
MỤC LỤC

BÀI 5.	ĐỒNG BỘ HÓA TIỀN TRÌNH, TIỂU TRÌNH.....	1
5.1	Mục tiêu.....	1
5.2	Nội dung thực hành.....	1
5.3	Sinh viên chuẩn bị.....	1
5.4	Hướng dẫn thực hành	12
5.5	Bài tập ôn tập.....	13

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. ĐỒNG BỘ HÓA TIỀN TRÌNH, TIỂU TRÌNH

5.1 Mục tiêu

- ✚ 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 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.

5.3.1 Semaphore

Trong hệ điều hành, semaphore được biết đến 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.

5.3.1.1 Các hàm cơ bản khi sử dụng semaphore

Chức năng	Tên hàm	Ghi chú	Ví dụ
Sử dụng thư viện semaphore-re	<code>#include <semaphore.h></code>	Khai báo thêm thư viện pthread và rt khi biên dịch.	<code>gcc -o filename filename.c -lpthread -lrt</code>
Định nghĩa 1 semaphore có tên	<code>sem_t sem_name;</code>		<code>sem_t sem;</code>

là <i>sem_name</i>			
Khởi tạo 1 biến semaphore -re	<i>int sem_init (sem_t *sem_name, int pshared, unsigned int value);</i>	<p><i>*sem_name</i>: con trỏ chỉ đến địa chỉ của biến semaphore (được khai báo như trên).</p> <p><i>pshared</i>:</p> <ul style="list-style-type: none"> - Nếu được đặt là 0: biến semaphore sẽ được chia sẻ giữa các tiến trình của cùng 1 tiến trình (và cần đặt ở nơi mà tất cả các tiến trình đều có 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). <p><i>value</i>: giá trị khởi tạo cho semaphore là số không âm.</p> <p><i>Giá trị trả về</i>:</p> <ul style="list-style-type: none"> - Là 0 nếu thành công - Là -1 nếu thất bại 	<i>sem_t sem; sem_init (&sem, 0, 10);</i>
Đợi 1	<i>int sem_wait(sem_t</i>	- Nếu giá trị của semaphore	<i>sem_wait(</i>

semapho -re	<i>*sem</i>);	<p>= 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)</p> <p>- Nếu giá trị của semaphore > 0: giá trị của semaphore trừ đi 1 và return, tiến trình tiếp tục chạy.</p> <p><i>Giá trị trả về:</i></p> <p>- Là 0 nếu thành công.</p> <p>- Là -1 nếu thất bại, giá trị của semaphore không thay đổi.</p>	<i>&sem</i>);
Mở khóa 1 semapho -re	<i>int sem_post(sem_t *sem</i>);	<p>Một trong các tiến trình/tiêu trình bị block bởi <i>sem_wait</i> sẽ được mở và sẵn sàng để thực thi.</p> <p><i>Giá trị trả về:</i></p> <p>- Là 0 nếu thành công</p> <p>- Là -1 nếu thất bại</p>	<i>sem_post(&sem</i>);
Lấy giá trị của 1 semapho -re	<i>int sem_getvalue(sem_t *sem, int *valp</i>);	<p>Lấy giá trị của semaphore và gán vào biến được xác định tại địa chỉ <i>valp</i>.</p> <p><i>Giá trị trả về:</i></p> <p>- Là 0 nếu thành công</p>	<i>sem_getvalue(&sem, &value);</i> Biến value lúc này có

		- Là -1 nếu thất bại	giá trị là giá trị của semaphore.
Hủy 1 biến semaphore	<i>int</i> <i>sem_destroy(sem_t *sem)</i>	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. <i>Giá trị trả về:</i> - Là 0 nếu thành công - Là -1 nếu thất bại	<i>sem_destroy(&sem);</i>

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:

sem_t sem; // Định nghĩa biến <i>sem</i> sem_init (&sem, 0, 0); // Biến <i>sem</i> có giá trị ban đầu <i>pshared</i> = 0 và <i>value</i> = 0	
PROCESS A	PROCESS B
processA	processB
{	{

<pre> while (true){ sem_wait(&sem); sells++; } </pre>	<pre> while (true){ products++; sem_post(&sem); } </pre>
--------------------------------------------------------------------------	-----------------------------------------------------------------------------

