

## 项目四 探究 LiteOS 中物理内存分配算法

22920212204396 黄子安

### 一、实验目的

1、理解 TLSF 算法，并根据实验要求改进分配算法

### 二、实验环境

1.物理机：windows 操作系统

2.VMware 虚拟机：ubuntu 18.04.6

3.开发板：imx6ull Mini

### 三、实验内容

**实验问题：**LiteOS 中的物理内存分配采用了 TLSF 算法，该算法较好地解决了最坏情况执行时间不确定(not bounded)或者复杂度过高(bounded with a too important bound)，以及碎片化问题(fragmentation)两个问题。

TLSF 算法仍存在优化空间，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。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[68~70]这一范围，而是比这个范围稍微大一点儿的范围(例如[71~73])。这样设计的好处就是[71~73]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。

通过后续分析代码发现 LiteOS 本身采用了 Good-fit，即会找稍微偏大一些的内存在进行分配

# 1、移植 TLSF

我们实验使用的鸿蒙 LiteOS 的 1.0 版本很不幸并没有实现 TLSF，但是通过查阅鸿蒙的历史发行版本可以发现在随后 1.1.0 版本中就支持了 TLSF 内存分配

更新说明

本版本完全继承了OpenHarmony 1.0的所有特性，并在OpenHarmony 1.0版本的基础上，对各模块进行了功能扩展和优化，详情请参考下表。

表 2 特性更新说明

类别	新增特性	修改特性	删除特性
内核	<ul style="list-style-type: none"><li>• LiteOS-M支持Cortex-M7、Cortex-M33和RISC-V芯片架构，新增对应的单板target样例。</li><li>• LiteOS-M支持MPU功能。</li><li>• LiteOS-M支持部分POSIX接口。</li><li>• LiteOS-M支持FatFS文件系统。</li><li>• LiteOS-M支持异常回调函数注册机制。</li><li>• LiteOS-M三方芯片易适配性架构调整。</li><li>• LiteOS-M、LiteOS-A支持堆内存预测功能，包括内存泄漏、脏内存、内存统计。</li><li>• LiteOS-M、LiteOS-A支持TLSF堆内存算法，提高内存申请和释放效率，降低碎片率。</li></ul>	LiteOS-A调度优化。	None

可以从对应[网站](#)上下载 1.1.0 版本的内核代码进行移植操作（这里从官网下载只是为了说明代码来源，在复现实验时直接使用附录里的修改后的 tlsf 代码即可，没必要去下载）

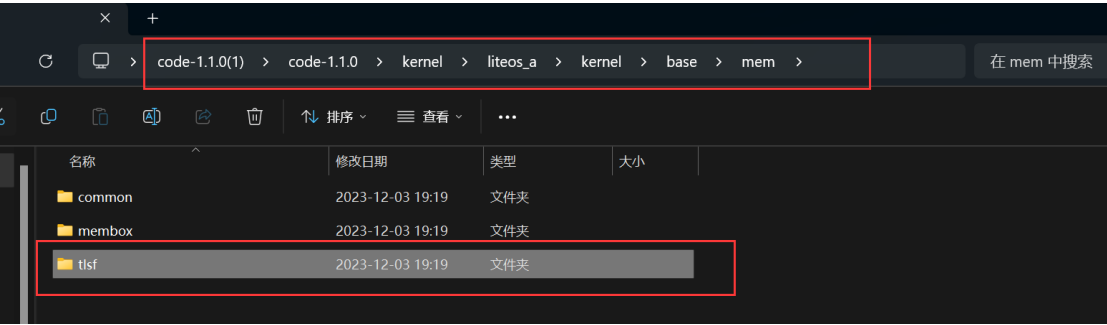
源码获取

通过镜像站点获取

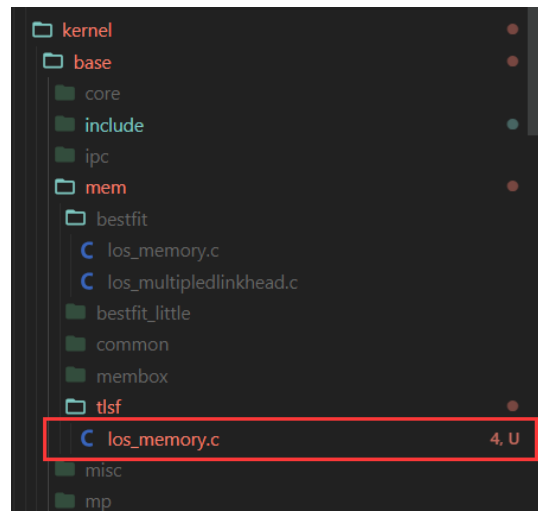
表 1 源码获取路径

版本源码	版本信息	下载站点	SHA256校验码
全量代码	1.1.0	<a href="#">站点</a>	SHA256 校验码
Hi3861解决方案（二进制）	1.1.0	<a href="#">站点</a>	SHA256 校验码
Hi3518解决方案（二进制）	1.1.0	<a href="#">站点</a>	SHA256 校验码
Hi3516解决方案（二进制）	1.1.0	<a href="#">站点</a>	SHA256 校验码
Release Notes	1.1.0	<a href="#">站点</a>	-

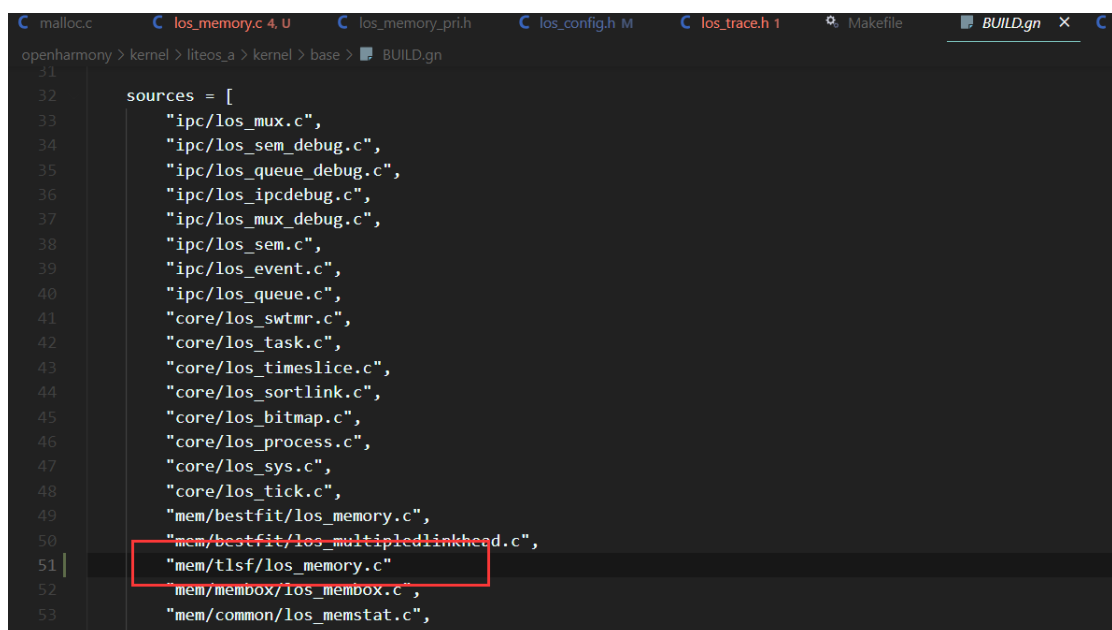
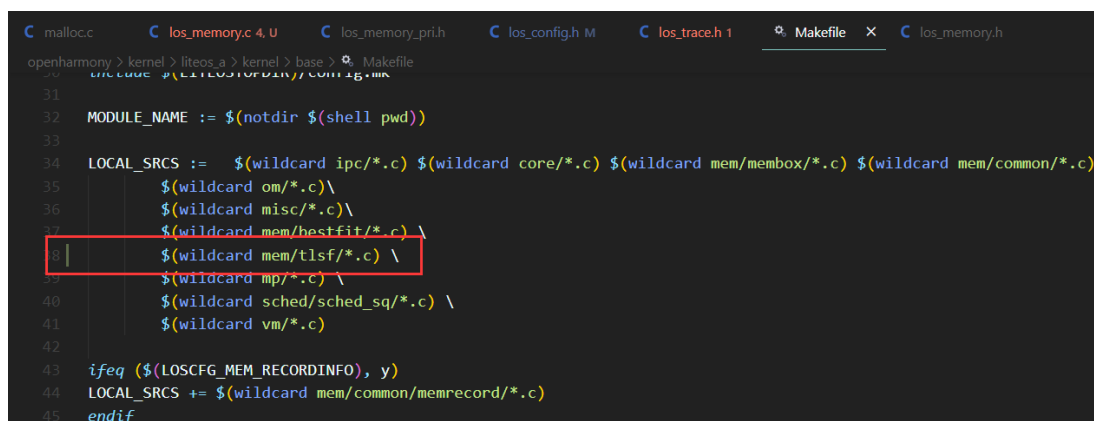
找到对应的文件 tlsf 文件，直接将其上传到虚拟机内核代码的相应位置



