HW9

姓名: 陈锐林, 学号:21307130148

2023 年 11 月 3 日

Question1

这题要补全 fork-join.c，根据题目和代码信息；我们要设置信号量，让输出顺序正常 (parent => child => parent)，通过在 child 中 sleep 来保证正确。所以设置信号量的思路即：parent 中 wait，child 完成并 post。代码和运行效果如下：

```c
void *child(void *arg) {
    sleep(1);   //new added
    printf("child\n");
    Sem_post(&s); //new added
    return NULL;
}
```

(a) child

```c
pthread_t p;
printf("parent: begin\n");
Sem_init(&s,0);  //new added
Pthread_create(&p, NULL, child, NULL);
Sem_wait(&s);  //new added
printf("parent: end\n");
```

(b) parent(main)

```
parent: begin
child
parent: end
```

Question2

这题要求实现，每个线程都要在另一个进来之后才能退出。题目也提示了要使用两个信号量；那么就简单了。分别用两个信号量控制两个 child。对于 child1，每当开始工作时，post child2 的信号量；并等待 child2 post 自己的信号量；child2 类似。这里给出代码 (main 函数初始化，略去) 和运行效果：

```
parent: begin
child 1: before
child 2: before
child 2: after
child 1: after
parent: end
```

```c
void *child_1(void *arg) {
    printf("child 1: before\n");
    Sem_post(&s2); //new
    Sem_wait(&s1); //new
    printf("child 1: after\n");
    return NULL;
}

void *child_2(void *arg) {
    printf("child 2: before\n");
    Sem_post(&s1); //new
    Sem_wait(&s2); //new
    printf("child 2: after\n");
    return NULL;
}
```

Question3

这里要实现 barrier 相关的事务；要求达成的效果是在给定线程数的情况下，先对所有线程做事情 A，再做事情 B。为了实现该代码，我们需要在 barrier 中保存线程数, 当前等到的线程数。对于某个线程，我们可以对当前等到的线程数作判断；如果总数不达标，就要调用 wait 等待。当总数够了，就 post。综上，我设置了信号量 t1,t2；总线程数 N，当前队列内线程数 q。t1 用来实现等待所有线程进队，t2 用来保证对 q 的更新不会出错。还需要说明初始化，q 和 N 不必多说；t1 该是 0，因为要保证从第一个线程开始就等待；t2 该是 1，保证每次对 q 的更新都顺利进行并及时 post 回去。代码和运行 (./barrier 2) 结果如下：

```c
typedef struct __barrier_t {
    sem_t t1,t2;
    int N;
    int q;
} barrier_t;

void barrier_init(barrier_t *b,
                  int num_threads) {
    Sem_init(&b->t1,0);
    Sem_init(&b->t2,1);
    b->q = 0;
    b->N = num_threads;
}
```

(a) 结构定义和初始化

```c
void barrier(barrier_t *b) {
    Sem_wait(&b->t2);
    if(b->q < b->N -1){
        b->q++;
        Sem_post(&b->t2);
        Sem_wait(&b->t1);
    }
    Sem_post(&b->t1);
}
```

(b) 函数 barrier

```
parent: begin
child 0: before
child 1: before
child 1: after
child 0: after
parent: end
```

Question4

(1) 参照前面的内容，可以较容易地实现。用 readers 表示读者数，为了保证更新的正确，需要一个信号量 count；为了保证写的唯一，需要信号量 write。为了解决冲突，构造读者优先锁，读者获取锁时，如果他是当前第一个在读的，那就要将 write 占用；读者释放锁时，如果当前只剩他一人了，就要及时放开锁。其他就是维护 readers，写者正常的 wait 和 post。(2) 饿死的情况是很有可能发生的，如果先进来俩个读者，他们赖着不走；这样无论之后来了多少写者，write 将一直为 0，写者难以正常写。如下面的例子，明明是两个循环，但是读者只能读到 0。(3) 代码 (额外添加的 sleep 没显示) 和执行 (./reader writer 3 2 2) 结果如下：

```c
typedef struct __rwlock_t {
    sem_t write;
    sem_t count;
    int readers;
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
    Sem_init(&rw->write,1);
    Sem_init(&rw->count,1);
    rw->readers = 1;
}

void rwlock_acquire_readlock(rwlock_t *rw) {
    Sem_wait(&rw->count);
    if(rw->readers == 0)
        Sem_wait(&rw->write);
    rw->readers++;
    Sem_post(&rw->count);
}
```

(a) part1

```c
void rwlock_release_readlock(rwlock_t *rw) {
    Sem_wait(&rw->count);
    if(rw->readers == 1)
        Sem_post(&rw->write);
    rw->readers--;
    Sem_post(&rw->count);
}

void rwlock_acquire_writelock(rwlock_t *rw) {
    Sem_wait(&rw->write);
}

void rwlock_release_writelock(rwlock_t *rw) {
    Sem_post(&rw->write);
}
```

