第三十一章-信号量

预备知识

课后作业 (编程)

在这个作业中，我们将使用信号量来解决一些常见的的并发问题。
其中很多都来自 Downey 的优秀的书籍 “Little Book of Semaphores”3,
它很好地将一些经典问题结合在一起，并引入了一些新的变体；有兴趣的读者应该看看这本小书来获得更多的乐趣。
以下每个问题都提供了一个代码框架；您的工作是完善代码，使其在给定的信号量下工作。
在 Linux 上，您将使用 Linux 系统提供的信号量；在 Mac(不支持信号量)上，您必须首先构建一个信号量实现(使用锁和条件变量，如本章所述)。好运!

查看Makefile文件:

```
FLAGS = -Wall -pthread -g

clean:
    rm -f *.out

b:
    gcc barrier.c $(FLAGS)

fj:
    gcc fork-join.c $(FLAGS)

mn:
    gcc mutex-nostarve.c $(FLAGS)

rw:
    gcc reader-writer.c $(FLAGS)

rwn:
    gcc reader-writer-nostarve.c $(FLAGS)

r:
    gcc rendezvous.c $(FLAGS)
```

参数解释:

`-Wall`: 编译后显示所有警告

`-pthread`: 用于在编译时增加对多线程的支持

`-g`: 为可执行程序添加调试信息，供gdb调试

`clean`: 删除所有.out文件

Problem1

问题描述

第一个问题就是实现和测试 fork/join 问题的解决方案，如本文所述。 即使在文本中描述了此解决方案，重新自己实现一遍也是值得的。
even Bach would rewrite Vivaldi，allowing one soon-to-be master to learn from an existing one。
有关详细信息，请参见 fork-join.c。 将添加 sleep(1) 到 child 函数内以确保其正常工作。

问题解答

```c
// fork-join.c
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include "common_threads.h"
#include <semaphore.h>

sem_t s;

void *child(void *arg) {
    sleep(1);
    printf("child\n");
    sem_post(&s);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    sem_init(&s, 0, 0);
    Pthread_create(&p, NULL, child, NULL);
    sem_wait(&s);
    printf("parent: end\n");
    return 0;
}
```

```
cd code && make fj && ./a.out
gcc fork-join.c -Wall -pthread -g
parent: begin
child
parent: end
```

Problem2

问题描述

现在，我们通过研究集合点问题 rendezvous problem 来对此进行概括。
问题如下：您有两个线程，每个线程将要在代码中进入集合点。 任何一方都不应在另一方进入之前退出代码的这一部分。
该任务使用两个信号量，有关详细信息，请参见 rendezvous.c。

问题解答

```c
// rendezvous.c
#include <stdio.h>
#include <unistd.h>
#include "common_threads.h"
#include <semaphore.h>

// If done correctly, each child should print their "before" message
// before either prints their "after" message. Test by adding sleep(1)
// calls in various locations.

sem_t s1, s2;

void *child_1(void *arg) {
    printf("child 1: before\n");
    sleep(1);
    sem_post(&s2);
    sem_wait(&s1);
    printf("child 1: after\n");
    return NULL;
}

void *child_2(void *arg) {
    printf("child 2: before\n");
    sleep(1);
    sem_post(&s1);
    sem_wait(&s2);
    printf("child 2: after\n");
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    printf("parent: begin\n");
    sem_init(&s1, 0, 0);
    sem_init(&s2, 0, 0);
    Pthread_create(&p1, NULL, child_1, NULL);
    Pthread_create(&p2, NULL, child_2, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("parent: end\n");
    return 0;
}
```

```
cd code && make r && ./a.out
parent: begin
child 1: before
child 2: before
child 2: after
child 1: after
parent: end
```

Problem4

问题描述

现在按照文本中所述，解决读者写者问题。 首先，不用考虑进程饥饿。 有关详细信息，请参见 reader-writer.c 中的代码。
将 sleep（）调用添加到您的代码中，以证明它可以按预期工作。 你能证明饥饿问题的存在吗？

