Introduction to HPC
HW4 Report

姓名: 任一

学号:2018011423

ry18@mails.tsinghua.edu.cn

2020 年 4 月 4 日

实验环境	
操作系统:	Windows10 家庭版 18362.72 Windows Subsystem for Linux
gcc 版本:	gcc version 7.5.0

1 第 1 题

本题主要锻炼了使用互斥量来实现生产者-消费者同步，一共分为 3 小问，我将逐一作答。

1.1 双线程生产者-消费者同步

这一小问的主要思路已经在题面中有很多提示，即令生产者和消费者线程共享一个互斥锁，并且设置一个全局变量判断当前是否有产品可以被消费。生产者发送消息后以及消费者接受消息后，都会调用 break 以退出循环。线程函数代码如下：

```
1    void *Send_msg(void* rank) {
2        long my_rank = (long) rank;
3        while (1) {
4            pthread_mutex_lock(&mutex);
5            if (my_rank == 0) {
6                if (message_available) {
7                    printf("%s\n", message);
8                    pthread_mutex_unlock(&mutex);
9                    break;
10               }
11           } else {
12               sprintf(message, "Hello from producer");
13               message_available = 1;
14               pthread_mutex_unlock(&mutex);
15               break;
16           }
17           pthread_mutex_unlock(&mutex);
18       }
19       return NULL;
20    } /* Send_msg */
```

实验结果如下：

```
nmren@DESKTOP-NV1R0OE:/mnt/d/Tsinghua/2020Spring/HPC/materials/ipp-source/ipp-source-use/HW/HW4/1$ gcc -g -Wall -o main_simple main_simple.c -lpthread
nmren@DESKTOP-NV1R0OE:/mnt/d/Tsinghua/2020Spring/HPC/materials/ipp-source/ipp-source-use/HW/HW4/1$ ./main_simple
Hello from producer
nmren@DESKTOP-NV1R0OE:/mnt/d/Tsinghua/2020Spring/HPC/materials/ipp-source/ipp-source-use/HW/HW4/1$
```

图 1: 第 1 题第 1 小问实验结果

1.2 2k 个线程生产者-消费者同步

若想把上面双进程的生产者消费者同步推广到 2k 个进程之间，需要至少设置 k 个[1] message_available 变量来判断哪个进程可以打印消息。此外，还需要设置一个控制写入的变量，以保证缓冲区不会在消息被对应的偶数号进程接受前，又被新的奇数号进程写入。线程函数代码如下：

[1]为了直接使用线程下标访问的便利性，下面的代码中事实上设置了 2k 个 message_available 变量，只有下标为偶数的有用。

```c
int okToWrite = 1;
void *Send_msg(void* rank) {
   long my_rank = (long) rank;
   while (1) {
      pthread_mutex_lock(&mutex);
      if (my_rank % 2 == 0) {
         if (message_available[my_rank]) {
            printf("%s, in consumer %ld\n", message, my_rank);
            okToWrite = 1;
            pthread_mutex_unlock(&mutex);
            break;
         }
      } else if (okToWrite) {
         sprintf(message, "Hello from producer %ld", my_rank);
         message_available[(my_rank + thread_count - 1) %
            thread_count] = 1;
         okToWrite = 0;
         pthread_mutex_unlock(&mutex);
         break;
      }
      pthread_mutex_unlock(&mutex);
   }
   return NULL;
} /* Send_msg */
```

实验结果如下：

```
nmren@DESKTOP-NV1R0OE:/mnt/d/Tsinghua/2020Spring/HPC/materials/ipp-source/ipp-source-use/HW/HW4/1$ gcc -g -Wall -o main_2k main_2k.c -lpthread
nmren@DESKTOP-NV1R0OE:/mnt/d/Tsinghua/2020Spring/HPC/materials/ipp-source/ipp-source-use/HW/HW4/1$ ./main_2k
Enter thread number(divisible by 2)
8
Hello from producer 1, in consumer 0
Hello from producer 3, in consumer 2
Hello from producer 5, in consumer 4
Hello from producer 7, in consumer 6
nmren@DESKTOP-NV1R0OE:/mnt/d/Tsinghua/2020Spring/HPC/materials/ipp-source/ipp-source-use/HW/HW4/1$
```