在 Liteos 中内存动态分配代码位于 mem 目录下，将文件拷贝到其中，之后将原来的 bestfit 中的 los\_memory.c 注释掉或者删除掉



在 Makefile 和 BUILD.gn 中增加新的文件目录，使得在编译的时候可以被扫描到从而被 build 到内核中



这个时候编译会报错，主要原因新旧版本有一些不兼容，比如变量名有改变等，需要手动进行修改，主要包括以下内容

该报错是因为 1.0 版本没有提供对应的头文件，该文件和动态内存分配无关，

```
mem/tlsf/los_memory.c:41:10: fatal error: 'los_trace_frame.h' file not found
#include "los_trace_frame.h"
      ^~~~~~
1 error generated.
/home/book/openharmony/kernel/liteos_a/tools/build/mk/module.mk:83:
recipe for target '/home/book/openharmony/kernel/liteos_a/out/imx6ull/obj/kernel/base/mem/tlsf/los_memory.o' failed
make[1]: *** [/home/book/openharmony/kernel/liteos_a/out/imx6ull/obj/kernel/base/mem/tlsf/los_memory.o] Error 1
make[1]: *** 正在等待未完成的任务....
```

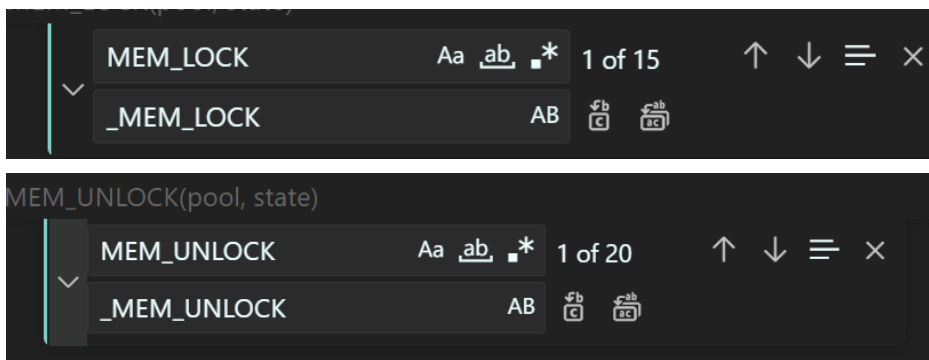
直接让这个这个条件编译不通过即可

```
#include "los_vm_phys.h"
#include "los_vm_boot.h"
#include "los_vm_filemap.h"
#include "los_task_pri.h"
#undef LOSCFG_KERNEL_TRACE
#ifdef LOSCFG_KERNEL_TRACE
#include "los_trace_frame.h"
#include "los_trace.h"
#endif
```

再次编译出现报错，因为两个自旋锁在 1.0 版本的另一个文件中被定义过了并且定义的参数个数和 1.1.0 版本不同，1.1.0 版本原本没有在另一个文件中定义只在 los\_memory.c 中定义，直接把这个文件中的这两个宏改个名字就行

```
make[1]: *** [kernel/liteos_a/out/imx6ull/obj/kernel/base/mem/tlsf/los_memory.o] Error 1
mem/tlsf/los_memory.c:168:9: error: 'MEM_LOCK' macro redefined [-Werror,-Wmacro-redefined]
#define MEM_LOCK(pool, state)      LOS_SpinLockSave(&(pool)->spinlock, &(state))
      ^
/home/book/openharmony/kernel/liteos_a/kernel/base/include/los_memory_pri.h:63:9: note: previous definition is here
#define MEM_LOCK(state)           LOS_SpinLockSave(&g_memSpin, &(state))
      ^
mem/tlsf/los_memory.c:169:9: error: 'MEM_UNLOCK' macro redefined [-Werror,-Wmacro-redefined]
#define MEM_UNLOCK(pool, state)    LOS_SpinUnlockRestore(&(pool)->spinlock, (state))
      ^
/home/book/openharmony/kernel/liteos_a/kernel/base/include/los_memory_pri.h:64:9: note: previous definition is here
#define MEM_UNLOCK(state)         LOS_SpinUnlockRestore(&g_memSpin, (state))
      ^
```

使用全局替换进行更改即可，在宏的最前面加一个下划线来区分即可



找不到宏的原因是在 1.1.0 版本中这个宏被定义在了另一个文件中，而在 1.0 的对应文件没有提供这个宏，因此只需要在程序开头手动添加这个宏即可，宏的内容可以在 1.1.0 版本对应文件里找到

```
make[1]: 进入目录"/home/book/openharmony/kernel/liteos_a/kernel/common"
make[1]: 对"all"无需做任何事。
make[1]: 离开目录"/home/book/openharmony/kernel/liteos_a/kernel/common"
make[1]: 进入目录"/home/book/openharmony/kernel/liteos_a/kernel/base"
mem/tlsf/los_memory.c:409:28: error: implicit declaration of function 'MEM_EXPAND_SIZE' is invalid in C99 [
    UINT32 expandDefault = MEM_EXPAND_SIZE(LOS_MemPoolSizeGet(pool));
                           ^
mem/tlsf/los_memory.c:1788:17: error: no member named 'totalUsedSize' in 'LOS_MEM_POOL_STATUS'; did you mea
    poolStatus->totalUsedSize += totalUsedSize;
                   ^~~~~~
    uwTotalUsedSize
```

```
#include "los_trace_frame.h"
#include "los_trace.h"
#endif
#define MEM_EXPAND_SIZE(poolSize)    (poolSize >> 3)
#ifdef __cplusplus
#ifdef __cplusplus
```

