# Project4 改进LiteOS中物理内存分配算法

## 姓名：任宇 学号：33920212204567

1. **实验目的**

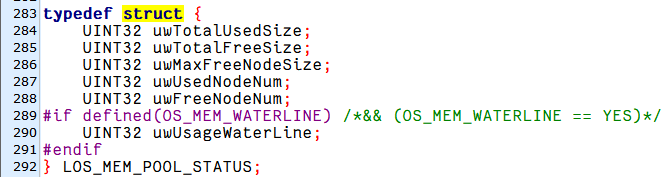
改进LiteOS-a中的物理内存分配算法，从Best-Fit改为Good-Fit，同时修改空闲内存块入队函数，使空闲内存块列表其按照内存块大小排序。

1. **实验环境**
   * 操作系统：
     + - 主机：Windows 10
       - 虚拟机：Ubuntu 18.04
   * 开发板：IMAX6ULL MIN
   * 文件传输工具：FileZilla
   * 终端工具：MobaXterm
2. **实验内容**
3. **移植TLSF算法到开发板上**：

实验所用的开发板的系统内核中并没有实现TLSF算法，因此我们需要将TLSF算法移植到系统内核中，阅读LiteOS-a的官方源码仓库中的TLSF算法实现代码，发现需要对los\_memory.h以及los\_memory.c这两个文件进行修改。因为代码量较大，因此在这里只给出重要部分的说明，完整代码请查看附件：

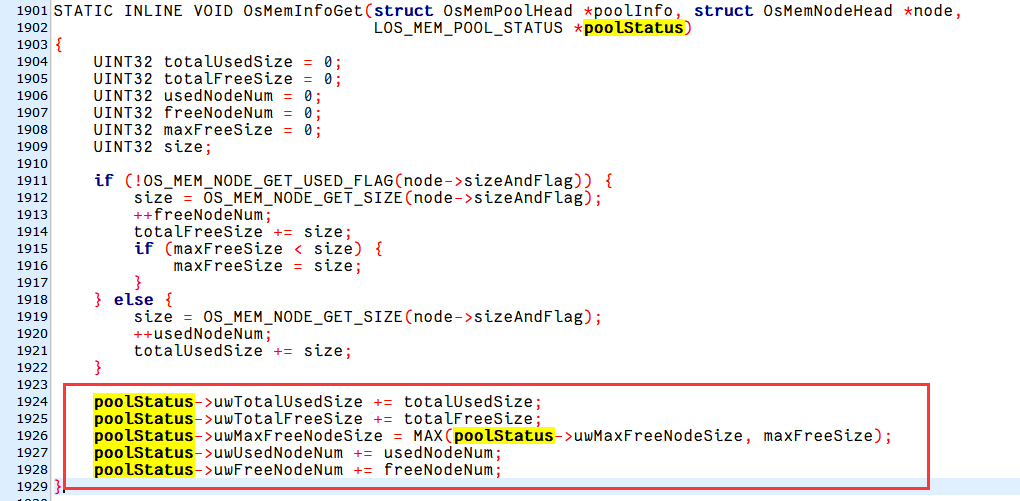
* los\_memory.h文件：

修改LOS\_MEM\_POOR\_STATUS结构体，注释条件编译中的部分条件，如图：



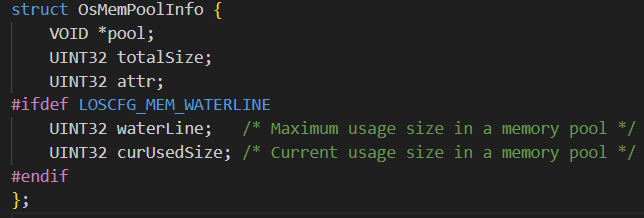
* los\_memory.c文件：

需要注意的是，新版liteOS-a中定义的LOS\_MEM\_POOL\_STATUS的变量其含义虽然与开发板内核中的一致，但是变量名称并不一致，为了不影响其他文件，修改TLSF算法中的对应变量，如图所示(部分示例)：

