Đại học Bách Khoa TP. HCM, Khoa KH & KTMT

Bài thực hành 9
Môn: Hệ điều hành

Phương-Duy Nguyễn
Email: duynguyen@cse.hcmut.edu.vn

Ngày 29 tháng 2 năm 2016

Mục lục

Chương 1

Giới thiệu chung

Nội dung sinh viên cần chuẩn bị trước bao gồm các nội dung liên quan đến việc giao tiếp giữa các process.

Nội dung bài thực hành giới thiệu với sinh viên nhiều cơ chế hỗ trợ việc giao tiếp giữa các process cũng như các chuẩn phát triển của các cơ chế này. Có nhiều cơ chế được đề cập, nội dung chính cần tập trung thử nghiệm là các cơ chế tiêu biểu là chia sẻ vùng nhớ (shared memory) và truyền thông điệp (message passing). Các nội dung còn lại, sinh viên về tham khảo thêm trong phụ lục của tài liệu này.

Yêu cầu Sinh viên hiện thực phần nội dung trong phần mô tả bài tập 5.

Chương 2

Inter-Process Communication - IPC

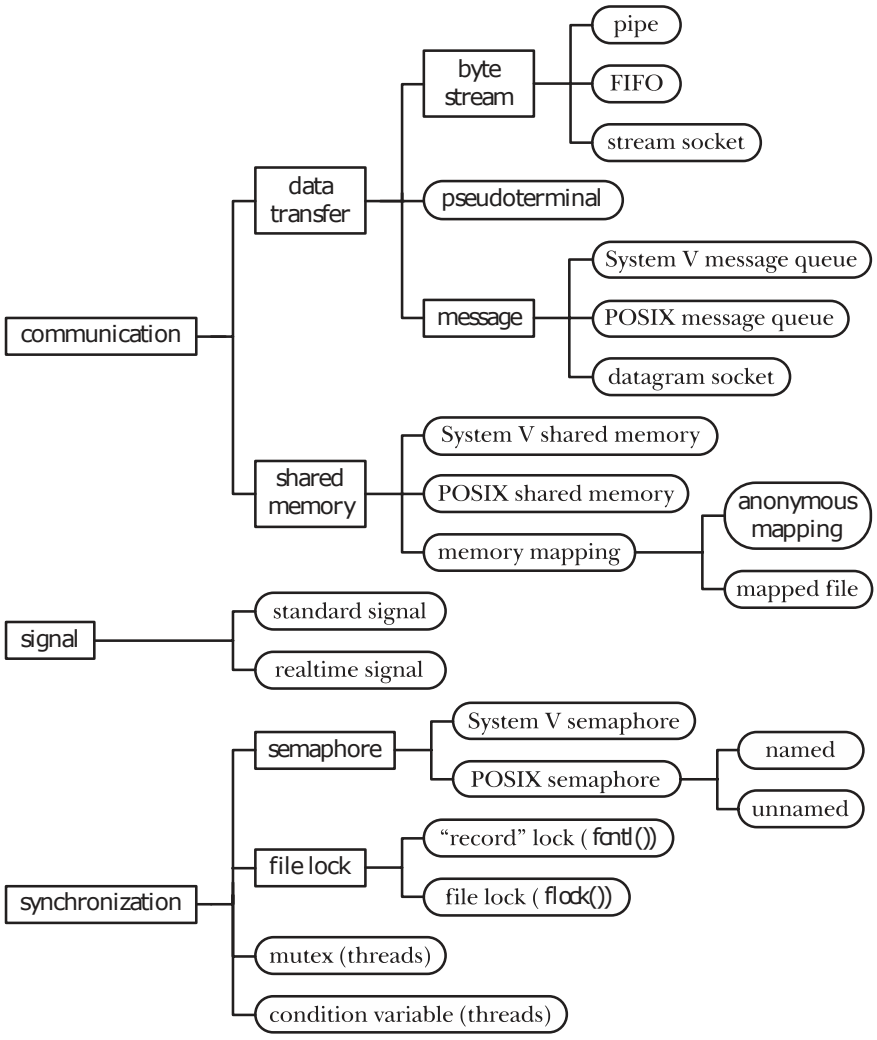
- IPC là viết tắt của Inter-process Communication - Giao tiếp giữa các process cho phép các process có thể giao tiếp với nhau.
- Mỗi process có một vùng không gian địa chỉ riêng biệt, cần tìm cách thức nào đó để có thể giao tiếp giữa các process. Đáp án là kernel, phần lõi hệ điều hành có quyền truy xuất tới tất cả các vùng nhớ, cho phép cấp các vùng không gian mà các process có thể sử dụng để giao tiếp với nhau.
- Các process có thể sử dụng các phương thức cổ điển như giao tiếp qua cùng một file để thực hiện các tác vụ đọc/ghi dữ liệu chung. Cách này tốn khá nhiều thời gian do tốc độ đọc ghi file chậm.
- Trong mội trường UNIX, hệ thống cung cấp nhiều phương thức cho phép các process, có thể ở cùng một máy tính hay ở hai máy tính khác nhau trong cùng mạng máy tính, giao tiếp với nhau.

Bảng 2.0.1: Phân loại IPC

Dạng	Mô tả	Phạm vi	Ứng dụng
File	Dữ liệu được đọc và ghi từ UNIX file thông thường. Các process có thể giao tiếp qua dữ liệu chứa trong file.	Cục bộ (local)	Chia sẻ một lượng dữ liệu lớn. Tốc độ giao tiếp là một hạn chế.
Pipe	Dữ liệu được truyền giữa hai process qua một dạng file đặc biệt. Dạng giao tiếp này chỉ tiến hành được giữa hai process cha con.	Cục bộ(local)	Mô hình chia sẻ dữ liệu đơn giản dạng như producer và consumer.
Named pipe	Dữ liệu được truyền giữa hai process qua một dạng file đặc biệt. Dạng giao tiếp này có thể tiến hành giữa hai process ở trên cùng một hệ thống	Cục bộ (local)	Cũng giống trên là mô hình dạng như producer và consumer, hoặc là command-and-control ví dụ như công cụ command-line query.
Signal	Một thông tin đặc biệt gây ra ngắt thông báo cho process về một điều kiện đặc biệt để xử lý.	Cục bộ (local)	Không dùng để truyền dữ liệu (chỉ là dạng số định danh ID), chủ yếu để quản lý process.
Shared memory	Thông tin được chia sẻ thông qua việc đọc và ghi một vùng nhớ chung	Cục bộ (local)	Có thể cộng tác nhiều loại task, cần lưu ý vấn đề bảo mật
Stream socket	Sau khi thiết lập qua một số bước, dữ liệu được truyền qua các tác vụ input/ouput cơ bản	Cục bộ (local) hoặc remote	Các dịch vụ mạng như FTP, ssh, HTTP (web server)
Datagram socket	Dạng socket không cần thiết lập kết nối, các gói tin chỉ gửi đi các gói tin và nhận các gói tin đến đích. Mỗi gói tin có địa chỉ và đường đi khác nhau.	Remote	Các dịch vụ broadcast.

