گزارش کار آزمایش ششم OS

۱. بله، برنامه در ابتدا ایراددار است؛ تصویری از این ایراد را در زیر می‌بینید.

```
Writer Process, with PID 4298, Count: 167
Writer Process, with PID 4298, Count: 168
 Reader Process, with PID 4300, Count: 167
Writer Process, with PID 4298, Count: 169
 Reader Process, with PID 4300, Count: 169
Writer Process, with PID 4298, Count: 170
 Reader Process, with PID 4300, Count: 170
Writer Process, with PID 4298, Count: 171
 Reader Process, with PID 4300, Count: 171
Writer Process, with PID 4298, Count: 172
 Reader Process, with PID 4300, Count: 172
Writer Process, with PID 4298, Count: 173
 Reader Process, with PID 4300, Count: 173
```

در اجرای بالا، با این که فرآیند Writer مقدار 168 را نوشته، امّا فرآیند Reader مقدار قدیمی count (167) را خوانده و چاپ کرده‌است.
این موضوع به این دلیل است که ممکن است فرآیند Reader یک مقدار را بخواند، اما پیش از آن که بتواند آن را چاپ کند، فرآیند Writer کنترل را دست گرفته، مقدار جدید را نوشته، و کنترل را به Reader بازگرداند. در این صورت Reader کار خود را که چاپ کردن مقدار قدیمی بود، از سر می‌گیرد.

برای حلّ این مشکل از Semaphore استفاده می‌کنیم. با استفاده از Semaphore می‌توانیم مطمئن باشیم داده‌ی مشترک هیچ‌گاه به طور مشترک توسط Readerها و Writer مورد دسترسی قرار نمی‌گیرد.
به عبارت دیگر، هر فرآیند می‌تواند داده را قفل کند، کارش را روی داده انجام دهد، و سپس قفل را آزاد کند. در این صورت دیگر، مشکل بالا رخ نخواهد داد.

توجّه کنید که Semaphore تضمین نمی‌کند که هر داده‌ی جدید، توسط Readerها یک بار خوانده شود؛ بلکه ممکن است بعضی داده‌ها چندین بار خوانده شده، و بعضی داده‌ها اصلا خوانده نشوند.

در صورت سؤال یک فرض دیگر هم شده، آن هم این که فرآیندهای Reader بتوانند هم‌زمان داده‌ی Count را بخوانند. برای برآورده شدن این فرض، نیاز به یک Semaphore دیگر و یک متغیر rc داریم. در متغیر rc تعداد فرآیندهای reader را که در هر لحظه از زمان به مقدار count دسترسی دارند، نگه می‌داریم. پس هر فرآیند reader که شروع به دسترسی به count می‌کند، rc را یک واحد اضافه می‌کند. هر موقع هم که کار آن فرآیند reader با count تمام شد، rc را یک واحد کم می‌کند.
می‌دانیم rc بین فرآیندهای مختلف reader مشترک است، پس برای آن که چندین reader هم‌زمان مقدار rc را تغییر ندهند، برای rc هم یک Semaphore (RC_SEM_NAME) تعریف می‌کنیم.

در ادامه قسمت‌های مهم کد آورده شده است:

```c
// Reader Process
while (*count < MAX_COUNT)
    {
        // printf("first rc: %d", *rc);
        sem_wait(rc_sem);
        (*rc)++;
        if (*rc == 1)
            sem_wait(sem);
        sem_post(rc_sem);

        printf("\x1b[31m\tReader Process, with PID %d, Count: %d\n\x1b[0m", getpid(), *count);

        sem_wait(rc_sem);
        (*rc)--;
        if (*rc == 0)
            sem_post(sem);
        sem_post(rc_sem);
        printf("");
    }
```

```c
// Writer
while (*count < MAX_COUNT)
    {
        sem_wait(sem);
        printf("\x1b[34mWriter Process, with PID %d, Count: %d\n\x1b[0m", getpid(), ++(*count));
        // usleep(1);
        sem_post(sem);
    }
```

نمونه خروجی برنامه‌ی نهایی:

```
Writer Process, with PID 1271, Count: 135
        Reader Process, with PID 1272, Count: 135
        Reader Process, with PID 1273, Count: 135
        Reader Process, with PID 1272, Count: 135
Writer Process, with PID 1271, Count: 136
        Reader Process, with PID 1273, Count: 136
        Reader Process, with PID 1272, Count: 136
        Reader Process, with PID 1273, Count: 136
Writer Process, with PID 1271, Count: 137
        Reader Process, with PID 1272, Count: 137
        Reader Process, with PID 1273, Count: 137
        Reader Process, with PID 1272, Count: 137
Writer Process, with PID 1271, Count: 138
        Reader Process, with PID 1273, Count: 138
        Reader Process, with PID 1272, Count: 138
        Reader Process, with PID 1273, Count: 138
Writer Process, with PID 1271, Count: 139
        Reader Process, with PID 1272, Count: 139
        Reader Process, with PID 1273, Count: 139
        Reader Process, with PID 1272, Count: 139
Writer Process, with PID 1271, Count: 140
        Reader Process, with PID 1273, Count: 140
        Reader Process, with PID 1272, Count: 140
        Reader Process, with PID 1273, Count: 140
        Reader Process, with PID 1272, Count: 140
```