最后还有一个报错就是 poolStatus 结构体中成员的名字在两个版本有所不同，在旧版本中多了前缀 uw，下边第一个图是 1.1.0 版本，第二个图是 1.0 版本

```
typedef struct {
    UINT32 uwTotalUsedSize;
    UINT32 uwTotalFreeSize;
    UINT32 uwMaxFreeNodeSize;
    UINT32 uwUsedNodeNum;
    UINT32 uwFreeNodeNum;
#ifdef OS_MEM_WATERLINE
    UINT32 uwUsageWaterLine;
#endif
} LOS_MEM_POOL_STATUS;
```

```
/**
 * @ingroup los_memory
 * Memory pool extern information structure
 */
typedef struct {
    UINT32 uwTotalUsedSize;
    UINT32 uwTotalFreeSize;
    UINT32 uwMaxFreeNodeSize;
    UINT32 uwUsedNodeNum;
    UINT32 uwFreeNodeNum;
#ifdef OS_MEM_WATERLINE
    UINT32 uwUsageWaterLine;
#endif
} LOS_MEM_POOL_STATUS;
```

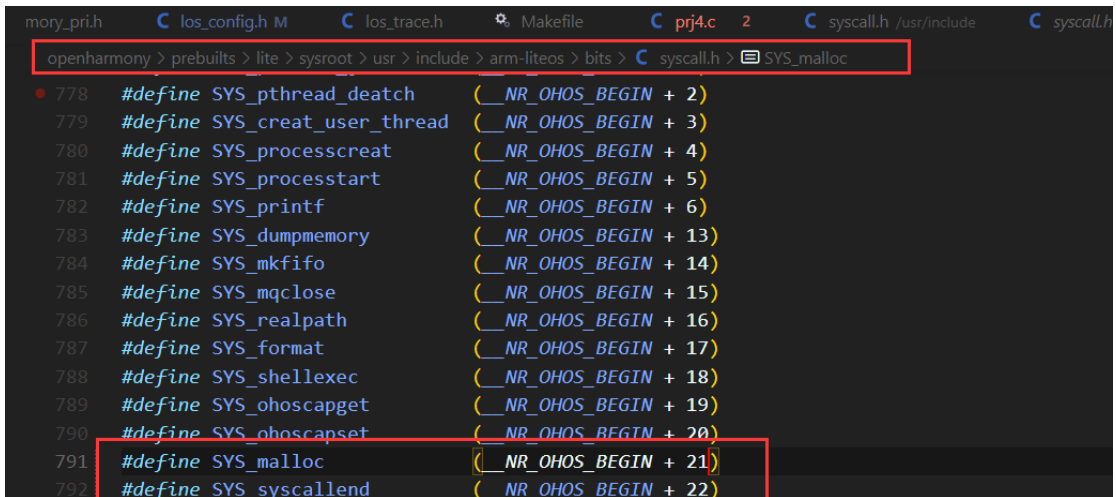
使用全局替换的方法对移植进来的文件逐个替换即可，这个时候再进行编译就不会出现报错了

```
/home/hza/llvm/bin/.../bin/llvm-objcopy -R .bss -O binary /home/book/openharmony/kernel/liteos_a/out/imx6ull/liteos_a/out/imx6ull/liteos.bin
/home/hza/llvm/bin/.../bin/llvm-objdump -t /home/book/openharmony/kernel/liteos_a/out/imx6ull/liteos |sort >/home/book/openharmony/kernel/liteos_a/out/imx6ull/liteos.sym.sorted
/home/hza/llvm/bin/.../bin/llvm-objdump -d /home/book/openharmony/kernel/liteos_a/out/imx6ull/liteos >/home/book/openharmony/kernel/liteos_a/out/imx6ull/liteos.asm
make[1]: 进入目录"/home/book/openharmony/kernel/liteos_a/apps"
make[2]: 进入目录"/home/book/openharmony/kernel/liteos_a/apps/shell"
make[2]: 离开目录"/home/book/openharmony/kernel/liteos_a/apps/shell"
make[2]: 进入目录"/home/book/openharmony/kernel/liteos_a/apps/init"
make[2]: 离开目录"/home/book/openharmony/kernel/liteos_a/apps/init"
make[1]: 离开目录"/home/book/openharmony/kernel/liteos_a/apps"
book@hza-virtual-machine:~/openharmony/kernel/liteos_a$
```

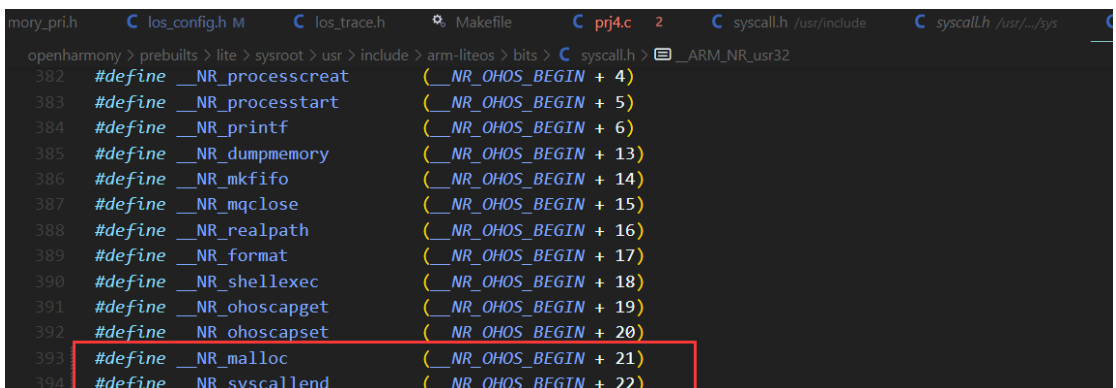
## 2、增加系统调用

增加的 TLSF 部分代码没法直接在应用程序中调用，因为应用程序的交叉编译、链接是通过静态库来完成的，虽然修改了内核代码但是静态库没有发生改变，需要重新编译静态库，但是鸿蒙 1.0 版本没提供对应的编译工具，在 1.1.0 中才提供工具，还需要经过修改才能使用，过程非常麻烦，为了方便直接使用系统调用来使用这部分的代码。

加系统调用和实验一相同 openharmony/prebuilts/lite/sysroot/usr/include/arm-liteos/bits/syscall.h 中定义了内核态和用户态使用的系统调用号，在里面加入新的 SYS\_malloc 和 \_\_NR\_malloc

