EXERCISES

Họ và tên: Đào Quốc Khánh
MSSV: 2013452

Problem 1:

- Chúng ta định nghĩa biến toàn cục "number" đại diện cho số lượng học sinh của lớp tại một thời điểm cụ thể.

1. Các trường hợp có thể xảy ra:

+ *Nhiều học sinh cùng đăng ký tại cùng một thời điểm*: Việc cập nhật đồng thời biến number (number++) có thể gây ra race condition.

+ *Nhiều học sinh cùng hủy đăng ký tại cùng một thời điểm*: Việc cập nhật đồng thời biến number (number--) có thể gây ra race condition.

+ *Tại một thời điểm cụ thể chỉ có 1 học sinh đăng ký*: không xảy ra lỗi

+ *Tại một thời điểm cụ thể chỉ có 1 học sinh hủy đăng ký*: không xảy ra lỗi

2. Chúng ta có thể sử dụng mutex để tránh xảy ra race condition trong 2 trường hợp đầu:

```
int number = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void enroll() {
    pthread_mutex_lock(&lock);
    if (number < class_size)
        number++;
    pthread_mutex_unlock(&lock);
}

void disenroll() {
    pthread_mutex_lock(&lock);
    if (number > 0)
        number--;
    pthread_mutex_unlock(&lock);
}
```

Problem 2:

1.

- cond_usg.c:

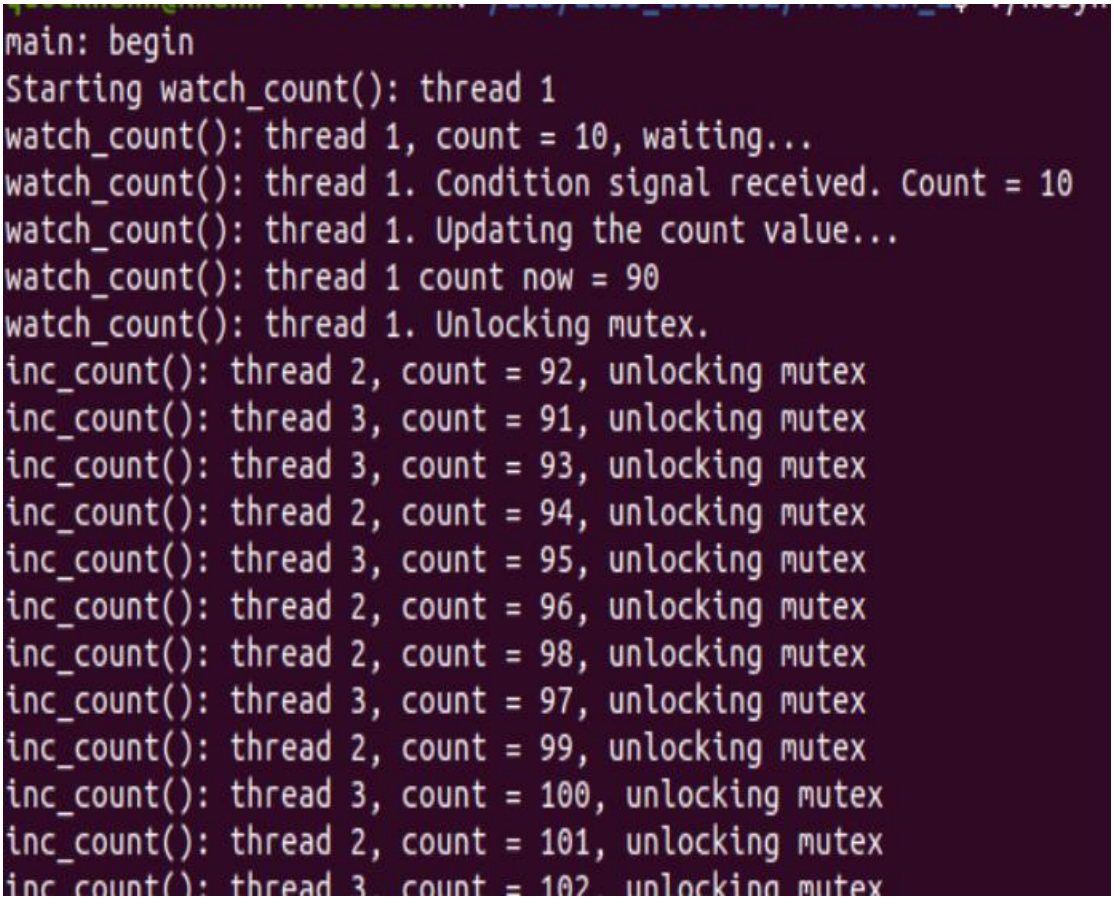
```
main: begin
inc_count(): thread 3, count = 11, unlocking mutex
Starting watch_count(): thread 1
inc_count(): thread 2, count = 12, unlocking mutex
watch_count(): thread 1, count = 12, waiting...
inc_count(): thread 2, count = 13, unlocking mutex
inc_count(): thread 3, count = 14, unlocking mutex
inc_count(): thread 2, count = 15, unlocking mutex
inc_count(): thread 3, count = 16, unlocking mutex
inc_count(): thread 2, count = 17, unlocking mutex
inc_count(): thread 3, count = 18, unlocking mutex
inc_count(): thread 2, count = 19, unlocking mutex
inc_count(): thread 3, count = 20, threshold reached.
Just sent signal
inc_count(): thread 3, count = 20, unlocking mutex
watch_count(): thread 1. Condition signal received. Count = 20
watch_count(): thread 1. Updating the count value...
watch_count(): thread 1 count now = 100
watch_count(): thread 1. Unlocking mutex.
inc_count(): thread 2, count = 101, unlocking mutex
inc_count(): thread 3, count = 102, unlocking mutex
inc_count(): thread 2, count = 103, unlocking mutex
```