图 2: 第 1 题第 2 小问实验结果

1.3 每个线程既是消费者又是生产者

对于更一般的情况，每个线程都是消费者和生产者，我们需要设立一个变量表示缓冲区中目前有无未发送的消息，如果没有未被接收的消息，并且当前进入互斥锁的线程没有发送过消息，则令其发送消息到缓冲区，并且使标记对应的线程可接收消息。如果有未被接受的消息，并且当前进入互斥锁的线程可以接收消息了，则令其接收消息。对于已经既发送过消息也接受过消息的线程，我们使其调用 break 退出循环。线程函数代码如下：

```c
    // indicates whether there're unreceived message in the buffer
    int messageBusy = 0;
    void *Send_msg(void* rank) {
       long my_rank = (long) rank;
       while (1) {
          pthread_mutex_lock(&mutex);
          if (!messageBusy) {
             if (!sent[my_rank]) {
                sent[my_rank] = 1;
                sprintf(message, "Hello from %ld", my_rank);
                message_available[(my_rank + 1) % thread_count] = 1;
                messageBusy = 1;
                if (sent[my_rank] && recved[my_rank]) {
                   pthread_mutex_unlock(&mutex);
                   break;
                }
             }
          } else {
             if (message_available[my_rank]) {
                printf("%s, in %ld\n", message, my_rank);
                recved[my_rank] = 1;
                messageBusy = 0;
                if (sent[my_rank] && recved[my_rank]) {
                   pthread_mutex_unlock(&mutex);
                   break;
                }
             }
          }
          pthread_mutex_unlock(&mutex);
       }

       return NULL;
    }  /* Send_msg */
```

实验结果如下：

```
nmren@DESKTOP-NV1R0OE:/mnt/d/Tsinghua/2020Spring/HPC/materials/ipp-source/ipp-source-use/HW/HW4/1$ gcc -g -Wall -o main_general main_general.c -lpthread
nmren@DESKTOP-NV1R0OE:/mnt/d/Tsinghua/2020Spring/HPC/materials/ipp-source/ipp-source-use/HW/HW4/1$ ./main_general
Enter thread number
5
Hello from 1, in 2
Hello from 2, in 3
Hello from 3, in 4
Hello from 4, in 0
Hello from 0, in 1
nmren@DESKTOP-NV1R0OE:/mnt/d/Tsinghua/2020Spring/HPC/materials/ipp-source/ipp-source-use/HW/HW4/1$
```

图 3: 第 1 题第 3 小问实验结果

2 第 2 题

本题中给出了一系列链表操作，我将逐一指出其问题

2.1 两个 Delete 操作同时执行

如图 4 所示，如果同时执行删除 2 和 5，理想的情况是 head_p 的指针指向 8，2 和 5 直接被删除。可能出现的一种错误情况是，head_p 的 next 指针指向了 5，而 2 的 next 指针指向了 8，这可能就会造成意想不到的错误。

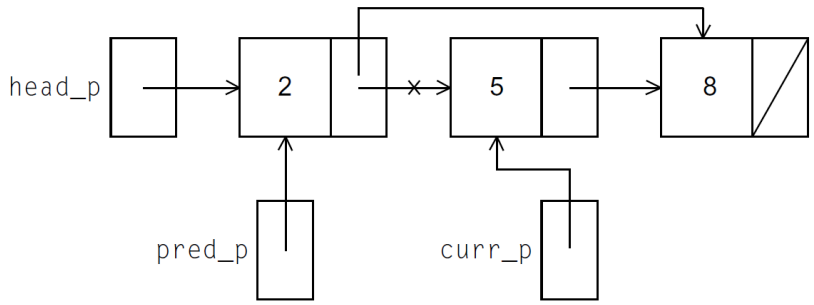

图 4: 删除链表中元素

2.2 一个 Insert 和一个 Delete 操作同时执行

如图 5 所示，如果此时同时删除 5 和插入 7，理想的情况是链表元素为 head_p, 2, 7, 8. 但可能出现的错误情况是：7 正在将 5 作为自己的前继，并且此时 5 已经基本删除完毕，即 2 的 next 指针连接到 8 了，这就造成了 7 的插入无效，这就可能造成一些意想不到的问题。