- Half-duplex UNIX pipes
- FIFOs (named pipes)
- Full-duplex pipes (STREAMs pipes)
- SystemV-style shared memory segments
- SystemV-style message queue
- SystemV-style semaphore sets
- Network sockets (Berkerley style)

Ghi chú: Với hệ thống System V, AT&T đã giới thiệu ba dạng của IPC (message queue, semaphore, và shared memory). Trong lúc cộng đồng POSIX chưa hoàn chỉnh tiêu chuẩn hóa các giao tiếp này, hầu hết các hiện thực hỗ trợ chuẩn System V. POSIX IPC ít được hiện thực phổ biến hơn. GNU/Linux tương thích một phần với POSIX và nhiều nhà cung cấp sẽ hướng đến việc hỗ trợ POSIX. Hai chuẩn này đều có chung các phương thức ví dụ như message queue và semaphore và shared memory. Chúng cung cấp các giao tiếp có một chút khác biệt nhưng những khái niệm cơ bản là giống nhau.



Hình 2.0.1: Phân loại Linux IPC

Chương 3

Giao tiếp giữa các quá trình - Shared memory

3.1 Giới thiệu shared memory

- Trong Shared memory, dữ liệu được truy xuất từ không gian địa chỉ chia sẻ. Shared memory là một dạng giao tiếp giữa các process. Trong đó, thông tin được chia sẻ bằng cách đọc và ghi từ một vùng nhớ chung.
- Phạm vi sử dụng của nó có thể cục bộ trong một hệ thống hoặc giữa các hệ thống (node) kết nối với nhau thông qua một thành phần trung gian ánh xạ giữa địa chỉ cục bộ và địa chỉ trên mạng liên kết (network).
- Shared memory được sử dụng để cộng tác công việc giữa nhiều hệ thống khác nhau, nhưng cần lưu ý về vấn đề bảo mật
- Cách sử dụng Shared memory
  - Vùng Shared memory phải được tạo trước.
  - Process phải gắn vùng shared memory vào không gian địa chỉ của mình trước khi sử dụng.
  - Sau khi dùng xong có thể gỡ vùng shared memory ra khỏi không gian địa chỉ của process.
- Các thao tác:

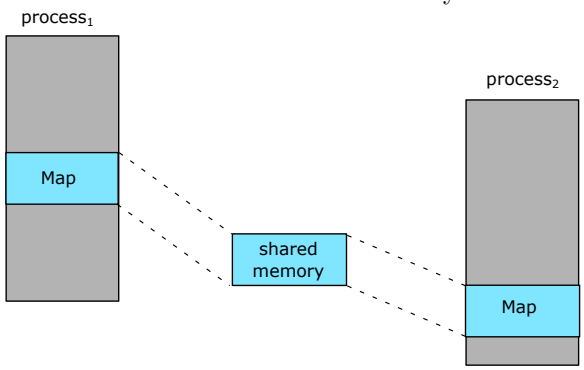
  `shmget()` Khởi tạo;

  `shmctl()` Lấy hoặc thay đổi thuộc tính;

  `shmat()` Gắn Shared memory vào không gian địa chỉ nhớ;

  `shmdt()` Gỡ Shared memory ra khỏi không gian địa chỉ nhớ.

Hình 3.1.1: Shared memory



- Ngoài API lập trình, các vùng shared memory trong hệ thống có thể được liệt kê trong terminal dùng lệnh ipcs và xóa bỏ một vùng shared memory dùng lệnh ipcrm. Chi tiết tham khảo phần phụ lục A

3.2 Cách sử dụng Shared memory

Gắn shared memory vào không gian địa chỉ sử dụng lệnh shmat()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

void *shmat(int shmid, void *shmaddr, int shmflg);
```

`shmid` Shared memory ID trả gề từ hàm `shmget()`

`shmaddr` Địa chỉ nơi gắn vùng nhớ chia sẻ

`shmflg` SHM_RDONLY (read-only) hoặc 0

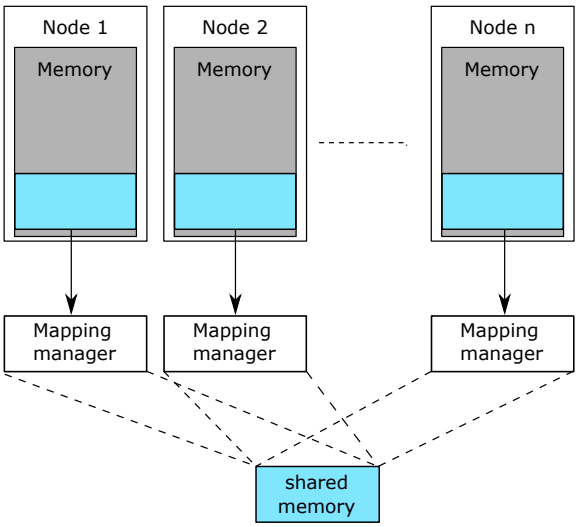
Gỡ shared memory khỏi không gian địa chỉ sử dụng lệnh shmdt()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmdt(void *shmaddr);
```