(b) part2

```
begin
read 0
read 0
read 0
read 0
read 0
read 0
write 1
write 2
write 3
write 4
end: value 4
```

Question5

我们能看到，上面的问题主要是因为读者对写锁 write 的无条件霸占。所以从这点出发，为了让写者可以有作用，我们可以储存排队要写的人的数量，并在读者尝试上锁时判断是不是还有人要写。自然而然，writers 变量的维护也需要一个信息量 wcount。读者现在流程就变成：上锁时，如果有写者在排队，那就要及时让出位置 (post wcount 和 write，再 wait read 和 write)；接着再做和之前类似的事。读者的释放操作可以保持不变，即使 post write 就好。对于写者的操作也有较大变化，当一个写者进来时，首先要对 writers 进行维护，紧接着要刺探是不是能写 (wait write)，写完了要及时把 writers 复原；写者释放时也还需要 post read。代码和运行 (和上面一样的命令) 截图如下：

```c
typedef struct __rwlock_t {
    sem_t rcount;
    sem_t wcount;
    sem_t write;
    sem_t read;
    int readers;
    int writers;
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
    Sem_init(&rw->write, 1);
    Sem_init(&rw->rcount, 1);
    Sem_init(&rw->wcount, 1);
    Sem_init(&rw->read, 0);
    rw->readers = 0;
    rw->writers = 0;
}
```

(a) part1

```c
void rwlock_acquire_readlock(rwlock_t *rw) {
    Sem_wait(&rw->rcount);
    Sem_wait(&rw->wcount);
    if(rw->writers >= 1) {
        Sem_post(&rw->write);
        Sem_post(&rw->wcount);
        Sem_wait(&rw->read);
        Sem_wait(&rw->write);
    } else {
        Sem_post(&rw->wcount);
    }
    if(rw->readers == 0) {
        Sem_wait(&rw->write);
    }
    rw->readers++;
    Sem_post(&rw->rcount);
}
```

(b) part2

```c
void rwlock_release_readlock(rwlock_t *rw) {
    Sem_wait(&rw->rcount);
    if(rw->readers == 1) {
        Sem_post(&rw->write);
    }
    rw->readers--;
    Sem_post(&rw->rcount);
}
void rwlock_acquire_writelock(rwlock_t *rw) {
    Sem_wait(&rw->wcount);
    rw->writers++;
    Sem_post(&rw->wcount);
    Sem_wait(&rw->write);
    Sem_wait(&rw->wcount);
    rw->writers--;
    Sem_post(&rw->wcount);
}
void rwlock_release_writelock(rwlock_t *rw) {
    Sem_post(&rw->read);
    Sem_post(&rw->write);
}
```

(a) part3

```
read 0
read 0
read 0
read 0
write 1
read 1
write 2
read 2
write 3
write 4
end: value 4
```

(b) part4

Question6

只要能保证每个线程都能得到锁并进行正常的工作就行。首先解决怎么展示锁被正常使用，这里直接调用函数 pthread_self() 函数展示被锁前后的线程 ID 即可，会发现输出是正常且对应的。说回到设计，我主要是用了两个信号量和一个标志。标志 value 用来指示目前状态；锁被占用，value 减小，释放则增大。可以看到，为了避免饥饿，在锁释放时对 value 判断，如果 value 还小于 0，说明还有线程卡在了等待阶段。信号量是 mutex 和 turn，初始化为 1/0；首个进程可顺利进行，无需关心 turn。在 acquire 阶段，mutex 作总体控制，保证 value 不会发生问题；但如 value 小于 0，在 acquire 阶段，该线程就等待 turn，直到拿到锁；这是和上面 release 阶段对应的。代码和运行效果如下：

```c
void ns_mutex_acquire(ns_mutex_t *m) {
    Sem_wait(&m->mutex);
    m->value--;
    if (m->value < 0) {
        Sem_post(&m->mutex);
        Sem_wait(&m->turn);
    }
    Sem_post(&m->mutex);
}
```

(a) part1

```c
void ns_mutex_release(ns_mutex_t *m) {
    Sem_wait(&m->mutex);
    m->value++;
    if (m->value <= 0) {
        Sem_post(&m->turn);
    }
    Sem_post(&m->mutex);
}
```

(b) part2

```
Started: 140562588829248
Got lock: 140562588829248
Started: 140562563651136
Got lock: 140562563651136
Started: 140562572043840
Got lock: 140562572043840
Started: 140562546865728
Got lock: 140562546865728
Started: 140562555258432
Got lock: 140562555258432
Started: 140562580436544
Got lock: 140562580436544
Started: 140562468435520
Got lock: 140562468435520
Started: 140562460042816
Got lock: 140562460042816
Started: 140562451650112
Got lock: 140562451650112
Started: 140562443257408
Got lock: 140562443257408
```