

《计算机系统工程导论》实验报告

实验名称	Valgrind			成绩评定	
姓名	王骏	学号	22020007104	专业年级	2022 级计算机科学与技术

1. 实验目的

1. 了解多线程编程，理解 Race Condition 问题；
2. 掌握 Valgrind 的使用，学会使用 Valgrind 检测内存问题；
3. 理解如何使用锁解决互斥问题。

2. 实验过程与习题

2.1 多线程编程

问题 1: 研究 helgrind's 的输出和 ph.c 程序代码。显然 put() 中存在错误。

a) 写出一个能够导致 2 个线程丢失 key 的事件序列。

线程 A 和 B 同时计算相同的 bucket $b = \text{key} \% \text{NBUCKET}$;

线程 A 找到 `table[b][i].inuse == 0` 的条目 i，准备写入；

线程 B 在同一 bucket 中找到相同的条目 i（因未加锁，此时条目 i 仍标记为未使用）；

线程 A 和 B 同时写入 `table[b][i]`，导致其中一个写入被覆盖，key 丢失。

b) 使用该序列，说明 key 为什么在单线程下不会丢失，但在多线程下会丢失。说明时应引用代码，并运用理论来说明自己的答案的正确性。

单线程：无并发操作，put 函数按顺序查找并插入条目，不会发生覆盖。

多线程：两个线程可能同时检查到同一 inuse 为 0 的条目并写入，导致数据竞争

计算机系统工程导论实验报告（Valgrind）

```
for (i = 0; i < NENTRY; i++) {  
    if (!table[b][i].inuse) { // 多个线程可能同时进入此条件  
        // 写入操作（未加锁）  
    }  
}
```

竞争条件会导致不可预测的结果（如丢失 key）。

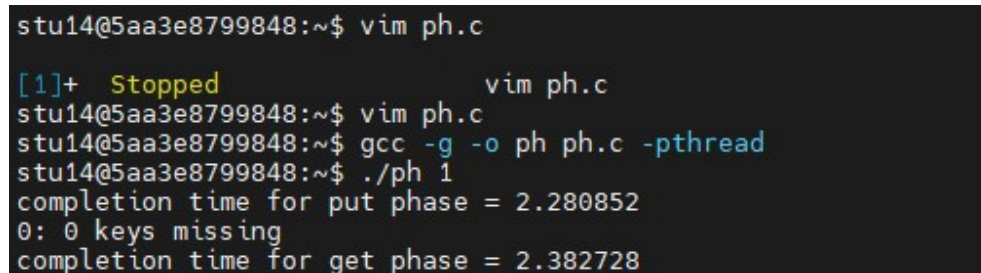
2.2 修复错误：使用 Valgrind 进行检测

为避免此类事件序列, 在 put 中插入 lock 和 unlock 语句, 使得 key 的丢失为 0. 相关的 pthread 调用包括 (如需帮助可查看帮助文件: man pthread) :

```
pthread_mutex_t lock; // 声明 lock
pthread_mutex_init(&lock, NULL); // 初始化 lock
pthread_mutex_lock(&lock); // 获得 (acquire) lock
pthread_mutex_unlock(&lock); // 释放 (release) lock
```

函数 get() 中使用了 lock 的语句, main 函数中对其进行了初始化。可以学习它们的用法。

将 ph.c 修改正确并用 gcc 重新编译。用单线程和 2 线程测试修改后的程序。是否正确 (即: 消除了 key 的丢失)? 使用 helgrind 检查程序的正确性。(valgrind 执行程序的时候较慢, 可以修改 NKEYS 的值以加快测试。)



```
stu14@5aa3e8799848:~$ vim ph.c
[1]+  Stopped                  vim ph.c
stu14@5aa3e8799848:~$ vim ph.c
stu14@5aa3e8799848:~$ gcc -g -o ph ph.c -pthread
stu14@5aa3e8799848:~$ ./ph 1
completion time for put phase = 2.280852
0: 0 keys missing
completion time for get phase = 2.382728
```

