ĐẠI HỌC QUỐC GIA TP HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

Tài liệu hướng dẫn thực hành

HỆ ĐIỀU HÀNH

Biên soạn: ThS Phan Đình Duy

ThS Nguyễn Thanh Thiện

KS Trần Đại Dương

ThS Trần Hoàng Lộc

KS Thân Thế Tùng

MỤC LỤC

NỘI QUY THỰC HÀNH

1. Sinh viên tham dự đầy đủ các buổi thực hành theo quy định của giảng viên hướng dẫn (GVHD) (6 buổi với lớp thực hành cách tuần hoặc 10 buổi với lớp thực hành liên tục).
2. Sinh viên phải chuẩn bị các nội dung trong phần "Sinh viên viên chuẩn bị" trước khi đến lớp. GVHD sẽ kiểm tra bài chuẩn bị của sinh viên trong 15 phút đầu của buổi học (nếu không có bài chuẩn bị thì sinh viên bị tính vắng buổi thực hành đó).
3. Sinh viên làm các bài tập ôn tập để được cộng điểm thực hành, bài tập ôn tập sẽ được GVHD kiểm tra khi sinh viên có yêu cầu trong buổi học liền sau bài thực hành đó. Điểm cộng tối đa không quá 2 điểm cho mỗi bài thực hành.

Bài 5. LÀM VIỆC VỚI TIỂU TRÌNH VÀ ĐỒNG BỘ HÓA TIỂU TRÌNH

5.1 Mục tiêu

- Hướng dẫn sinh viên các thao tác làm việc với tiểu trình
- Giới thiệu đến sinh viên 2 thư viện Semaphore và thư viện Mutex dùng để thực hiện việc đồng bộ hóa tiến trình, tiểu trình.
- Sinh viên thực hiện và hiểu được tầm quan trọng của việc đồng bộ hóa tiến trình, tiểu trình.

5.2 Nội dung thực hành

- Viết chương trình đa tiểu trình
- Viết chương trình áp dụng các kỹ thuật đồng bộ sử dụng semaphore và mutex.

5.3 Sinh viên chuẩn bị

Để thực hiện bài thực hành này, sinh viên phải đảm bảo những điều sau:

- Đã cài đặt C compiler cho hệ điều hành Linux.
- Biết cách viết, build và chạy một chương trình trên hệ điều hành Linux.

5.4 Sinh viên thực hành

5.4.1 Tiểu trình

5.4.1.1 Khái niệm

Tiểu trình là các luồng điều khiển riêng biệt, thường là trong một chương trình (hoặc trong nhân Hệ điều hành). Nhiều tiểu trình cùng chia sẻ không gian địa chỉ và các tài nguyên khác, nhờ thế chúng có nhiều ưu điểm như:

- Truyền thông tốc độ cao giữa các tiểu trình
- Chuyển đổi ngữ cảnh nhanh giữa các tiểu trình
- Được sử dụng nhiều trong các chương trình yêu cầu xử lý lớn
- Một chương trình có thể sử dụng nhiều CPU cùng một lúc (lập trình song song)

Nhờ những ưu điểm của nó, tiểu trình ngày nay trở thành mức trừu tượng lập trình hiện đại và phổ biến. Nhiều tiểu trình (đa tiểu trình – đa luồng) cùng thực thi trong một chương trình trong một không gian địa chỉ (chia sẻ bộ nhớ). Chúng cũng có thể chia sẻ việc mở tệp tin và sử dụng chung các tài nguyên khác. Tiểu trình đang trở thành mức lập trình song song chủ yếu trong các hệ thống đa bộ xử lý.

Nhưng chính vì do các tiểu trình cùng chia sẻ tài nguyên nên có một vấn đề cần phải giải quyết đó là sự tranh chấp tài nguyên giữa các tiểu trình, đòi hỏi nhiều nỗ lực đồng bộ hóa tiểu trình để thực thi sao cho hiệu quả.