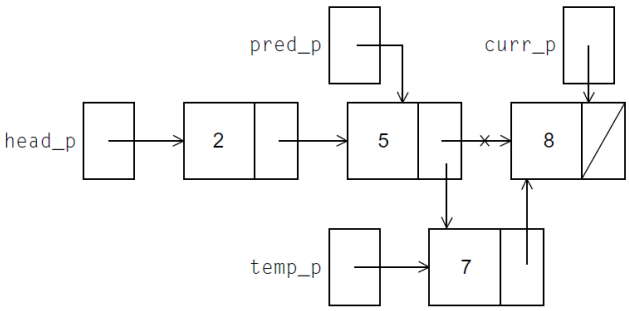

图 5: 插入一个元素

2.3 一个 Member 和一个 Delete 操作同时执行

如图 6 所示，如果此时同时执行 Member(5) 和 Delete(5)，理想情况是删除 5 成功并报告已经没有 5 了。可能造成的错误是，报告了 Member(5) 报告 5 仍然存在，但是 5 已被删除。这就可能造成一些意想不到的问题。

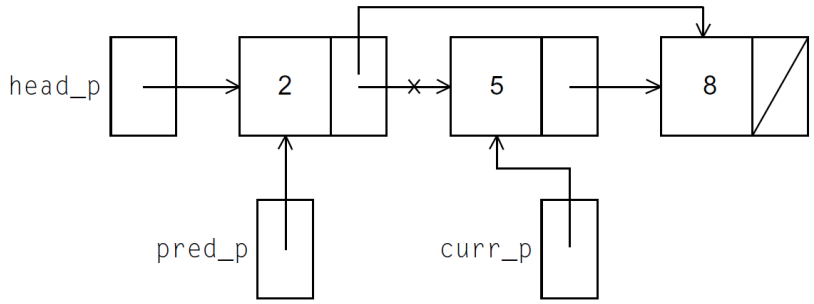

图 6: 删除链表中元素

2.4 两个 Insert 同时执行

如图 7 所示，如果同时想插入 6 和 7，理想情况是操作完毕后，链表中元素为 head_p, 2, 5, 6, 7. 但有可能出现的一种错误情况是，6 先被插入到 5 和 8 之间，但是很快 7 也被插入到 5 和 8 之间，并且 5 的后继被设定为了 7，这样就会导致 6 的插入无效，链表中的元素变为 head_p, 2, 5, 7，这可能就会造成一些意想不到的问题。

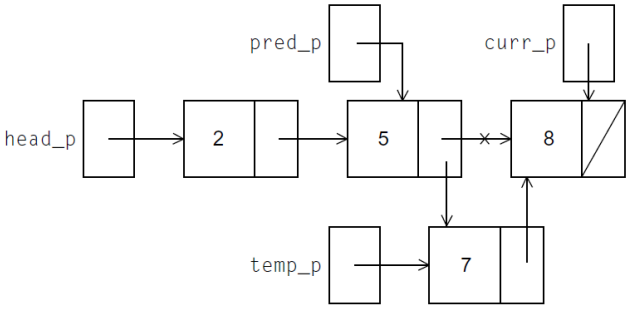

图 7: 插入一个元素

2.5 一个 Insert 和一个 Member 同时执行

如图 8 所示，如果同时执行 Insert(7) 和 Member(7)，理想情况是插入 7 成功并且报告有 7. 一种可能错误的情况是 Member(7) 先执行并且报告没有 7，但 7 已经被插入了。这里就可能造成一些意想不到的问题。

