Project4 改进 LiteOS 中物理内存分配算法

姓名：任宇　　学号：33920212204567

一、 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。

二、 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

三、 实验内容

1. 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：

```c
typedef struct {
    UINT32 uwTotalUsedSize;
    UINT32 uwTotalFreeSize;
    UINT32 uwMaxFreeNodeSize;
    UINT32 uwUsedNodeNum;
    UINT32 uwFreeNodeNum;
#if defined(OS_MEM_WATERLINE) /*&& (OS_MEM_WATERLINE == YES)*/
    UINT32 uwUsageWaterLine;
#endif
} LOS_MEM_POOL_STATUS;
```

- 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)：

```c
STATIC INLINE VOID OsMemInfoGet(struct OsMemPoolHead *poolInfo, struct OsMemNodeHead *node,
                                LOS_MEM_POOL_STATUS *poolStatus)
{
    UINT32 totalUsedSize = 0;
    UINT32 totalFreeSize = 0;
    UINT32 usedNodeNum = 0;
    UINT32 freeNodeNum = 0;
    UINT32 maxFreeSize = 0;
    UINT32 size;

    if (!OS_MEM_NODE_GET_USED_FLAG(node->sizeAndFlag)) {
        size = OS_MEM_NODE_GET_SIZE(node->sizeAndFlag);
        ++freeNodeNum;
        totalFreeSize += size;
        if (maxFreeSize < size) {
            maxFreeSize = size;
        }
    } else {
        size = OS_MEM_NODE_GET_SIZE(node->sizeAndFlag);
        ++usedNodeNum;
        totalUsedSize += size;
    }

    poolStatus->uwTotalUsedSize += totalUsedSize;
    poolStatus->uwTotalFreeSize += totalFreeSize;
    poolStatus->uwMaxFreeNodeSize = MAX(poolStatus->uwMaxFreeNodeSize, maxFreeSize);
    poolStatus->uwUsedNodeNum += usedNodeNum;
    poolStatus->uwFreeNodeNum += freeNodeNum;
}
```

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

```c
struct OsMemPoolInfo {
    VOID *pool;
    UINT32 totalSize;
    UINT32 attr;
#ifdef LOSCFG_MEM_WATERLINE
    UINT32 waterLine;   /* Maximum usage size in a memory pool */
    UINT32 curUsedSize; /* Current usage size in a memory pool */
#endif
};
```

- 动态内存池头结构体——OsMemPoolHead

```c
struct OsMemPoolHead {
    struct OsMemPoolInfo info; //内存池信息
    UINT32 freeListBitmap[OS_MEM_BITMAP_WORDS]; //位图
    //空闲内存块链表
    struct OsMemFreeNodeHead *freeList[OS_MEM_FREE_LIST_COUNT];
    SPIN_LOCK_S spinlock;
#ifdef LOSCFG_MEM_MUL_POOL
    VOID *nextPool;   //多内存池的情况
#endif
};
```

- 动态内存节点头结构体——OsMemNodeHead

```
120 struct OsMemNodeHead {
121     UINT32 magic;
122     union {
123         struct OsMemNodeHead *prev;
124         struct OsMemNodeHead *next;
125     } ptr;
126 #ifdef LOSCFG_MEM_LEAKCHECK
127     UINTPTR linkReg[LOS_RECORD_LR_CNT];
128 #endif
129     UINT32 sizeAndFlag; //大小和标志
130 };
```

- 空闲内存节点结构体——OsMemFreeNodeHead

```
139 struct OsMemFreeNodeHead {
140     struct OsMemNodeHead header;
141     struct OsMemFreeNodeHead *prev;
142     struct OsMemFreeNodeHead *next;
143 };
```

- 获取FL函数——OsMemFlGet

```
104 STATIC INLINE UINT32 OsMemFlGet(UINT32 size)
105 {
106     if (size < OS_MEM_SMALL_BUCKET_MAX_SIZE) {
107         return ((size >> 2) - 1); /* 2: The small bucket setup is 4. */
108     }
109     return OsMemLog2(size);
110 }
```

