

# 项目4 改进 *LiteOS* 中物理内存分配算法

22920212204392 黄勛

## 1 实验目的

## 2 实验环境

## 3 实验思路

## 4 实验内容

### 4.1 修改内存分配算法

4.1.1 ./liteos\_a/kernel/base/include/los\_multipledlinkhead\_pri.h

4.1.2 ./liteos\_a/kernel/base/mem/bestfit/los\_multipledlinkhead.c

4.1.3 ./liteos\_a/kernel/base/mem/bestfit/los\_memory.c

### 4.2 在系统调用中检验分配

#### 4.2.1 新增系统调用号

4.2.1.1 ./prebuilts/lite/sysroot/usr/include/arm-liteos/bits/syscall.h

4.2.1.2 ./third\_party/musl/kernel/obj/include/bits/syscall.h

#### 4.2.2 新增系统调用的函数声明

4.2.2.1 /home/book/openharmony/kernel/liteos\_a/syscall/los\_syscall.h

#### 4.2.3 新增系统调用的函数实现

#### 4.2.4 新增系统调用号和系统调用函数的映射关系

4.2.4.1 /home/book/openharmony/kernel/liteos\_a/syscall/syscall\_lookup.h

### 4.3 烧录运行

## 5 实验结果与分析

### 5.1 best-fit 算法结果

### 5.2 good-fit 算法结果

## 6 实验分析

### 6.1 问题分析

6.1.1 实际有内存但申请无内存块分配

6.1.2 疯狂输出，按键无反应

### 6.2 代码分析

#### 6.2.1 动态内存结构体定义

6.2.1.1 动态内存池信息结构体 `LosMemPoolInfo`

6.2.1.2 多双向链表表头结构体 `LosMultipleDlinkHead`

6.2.2 函数 `OsDlnkMultiHead()`

6.2.3 函数 `LOS_MemAlloc()`

6.2.4 函数 `OsMemAllocWithCheck(pool, size)`

## 7 实验总结

## 8 参考文献

## 9 附录

## 1 实验目的

`LiteOS` 中的物理内存分配采用了TLSF算法，采用的 `Best-fit` 策略最主要的问题还在于第三步，仍然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。

本次project需要实现 `Good-fit` 思路，与 `Best-fit` 不同之处在于 `Good-fit` 并不保证找到满足需求的最小空闲块，而是尽可能接近要分配的大小。

以搜索大小为69字节的空闲块为例，`Good-fit` 并不是找到  $[68 \sim 70]$  这一范围，而是比这个范围稍微大一点儿的范围(例如  $[71 \sim 73]$ )。这样设计的好处就是  $[71 \sim 73]$  对应的空闲块链中每一块都能满足需求，不需要检索空闲块链表找到最小的，而是直接取空闲块链中第一块即可。整体上还不会造成太多碎片。`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。

## 2 实验环境

宿主机操作系统: `Windows 10`

虚拟机操作系统: `Ubuntu 18.04.6`，在完成Project3的虚拟机中完成

开发板: `IMAX6ULL MINI`

终端工具: `MobaXterm`

## 3 实验思路

本次project主要的内容即找到 `LiteOS` 分配物理内存的代码部分，按照要求修改分配的方法，并且设计一种方法验证分配的正确性即可。

如何思考设计出分配的方法可以参考 6.2代码分析。

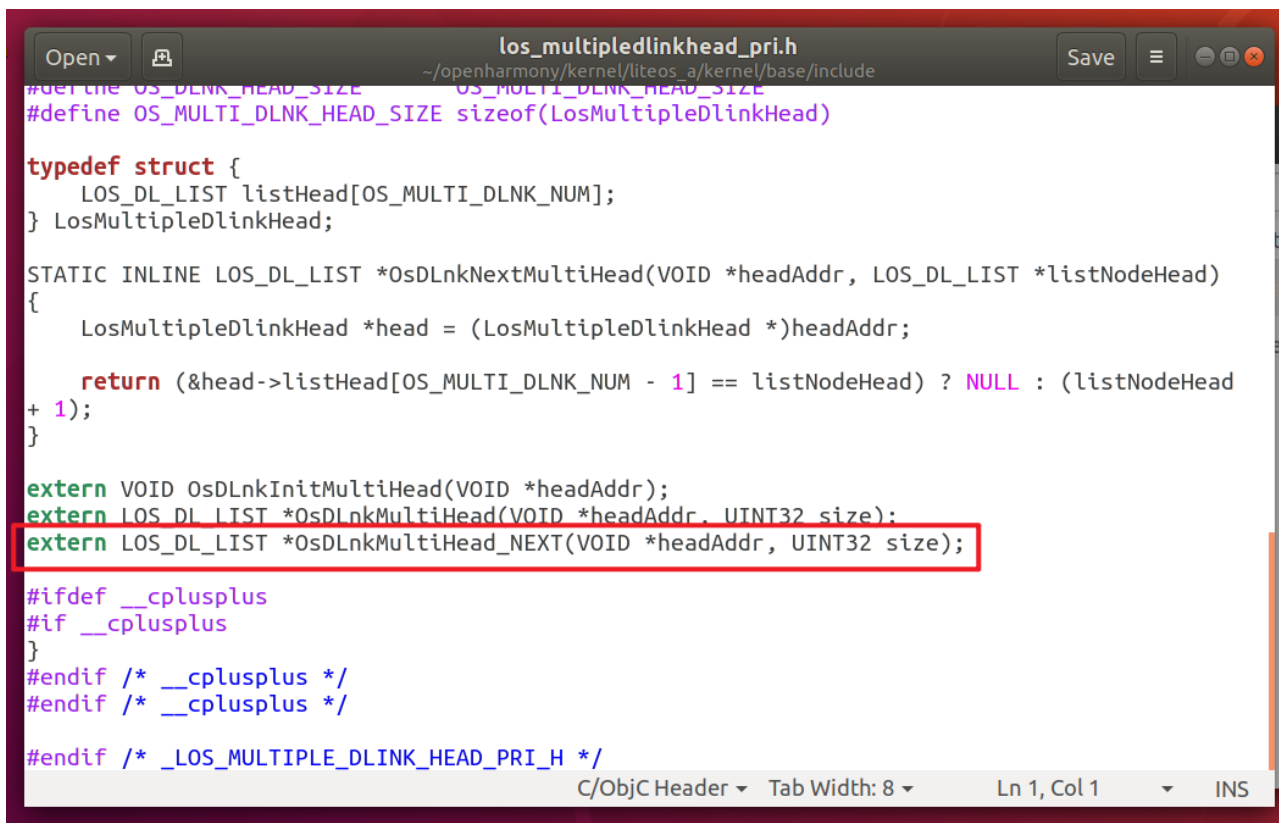
## 4 实验内容




### 4.1 修改内存分配算法

#### 4.1.1 `./liteos_a/kernel/base/include/los_multiplinkhead_pri.h`

声明找到空闲块链的下一块的索引的函数

```
extern LOS_DL_LIST *OsDlnkMultiHead_NEXT(VOID *headAddr, UINT32 size);
```



```
Open ▾  los_multipliedlinkhead_pri.h Save   
~/openharmy/kernel/liteos_a/kernel/base/include
#define OS_MULTI_DLNK_HEAD_SIZE OS_MULTI_DLNK_HEAD_SIZE
#define OS_MULTI_DLNK_HEAD_SIZE sizeof(LosMultipleDlinkHead)

typedef struct {
    LOS_DL_LIST listHead[OS_MULTI_DLNK_NUM];
} LosMultipleDlinkHead;

STATIC INLINE LOS_DL_LIST *OsDlnkNextMultiHead(VOID *headAddr, LOS_DL_LIST *listNodeHead)
{
    LosMultipleDlinkHead *head = (LosMultipleDlinkHead *)headAddr;

    return (&head->listHead[OS_MULTI_DLNK_NUM - 1] == listNodeHead) ? NULL : (listNodeHead
+ 1);
}

extern VOID OsDlnkInitMultiHead(VOID *headAddr);
extern LOS_DL_LIST *OsDlnkMultiHead(VOID *headAddr, UINT32 size);
extern LOS_DL_LIST *OsDlnkMultiHead_NEXT(VOID *headAddr, UINT32 size);

#ifdef __cplusplus
#if __cplusplus
}
#endif /* __cplusplus */
#endif /* __cplusplus */

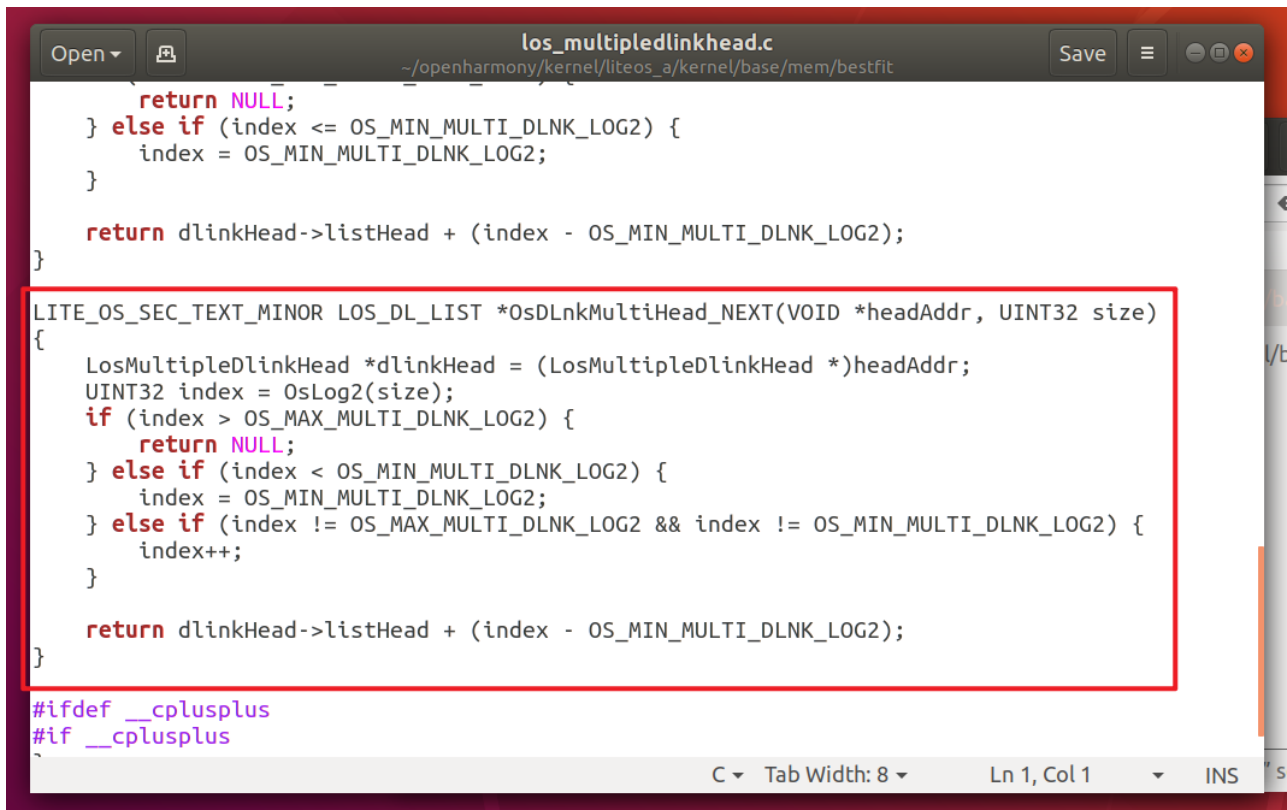
#endif /* _LOS_MULTIPLE_DLINK_HEAD_PRI_H */
C/ObjC Header ▾ Tab Width: 8 ▾ Ln 1, Col 1 ▾ INS
```