`shmaddr` Địa chỉ shared memory , được trả về từ hàm `shmat()`

Hình 3.1.2: Distributed shared memory



KHỞI TẠO VÀ TÙY CHỈNH SHARED MEMORY sử dụng lệnh shmget() để khởi tạo vùng shared memory, lệnh shmctl() để tùy chỉnh(/hoặc xóa bỏ) vùng shared memory.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, int size, int shmflg);
```

Return value Trả về 0 nếu thực thi lệnh thành công, trả về -1 nếu có lỗi xảy ra

`key` là giá trị key value để so sánh với các key có trong kernel cho các vùng shared memory khác. Có thể dùng hàm ftok(const char *pathname, int proj_id); để chuyển đường dẫn một file và project identifier thành IPC key.

`size` Kích thước vùng nhớ (Đơn vị byte)

`shmflg` IPC_CREAT để tạo vùng dữ liệu mới. IPC_EXCL dùng cùng IPC_CREAT để thiết lập báo lỗi khi vùng nhớ đã tồn tại.

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Return value Trả về 0 nếu thực thi lệnh thành công, trả về -1 nếu có lỗi xảy ra

`shmid` shared memory ID trả về từ `shmget()`

`cmd` IPC_STAT, IPC_SET để lấy/chỉnh sửa thiết lập shared memory hoặc IPC_RMID để xóa bỏ vùng shared memory.

`buf` cấu trúc dữ liệu để quản lý thiết lập của shared memory, được khai báo trong `linux/shm.h`. Trong trường hợp các thiết lập không đổi, tham số có thể được truyền vào là NULL (giá trị 0).

3.3 Ví dụ sử dụng shared memory

3.3.1 Ví dụ 1

Ví dụ này minh họa việc chia sẻ dữ liệu giữa hai process có quan hệ cha-con:

Mô tả chương trình hiện thực hai process có quan hệ cha con chia sẻ cùng một vùng nhớ shared memory. `Lệnh fork()` tạo process mới được gọi sau lệnh `shmget()` khởi tạo vùng nhớ và lệnh `shmat()` gắn vùng nhớ vào địa chỉ process. Lúc này, vùng nhớ chia sẻ shared memory có thể được truy xuất bởi cả hai process cha-con.

```
1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <sys/ipc.h>
4  #include <sys/shm.h>
5
6  int main(int argc, char* argv[])
7  {
8
9      int *shm, shmid;
10
11     shmid = shmget(IPC_PRIVATE, 128, IPC_CREAT|0666);
12
13     shm = (int*) shmat(shmid,NULL,0);
14
15     if(fork() == 0){ /*child*/
16         shm[0]=111;
17         shm[1]=999;
18
19         sleep(3);
20
21         printf("Process %d reads: Sum = %d\n",
22                 getpid(), shm[2]);
23
24         shmdt((void*)shm);
```

```
25          shmctl(shmid, IPC_RMID, (struct shmid_ds *)0);
26      } else { /*parent*/
27          sleep(1);
28
29          printf("Process %d writes to shared memory\n",
30                  getpid());
31
32          shm[2] = shm[0]+shm[1];
33
34          shmdt((void*)shm);
35      }
36
37      return 0;
38  }
```

Biên dịch và thực thi với file hiện thực mã nguồn chương trình được đặt tên là `vd1_shm.c`

```
$ gcc -o vd1_shm vd1_shm.c

$ ./vd1_shm
Process 7558 writes to shared memory
Process 7559 reads: Sum = 1110
```

3.3.2 Ví dụ 2

Mô tả chương trình minh họa chia sẻ dữ liệu giữa hai process bất kỳ. Ví dụ gồm hai file mã nguồn hiện thực hai process đóng vai trò server và client. Process server khởi tạo chuỗi dữ liệu trong vùng nhớ chia sẻ. Process client

```
1   /* Source code shm_server.c */
2   #include <stdio.h>
3   #include <sys/types.h>
4   #include <sys/ipc.h>
5   #include <sys/shm.h>
6
7
8   int main(int argv, char* argn)
9   {
10      int shmid;
11      char c;
12      char *shm, *s;
13
14      /* Use ftok to generate a key associated with a file */
15      key_t key = ftok("/tmp/shm",'a');
```

```
16
17      /* Create/Locate the memory segment */
18      // TODO: add code to create memory segment using shmget()
19      //       size is fixed to 128
20      //       key is provided by key_t key
21
22      /* Attach the memory segment to our address space */
23      // TODO: add code to attach memory segment using shmat()
24      //       the return address is assign to pointer *shm
25
26      /* Now we put some thing in to the memory for the
27       * other process to read */
28      s = shm;
29      for (c = 'a'; c <= 'z'; c++)
30        *s++ = c;
31      *s = NULL;
32
33      /* We wait until the process acknowledge by
34       * changing the first character of the memory */
35      while (*shm != '*')
36        sleep(1);
37
38      //TODO: Implement codeto remove the memory segment using shmctl()
39
40      return 0;
41  }
```

```
1   /* Source code shm_client.c */
2   #include <stdio.h>
3   #include <sys/types.h>
4   #include <sys/ipc.h>
5   #include <sys/shm.h>
6
7   int main(int argv, char* argn)
8   {
9       int shmid;
10      char c;
11      char *shm, *s;
12
13      /* Use ftok to generate a key associated with a file */
14      key_t key = ftok("/tmp/shm",'a');
15
16      /* Create/Locate the memory segment */
17      // TODO: add code to create memory segment using shmget()
18      //       size is fixed to 128
```

```
19      //        key is provided by key_t key
20
21      /* Attach the memory segment to our address space */
22      // TODO: add code to attach memory segment using shmat()
23      //        the return address is assign to pointer *shm
24
25      /* Read whatever data put in the memory*/
26      for (s = shm; *s != NULL; s++)
27        printf("%c\n", *s);
28
29      /* Acknowledge the read is completed */
30      *shm = '*';
31
32      return 0;
33  }
```

Biên dịch và thực thi với file hiện thực mã nguồn chương trình được đặt tên là `shm_client.c` và `shm_server.c`