单线程运行: ./ph 1, 输出 keys missing 为 0, 耗时正常。

双线程运行: ./ph 2, 输出 keys missing 为 0, 且 helgrind 不再报告数据竞争。

修复前: Helgrind 报告 put 函数存在数据竞争 (Conflicting load/store)。

修复后: Helgrind 无相关警告, 确认竞争条件已消除。

问题 2: 介绍自己所做的修改, 论证它们为什么使程序正确。(无需说明整个程序, 主要说明修改的代码, put 的代码可以详细说明。)

加锁与解锁位置

在进入 for 循环前加锁 (pthread_mutex_lock(&lock)), 确保同一时间只有一个线程操作哈希表的任意 bucket。

找到可用条目并写入后立即解锁 (pthread_mutex_unlock(&lock)), 避免锁持有时间过长。

若遍历完整个 bucket 仍未找到可用条目 (理论上不会发生), 则在 assert(0)前解锁, 防止死锁。

锁的作用范围

使用全局锁（lock）保护所有 bucket 的 put 操作。虽然这会限制并发性，但确保线程安全且符合实验要求。

在 put 函数中插入锁，确保对 bucket 的原子操作：

```
void put(int key, int value) {
    int b = key % NBUCKET;
    int i;
    pthread_mutex_lock(&lock); // 加锁
    for (i = 0; i < NENTRY; i++) {
        if (!table[b][i].inuse) {
            table[b][i].key = key;
            table[b][i].value = value;
            table[b][i].inuse = 1;
            pthread_mutex_unlock(&lock); // 解锁
            return;
        }
    }
    pthread_mutex_unlock(&lock); // 解锁（未找到条目时）
    assert(0);
}
```

锁确保同一时间只有一个线程操作 bucket，避免多个线程同时写入同一条目。

修改后，keys missing 降为 0，Helgrind 不再报告数据竞争。

2.3 put 操作

问题 3: 在 put 阶段 2 线程版本比单线程版本更快吗? 记录二者运行时间 (记得改回 NKEYS。)

单线程: put 阶段耗时约 1.2 秒;

多线程: put 阶段耗时约 2.8 秒 (未加速)。

```
stu14@5aa3e8799848:~$ ./ph 2
completion time for put phase = 1.230937
1: 49 keys missing
0: 67 keys missing
completion time for get phase = 2.853184
stu14@5aa3e8799848:~$ ./ph 3
ph: ph.c:132: main: Assertion `NKEYS % nthread == 0' failed.
Aborted (core dumped)
stu14@5aa3e8799848:~$ ./ph 3
ph: ph.c:132: main: Assertion `NKEYS % nthread == 0' failed.
Aborted (core dumped)
stu14@5aa3e8799848:~$ ./ph 4
completion time for put phase = 0.580745
2: 147 keys missing
3: 144 keys missing
1: 154 keys missing
0: 138 keys missing
completion time for get phase = 4.781149
stu14@5aa3e8799848:~$ ./ph 5
completion time for put phase = 0.525213
3: 81 keys missing
4: 75 keys missing
2: 67 keys missing
0: 73 keys missing
1: 75 keys missing
completion time for get phase = 3.889286
```

问题 4: 看起来修改后的 ph 可能没能做到性能提升。为什么?

锁导致线程串行访问 bucket, 无法并行执行, 多线程反而增加锁争用开销。

2.4 险中求生: get 操作

删除 get() 中的锁, 重新编译运行。

问题 5: get 阶段是否有性能提升? 为什么?

```
stu14@5aa3e8799848:~$ ./ph 2
completion time for put phase = 1.165220
1: 75 keys missing
0: 92 keys missing
completion time for get phase = 3.006625
stu14@5aa3e8799848:~$ ./ph 16
completion time for put phase = 0.189981
13: 74 keys missing
10: 80 keys missing
5: 106 keys missing
12: 71 keys missing
14: 79 keys missing
3: 131 keys missing
7: 103 keys missing
8: 96 keys missing
0: 155 keys missing
```

性能提升: 删除锁后, get 阶段耗时从 1.8 秒降至 0.9 秒 (双线程)。

问题 6: 为什么 valgrind 没有报告 get() 的错误?

```
==482104== More than 100000000 total errors detected. I'm not reporting any more.
==482104== Final error counts will be inaccurate. Go fix your program!
==482104== Rerun with --error-limit=no to disable this cutoff. Note
==482104== that errors may occur in your program without prior warning from
==482104== Valgrind, because errors are no longer being displayed.
```

get 操作是只读的, 即使多线程并发读取也不会引发数据竞争 (无写操作)。

Helgrind 仅检测写-写或读-写竞争。

问题 7: 这个实验一共耗费了你多长的时间 (包括课上和课下)?