#### 4.1.2 ./liteos\_a/kernel/base/mem/bestfit/los\_multipliedlinkhead.c

实现找到空闲块链的下一块的索引的函数

```
LITE_OS_SEC_TEXT_MINOR LOS_DL_LIST *OsDlnkMultiHead_NEXT(VOID *headAddr, UINT32
size)
{
    LosMultipleDlinkHead *dlinkHead = (LosMultipleDlinkHead *)headAddr;
    UINT32 index = OsLog2(size);
    if (index > OS_MAX_MULTI_DLNK_LOG2) {
        return NULL;
    } else if (index < OS_MIN_MULTI_DLNK_LOG2) {
        index = OS_MIN_MULTI_DLNK_LOG2;
    } else if (index != OS_MAX_MULTI_DLNK_LOG2 && index !=
OS_MIN_MULTI_DLNK_LOG2) {
        index++; //关键在此，找到空闲的下一块
    }

    return dlinkHead->listHead + (index - OS_MIN_MULTI_DLNK_LOG2);
}
```



```
Open  los_multipliedlinkhead.c  Save
~/openharmony/kernel/liteos_a/kernel/base/mem/bestfit

    return NULL;
} else if (index <= OS_MIN_MULTI_DLNK_LOG2) {
    index = OS_MIN_MULTI_DLNK_LOG2;
}

return dlinkHead->listHead + (index - OS_MIN_MULTI_DLNK_LOG2);
}

LITE_OS_SEC_TEXT_MINOR LOS_DL_LIST *OsDlnkMultiHead_NEXT(VOID *headAddr, UINT32 size)
{
    LosMultipleDlinkHead *dlinkHead = (LosMultipleDlinkHead *)headAddr;
    UINT32 index = OsLog2(size);
    if (index > OS_MAX_MULTI_DLNK_LOG2) {
        return NULL;
    } else if (index < OS_MIN_MULTI_DLNK_LOG2) {
        index = OS_MIN_MULTI_DLNK_LOG2;
    } else if (index != OS_MAX_MULTI_DLNK_LOG2 && index != OS_MIN_MULTI_DLNK_LOG2) {
        index++;
    }

    return dlinkHead->listHead + (index - OS_MIN_MULTI_DLNK_LOG2);
}

#ifdef __cplusplus
#if __cplusplus
```

#### 4.1.3 ./liteos\_\_a/kernel/base/mem/bestfit/los\_memory.c

修改寻找空闲块的函数，将 `OsDlnkMultiHead` 修改为 `OsDlnkMultiHead_NEXT`，并且添加调试信息。

```
/*
 * Description : find suitable free block use "good fit" algorithm
 * Input      : pool      --- Pointer to memory pool
 *              allocSize --- Size of memory in bytes which note need allocate
 * Return     : NULL      --- no suitable block found
 *              tmpNode   --- pointer a suitable free block
 */
STATIC INLINE LosMemDynNode *OsMemFindSuitableFreeBlock(VOID *pool, UINT32
allocSize)
{
    LOS_DL_LIST *listNodeHead = NULL;
    LosMemDynNode *tmpNode = NULL;
    UINT32 maxCount = (LOS_MemPoolSizeGet(pool) / allocSize) << 1;
    UINT32 count;
    UINT32 listNodeCount = 0;
#ifdef LOSCFG_MEM_HEAD_BACKUP
    UINT32 ret = LOS_OK;
#endif
    for (listNodeHead = OS_MEM_HEAD_NEXT(pool, allocSize); listNodeHead !=
NULL;
        listNodeHead = OsDlnkNextMultiHead(OS_MEM_HEAD_ADDR(pool),
listNodeHead)) { // edit this
        count = 0;
```


```

        LOS_DL_LIST_FOR_EACH_ENTRY(tmpNode, listNodeHead, LosMemDynNode,
selfNode.freeNodeInfo) {
            listNodeCount++;
            if (count++ >= maxCount) {
                PRINT_ERR("[%s:%d]node: execute too much time\n", __FUNCTION__,
__LINE__);
                break;
            }

#ifdef LOSCFG_MEM_HEAD_BACKUP
            if (!OsMemChecksumVerify(&tmpNode->selfNode)) {
                PRINT_ERR("[%s]: the node information of current node is bad
!!\n", __FUNCTION__);
                OsMemDispCtlNode(&tmpNode->selfNode);
                ret = OsMemBackupRestore(pool, tmpNode);
            }
            if (ret != LOS_OK) {
                break;
            }
#endif

            if (((UINTPTR)tmpNode & (OS_MEM_ALIGN_SIZE - 1)) != 0) {
                LOS_Panic("[%s:%d]Mem node data
error:OS_MEM_HEAD_ADDR(pool)=%p, listNodeHead:%p,"
                        "allocSize=%u, tmpNode=%p\n",
                        __FUNCTION__, __LINE__, OS_MEM_HEAD_ADDR(pool),
listNodeHead, allocSize, tmpNode);
                break;
            }
            if (tmpNode->selfNode.sizeAndFlag >= allocSize) {
                if(allocSize == 1212 || allocSize == 72){ // 数值仅做测试
                    PRINTK("proj4:listNodeCount=%d, tmpNode=%p, allocSize=%d,
tmpNode->selfNode.sizeAndFlag=%d\n",
                        listNodeCount, tmpNode, allocSize, tmpNode-
>selfNode.sizeAndFlag);
                }
                return tmpNode;
            }
        }
    }
    return NULL;
}

```

```
Open ▾  los_memory.c ~/openharmony/kernel/liteos_a/kernel/base/mem/bestfit Save ≡ ◀ ▶ ⌂ ✕

} while (0);

/*
 * Description : find suitable free block use "good fit" algorithm
 * Input      : pool      --- Pointer to memory pool
 *              allocSize  --- Size of memory in bytes which need allocate
 * Return     : NULL      --- no suitable block found
 *              tmpNode    --- pointer a suitable free block
 */
STATIC INLINE LosMemDynNode *OsMemFindSuitableFreeBlock(VOID *pool, UINT32 allocSize)
{
    LOS_DL_LIST *listNodeHead = NULL;
    LosMemDynNode *tmpNode = NULL;
    UINT32 maxCount = (LOS_MemPoolSizeGet(pool) / allocSize) << 1;
    UINT32 count;
    UINT32 listNodeCount = 0;
#ifdef LOSCFG_MEM_HEAD_BACKUP
    UINT32 ret = LOS_OK;
#endif
    for (listNodeHead = OS_MEM_HEAD_NEXT(pool, allocSize); listNodeHead != NULL;
        listNodeHead = OsDlnkNextMultiHead(OS_MEM_HEAD_ADDR(pool), listNodeHead)) {
        count = 0;
        LOS_DL_LIST_FOR_EACH_ENTRY(tmpNode, listNodeHead, LosMemDynNode,
selfNode.freeNodeInfo) {
            listNodeCount++;
        }
    }
}

C ▾ Tab Width: 8 ▾ Ln 740, Col 50 ▾ INS
```