```
$ gcc -o vd2_server shm_server.c

$ gcc -o vd2_client shm_client.c

$ ./vd2_server

$ ./vd2_client
a
b
c
...
z
```

Chương 4

Giao tiếp giữa các quá trình - Message queue

4.1 Giới thiệu Message queue

Message queue cho phép các process trao đổi dữ liệu dưới dạng các thông điệp (message). Message queue được hởi tạo bởi lệnh `msgget()`. Các thông điệp được truyền tới và lấy khỏi hàng đợi (queue) bởi lệnh msgsnd() và msgrcv().
Ngoài API lập trình, các message queue trong hệ thống có thể được liệt kệ trong terminal dùng lệnh ipcs và xóa bỏ một message queue dùng lệnh ipcrm. Chi tiết tham khảo phần phụ lục A
Các thao tác:

`msgget()` Khởi tạo;

`msgctl()` Xóa bỏ hoặc thay đổi thuộc tính;

`msgsnd()` Gửi thông điệp;

`msgrcv()` Nhận thông điệp.

4.2 Cách sử dụng Message queue

Khởi tạo và xóa Message queue sử dụng lệnh msgget() để khởi tạo message queue, lệnh msgctl() để tùy chỉnh(/hoặc xóa) message queue.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget ( key_t key, int msgflg );
 RETURNS: message queue identifier on success;
```

```
−1 on erorr: errno =  EACCESS (permission denied)
                      EEXIST (Queue exists, cannot create)
                      EIDRM (Queue is marked for deletion)
                      ENOENT (Queue does not exist)
                      ENOMEM (Not enough memory to create queue)
                      ENOSPC (Maximum queue limit exceeded)
```

Return value Trả về 0 nếu thực thi lệnh thành công, trả về -1 nếu có lỗi xảy ra

`key` là giá trị key value để so sánh với các key có trong kernel cho các message queue. Có thể dùng hàm ftok(const char *pathname, int proj_id); để chuyển đường dẫn một file và project identifier thành IPC key.

`msgflg` IPC_CREAT để tạo message queue mới. IPC_EXCL dùng cùng IPC_CREAT để thiết lập báo lỗi khi message queue đã tồn tại.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl ( int msgqid, int cmd, struct msqid_ds *buf );
  RETURNS: 0 on success
          −1 on error
```

Return value Trả về 0 nếu thực thi lệnh thành công, trả về -1 nếu có lỗi xảy ra

`msgid` message queue ID trả về từ `msgget()`

`cmd` IPC_STAT, IPC_SET để lấy/chỉnh sửa thiết lập shared memory hoặc IPC_RMID để xóa bỏ vùng shared memory.

`buf` cấu trúc dữ liệu để quản lý thiết lập của message queue, được khai báo trong `linux/msg.h`. Trong trường hợp các thiết lập không đổi, tham số có thể được truyền vào là NULL (giá trị 0).

Gửi và nhận message sau khi đã tạo và có queue id, chúng ta sử dụng các hàmg `msgsnd()` và `msgrcv()` để gửi/nhận thông điệp.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
  RETURNS: 0 on success
          −1 on error
```

Return value Trả về 0 nếu thực thi lệnh thành công, trả về -1 nếu có lỗi xảy ra

`msqid` message queue ID trả về từ `msgget()`

`msgp` là con trỏ tỏ đến cấu trúc được người gọi hàm định nghĩa trước theo định dạng tổng quát sau:

```
struct msgbuf{
 long mtype; /* Message type, indicate the kind (category)
                of the message, must be > 0      */
 char mtext[1]; /* Message data          */
};
```

`msgsz` xác định độ dài của dữ liệu `mtext`

`msgflg` xác định thiết lập cho thông điệp được gửi, nếu cờ IPC_NOWAIT được thiết lập thì hàm khi không gửi được thông điệp do queue bị đầy sẽ trả về ngay mã lỗi thay vì chờ đến khi queue có chỗ trống.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

size_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,
              int msgflg);
  RETURNS:  Number of bytes copied into message buffer
            -1 on error
```

Return value Trả về 0 nếu thực thi lệnh thành công, trả về -1 nếu có lỗi xảy ra

`msqid` message queue ID trả về từ `msgget()`

`msgp` là con trỏ tỏ đến cấu trúc được người gọi hàm định nghĩa trước theo định dạng tổng quát sau:

```
struct msgbuf{
  long mtype;    /* Message type, indicate the kind
                    (category) of the message, must be > 0 */
  char mtext[1];/* Message data */
};
```

`msgsz` xác định độ dài của dữ liệu `mtext`

`msgtyp` xác định loại thông điệp sẽ được nhận

- Nếu msgtyp = 0, thông điệp đầu tiên sẽ được nhận.
- Nếu msgtyp > 0, chỉ nhận các thông điệp có kiểu mtype=msgtyp được nhận

– Nếu msgtyp < 0, thông điệp đầu tiên có kiểu mtype có giá trị nhỏ nhất gần với giá trị tuyệt đối của msgtyp được nhận.

`msgflg` xác định thiết lập cho thông điệp được gửi, nếu cờ IPC_NOWAIT được thiết lập thì hàm khi không gửi được thông điệp do queue bị đầy sẽ trả về ngay mã lỗi thay vì chờ đến khi queue có chỗ trống.

4.3 Ví dụ sử dụng Message queue

```
1  #include <sys/types.h>
2  #include <sys/ipc.h>
3  #include <sys/msg.h>
4
5  #define MSG_QUEUE_PATH "/tmp/OSlab_msg_queue"
6  #define MAX_MSG_LEN     256
7
8  struct msgbuf{
9     long mtype; /* Message type, indicate the kind
10                   (category) of the message, must be > 0 */
11    char mtext[MAX_MSG_LEN]; /* Message data */
12 };
13
14 int send_message(int qid,        /* Message queue ID */
15                  long msg_type, /* Message type, must be > 0 */
16                  char *text)    /* Message data  */
17 {
18    struct msgbuf msg_buffer;
19
20    // Do something to validate input data
21
22    /* Send a message to queue */
23    printf("Sendng a message...\n");
24
25    msg_buffer.mtype = msg_type;
26    strcpy((char*) msg_buffer.mtext, text);
27
28   if((msgsnd(qid, (struct msgbuf *) &msg_buffer,
29              strlen(msg_buffer.mtext)+1, 0)) == -1)
30    {
31       fprintf(stderr,"msgsnd\n");
32    }
33
34 }
35
36 int read_and_echo_message(int qid) /* Message queue ID */
```

