OS 第三章编程作业

191220008 陈南疃

一、算法并行性问题

根据若干次尝试的结果，我将排序的元素数量设置为1000000，且每个数均为随机生成数。

观察非多线程版本和多线程版本的时间开销。

1、归并排序：非多线程版本

```
cnt@ubuntu:~/os2021/chapter3/Ex1$ time ./mergesort

real    0m0.331s
user    0m0.289s
sys     0m0.040s
cnt@ubuntu:~/os2021/chapter3/Ex1$
```

2、归并排序：简单的多线程版本-串行合并

```
cnt@ubuntu:~/os2021/chapter3/Ex1$ time ./parallel_mergesort -t 0

real    0m0.363s
user    0m0.301s
sys     0m0.052s
cnt@ubuntu:~/os2021/chapter3/Ex1$ time ./parallel_mergesort -t 1

real    0m0.210s
user    0m0.162s
sys     0m0.040s
cnt@ubuntu:~/os2021/chapter3/Ex1$ time ./parallel_mergesort -t 2

real    0m0.120s
user    0m0.092s
sys     0m0.017s
cnt@ubuntu:~/os2021/chapter3/Ex1$ time ./parallel_mergesort -t 3

real    0m0.073s
user    0m0.060s
sys     0m0.012s
cnt@ubuntu:~/os2021/chapter3/Ex1$ time ./parallel_mergesort -t 4

real    0m0.050s
user    0m0.039s
sys     0m0.009s
cnt@ubuntu:~/os2021/chapter3/Ex1$ time ./parallel_mergesort -t 5

real    0m0.037s
user    0m0.027s
sys     0m0.005s
cnt@ubuntu:~/os2021/chapter3/Ex1$ time ./parallel_mergesort -t 6

real    0m0.025s
user    0m0.020s
sys     0m0.004s
```

```
cnt@ubuntu:~/os2021/chapter3/Ex1$ time ./parallel_mergesort -t 7

real    0m0.022s
user    0m0.014s
sys     0m0.007s
cnt@ubuntu:~/os2021/chapter3/Ex1$ time ./parallel_mergesort -t 8

real    0m0.020s
user    0m0.019s
sys     0m0.000s
cnt@ubuntu:~/os2021/chapter3/Ex1$ time ./parallel_mergesort -t 9

real    0m0.020s
user    0m0.004s
sys     0m0.015s
cnt@ubuntu:~/os2021/chapter3/Ex1$ time ./parallel_mergesort -t 10

real    0m0.018s
user    0m0.013s
sys     0m0.005s
cnt@ubuntu:~/os2021/chapter3/Ex1$ time ./parallel_mergesort -t 11

real    0m0.028s
user    0m0.015s
sys     0m0.006s
cnt@ubuntu:~/os2021/chapter3/Ex1$ time ./parallel_mergesort -t 12

real    0m0.020s
user    0m0.012s
sys     0m0.008s
cnt@ubuntu:~/os2021/chapter3/Ex1$ time ./parallel_mergesort -t 13

real    0m0.020s
user    0m0.011s
sys     0m0.008s
```

在非多线程版本下，输入的最后一个参数是对最大线程数目n的设置。

通过观察可以发现：

- 当n=0时，时间开销与非多线程版本基本一致（因为n=0时，我的代码将不会创建多线程，默认使用非多线程的函数调用）。
- 当n=1~8时，时间开销为递减趋势，符合多线程的特征。
- 当n>8时，时间开销的递减趋势比较缓慢，不太明显。

（因为机器的总逻辑核数为8，当n<=8时，每增加一个线程，几乎就多一个核，时间开销比较明显；当n>8时，核被占满，递增幅度大大降低。）

二、信号量与PV操作实现同步问题

我分别用PV操作核锁实现了**读者优先**的读者写者的同步。

1、使用pthread提供的信号量与PV操作，实现共享数据上读者-写者线程间的正确同步

```
cnt@ubuntu:~/os2021/chapter3/Ex2$ ./readerwriter_pv
写者请求写文件
写者写文件中
读者请求读文件
读者请求读文件
读者请求读文件
读者请求读文件
读者请求读文件
写者停止写文件
读者读文件中，共有1个读者
读者读文件中，共有2个读者
读者读文件中，共有3个读者
读者读文件中，共有4个读者
读者读文件中，共有5个读者
读者停止读文件，共有4个读者
读者停止读文件，共有3个读者
读者停止读文件，共有2个读者
读者停止读文件，共有1个读者
读者停止读文件，共有0个读者
写者请求写文件
写者写文件中
写者停止写文件
^C
```