- 获取SL函数——OsMemSLGET

```
113 STATIC INLINE UINT32 OsMemSlGet(UINT32 size, UINT32 fl)
114 {
115     return (((size << OS_MEM_SLI) >> fl) - OS_MEM_FREE_LIST_NUM);
116 }
```

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

```
627 STATIC INLINE UINT32 OsMemFreeListIndexGet(UINT32 size)
628 {
629     UINT32 fl = OsMemFlGet(size);
630     if (size < OS_MEM_SMALL_BUCKET_MAX_SIZE) {
631         return fl; //小桶直接返回其索引
632     }
633     UINT32 sl = OsMemSlGet(size, fl);
634     return (OS_MEM_SMALL_BUCKET_COUNT + ((fl - OS_MEM_LARGE_START_BUCKET) << OS_MEM_SLI) + sl);
635 }
```

2. 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：

```
813  STATIC INLINE struct OsMemNodeHead *OsMemFreeNodeGet(VOID *pool, UINT32 size)
814  {
815      struct OsMemPoolHead *poolHead = (struct OsMemPoolHead *)pool;
816      UINT32 index;
817      struct OsMemFreeNodeHead *firstNode = OsMemFindNextSuitableBlock(pool, size, &index);
818      if (firstNode == NULL) {
819          return NULL;
820      }
821
822      OsMemListDelete(poolHead, index, firstNode);
823
824      return &firstNode->header;
825  }
```

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：

```
666  STATIC INLINE struct OsMemFreeNodeHead *OsMemFindNextSuitableBlock(VOID *pool, UINT32 size, UINT32 *outIndex)
667  {
668      struct OsMemPoolHead *poolHead = (struct OsMemPoolHead *)pool;
669      UINT32 fl = OsMemFlGet(size);
670      UINT32 sl;
671      UINT32 index, tmp;
672      UINT32 curIndex = OS_MEM_FREE_LIST_COUNT;
673      UINT32 mask;
674      if (size < OS_MEM_SMALL_BUCKET_MAX_SIZE) {
675          index = fl;
676      } else {
677          sl = OsMemSlGet(size, fl);
678          curIndex = ((fl - OS_MEM_LARGE_START_BUCKET) << OS_MEM_SLI) + sl + OS_MEM_SMALL_BUCKET_COUNT;
679          index = curIndex + 1;
680      }
681      tmp = OsMemNotEmptyIndexGet(poolHead, index);
682      if (tmp != OS_MEM_FREE_LIST_COUNT) {
683          index = tmp;
684          goto Finish;
685      }
686      for (index = LOS_Align(index + 1, 32); index < OS_MEM_FREE_LIST_COUNT; index += 32) {
687          mask = poolHead->freeListBitmap[BITMAP_INDEX(index)];
688          if (mask != 0) {
689              index = OsMemFFS(mask) + index;
690              goto Finish;
691          }
692      }
693      if (curIndex == OS_MEM_FREE_LIST_COUNT) {
694          return NULL;
695      }
696      *outIndex = curIndex;
697      return OsMemFindCurSuitableBlock(poolHead, curIndex, size);
698  Finish:
699      *outIndex = index;
700      return poolHead->freeList[index];
701  }
```

OsMemFindCurSuitableBlock 函数的实现如图所示：

```
637 STATIC INLINE struct OsMemFreeNodeHead *OsMemFindCurSuitableBlock(struct OsMemPoolHead *poolHead,
638                                                                    UINT32 index, UINT32 size)
639 {
640     struct OsMemFreeNodeHead *node = NULL;
641     //遍历链表，找到第一个满足条件的空闲内存块
642     for (node = poolHead->freeList[index]; node != NULL; node = node->next) {
643         if (node->header.sizeAndFlag >= size) {
644             return node;
645         }
646     }
647
648     return NULL;
649 }
```

3. 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：

```
714 /*STATIC INLINE VOID OsMemListAdd(struct OsMemPoolHead *pool, UINT32 listIndex, struct OsMemFreeNodeHead *node)
715 {
716     struct OsMemFreeNodeHead *firstNode = pool->freeList[listIndex];
717     if (firstNode != NULL) {
718         firstNode->prev = node;
719     }
720     node->prev = NULL;
721     node->next = firstNode;
722     pool->freeList[listIndex] = node;
723     OsMemSetFreeListBit(pool, listIndex);
724     node->header.magic = OS_MEM_NODE_MAGIC;
725 }*/
```