3 第 3 题

这样做是不安全的，可能出现错误。如图 9 所示，如果同时插入 3 和删除 5，在读的阶段两个操作不互斥，则删除 5 和插入 3 操作的 curr_p 都会指向 5。在写的阶段，二者互斥，如果删除 5 先进行了，则 5 的指针就会被释放，此时插入 3，3 的后继仍然会被设定为 5，但 5 已经不存在了，3 无法和后面的 8 连接上，这就会导致错误。

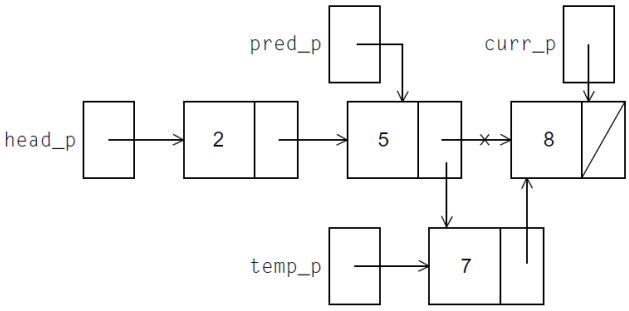

图 8: 插入一个元素

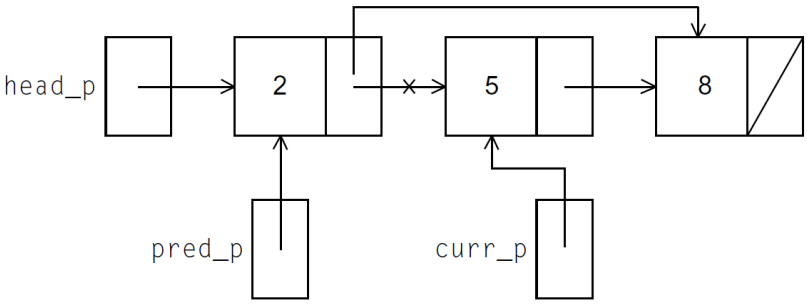

图 9: 删除链表中元素

4 第 4 题

4.1 解题思路

本题需要在主线程创建任务队列，然后让新生成的线程去执行任务，在任务执行结束后结束所有线程并退出。我的主要思路如下：

首先在主线程内输入许多任务，放入一个队列里。[2] 然后生成许多线程，在主线程没有开始逐个遍历任务时，生成的线程都需要挂起等待。接下来主线程逐个循环任务，对于每个任务一次只会有一个线程执行，并且在这个线程执行完任务后，该线程继续回到挂起等待任务的状态，主线程然后才会遍历到下一个任务。当主线程的全部任务已经遍历完后，主线程通知其他线程结束任务并结束线程，释放内存等等。

这里有一些问题值得讨论。例如如何保证每个线程执行完任务后，主线程才可以遍历到下一个任务。这个问题我是通过条件变量来解决的，我会在主线程的任务循环里加入一个条件变量，只有每个线程执行完任务并发送信号后，主线程才能继续进行下一个任务循环。此外，我还在线程函数中使用了条件变量，只有主线程发送信号时，才会有 1 个线程领取任务并执行，当主线程发送 broadcast 信号并标记没有任务时，所有线程都会退出。这里需要注意的一个问题是，当主线程发送 broadcast 信号并标记没有任务时，如果有线程正在执行任务，就可能无法正常退出。我的解决方法是在条件等待前加入是否有任务的判断，如果没有任务了，正在执行前一个任务的线程返回到条件等待前，就可以通过没有任务的判断退出。