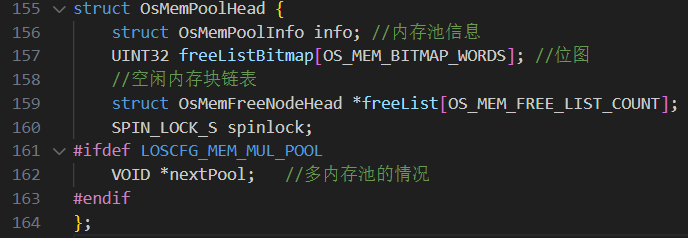


TLSF算法中的重要结构体和重要函数：

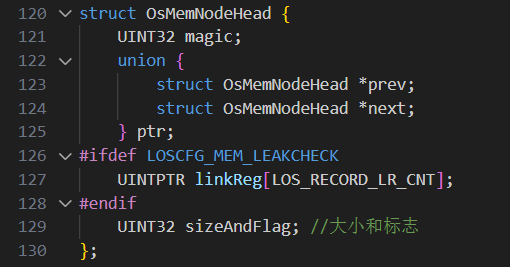
* 动态内存池信息结构体——OsMemPoolInfo



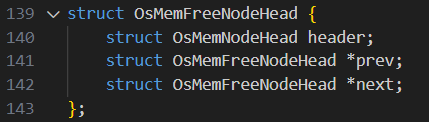
* 动态内存池头结构体——OsMemPoolHead



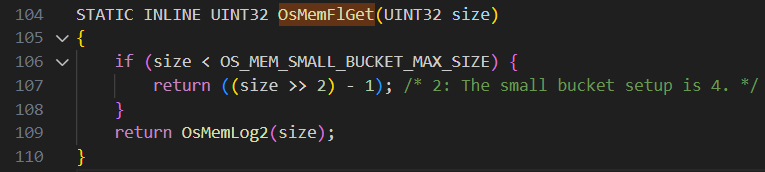
* 动态内存节点头结构体——OsMemNodeHead



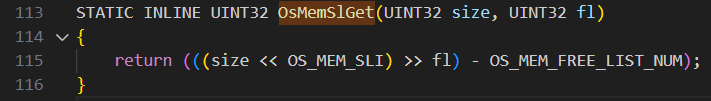
* 空闲内存节点结构体——OsMemFreeNodeHead



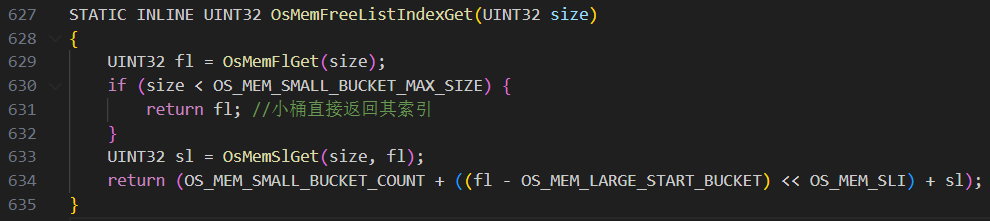
* 获取FL函数——OsMemFlGet



* 获取SL函数——OsMemSLGET