## 4.2 在系统调用中检验分配

此部分前文内容同project1，仅在调用函数上有差异。


### 4.2.1 新增系统调用号

#### 4.2.1.1 ./prebuilts/lite/sysroot/usr/include/arm-liteos/bits/syscall.h

在最下方添加用于内核态的系统调用号 `__NR_hxsyscall` 和用于用户态的系统调用号

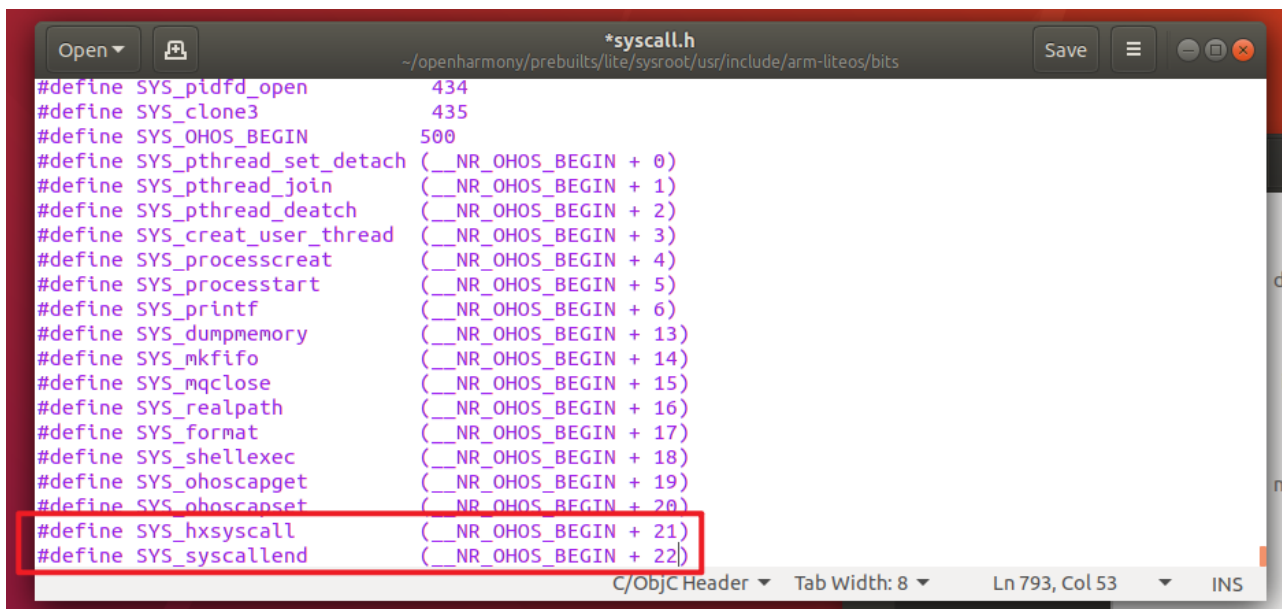
`SYS_hxsyscall`

同时，需要将 `__NR_syscallend` 数值增加一，用于系统调用号的边界判断

```
Open ▾  *syscall.h ~/openharmony/prebuilts/lite/sysroot/usr/include/arm-liteos/bits Save ≡ ◀ ▶ ⌂ ✕

/* OHOS customized syscalls, not compatible with ARM EABI */
#define __NR_OHOS_BEGIN 500
#define __NR_pthread_set_detach (__NR_OHOS_BEGIN + 0)
#define __NR_pthread_join (__NR_OHOS_BEGIN + 1)
#define __NR_pthread_deatch (__NR_OHOS_BEGIN + 2)
#define __NR_creat_user_thread (__NR_OHOS_BEGIN + 3)
#define __NR_processcreat (__NR_OHOS_BEGIN + 4)
#define __NR_processtart (__NR_OHOS_BEGIN + 5)
#define __NR_printf (__NR_OHOS_BEGIN + 6)
#define __NR_dumpmemory (__NR_OHOS_BEGIN + 13)
#define __NR_mkfifo (__NR_OHOS_BEGIN + 14)
#define __NR_mqclose (__NR_OHOS_BEGIN + 15)
#define __NR_realpath (__NR_OHOS_BEGIN + 16)
#define __NR_format (__NR_OHOS_BEGIN + 17)
#define __NR_shellexec (__NR_OHOS_BEGIN + 18)
#define __NR_ohoscapget (__NR_OHOS_BEGIN + 19)
#define __NR_ohoscapset (__NR_OHOS_BEGIN + 20)
#define __NR_hxsyscall (__NR_OHOS_BEGIN + 21)
#define __NR_syscallend (__NR_OHOS_BEGIN + 22)

C/ObjC Header ▾ Tab Width: 8 ▾ Ln 398, Col 41 ▾ INS
```

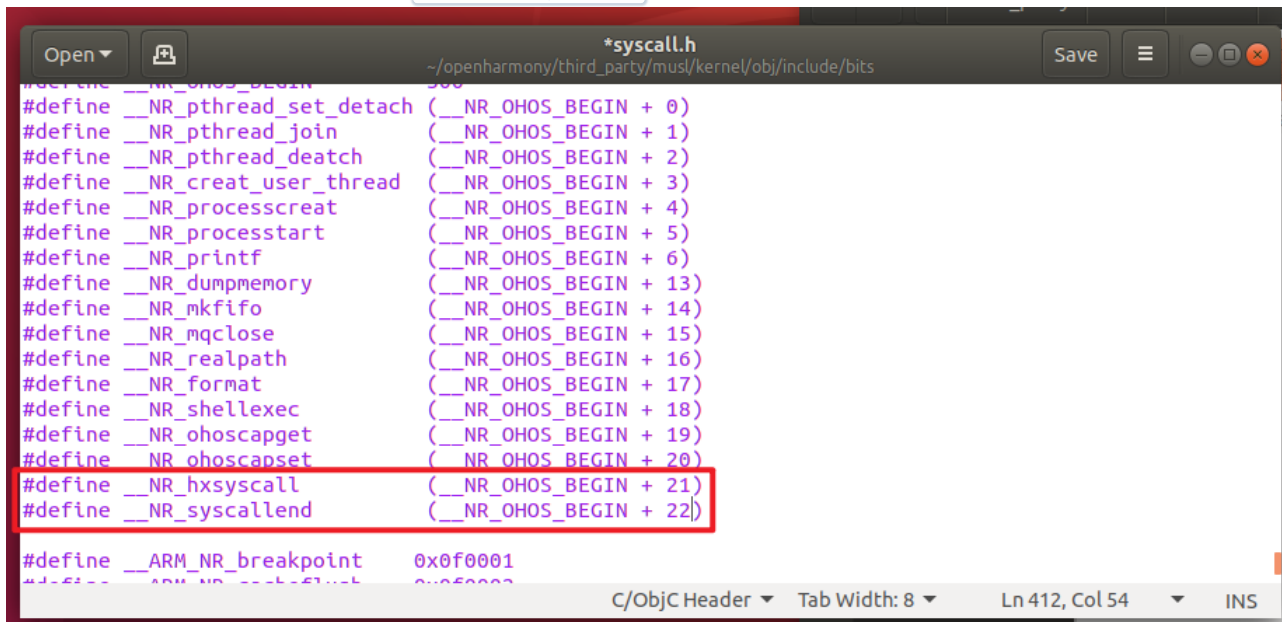


```
*syscall.h
~/openharmy/prebuilts/lite/sysroot/usr/include/arm-liteos/bits

#define SYS_pidfd_open 434
#define SYS_clone3 435
#define SYS_OHOS_BEGIN 500
#define SYS_pthread_set_detach (__NR_OHOS_BEGIN + 0)
#define SYS_pthread_join (__NR_OHOS_BEGIN + 1)
#define SYS_pthread_deatch (__NR_OHOS_BEGIN + 2)
#define SYS_creat_user_thread (__NR_OHOS_BEGIN + 3)
#define SYS_processcreat (__NR_OHOS_BEGIN + 4)
#define SYS_processtart (__NR_OHOS_BEGIN + 5)
#define SYS_printf (__NR_OHOS_BEGIN + 6)
#define SYS_dumpmemory (__NR_OHOS_BEGIN + 13)
#define SYS_mkfifo (__NR_OHOS_BEGIN + 14)
#define SYS_mqclose (__NR_OHOS_BEGIN + 15)
#define SYS_realpath (__NR_OHOS_BEGIN + 16)
#define SYS_format (__NR_OHOS_BEGIN + 17)
#define SYS_shellexec (__NR_OHOS_BEGIN + 18)
#define SYS_ohoscapget (__NR_OHOS_BEGIN + 19)
#define SYS_ohoscapset (__NR_OHOS_BEGIN + 20)
#define SYS_hxsyscall (__NR_OHOS_BEGIN + 21)
#define SYS_syscallend (__NR_OHOS_BEGIN + 22)
```

#### 4.2.1.2 ./third\_party/musl/kernel/obj/include/bits/syscall.h

添加用于内核态的系统调用号 `__NR_hxsyscall`



```
*syscall.h
~/openharmy/third_party/musl/kernel/obj/include/bits

#define __NR_pthread_set_detach (__NR_OHOS_BEGIN + 0)
#define __NR_pthread_join (__NR_OHOS_BEGIN + 1)
#define __NR_pthread_deatch (__NR_OHOS_BEGIN + 2)
#define __NR_creat_user_thread (__NR_OHOS_BEGIN + 3)
#define __NR_processcreat (__NR_OHOS_BEGIN + 4)
#define __NR_processtart (__NR_OHOS_BEGIN + 5)
#define __NR_printf (__NR_OHOS_BEGIN + 6)
#define __NR_dumpmemory (__NR_OHOS_BEGIN + 13)
#define __NR_mkfifo (__NR_OHOS_BEGIN + 14)
#define __NR_mqclose (__NR_OHOS_BEGIN + 15)
#define __NR_realpath (__NR_OHOS_BEGIN + 16)
#define __NR_format (__NR_OHOS_BEGIN + 17)
#define __NR_shellexec (__NR_OHOS_BEGIN + 18)
#define __NR_ohoscapget (__NR_OHOS_BEGIN + 19)
#define __NR_ohoscapset (__NR_OHOS_BEGIN + 20)
#define __NR_hxsyscall (__NR_OHOS_BEGIN + 21)
#define __NR_syscallend (__NR_OHOS_BEGIN + 22)

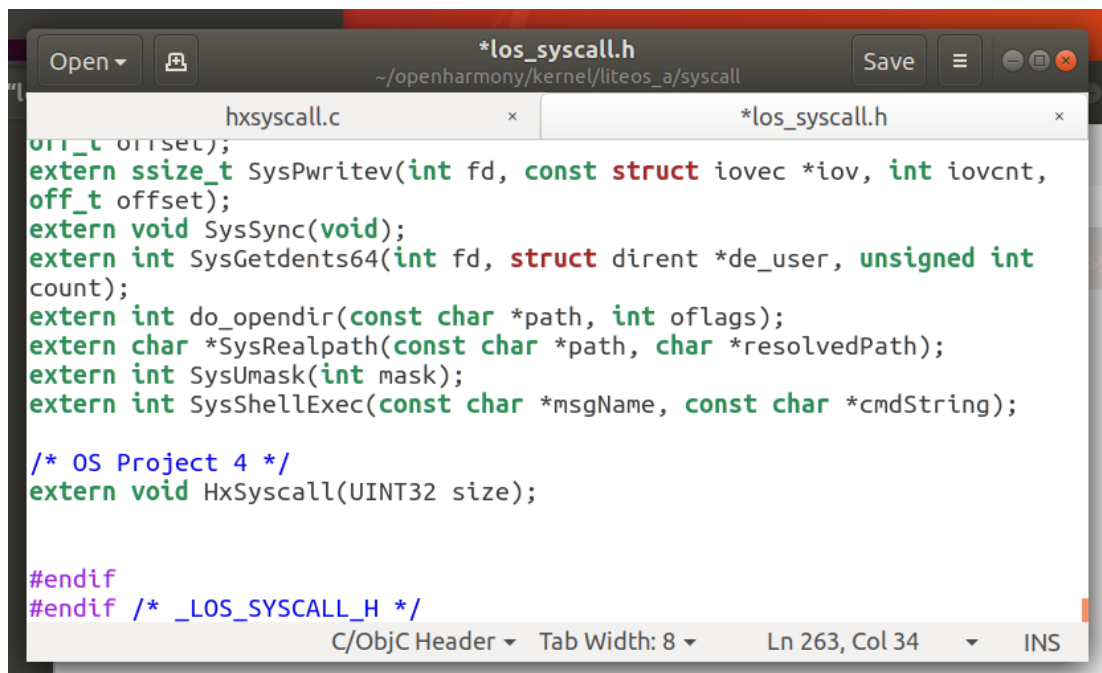
#define __ARM_NR_breakpoint 0x0f0001
#define __ARM_NR_watchpoint 0x0f0002
```

