1. 背景介绍

计算机系统中，进程和线程是非常重要的两个概念，当一个进程在运行时，进程的内存空间是共享的，即每一个线程都可以使用这些共享空间。线程本身只占用很小的一部分系统资源，当一个线程使用某些共享内存时，其他的线程必须等待它结束，才能使用这一块内存，因此就产生了线程安全的问题。 常见的线程安全解决办法有一下几种：

[https://blog.csdn.net/zhouzhaoxiong1227/article/details/74932180]

1. **竞争和原子操作**，单指令的操作称为原子的，无论如何，单条指令执行是不会被打断的，绝大部分原子操作都是线程安全的。

[https://blog.csdn.net/u012569119/article/details/47850511]

1. **同步与锁**，锁可以保护正在被使用的数据或者资源，每个线程在访问数据或资源之前首先试图获取锁，并在访问结束之后释放锁；在锁已经被占用的时候试图获取锁时，线程会等待，直到锁重新可用。同步（synchronized）则是java中类似原理的关键字。
2. **可重入代码，**函数被重新调用时，不会产生任何不良的结果，可以保证线程的安全性。值得注意的是，并非所有线程安全的代码都是可重入代码。
3. **线程本地存储，**如果一段代码中的数据必须要与其他代码共享，如果可以把这些共享数据的可见范围限制在单个线程以内，则可以实现线程之间不出现数据争用的问题。

[https://blog.csdn.net/u011363729/article/details/79997572]

这里，我们采用有锁和无锁（线程本地存储）两种方法来实现针对多线程malloc的线程安全问题。

1. 有锁版本

通过设置pthread\_mutex\_t lock = PTHREAD\_MUTEX\_INITIALIZER; 既可以设置一个互斥锁，再再malloc中开始和结束的时候，进行打开锁和关断锁即可保证其他线程在尝试运行malloc时的阻塞。最终实现线程安全。

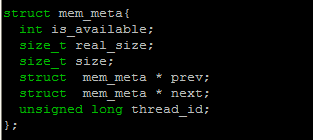
上锁：pthread\_mutex\_lock(&lock);

去锁：pthread\_mutex\_unlock(&lock);



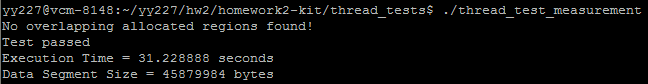
1. 无锁版本

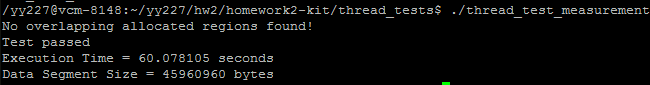
通过调用pthread\_self()指令，可以知道是哪一条线程调用了当前的函数。规定第一次拥有某块区域的线程，可以拥有管理那一块区域的权利，其他线程无法对其进行malloc或者free。并在mem\_meta管理头中保存线程名称(thread\_id)信息。这样就把堆中的每一块区域划分给不同的线程管理。每一个线程拥有其自己的数据结构。





1. 实验结果





|  |  |  |
| --- | --- | --- |
|  | Execution Time | Data Segment Size |
| LOCK | 60.078105 s | 45 960 960 bytes |
| NOLOCK | 31.228888 s | 45 879 984 bytes |

1. 分析和结论
2. 执行时间而言，LOCK版本显著大于NOLOCK版本，这主要是因为在LOCK版本中发生的线程阻塞现象导致。当线程1对数据上锁后，其他线程调用malloc的时候，就必须等待线程1的malloc的结束，才能使用数据。而对于NOLOCK版本而言，则无需等待，直接寻找属于自己的malloc区域。所以速度较快。
3. Data Segment Size指标中，两者差不多。其实从数据利用效率的角度来说，NOLOCK版本应当低于LOCK版本，即NOLOCK的这一指标，应当是大于LOCK中。后来观察测试代码得知了原因：测试代码中，先进行若干次malloc，再进行相应数量的free。即结束了测试，所以没有二次malloc的过程，故空间利用率低的问题，没办法被显示出来。最终分配的bytes大小可以由以下公式计算：



 计算得到的Bytes理论值为46 080 000，与实际结果接近。

1. 从这个实验也可以看出来指导报告中的“为什么sbrk不是线程安全的函数”，在无锁模式中，当所有的线程同时找不到合适的free空间，需要使用sbrk新开空间的时候。如果不使用锁，可能会出现同时分配给了多个线程，则还是会导致冲突。所以必须要采用锁。
2. 展望
3. 对于free的操作还是可以进一步优化。为了简单起见，本代码仍然在free中保留了原来的线程信息，但是free在合并的时候应该可以进一步优化以获得更大的free空间。
4. 也可以添加测试，观察Data Segment Size和LOCK之间的关系。