* 获取内存块大小对应索引函数——OsMemFreeListIndexGet

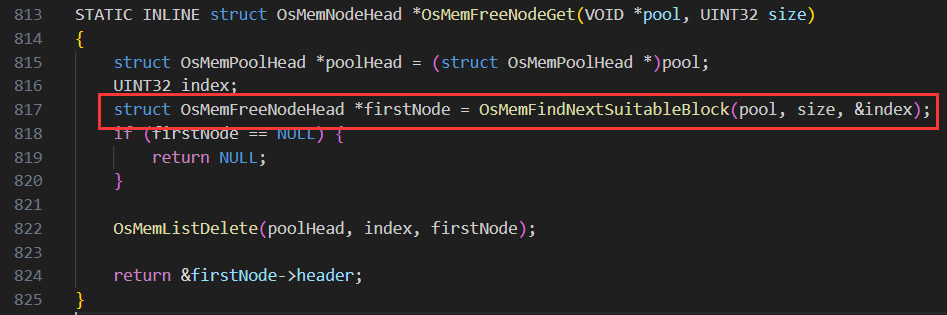


1. **修改原先分配算法中的bestfit为goodfit：**

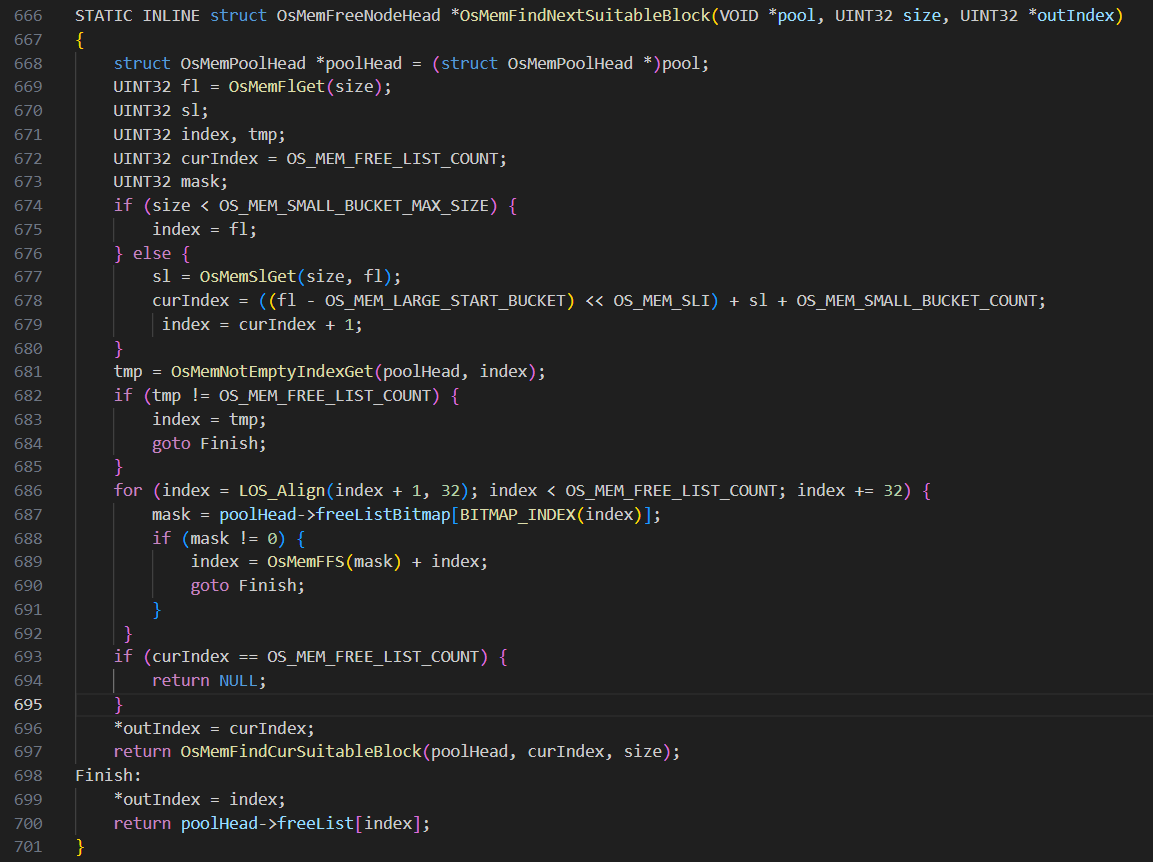
最新版本的LiteOS-a的内核中已经实现了goodfit，即当查找分配大小所对应的空闲内存块时，会先往所对应索引的下一个索引查找空闲内存块，这样虽然造成了一些内存碎片，但是大大节约了遍历链表的时间。举个例子，如果分配一个34字节大小的内存块，我们并不在[32,35]区间内查找，而是直接在下一个区间，即[36,39]中查找。

仿照其代码修改TLSF算法：

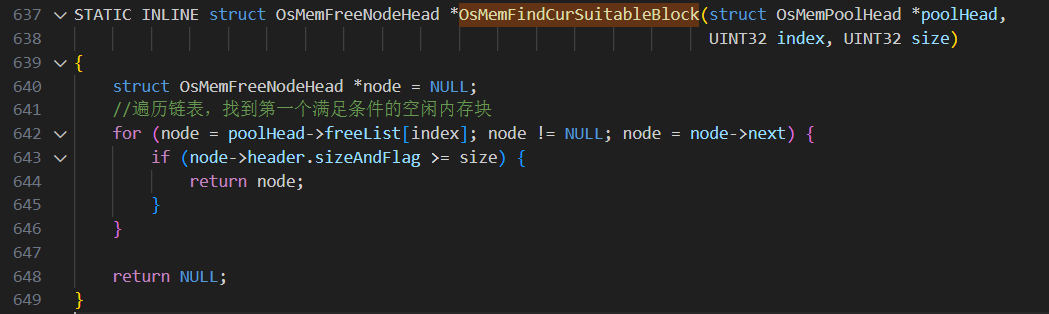
在获取空闲内存块时，调用OsMemFindNextSuitableBlockd函数：