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

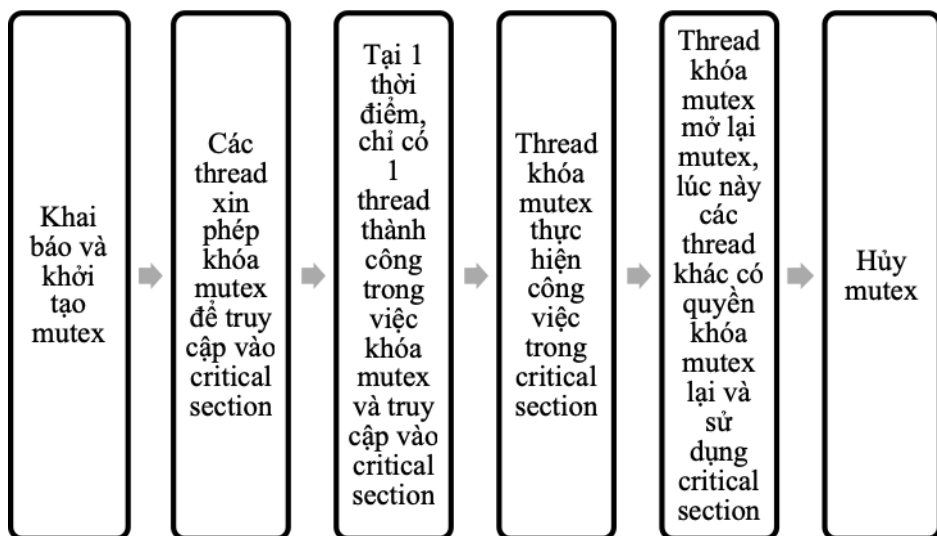
PROCESS A nhanh hơn PROCESS B (bán nhanh hơn làm)	PROCESS B nhanh hơn PROCESS A (làm nhanh hơn bán)
<p>Mỗi khi PROCESS A muốn tăng biến sells (bán hàng), nó sẽ gặp hàm sem_wait(&sem) trước, hàm này sẽ kiểm tra xem giá trị của <i>sem</i> liệu có lớn hơn 0 (có hàng không).</p> <p>+ Nếu <i>sem.value</i> = 0: PROCESS A bị block không bán nữa.</p> <p>+ Nếu <i>sem.value</i> > 0: PROCESS A được phép tăng sells (được phép bán hàng) và giảm <i>sem.value</i> đi 1.</p>	<p>Sau khi PROCESS B tăng biến products (làm ra hàng mới), nó sẽ gọi hàm sem_post(&sem) để tăng giá trị của <i>sem</i> 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”.</p> <p>PROCESS B chạy được 1 đoạn thời gian sẽ phải nhường lại cho PROCESS A, lúc này PROCESS A sẽ trừ giá trị của <i>sem</i> đi 1 thông qua hàm</p>

<p>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 <i>sem</i> và sau đó khi tới phiên của PROCESS A, nó sẽ có thể tăng giá trị của sells (bán hàng).</p>	<p>sem_wait, rồi sau đó mới tăng giá trị của sells.</p>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------

5.3.2 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 <i>mutex_</i>	<i>pthread_mutex_t mutex_name</i>	Thông thường mutex được khai như một biến toàn cục	<i>pthread_mutex_t mutex;</i>

<i>name</i>			
Khởi tạo mutex 1	<i>int</i> <i>pthread_mutex_init</i> <i>(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);</i>	<i>*mutex</i> : con trỏ chỉ đến địa chỉ của mutex (được khai báo như trên). <i>*attr</i> : 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. <i>Giá trị trả về:</i> - Là 0 nếu thành công - Là -1 nếu thất bại	<i>pthread_mutex_t</i> <i>mutex;</i> <i>pthread_mutex_init(&mutex, NULL);</i>
Khóa mutex 1	<i>int</i> <i>pthread_mutex_lock(pthread_mutex_t *mutex);</i>	Khóa mutex được tham chiếu bởi con trỏ <i>*mutex</i> lại. Nếu như mutex này đã bị khóa bởi 1 thread khác trước đó thì thread đang gọi hàm khóa sẽ bị khóa lại cho đến khi mutex được mở ra. <i>Giá trị trả về:</i> - Là 0 nếu thành công - Là -1 nếu thất bại	<i>pthread_mutex_lock(&mutex)</i>
Mở khóa 1	<i>int</i> <i>pthread_mutex_unlock</i>	Mở khóa mutex được tham chiếu bởi con trỏ <i>*mutex</i> .	<i>pthread_mutex_unlock(&mutex)</i>

mutex	<code>unlock(pthread_mutex_t *mutex)</code>	Sau khi mở khóa, các thread khác sẽ được quyền tranh chấp quyền khóa mutex. <i>Giá trị trả về:</i> - Là 0 nếu thành công - Là -1 nếu thất bại	
Hủy mutex	<code>int pthread_mutex_destroy(pthread_mutex_t *mutex)</code>	<i>Hủy mutex được tham chiếu bởi con trỏ *mutex.</i>	<code>pthread_mutex_destroy(&mutex)</code>

5.3.3 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.

5.4 Hướng dẫn thực hành

1. Hiện thực hóa mô hình trong ví dụ **5.3.1.2**, tuy nhiên thay bằng điều kiện sau: **$\text{sells} \leq \text{products} \leq \text{sells} + [2 \text{ số cuối của MSSV} + 10]$**
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() {	processB() {

<pre> while(1){ x = x + 1; if (x == 20) x = 0; print(x); } </pre>	<pre> while(1){ x = x + 1; if (x == 20) x = 0; print(x); } </pre>
---------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------

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

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

5.5 Bài tập ôn tập

- 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) \rightarrow (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