L1：内核内存分配 (kalloc)

截止日期

Soft Deadline: 4月14日23:59:59。

你的所有操作系统实验将在你之前的代码上完成，你只需要维护一份内容不断增加的实验报告(pdf格式)，其中每次实验的内容**不建议超过2页A4纸**。请在实验报告中描述你在实验中遇到的特别值得一提的事件，例如你代码的架构设计、特别精巧的实现、遇到印象深刻的bug等。

收到的作业

真正的“操作系统”

这次我们真的来写操作系统了，有一点小激动嘛！框架代码的目录名也变成了“kernel”。框架代码与游戏十分类似，甚至比游戏还更少一些，不过这次 `main` 函数上来就开启了多处理器。

多处理器编程多多少少让大家有些陌生，因此在我们进一步向前之前，我们先在多处理器上实现一个简单的程序——内存分配和回收。物理内存管理是操作系统内核中最重要的基础设施之一，因此也值得花一个实验的时间好好研究一下。

背景

在实现操作系统内核时，我们经常会面对存储空间的分配：当我们创建一个进程/线程时，我们需要分配数据结构(相应在终止的时候回收)；打开一个文件时，需要分配空间存储偏移量；……很自然我们要实现一组API来管理内存空间的分配和释放，否则对于每一种类型的资源，都要手工实现分配和释放。

在多处理器系统中，各个处理器上的代码会并发地申请或释放内存。这就给内存分配和释放带来额外的挑战：一方面，我们希望不同处理器能并行、高效地申请内存，不会因为同时申请而发生一个处理器等另一个处理器的情况，且在很短的时间内完成分配；另一方面，在此要求下实现malloc/free (尤其是不使用互斥锁)又是十分困难的。做好准备好了？做不好也得做好了。

实验描述

获取实验代码

本学期的所有代码(minilab, OSlab)都在同一个目录中完成。请参考代码获取与提交 (OS2019_Code)。

在你获得os-workbench之后，执行

```
git pull origin L1
```

会得到 `kernel/` 目录。它与Lab0的框架十分类似(但目录组织和Makefile稍有区别，大家可以阅读一下。

实验提交

在 `kernel/` 执行 `make submit` 提交。考虑到以后实验都使用这个目录，我们没有设置 `TASK` 环境变量，请设置成正确的名称(L1)后 `make submit` 。

在这个实验中，实现以下函数：

```
static void *kalloc(size_t size) {
  // 内存分配
}

static void kfree(void *ptr) {
  // 内存释放
}
```

功能描述

在AM启动后，`[_heap.start, _heap.end)` 都是可用的内存，`kalloc/kfree` 会管理这个区域的内存。其中 `dkalloc` 用于分配 `size` 字节的内存，返回分配内存的地址；`kfree` 用于回收一个已分配的内存地址。如果 `ptr` 为 `NULL` ，不执行任何操作。

更直观地说，`kalloc(s)` 返回一个指针p，并且保证内存的$[p, p+s)$被“分配”，即能被进程(包括进程内的所有线程)使用，直到调用 `kfree(p)` 为止，这段内存再次被“释放”，释放后再访问这段内存的行为未定义(use-after-free bug)。你基本上可以把本实验理解成在bare-metal上实现C标准库中的 `malloc` / `free` 。

允许在中断处理程序中使用 `kalloc/kfree` 。(提示：使用自旋锁时应当关闭当前处理器上的中断)。

功能正确性

我们希望你尽可能保证算法实现的正确性——但这不是绝对的，例如你可以不实现free：

```
static void free(void *ptr) {}
```

这样 kalloc 的实现就非常简单了——简单到只需要维护一个指针即可。我们不严格禁止这样的实现，而且你会发现，即便这样实现，在内存充足的系统里，你编写的操作系统依然是可以运行的(只是运行一段时间以后，就会out of memory无法再运行下去)。