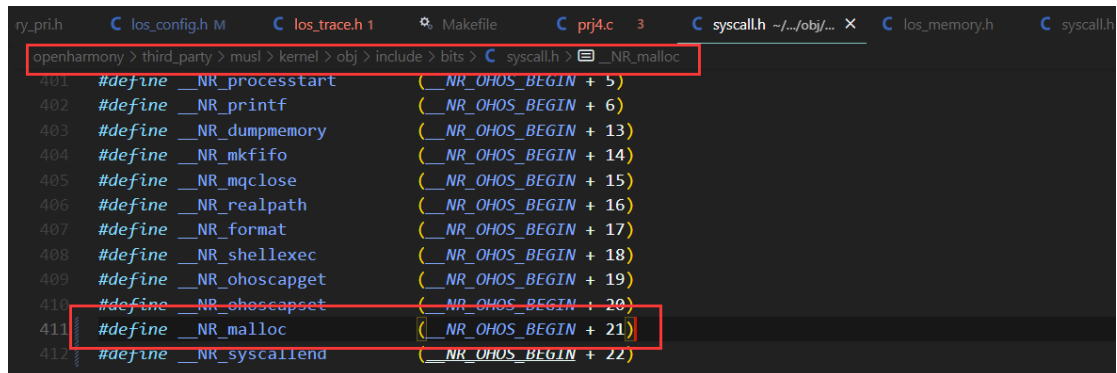


```
mory_pri.h  C los_config.h M  C los_trace.h  Makefile  C prj4.c  2  C syscall.h /usr/include  C syscall.h
openharmony > prebuilts > lite > sysroot > usr > include > arm-liteos > bits > C syscall.h > SYS_malloc
778 #define SYS_pthread_deatch    (__NR_OHOS_BEGIN + 2)
779 #define SYS_creat_user_thread  (__NR_OHOS_BEGIN + 3)
780 #define SYS_processcreat       (__NR_OHOS_BEGIN + 4)
781 #define SYS_processtart        (__NR_OHOS_BEGIN + 5)
782 #define SYS_printf             (__NR_OHOS_BEGIN + 6)
783 #define SYS_dumpmemory         (__NR_OHOS_BEGIN + 13)
784 #define SYS_mkfifo            (__NR_OHOS_BEGIN + 14)
785 #define SYS_mqclose            (__NR_OHOS_BEGIN + 15)
786 #define SYS_realpath           (__NR_OHOS_BEGIN + 16)
787 #define SYS_format             (__NR_OHOS_BEGIN + 17)
788 #define SYS_shellexec          (__NR_OHOS_BEGIN + 18)
789 #define SYS_ohoscapget         (__NR_OHOS_BEGIN + 19)
790 #define SYS_ohoscapset         (__NR_OHOS_BEGIN + 20)
791 #define SYS_malloc             (__NR_OHOS_BEGIN + 21)
792 #define SYS_syscallend         (__NR_OHOS_BEGIN + 22)
```



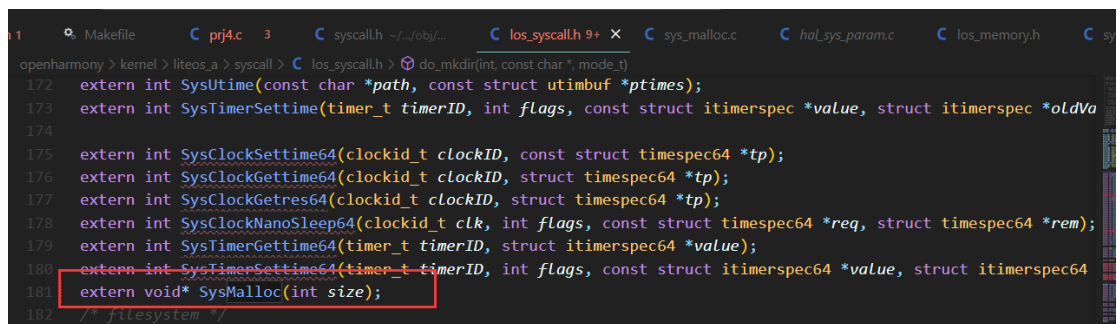
```
mory_pri.h  C los_config.h M  C los_trace.h  Makefile  C prj4.c  2  C syscall.h /usr/include  C syscall.h /usr/include/sys
openharmony > prebuilts > lite > sysroot > usr > include > arm-liteos > bits > C syscall.h > _ARM_NR_usr32
382 #define __NR_processcreat      (__NR_OHOS_BEGIN + 4)
383 #define __NR_processtart        (__NR_OHOS_BEGIN + 5)
384 #define __NR_printf             (__NR_OHOS_BEGIN + 6)
385 #define __NR_dumpmemory         (__NR_OHOS_BEGIN + 13)
386 #define __NR_mkfifo            (__NR_OHOS_BEGIN + 14)
387 #define __NR_mqclose            (__NR_OHOS_BEGIN + 15)
388 #define __NR_realpath           (__NR_OHOS_BEGIN + 16)
389 #define __NR_format             (__NR_OHOS_BEGIN + 17)
390 #define __NR_shellexec          (__NR_OHOS_BEGIN + 18)
391 #define __NR_ohoscapget         (__NR_OHOS_BEGIN + 19)
392 #define __NR_ohoscapset         (__NR_OHOS_BEGIN + 20)
393 #define __NR_malloc             (__NR_OHOS_BEGIN + 21)
394 #define __NR_syscallend         (__NR_OHOS_BEGIN + 22)
```

在 openharmony/third\_party/musl/kernel/obj/include/bits/syscall.h 中添加系统调用号



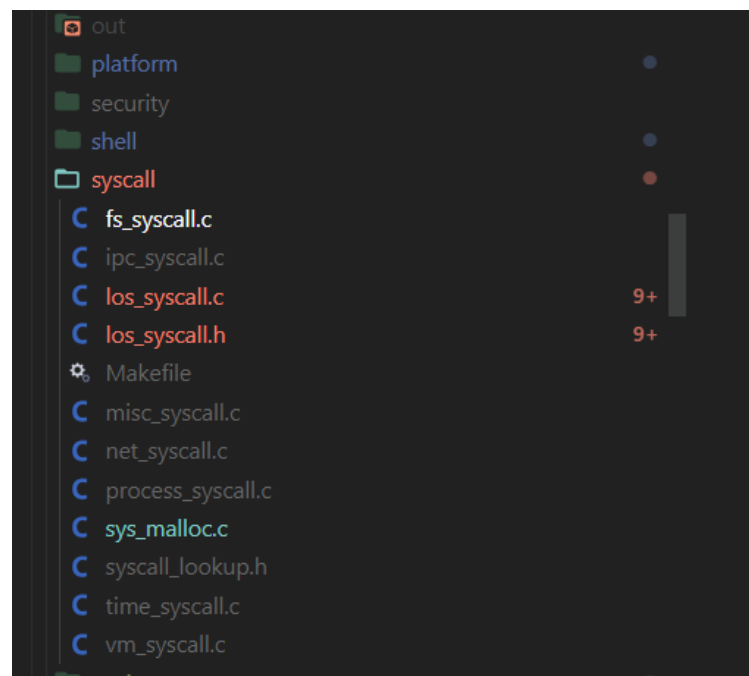
```
ry_pri.h  C  los_config.h M  C  los_trace.h 1  Makefile  C  prj4.c 3  C  syscall.h ~/.../obj/... x  C  los_memory.h  C  syscall.h
openharmony > third_party > musl > kernel > obj > include > bits > C  syscall.h > NR_malloc
401 #define __NR_processtart    (__NR_OHOS_BEGIN + 5)
402 #define __NR_printf         (__NR_OHOS_BEGIN + 6)
403 #define __NR_dumpmemory     (__NR_OHOS_BEGIN + 13)
404 #define __NR_mkfifo         (__NR_OHOS_BEGIN + 14)
405 #define __NR_mqclose        (__NR_OHOS_BEGIN + 15)
406 #define __NR_realpath       (__NR_OHOS_BEGIN + 16)
407 #define __NR_format         (__NR_OHOS_BEGIN + 17)
408 #define __NR_shellexec      (__NR_OHOS_BEGIN + 18)
409 #define __NR_ohoscapget     (__NR_OHOS_BEGIN + 19)
410 #define __NR_ohoscapet      (__NR_OHOS_BEGIN + 20)
411 #define __NR_malloc         (__NR_OHOS_BEGIN + 21)
412 #define __NR_syscallend     (__NR_OHOS_BEGIN + 22)
```