#### 4.2.2 新增系统调用的函数声明

##### 4.2.2.1 /home/book/openharmony/kernel/liteos\_a/syscall/los\_syscall.h

新增系统调用函数 `HxSyscall` 声明。当发生对应的系统调用时，该函数将被执行

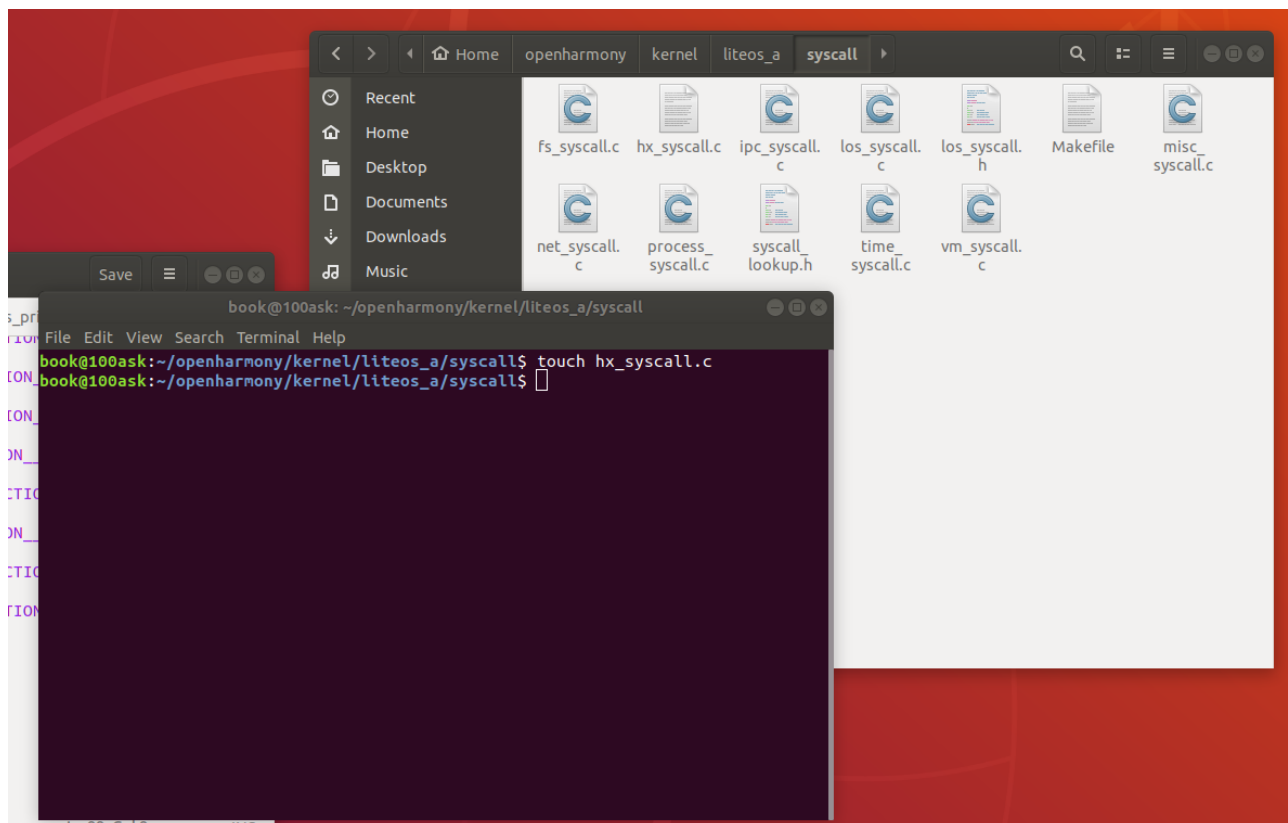




### 4.2.3 新增系统调用的函数实现

不难发现，所有在 `los_syscall.h` 文件中使用 `extern` 修饰的函数声明，其实现都定义在同级目录下的 `.c` 文件中

因此，可以在 `los_syscall.c` 文件的同级目录下，新建一个 `hx_syscall.c` 文件，并在其中定义 `HxSyscall` 函数的实现



编写程序：

```
#include "los_printf.h"
```

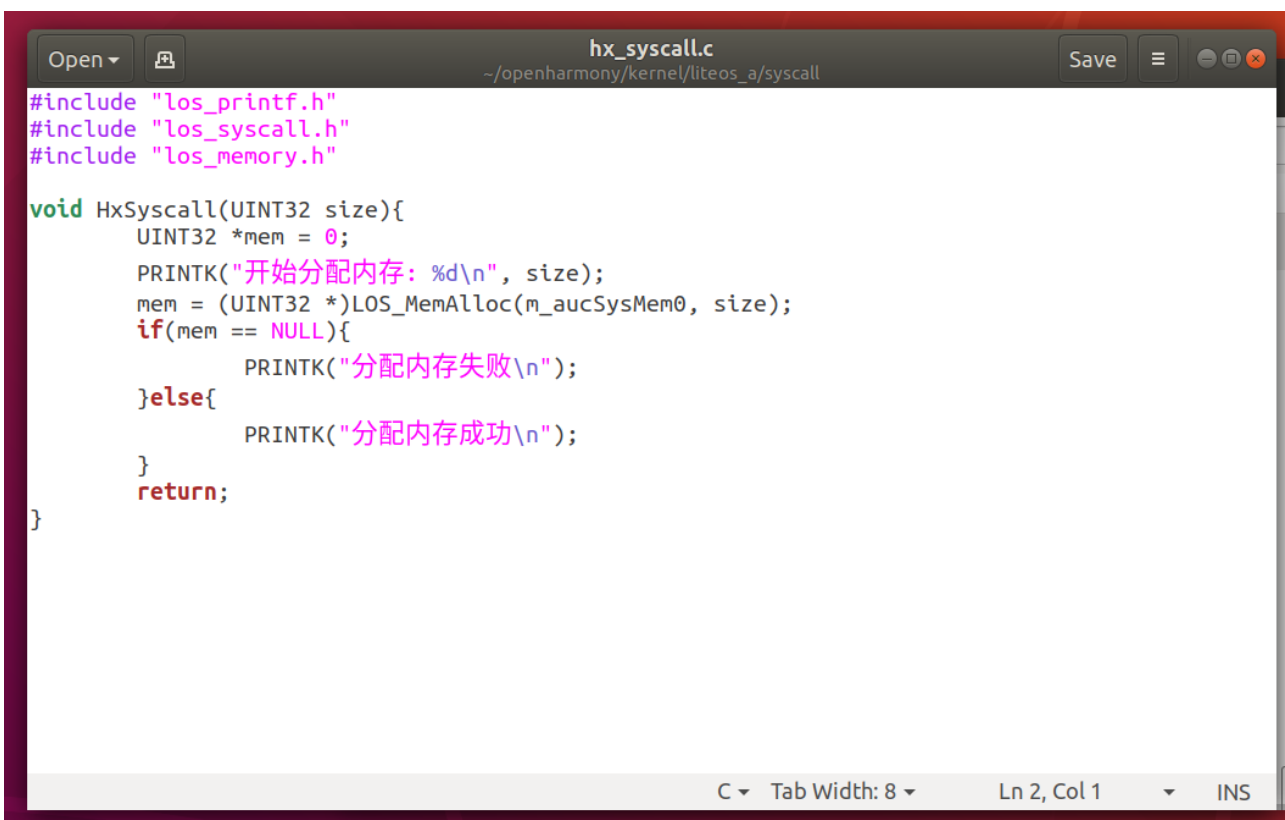


```

#include "los_syscall.h"
#include "los_memory.h"

void HxSyscall(UINT32 size){
    UINT32 *mem = 0;
    PRINTK("开始分配内存: %d\n", size);
    mem = (UINT32 *)LOS_MemAlloc(m_aucSysMem0, size); // 分配内存
    if(mem == NULL){
        PRINTK("分配内存失败\n");
    }else{
        PRINTK("分配内存成功\n");
    }
    return;
}

```



```

hx_syscall.c
~/openharmony/kernel/liteos_a/syscall

#include "los_printf.h"
#include "los_syscall.h"
#include "los_memory.h"

void HxSyscall(UINT32 size){
    UINT32 *mem = 0;
    PRINTK("开始分配内存: %d\n", size);
    mem = (UINT32 *)LOS_MemAlloc(m_aucSysMem0, size);
    if(mem == NULL){
        PRINTK("分配内存失败\n");
    }else{
        PRINTK("分配内存成功\n");
    }
    return;
}

C Tab Width: 8 Ln 2, Col 1 INS

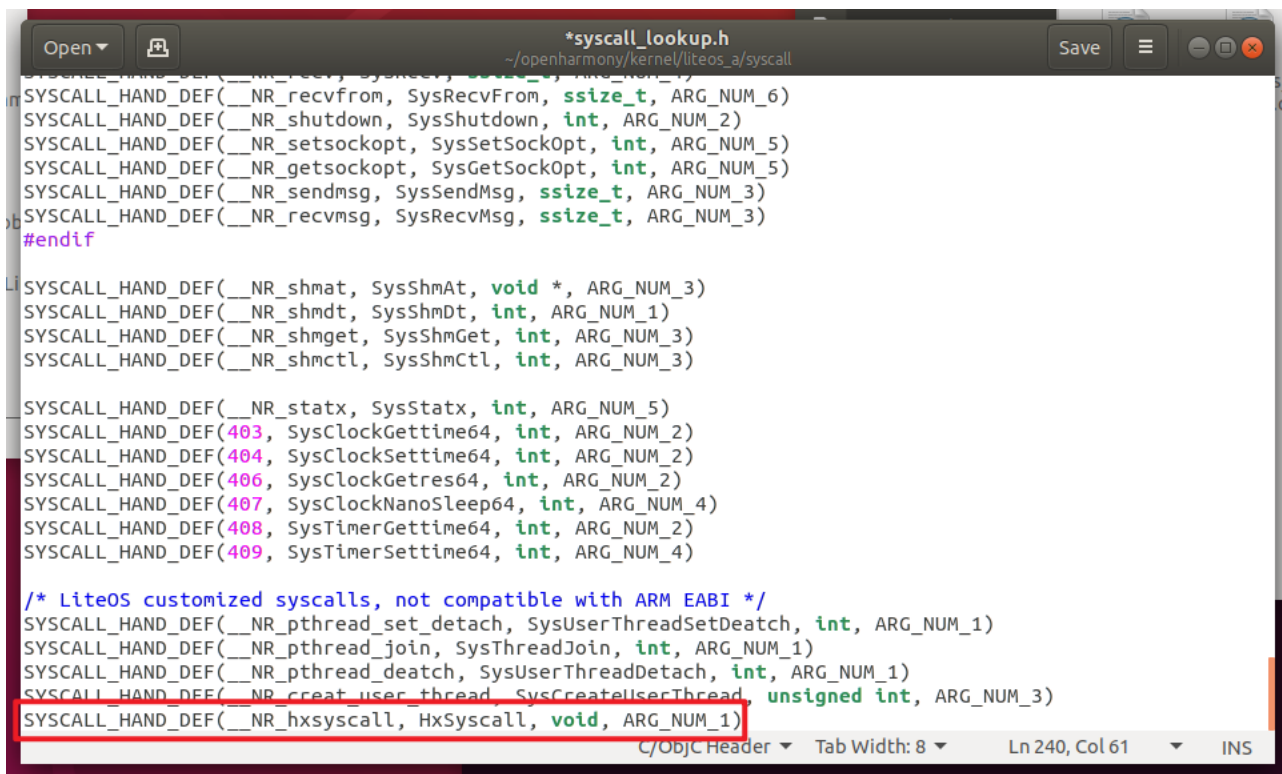
```