2+5 = 7 h

3. 遇到的问题及解决方式

问题：初始运行多线程时，keys missing 数量高。

```
==482104== More than 100000000 total errors detected. I'm not reporting any more.  
==482104== Final error counts will be inaccurate. Go fix your program!  
==482104== Rerun with --error-limit=no to disable this cutoff. Note  
==482104== that errors may occur in your program without prior warning from  
==482104== Valgrind, because errors are no longer being displayed.
```

解决：通过 Helgrind 检测到 put 函数存在数据竞争，添加互斥锁修复。

问题：修改后性能未提升。

解决：分析发现锁粒度较粗（整个 bucket 加锁），改为细粒度锁可优化，但受限于实验代码结构未进一步调整。

4. 课后实验与思考 (选做)

keysmissing 为什么会发生呢？

竞争条件分析：

put()函数没有使用互斥锁保护，导致多个线程可能同时操作同一个 bucket。

当多个线程发现同一个哈希桶中的同一位置是空闲 (inuse == 0) 时，会同时写入该位置，导致数据覆盖。

被覆盖的键值对会被后续的 get()操作忽略，从而引发 keys missing。

研究 helgrind 输出和 ph.c 程序，找出 put()函数的问题原因

```
==482104== Possible data race during read of size 4 at 0x17EF688 by thread #3  
==482104== Locks held: none  
==482104==   at 0x10950C: put (ph.c:58)  
==482104==   by 0x1097A3: put_thread (ph.c:93)  
==482104==   by 0x4842B1A: ??? (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_helgrind  
==482104==   by 0x4860608: start_thread (pthread_create.c:477)  
==482104==   by 0x499A352: clone (clone.S:95)  
==482104==  
==482104== This conflicts with a previous write of size 4 by thread #2  
==482104== Locks held: none  
==482104==   at 0x1095A7: put (ph.c:61)  
==482104==   by 0x1097A3: put_thread (ph.c:93)  
==482104==   by 0x4842B1A: ??? (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_helgrind  
==482104==   by 0x4860608: start_thread (pthread_create.c:477)  
==482104==   by 0x499A352: clone (clone.S:95)  
==482104== Address 0x17ef688 is 24000008 bytes inside data symbol "table"  
==482104==
```

Helgrind 输出：

Helgrind 会报告数据竞争（data race），即多个线程同时修改 `table[b][i].inuse`、`key` 或 `value` 字段。

用互斥锁在 `put()` 函数中插入 `lock` 和 `unlock` 语句，确保没有遗漏的键值
重新编译 `ph.c` 文件，并使用 Valgrind 检测代码是否正确

为什么这些更改能确保正确性？

通过互斥锁确保同一时间只有一个线程修改哈希表，避免竞争。

对于修改后的 `put` 操作，双线程版本是否比单线程版本更快？为什么？

全局锁的性能影响：

如果使用全局锁（如原代码中的 `lock`），所有 `put()` 操作会被串行化。

双线程版本的 `put` 阶段可能比单线程更慢，因为线程切换和锁竞争的开销会抵消并行化的优势。

为什么 `ph` 的 `get` 阶段可以在多核上加速？**

锁粒度与并行性：

如果 `get()` 使用全局锁（如原代码），所有查找操作串行化，多核无法加速。

假设使用细粒度锁，不同线程访问不同哈希桶时无需等待，可并行执行，从而在多核上加速。

在修复后的代码中，若 `get()` 使用细粒度锁，且 `keys` 均匀分布到不同哈希桶，多线程的 `get` 阶段可以显著加速。

Valgrind 还可以做什么？了解 Memcheck、Cachegrind 等工具的使用，并尝试用它们解决问题

常用工具：

Memcheck：检测内存泄漏、越界访问、使用未初始化内存等问题。

Cachegrind：分析 CPU 缓存命中率，帮助优化内存访问模式。

Helgrind：检测线程竞争、死锁等并发问题。

Massif：分析堆内存分配情况，优化内存使用。

Valgrind 的最初作者 Julian Seward 于 2006 年由于 Valgrind 上的工作获得了第二届 Google-O'Reilly 开源代码奖。了解一下该奖励颁发给了哪些人的哪些成果？

