>>> LAB TUTORIAL <<<

INTRODUCTION TO OPERATING SYSTEM / Course ID 502047

Lab Part 8 – Synchronization Tools

Mục tiêu	Lý thuyết liên quan	Tài nguyên
Atomic variables		Yêu cầu gcc 4.9
Barrier		
Semaphores		Sử dụng image Ubuntu 14 / 16
Mutex Locks		
Condition variables		
Monitor		C++

Yêu cầu nộp bài: các tập tin mã nguồn .c và tập tin khả thực thi .out của các "ví dụ" và bài tập cuối hướng dẫn.

Preferences

[1] Abraham Silberschatz, Peter B. Galvin, Greg Gagne, [2018], Operating System Concepts, 10th edition, John Wiley & Sons, New Jersey.

Programming Problems of Chapter 6 and 7.

[2] Greg Gagne , [2019], GitHub OS-BOOK OSC10e, Westminster College, United States

Access https://github.com/greggagne/osc10e in September 2019.

[3] Process Synchronization, *geeksforgeeks.org*

1. Semaphore

Semaphores được sử dụng để đồng bộ các tiến trình và tiểu trình. Semaphores được kết hợp với hàng đợi thông điệp và bộ nhớ dùng chung trong các IPC cơ sở trong các hệ thống Unix. Có hai loại semaphores, semaphores System-V truyền thống và semaphores POSIX. Trong bài học này, semaphores POSIX sẽ được giới thiệu.

Có hai loại semaphores POSIX - được đặt tên và vô danh. Như thuật ngữ cho thấy, semaphores được đặt tên có một tên, đó là định dạng / somename. Ký tự đầu tiên là dấu gạch chéo về phía trước, theo sau là một hoặc nhiều ký tự, không có ký tự nào là dấu gạch chéo. Đầu tiên chúng ta sẽ xem xét các semaphores được đặt tên và sau đó là semaphore vô danh.

Các chương trình sử dụng ngữ nghĩa POSIX cần được liên kết với thư viện pthread.

Một biến semaphore sẽ được gán đến các tài nguyên. Khi một tiến trình muốn sử dụng tài nguyên nó sẽ kiểm tra biến semaphore của tài nguyên này. Nếu giá trị của biến này khác 0, tài nguyên đang có sẵn. Nếu giá trị biến này bằng 0, tài nguyên đang được sử dụng.

1.1 Thư viện và các lời gọi

```
#include <sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
sem_open
sem_post
sem_wait
sem_trywait
sem_timedwait
sem_getvalue
sem_unlink
sem_init
sem_destroy
```

1.2 Các ví dụ sai sót

Ví dụ 1.1: Trong ví dụ này, hai tiểu trình được tạo ra và trong thân hàm của các tiểu trình này, một biến counter được cập nhật để ghi lại số thứ tự của tiểu trình khi bắt đầu và khi hoàn thành.

```
>gcc vidu1_1.c -o vidu1_1.out -lpthread
>./vidu1_1.out
Job 1 has started
Job 2 has started
Job 2 has finished
Job 2 has finished
```

Tại sao lại có 2 dòng “Job 2 has finished”?

Ví dụ 1.2 Bài toán counter++ và counter--.

```
>gcc vidu1_2.c -o vidu1_2.out -lpthread
>./vidu1_2.out
```

1.3 Atomic Variable

Thư viện

```
#include <stdatomic.h>
```

Ví dụ 1.3 Sự khác nhau của biến số atomic và biến số thông thường.

```
>gcc vidu1_3.c -o vidu1_3.out -lpthread
>./vidu1_3.out
The atomic counter is 1000
The non-atomic counter is 996
>./vidu1_3.out
The atomic counter is 1000
The non-atomic counter is 991
>./vidu1_3.out
The atomic counter is 1000
The non-atomic counter is 997
```

Ví dụ 1.4 Atomic operation.