```
37 {
38     struct msgbuf msg_buffer;
39     int ret = 0;
40 
41     /* Send a message to queue */
42     printf("Reading a message...\n");
43 
44    if((msgrcv(qid, (struct msgbuf *) &msg_buffer,
45                MAX_MSG_LEN, 0, 0)) == -1)
46     {
47         perror("msgrcv - empty buffer");
48     }
49 
50     // Do something else to display the received data
51     // Ex: printf("Type: %ld Text: %s\n",
52     //              msg_buffer.mtype, msg_buffer.mtext);
53 }
54 
55 int main(int argc, char* argv[])
56 {
57     /* Use ftok to generate a key associated with a file */
58     key_t key = ftok(MSG_QUEUE_PATH, 'a');
59 
60     /* Create/locate the message queue */
61     if((msgqueue_id=msgget(key, IPC_CREAT | IPC_EXCL | 0666)) < 0) {
62         perror("msgget - msgsrv must be called once");
63         return -1;
64     }
65 
66     /* Send message got from parsing program argument
67         USAGE msgtool (s)end <messagetext>                */
68     send_message(msgqueue_id, 1, argv[2]);
69 
70     read_and_echo_message(msgqueue_id)
71 
72     /* Remove the queue identified by qid */
73     msgctl(qid, IPC_RMID,0);
74 }
```

Chương 5

Bài tập

Cấu hình các tập tin `shm_server.c`3.3.2 và `shm_client.c`3.3.2 phù hợp với hệ thống của sinh viên và tiến hành biên dịch, chạy chương trình.

Phụ lục A

IPC System-V Objects

A.1 Giới thiệu System V API

System V IPC (message queue, semaphore, và shared memory) được AT&T giới thiệu trong lúc cộng đồng POSIX chưa hoàn chỉnh tiêu chuẩn hóa các giao tiếp này. Hầu hết các hiện thực hỗ trợ chuẩn System-V. GNU/Linux tương thích một phần với POSIX và nhiều nhà cung cấp sẽ hướng đến việc hỗ trợ POSIX.

- SystemV-style shared memory segments
- SystemV-style message queue
- SystemV-style semaphore sets

A.2 IPC Identifier

IPC object mỗi đối tượng IPC có một định danh tương ứng. Các đối tượng IPC là message queue, semaphore set hoặc shared memory segment. Để có thể tạo ra một định danh duy nhất, cần thiết phải cung cấp một `key`. Key này phải được thống nhất giữa server và client. Để tránh việc sử dụng lặp lại các key, hàm ftok() được dùng để sinh ra các giá trị dùng chung cho server và client.

Bảng A.1.1: Summary of programming interfaces for System IPC objects

Interface	Message queues	Semaphores	Shared memory
Header file	`<sys/msg.h>`	`<sys/sem.h>`	`<sys/shm.h>`
Associated data structure	msqid_ds	semid_ds	shmid_ds
Create/open object	msgget()	semget()	shmget() + shmat()
Close object	N/A	N/A	shmdt()
Control operations	msgctl()	semctl()	shmctl()
Performing IPC	msgsnd()-write message msgrcv()-read message	semop() - test/adjust semaphore	access memory in shared region

```
LIBRARY FUNCTION: ftok();

PROTOTYPE: key_t ftok ( char *pathname, char proj );
  RETURNS: new IPC key value if successful
           -1 if unsuccessful, errno set to return of stat() call
```

A.3 Thông tin các IPC

Lệnh ipcs dùng để liệt kê thông tin về các IPC mà process có quyền truy xuất.

```
ipcs      -q:    Show only message queues
ipcs      -s:    Show only semaphores
ipcs      -m:    Show only shared memory
ipcs      -a:    Show all
ipcs --help:     Additional arguments
```

```
# ipcs -a
------ Shared Memory Segments --------
key        shmid   owner  perms bytes nattch   status
0xc616cc44 1568768 tc     66    4096  0

------ Semaphore Arrays --------
key        semid   owner  perms nsems
0x4b0d4514 14418   tc     660   204

------ Message Queues --------
key        msqid   owner  perms used-bytes   messages
0x000005a4 32768   root   644   0            0
```

A.4 Xóa bỏ IPC

ipcrm được dùng để xóa một đối tượng IPC ra khỏi hệ thống.

```
ipcrm <msg | sem | shm>  <IPC ID>
```

Phụ lục B

Giao tiếp giữa các quá trình - SIGNALs

B.1 Signal

là cơ chế để giao tiếp và thao tác trên các process (quá trình) trong môi trường Linux. Signal là một thông điệp đặc biệt được gửi đến process. Có nhiều loại thông điệp như vậy và mỗi thông điệp có ý nghĩa riêng tương ứng với một hành vi khác nhau của process. Khi một process nhật một signal, nó có thể thực hiện các thao tác khác nhau phụ thuộc vào cách cài đặt xử lý signal.
Mỗi signal sẽ có một cài đặt mặc định, cách này sẽ định nghĩa hành vi của process trong trường hợp process không có một định nghĩa hành vi nào khác. Một chương trình định nghĩa hành vi riêng cho một signal được gọi là `signal-handler`. Cần lưu ý là trong khi *signal-handler* được gọi, chương trình đang thực thi sẽ bị tạm dừng; khi *signal-handler* thực thi xong, chương trình mới được bắt đầu trở lại. Signals có nhiều ứng dụng, trong phạm vi bài này chúng ta chỉ xem xét các signal quan trọng và kĩ thuật để quản lý process.

Hàm sigaction là hàm được sử dụng để cài đặt xử lý signal. Tham số đầu tiên là `signal number`. Hai tham số tiếp theo là con trỏ đến cấu trúc `sigaction`. Thành phần đầu tiên của cấu trúc này là cài đặt xử lý `action` dự định cho signal number, thành phần thứ hai nhận kết quả trả về của `old_action` bố trí trước đó. Trong cấu trúc `sigaction`, trường quan trọng nhất là `sa_handler`. Trường này có thể nhận một trong 3 giá trị

- `SIG_DFL` thiết lập cách bố trí mặc định
- `SIG_IGN` thiết lập signal bị bỏ qua
- một con trỏ tới hàm *signal-handler*. Hàm này phải được khai báo nhận 1 tham số là signal number và kết quả trả về là `void`.