5.4.1.2 Tiểu trình trong Linux

Trong nhân Hệ điều hành Linux, tiểu trình được hiện thực như tiến trình, tiểu trình đơn thuần là tiến trình mà có thể chia sẻ một số tài nguyên nhất định với các tiến trình khác. Đối với một số Hệ điều hành khác, ví dụ như MS Windows, tiểu trình và tiến trình đều là các khái niệm riêng biệt và được hỗ trợ đầy đủ.

Trong bài thực hành này POSIX thread (pthread) sẽ được sử dụng để lập trình tiểu trình. Nó cho phép chúng ta tạo ra các ứng dụng chạy song song theo luồng, phù hợp với các hệ thống đa bộ xử lý. POSIX là viết tắt của Portable Operating Systems Interface là mô tả các API (Application Programming Interface) bao gồm hàm và chức năng của chúng.

Các thao tác của tiểu trình bao gồm: tạo tiểu trình, đồng bộ tiểu trình (hợp – join, khóa – blocking), lập lịch, quản lý dữ liệu và tương tác giữa các tiểu trình.

Mỗi tiểu trình là độc lập với nhau, nghĩa là nó không biết hệ thống có bao nhiêu tiểu trình và nó được sinh ra từ đâu.

Các tiểu trình trong cùng một chương trình chia sẻ không gian địa chỉ, PC, dữ liệu, tập tin, signal, user ID, group ID. Nhưng chúng

cũng có những tài nguyên riêng của chúng, bao gồm: ID của tiểu trình, các thanh ghi, ngăn xếp, signal mask, độ ưu tiên.

5.4.1.3 Tạo tiểu trình

Để tạo tiểu trình, sử dụng hàm pthread_create() như bên dưới:

```
int pthread_create(pthread_t * thread, pthread_attr_t *
attr, void * (*start_routine)(void *), void * arg);
```

Trong đó:

- thread là biến tham chiếu kiểu pthread_t được dùng như PID
- attr là biến tham chiếu kiểu pthread_attrt thể hiện thuộc tính của thread (dùng NULL nếu đặt thuộc tính mặc định)
- *start_rouine là một con trỏ hàm kiểu void đến chức năng mong muốn tiểu trình thực thi
- *arg là con trỏ đối số cho hàm kiểu void

Nếu tiểu trình được tạo thành công, hàm pthread_create() sẽ trả về số nguyên 0, ngược lại sẽ là một số khác 0.

Dùng một công cụ soạn thảo văn bản để soạn và dùng gcc với cờ -pthread để biên dịch chương trình sau. Chương trình sẽ in ra vô hạn dòng chữ: “Hello, How are you?” và “I’m fine, and you?”

```
/*#####################################
# University of Information Technology            #
# IT007 Operating System                          #
# <Your name>, <your Student ID>                  #
# File: example_thread_creation.c                 #
#####################################*/

#include <pthread.h>
#include <stdio.h>
void *thread_print(void * messenge) {
  while(1) {
        printf("Hello, How are you?\n");
    }
}

int main() {
   pthread_t idthread;
   pthread_create(
                              &idthread,
                              NULL,
                              &thread_print,
                              NULL);
   while(1) {
        printf("I'm fine, and you?\n");
   }
   return 0;
}
```

Trong đó, idthread là tiểu trình sẽ in ra “Hello, How are you?”, main là tiểu trình sẽ in ra “I’m fine, and you?”. Nhấn CRT+C để kết thúc.

5.4.1.4 Dừng tiểu trình

Để dừng một pthread có thể sử dụng hàm pthread_exit(), nếu hàm này được pthread gọi ngoài hàm main() thì nó sẽ dừng pthread gọi hàm này; nếu hàm này được gọi trong main() thì nó sẽ đợi các pthread trong nó dừng rồi nó mới dừng.