Ngoài các biến số đơn nguyên, các thao tác cũng có thể được định nghĩa[1] là đơn nguyên, khi đó quá trình thực thi các thao tác này sẽ tránh khỏi tình trạng cạnh tranh giữa các tiến trình.

```
>gcc vidu1_4.c -o vidu1_4.out -lpthread
>./vidu1_4.out
```

3. POSIX Unnamed Semaphore calls

Trong ví dụ trước, các semaphores là cục bộ của một tiến trình; chúng chỉ được sử dụng bởi chủ đề của nó. Không có quá trình khác sử dụng chúng. Vì vậy, có vẻ như lãng phí nỗ lực để có tên semaphore trên toàn hệ thống và sử dụng các cuộc gọi như sem_open. Có những semaphores không tên POSIX có thể làm những gì chúng ta cần một cách đơn giản và hiệu quả hơn nhiều. Đầu tiên, hệ thống gọi,

sem_init

```
#include <semaphore.h>
```

[1] http://en.cppreference.com/w/c/atomic

```
int sem_init (sem_t *sem, int pshared, unsigned int value);
```

sem_init tương đương với **sem_open** và dùng cho các semaphores không tên. Cần định nghĩa một biến kiểu sem_t và chuyển đổi số con trỏ của nó là sem trong lời gọi. Hoặc, có thể định nghĩa một con trỏ và cấp phát bộ nhớ động bằng cách sử dụng lời gọi malloc. sem_init khởi tạo semaphore được trỏ bởi sem với giá trị số nguyên value. Đối số thứ hai được pshared cho biết semaphore này được dùng chia sẻ giữa các tiểu tình của một tiến trình hay giữa các tiến trình. Nếu pshared có giá trị 0, semaphore được chia sẻ giữa các tiểu trình của một tiến trình. Semaphore nên được đặt tại một nơi mà mọi tiểu trình có thể tìm thấy. Nếu pshared có giá trị khác 0, semaphore được chia sẻ bởi các tiến trình. Trong trường hợp đó, semaphore phải được đặt trong một phân đoạn bộ nhớ dùng chung được gắn với các tiến trình liên quan.

sem_destroy

```
#include <semaphore.h>
int sem_destroy (sem_t *sem);
```

sem_destroy xoá semaphore không tên đang liên kết với name.

Ví dụ 3.1

1 2	

4. Mutex lock

Các tiểu trình POSIX cung cấp nhiều luồng thực thi bên trong một tiến trình. Các tiểu trình đều có ngăn xếp riêng nhưng chúng chia sẻ biến số toàn cục và các heap. Vì vậy, các biến toàn cục có thể nhìn thấy (và truy cập) bởi nhiều tiểu trình. Hơn nữa, các tiểu trình cũng cần đồng bộ hóa hành động của chúng để chúng cùng thực hiện các mục tiêu chung của tiến trình cha. Các vấn đề cốt lõi của lập trình đồng thời, loại trừ lẫn nhau và đồng bộ hóa có liên quan đến các tiểu trình cũng giống như các vấn đề của hệ thống đa vi xử lý.

Pthreads có một semaphore đặc biệt để loại trừ lẫn nhau được gọi là đối tượng **mutex**. với các lời gọi tương tự P(mutex) và V(mutex) ở đầu và cuối đoạn mã nguy cơ thì chỉ có một tiểu trình được đi vào đoạn mã nguy cơ, tại mọi thời điểm mà tiến trình chạy trong hệ thống.

4.1 Tạo ra Pthread mutex

Các đơn giản nhất để khởi tạo một mutex là định nghĩa và khởi tạo nó như là một biến số toàn cục.

```
pthread_mutex_t new_mutex = PTHREAD_MUTEX_INITIALIZER;
```

Cách này chỉ thực hiện với các biến số toàn cục. Với các biến số cấp phát động và tự động, lời gọi `pthread_mutex_init` cần được sử dụng để khởi tạo mutex.

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const
   pthread_mutexattr_t *restrict attr)
```

- Lời gọi trên tạo một mutex, được tham chiếu bởi mutex, với các thuộc tính được chỉ định bởi attr. Nếu attr là NULL, thuộc tính mutex mặc định (NONRECURSIVE) được sử dụng.

- Giá trị trả về:

+ Nếu thành công, trả về 0 và trạng thái của mutex sẽ được khởi tạo và trạng thái "mở".

+ Thất bại, trả về -1.

Lời gọi `pthread_mutex_lock` và `pthread_mutex_unlock` được mô tả sau đây.

4.2 Lời gọi pthread_mutex_lock

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

- `pthread_mutex_lock` khoá mutex được xác định bởi con trỏ truyền vào qua đối số. Nếu mutex đã bị khoá, lời gọi bị chặn lại cho đến khi mutex chuyển trạng thái và có thể bị khoá trở lại. `pthread_mutex_lock` cung cấp thao tác P cho semaphore mutex.

- Giá trị trả về: Thành công: 0; Thất bại: -1.

4.3 Lời gọi pthread_mutex_unlock

```
#include <pthread.h>
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

`pthread_mutex_unlock` mở khoá mutex được xác định bởi con trỏ truyền vào qua đối số. `pthread_mutex_unlock` cung cấp thao tác V cho semaphore mutex.