此处使用 LOS\_MemAlloc 申请内存空间

#### 4.2.4 新增系统调用号和系统调用函数的映射关系

##### 4.2.4.1 /home/book/openharmony/kernel/liteos\_a/syscall/syscall\_lookup.h

新增系统调用号 `__NR_hxsyscall` 和系统调用函数 `HxSyscall` 之间的映射关系，返回值类型为 `void`，参数个数为 `1`



```
*syscall_lookup.h
~/openharmony/kernel/liteos_a/syscall

SYSCALL_HAND_DEF(__NR_recvfrom, SysRecvFrom, ssize_t, ARG_NUM_6)
SYSCALL_HAND_DEF(__NR_shutdown, SysShutdown, int, ARG_NUM_2)
SYSCALL_HAND_DEF(__NR_setsockopt, SysSetSockOpt, int, ARG_NUM_5)
SYSCALL_HAND_DEF(__NR_getsockopt, SysGetSockOpt, int, ARG_NUM_5)
SYSCALL_HAND_DEF(__NR_sendmsg, SysSendMsg, ssize_t, ARG_NUM_3)
SYSCALL_HAND_DEF(__NR_recvmsg, SysRecvMsg, ssize_t, ARG_NUM_3)
#endif

SYSCALL_HAND_DEF(__NR_shmat, SysShmAt, void *, ARG_NUM_3)
SYSCALL_HAND_DEF(__NR_shmdt, SysShmDt, int, ARG_NUM_1)
SYSCALL_HAND_DEF(__NR_shmget, SysShmGet, int, ARG_NUM_3)
SYSCALL_HAND_DEF(__NR_shmctl, SysShmCtl, int, ARG_NUM_3)

SYSCALL_HAND_DEF(__NR_statx, SysStatx, int, ARG_NUM_5)
SYSCALL_HAND_DEF(403, SysClockGettime64, int, ARG_NUM_2)
SYSCALL_HAND_DEF(404, SysClockSettime64, int, ARG_NUM_2)
SYSCALL_HAND_DEF(406, SysClockGetres64, int, ARG_NUM_2)
SYSCALL_HAND_DEF(407, SysClockNanoSleep64, int, ARG_NUM_4)
SYSCALL_HAND_DEF(408, SysTimerGettime64, int, ARG_NUM_2)
SYSCALL_HAND_DEF(409, SysTimerSettime64, int, ARG_NUM_4)

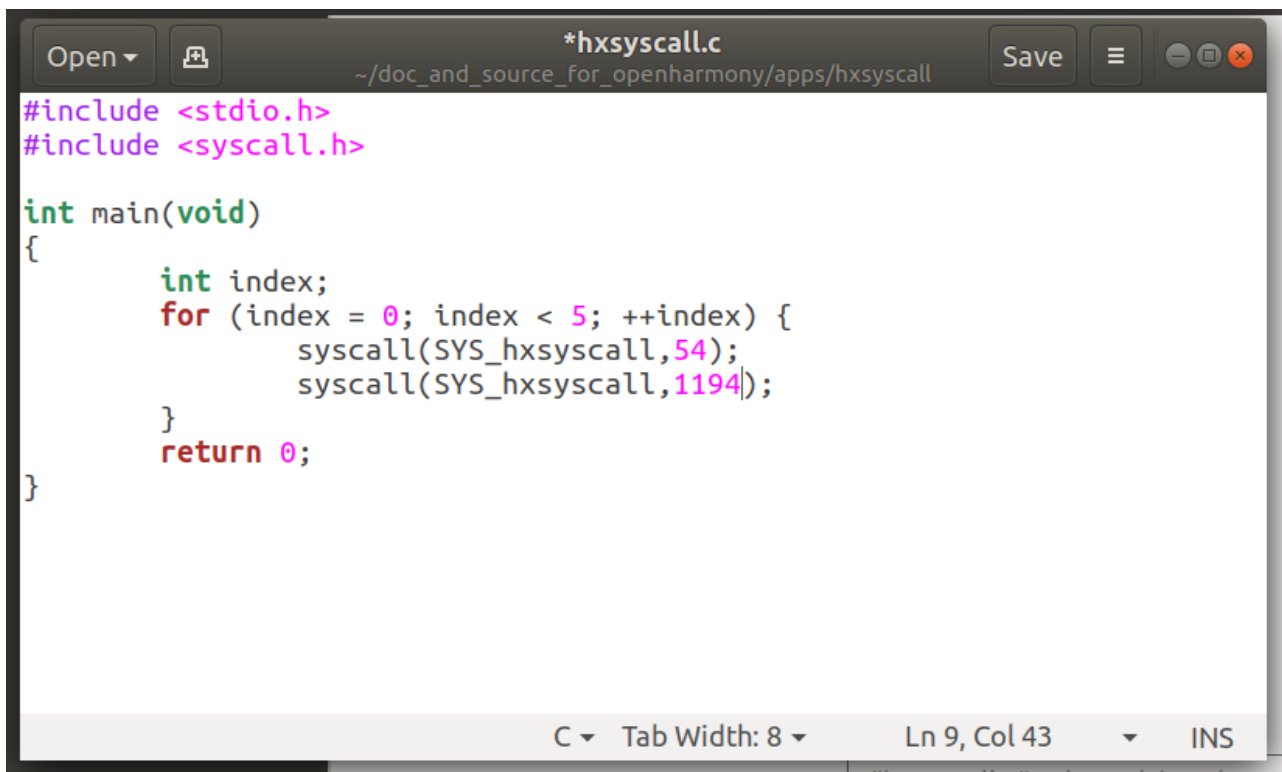
/* LiteOS customized syscalls, not compatible with ARM EABI */
SYSCALL_HAND_DEF(__NR_pthread_set_detach, SysUserThreadSetDeatch, int, ARG_NUM_1)
SYSCALL_HAND_DEF(__NR_pthread_join, SysThreadJoin, int, ARG_NUM_1)
SYSCALL_HAND_DEF(__NR_pthread_deatch, SysUserThreadDetach, int, ARG_NUM_1)
SYSCALL_HAND_DEF(__NR_creat_user_thread, SysCreateUserThread, unsigned int, ARG_NUM_3)
SYSCALL_HAND_DEF(__NR_hxsyscall, HxSyscall, void, ARG_NUM_1)
```

### 4.3 烧录运行

编写 `hxsyscall.c` 文件，直接在 `main` 函数中执行调用号为 `SYS_hxsyscall` 的系统调用

ps: 54和1194两个数是测试得到的，也可以换成其他的，但4.1.3相对应函数中的数也要改，具体可看实验结果分析

```
#include <stdio.h>
#include <syscall.h>
int main(void)
{
    int index;
    for (index = 0; index < 5; ++index) {
        syscall(SYS_hxsyscall, 54);
        syscall(SYS_hxsyscall, 1194);
    }
    return 0;
}
```



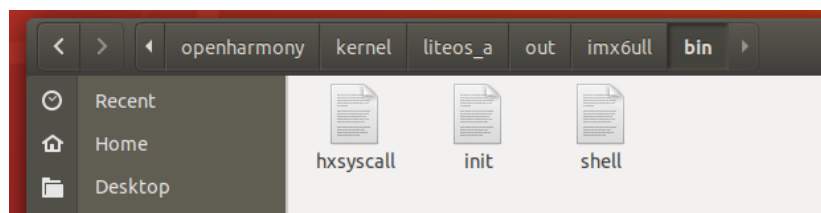
The screenshot shows a code editor window titled `*hxsyscall.c` with the file path `~/doc_and_source_for_openharmony/apps/hxsyscall`. The code is as follows:

```
#include <stdio.h>
#include <syscall.h>

int main(void)
{
    int index;
    for (index = 0; index < 5; ++index) {
        syscall(SYS_hxsyscall, 54);
        syscall(SYS_hxsyscall, 1194);
    }
    return 0;
}
```

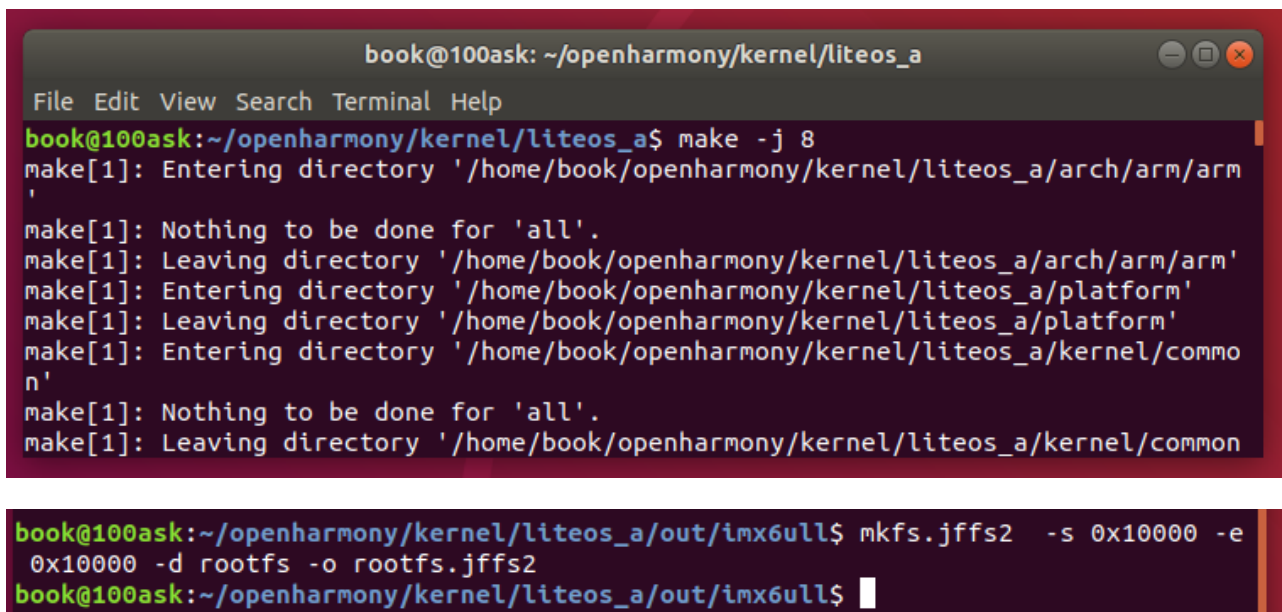
The status bar at the bottom indicates `C`, `Tab Width: 8`, `Ln 9, Col 43`, and `INS`.