OsMemFindNextSuitableBlock函数的具体逻辑为：首先计算当前预分配大小内存块所在的索引curIndex的值，接着将其加1得到下一块索引index的值，然后查看index所对应的链表是否为空，若不为空，则直接返回其头结点（一定满足大小要求），若为空，则继续下一步。使用函数会遍历内存池的位图以获取挂载空闲内存块的空闲内存链表索引值。如果成功获取到满足大小的空闲内存块，返回空闲链表索引值，否则继续下一步。如果以上条件都不满足，那么就说明只能在当前索引所对应的空闲内存链表中查找合适的内存块，因此返回调用OsMemFindCurSuitableBlock函数的结果，具体代码如下所示：

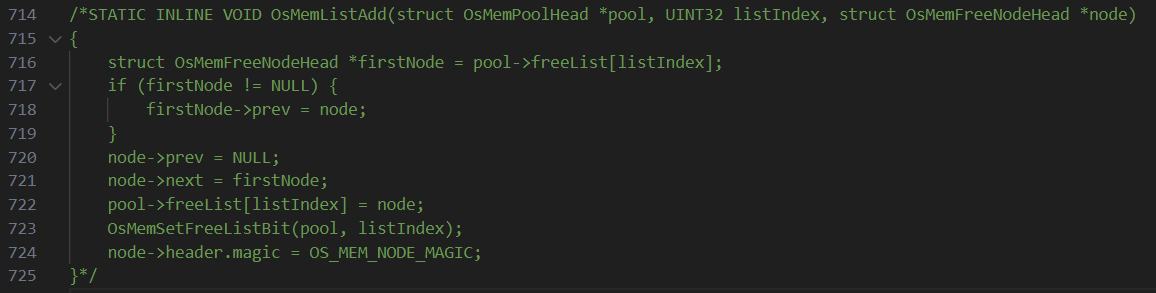


OsMemFindCurSuitableBlock函数的实现如图所示：

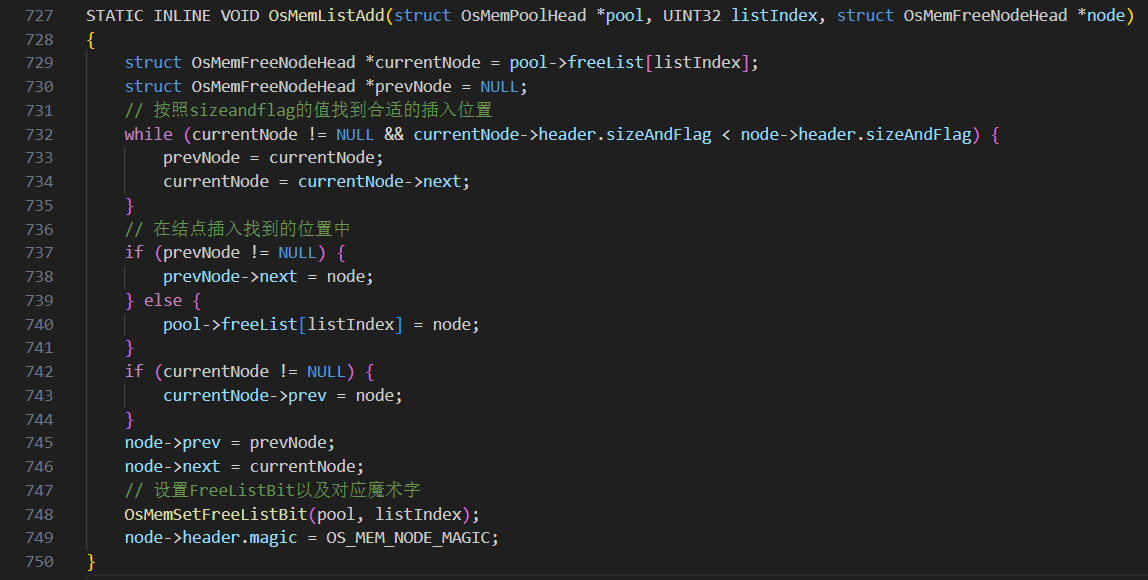


1. **修改空闲内存块入队函数，使其按照内存块大小排序：**

在原先的OsMemListAdd函数中，待插入结点直接头插进对应空闲内存块链表，如图所示：