本题的链表部分实现，主要参考了课本示例代码中 ch4/linked_list.c 中的链表实现方式，线程

[2]这一步可以手动输入任务编号，也可以直接输入 make check 进行随机生成并与串行程序对拍。

的操作部分均由我设计并实现。

4.2 运行方式

在目录下运行 make 即可进行编译。运行 make run 命令即可运行并行链表操作的程序，不过这种运行方式需要手动输入任务序列以及输入链表的操作。更方便并且易于对比并行串行结果的运行方式是输入 make check, 该命令会自动调用 check.py 以及 datamaker.py, 随机生成输入数据并通过对比整个链表操作过程中的输出来验证正确性。具体参数如线程数、任务数在 makefile 中有对应变量可以更改，若想更改任务数，需要将 makefile 和 datamaker.py 中的 TASKS_COUNT 变量统一更改为 1 个值。默认的任务数为 60，线程数为 10.

4.3 测试结果

使用 10 线程，15 个任务随机输入，对拍 2000 余组数据均正确。

在本题中，我着实感受到了不同线程之间异步执行的复杂性，这需要非常周密的考虑和非常全面的测试。

5 第 5 题

5.1 解题思路

本题需要通过 Pthreads 的并行化方式，计算斐波那契数列第 n 项。我的思路如下。

斐波那契数列数列计算公式为 $fib(n+2) = fib(n+1) + fib(n), fib(0) = 0, fib(1) = 1$. 但是这样的公式不便于并行化，因为这样计算的话，每一项都依赖于前一项的结果，这是一个典型的串行思路。因此我选择了另外一种计算斐波那契数列的等价公式，即：

$$\begin{pmatrix} f_{n+1} \\ f_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f_n \\ f_{n-1} \end{pmatrix} \quad \begin{pmatrix} f_n \\ f_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} f_1 \\ f_0 \end{pmatrix}$$

由于 f_1, f_2 已知，我们只需要计算 $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$ 即可得到结果。注意到同一个矩阵的 n 次方的操作是可交换的，我们就可以利用这个性质，使用类似全局求和的思路，巧妙而简便地计算出 f_n.

具体实现思路如下：首先输入线程数和要计算的项数，然后将待计算的项数尽量平均分配给每个线程。之后在线程函数中，先令每个线程计算其分配到的局部矩阵乘方数，然后通过互斥锁，将局部的矩阵乘方结果乘到全局的矩阵乘方结果上，待所有线程计算完毕后，在主线程中读取结果。

这里有个小问题需要注意，即主进程要在所有线程计算完毕后才能读取结果，否则会导致错误。我的解决方式是：在主进程中加入循环，循环里放入条件变量，并且每个线程结束后发送信号，只有当完成线程函数的线程数达到总线程数时，主线程才会继续进行。这样就可以保证所有线程计算时，主线程不继续进行，只有所有线程计算完毕后主线程才可以继续进行。

5.2 运行方式

在 5 文件夹下运行 make 命令即可编译程序，make run 命令即可运行程序，make check 命令即可随机生成输入数据，使程序进行对拍。

5.3 结果分析

对拍 1000 余组的数据，并行结果与串行结果均一致，保证了程序的正确性。不过由于 long long 的数据范围较为有限，$fib(92) \approx 7 \times 10^{18}$ 就已经接近溢出了，因此在这样小的数据范围下，据我观察并行结果都要比串行结果慢很多。例如 10 线程计算 $fib(92)$ 时，并行计算时间约为 $1.6 \times 10^{-5}s$, 串行计算时间为 $1.2 \times 10^{-6}s$. 同样，由于 long long 的数据范围有限，不便于做更大规模的测试。[3]

6 总结

在本次实验中，我学习到了 Pthreads 编程的方法和技巧，也深刻地体会了多线程之间异步执行的复杂性，锻炼了较为缜密的思维以及编程和调试的能力。感谢老师和助教给予的帮助！

[3]数值溢出时，并行与串行的结果也是一致的，并且较大范围内的数据下能够体现并行的速度优势。例如 20 线程计算 $fib(100000)$ 时，并行计算时间约为 $5.1 \times 10^{-5}s$，串行时间约为 $3.8 \times 10^{-4}s$。但这样的测试可靠性有待商榷，因此不做详细分析。