对 `hxsyscall.c` 文件进行编译，然后复制到 `rootfs` 目录下的 `bin` 目录中，重新制作 `rootfs.jffs2`



重新编译和烧录

```
mkfs.jffs2 -s 0x10000 -e 0x10000 -d rootfs -o rootfs.jffs2
```



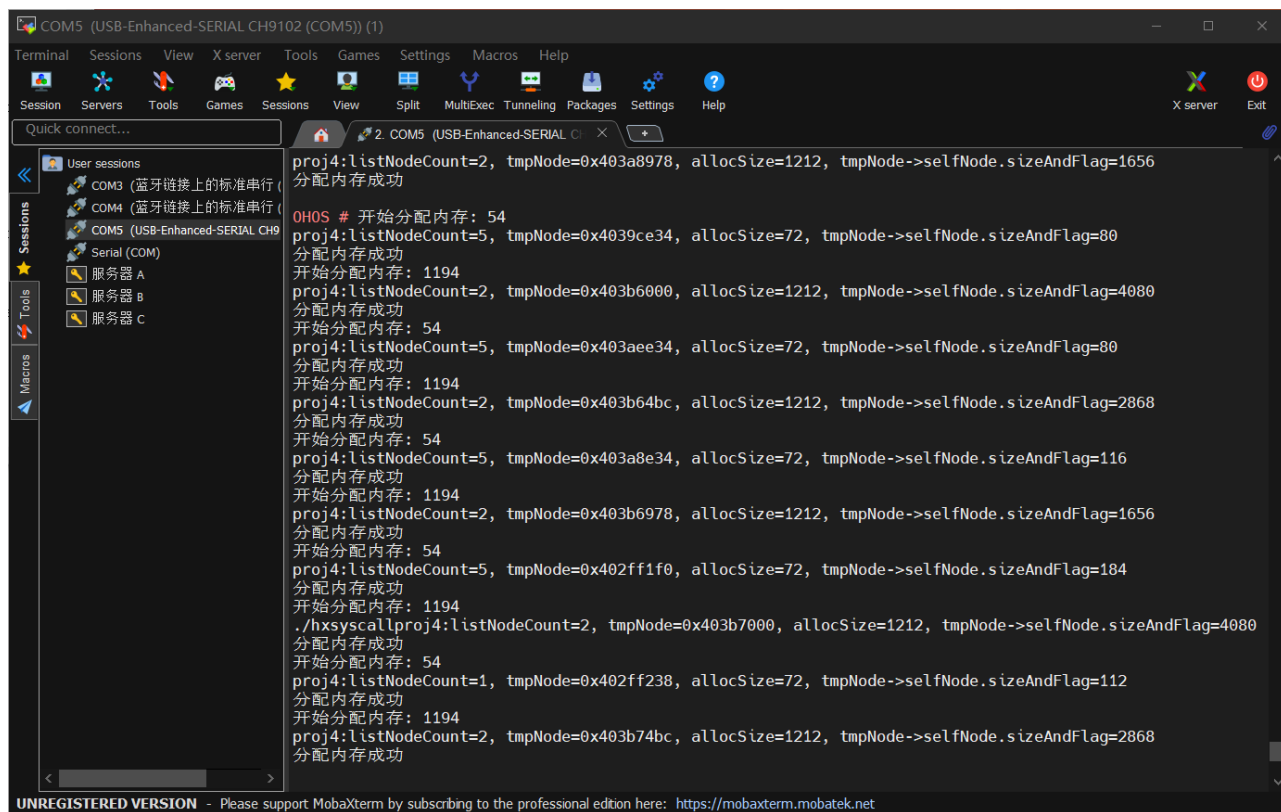
The screenshot shows a terminal window with the following commands and output:

```
book@100ask: ~/openharmony/kernel/liteos_a
File Edit View Search Terminal Help
book@100ask:~/openharmony/kernel/liteos_a$ make -j 8
make[1]: Entering directory '/home/book/openharmony/kernel/liteos_a/arch/arm/arm'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/home/book/openharmony/kernel/liteos_a/arch/arm/arm'
make[1]: Entering directory '/home/book/openharmony/kernel/liteos_a/platform'
make[1]: Leaving directory '/home/book/openharmony/kernel/liteos_a/platform'
make[1]: Entering directory '/home/book/openharmony/kernel/liteos_a/kernel/common'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/home/book/openharmony/kernel/liteos_a/kernel/common'

book@100ask:~/openharmony/kernel/liteos_a/out/imx6ull$ mkfs.jffs2 -s 0x10000 -e
0x10000 -d rootfs -o rootfs.jffs2
book@100ask:~/openharmony/kernel/liteos_a/out/imx6ull$
```

## 5 实验结果与分析

### 5.1 best-fit 算法结果



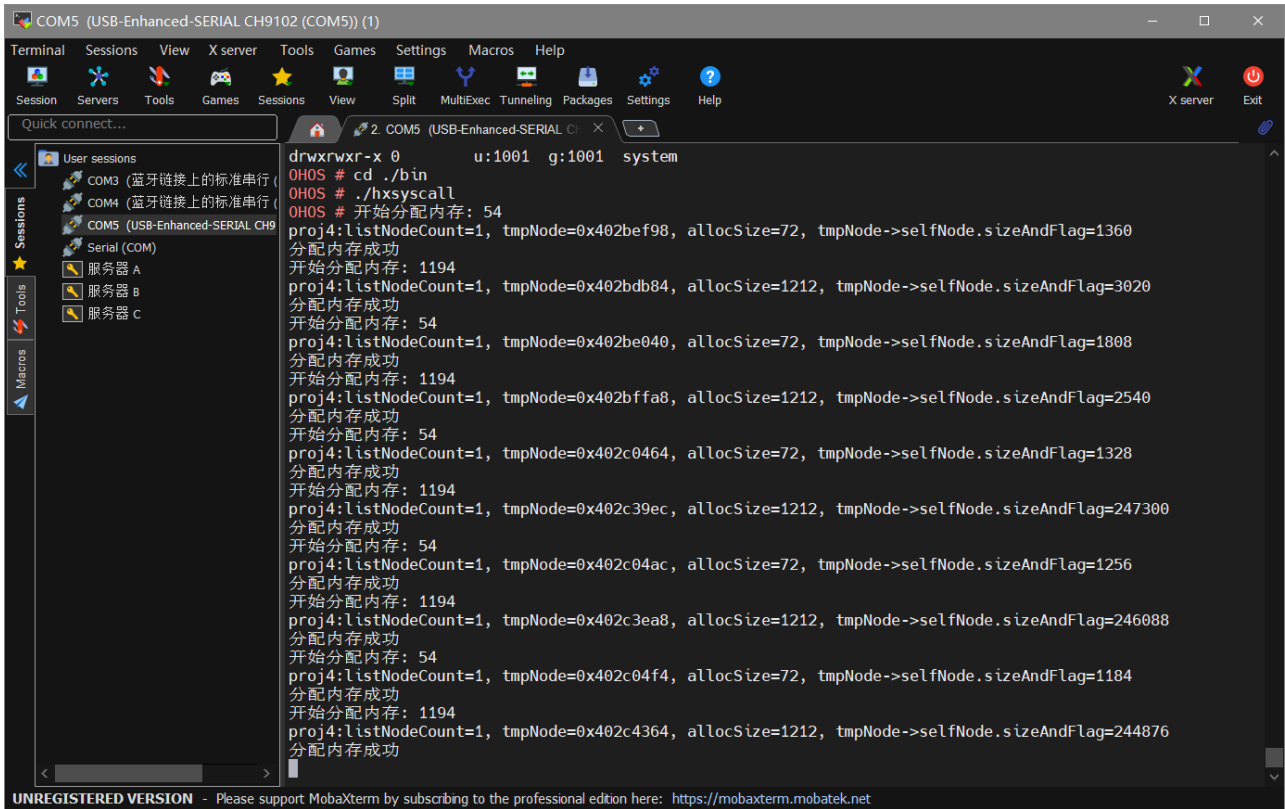
The screenshot shows a MobaXterm terminal window titled 'COM5 (USB-Enhanced-SERIAL CH9102 (COM5)) (1)'. The terminal displays the output of a memory allocation program using the best-fit algorithm. The output shows a series of allocation attempts for different project sizes (proj4) with varying listNodeCount, tmpNode addresses, allocSize, and selfNode.sizeAndFlag. The program repeatedly prints '分配内存成功' (Memory allocation successful) and '开始分配内存: 1194' (Start allocating memory: 1194). The allocSize values are consistently larger than the requested size, indicating the overhead of the OS and memory alignment. The terminal also shows a sidebar with 'User sessions' and 'Servers'.

```
COM5 (USB-Enhanced-SERIAL CH9102 (COM5)) (1)
Terminal Sessions View X server Tools Games Settings Macros Help
Session Servers Tools Games Sessions View Split MultiExec Tunneling Packages Settings Help
Quick connect...
User sessions
COM3 (蓝牙链接上的标准串行 (
COM4 (蓝牙链接上的标准串行 (
COM5 (USB-Enhanced-SERIAL CH9
Serial (COM)
服务器 A
服务器 B
服务器 C
proj4:listNodeCount=2, tmpNode=0x403a8978, allocSize=1212, tmpNode->selfNode.sizeAndFlag=1656
分配内存成功
OH0S # 开始分配内存: 54
proj4:listNodeCount=5, tmpNode=0x4039ce34, allocSize=72, tmpNode->selfNode.sizeAndFlag=80
分配内存成功
开始分配内存: 1194
proj4:listNodeCount=2, tmpNode=0x403b6000, allocSize=1212, tmpNode->selfNode.sizeAndFlag=4080
分配内存成功
开始分配内存: 54
proj4:listNodeCount=5, tmpNode=0x403aee34, allocSize=72, tmpNode->selfNode.sizeAndFlag=80
分配内存成功
开始分配内存: 1194
proj4:listNodeCount=2, tmpNode=0x403b64bc, allocSize=1212, tmpNode->selfNode.sizeAndFlag=2868
分配内存成功
开始分配内存: 54
proj4:listNodeCount=5, tmpNode=0x403a8e34, allocSize=72, tmpNode->selfNode.sizeAndFlag=116
分配内存成功
开始分配内存: 1194
proj4:listNodeCount=2, tmpNode=0x403b6978, allocSize=1212, tmpNode->selfNode.sizeAndFlag=1656
分配内存成功
开始分配内存: 54
proj4:listNodeCount=5, tmpNode=0x402ff1f0, allocSize=72, tmpNode->selfNode.sizeAndFlag=184
分配内存成功
开始分配内存: 1194
./hxsyscallproj4:listNodeCount=2, tmpNode=0x403b7000, allocSize=1212, tmpNode->selfNode.sizeAndFlag=4080
分配内存成功
开始分配内存: 54
proj4:listNodeCount=1, tmpNode=0x402ff238, allocSize=72, tmpNode->selfNode.sizeAndFlag=112
分配内存成功
开始分配内存: 1194
proj4:listNodeCount=2, tmpNode=0x403b74bc, allocSize=1212, tmpNode->selfNode.sizeAndFlag=2868
分配内存成功
UNREGISTERED VERSION - Please support MobaXterm by subscribing to the professional edition here: https://mobaxterm.mobatek.net
```