۲. بله، برای مثال سه سناریوی ایجاد بن‌بست این است که هر فیلسوف چوب سمت چپ خود را بردارد. در این صورت هیچ‌یک از فیلسوف‌ها نمی‌توانند غذا بخورند، هیچ‌یک هم چوب خود را زمین نمی‌گذارند.

در پیاده‌سازی انجام شده برای این سوال، امکان بروز سناریوی بالا وجود ندارد، چرا که هر فیلسوف ابتدا آزاد بودن چوب‌های سمت چپ و راست خود را چک کرده، و سپس در صورت امکان آن‌ها را برمی‌دارد.
عمل چک شدن در کد:

```
pthread_mutex_lock(&mutex_chopsticks[left]);
pthread_mutex_lock(&mutex_chopsticks[right]);
if (chopsticks[right] || chopsticks[left])
{
    // give up the chopsticks unless you can take both at once.
    pthread_mutex_unlock(&mutex_chopsticks[left]);
    pthread_mutex_unlock(&mutex_chopsticks[right]);
    printf("\x1b[31mPhilospher %d tried to eat, but failed. One of the chopsticks %d or %d is currently
            used.\n\x1b[0m", i, left, right);
    usleep(random( ) % 1000); // think.
    continue;
}
chopsticks[right] = 1; // take chopsticks.
chopsticks[left] = 1;

pthread_mutex_unlock(&mutex_chopsticks[left]);
pthread_mutex_unlock(&mutex_chopsticks[right]);
```

در صورت آزاد بودن، فیلسوف دو چوب را برداشته، برای یک مدّت زمان رندوم غذا خورده، و سپس چوب‌ها را آزاد می‌کند.

```
printf("\x1b[34mPhilosopher %d is eating using chopstick[%d] and chopstick[%d]\n\x1b[34m", i, left, right);
usleep(random( ) % 500);
printf("Philosopher %d finished eating\n", i);
printf("Philospher %d is thinking\n", i);
pthread_mutex_lock(&mutex_chopsticks[left]); // give up chopsticks.
pthread_mutex_lock(&mutex_chopsticks[right]);
chopsticks[right] = 0;
chopsticks[left] = 0;
pthread_mutex_unlock(&mutex_chopsticks[left]);
pthread_mutex_unlock(&mutex_chopsticks[right]);
usleep(random( ) % 1000);
```

خروجی برنامه در ادامه قابل مشاهده است.

```
Philosopher 1 is eating using chopstick[0] and chopstick[1]
Philospher 0 tried to eat, but failed. One of the chopsticks 4 or 0 is currently used.
Philosopher 4 is eating using chopstick[3] and chopstick[4]
Philospher 2 tried to eat, but failed. One of the chopsticks 1 or 2 is currently used.
Philosopher 1 finished eating
Philospher 3 tried to eat, but failed. One of the chopsticks 2 or 3 is currently used.
Philospher 1 is thinking
Philosopher 1 is eating using chopstick[0] and chopstick[1]
Philospher 0 tried to eat, but failed. One of the chopsticks 4 or 0 is currently used.
Philospher 2 tried to eat, but failed. One of the chopsticks 1 or 2 is currently used.
Philospher 3 tried to eat, but failed. One of the chopsticks 2 or 3 is currently used.
Philosopher 4 finished eating
Philosopher 1 finished eating
Philospher 1 is thinking
Philospher 4 is thinking
Philosopher 3 is eating using chopstick[2] and chopstick[3]
Philosopher 3 finished eating
Philospher 3 is thinking
Philosopher 0 is eating using chopstick[4] and chopstick[0]
Philospher 1 tried to eat, but failed. One of the chopsticks 0 or 1 is currently used.
Philosopher 2 is eating using chopstick[1] and chopstick[2]
Philospher 4 tried to eat, but failed. One of the chopsticks 3 or 4 is currently used.
Philospher 3 tried to eat, but failed. One of the chopsticks 2 or 3 is currently used.
Philosopher 2 finished eating
Philospher 2 is thinking
Philosopher 0 finished eating
Philospher 0 is thinking
Philosopher 3 is eating using chopstick[2] and chopstick[3]
Philospher 2 tried to eat, but failed. One of the chopsticks 1 or 2 is currently used.
Philosopher 3 finished eating
Philospher 3 is thinking
Philosopher 3 is eating using chopstick[2] and chopstick[3]
Philosopher 1 is eating using chopstick[0] and chopstick[1]
```