我设置了五个读者和一个写者，写者每隔一段时间发起写请求。

首先，写者发起写文件请求，由于没有读者在读，因此可以写文件。

在写文件的过程中，有五个读者发起读文件请求，由于代码是读者优先的，因此当写者结束写文件时，读者优先开始读文件。

在读者读文件的过程中，写者无法写文件。

当所有读者均结束读文件后，写者才能开始写文件。

如图，结果与分析完全一致，因此，读者优先的读者写者同步问题实现正确。

2、使用pthread提供的读者-写者锁，实现共享数据上读者-写者线程间的正确同步

```
cnt@ubuntu:~/os2021/chapter3/Ex2$ ./readerwriter_lock
写者请求写文件
写者写文件中
读者请求读文件
读者请求读文件
读者请求读文件
读者请求读文件
读者请求读文件
写者停止写文件
读者读文件中，共有1个读者
读者读文件中，共有2个读者
读者读文件中，共有3个读者
读者读文件中，共有4个读者
读者读文件中，共有5个读者
读者停止读文件，共有4个读者
读者停止读文件，共有3个读者
读者停止读文件，共有2个读者
读者停止读文件，共有1个读者
读者停止读文件，共有0个读者
写者请求写文件
写者写文件中
写者停止写文件
^C
```

结果与PV操作的实现一致，说明读者写者锁实现正确。

三、管程机制实现与管程应用问题

1、只能使用pthread库提供的一般信号量(semaphore)，参考课堂上介绍的使用信号量与PV操作实现Hoare类型管程，来实现CircleBuffer

```
cnt@ubuntu:~/os2021/chapter3/Ex3$ ./circlebuffer_hoare
5
缓冲区为空，等待添加中
将2加入缓冲区buffer[0]
从缓冲区buffer[0]中拿出2
缓冲区为空，等待添加中
将4加入缓冲区buffer[1]
从缓冲区buffer[1]中拿出4
缓冲区为空，等待添加中
将4加入缓冲区buffer[2]
从缓冲区buffer[2]中拿出4
缓冲区为空，等待添加中
将0加入缓冲区buffer[3]
从缓冲区buffer[3]中拿出0
将1加入缓冲区buffer[4]
从缓冲区buffer[4]中拿出1
将3加入缓冲区buffer[0]
从缓冲区buffer[0]中拿出3
将0加入缓冲区buffer[1]
从缓冲区buffer[1]中拿出0
将1加入缓冲区buffer[2]
从缓冲区buffer[2]中拿出1
将4加入缓冲区buffer[3]
从缓冲区buffer[3]中拿出4
缓冲区为空，等待添加中
将4加入缓冲区buffer[4]
从缓冲区buffer[4]中拿出4
缓冲区为空，等待添加中
将3加入缓冲区buffer[0]
从缓冲区buffer[0]中拿出3
缓冲区为空，等待添加中
将3加入缓冲区buffer[1]
从缓冲区buffer[1]中拿出3
缓冲区为空，等待添加中
将4加入缓冲区buffer[2]
从缓冲区buffer[2]中拿出4
^C
```

第一行输入缓冲区的大小。

随机产生0~4的数，加入缓冲区，然后拿出缓冲区。

如图，加入和拿出交替进行，显然，得到了正确的同步。

2、使用pthread库提供的互斥信号量(mutex)和条件变量(condition)，实现CircleBuffer

```
cnt@ubuntu:~/os2021/chapter3/Ex3$ ./circlebuffer_mutexcondition
5
缓冲区为空，等待添加中
将3加入缓冲区buffer[0]
从缓冲区buffer[0]中拿出3
将1加入缓冲区buffer[1]
从缓冲区buffer[1]中拿出1
将1加入缓冲区buffer[2]
从缓冲区buffer[2]中拿出1
将2加入缓冲区buffer[3]
从缓冲区buffer[3]中拿出2
将3加入缓冲区buffer[4]
从缓冲区buffer[4]中拿出3
将1加入缓冲区buffer[0]
从缓冲区buffer[0]中拿出1
将2加入缓冲区buffer[1]
从缓冲区buffer[1]中拿出2
^C
```

如图，结果与1中类似，加入和拿出交替进行，显然，得到了正确的同步。

四、死锁问题

1、使用pthread提供的信号量与PV操作，实现一个多线程程序，使得多个线程在运行中发生死锁

```
cnt@ubuntu:~/os2021/chapter3/Ex4$ ./deadlock_pv
哲学家0思考中
哲学家1思考中
哲学家2思考中
哲学家3思考中
哲学家4思考中
哲学家4拿起左边的叉子
哲学家2拿起左边的叉子
哲学家1拿起左边的叉子
哲学家0拿起左边的叉子
哲学家3拿起左边的叉子
^C
```

我采用了哲学家进餐问题的写法：

当五位哲学家均拿起了左边的叉子后，无法再拿起右边的叉子，所以将会永久等待，发生死锁。

如图，当五位哲学家都拿起左边的叉子后，发生永久等待，因此结果与分析一致，说明实现正确

五、思考与感悟

这次的编程作业量较大，但覆盖的知识点比较全面，通过代码实现，加深了对第三章知识点的理解，同时也对c和c++中多线程的相关实现有了进一步的了解，收获颇多。

此外，对于测试方法的构造有一定的讲究，如果测试方法有误，则结果可能不能让人满意，或者是不能够体现同步的正确性，因此需要对相关知识点有较为深刻的理解才能构造出合理的测试方法。