- Giá trị trả về: Thành công: 0; Thất bại: -1.

4.4 Huỷ bỏ lock mutex

```
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

- Xóa một đối tượng mutex, xác định một mutex. Mutexes được sử dụng để bảo vệ tài nguyên được chia sẻ. mutex được đặt thành một giá trị không hợp lệ, nhưng có thể được khởi tạo lại bằng cách sử dụng pthread_mutex_init ().

- Giá trị trả về: Thành công: 0; Thất bại: -1.

Ví dụ 4.1: Sửa lại sự sai sót của ví dụ 1.1 bằng khoá Mutex.

5. Condition Variables

Các biến điều kiện [condition variable] cung cấp một cách khác cho các tiến trình để đồng bộ hóa. Trong khi các biết mutex thực hiện đồng bộ hóa bằng cách kiểm soát truy cập của tiểu trình vào dữ liệu, các biến điều kiện cho phép các tiểu trình xử lý đồng bộ hóa dựa trên giá trị thực của dữ liệu.

Nếu không có các biến điều kiện, lập trình viên sẽ cần phải có các tiểu trình liên tục thăm dò [polling] (có thể trong đoạn mã nguy cơ), để kiểm tra xem điều kiện có được đáp ứng hay không. Điều này có thể rất tốn tài

nguyên vì tiểu trình sẽ liên tục bận rộn trong hoạt động này [busy waiting]. Một biến điều kiện là một cách để đạt được cùng một mục tiêu mà không cần thăm dò.

Một biến điều kiện luôn được sử dụng cùng với khóa mutex.

5.1 Tạo ra và hủy bỏ các biến điều kiện

```
pthread_cond_init (condition,attr)
pthread_cond_destroy (condition)
pthread_condattr_init (attr)
pthread_condattr_destroy (attr)
```

Các biến điều kiện phải được khai báo với loại pthread_cond_t và phải được khởi tạo trước khi chúng có thể được sử dụng. Có hai cách để khởi tạo một biến điều kiện:

Khai báo tĩnh:

```
pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;
```

Khai báo động với hàm pthread_cond_init(). ID của biến điều kiện đã tạo được trả về tiểu trình gọi thông qua tham số điều kiện. Phương pháp này cho phép thiết lập các thuộc tính đối tượng biến điều kiện thông qua attr.

Đối tượng attr tùy chọn được sử dụng để đặt thuộc tính biến điều kiện. Chỉ có một thuộc tính được xác định cho các biến điều kiện: process-shared, cho phép biến điều kiện được nhìn thấy bởi các tiểu trình trong các tiến trình khác. Đối tượng thuộc tính, nếu được sử dụng, phải là loại pthread_condattr_t (có thể được chỉ định là NULL để chấp nhận mặc định).

Lưu ý rằng không phải tất cả các hiện thực có thể cung cấp thuộc tính process-shared.

Lời gọi pthread_condattr_init() và pthread_condattr_destroy() được dùng để tạo và hủy các đối tượng thuộc tính của các biến số điều kiện. pthread_cond_destroy() dược dùng để hủy bỏ một biến điều kiện không còn sử dụng nữa.

Ra hiệu và chờ tín hiệu với biến điều kiện

