内存管理实验

计算机的内存在不断增加，但是依然难以满足软件发展的需求。内存管理的好坏会直接影响操作系统的整体性能。本实验将讨论内存管理的相关技术，读者将在本实验中编写内存管理的程序，了解实现内存管理的方法。

实验目的：

通过本实验，读者应达到如下要求：

1. 理解操作系统原理中的内存管理机制。
2. 掌握理论课上基础的内存分配算法。
3. 学习并理解内存池、伙伴系统和slab分配器。
4. 探索扩展内存管理系统以支持更多的应用场景。

实验准备：

1. 实验环境：Linux操作系统
2. 熟悉C语言编程：链表、指针、地址等知识。
3. 了解操作系统内存管理的基本概念，理解操作系统内存分配的基础原理。
4. 理解操作系统内存管理中内存池、伙伴系统和slab分配器的基本原理。
5. 阅读并理解所给的代码用例和测试框架。

实验基本知识和原理：

本小节交代内存管理的基础知识和拓展实验采用的原理

1.基础知识

内存分配算法是操作系统中用于管理内存资源的重要机制，以下是首次适应、循环首次适应、最佳适应和最坏适应四种常见的内存分配策略的详细介绍：

1. 首次适应（First Fit）：首次适应算法在内存池中从头开始搜索，找到第一个足够大的空闲内存块，并从该块中分配所需的内存。如果该块比请求的内存大，剩余的部分可以保留或分割成更小的块供以后使用。首次适应的优点是分配速度快，但可能导致内存碎片的产生。
2. 循环首次适应（Next Fit）: 循环首次适应是首次适应的变种，它在上次搜索结束的地方继续搜索，形成一个循环。这种方法可以避免内存池的前端出现大量碎片。与首次适应类似，它也快速且简单，但循环的特性使得内存使用更加均匀。
3. 最佳适应（Best Fit）: 最佳适应算法在分配内存时，会搜索整个内存池，找到能够满足请求大小且剩余空间最小的空闲内存块。这种方式试图最小化内存浪费，但可能会导致较大的内存块被分割成小块，从而产生外部碎片。最佳适应算法通常分配速度较慢，因为它需要遍历整个空闲内存列表来找到最合适的块。
4. 最坏适应（Worst Fit）: 最坏适应算法与最佳适应相反，它会选择最大的空闲内存块进行分配，即使这个块比请求的内存大得多。这种方法的优点是可以减少内存碎片，因为它倾向于使用最大的空闲块，从而留下较小的空闲块。然而，这可能导致内存利用率降低，因为较大的块可能长时间不被使用。

总结 : 每种内存分配算法都有其优缺点，适用于不同的场景。例如，首次适应和循环首次适应由于其简单性和快速性，在许多系统中被广泛使用。而最佳适应和最坏适应则在需要考虑内存利用率和减少碎片的场景下更为合适。在实际应用中，操作系统可能会根据具体需求选择或组合不同的内存分配策略。

2.扩展实验原理

Linux系统采用slab等内存管理方式，本实验通过c语言的简单模拟，对一些原理做出阐释，同时引导学生思考存储方式的优缺点。

1. 内存池（Memory Pool）：预先分配一块内存，并将其划分为多个固定大小的块，用于高效的内存分配和回收。
2. 伙伴系统（Buddy System）：一种内存分配算法，将内存分为多个大小为2的幂的块，用于快速分配和回收。
3. Slab分配器：一种用于分配相同大小对象的内存分配器，通过缓存来减少内存分配的开销。
4. 虚拟存储：允许系统通过在磁盘上交换数据来扩展可用的内存空间。

实验说明：

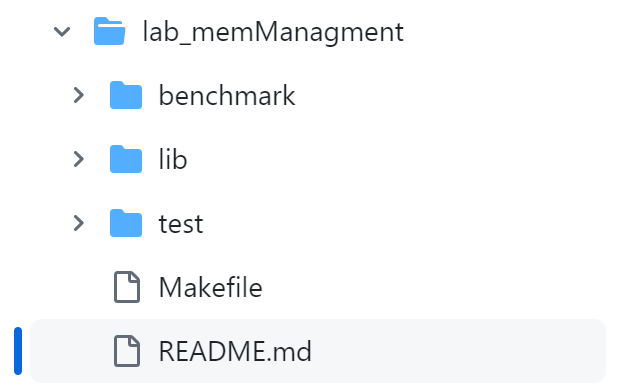
本小节主要对实验内容做介绍，首先是对测试框架的介绍，然后是使用方式的的讲解，最后是测试结果的展示。

1. 测试框架

这一部分会以电子资源的形式提供，此处附上源链接，如果链接失效请联系xxx(待议)