在 openharmony/kernel/liteos\_a/syscall/los\_syscall.h 中添加系统调用处理函数的声明



```
1  Makefile  C  prj4.c 3  C  syscall.h ~/.../obj/...  C  los_syscall.h 9+ x  C  sys_malloc.c  C  hal_sys_param.c  C  los_memory.h  C  sy
openharmony > kernel > liteos_a > syscall > C  los_syscall.h > do_mkdir(int, const char *, mode_t)
172 extern int SysUtime(const char *path, const struct utimbuf *ptimes);
173 extern int SysTimerSettime(timer_t timerID, int flags, const struct itimerspec *value, struct itimerspec *oldVa
174
175 extern int SysClockSettime64(clockid_t clockID, const struct timespec64 *tp);
176 extern int SysClockGettime64(clockid_t clockID, struct timespec64 *tp);
177 extern int SysClockGetres64(clockid_t clockID, struct timespec64 *tp);
178 extern int SysClockNanoSleep64(clockid_t clk, int flags, const struct timespec64 *req, struct timespec64 *rem);
179 extern int SysTimerGettime64(timer_t timerID, struct itimerspec64 *value);
180 extern int SysTimerSettime64(timer_t timerID, int flags, const struct itimerspec64 *value, struct itimerspec64
181 extern void* SysMalloc(int size);
182 /* filesystem */
```

在 syscall 目录下新建一个 sys\_malloc.c 用于存放函数的实现





```
h1  Makefile  C prj4.c 3  C syscall.h ~/.../obj/...  C los_syscall.h 9+  C sys_malloc.c  C hal_sys_param.c  C lo
openharmony > kernel > liteos_a > syscall > C sys_malloc.c > SysMalloc(int)
1  #include "los_printf.h"
2  #include "los_syscall.h"
3  #include "los_memory.h"
4  void* SysMalloc(int size)
5  {
6      return (void*) LOS_MemAlloc((void *)OS_SYS_MEM_ADDR,size);
7  }
```

之后建立处理函数和系统调用号的映射关系

```
M  C los_trace.h 1  Makefile  C prj4.c 1  C syscall.h ~/.../arm-liteos/...  C syscall.h ~/.../obj/...  C los_syscall.h 9+  C syscall_loo
openharmony > kernel > liteos_a > syscall > C syscall_lookup.h
233  SYSCALL_HAND_DEF(409, SysTimerSetTime64, int, ARG_NUM_4)
234
235  /* LiteOS customized syscalls, not compatible with ARM EABI */
236  SYSCALL_HAND_DEF(__NR_pthread_set_detach, SysUserThreadSetDeatch, int, ARG_NUM_1)
237  SYSCALL_HAND_DEF(__NR_pthread_join, SysThreadJoin, int, ARG_NUM_1)
238  SYSCALL_HAND_DEF(__NR_pthread_deatch, SysUserThreadDetach, int, ARG_NUM_1)
239  SYSCALL_HAND_DEF(__NR_create_user_thread, SysCreateUserThread, unsigned int, ARG_NUM_3)
240  SYSCALL_HAND_DEF(__NR_malloc, SysMalloc, void*, ARG_NUM_1)
```

在 tlfs 的内存分配函数这里加一行输出，用于判断是否执行了该部分代码

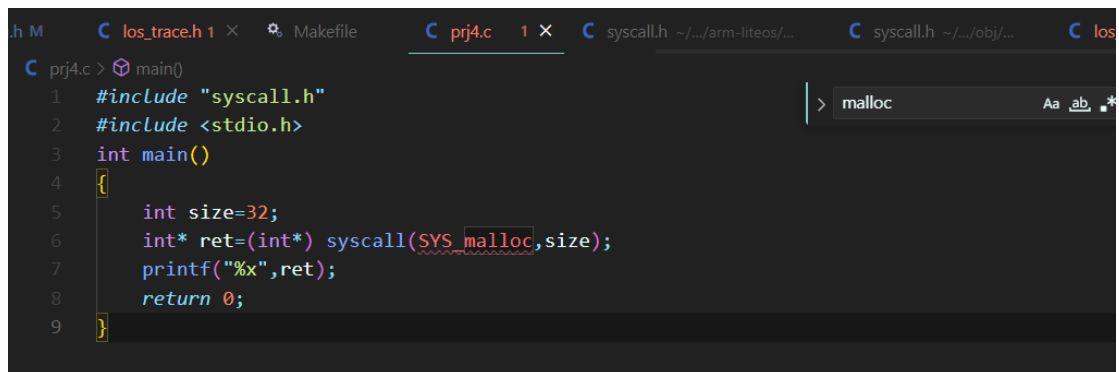
```
C malloc.c  C los_memory.c ~/.../tlfs 4, U X  liteos.bin  C los_memory.h 4  C los_memory.c ~/.../bestit  C los_memory_pri.h
openharmony > kernel > liteos_a > kernel > base > mem > tlfs > C los_memory.c > LOS_MemAlloc(VOID *, UINT32)
951  return OsMemCreateUsedNode((VOID *)allocNode);
952  }
953  //pool 动态内存池的起始位置
954  VOID *LOS_MemAlloc(VOID *pool, UINT32 size)
955  {
956  #ifdef LOSCFG_KERNEL_TRACE
957      UINT64 start = HalClockGetCycles();
958  #endif
959  PRINTK("#####");
960  if ((pool == NULL) || (size == 0)) {
961      return (size > 0) ? OsVmBootMemAlloc(size) : NULL;
962  }
```

编译内核之后烧写，可以看到输出了很多井号，证明确实执行了该部分代码，不过 shell 没有起来，后来经过判断是因为这行打印代码的问题

```
3. 开发板
#####cpu 0 entering scheduler
proc fs init ...
#####Mount procs finished.
#####mem dev init ...
#####spinor_init init ...
src/common/spinor.c spinor_init 155
#####register partition ...
#####mount /dev/spinorblk0 / ...
#####
#####DeviceManagerStart start ...
#####[ERR][HDF:E/hcs_blob_if]CheckHcsBlobLength: the blobLength: 1276
G_TAG]cmp HDF_PLATFORM_I2C HDF_PLATFORM_I2C
#####
G_TAG]cmp HDF_PLATFORM_HELLO HDF_PLATFORM_I2C
[ERR][HDF:E/HDF_LOG_TAG]cmp HDF_PLATFORM_HELLO HDF_TOUCHSCREEN
[ERR][HDF:E/HDF_LOG_TAG]cmp HDF_PLATFORM_HELLO HDF_PLATFORM_HELLO
#####
##[ERR][HDF:E/HDF_LOG_TAG]cmp HDF_TOUCHSCREEN HDF_PLATFORM_I2C
[ERR][HDF:E/HDF LOG TAG]cmp HDF TOUCHSCREEN HDF TOUCHSCREEN
```