```
int sigaction(int signum, const struct sigaction *act,
                          struct sigaction *oldact);
```

Một số lưu ý (i) Signal là không đồng bộ, do vậy chương trình chính có thể ở trạng thái không ổn định khi một signal được xử lý và `singal handler` đang thực thi. Do đó, không nên có các thao tác I/O hoặc lời gọi hàm hệ thống từ trong `signal handler`. (ii) `Signal handler` chỉ nên thực hiện một lượng công việc nhỏ để đáp ứng signal và trả điều khiển về chương trình chính. Với đặc thù như vậy, thường thao tác này chỉ ghi nhận lại signal đã xảy ra rồi sau đó chương trình chính sẽ kiểm tra định kỳ xem có signal xảy ra và tiến hành các hành vi tương ứng. (iii) Một điểm cần lưu ý nữa là một `signal handler` có thể bị ngắt (interrupt) bởi một signal khác. Trường hợp này rất ít khi xảy ra nhưng nếu nó xảy ra sẽ rất khó gỡ lỗi (debug). Do vậy, người lập trình cần kiểm tra rất kỹ về các thao tác trong `signal handler`.

```
1  #include <signal.h>
2  #include <sys/types.h>
3
4  #include <stdio.h>
5
6  sig_atomic_t sigusr1_count = 0;
7
8  void handler (int signal_number)
9  {
10   ++sigusr1_count;
11 }
12
13 int  main(int argv, char** argn)
14 {
15   struct sigaction sa;
16
17   memset(&sa, 0, sizeof (sa));
18   sa.sa_handler = &handler;
19   sigaction (SIGUSR1, &sa, NULL);
20
21   /* Main program goes here*/
22   /* .... */
23   /*
24      Example code:
25      for (int i=0; i<10; i++)
26        sleep(1);
27   */
28
29   printf("SIGUSR1 was raised % times", sigusr1_count);
30   return 0;
```

```
31 }
```

B.2 Send signal

để gửi signal SIGUSR1 đến process của chương trình, chúng ta có thể sử dụng hàm `kill` trong bash.

```
$ kill -SIGUSR1 `pidof program_name`
```

Phụ lục C

Giao tiếp giữa các quá trình - Pipe

C.1 Giới thiệu pipe

là quá trình mà *output* của một `process` làm *input* cho một process khác. Ví dụ này chúng ta đã gặp trong quá trình làm việc với môi trường Bash script sử dụng ký tự piper `|`. UNIX cho phép sử dụng hai cách để tạo một `pipe`.

popen() **Formatted piping**

```
FILE *popen(char *command, char *type)
```

tạo một pipe giao tiếp I/O để kết nối với `process command` và tạo ra `pipe`. Tham số `type` là "r" ở chế độ `read` và "w" ở chế độ `write`.

- Hàm trả về NULL nếu bị lỗi hoặc một `stream pointer`, có thể được đóng bằng lệnh `pclose(FILE *stream)`
- Hàm `fprintf()` và `fscanf()` để giao tiếp với `stream` này.

pipe() **Low level piping**

```
SYSTEM CALL: pipe();

PROTOTYPE: int pipe( int fd[2] );
  RETURNS: 0 on success
          -1 on error: errno = EMFILE (no free descriptors)
                               EMFILE (system file table is full)
                               EFAULT (fd array is not valid)
 NOTES: fd[0] is set up for reading, fd[1] is set up for writing
```

tạo ra `pipe` với hai `file descriptor`: `fd[0]` để reading và `fd[1]` để writing.

– `Pipe` được tạo có thể được đóng bởi hàm `close(int fd)`. Sinh viên cần để ý trường hợp này tham số đầu vào của hàm close nhận một `int`.

– Các hàm tương tác với pipe được tạo là `read()` và `write()`.

Khi sử dụng system call này, sinh viên cần hết sức lưu ý process child khi gọi `fork()` có thể thực thi `exec()` một chương trình khác , vốn có thể được thừa kế các stream. Trong trường hợp đó chúng ta có thể sử dụng các hàm `dup()` và `dup2()` để xử lý. Trường hợp này là một gợi ý sinh viên tự tìm hiểu thêm.

C.2 popen-Formatted piping

Sinh viên xem xét các ví dụ tạo pipe()

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4
5  int main(void)
6  {
7          int     fd[2], nbytes;
8          pid_t   childpid;
9          char    string[] = "Hello, world!\n";
10         char    readbuffer[80];
11
12         pipe(fd);
13
14         if((childpid = fork()) == -1)
15         {
16                 perror("fork");
17                 exit(1);
18         }
19
20         if(childpid == 0)
21         {
22                 /* Child process closes up input side of pipe */
23                 close(fd[0]);
24
25                 /* Send "string" through the output side of pipe */
26                 write(fd[1], string, (strlen(string)+1));
27                 exit(0);
28         }
29         else
30         {
31                 /* Parent process closes up output side of pipe */
```

```
32                      close(fd[1]);
33
34                      /* Read in a string from the pipe */
35                      nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
36                      printf("Received string: %s", readbuffer);
37              }
38
39              return(0);
40      }
```

C.3 pipe-Low level piping

Sau khi tham khảo ví dụ sử dụng Lowlevel pipe thông qua lời gọi các system call, sinh viên có thể tham khảo thêm một thay thế khác theo hướng dễ sử dụng hơn. Thư viện hàm chuẩn được thay thế là hàm `popen()` cung cấp bên trong một lời gọi tạo một pipe half-duplex. Cần lưu ý là các pipe được tạo bằng hàm `popen()` phải được đóng lại bằng hàm `pclose()`. Xem xét ví dụ minh họa cách sử dụng hàm thư viện này. Sinh viên xem xét các ví dụ tạo pipe()

```
1   #include <stdio.h>
2
3   int main(int argc, char *argv[])
4   {
5           FILE *pipe_fp, *infile;
6           char readbuf[80];
7
8           if( argc != 3) {
9                   fprintf(stderr, "USAGE:  popen_ex [command] [filename]\n");
10                  exit(1);
11          }
12
13          /* Open up input file */
14          if (( infile = fopen(argv[2], "rt")) == NULL)
15          {
16                  perror("fopen");
17                  exit(1);
18          }
19
20          /* Create one way pipe line with call to popen() */
21          if (( pipe_fp = popen(argv[1], "w")) == NULL)
22          {
23                  perror("popen");
24                  exit(1);
25          }
26
```

```
27          /* Processing loop */
28          do {
29                  fgets(readbuf, 80, infile);
30                  if(feof(infile)) break;
31
32                  fputs(readbuf, pipe_fp);
33          } while(!feof(infile));
34
35          fclose(infile);
36          pclose(pipe_fp);
37
38          return(0);
39  }
```