```
pthread_cond_wait (condition,mutex)
pthread_cond_signal (condition)
pthread_cond_broadcast (condition)
```

- Lời gọi pthread_cond_wait () chặn tiểu trình đã thực hiện lời gọi cho đến khi điều kiện đã được chỉ định trước có tín hiệu. Lời gọi này cần thực thi khi mutex bị khóa và nó sẽ tự động giải phóng mutex trong khi nó chờ. Sau khi tín hiệu được nhận và tiểu trình được đánh thức, mutex sẽ tự động bị khóa để sử dụng bởi tiểu trình. Lập trình viên cần mở khóa mutex khi tiểu trình kết thúc với nó.

- *Ghi chú: Sử dụng vòng lặp WHILE thay vì câu lệnh IF (xem lời gọi watch_count trong ví dụ 5.1) để kiểm tra điều kiện chờ đợi để tránh một số vấn đề tiềm ẩn, chẳng hạn như:*

> *+ Nếu một số tiểu trình đang chờ tín hiệu đánh thức giống nhau, chúng sẽ lần lượt lấy được mutex và bất kỳ một trong số chúng sau đó có thể sửa đổi điều kiện mà tất cả chúng chờ đợi.*
>
> *+ Nếu tiểu trình nhận được tín hiệu do lỗi chương trình*

+ Thư viện Pthreads được phép đưa ra các đánh thức giả cho một chuỗi chờ mà không vi phạm tiêu chuẩn.

- Lời gọi pthread_cond_signal() được sử dụng để báo hiệu (hoặc đánh thức) một tiểu trình khác đang chờ trên biến điều kiện. Nó nên được gọi sau khi mutex bị khóa và phải mở khóa mutex để hoàn thành lời gọi pthread_cond_wait().

- Lời gọi pthread_cond_broadcast () nên được sử dụng thay cho pthread_cond_signal () nếu có nhiều hơn một tiểu trình trong trạng thái chờ chặn.

- Sẽ xuất hiện lỗi luận lý khi gọi pthread_cond_signal() trước khi gọi pthread_cond_wait().

- Lưu ý: Khóa và mở khóa của biến mutex liên quan cần phải được thực thi đúng và hợp lý. Ví dụ:

+ Không khóa mutex trước khi gọi pthread_cond_wait () có thể khiến nó KHÔNG bị chặn.

+ Không mở khóa mutex sau khi gọi pthread_cond_signal() có thể không cho phép một lời gọi pthread_cond_wait() phù hợp hoàn thành (nó sẽ vẫn bị chặn).

Ví dụ 5.1

- Tiến trình cha tạo ra ba tiểu trình.

- Hai trong số các tiểu trình thực hiện tác vụ và cập nhật một biến "count.

- Tiểu trình thứ ba chờ cho đến khi biến count đạt đến một giá trị được chỉ định.

```
Starting watch_count(): thread 1
inc_count(): thread 2, count = 1, unlocking mutex
inc_count(): thread 3, count = 2, unlocking mutex
watch_count(): thread 1 going into wait...
inc_count(): thread 3, count = 3, unlocking mutex
inc_count(): thread 2, count = 4, unlocking mutex
inc_count(): thread 3, count = 5, unlocking mutex
inc_count(): thread 2, count = 6, unlocking mutex
inc_count(): thread 3, count = 7, unlocking mutex
inc_count(): thread 2, count = 8, unlocking mutex
inc_count(): thread 3, count = 9, unlocking mutex
inc_count(): thread 2, count = 10, unlocking mutex
inc_count(): thread 3, count = 11, unlocking mutex
inc_count(): thread 2, count = 12  Threshold reached. Just sent signal.
inc_count(): thread 2, count = 12, unlocking mutex
watch_count(): thread 1 Condition signal received.
watch_count(): thread 1 count now = 137.
inc_count(): thread 3, count = 138, unlocking mutex
inc_count(): thread 2, count = 139, unlocking mutex
inc_count(): thread 3, count = 140, unlocking mutex
inc_count(): thread 2, count = 141, unlocking mutex
inc_count(): thread 3, count = 142, unlocking mutex
inc_count(): thread 2, count = 143, unlocking mutex
inc_count(): thread 3, count = 144, unlocking mutex
inc_count(): thread 2, count = 145, unlocking mutex
Main(): Waited on 3 threads. Final value of count = 145. Done.
```

Ví dụ 5.2

```
$ gcc vidu5_2.c -o vidu5_2.out -lpthread
```

```
$ ./vidu5_2.out
Thread 0: 1
Thread 3: 1
Thread 2: 1
Thread 7: 1
...
```

5.4 Reader-Writer problem using Monitors (pthreads)

Có một tài nguyên chia sẻ được truy cập bởi nhiều tiến trình, là các bộ ghi và các bộ đọc. Nhiều bộ đọc có thể đọc tài nguyên chia sẻ cùng một lúc, nhưng chỉ một bộ ghi có thể ghi vào tài nguyên chia sẻ tại một thời điểm. Khi bộ ghi đang ghi dữ liệu vào tài nguyên, không có tiến trình nào khác có thể truy cập tài nguyên. Một bộ ghi không thể ghi vào tài nguyên nếu có bất kỳ bộ đọc nào truy cập vào tài nguyên tại thời điểm đó. Tương tự, một bộ đọc không thể đọc nếu có một bộ ghi đang truy cập tài nguyên hoặc nếu có bất kỳ bộ ghi nào đang chờ.

Bài toán Bộ ghi - Bộ đọc có thể được giải quyết bằng một bộ quan sát [monitor] trong thư viện pthreads.

Ví dụ 5.3 Bài toán Bộ ghi - Bộ đọc