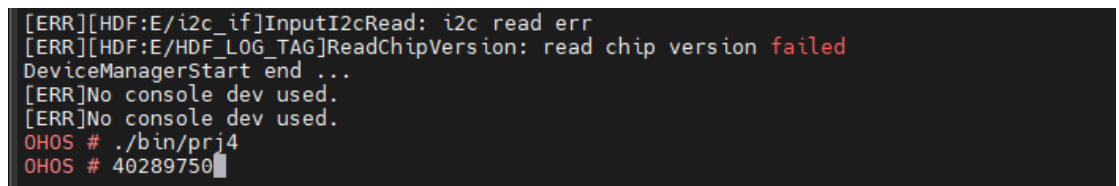


去掉打印代码，编写一个用户态程序用于判断是否可以正确被调用



```
prj4.c > main()
1  #include "syscall.h"
2  #include <stdio.h>
3  int main()
4  {
5      int size=32;
6      int* ret=(int*) syscall(SYS_malloc,size);
7      printf("%x",ret);
8      return 0;
9  }
```

交叉编译再运行应用程序，可以看到输出了一个地址，初步判断应该是移植成功了



```
[ERR][HDF:E/i2c_if]InputI2cRead: i2c read err
[ERR][HDF:E/HDF_LOG_TAG]ReadChipVersion: read chip version failed
DeviceManagerStart end ...
[ERR]No console dev used.
[ERR]No console dev used.
OHOS # ./bin/prj4
OHOS # 40289750
```

### 3、Good-fit 代码分析

接下来分析下 LiteOS 该部分代码为什么本身就是 Good-fit 算法，在 LOS\_MemAlloc 完成参数检查之后将会调用 OsMemAlloc 进行内存分配

```
//pool 动态内存池的起始位置
VOID *LOS_MemAlloc(VOID *pool, UINT32 size)
{
#ifdef LOSCFG_KERNEL_TRACE
    UINT64 start = HalClockGetCycles();
#endif
    if ((pool == NULL) || (size == 0)) {
        return (size > 0) ? OsVmBootMemAlloc(size) : NULL;
    }

    if (size < OS_MEM_MIN_ALLOC_SIZE) {
        size = OS_MEM_MIN_ALLOC_SIZE;
    }

    struct OsMemPoolHead *poolHead = (struct OsMemPoolHead *)pool;
    VOID *ptr = NULL;
    UINT32 intSave;

    do {
        if (OS_MEM_NODE_GET_USED_FLAG(size) || OS_MEM_NODE_GET_ALIGNED_FLAG(size)) {
            break;
        }
        _MEM_LOCK(poolHead, intSave);
        ptr = OsMemAlloc(poolHead, size, intSave);
        _MEM_UNLOCK(poolHead, intSave);
    } while (0);
}
```

在 OsMemAlloc 中会先对申请的内存大小进行内存对齐操作，之后调用 OsMemFreeNodeGet 去从空闲内存链表中获取一个满足申请大小的空闲内存块

```
STATIC INLINE VOID *OsMemAlloc(struct OsMemPoolHead *pool, UINT32 size, UINT32 intSave)
{
    struct OsMemNodeHead *allocNode = NULL;

#ifdef LOSCFG_BASE_MEM_NODE_INTEGRITY_CHECK
    if (OsMemAllocCheck(pool, intSave) == LOS_NOK) {
        return NULL;
    }
#endif
    //内存对齐 申请的大小+头节点
    UINT32 allocSize = OS_MEM_ALIGN(size + OS_MEM_NODE_HEAD_SIZE, OS_MEM_ALIGN_SIZE);
    if (allocSize == 0) {
        return NULL;
    }

#ifdef OS_MEM_EXPAND_ENABLE
    retry:
#endif
    //从空闲内存链表中获取一个满足申请大小的空闲内存块
    allocNode = OsMemFreeNodeGet(pool, allocSize);
    if (allocNode == NULL) {
#ifdef OS_MEM_EXPAND_ENABLE
        if (pool->info.attr & OS_MEM_POOL_EXPAND_ENABLE) {
            INT32 ret = OsMemPoolExpand(pool, allocSize, intSave);
            if (ret == 0) {
                goto retry;
            }
        }
#endif
    }
}

#endif
```

OsMemFreeNodeGet 中会调用 OsMemFindNextSuitableBlock 找到大小合适的内存块，之后执行删除操作将这个内存块从链表中删除

```
STATIC INLINE struct OsMemNodeHead *OsMemFreeNodeGet(VOID *pool, UINT32 size)
{
    struct OsMemPoolHead *poolHead = (struct OsMemPoolHead *)pool;
    UINT32 index;
    struct OsMemFreeNodeHead *firstNode = OsMemFindNextSuitableBlock(pool, size, &index);
    if (firstNode == NULL) {
        return NULL;
    }
    OsMemListDelete(poolHead, index, firstNode);

    return &firstNode->header;
}
```

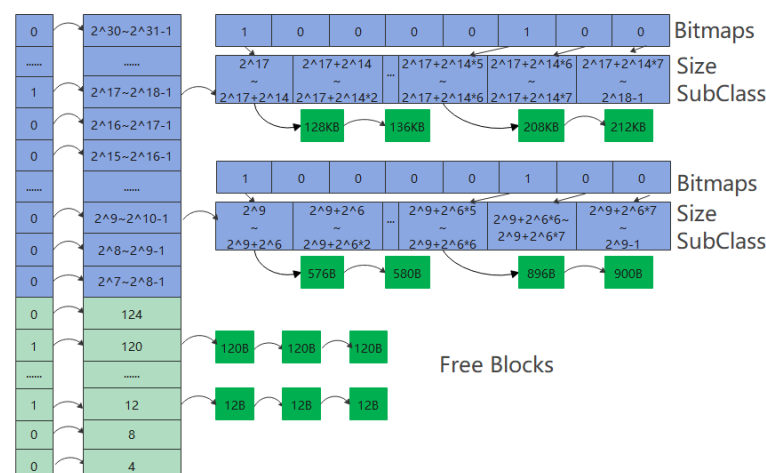
之后就是 Good-Fit 算法的核心部分，首先会对传入的参数取第一级索引 fl

```
STATIC INLINE struct OsMemFreeNodeHead *OsMemFindNextSuitableBlock(VOID *pool, UINT32 size, UINT32 *outIndex)
{
    struct OsMemPoolHead *poolHead = (struct OsMemPoolHead *)pool;
    UINT32 fl = OsMemFlGet(size);
    UINT32 sl;
    UINT32 index, tmp;
    UINT32 curIndex = OS_MEM_FREE_LIST_COUNT;
    UINT32 mask;
    do {
        if (size < OS_MEM_SMALL_BUCKET_MAX_SIZE) {
            index = fl;
        } else {
            sl = OsMemSlGet(size, fl);
            curIndex = ((fl - OS_MEM_LARGE_START_BUCKET) << OS_MEM_SLI) + sl + OS_MEM_SMALL_BUCKET_COUNT;
            index = curIndex + 1;
        }
        //curIndex算的就是size对应位图中的位的下标, index是后一个
        //找比自己大一点的size (局限于自己所在的字内), 也就是good-fit
        tmp = OsMemNotEmptyIndexGet(poolHead, index);
        if (tmp != OS_MEM_FREE_LIST_COUNT) {
            index = tmp;
            goto DONE; //找到就结束了
        }
        //如果在自己字内往上找没找着, 就跳到后面的字看看
        for (index = LOS_Align(index + 1, 32); index < OS_MEM_FREE_LIST_COUNT; index += 32) {
            mask = poolHead->freeListBitmap[index >> 5]; /* 5: Divide by 32 to calculate the index of the first word in the bucket */
            if (mask != 0) {
                index = OsMemFFS(mask) + index;
                goto DONE;
            }
        }
    } while (1);
}
```