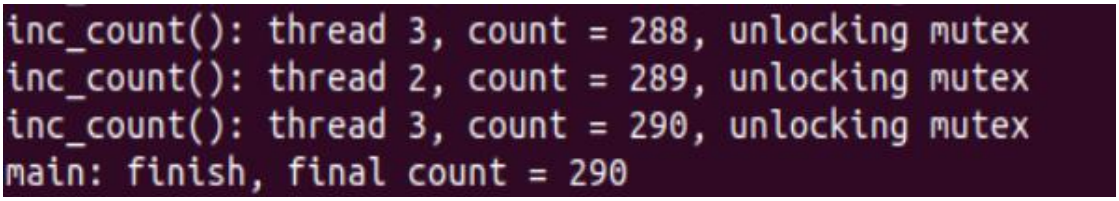
```
inc_count(): thread 2, count = 285, unlocking mutex
inc_count(): thread 3, count = 286, unlocking mutex
inc_count(): thread 2, count = 287, unlocking mutex
inc_count(): thread 3, count = 288, unlocking mutex
inc_count(): thread 2, count = 289, unlocking mutex
inc_count(): thread 3, count = 290, unlocking mutex
main: finish, final count = 290
```

--> Thread 1 sẽ đợi thread 2 và 3 tuần tự cập nhật giá trị của count đến 20 sau đó update nó lên thêm 80 (count now = 100), sau đó kết thúc. Các thread 2 và 3 sẽ tiếp tục cập nhật giá trị cho count đến khi kết thúc vòng lặp của mình. Kết quả trả về final count sẽ luôn bằng 290.

- Nếu như xóa bỏ tất cả các khóa mutex cũng như các tín hiệu chờ, gửi điều kiện, chương trình sẽ thực thi và cho ra:

- nosynch.c:





--> Kết quả trả về final count vẫn sẽ là 290, tuy nhiên ở hàm watch_count(), chương trình sẽ không đợi đến khi count == 20 mà sẽ update ngay với một giá trị count nào đó <= 20 mà nó bắt gặp.