此外，因为内存分配无法“预测未来”，一旦内存分配出去，就不能移动了，从而可能在系统中总可用内存充足时，无法分配足够的连续内存。所以根据分配策略不同，实际内存分配成功与否也是不确定的。你只需要尽力实现kalloc/kfree的正确性即可，我们允许在内存紧张时分配失败。

因此，你可以选择任意你喜欢的分配策略，但你需要设计测试用例尽可能地测试你的算法。

设计测试框架

你不能修改 framework/ 目录下的代码。我们在测试时，会使用我们自己实现的主函数，在 os→init() 完成后执行我们的测试代码。为了尽可能地从测试中生还下来，你当然希望自己设计好测试框架，实现 make test 可以自动完成对你代码的测试。

实现测试，你需要的是：(1) 好的workload，在多个处理器上模拟各种不同类型的内存访问模式：频繁的小内存申请；频繁的大内存申请；混杂的申请；多处理器竞争的申请/释放…… (2) 验证在workload下的正确性。能够快速反复运行测试是成功的第一步。

一个额外的提示是，在native上测试会更容易一些。

线程安全性

大家第一次在多处理器上编程，一定会遇到很多麻烦。因此我们提供了多处理器测试的途径：

```
make run2
make run4
```

分别可以运行在2个/4个处理器上，每个处理器都会打印出一个Hello信息，例如一次执行看到的结果：

```
Hello from CPU #2
Hello from CPU #3
Hello from CPU #4
Hello from CPU #1
```

打印的顺序

为什么顺序是2, 3, 4, 1，而不是1, 2, 3, 4呢？结合AM代码可以回答这个问题。

虚拟机 vs. 真机

我们**强烈建议在物理机上安装Linux完成实验**，此时QEMU可以开启KVM硬件虚拟化(我们已经帮大家开启了)，从而更好地暴露你程序中的bug。如果在虚拟机中完成实验，虚拟机用**线程**模拟处理器，很多在真实机器上可能产生的调度，在tcg模式下可能被屏蔽，从而导致损失成绩。

当然我们也会在tcg模式下测试你的代码，如果正确可以得到较多的基本分数。

联想到之前课上各种黑人问号的例子，你不禁想，要不还是上个锁吧：

```
void *kalloc(size_t size) {
  spin_lock(&alloc_lk); // 上锁
  void *ret = kalloc_unsafe(size); // 具体实现分配的算法
  spin_unlock(&alloc_lk); // 解锁
  return ret;
}
void kfree(void *ptr) {
  spin_lock(&alloc_lk); // 上锁
  kfree_unsafe(ptr); // 具体实现释放的算法
  spin_unlock(&alloc_lk); // 解锁
}
```

如果你希望实现各种fancy的算法，这是个不错的主意——从一个你几乎能确信正确性的策略开始，再逐步把锁拆开，例如在大内存分配时上锁，小内存时则使用无锁的算法。

顺序测试下功能正确不等于并发意义下功能正确

很可能你的顺序测试完全通过，但多处理器跑起来就挂了。不要慌，多加点printf logs，慢慢诊断可能的问题。

然后你会惊奇地发现，也许加了一个printf，错误就不见了。这时候你可能需要调整workloads、增加延迟等保证bug的稳定再现，然后再进行诊断。

性能

因为kalloc/kfree类似于操作系统的“轮子”——各种操作系统服务都建立在内存管理的基础上。因此这个轮子实现的好坏直接关系了操作系统的性能。所以如果大家查看Linux Kernel的内存管理算法，会发现它为了能在各种内存分配场景下都有不错的表现，非常复杂。

在这个实验里，你可以自由体验任何内存管理算法。以下事实可能会对你有帮助：

- 在我们的操作系统里，内存分配并不是非常频繁的操作，因此通常不会是性能瓶颈；
- 考虑操作系统中实际的workload，主要是频繁的小内存分配和不太频繁的大内存分配；
- 分配内存的主要是操作系统中的对象，少部分时候是字符串。