而在修改后，待插入结点会按照其对应大小插入空闲内存块链表，使得空闲内存块链表是按照从小到大的顺序排列，如图所示：

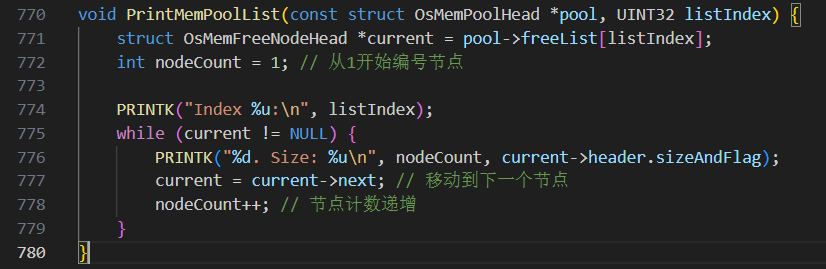


1. **验证：**

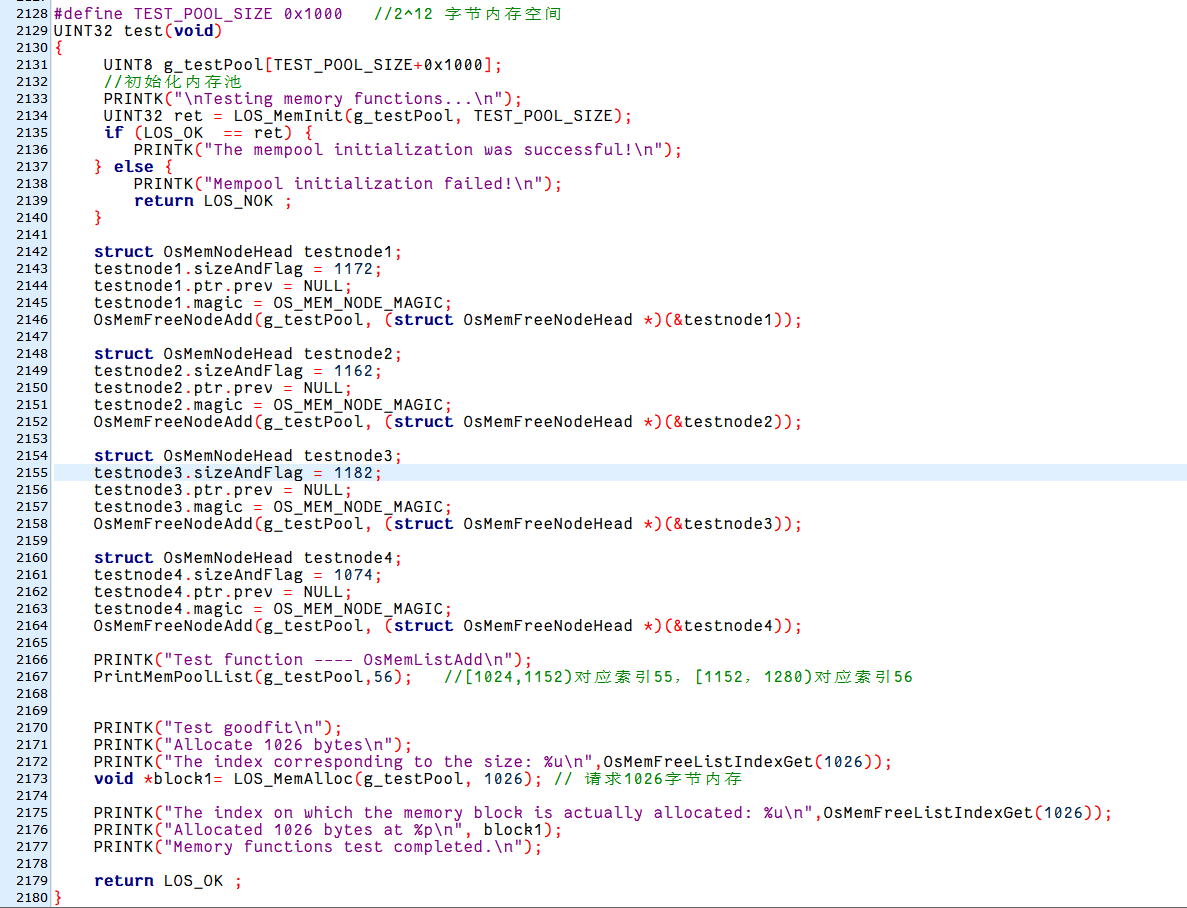
验证所作修改：

对于goodfit：当分配内存时，首先计算出待分配内存对应的索引，接着打印出实际分配内存块所对应的索引，查看二者关系。

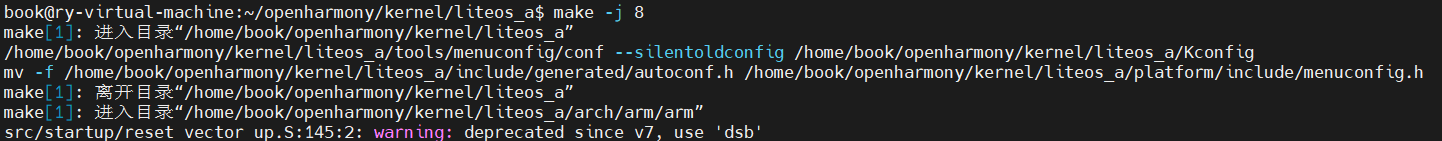
对于按内存块大小排序链表：首先打印出原先OsMemListAdd函数所实现的链表，接着打印出修改后OsMemListAdd函数所实现的链表，观察是否按照大小排序。为此在los\_memory当中添加PrintMemPoolList函数，实现链表的打印，如图所示：

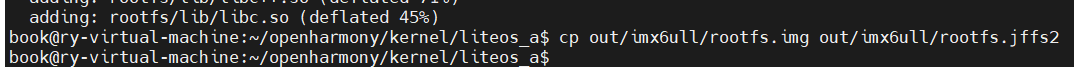


在los\_memory.c文件中编写测试函数以验证所作修改，并在syscall\_demo.c文件中调用该测试函数，然后通过project4.c文件调用，测试函数代码如下：