- Ở chương trình này, chúng ta vẫn hạn chế tối thiểu việc gặp phải race condition (gần như không xảy ra) bằng việc gọi hàm sleep(1) trong inc_count() sau mỗi vòng lặp. Thread 2 và thread 3 sẽ chạy tuần tự theo từng giây và gần như không bắt gặp trường hợp 2 thread update count tại cùng 1 thời điểm. Bên cạnh đó, watch_count() vẫn sẽ thực thi được vòng

while do tại thời điểm đang xét count luôn luôn < 20 (phải sau ít nhất 5s count mới có thể update lên >=20 được) nên giá trị count cũng luôn được cập nhật thêm 80 ở watch_count. Do đó, kết quả cuối cùng final count gần như vẫn sẽ luôn luôn là 290.

(Tuy nhiên, nếu bỏ đi câu lệnh sleep sẽ dẫn đến kết quả cuối cùng của count có thể là 210 do thread 2 và 3 có thể update giá trị của count > 20 trước khi thread 1 thực hiện đến vòng lặp while, dẫn đến thiếu hụt 1 lượng 80. Ở trường hợp này, TCOUNT = 100 đủ nhỏ để thread 2 và 3 không có cơ hội thực hiện xen kẽ đồng thời với nhau gây ra race condition. Tuy nhiên, với TCOUNT đủ lớn (VD: 1.000.000), thread 2 và 3 có thể thực thi đồng thời tại 1 thời điểm gây ra race condition! --> đã thử nghiệm và kiểm chứng)

2.

- **count_mutex** để tránh xảy ra race condition khi update giá trị của count giữa thread 1, 2 và 3 cũng như tránh việc lấy sai giá trị của count để xét trong vòng while ở thread 1.
- **count_threshold_cv** là biến điều kiện dùng cho thread 1 nhằm chờ đợi tín hiệu “count == 20” được gửi đến từ thread 2 hoặc 3 rồi mới tiếp tục thực thi công việc.

Problem 3:

- Chúng ta có 2 file: “pi_mutex.c” và “pi_sema.c” đại diện cho 2 cách triển khai code dùng mutex và semaphore

- Thực thi: (làm việc với số lượng thread là 100, số lượng điểm ngẫu nhiên là 100.000.000 điểm)

+ pi_mutex.c:

```
quockhanh@Khanh-VirtualBox:~/Lab/Lab3_2013452/Problem_3$ time ./pi_mutex 100000
000
3.141584

real    0m12,049s
user    0m21,927s
sys     0m25,888s
```

+ pi_sema.c:

```
quockhanh@Khanh-VirtualBox:~/Lab/Lab3_2013452/Problem_3$ time ./pi_sema 1000000
00
3.141570

real    0m14,745s
user    0m18,696s
sys     0m22,064s
```

- So với chương trình tính số Pi đã hiện thực ở Lab 2:

```
quockhanh@Khanh-VirtualBox:~/Lab/Lab2_2013452/Problem_2$ time ./pi_multi-thread
 100000000
3.141536

real    0m1,613s
user    0m6,380s
sys     0m0,032s
```

--> Chúng ta nhận thấy, việc liên tục cập nhật giá trị vào một biến toàn cục khi điểm đang xét thỏa mãn nằm trong đường tròn đòi hỏi phải sử dụng mutex hoặc semaphore để tránh xảy ra race condition. Tuy nhiên, việc này có thể làm giảm hiệu suất của chương trình do tại một thời điểm, chỉ có duy nhất một thread thực thi được critical code, những thread còn lại phải ở trong trạng thái chờ đợi đến lượt thực thi.

- Việc mỗi thread cập nhật giá trị vào các biến riêng biệt như đã hiện thực ở Lab 2 sẽ cắt giảm được khoảng thời gian chờ đợi này, và rõ ràng như chúng ta đã thấy, thời gian thực thi được giảm đi rất nhiều lần.