链接：<https://github.com/openeuler-riscv/minimal_uniproton>

目录结构如图所示：



图表 1 测试框架的结构

其中benchmark从其他仓库克隆而来，详细的介绍在README.md中，在此不做过多赘述。

1. 框架的使用方式：

环境要求：

操作系统：Linux

使用工具：GNU Binary Utilities, GNU Make

内置的内存分配算法

First Fit：首次适应算法

Best Fit：最佳适应算法

Worst Fit：最坏适应算法

Memory Pool：内存池

Buddy Allocator：伙伴分配器

easy small memory allocator (like slab)

框架的使用流程

我们可以通过测试框架，验证各个算法的正确性和测试各自算法的性能



图表 2 测试框架的使用方式

我们可以通过testcase来测试算法的正确性，测试样例中包含一些极端的情况，如极大极小值等。Benchmark可以帮助我们测试算法的性能，可以用来比较不同的算法。

接入自己算法的方式

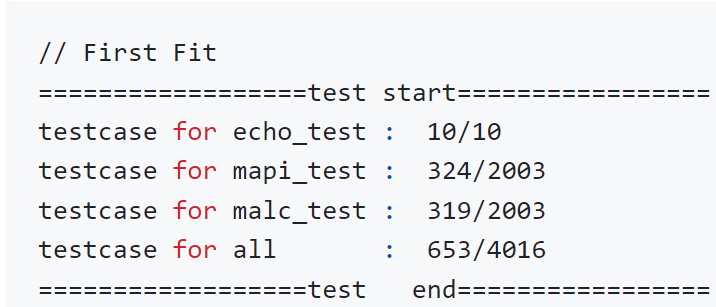
原始框架下，支持的算法如图所示：



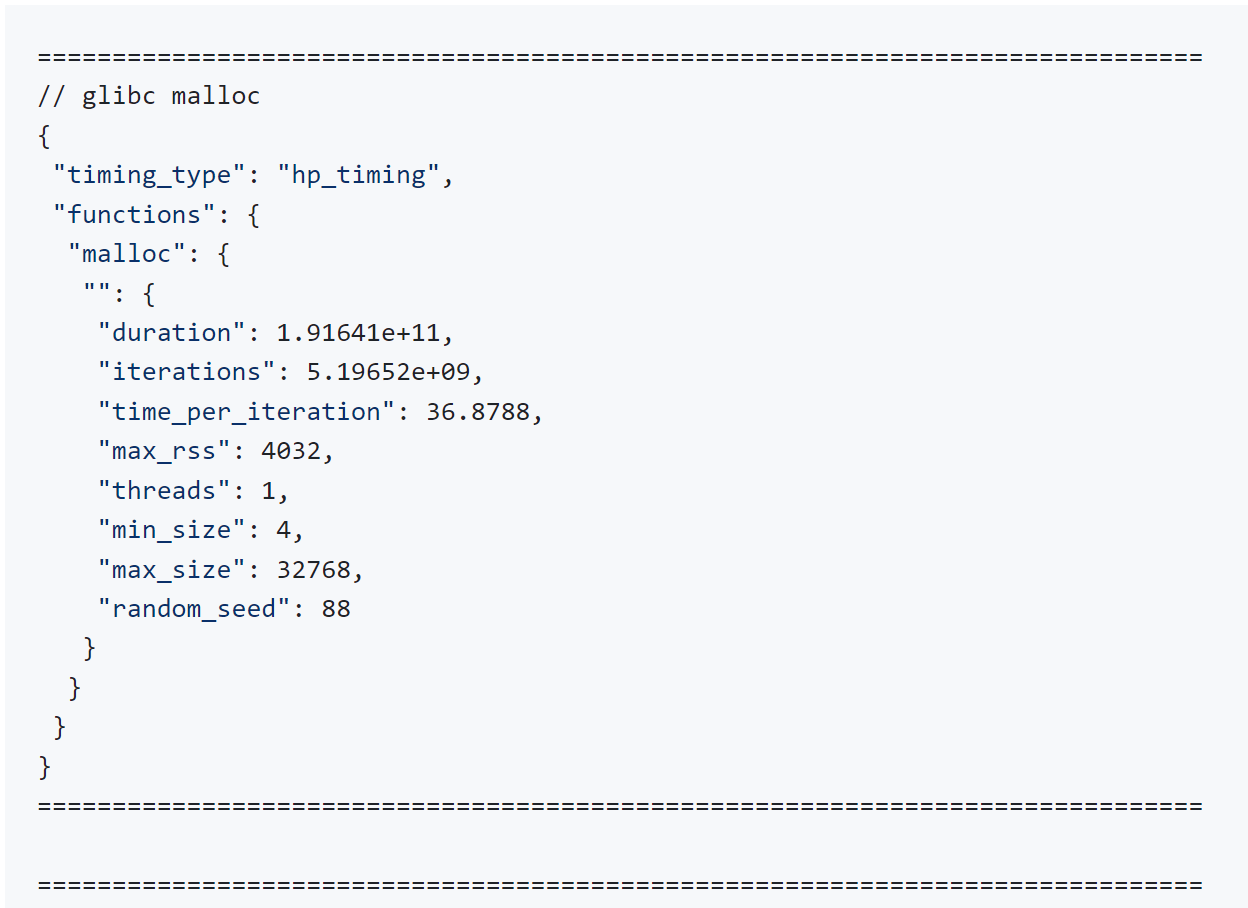
怎样编写测试自己的分配回收函数

1. 找到文件 **lib/include/memory\_lab.h** 文件，按照格式添加自己的内存分配的算法的宏，如 #define SELF\_ALLOCATOR 10
2. 在 **lib/include/memory\_lab.h** 文件中 根据 VOID\_ALLOCATOR 的内容，实现四个要求函数，其他的分配算法采取的是把 port user api 在/lib/include 目录下新建 .h 文件来存放，在 **lib/include/memory\_lab.h** 文件中包含对应头文件，实现对应要求的四个函数
3. 在 **lib/src/** 下创建新的目录用来存放自己的源码，然后实现对应的函数即可
4. 测试的结果

这个是算法正确性测试的结果，第一项测试完全通过代表代码的基础逻辑没有问题，但是存在着边缘条件的处理等问题。



这是算法的性能测试结果。基于各项属性对比的结果，可以对算法进行调整。



测试样例分析介绍和性能指标介绍在github仓库有，请同学们仔细寻找，认真阅读。

实验内容：

本小节主要讲述应该怎样完成实验，实验的报告怎么写，实验的评分标准是什么

1. 实验过程
2. 算法实现：基于给出的算法和测试框架，写出通过样例比基础算法更好的分配释放函数。
3. 代码分析：分析提供的文件中的代码，理解其工作原理。