问题解答

```
// reader-writer.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "common_threads.h"
#include <semaphore.h>
//
// Your code goes in the structure and functions below
//

typedef struct __rwlock_t {
    sem_t lock;
    sem_t write_lock;
    int reader_number;
} rwlock_t;


void rwlock_init(rwlock_t *rw) {
    sem_init(&rw->lock, 0, 1);
    sem_init(&rw->write_lock, 0, 1);
    rw->reader_number = 0;
}

void rwlock_acquire_readlock(rwlock_t *rw) {
    sleep(1);
    sem_wait(&rw->lock);
    rw->reader_number++;
    if (rw->reader_number == 1) {
        sem_wait(&rw->write_lock);
    }
    sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock);
    rw->reader_number--;
    if (rw->reader_number == 0) {
        sem_post(&rw->write_lock);
    }
    sem_post(&rw->lock);
}

void rwlock_acquire_writelock(rwlock_t *rw) {
    sleep(1);
    sem_wait(&rw->write_lock);
}
```

```
void rwlock_release_writelock(rwlock_t *rw) {
    sem_post(&rw->write_lock);
}

//
// Don't change the code below (just use it!)
//

int loops;
int value = 0;

rwlock_t lock;

void *reader(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        rwlock_acquire_readlock(&lock);
        printf("read %d\n", value);
        rwlock_release_readlock(&lock);
    }
    return NULL;
}

void *writer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        rwlock_acquire_writelock(&lock);
        value++;
        printf("write %d\n", value);
        rwlock_release_writelock(&lock);
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    assert(argc == 4);
    int num_readers = atoi(argv[1]);
    int num_writers = atoi(argv[2]);
    loops = atoi(argv[3]);

    pthread_t pr[num_readers], pw[num_writers];

    rwlock_init(&lock);

    printf("begin\n");

    int i;
    for (i = 0; i < num_readers; i++)
        Pthread_create(&pr[i], NULL, reader, NULL);
    for (i = 0; i < num_writers; i++)
        Pthread_create(&pw[i], NULL, writer, NULL);

    for (i = 0; i < num_readers; i++)
        Pthread_join(pr[i], NULL);
    for (i = 0; i < num_writers; i++)
        Pthread_join(pw[i], NULL);
    printf("end: value %d\n", value);

    return 0;
}
```

```
cd code && make rw && ./a.out 5 5 10
begin
read 0
read 0
read 0
read 0
write 1
write 2
write 3
read 3
write 4
write 5
read 5
read 5
read 5
write 6
write 7
read 7
write 8
read 8
write 9
write 10
read 10
read 10
read 10
write 11
write 12
read 12
write 13
read 13
write 14
write 15
read 15
read 15
read 15
read 15
write 16
write 17
write 18
write 19
read 19
write 20
read 20
read 20
read 20
read 20
read 20
write 21
write 22
write 23
write 24
write 25
read 25
read 25
read 25
read 25
read 25
write 26
write 27
write 28
write 29
write 30
read 30
read 30
read 30
read 30
read 30
write 31
write 32
write 33
write 34
write 35
read 35
read 35
read 35
read 35
read 35
write 36
write 37
write 38
write 39
write 40
read 40
read 40
read 40
read 40
read 40
write 41
write 42
write 43
write 44
write 45
read 45
read 45
read 45
read 45
read 45
write 46
write 47
write 48
write 49
write 50
end: value 50
```

5,5,10 三个参数分别为读者数，写者数，每个读者、写者进行的读写操作数。
当读者数量远大于写者时，写者可能饿死，读者不需要锁就能进入临界区，只要有一个读者获得锁，其他读者线程就能运行，读者数量可能一直大于0，而写者始终无法获取锁。