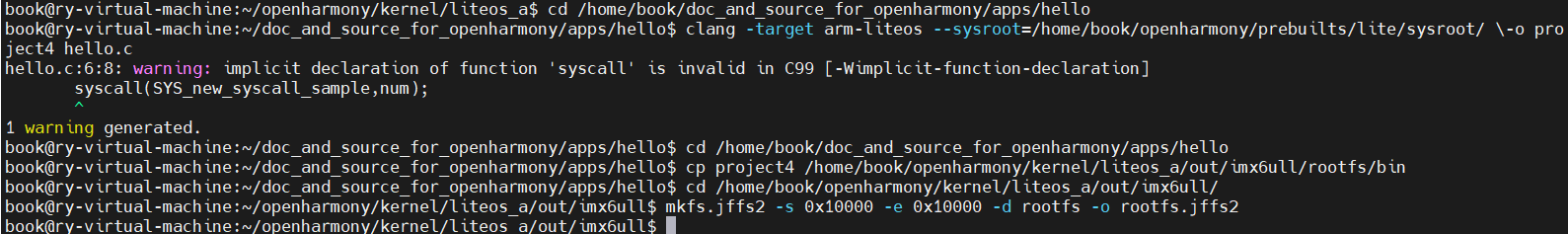


编译系统内核，如图所示：

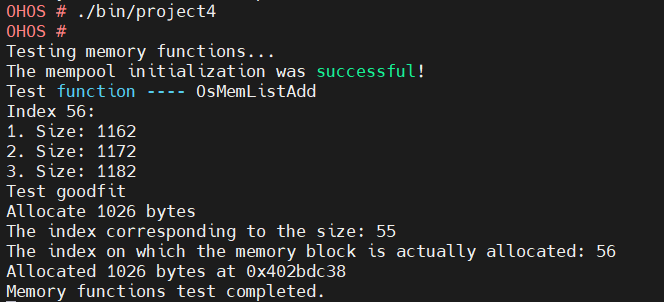




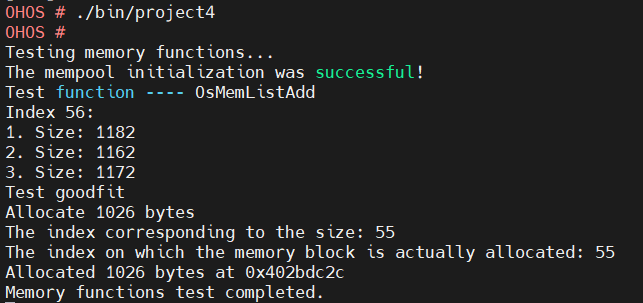
编译project4.c文件，并将其添加到bin目录中：



观察结果，可见修改后的函数正确运行并输出预期结果，实验成功，实验结果如图所示。



修改前结果如图所示：



1. **实验结果**

原始的TLSF算法存在优化空间——Best-fit策略需要检索对应范围的那一条空闲块链表，存在潜在的时间复杂度。而Good-fit分配策略将动态内存的分配与回收时间复杂度都降到了O(1)时间复杂度，并且保证系统运行时不会产生过多碎片。同时，将Good-Fit策略与内存块大小排序相结合，也可以有效减少内存碎片，提高内存分配的速度和效率。

总体来看，这些改进对于提高LiteOS-a系统的性能和稳定性是积极的。

1. **实验分析**
2. 内存分配效率提升

原因分析：Good-Fit策略相比于Best-Fit策略，在选择内存块时更倾向于选择“足够好”的块，而不是最佳匹配。这减少了搜索时间，特别是在内存分配请求频繁的情况下。

影响：改进后的算法能够更快地响应内存分配请求，对于需要快速内存分配的应用场景尤为有利。

1. 内存利用率增加

原因分析：通过对空闲内存块进行排序，算法能够更有效地管理内存块，减少了内存碎片。这种方法提高了内存块的可重用性，减少了因为大小不匹配而无法使用的内存块。

影响：系统能够更有效地利用可用内存，减少内部碎片。

1. **实验总结**

本次实验主要目标是改进LiteOS-a中的TLSF（Two-Level Segregated Fit）内存分配算法。通过将分配策略从Best-Fit改为Good-Fit，并优化空闲内存块的排序机制，实验旨在提高内存分配的效率和系统的总体内存利用率。

主要发现：

1. 内存分配效率提升：改进后的Good-Fit策略减少了内存分配的平均时间，加快了内存分配过程，特别是在内存请求频繁的应用场景中。

2. 内存利用率增加：通过空闲内存块的大小排序，算法减少了内存碎片，提高了内存的整体利用率。

结论：

Good-Fit策略结合内存块大小排序的方法，有效减少了内存碎片，提高了内存分配效率，增强了系统对内存资源的管理能力。然而，这种改进需要额外的计算资源来维护内存块的排序，这在极端的资源受限情况下可能成为考虑因素。

后续方向：

考虑对改进后的算法进行进一步的优化，如调整内存块的大小阈值，以更好地适应不同的应用场景。