Phụ lục D

Giao tiếp giữa các quá trình - Socket

D.1 Giới thiệu socket

`Sockets` cho phép giao tiếp point-to-point, 2 chiều giữa hai process. `Socket` được dùng rất phổ biến và là thành phần cơ bản trong giao tiếp giữa các quá trình và giao tiếp giữa các hệ thống. Khái niệm `socket`, tồn tại trong lĩnh vực thông tin liên lạc, là một khái niệm trừu tượng cung cấp cấu trúc định địa chỉ cho một tập hợp các giao thức. Socket có thể được sử dụng trên môi trường Internet và cả giao tiếp giữa các quá trình trên một hệ thống riêng lẻ.
Môi trường UNIX cung cấp một không gian địa chỉ socket trên mỗi hệ thống đơn lẻ. Socket có thể được dùng để giao tiếp giữa các quá trình trên các hệ thống khác nhau. Có thể hình dung không gian kết nối các socket giữa các hệ thống được gọi là "Internet". Việc giao tiếp trong không gian "Internet" sử dụng bộ giao thức internet TCP/IP. Khái niệm địa chỉ `socket address` là sự kết hợp giữa địa chỉ IP `IP address` và `socket number`. Các khái niệm này hiện tại được sử dụng khá phổ biến nên sinh viên có thể tự tìm hiểu.
Các dạng socket `socket type` định nghĩa thuộc tính giao tiếp mà ứng dụng nhìn thấy. Các process giao tiếp chỉ giữa các socket cùng loại. Trong môi trường Internet, các dạng socket sau đây, thường là dạng hướng dữ liệu (đơn vị giao tiếp truyền nhận là các gói dữ liệu), được dùng phổ biến:

Stream socket cung cấp luồng dữ liệu hai chiều, tuần tự và không trùng lặp. Một `stream` có thể hình dung giống cuộc trò chuyện qua điện thoại. Dạng socket này sử dụng hằng gợi nhớ `SOCKET_STREAM` và trong môi trường Internet sử dụng giao thức Tranmission Control Protocol (TCP).

Datagram socket hỗ trợ luồng dữ liệu hai chiều. Dữ liệu được nhận có thể không theo đúng thứ tự mà nó được gửi đi. Mỗi đơn vị dữ liệu gửi và nhận trong dạng socket này được xử lý và định tuyến đường đi độc lập, đây là cách giao

tiếp không hướng kết nối. Trong môi trường Internet, dạng socket này sử dụng giao thức User Datagram Protocol (UDP).

Raw socket thường được sử dụng giao tiếp giữa các bộ định tuyến (`router`) và các thiết bị mạng khác. Dạng giao tiếp này cung cấp cách thức để truy xuất các giao thức giao tiếp bên dưới.

Tạo và naming socket Hàm đầu tiên `socket()` để tạo socket ở domain và dạng xác định trước. Nếu tham số protocol không được gán, giá trị mặc định của hệ thống cho dạng socket `socket type` được sử dụng. Giá trị trả về của hàm là `socket handler`. Trong môi trường UNIX trên hệ thống đơn lẻ, kết nối thường được tạo ra giữa hai tên đường dẫn `path name`. Trong môi trường Internet, kết nối được tạo ra giữa hai địa chỉ kết hợp bởi `internet address` và `socket number`. Cần lưu ý trong hầu hết các môi trường, các kết nối phải là duy nhất.
Hàm thứ hai `bind()` để liên kết một đường dẫn `path` hay `internet address` với một `socket`. Thao tác như vậy được gọi là `naming` socket, và socket sau khi được naming gọi là `named socket`. Sau khi đã hiểu, sinh viên cần ghi nhớ các thuật ngữ `bind`, `naming` có thể được sử dụng thay thế lẫn nhau.
Sau khi liên kết, các hàm `unlink()` hay `rm()` có thể được sử dụng để xóa một socket.

```
int socket(int domain, int type, int protocol)

int bind(int s, const struct sockaddr *name, int namelen)
```

D.2 Stream socket

Kết nối stream socket việc kết nối socket thường được thực hiện không đối xứng, nghĩa là một process thường đóng vai trò `server` và process còn lại đóng vai trò `client`. Trong đó, `server` thường liên kết "`bind`" socket của mình với một đường dẫn `path` hoặc địa chỉ `address` được thỏa thuận trước. Trong trường hợp `SOCK_STREAM` socket, server sẽ gọi hàm `int listen(int s, int backlog)`. Một `client` khởi tạo kết nối đến server socket bằng cách gọi hàm `int connect(int s, struct sockaddr *name, int namelen)`.
Nếu client socket chưa được `bind` tại thời điểm kết nối, hệ thống sẽ tự động chọn và bind một "name" cho socket đó. Với `SOCK_STREAM` socket, `server` gọi `accept()` để hoàn tất kết nối.
Hàm `int accept(int s, struct sockaddr *addr, int *addrlen)` trả về một `socket description` chỉ có hiệu lực với kết nối cụ thể đó. Một `server` có thể có nhiều connection `SOCK_STREAM` cùng lúc tại một thời điểm.

Truyền dữ liệu `stream data` và đóng kết nối Có nhiều hàm được sử dụng để gửi và nhận dữ liệu từ `SOCK_STREAM` socket. Các hàm có thể là `write()`,`read()`, `int send(int s, const char *msg, int len, int flags)`, `int recv(int s,`

`char *buf, int len, int flags)`. Việc đóng `SOCK_STREAM` socket có thể được thực hiện bằng cách gọi hàm `close()`.