4. 扩展思考：了解benchmark，分析算法的性能，在通过样例的算法基础上，提出算法性能改进的措施。
5. 评分总则

评分将从基础实验和进阶实验两部分进行，其中进阶实验是大作业的先导知识，完成进阶实验有利于大作业的编写。所以我们的评分建议是，在基础实验完成的前提下，让学生获得课程及格的成绩；在进阶实验完成的过程中，让学生逐渐获得从良好到优秀的成绩。

1. 基础实验评分：

1 编写完首次适应，最佳，最坏适应算法，要求优于基础算法，包括但不限于通过的样例比基础多，性能指标更好等。

2 对内存池、slab分配器、buddy伙伴系统的代码部分进行分析，在实验报告中提出自己的思考。

1. 进阶实验评分：
2. 实现比现有内存池更好的内存池分配和回收，通过测试框架进行验证。
3. 实现比现有slab分配器更好的内存分配和回收方法，通过测试框架进行验证。
4. 分析UniProton Buddy Allocator，使所有的算法都能百分百通过测试框架的样例。

实验总结：

1. 通过本次实验，学生应能够深入理解内存管理的核心概念和算法。
2. 学生应掌握如何在实际编程中应用这些算法，以及如何对现有系统进行改进以满足特定的需求。
3. 实验结束后，学生应能够对内存分配算法的效率和适用场景有自己的见解。

参考代码：

参考代码1：基础算法的定义

#ifndef \_\_BASIC\_ALOG\_H

#define \_\_BASIC\_ALOG\_H

typedef struct memblock\_s {

size\_t size; // 当前块的大小

int free; // 是否空闲

struct memblock\_s \*next; // 下一个块

} memblock\_t;

typedef struct mempool\_s {

int block\_size; // 每个块的大小

int free\_count; // 剩余块的数量

void \*mem; // 内存池起始地址

void \*ptr; // 指向第一个空闲块

} mempool\_t;

extern mempool\_t default\_mempool;

int memp\_init(mempool\_t \*mp, uint64\_t addr, size\_t block\_size);

void \*m\_malloc\_first\_fit(mempool\_t \*mp, size\_t size);

void \*m\_malloc\_best\_fit(mempool\_t \*mp, size\_t size);

void \*m\_malloc\_worst\_fit(mempool\_t \*mp, size\_t size);

void m\_free(mempool\_t \*mp, void \*ptr);