1. **参考文献**
2. [kernel/base/mem/tlsf · OpenHarmony/kernel\_liteos\_a - 码云 - 开源中国 (gitee.com)](https://gitee.com/openharmony/kernel_liteos_a/tree/master/kernel/base/mem/tlsf)
3. [鸿蒙轻内核M核源码分析系列九 动态内存Dynamic Memory 第一部分-云社区-华为云 (huaweicloud.com)](https://bbs.huaweicloud.com/blogs/280792)
4. [鸿蒙轻内核M核源码学习 动态内存（1）-云社区-华为云 (huaweicloud.com)](https://bbs.huaweicloud.com/blogs/309066)
5. **附录**
6. **los\_memory.h及los\_memory.c(代码量较大，详见附件)**
7. **syscall\_demo.c :**
8. #include "los\_printf.h"
9. #include "los\_config.h"
10. #include "los\_memory.h"
11. #include <stdlib.h>
12. #ifdef \_\_cplusplus
13. #if \_\_cplusplus
14. extern "C" {
15. #endif /\* \_\_cpluscplus \*/
16. #endif /\* \_\_cpluscplus \*/
17. int SysNewSyscallSample(int num)
18. {
19. UINT32 ret;
20. ret=test();
21. return LOS\_OK;
22. }
23. #ifdef \_\_cplusplus
24. #if \_\_cplusplus
25. }
26. #endif /\* \_\_cpluscplus \*/
27. #endif /\* \_\_cpluscplus \*/
28. **project4.c :**
29. #include <stdio.h>
30. #include <syscall.h>
31. void newSyscallSample(int num)
32. {
33. syscall(SYS\_new\_syscall\_sample,num);
34. return;
35. }
36. int main(void)
37. {
38. newSyscallSample(0);
39. return 0;
40. }