> **抛开workload谈优化，就是耍流氓**
>
> 怎么获得workloads呢？请参考课程的课件 (/wiki/OS2019_C3.slides)。

实验指南

这个实验给你的完全是“白板”的代码，不包含任何测试。你必须为你自己的代码负责，自己写好测试，自己说服自己你的代码是正确的——迟早有一天大家需要脱离框架，自己写出正确的代码。

代码组织

实验框架代码由三个目录组成：

- framework - 框架代码，你不能修改其中的文件。我们会通过替换框架代码来测试你提交的作业。
- include - 你的头文件存放的目录，可以在其中建立任意的头文件，例如 `my_os.h` ，在源代码中使用 `#include <my_os.h>` 引用。
- src - 实际实现的源代码目录。添加的 `.c` 文件都会被编译。

理解框架代码的编译，最好的办法是看Makefile啦：

```
NAME            := kernel
SRCS            := framework/main.c $(shell find -L ./src/ -name "*.c")
INC_DIR         := include/ framework/
```

然后：

```
export AM_HOME := $(PWD)/../abstract-machine
ifeq ($(ARCH),)
export ARCH := x86-qemu
endif

PREBUILD := git
include $(AM_HOME)/Makefile.app
include ../Makefile.lab
```

最后

```
QEMU_FLAGS := -serial stdio -machine accel=kvm:tcg -drive format=raw,f
run2: image
    qemu-system-i386 -smp 2 $(QEMU_FLAGS)
run4: image
    qemu-system-i386 -smp 4 $(QEMU_FLAGS)
```

静态内核模块

我们用C语言实现了简单的“模块”机制，这样可以帮助大家理解操作系统执行的流程，更好地理解操作系统是如何实现的。让我们看看模块是如何组织的。使用模块分为模块的声明和定义两部分，和C中变量的声明/定义类似。

```
#define MODULE(name) \
  mod_##name##_t; \
  extern mod_##name##_t *name
#define MODULE_DEF(name) \
  extern mod_##name##_t __##name##_obj; \
  mod_##name##_t *name = &__##name##_obj; \
  mod_##name##_t __##name##_obj =
```

我们的实现主要在PMM模块:

```
typedef struct {
  void (*init)();
  void *(*alloc)(size_t size);
  void (*free)(void *ptr);
} MODULE(pmm);
```

模块包含三个函数指针： `pmm→init()` 初始化模块，在系统初始化时，由启动的一个处理器调用(在 `os→init()` 中调用，而这是在 `_mpe_init()` 调用前调用的)。你会在这里完成你表示堆区数据结构的初始化、锁的初始化等内容。 `pmm→alloc()` 和 `pmm→free()` 负责内存管理。

模块的声明(`MODULE(pmm)`)宏展开如下:

```
typedef struct {
  void (*init)();
  void *(*alloc)(size_t size);
  void (*free)(void *ptr);
} mod_pmm_t;
extern mod_pmm_t *pmm;
```

声明了 pmm 这个模块变量，实际的定义在 alloc.c：

```
MODULE_DEF(pmm) {
  .init = pmm_init,
  .alloc = kalloc,
  .free = kfree,
};
```

它会定义 pmm，并在静态时完成函数指针的初始化。

框架代码的运行流程

AbstractMachine应用的代码都从 main 开始执行，非常简单：

```
int main() {
  _ioe_init();
  _cte_init(os→trap);
  // call sequential init code
  os→init();
  _mpe_init(os→run); // all cores call os→run()
  return 1;
}
```

处理器首先执行一定的初始化操作。初始时，只有一个处理器(编号为0)运行代码：

- 初始化IOE;
- 初始化CTE中断管理，在每次中断/异常到来时都执行os模块的代码 os->trap()；
- 调用os模块的初始化；
- 启动多处理器，每个处理器都执行 os→run()。

os模块中的 trap 和 on_irq 将会在下个实验中用到，大家留空即可。