#endif

参考代码2：基础算法的实现

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdint.h>

#include <basic\_alog.h>

#define MEM\_PAGE\_SIZE 0x1000

mempool\_t default\_mempool;

void print\_memory\_pool(mempool\_t \*mp) {

memblock\_t \*current = (memblock\_t \*)mp->ptr;

printf("Memory Pool State:\n");

while (current != NULL) {

printf("Block at %p, size: %zu, free: %d, next: %p\n",

(void \*)current, current->size, current->free, (void \*)current->next);

current = current->next;

}

printf("\n");

}

void \*m\_malloc\_first\_fit(mempool\_t \*mp, size\_t size) {

memblock\_t \*current = (memblock\_t \*)mp->ptr;

while (current != NULL) {

if (current->free && current->size >= size) {

current->free = 0;

// 如果块大小远大于请求大小，则拆分块

if (current->size > size + sizeof(memblock\_t)) {

memblock\_t \*new\_block = (memblock\_t \*)((char \*)current + sizeof(memblock\_t) + size);

new\_block->size = current->size - size - sizeof(memblock\_t);

new\_block->free = 1;

new\_block->next = current->next;

current->size = size;

current->next = new\_block;

}

mp->free\_count--;

return (char \*)current + sizeof(memblock\_t);

}

current = current->next;

}

return NULL;

}

void \*m\_malloc\_best\_fit(mempool\_t \*mp, size\_t size) {

memblock\_t \*current = (memblock\_t \*)mp->ptr;

memblock\_t \*best\_fit = NULL;

while (current != NULL) {

if (current->free && current->size >= size) {

if (best\_fit == NULL || current->size < best\_fit->size) {

best\_fit = current;

}

}

current = current->next;

}

if (best\_fit != NULL) {

best\_fit->free = 0;

// 如果块大小远大于请求大小，则拆分块

if (best\_fit->size > size + sizeof(memblock\_t)) {

memblock\_t \*new\_block = (memblock\_t \*)((char \*)best\_fit + sizeof(memblock\_t) + size);

new\_block->size = best\_fit->size - size - sizeof(memblock\_t);

new\_block->free = 1;

new\_block->next = best\_fit->next;

best\_fit->size = size;

best\_fit->next = new\_block;

}

mp->free\_count--;

return (char \*)best\_fit + sizeof(memblock\_t);

}

return NULL;

}

void \*m\_malloc\_worst\_fit(mempool\_t \*mp, size\_t size) {

memblock\_t \*current = (memblock\_t \*)mp->ptr;

memblock\_t \*worst\_fit = NULL;

while (current != NULL) {

if (current->free && current->size >= size) {

if (worst\_fit == NULL || current->size > worst\_fit->size) {

worst\_fit = current;

}

}

current = current->next;

}

if (worst\_fit != NULL) {

worst\_fit->free = 0;

// 如果块大小远大于请求大小，则拆分块

if (worst\_fit->size > size + sizeof(memblock\_t)) {

memblock\_t \*new\_block = (memblock\_t \*)((char \*)worst\_fit + sizeof(memblock\_t) + size);

new\_block->size = worst\_fit->size - size - sizeof(memblock\_t);

new\_block->free = 1;

new\_block->next = worst\_fit->next;

worst\_fit->size = size;

worst\_fit->next = new\_block;

}

mp->free\_count--;

return (char \*)worst\_fit + sizeof(memblock\_t);

}

return NULL;

}

void m\_free(mempool\_t \*mp, void \*ptr) {

if (!mp || !ptr) return;

memblock\_t \*block = (memblock\_t \*)((char \*)ptr - sizeof(memblock\_t));

block->free = 1;

// 合并相邻的空闲块

memblock\_t \*current = (memblock\_t \*)mp->ptr;

while (current != NULL && current->next != NULL) {

if (current->free && current->next->free) {

current->size += sizeof(memblock\_t) + current->next->size;

current->next = current->next->next;

} else {

current = current->next;

}

}

mp->free\_count++;

//printf("Block freed at %p\n", ptr);

}

int memp\_init(mempool\_t \*mp, uint64\_t addr, size\_t block\_size) {

if (!mp) return -1;

memset(mp, 0, sizeof(mempool\_t));

mp->block\_size = block\_size;

mp->free\_count = 1; // 初始时只有一个大块

mp->mem = (void\*)addr;

if (!mp->mem) return -1;

mp->ptr = mp->mem;

memblock\_t \*initial\_block = (memblock\_t \*)mp->ptr;

initial\_block->size = MEM\_PAGE\_SIZE - sizeof(memblock\_t);

initial\_block->free = 1;

initial\_block->next = NULL;

return 0;

}