALL(PM_PUTCHAR)    =     do_putchar,
CALL(PM_PROFORK)    =     do_processfork
```

proto.h

```
/* my syscall lib */   //   在最後一行加上\
int do_spawn(char *prog, char *arg[], int background);
int do_waitpid (pid_t pid);
void do_putchar(char c);
void do_putchar(char c);
int do_processfork();
```

misc.c

```
void do_putchar (char c) {
        putchar(c);
}
char input_buffer[100];    //    getchar
int buffer_index = 0;    //    getchar
void do_getchar(char c) {
        if (c == '\n') {
                input_buffer[buffer_index] = '\0';
                buffer_index = 0;
        } else if (c == '\b') {
                Backspace();
        } else {
                if (buffer_index < BUFFER_SIZE - 1) {
                        input_buffer[buffer_index++] = c;
                }
        }
}
int do_waitpid(pid_t pid) {    //    Waitpid 函式
        int status;
        pid_t ret_pid = waitpid(pid, &status, 0);    // 等待子 Process 結束
        if (ret_pid == -1) {
                perror("waitpid");
                return -1;    // 表示有問題
        }
        if (WIFEXITED(status)) {
                printf("Child process %d exited with status %d\n", ret_pid,
WEXITSTATUS(status));
        } else {
                printf("Child process %d did not exit normally\n", ret_pid);
        }
        return ret_pid;    // return Process 的 ID
}
```

```
int do_processfork() {    // 建立 Process 的函式
    pid_t pid = fork();    // 調用 fork 函式創建一個新的 Process
    if (pid == -1) {
        return -2;    // return 錯誤
    } else if (pid == 0) {
        // 子進程
        //printf("Child process created with PID: %d\n", getpid());
    } else {
        return pid;    // return Process 的 ID
    }
    return pid;
}
int do_spawn(char *prog, char *arg[], int background) {    // 執行指令及分配 Process
    int result = ProcessFork();    // 調用 ProcessFork 函式創建一個新的 Process
    if (result < 0) {
        return result;    // 如果創建失敗，直接返回錯誤碼
    } else if(result == 0) {
        execvp(prog, arg);
        perror("execvp");
    }
    if(!background)    // 在背景執行，不須等待
        return Waitpid(result);    // return Process 的 ID
    usleep(50);
    return result;
}
```