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：

```
727 STATIC INLINE VOID OsMemListAdd(struct OsMemPoolHead *pool, UINT32 listIndex, struct OsMemFreeNodeHead *node)
728 {
729     struct OsMemFreeNodeHead *currentNode = pool->freeList[listIndex];
730     struct OsMemFreeNodeHead *prevNode = NULL;
731     // 按照sizeandflag的值找到合适的插入位置
732     while (currentNode != NULL && currentNode->header.sizeAndFlag < node->header.sizeAndFlag) {
733         prevNode = currentNode;
734         currentNode = currentNode->next;
735     }
736     // 在结点插入找到的位置中
737     if (prevNode != NULL) {
738         prevNode->next = node;
739     } else {
740         pool->freeList[listIndex] = node;
741     }
742     if (currentNode != NULL) {
743         currentNode->prev = node;
744     }
745     node->prev = prevNode;
746     node->next = currentNode;
747     // 设置FreeListBit以及对应魔术字
748     OsMemSetFreeListBit(pool, listIndex);
749     node->header.magic = OS_MEM_NODE_MAGIC;
750 }
```

4. 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：

```
770  void PrintMemPoolList(const struct OsMemPoolHead *pool, UINT32 listIndex) {
771      struct OsMemFreeNodeHead *current = pool->freeList[listIndex];
772      int nodeCount = 1; // 从1开始编号节点
773
774      PRINTK("Index %u:\n", listIndex);
775      while (current != NULL) {
776          PRINTK("%d. Size: %u\n", nodeCount, current->header.sizeAndFlag);
777          current = current->next; // 移动到下一个节点
778          nodeCount++; // 节点计数递增
779      }
780  }
```

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：

```
2128 #define TEST_POOL_SIZE 0x1000   //2^12 字节内存空间
2129 UINT32 test(void)
2130 {
2131      UINT8 g_testPool[TEST_POOL_SIZE+0x1000];
2132      //初始化内存池
2133      PRINTK("\nTesting memory functions...\n");
2134      UINT32 ret = LOS_MemInit(g_testPool, TEST_POOL_SIZE);
2135      if (LOS_OK  == ret) {
2136         PRINTK("The mempool initialization was successful!\n");
2137     } else {
2138         PRINTK("Mempool initialization failed!\n");
2139         return LOS_NOK ;
2140     }
2141
2142     struct OsMemNodeHead testnode1;
2143     testnode1.sizeAndFlag = 1172;
2144     testnode1.ptr.prev = NULL;
2145     testnode1.magic = OS_MEM_NODE_MAGIC;
2146     OsMemFreeNodeAdd(g_testPool, (struct OsMemFreeNodeHead *)(&testnode1));
2147
2148     struct OsMemNodeHead testnode2;
2149     testnode2.sizeAndFlag = 1162;
2150     testnode2.ptr.prev = NULL;
2151     testnode2.magic = OS_MEM_NODE_MAGIC;
2152     OsMemFreeNodeAdd(g_testPool, (struct OsMemFreeNodeHead *)(&testnode2));
2153
2154     struct OsMemNodeHead testnode3;
2155     testnode3.sizeAndFlag = 1182;
2156     testnode3.ptr.prev = NULL;
2157     testnode3.magic = OS_MEM_NODE_MAGIC;
2158     OsMemFreeNodeAdd(g_testPool, (struct OsMemFreeNodeHead *)(&testnode3));
2159
2160     struct OsMemNodeHead testnode4;
2161     testnode4.sizeAndFlag = 1074;
2162     testnode4.ptr.prev = NULL;
2163     testnode4.magic = OS_MEM_NODE_MAGIC;
2164     OsMemFreeNodeAdd(g_testPool, (struct OsMemFreeNodeHead *)(&testnode4));
2165
2166     PRINTK("Test function ---- OsMemListAdd\n");
2167     PrintMemPoolList(g_testPool,56);   //[1024,1152)对应索引55，[1152，1280)对应索引56
2168
2169
2170     PRINTK("Test goodfit\n");
2171     PRINTK("Allocate 1026 bytes\n");
2172     PRINTK("The index corresponding to the size: %u\n",OsMemFreeListIndexGet(1026));
2173     void *block1= LOS_MemAlloc(g_testPool, 1026); // 请求1026字节内存
2174
2175     PRINTK("The index on which the memory block is actually allocated: %u\n",OsMemFreeListIndexGet(1026));
2176     PRINTK("Allocated 1026 bytes at %p\n", block1);
2177     PRINTK("Memory functions test completed.\n");
2178
2179     return LOS_OK ;
2180 }
```