可以看到 best-fit 方法总需要遍历几个节点才能够找到空闲节点，多次运行hxsyscall得到的结果不同。

allocSize 比申请的 size 来得大的原因是存在 OS\_MEM\_NODE\_HEAD\_SIZE 以及内存对齐。

### 5.2 good-fit 算法结果



```
COM5 (USB-Enhanced-SERIAL CH9102 (COM5)) (1)
Terminal Sessions View X server Tools Games Settings Macros Help
Session Servers Tools Games Sessions View Split MultiExec Tunneling Packages Settings Help
Quick connect...
User sessions
COM3 (蓝牙链接上的标准串行 (
COM4 (蓝牙链接上的标准串行 (
COM5 (USB-Enhanced-SERIAL CH9
Serial (COM)
服务器 A
服务器 B
服务器 C
drwxrwxr-x 0 u:1001 g:1001 system
OHOS # cd ./bin
OHOS # ./hxsyscall
OHOS # 开始分配内存: 54
proj4:listNodeCount=1, tmpNode=0x402bef98, allocSize=72, tmpNode->selfNode.sizeAndFlag=1360
分配内存成功
开始分配内存: 1194
proj4:listNodeCount=1, tmpNode=0x402bdb84, allocSize=1212, tmpNode->selfNode.sizeAndFlag=3020
分配内存成功
开始分配内存: 54
proj4:listNodeCount=1, tmpNode=0x402be040, allocSize=72, tmpNode->selfNode.sizeAndFlag=1808
分配内存成功
开始分配内存: 1194
proj4:listNodeCount=1, tmpNode=0x402bffa8, allocSize=1212, tmpNode->selfNode.sizeAndFlag=2540
分配内存成功
开始分配内存: 54
proj4:listNodeCount=1, tmpNode=0x402c0464, allocSize=72, tmpNode->selfNode.sizeAndFlag=1328
分配内存成功
开始分配内存: 1194
proj4:listNodeCount=1, tmpNode=0x402c39ec, allocSize=1212, tmpNode->selfNode.sizeAndFlag=247300
分配内存成功
开始分配内存: 54
proj4:listNodeCount=1, tmpNode=0x402c04ac, allocSize=72, tmpNode->selfNode.sizeAndFlag=1256
分配内存成功
开始分配内存: 1194
proj4:listNodeCount=1, tmpNode=0x402c3ea8, allocSize=1212, tmpNode->selfNode.sizeAndFlag=246088
分配内存成功
开始分配内存: 54
proj4:listNodeCount=1, tmpNode=0x402c04f4, allocSize=72, tmpNode->selfNode.sizeAndFlag=1184
分配内存成功
开始分配内存: 1194
proj4:listNodeCount=1, tmpNode=0x402c4364, allocSize=1212, tmpNode->selfNode.sizeAndFlag=244876
分配内存成功
UNREGISTERED VERSION - Please support MobaXterm by subscribing to the professional edition here: https://mobaxterm.mobatek.net
```

可以多次运行，可以发现 `listNodeCount` 恒为1，也就是OS总能在检查第一个空闲块时便找到可以分配的块，满足了 good-fit 的  $O(1)$  要求

从运行的结果看，程序成功实现了project的目的！

## 6 实验分析

### 6.1 问题分析

#### 6.1.1 实际有内存但申请无内存块分配

```
COM5 (USB-Enhanced-SERIAL CH9102 (COM5)) (1)
Terminal Sessions View X server Tools Games Settings Macros Help
Session Servers Tools Games Sessions View Split MultiExec Tunneling Packages Settings Help
Quick connect...
User sessions
COM3 (蓝牙链接上的标准串行 (
COM4 (蓝牙链接上的标准串行 (
COM5 (USB-Enhanced-SERIAL CH9
Serial (COM)
服务器 A
服务器 B
服务器 C

OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f918, allocSize = 32780
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f918, allocSize = 32780
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f918, allocSize = 32780
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f918, allocSize = 32780
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f918, allocSize = 32780
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f918, allocSize = 32780
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f918, allocSize = 32780
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f918, allocSize = 32780
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f918, allocSize = 32780
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f918, allocSize = 32780
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f918, allocSize = 32780
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f918, allocSize = 32780
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f918, allocSize = 32780
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f918, allocSize = 32780
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f918, allocSize = 32780
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f918, allocSize = 32780
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f918, allocSize = 32780
[ERR]OsMemPoolExpand alloc failed size = 36864
[ERR]-----
pool addr      pool size  used size  free size  max free node size  used node num  free n
-----
0x4024f8ac     0x10dc754  0x0       0xb0660    0xb0660          0x0            0x1

[ERR][OsMemAllocWithCheck] No suitable free block, require free node size: 0x800c
[ERR]-----
[ERR]OsSwitchTmpTTB 734

UNREGISTERED VERSION - Please support MobaXterm by subscribing to the professional edition here: https://mobaxterm.mobatek.net
```

这个错误在我申请大块内存时出现，出现这个问题是当申请的内存较大时，我们直接进行index+1可能会出现超出TLFS算法的bitmap范围，或是较大块内存均已分配完。因此需要单独判断是否达到最大level，如果已经是最大level则不在下一块找。

6.1.2 疯狂输出，按键无反应

```
COM5 (USB-Enhanced-SERIAL CH9102 (COM5)) (1)
Terminal Sessions View X server Tools Games Settings Macros Help
Session Servers Tools Games Sessions View Split MultiExec Tunneling Packages Settings Help
Quick connect...
User sessions
COM3 (蓝牙链接上的标准串行 (
COM4 (蓝牙链接上的标准串行 (
COM5 (USB-Enhanced-SERIAL CH9
Serial (COM)
服务器 A
服务器 B
服务器 C

OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f8f0, allocSize = 32
进入 good-fit 逻辑
OsMemFindSuitableFreeBlock 利用 good-fit 找到合适的内存块 tmpNode = 0x403507f4, tmpNode->selfNode.sizeAn
dFlag = 2076, allocSize = 32
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f8c8, allocSize = 32
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f8d0, allocSize = 32
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f8d8, allocSize = 32
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f8e0, allocSize = 32
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f8e8, allocSize = 32
进入 good-fit 逻辑
OsMemFindSuitableFreeBlock 利用 good-fit 找到合适的内存块 tmpNode = 0x40350814, tmpNode->selfNode.sizeAn
dFlag = 2044, allocSize = 32
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f8f0, allocSize = 72
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f8f8, allocSize = 72
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f900, allocSize = 72
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f908, allocSize = 72
进入 good-fit 逻辑
OsMemFindSuitableFreeBlock 利用 good-fit 找到合适的内存块 tmpNode = 0x40252f8c, tmpNode->selfNode.sizeAn
dFlag = 18960, allocSize = 72
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f8c8, allocSize = 32
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f8d0, allocSize = 32
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f8d8, allocSize = 32
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f8e0, allocSize = 32
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f8e8, allocSize = 32
进入 good-fit 逻辑
OsMemFindSuitableFreeBlock 利用 good-fit 找到合适的内存块 tmpNode = 0x40350814, tmpNode->selfNode.sizeAn
dFlag = 2044, allocSize = 32
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f910, allocSize = 16412
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f918, allocSize = 16412
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f920, allocSize = 16412
OsMemFindSuitableFreeBlock 找到合适的内存范围 listNodeHead = 0x4024f928, allocSize = 16412
进入 good-fit 逻辑
OsMemFindSuitableFreeBlock 利用 good-fit 找到合适的内存块 tmpNode = 0x4029a7c0, tmpNode->selfNode.sizeAn
dFlag = 415792, allocSize = 16412

UNREGISTERED VERSION - Please support MobaXterm by subscribing to the professional edition here: https://mobaxterm.mobatek.net
```