Dùng một công cụ soạn thảo văn bản để soạn và dùng gcc với cờ -pthread để biên dịch chương trình bên dưới:

```
/*#######################################
# University of Information Technology          #
# IT007 Operating System                        #
# <Your name>, <your Student ID>                #
# File: example_thread_selfexit.c               #
#######################################*/

#include <pthread.h>
#include <stdio.h>
#inlucde <stdlib.h>
#inlcude <unistd.h>

#define NUM_THREADS  2

void *thread_print(void *threadid)
{
   long tid;
```

```
   tid = (long)threadid;
   printf("Hello IT007! I'm Thread #%ld ^_^!!!\n", tid);
   pthread_exit(NULL);
}

int main()
{
   pthread_t threads[NUM_THREADS];
   int check;
   long tID;

   for(tID = 0; tID < NUM_THREADS; tID++){
     printf("I'm Main Thread: create Thread: #%ld\n",
tID);
     check = pthread_create(
                        &threads[tID],
                        NULL,
                        thread_print,
                        (void *)tID);
     if (check != 0){
       printf("ERROR!!! I'm Main Thread, can't create
Thread #%ld ", tID);
       exit(-1);
       }
     }
   sleep(100);
   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

Dùng lệnh top/ps để kiểm chứng các tiểu trình được tạo mới kết thúc trước khi tiểu trình main kết thúc (gợi ý: có thể điều chỉnh chương trình để lấy định danh của thread để tìm kiếm nhanh hơn). Tiếp tục biên soạn và dùng gcc với cờ -pthread để biên dịch chương trình bên dưới:

```
/*#######################################
# University of Information Technology            #
# IT007 Operating System                          #
# <Your name>, <your Student ID>                  #
# File: example_thread_mainexit.c                 #
#######################################*/

#include <pthread.h>
#include <stdio.h>
#inlucde <stdlib.h>
#inlcude <unistd.h>

#define NUM_THREADS  2

void *thread_print(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello IT007! I'm Thread #%ld ^_^!!!\n", tid);
   sleep(100);
}

int main()
{
```

```
   pthread_t threads[NUM_THREADS];
   int check;
   long tID;

   for(tID = 0; tID < NUM_THREADS; tID++){
     printf("I'm Main Thread: create Thread: #%ld\n",
tID);
     check = pthread_create(
                          &threads[tID],
                          NULL,
                          thread_print,
                          (void *)tID);
     if (check != 0){
       printf("ERROR!!! I'm Main Thread, I can't create
Thread #%ld ", tID);
       exit(-1);
       }
     }

   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

Dùng lệnh top/ps để kiểm chứng tiểu trình main kết thúc nhưng các tiểu trình được tạo mới vẫn chưa kết thúc, sau đó dùng lệnh kill để hủy các tiểu trình được tạo.

5.4.1.5 Hợp và gỡ tiểu trình

Để kết hợp các pthread, có thể sử dụng hàm pthread_join(threadid, status), pthread_join() sẽ ngưng pthread đang gọi tới khi threadid kết thúc. Khi threaded kết thúc, pthread_join() sẽ trả về giá trị 0.

Để tháo gỡ các pthread, có thể sử dụng hàm pthread_detach(threadid).

```
/*#######################################
# University of Information Technology            #
# IT007 Operating System                          #
# <Your name>, <your Student ID>                  #
# File: example_thread_join.c                     #
#######################################*/
#include <pthread.h>
#include <stdio.h>
#inlucde <stdlib.h>
#inlcude <unistd.h>

#define NUM_THREADS  2

void *thread_print(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello IT007! I'm Thread #%ld ^_^!!!\n", tid);
   sleep(100);
```

```
}

int main()
{
   pthread_t threads[NUM_THREADS];
   int check;
   long tID;

   for(tID = 0; tID < NUM_THREADS; tID++){
     printf("I'm Main Thread: create Thread: #%ld\n",
tID);
     check = pthread_create(
                          &threads[tID],
                          NULL,
                          thread_print,
                          (void *)tID);
         if (check != 0){
               printf("ERROR!!! I'm Main Thread,  I
can't create Thread #%ld ", tID);
                exit(-1);
         } //end if
         pthread_join(threads[tID], NULL);
   } //end for

   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

Khi có thêm pthread_join(), để có thể thực thi tiếp vòng lặp for thì threads[tID] phải kết thúc trước.

5.4.1.6 Truyền dữ liệu cho tiểu trình

Đối số cuối cùng của hàm pthread_create() là một con trỏ đối số cho thủ tục mà tiểu trình được tạo ra sẽ thực thi. Trong các ví dụ trước, đối số truyền vào là đơn kiểu dữ liệu, để có thể truyền nhiều đối số với đa dạng kiểu dữ liệu hơn thì chúng ta có thể sử dụng kiểu cấu trúc như bên dưới:

```
/*#####################################
# University of Information Technology            #
# IT007 Operating System                           #
# <Your name>, <your Student ID>                   #
# File: example_thread_structure.c                 #
#####################################*/
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS  2

struct struct_print_parms{
    char character;
    int count;
};

void* char_print (void* args) {
    struct struct_print_parms* p = (struct
struct_print_parms*) args;
    int i;
    for (i=0; I <p->count; i++)
        printf ("%c\n", p->character);
```

```
    return NULL;
}

int main () {
    pthread_t tid;
    struct struct_print_parms th_args;
    th_args.character = 'X';
    th_args.count = 5;
    pthread_create(&tid, NULL, &char_print, &th_args);
    pthread_join (tid, NULL);
    return 0;
}
```

5.4.2 Semaphore

Trong hệ điều hành, semaphore là 1 biến được sử dụng để điều khiển sự truy xuất vào các tài nguyên chung của tiểu trình trong xử lý song song hoặc các môi trường đa người dùng. Nói cách khác, khi có hai hay nhiều tiểu trình cùng muốn sử dụng một tài nguyên nào đó, để đảm bảo sự tranh chấp được diễn ra "công bằng", người ta sử dụng semaphore để điều khiển xem tiến trình nào được tiến vào vùng tranh chấp và sử dụng tài nguyên, khi tiến trình đó thoát khỏi vùng tranh chấp thì các tiến trình nào sẽ được vào tiếp theo.

Semaphore được xem như một danh sách các đơn vị còn trống của một tài nguyên trong máy tính. Có 2 thao tác cơ bản trên semaphore là yêu cầu tài nguyên và giải phóng tài nguyên. Nếu cần

thiết, semaphore còn có thể làm cờ để đợi cho đến khi tài nguyên được một tiểu trình khác giải phóng.

5.4.2.1 Các hàm cơ bản khi sử dụng semaphore

Chức năng	**Tên hàm**	**Ghi chú**	**Ví dụ**
Sử dụng thư viện semapho-re	*#include <semaphore.h>*	Khai báo thêm thư viện pthread và rt khi biên dịch.	*gcc -o filename filename.c -lpthread -lrt*
Định nghĩa 1 semapho re có tên là *sem_na me*	*sem_t sem_name;*		*sem_t sem;*
Khởi tạo 1 biến semapho-re	*int sem_init (sem_t *sem_name, int pshared, unsigned int value);*	**sem_name:* con trỏ chỉ đến địa chỉ của biến semaphore (được khai báo như trên). *pshared:* - Nếu được đặt là 0: biến semaphore sẽ được chia sẻ giữa các tiểu trình của cùng 1 tiến trình (và cần đặt ở nơi mà tất cả các tiểu trình đều có	*sem_t sem;* *sem_init (&sem, 0, 10);*

		thể truy xuất được như biến toàn cục hoặc biến động). - Nếu được đặt khác 0: biến semaphore sẽ được chia sẻ giữa những tiến trình với nhau và cần được đặt ở vùng nhớ được chia sẻ (shared memory). *value:* giá trị khởi tạo cho semaphore là số không âm. *Giá trị trả về:* - Là 0 nếu thành công - Là -1 nếu thất bại	