```
>gcc -pthread vidu5_3.c -o vidu5_3.out
>./vidu5_3.out
```

Bài tập.

1. Tạo 2 tiểu trình con, một tiểu trình in ra các số lẻ từ 1 đến 11, một tiểu trình in ra số chẵn từ 2 đến 10. Hãy sử dụng semaphore sau cho màn hình in ra dãy số đúng theo thứ tự từ 1 đến 11.

2. a (Bài tập 4.24) Một cách giá trị π khá thú vị là sử dụng kỹ thuật Monte Carlo, liên quan đến ngẫu nhiên. Kỹ thuật này hoạt động như sau: Giả sử bạn có một vòng tròn bán kính là 1 nội tiếp trong một hình vuông cạnh là 2, như thể hiện trong hình sau:

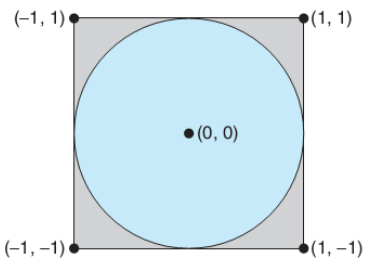


Figure 4.25 Monte Carlo technique for calculating π.

- Đầu tiên, tạo một chuỗi các điểm ngẫu nhiên dưới dạng tọa độ (x, y) đơn giản. Những điểm này phải nằm trong tọa độ Descartes bị ràng buộc hình vuông. Trong tổng số điểm ngẫu nhiên được tạo, một số sẽ xảy ra trong vòng tròn.
- Tiếp theo, ước tính π bằng cách thực hiện phép tính sau: π = 4 × (số điểm trong vòng tròn) / (tổng số điểm)

Hãy viết một phiên bản đa luồng của thuật toán này để tạo ra **một tiểu trình riêng biệt** sinh ra một số lượng điểm ngẫu nhiên; sau đó tính số lượng điểm nằm trong hình và lưu trữ kết quả đó trong một biến toàn cục. Khi tiểu trình này kết thúc, tiến trình cha sẽ tính toán và xuất giá trị ước tính của π. Hãy đánh giá độ chính xác của số π với số lượng điểm ngẫu nhiên được tạo ra. Theo nguyên tắc, số lượng điểm càng lớn, giá trị tính càng tiến gần π.

2.b (Bài tập 7.17) Trong câu a, chỉ có một tiểu trình sinh điểm ngẫu nhiên, hãy thay đổi chương trình để sinh ra **nhiều tiểu trình**, mỗi tiểu trình tạo ra các điểm ngẫu nhiên và xác định xem các điểm có nằm trong vòng tròn hay không. Mỗi tiểu trình sẽ phải cập nhật tổng số điểm nằm trong và ngoài vòng tròn (là các biến toàn cục). Sử dụng khóa mutex để đảm bảo điều kiện cạnh tranh khi các tiểu trình đồng thời cập nhật giá trị toàn cục đã mô tả.

Phần mở rộng - Đọc thêm - SV tự học

2. POSIX Named Semaphore calls

2.1 Lời gọi sem_open

```
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
sem_t *sem_open (const char *name, int oflag);  //form #1
sem_t *sem_open (const char *name, int oflag,
                 mode_t mode, unsigned int value);   //form #2
```

sem_open là lời gọi để bắt đầu một semaphore. sem_open mở một semaphore hiện có hoặc tạo một semaphore mới và mở nó cho các hoạt động tiếp theo. Tham số đầu tiên, name, là tên của semaphore, một chuỗi kí tự hằng số. oflag có thể chứa O_CREAT, trong trường hợp đó, semaphore được tạo nếu nó chưa tồn tại. Nếu cả O_CREAT và O_EXCL được chỉ định, lời gọi sẽ báo lỗi nếu semaphore có tên được chỉ định đã tồn tại. Nếu tham số oflags có thiết lập O_CREAT, hình thức thứ hai của sem_open phải được sử dụng, có thêm 2 tham số: chế độ mode và giá trị nguyên không âm. Tham số mode chỉ định các quyền cho semaphore, được che dấu bằng umask cho tiến trình, tương tự như mode trong lời gọi hệ thống open() cho các tập tin. Tham số cuối cùng là giá trị khởi tạo cho semaphore. Nếu O_CREAT được chỉ định trong oflag và semaphore đã tồn tại, cả tham số chế độ và giá trị đều bị bỏ qua.

sem_open trả về một con trỏ tới semaphore khi thành công. Con trỏ này phải được sử dụng trong các cuộc gọi tiếp theo cho semaphore. Nếu cuộc gọi thất bại, sem_open trả về SEM_FAILED và errno được đặt lỗi tương ứng.

Trong Linux, các ngữ nghĩa POSIX được tạo trong thư mục /dev/shm. Các semaphores được đặt tên với một tiền tố, sem. theo sau là tên được truyền trong lệnh gọi sem_open.

2.2 Lời gọi sem_post