Problem5

问题描述

让我们再次看一下读者写者问题，但这一次需要考虑进程饥饿。您如何确保所有读者和写者运行？ 有关详细信息，请参见 reader-writer-nostarve.c。

问题解答

```c
// reader-writer-nostarve.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "common_threads.h"
#include <semaphore.h>
//
// Your code goes in the structure and functions below
//

typedef struct __rwlock_t {
    sem_t lock;
    sem_t write_lock;
    sem_t write_waiting;
    int reader_number;
} rwlock_t;


void rwlock_init(rwlock_t *rw) {
    sem_init(&rw->lock, 0, 1);
    sem_init(&rw->write_lock, 0, 1);
    sem_init(&rw->write_waiting, 0, 1);
    rw->reader_number = 0;
}

void rwlock_acquire_readlock(rwlock_t *rw) {
    sleep(1);
    sem_wait(&rw->write_waiting);
    sem_wait(&rw->lock);
    rw->reader_number++;

    if (rw->reader_number == 1) {
        sem_wait(&rw->write_lock);
    }
    sem_post(&rw->lock);
    sem_post(&rw->write_waiting);

}

void rwlock_release_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock);
    rw->reader_number--;
    if (rw->reader_number == 0) {
        sem_post(&rw->write_lock);
    }
    sem_post(&rw->lock);
}

void rwlock_acquire_writelock(rwlock_t *rw) {
    sleep(1);
    sem_wait(&rw->write_waiting);
    sem_wait(&rw->write_lock);
    sem_post(&rw->write_waiting);
}

void rwlock_release_writelock(rwlock_t *rw) {
    sem_post(&rw->write_lock);
}

//
// Don't change the code below (just use it!)
//

int loops;
int value = 0;

rwlock_t lock;

void *reader(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        rwlock_acquire_readlock(&lock);
        printf("read %d\n", value);
        rwlock_release_readlock(&lock);
    }
    return NULL;
}

void *writer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        rwlock_acquire_writelock(&lock);
        value++;
        printf("write %d\n", value);
        rwlock_release_writelock(&lock);
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    assert(argc == 4);
    int num_readers = atoi(argv[1]);
    int num_writers = atoi(argv[2]);
    loops = atoi(argv[3]);

    pthread_t pr[num_readers], pw[num_writers];

    rwlock_init(&lock);

    printf("begin\n");

    int i;
    for (i = 0; i < num_readers; i++)
```

```
        Pthread_create(&pr[i], NULL, reader, NULL);
    for (i = 0; i < num_writers; i++)
        Pthread_create(&pw[i], NULL, writer, NULL);

    for (i = 0; i < num_readers; i++)
        Pthread_join(pr[i], NULL);
    for (i = 0; i < num_writers; i++)
        Pthread_join(pw[i], NULL);

    printf("end: value %d\n", value);

    return 0;
}
```

```
cd code && make rwn && ./a.out 100 5 10
begin
read 0
...(共100次)
read 0
write 1
write 2
write 3
write 4
write 5
read 5
...(共100次)
read 5
write 6
write 7
write 8
write 9
write 10
read 10
...(共99次)
read 10
write 11
read 11
write 12
write 13
write 14
write 15
read 15
...(共99次)
read 15
write 16
read 16
write 17
write 18
write 19
write 20
read 20
...(共99次)
read 20
write 21
read 21
```

```
write 22
write 23
write 24
write 25
read 25
...(共99次)
read 25
write 26
read 26
write 27
write 28
write 29
write 30
read 30
...(共99次)
read 30
write 31
read 31
write 32
write 33
write 34
write 35
read 35
...(共99次)
read 35
write 36
read 36
write 37
write 38
write 39
write 40
read 40
...(共99次)
read 40
write 41
read 41
write 42
write 43
write 44
write 45
read 45
...(共99次)
read 45
write 46
read 46
write 47
write 48
write 49
write 50
end: value 50
```

新增 write_waiting 锁，写者读者都需要竞争这个锁，正如书上所说读者写者锁看似很酷，事实上意味着复杂与缓慢。

Problem6

问题描述

使用信号量构建一个没有饥饿的互斥量，其中任何试图获取该互斥量的线程都将最终获得它。有关更多信息，请参见 mutex-nostarve.c 中的代码。

问题解答

```c
// mutex-nostarve.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include "common_threads.h"
#include "semaphore.h"
//
// Here, you have to write (almost) ALL the code. Oh no!
// How can you show that a thread does not starve
// when attempting to acquire this mutex you build?
//

typedef struct __ns_mutex_t {
    sem_t mutex;
    sem_t barrier;
    sem_t lock;
    int num_threads;
} ns_mutex_t;

ns_mutex_t m;

void ns_mutex_init(ns_mutex_t *m, int num_threads) {
    sem_init(&m->barrier, 0, -num_threads + 1);
    sem_init(&m->mutex, 0, 1);//互斥锁
}

void ns_mutex_acquire(ns_mutex_t *m) {
    sem_wait(&m->lock);
}

void ns_mutex_release(ns_mutex_t *m) {
    sem_post(&m->lock);

    sem_post(&m->barrier);
    sem_wait(&m->barrier);

    sem_wait(&m->mutex);
    for (int i = 0; i < m->num_threads; i++) {
        sem_post(&m->barrier);
        //确认所有线程都执行过临界区后再继续执行
    }
    sem_post(&m->mutex);
}


void *worker(void *arg) {
    return NULL;
}
```

```c
int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    printf("parent: end\n");
    return 0;
}
```

首先，为什么用互斥量加锁会饿死？比如多线程死循环，反复进入临界区，分别为线程a,b,c,d...
互斥量无法保证a,b,c,d...都会运行，运行序列可能是 a,b,a,a,a,a,b,b,b,...因此其他线程饿死。
实现在 code/mutex-nostarve.c，原理为：释放临界区锁后，调用barrier.c内实现的barrier函数，等待所有线程执行完再继续进行。