第一级索引值 fl 的定义根据 size 的大小决定，如果 size 大小小于 127，分配的内存应该位于小桶中，小桶中每一位递增的大小都是 4 个字节，也就是对于 [4,127]分为 31 个小区间，每个区间对应内存块大小为 4 的倍数，这部分不需要二级位图，所以  $fl = size / 4 - 1$ （减 1 是因为小桶的下标索引从 0 开始）；如果对于 size 比 127 大，则分配到大桶中，大桶中每一位之间的增长不再是等差数列，fl 将用于计算后续的索引值

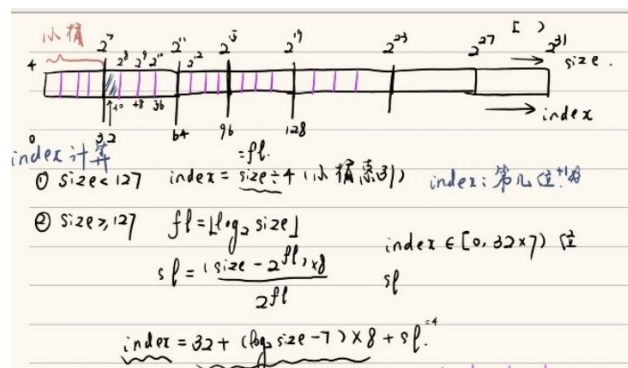
```
/* Get the first Level: f = Log2(size). */
STATIC INLINE UINT32 OsMemFlGet(UINT32 size)
{
    if (size < OS_MEM_SMALL_BUCKET_MAX_SIZE) {
        return ((size >> 2) - 1); /* 2: The small bucket setup is 4. */
    }
    return OsMemLog2(size);
}
```

Bitmaps Size Class



这里位图总共有  $31+24*8$  位, 对应小桶有 **31** 个位, 而大桶一级索引有 **24** 个, 假设一级索引是  $fl$ , 则对应的一级索引区间内存块大小是  $[2^{fl}, 2^{fl+1})$ , 之后再会对一级索引内部进行细分成 **8** 份, 每一个用一个二级索引去表示, 大桶区域的一级索引不会存储, 只会直接存二级索引, 所以位图是  $31+24*8$  位, 最后使用 7 个 `UINT32` 进行存储, 因此可以看到在代码中有一个数组是 `freeListBitmap[7]`, 这些数组的地址连续, 即可以把一级索引位和二级索引为看成全部线性排列在一起, 之后使用下标 `curIndex` 和 `Index` 去检索整个该线性排列

接下来为了方便理解代码先给出 `index` 的定义, `index` 的定义是 `size` 对应内存块位图中位的下标值的下一位。如下图所示, 这里大的格子表示一个字, 里面被紫色的线再分为 4 个字节, 之后每个字节里面又分成 8 个位 (图里没有画出), 前面第一个字的前 31 位被用作小桶的位图 (图没有画准, 并不是第一个字全给了小桶), 后面 6 个是大桶。代码中计算的 `curIndex` 就是下标值, 比如 `size` 是 129, 则 `curIndex` 是 31, 也就是 129 大小的块被挂在下标是 31 对应的链表下, `sl` 计算的是字节内部的位偏移, 因为一个字节表示范围是  $[2^{fl}, 2^{fl+1})$ , 之后将这个范围分成 8 份, 所以每一份大小是  $(2^{fl+1} - 2^{fl})/8$ , `size` 超出边界值大小是  $size - 2^{fl}$ , 两者相除就是字节内部偏移量, 也就是二级索引值。而 `index` 是 `curIndex+1`, 也就是挂了对应内存块的下一个位, 所以查看 `index` 及 `index` 以上对应的位就是 Goodfit 算法



该部分代码通过 `index` 找到对应位图的值用于判断是否有空闲内存块, 这里 Goodfit 不只看下一位, 而是对这个字内比 `size` 大的对应位都进行判断

```

STATIC INLINE UINT32 OsMemNotEmptyIndexGet(struct OsMemPoolHead *poolHead, UINT32 index)
{
    // 根据空闲内存链表索引获取位图字
    UINT32 mask = poolHead->freeListBitmap[index >> 5]; /* 5: Divide by 32 to calculate the index of the bit map word */

    mask &= ~(1 << ((index & OS_MEM_BITMAP_MASK) - 1));
    // 其中index & OS_MEM_BITMAP_MASK对索引只取低5位后, 也就是在对应字里面的位偏移
    // 表达式~((1 << ((index & OS_MEM_BITMAP_MASK) - 1)) - 1)则用于生成掩码
    // 比如size对应位在字内偏移位数是4, 则生成掩码是FFF0,
    // 也就是考虑第4位开始这个字里是否有1, 相当于往后寻找了, 实现goodfit
    // 同理如果偏移是x (从0开始算), 则掩码是1111...1000...0 (总共32位, 低位有x个0)
    // 又因为index是对应位下标 (curIndex) + 1, 所以就是实现了goodfit

    if (mask != 0) {
        // 如果这个字内比指定size大的内存块存在
        index = OsMemFFS(mask) + (index & ~OS_MEM_BITMAP_MASK); // 获得这个内存块对应的位置index
        return index;
    }

    return OS_MEM_FREE_LIST_COUNT; // 否则偏移变为最大值, 这里是 (31+24*8, index从0开始, 所以这个位不存在)
}

```

之后如果这个字内比自己大的部分找不着就会去后面的字看看，如果后面的字也都没有，才会遍历真正对应位 `curIndex` 的链表取寻找内存块。

```
    }
    //curIndex算的就是size对应位图中的位的下标，index是后一个
    //找比自己大一点的size (局限于自己所在的字内)，也就是good-fit
    tmp = OsMemNotEmptyIndexGet(poolHead, index);
    if (tmp != OS_MEM_FREE_LIST_COUNT) {
        index = tmp;
        goto DONE; //找到就结束了
    }
    //如果在自己字内往上找没找着，就跳到后面的字看看
    for (index = LOS_Align(index + 1, 32); index < OS_MEM_FREE_LIST_COUNT; index += 32) {
        mask = poolHead->freeListBitmap[index >> 5]; /* 5: Divide by 32 to calculate the index */
        if (mask != 0) {
            index = OsMemFFS(mask) + index;
            goto DONE;
        }
    }
} while (0);

if (curIndex == OS_MEM_FREE_LIST_COUNT) {
    return NULL;
}
//如果往上找也没空了，那就只能顺着自己位的链表看看了
*outIndex = curIndex;
return OsMemFindCurSuitableBlock(poolHead, curIndex, size);
DONE:
*outIndex = index;
return poolHead->freeList[index];
}
```