Đợi 1 semapho -re	*int sem_wait(sem_t *sem);*	- Nếu giá trị của semaphore = 0: tiến trình bị block cho đến khi giá trị của semaphore > 0 (để có thể trừ đi 1). Lưu ý: giá trị của semaphore không là số âm (xem khai báo ở trên) - Nếu giá trị của semaphore > 0: giá trị của semaphore trừ đi 1 và return, tiến trình tiếp tục chạy. *Giá trị trả về:* - Là 0 nếu thành công.	*sem_wait(&sem);*

		- Là -1 nếu thất bại, giá trị của semaphore không thay đổi.	
Mở khóa 1 semapho -re	*int sem_post(sem_t *sem);*	Một trong các tiến trình/tiểu trình bị block bởi *sem_wait* sẽ được mở và sẵn sàng để thực thi. *Giá trị trả về:* - Là 0 nếu thành công - Là -1 nếu thất bại	*sem_post(&sem);*
Lấy giá trị của 1 semapho -re	*int sem_getvalue(sem_t *sem, int *valp);*	Lấy giá trị của semaphore và gán vào biến được xác định tại địa chỉ *valp.* *Giá trị trả về:* - Là 0 nếu thành công - Là -1 nếu thất bại	*sem_getvalue(&sem, &value);* Biến value lúc này có giá trị là giá trị của semaphore.
Hủy 1 biến semapho -re	*int sem_destroy(sem_t *sem)*	Hủy đi 1 biến semaphore. Lưu ý: nếu đã quyết định hủy biến semaphore thì cần chắc chắn là không còn tiến trình/tiểu trình nào truy xuất vào biến semaphore đó nữa. *Giá trị trả về:* - Là 0 nếu thành công - Là -1 nếu thất bại	*sem_destroy(&sem);*

5.3.1.2. Ví dụ về semaphore

Ví dụ có 2 process được thực thi song song như sau:

PROCESS A	**PROCESS B**

processA
{
while (true)
sells++;
}
	processB
	{
	while (true)
	products++;
	}

Process A mô tả số lượng hàng bán được: ***sells***

Process B mô tả số lượng sản phẩm được làm ra: ***products***

Biết rằng ban đầu chúng ta chưa có hàng và cũng chưa bán được gì: **sells = products = 0**

Do khả năng tạo ra hàng hóa và khả năng bán hàng là không đồng đều, có lúc bán đắt thì sẽ ***sells*** tăng nhanh, lúc bán ế thì ***sells*** tăng chậm lại. Lúc công nhân làm việc hiệu quả thì sẽ tạo ra ***products*** nhanh, ngược lại lúc công nhân mệt thì sẽ làm ra ***products*** chậm lại. Tuy nhiên, dù bán đắt hay ế, làm nhanh hay chậm thì vẫn phải đảm bảo một điều là phải "*có hàng thì mới bán được"*, nói cách khác ta phải đảm bảo: **products >= sells.**

Vậy yêu cầu đặt ra là sử dụng semaphore để đồng bộ 2 tiến trình: A (bán hàng) và B (tạo ra hàng) theo điều kiện trên?

Phân tích bài toán trên ta thấy như sau:

- PROCESS A muốn "bán hàng" thì phải kiểm tra xem liệu có hàng để bán hay không?
- PROCESS B khi "tạo ra hàng" xong sẽ thông báo là hàng đã có để bán!

Từ các ý trên ta nhận thấy ta có thể sử dụng 1 semaphore làm điều kiện để kiểm tra việc bán hàng của A và B như sau:

<table>
<tr><td colspan="2">sem_t sem; // Định nghĩa biến sem
sem_init (&sem, 0, 0); // Biến sem có giá trị ban đầu pshared = 0 và value = 0</td></tr>