编译系统内核，如图所示：

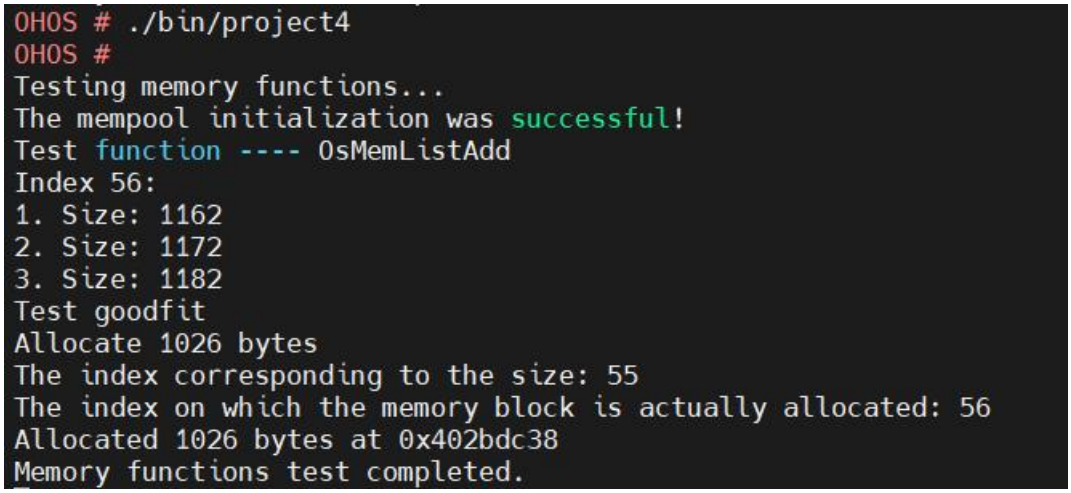
```
book@ry-virtual-machine:~/openharmony/kernel/liteos_a$ make -j 8
make[1]: 进入目录“/home/book/openharmony/kernel/liteos_a”
/home/book/openharmony/kernel/liteos_a/tools/menuconfig/conf --silentoldconfig /home/book/openharmony/kernel/liteos_a/Kconfig
mv -f /home/book/openharmony/kernel/liteos_a/include/generated/autoconf.h /home/book/openharmony/kernel/liteos_a/platform/include/menuconfig.h
make[1]: 离开目录“/home/book/openharmony/kernel/liteos_a”
make[1]: 进入目录“/home/book/openharmony/kernel/liteos_a/arch/arm/arm”
src/startup/reset_vector_up.S:145:2: warning: deprecated since v7, use 'dsb'
```

```
  adding: rootfs/lib/libc.so (deflated 45%)
book@ry-virtual-machine:~/openharmony/kernel/liteos_a$ cp out/imx6ull/rootfs.img out/imx6ull/rootfs.jffs2
book@ry-virtual-machine:~/openharmony/kernel/liteos_a$
```