```
#include <semaphore.h>
int sem_post (sem_t *sem);
```

sem_post tăng semaphore. Nó cung cấp hoạt động V cho semaphore, tương tự signal(). Nó trả về 0 khi thành công và -1 khi lỗi.

2.3 Lời gọi sem_wait / sem_trywait / sem_timedwait

```
#include <semaphore.h>
int sem_wait (sem_t *sem);
sem_timedwait

struct timespec {
    time_t tv_sec;      /* Seconds */
    long   tv_nsec;     /* Nanoseconds [0 .. 999999999] */
};
```

sem_wait giảm các semaphore được chỉ đến bởi *sem. Nếu giá trị semaphore là khác không, sự sụt giảm xảy ra ngay lập tức. Nếu giá trị semaphore bằng 0, các lời gọi bị chặn cho đến khi semaphore trở nên lớn hơn 0 và việc giảm dần được thực hiện. sem_wait trả về 0 khi thành công và -1 khi lỗi. Trong trường hợp có lỗi, giá trị semaphore được giữ nguyên và errno được đặt thành số lỗi thích hợp. sem_wait cung cấp lệnh gọi cho hoạt động P cho semaphore.

sem_trywait giống như lời gọi sem_wait, ngoại trừ: nếu giá trị semaphore bằng 0, nó không chặn mà trả về ngay lập tức với errno được đặt thành EAGAIN.

sem_timedwait cũng giống như cuộc gọi sem_wait, ngoại trừ: có bộ đếm thời gian được chỉ định với con trỏ, abs_timeout. Nếu giá trị semaphore lớn hơn 0, nó sẽ bị giảm và giá trị thời gian chờ được trỏ bởi abs_timeout không được sử dụng. Trong trường hợp đó, lời gọi hoạt động giống như lời gọi sem_wait. Nếu giá trị semaphore bằng 0, các lời gọi bị chặn, thời lượng chặn tối đa là thời gian cho đến khi bộ hẹn giờ tắt. Nếu giá trị semaphore trở nên lớn hơn 0 trong khoảng thời gian chặn, semaphore bị giảm ngay lập tức và lời gọi được đánh thức trở lại. Nếu không, bộ hẹn giờ sẽ tắt và cuộc gọi trở lại với errno được đặt thành ETIMEDOUT. Bộ định thời được chỉ định trong struct timespec{};

sem_getvalue

```
#include <semaphore.h>
int sem_getvalue (sem_t *sem, int *sval);
```

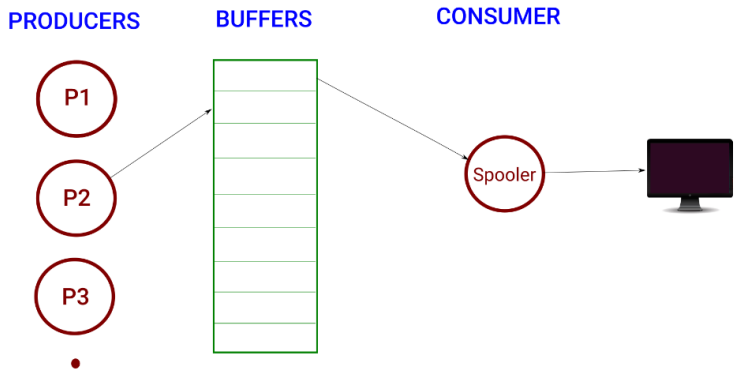
sem_getvalue nhận giá trị của semaphore được chỉ bởi sem. Giá trị được trả về trong số nguyên được chỉ bởi sval. Nó trả về 0 khi thành công và -1 khi lỗi, với lỗi không cho biết lỗi thực tế.

sem_unlink

```
#include <semaphore.h>
int sem_unlink (const char *name);
```

sem_unlink xoá semaphore có tên đang liên kết với name.

Ví dụ 2.1



Hình 1. Mô hình nhiều Producer và một Consumer

Tải về toàn bộ code ở GitHub, sau đây chỉ là những lệnh quan trọng.

```
1
2
```

```
>./vidu2_1.out
```