<tr><td>PROCESS A</td><td>PROCESS B</td></tr>
<tr><td>processA
{
while (true){
sem_wait(&sem);
sells++;
}
}</td><td>processB
{
while (true){
products++;
sem_post(&sem);
}
}</td></tr>
</table>

Với 2 PROCESS A và PROCESS B, ta có 2 trường hợp như sau:

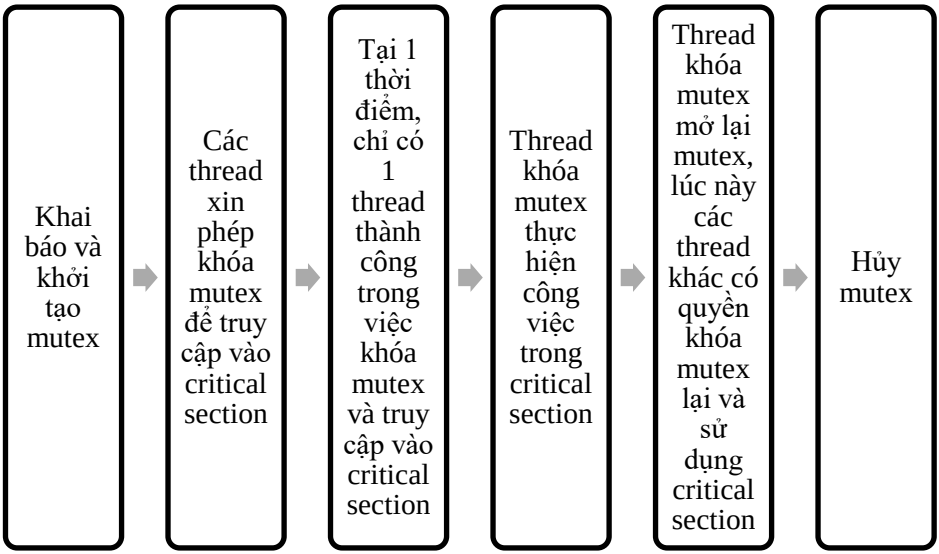
PROCESS A nhanh hơn PROCESS B (bán nhanh hơn làm)	**PROCESS B nhanh hơn PROCESS A (làm nhanh hơn bán)**

Mỗi khi PROCESS A muốn tăng biến sells (bán hàng), nó sẽ gặp hàm **sem_wait(&sem)** trước, hàm này sẽ kiểm tra xem giá trị của ***sem*** liệu có lớn hơn 0 (có hàng không). + Nếu ***sem.value = 0***: PROCESS A bị block không bán nữa. + Nếu ***sem.value > 0***: PROCESS A được phép tăng sells (được phép bán hàng) và giảm ***sem.value*** đi 1. PROCESS A sau khi chạy được 1 đoạn thời gian sẽ được dừng và chuyển cho PROCESS B chạy (do quy tắc lập lịch của hệ điều hành), lúc này PROCESS B sẽ tăng products (làm ra hàng) đồng thời tăng giá trị của ***sem*** và sau đó khi tới phiên của PROCESS A, nó sẽ có thể tăng giá trị của sells (bán hàng).	Sau khi PROCESS B tăng biến products (làm ra hàng mới), nó sẽ gọi hàm **sem_post(&sem)** để tăng giá trị của ***sem*** lên 1, lúc này PROCESS A nếu như đang bị block do hàm **sem_wait** trước đó sẽ được mở ra và sẵn sàng để "bán hàng". PROCESS B chạy được 1 đoạn thời gian sẽ phải nhường lại cho PROCESS A, lúc này PROCESS A sẽ trừ giá trị của ***sem*** đi 1 thông qua hàm **sem_wait**, rồi sau đó mới tăng giá trị của sells.

5.4.3 Mutex

Mutex là một trường hợp đơn giản của semaphore: **0 <= sem.value <= 1**

Thông thường, mutex được sử dụng như sau:



- **Các hàm cơ bản khi sử dụng Mutex**

Để có thể sử dụng mutex, ta cần phải include thư viện *pthread.h*.

Sau khi include thư viện trên, ta có thể sử dụng mutex thông qua các hàm:

Chức năng	**Tên hàm**	**Ghi chú**	**Ví dụ**
Khai báo 1 mutex có tên là *mutex_ name*	*pthread_mutex_t mutex_name*	Thông thường mutex được khai như một biến toàn cục	*pthread_mutex_t mutex;*
Khởi tạo 1 mutex	*int pthread_mutex_i nit (pthread_mutex _t *mutex, const pthread_mutexat tr_t *attr);*	**mutex:* con trỏ chỉ đến địa chỉ của mutex (được khai báo như trên). **attr:* con trỏ chỉ đến địa chỉ nơi mà chứa các thuộc tính cần khởi tạo ban đầu cho mutex. Nếu ở đây để là NULL thì mutex sẽ được khởi tạo với giá trị mặc định. *Giá trị trả về:* - Là 0 nếu thành công - Là -1 nếu thất bại	*pthread_mutex_t mutex;* *pthread_mutex_i nit(&mutex, NULL);*
Khóa 1 mutex	*int pthread_mutex_l ock(pthread_mu tex_t *mutex);*	Khóa mutex được tham chiếu bởi con trỏ **mutex* lại. Nếu như mutex này đã bị khóa bởi 1 thread khác trước đó thì thread đang gọi hàm khóa sẽ	*pthread_mutex_l ock(&mutex)*