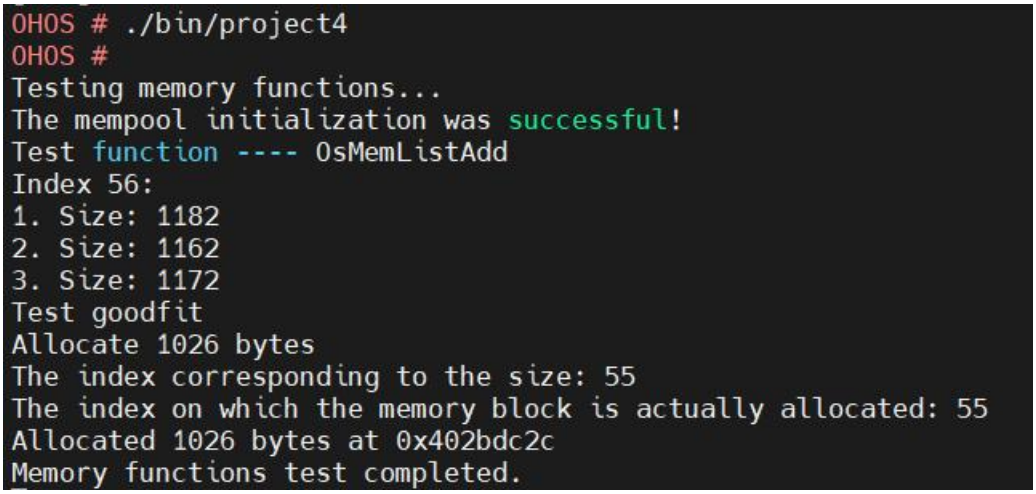
编译 project4.c 文件，并将其添加到 bin 目录中：

```
book@ry-virtual-machine:~/openharmony/kernel/liteos_a$ cd /home/book/doc_and_source_for_openharmony/apps/hello
book@ry-virtual-machine:~/doc_and_source_for_openharmony/apps/hello$ clang -target arm-liteos --sysroot=/home/book/openharmony/prebuilts/lite/sysroot/ \-o pro
ject4 hello.c
hello.c:6:8: warning: implicit declaration of function 'syscall' is invalid in C99 [-Wimplicit-function-declaration]
      syscall(SYS_new_syscall_sample,num);
      ^
1 warning generated.
book@ry-virtual-machine:~/doc_and_source_for_openharmony/apps/hello$ cd /home/book/doc_and_source_for_openharmony/apps/hello
book@ry-virtual-machine:~/doc_and_source_for_openharmony/apps/hello$ cp project4 /home/book/openharmony/kernel/liteos_a/out/imx6ull/rootfs/bin
book@ry-virtual-machine:~/doc_and_source_for_openharmony/apps/hello$ cd /home/book/openharmony/kernel/liteos_a/out/imx6ull/
book@ry-virtual-machine:~/openharmony/kernel/liteos_a/out/imx6ull$ mkfs.jffs2 -s 0x10000 -e 0x10000 -d rootfs -o rootfs.jffs2
book@ry-virtual-machine:~/openharmony/kernel/liteos_a/out/imx6ull$
```

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：



四、 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。

五、 实验分析

1. 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。

2. 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。

六、 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。

七、 参考文献

1. kernel/base/mem/tlsf · OpenHarmony/kernel_liteos_a - 码云 - 开源中国(gitee.com)
2. 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)
3. 鸿蒙轻内核 M 核源码学习 动态内存（1）-云社区-华为云 (huaweicloud.com)

八、 附录

1. los_memory.h 及 los_memory.c(代码量较大，详见附件)

2. syscall_demo.c ：

```
#include "los_printf.h"
#include "los_config.h"
#include "los_memory.h"
#include <stdlib.h>

#ifdef __cplusplus
#if __cplusplus
extern "C" {
#endif /* __cpluscplus */
#endif /* __cpluscplus */
```

```
11.
12.int SysNewSyscallSample(int num)
13.{
14.    UINT32 ret;
15.    ret=test();
16.    return LOS_OK;
17.}
18.
19.#ifdef __cplusplus
20.#if __cplusplus
21.}
22.#endif /* __cpluscplus */
23.#endif /* __cpluscplus */
```

3. project4.c ：

```
1. #include <stdio.h>
2. #include <syscall.h>
3.
4. void newSyscallSample(int num)
5. {
6.        syscall(SYS_new_syscall_sample,num);
7.     return;
8. }
9.
10.int main(void)
11.{
12.  newSyscallSample(0);
13.    return 0;
14.}
```