不能在 `OsMemFindSuitableFreeBlock` 每次都输出分配结果，否则会无响应。

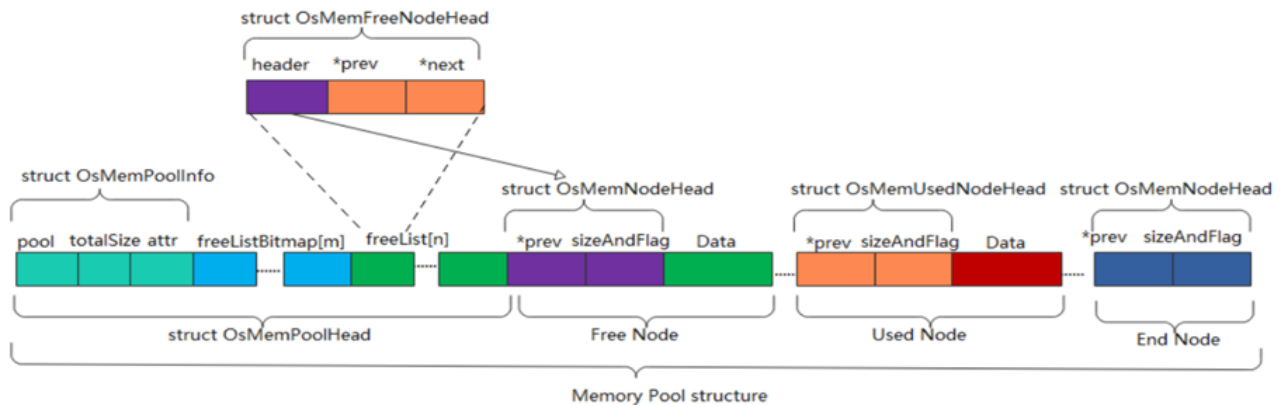


## 6.2 代码分析

### 6.2.1 动态内存结构体定义

分析这部分结构体非常重要，否则看不明白分配内存的算法的每句代码究竟是什么意义。

下面这张图很形象的描述了具体的内容。



#### 6.2.1.1 动态内存池信息结构体 `LosMemPoolInfo`

在 `kernel\base\include\los_memory_pri.h` 中，定义了内存池信息结构体 `LosMemPoolInfo`。这是动态内存池的第一部分，维护内存池的开始地址和大小信息。动态内存 `bestfit` 算法和 `bestfit_little` 算法中都定义了该结构体，两个主要的成员是内存池开始地址 `.pool` 和内存池大小 `.poolSize`。

```
typedef struct {
    VOID          *pool;          /* 内存池的内存开始地址 */
    UINT32        poolSize;       /* 内存池大小 */

#ifdef LOSCFG_MEM_TASK_STAT
    Memstat      stat;
#endif

#ifdef LOSCFG_MEM_MUL_POOL
    VOID          *nextPool;
#endif

#ifdef LOSCFG_KERNEL_MEM_SLAB_EXTENTION
    struct LosSlabControlHeader slabCtrlHdr;
#endif
} LosMemPoolInfo;
```

#### 6.2.1.2 多双向链表表头结构体 `LosMultipleDlinkHead`

在文档 `kernel\base\include\los_multipliedlinkhead_pri.h` 中，定义了内存池多双向链表表头结构体 `LosMultipleDlinkHead`。这是动态内存池的第二部分，结构体本身是一个数组，每个元素是一个双向链表，所有 `free` 节点的控制头都会被分类挂在这个数组的双向链表中。



(关键) 假设内存允许的最小节点为  $2^{\min}$  字节, 则数组的第一个双向链表存储的是所有  $\text{size}$  为  $2^{\min} < \text{size} < 2^{(\min+1)}$  的 `free` 节点, 第二个双向链表存储的是所有  $\text{size}$  为  $2^{(\min+1)} < \text{size} < 2^{(\min+2)}$  的 `free` 节点, 依次类推第  $n$  个双向链表存储的是所有  $\text{size}$  为  $2^{(\min+n-1)} < \text{size} < 2^{(\min+n)}$  的 `free` 节点。每次申请内存的时候, 会从这个数组检索最合适大小的 `free` 节点以分配内存。每次释放内存时, 会将该内存作为 `free` 节点存储至这个数组以便下次再使用。

结构体源代码如下, 非常简单, 是一个长度为 `OS_MULTI_DLNK_NUM` 的双向链表数组。

```
typedef struct {
    LOS_DL_LIST listHead[OS_MULTI_DLNK_NUM];
} LosMultipleDlinkHead;
```

针对内存分配算法的修改很多都与这部分有关。

### 6.2.2 函数 `OsDlnkMultiHead()`

我们修改了这个函数。函数需要2个参数, `VOID *headAddr` 为第二部分的多链表数组的起始地址, `UINT32 size` 为内存块的大小。该函数把内存池第三部分的内存块的大小映射到第二部分的链表位置上, 下方是源代码。

```
STATIC INLINE UINT32 OsLog2(UINT32 size)
{
(1)    return (size > 0) ? (UINT32)LOS_HighBitGet(size) : 0;
}

LITE_OS_SEC_TEXT_MINOR LOS_DL_LIST *OsDlnkMultiHead(VOID *headAddr, UINT32
size)
{
    LosMultipleDlinkHead *dlinkHead = (LosMultipleDlinkHead *)headAddr;
(2)    UINT32 index = OsLog2(size);
    if (index > OS_MAX_MULTI_DLNK_LOG2) {
(3)        return NULL;
    } else if (index <= OS_MIN_MULTI_DLNK_LOG2) {
(4)        index = OS_MIN_MULTI_DLNK_LOG2;
    }

(5)    return dlinkHead->listHead + (index - OS_MIN_MULTI_DLNK_LOG2);
}
```

(1)处的函数 `OsLog2()` 名称中的 `Log` 是对数英文 `logarithm` 的缩写, 函数用于计算以2为底的对数的整数部分, 输入参数是内存池第三部分的内存块的大小 `size`, 输出是第二部分多链表数组的数组索引, 见代码片段(2)。

(3)处如果索引大于 `OS_MAX_MULTI_DLNK_LOG2`, 无法分配这么大的内存块, 返回 `NULL`。

(4)处如果索引小于等于 `OS_MIN_MULTI_DLNK_LOG2`, 则使用最小值作为索引。

(5)处返回多链表表头中的链表头节点。

### 6.2.3 函数 `LOS_MemAlloc()`

```
LITE_OS_SEC_TEXT VOID *LOS_MemAlloc(VOID *pool, UINT32 size)
{
    VOID *ptr = NULL;
    UINT32 intSave;

(1)  if ((pool == NULL) || (size == 0)) {
        return NULL;
    }

    if (g_MALLOC_HOOK != NULL) {
        g_MALLOC_HOOK();
    }

    MEM_LOCK(intSave);
    do {
(2)      if (OS_MEM_NODE_GET_USED_FLAG(size) ||
OS_MEM_NODE_GET_ALIGNED_FLAG(size)) {
            break;
        }

(3)      ptr = OsSlabMemAlloc(pool, size);
            if (ptr == NULL) {
(4)          ptr = OsMemAllocWithCheck(pool, size);
            }
        } while (0);
#ifdef LOSCFG_MEM_RECORDINFO
        OsMemRecordMalloc(ptr, size);
#endif
        OsLmsSetAfterMalloc(ptr);

    MEM_UNLOCK(intSave);

    LOS_TRACE(MEM_ALLOC, pool, (UINTPTR)ptr, size);
    return ptr;
}
```

(1)处对参数进行校验，内存池地址不能为空，申请的内存大小不能为0。

(2)处判断申请的内存大小是否已标记为使用或对齐。把下一个可用节点赋值给 `nodeTmp`。

(3)处如果支持 `SLAB`，则先尝试从 `SLAB` 中获取内存，否则执行(4)调用函数 `OsMemAllocWithCheck(pool, size)` 申请内存块。

#### 6.2.4 函数 `OsMemAllocWithCheck(pool, size)`

```
STATIC VOID *OsMemAllocWithCheck(LosMemPoolInfo *pool, UINT32 size)
{
    LosMemDynNode *allocNode = NULL;
    UINT32 allocSize;

#ifdef LOSCFG_BASE_MEM_NODE_INTEGRITY_CHECK
    LosMemDynNode *tmpNode = NULL;
    LosMemDynNode *preNode = NULL;
#endif
(1) const VOID *firstNode = (const VOID *)((UINT8 *)OS_MEM_HEAD_ADDR(pool) +
    OS_DLNK_HEAD_SIZE);

#ifdef LOSCFG_BASE_MEM_NODE_INTEGRITY_CHECK
    if (OsMemIntegrityCheck(pool, &tmpNode, &preNode)) {
        OsMemIntegrityCheckError(tmpNode, preNode);
        return NULL;
    }
#endif

(2) allocSize = OS_MEM_ALIGN(size + OS_MEM_NODE_HEAD_SIZE, OS_MEM_ALIGN_SIZE);
    allocNode = OsMemFindSuitableFreeBlock(pool, allocSize);
    if (allocNode == NULL) {
        OsMemInfoAlert(pool, allocSize);
        return NULL;
    }
(3) if ((allocSize + OS_MEM_NODE_HEAD_SIZE + OS_MEM_ALIGN_SIZE) <= allocNode->
    selfNode.sizeAndFlag) {
        OsMemSplitNode(pool, allocNode, allocSize);
    }
(4) OsMemListDelete(&allocNode->selfNode.freeNodeInfo, firstNode);
    OsMemSetMagicNumAndTaskID(allocNode);
    OS_MEM_NODE_SET_USED_FLAG(allocNode->selfNode.sizeAndFlag);
(5) OS_MEM_ADD_USED(&pool->stat, OS_MEM_NODE_GET_SIZE(allocNode->
    selfNode.sizeAndFlag),
        OS_MEM_TASKID_GET(allocNode));
    OsMemNodeDebugOperate(pool, allocNode, size);
(6) return (allocNode + 1);
}
```

(1)处获取内存池中第一个内存节点

(2)计算出对齐后的内存大小，然后调用函数 `OsMemFindSuitableFreeBlock()` 获取适合的内存块，如果找不到适合的内存块，函数返回 `NULL`。

(3)处如果找到的内存块大于需要的内存大小，则执行分割操作。

(4)处把已分配的内存节点从链表中删除，然后设置魔术字和使用该内存块的任务 `Id`，然后标记该内存块已使用。

(5)处如果开启宏 `LOSCFG_MEM_TASK_STAT`，还需要做些记录操作，自行分析即可。

(6)处返回内存块的数据区的地址，这个是通过内存控制节点 `+1` 定位到数据区内存地址实现的。申请内存完成，调用申请内存的函数中可以使用申请的内存了。

分析到此，我们就可知要完成本project的任务，需要修改函数

`OsMemFindSuitableFreeBlock()` 了。

## 7 实验总结

通过这次project，我成功地在 liteos 中修改了物理内存分配的方式，实现了 Good-fit 算法。在实现中有尝试不同的方法来达到目的，如直接修改分配函数、修改统一的宏等，也发现了操作系统代码的复杂性，牵一发而动全身，做修改的时候一定要注意会造成什么影响。

## 8 参考文献

tlsf算法思路简介: <https://www.jianshu.com/p/01743e834432>

代码阅读: <https://bbs.huaweicloud.com/blogs/260204>

## 9 附录

代码均附在步骤内。