		bị khóa lại cho đến khi mutex được mở ra. *Giá trị trả về:* - Là 0 nếu thành công - Là -1 nếu thất bại	
Mở khóa 1 mutex	*int pthread_mutex_unlock(pthread_mutex_t *mutex)*	Mở khóa mutex được tham chiếu bởi con trỏ **mutex*. Sau khi mở khóa, các thread khác sẽ được quyền tranh chấp quyền khóa mutex. *Giá trị trả về:* - Là 0 nếu thành công - Là -1 nếu thất bại	*pthread_mutex_unlock(&mutex)*
Hủy 1 mutex	*int pthread_mutex_destroy(pthread_mutex_t *mutex)*	*Hủy mutex được tham chiếu bởi con trỏ *mutex*.*	*pthread_mutex_destroy(&mutex)*

5.4.4 Câu hỏi chuẩn bị

Sinh viên chuẩn bị câu trả lời cho những câu hỏi sau trước khi bắt đầu phần thực hành:

- Phân biệt các khái niệm chương trình (program), tiến trình (process) và tiểu trình (thread)?
- Sự tranh chấp xảy ra khi nào? Cho ví dụ.

- Phân biệt sự khác nhau giữa 2 nhóm giải pháp: “busy waiting” và “sleep & wake up”. Liệt kê một số hệ điều hành sử dụng 2 nhóm giải pháp trên.

5.5 Bài tập thực hành

1. Hiện thực hóa mô hình trong ví dụ **5.3.1.2**, tuy nhiên thay bằng điều kiện sau: **sells <= products <= sells + [4 số cuối của MSSV]**
2. Cho một mảng a được khai báo như một mảng số nguyên có thể chứa n phần tử, a được khai báo như một biến toàn cục. Viết chương trình bao gồm 2 thread chạy song song:

   - Một thread làm nhiệm vụ sinh ra một số nguyên ngẫu nhiên sau đó bỏ vào a. Sau đó đếm và xuất ra số phần tử của a có được ngay sau khi thêm vào.
   - Thread còn lại lấy ra một phần tử trong a (phần tử bất kỳ, phụ thuộc vào người lập trình). Sau đó đếm và xuất ra số phần tử của a có được ngay sau khi lấy ra, nếu không có phần tử nào trong a thì xuất ra màn hình “Nothing in array a”.

   Chạy thử và tìm ra lỗi khi chạy chương trình trên khi chưa được đồng bộ. Thực hiện đồng bộ hóa với semaphore.
3. Cho 2 process A và B chạy song song như sau:

```
int x = 0;
```

PROCESS A	PROCESS B

processA()
{
while(1){
x = x + 1;
if (x == 20)
x = 0;
print(x);
}
}
	processB()
	{
	while(1){
	x = x + 1;
	if (x == 20)
	x = 0;
	print(x);
	}
	}

Hiện thực mô hình trên C trong hệ điều hành Linux và nhận xét kết quả.

4. Đồng bộ với mutex để sửa lỗi bất hợp lý trong kết quả của mô hình Bài 3.

5.6 Bài tập ôn tập

1. Biến ans được tính từ các biến x1, x2, x3, x4, x5, x6 như sau:

w = x1 * x2; (a)

v = x3 * x4; (b)

y = v * x5; (c)

z = v * x6; (d)

y = w * y; (e)

z = w * z; (f)

ans = y + z; (g)

Giả sử các lệnh từ (a) → (g) nằm trên các thread chạy song song với nhau. Hãy lập trình mô phỏng và đồng bộ trên C trong hệ điều hành Linux theo thứ tự sau:

- (c), (d) chỉ được thực hiện sau khi v được tính
- (e) chỉ được thực hiện sau khi w và y được tính
- (g) chỉ được thực hiện sau khi y và z được tính