所以内核本身就已经实现了 Good-fit，接下来测试下这个代码

## 4、测试 TLSF 算法

根据算法原理，尝试去新建一个内存池，然后在 `index` 为 47 和 48 的位置上插入几个节点，之后分配一个内存查看具体分配到的节点大小是多少的

根据代码在 `los_memory.c`(其实最好将测试代码放在系统调用的实现函数中，但是测试代码中会用到内存有关的一些结构体，直接在内核文件中创建函数比较方便些)先创建一个内存池，之后在其中加入若干个节点，这些节点对应的 `index` 为 47 或 48，最后分配一个 535 大小的节点进行测试

```
2019 #define TEST_POOL_SIZE 0x1000//2^12字节内存空间
2020 UINT32 test(void) {
2021     UINT8 g_testPool[TEST_POOL_SIZE + 0x1000];
2022     PRINTK( "\nTesting memory functions. . . \n" );
2023     UINT32 ret = LOS_MemInit(g_testPool,TEST_POOL_SIZE);
2024     if (LOS_OK == ret) {
2025         PRINTK("Memory pool initialization is successful ! \n" );
2026     }else {
2027         PRINTK("Mempool initialization failed ! \n");
2028         return LOS_NOK;
2029     }
2030 }
```

```

int node_size[9]={578,513,535,515,556,545,580,516,589};
for(int i=0;i<9;++i)
{
    struct OsMemNodeHead testnode;
    testnode.sizeAndFlag = node_size[i];
    testnode.ptr.prev=NULL;
    testnode.magic = OS_MEM_NODE_MAGIC;
    OsMemFreeNodeAdd(g_testPool,(struct OsMemFreeNodeHead *)&testnode);
}

PRINTK("The index of the malloc size: %d\n",OsMemFreeListIndexGet(535));
void *block = LOS_MemAlloc(g_testPool,535);
(void)block;
return LOS_OK;
}

```

在头文件中加入函数的声明，这里要注意文件路径，在海思目录下有个同名的头文件，不可以加在那边

```

C sys_malloc.c U C los_memory.h ~/.../Huawei_LiteOS/... 4 C los_memory.h ~/.../liteos_a/... X C los_
openhmony > kernel > liteos_a > kernel > include > C los_memory.h > test(void)
830 #ifdef __cplusplus
831 #if __cplusplus
832 }
833 #endif /* __cplusplus */
834 #endif /* __cplusplus */
835
836 #endif /* _LOS_MEMORY_H */
837 extern UINT32 test(void);

```

为了使现象更加明显，在 OsMemFindNextSuitableBlock 函数中输出对应的日志信息，这里内存分配的时候会加上头部大小和进行内存对齐，所以我们虽然申请了 535 但实际分配的并不是 535

```

STATIC INLINE struct OsMemFreeNodeHead *OsMemFindNextSuitableBlock(VOID *pool, UINT32 size, UINT32 *outIndex)
{
    struct OsMemPoolHead *poolHead = (struct OsMemPoolHead *)pool;
    UINT32 fl = OsMemFlGet(size);
    UINT32 sl;
    UINT32 index, tmp;
    UINT32 curIndex = OS_MEM_FREE_LIST_COUNT;
    UINT32 mask;
    if(size==OS_MEM_ALIGN(535 + OS_MEM_NODE_HEAD_SIZE, OS_MEM_ALIGN_SIZE))
    {
        PRINTK("The size actually allocated:%d\n",size);
    }
}
do {

```

```

617     *outIndex = curIndex;
618     return OsMemFindCurSuitableBlock(poolHead, curIndex, size);
619 DONE:
620     *outIndex = index;
621     if(size==OS_MEM_ALIGN(535 + OS_MEM_NODE_HEAD_SIZE, OS_MEM_ALIGN_SIZE))
622     {
623         PRINTK("The actually allocate node size is %d\n",poolHead->freeList[index]->header.sizeAndFlag);
624     }
625     return poolHead->freeList[index];
626 }

```

## 四、实验结果

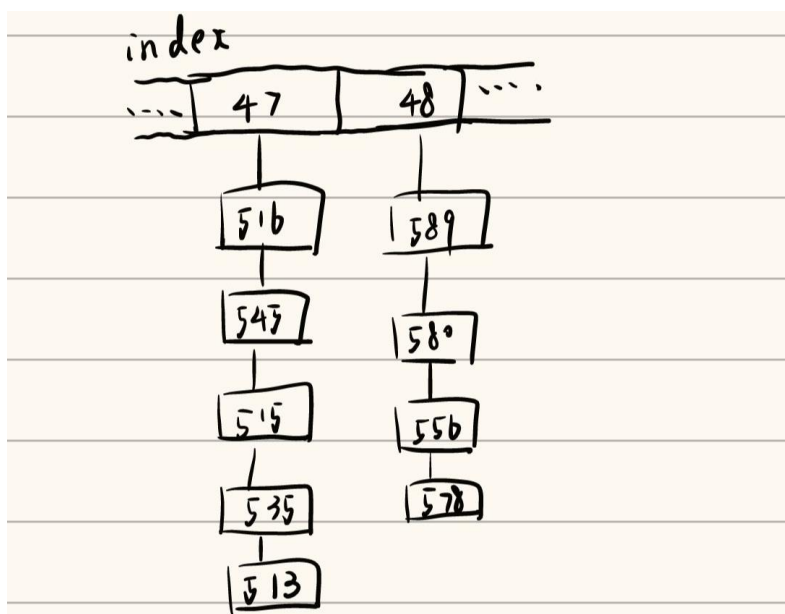
```

[ERR]No console dev used.
[ERR]No console dev used.
OHOS # ./bin/prj4
OHOS #
Testing memory functions. . .
Memory pool initialization is successful !
The index of the malloc size: 47
The size actually allocated:548
The actually allocate node size is 589

```

## 五、实验分析

一开始代码在内存池中插入了 9 个节点，根据内核的代码可以知道插入的时候使用头插法，再 index 和节点插入顺序可以知道插入后的内存池链表如下图所示



在测试函数中申请分配了 535 字节的内存，加上头部的大小和内存对齐之后分配的大小为 548 字节，而申请这 548 字节的内存所拿到的节点大小是 589 的节点，因此可以见 LiteOs 使用了 GoodFit 的内存分配算法从而快速拿到需要的内存



## 六、实验总结

本次实验先进行了移植操作，之后通过系统调用的方式对 `tlsf` 算法进行测试，通过内核代码的角度对 C 语言中的 `malloc` 函数有了进一步认知，从另一个角度看到了 C 语言代码和操作系统的息息相关

## 七、参考资料

[1] <http://t.zoukankan.com/huaweiyun-p-14926119.html>

## 八、附录

```
1. #define TEST_POOL_SIZE 0x1000 //2^12 字节内存空间
2. UINT32 test(void) {
3.     UINT8 g_testPool[TEST_POOL_SIZE + 0x1000];
4.     PRINTK( "\nTesting memory functions. . . \n" );
5.     UINT32 ret = LOS_MemInit(g_testPool, TEST_POOL_SIZE);
6.     if (LOS_OK == ret) {
7.         PRINTK("Memory pool initialization is successful ! \n" )
8.         ;
9.     }else {
10.        PRINTK("Mempool initialization failed ! \n");
11.        return LOS_NOK;
12.    }
13.    int node_size[9]={578,513,535,515,556,545,580,516,589};
14.    for(int i=0;i<9;++i)
15.    {
16.        struct OsMemNodeHead testnode;
17.        testnode.sizeAndFlag = node_size[i];
18.        testnode.ptr.prev=NULL;
19.        testnode.magic = OS_MEM_NODE_MAGIC;
20.        OsMemFreeNodeAdd(g_testPool, (struct OsMemFreeNodeHead *)(&testnode));
21.    }
22.
23.    PRINTK("The index of the malloc size: %d\n", OsMemFreeListIndexGet(535));
24.    void *block = LOS_MemAlloc(g_testPool, 535);
25.    (VOID)block;
26.    return LOS_OK;
27. }
```