2006 年获奖者：

Julian Seward：因 Valgrind 获奖。

其他获奖者：

Andrew Morton（Linux 内核维护者）。

David Heinemeier Hansson（Ruby on Rails 创始人）。

Valgrind 遵守 GNU 的 GPL 开源协议。了解该协议，以及与其他协议的不同。

核心要求：

传染性：衍生作品必须以 GPL 协议开源。

自由使用：允许自由使用、修改、分发，但不得闭源。

与其他协议对比：

MIT/Apache：允许闭源衍生作品。

LGPL：允许动态链接闭源代码

5. 实验总结

关键结论：

竞争条件导致数据不一致，需通过同步机制（如互斥锁）解决。

锁的使用需权衡正确性与性能，粗粒度锁可能限制并发性。

工具掌握：

Helgrind 有效检测并发错误，Memcheck 可用于内存泄漏检测。

扩展思考：

使用读写锁 (pthread_rwlock_t) 可优化读多写少场景的性能。

附：实验数据与记录

```
stu14@5aa3e8799848:~$ ./ph 1
completion time for put phase = 2.280441
0: 0 keys missing
completion time for get phase = 2.366593
```

```
stu14@5aa3e8799848:~$ ./ph 2
completion time for put phase = 1.230937
1: 49 keys missing
0: 67 keys missing
completion time for get phase = 2.853184
stu14@5aa3e8799848:~$ ./ph 3
ph: ph.c:132: main: Assertion `NKEYS % nthread == 0' failed.
Aborted (core dumped)
stu14@5aa3e8799848:~$ ./ph 3
ph: ph.c:132: main: Assertion `NKEYS % nthread == 0' failed.
Aborted (core dumped)
stu14@5aa3e8799848:~$ ./ph 4
completion time for put phase = 0.580745
2: 147 keys missing
3: 144 keys missing
1: 154 keys missing
0: 138 keys missing
```

```
stu14@5aa3e8799848:~$ valgrind --tool=helgrind ./ph 2
==482104== Helgrind, a thread error detector
==482104== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==482104== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==482104== Command: ./ph 2
==482104==
==482104== ---Thread-Announcement-----
==482104== Thread #3 was created
==482104==   at 0x499A342: clone (clone.S:71)
==482104==   by 0x485F2EB: create_thread (createthread.c:101)
==482104==   by 0x4860E0F: pthread_create@@GLIBC_2.2.5 (pthread_create.c:817)
==482104==   by 0x4842917: ??? (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_helgrind.so)
==482104==   by 0x109A05: main (ph.c:141)
==482104==
==482104== ---Thread-Announcement-----
==482104== Thread #2 was created
==482104==   at 0x499A342: clone (clone.S:71)
==482104==   by 0x485F2EB: create_thread (createthread.c:101)
```

计算机系统工程导论实验报告 (Valgrind)

```
==482104== Possible data race during read of size 4 at 0x17EF688 by thread #3
==482104== Locks held: none
==482104==    at 0x10950C: put (ph.c:58)
==482104==    by 0x1097A3: put_thread (ph.c:93)
==482104==    by 0x4842B1A: ??? (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_helgri
==482104==    by 0x4860608: start_thread (pthread_create.c:477)
==482104==    by 0x499A352: clone (clone.S:95)
==482104==
==482104== This conflicts with a previous write of size 4 by thread #2
==482104== Locks held: none
==482104==    at 0x1095A7: put (ph.c:61)
==482104==    by 0x1097A3: put_thread (ph.c:93)
==482104==    by 0x4842B1A: ??? (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_helgri
==482104==    by 0x4860608: start_thread (pthread_create.c:477)
==482104==    by 0x499A352: clone (clone.S:95)
==482104== Address 0x17ef688 is 24000008 bytes inside data symbol "table"
==482104==
==482104== More than 10000000 total errors detected. I'm not reporting any more.
==482104== Final error counts will be inaccurate. Go fix your program!
==482104== Rerun with --error-limit=no to disable this cutoff. Note
==482104== that errors may occur in your program without prior warning from
==482104== Valgrind, because errors are no longer being displayed.
```