D.3 Datagram socket

`Datagram socket` không cần thiết lập kết nối. Mỗi thông điệp sẽ mang theo địa chỉ đích. Các hàm để gửi dữ liệu được sử dụng là `sendto()` hoặc `sendmsg()`. Các hàm để nhận dữ liệu được sử dụng là `recvfrom()` hoặc `recvmsg()`.
Datagram socket có thể sử dụng hàm `connect()` để xác định socket đích (`destination socket`). Sau khi connect, các hàm `send()` và `recv()` cũng được sử dụng để gửi và nận dữ liệu. Còn các hàm `accept()` và `listen()` không dùng với datagram socket.
Xem xét ví dụ sau minh họa việc thiết kết nối socket server và socket client:

SOCKET_SERVER.C hiện thực thành phần `server`

D.4 Kết nối socket giữa server và client

```
1  #include <sys/types.h>
2  #include <sys/socket.h>
3  #include <sys/un.h>
4  #include <stdio.h>
5
6  #define NSTRS       3           /* no. of strings  */
7  #define ADDRESS     "mysocket"  /* addr to connect */
8
9  /*
10  * Strings we send to the client.
11  */
12 char *strs[NSTRS] = {
13     "This is the first string from the server.\n",
14     "This is the second string from the server.\n",
15     "This is the third string from the server.\n"
16 };
17
18 int main(int argv, char* argn[])
19 {
20     char c;
21     FILE *fp;
22     int fromlen;
23     register int i, s, ns, len;
24     struct sockaddr_un saun, fsaun;
25
26     /*
```

```
27      * Get a socket to work with.  This socket will
28      * be in the UNIX domain, and will be a
29      * stream socket.
30      */
31     if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
32         perror("server: socket");
33         return 1;
34     }
35
36     /*
37      * Create the address we will be binding to.
38      */
39     saun.sun_family = AF_UNIX;
40     strcpy(saun.sun_path, ADDRESS);
41
42     /*
43      * Try to bind the address to the socket.  We
44      * unlink the name first so that the bind won't
45      * fail.
46      *
47      * The third argument indicates the "length" of
48      * the structure, not just the length of the
49      * socket name.
50      */
51     unlink(ADDRESS);
52     len = sizeof(saun.sun_family) + strlen(saun.sun_path);
53
54     if (bind(s, (struct sockaddr*)&saun, len) < 0) {
55         perror("server: bind");
56         return 1;
57     }
58
59     /*
60      * Listen on the socket.
61      */
62     if (listen(s, 5) < 0) {
63         perror("server: listen");
64         return 1;
65     }
66
67     /*
68      * Accept connections.  When we accept one, ns
69      * will be connected to the client.  fsaun will
70      * contain the address of the client.
71      */
72     if ((ns = accept(s, NULL, NULL)) < 0) {
```

```
73          perror("server: accept");
74          return 1;
75      }
76
77      /*
78       * We'll use stdio for reading the socket.
79       */
80      fp = fdopen(ns, "r");
81
82      /*
83       * First we send some strings to the client.
84       */
85      for (i = 0; i < NSTRS; i++)
86          send(ns, strs[i], strlen(strs[i]), 0);
87
88      /*
89       * Then we read some strings from the client and
90       * print them out.
91       */
92      for (i = 0; i < NSTRS; i++) {
93          while ((c = fgetc(fp)) != EOF) {
94              putchar(c);
95
96              if (c == '\n')
97                  break;
98          }
99      }
100
101     /*
102      * We can simply use close() to terminate the
103      * connection, since we're done with both sides.
104      */
105     close(s);
106
107     return 0;
108 }
```

SOCKET_CLIENT.C hiện thực thành phần `client`

```
1  #include <sys/types.h>
2  #include <sys/socket.h>
3  #include <sys/un.h>
4  #include <stdio.h>
5
6  #define NSTRS       3           /* no. of strings  */
```

```
7  #define ADDRESS     "mysocket"  /* addr to connect */
8
9  /*
10  * Strings we send to the server.
11  */
12 char *strs[NSTRS] = {
13     "This is the first string from the client.\n",
14     "This is the second string from the client.\n",
15     "This is the third string from the client.\n"
16 };
17
18 int main(int argv, char* argn[])
19 {
20     char c;
21     FILE *fp;
22     register int i, s, len;
23     struct sockaddr_un saun;
24
25     /*
26      * Get a socket to work with.  This socket will
27      * be in the UNIX domain, and will be a
28      * stream socket.
29      */
30     if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
31         perror("client: socket");
32         return 1;
33     }
34
35     /*
36      * Create the address we will be connecting to.
37      */
38     saun.sun_family = AF_UNIX;
39     strcpy(saun.sun_path, ADDRESS);
40
41     /*
42      * Try to connect to the address.  For this to
43      * succeed, the server must already have bound
44      * this address, and must have issued a listen()
45      * request.
46      *
47      * The third argument indicates the "length" of
48      * the structure, not just the length of the
49      * socket name.
50      */
51     len = sizeof(saun.sun_family) + strlen(saun.sun_path);
52
```

```
53      if (connect(s, (struct sockaddr*)&saun, len) < 0) {
54          perror("client: connect");
55          return 1;
56      }
57
58      /*
59       * We'll use stdio for reading
60       * the socket.
61       */
62      fp = fdopen(s, "r");
63
64      /*
65       * First we read some strings from the server
66       * and print them out.
67       */
68      for (i = 0; i < NSTRS; i++) {
69          while ((c = fgetc(fp)) != EOF) {
70              putchar(c);
71
72              if (c == '\n')
73                  break;
74          }
75      }
76
77      /*
78       * Now we send some strings to the server.
79       */
80      for (i = 0; i < NSTRS; i++)
81          send(s, strs[i], strlen(strs[i]), 0);
82
83      /*
84       * We can simply use close() to terminate the
85       * connection, since we're done with both sides.
86       */
87      close(s);
88
89      return 0;
90  }
```

Revision History

Revision	Date	Author(s)	Description
1.0	25.04.15	PD Nguyen	created
2.0	27.02.16	PD Nguyen	restruct the lab