操作系统 L1-L2 实验报告

张天昀 171860508

2019 年 5 月 5 日

L2 多线程

L2 太难了 + 要素过多，不知道该从哪里写起。

概要

本次实验新增的内容有：

- `mt-safe.h`：打包 `printf` 来提供多线程输出；
- `os.[ch]`：定义 `handler` 结构体和中断处理函数；
- `semaphore.[ch]`：信号量和相关函数；
- `thread.[ch]`：多线程调度、信号量等待；
- `syscall.[ch]`：系统调用（仅实现了睡眠操作）。

所有的数据结构全都是链表，基本没有容量问题。

信号量睡眠

为了实现信号量等待时的睡眠，我分别尝试了以下方法：

- 设置状态为睡眠然后 `_yield()`，会因为此时别的进程已经发出信号但没有收到而死锁。
- 设置状态为即将睡眠然后 `_yield()`，会在 `_yield()` 前一刻被中断（或者被其他处理器抢先处理）导致切换到了即将睡眠的线程然后 boom。

最后我成功浪费了半个五一假期来解决这个问题导致的各种奇葩死锁和 bug。假期结束后在床上突然领悟到只要用一把大锁保证程序进入 `sem_wait` 后一定能够成功睡眠（原子性）即可。

于是从 PA 把 `syscall` 的内容移植了过来，设计了两个新的系统调用选项：

- `SYS_sem_wait` 系统调用首先检查睡眠前是否已经被唤醒（即不需要进入睡眠状态），如果需要睡眠则将当前线程设置为睡眠状态、设置唤醒变量，然后调用 `sched()` 切换线程；

- `SYS_sem_signal` 系统调用记录当前唤醒信号，并将所有需要唤醒的睡眠线程唤醒。

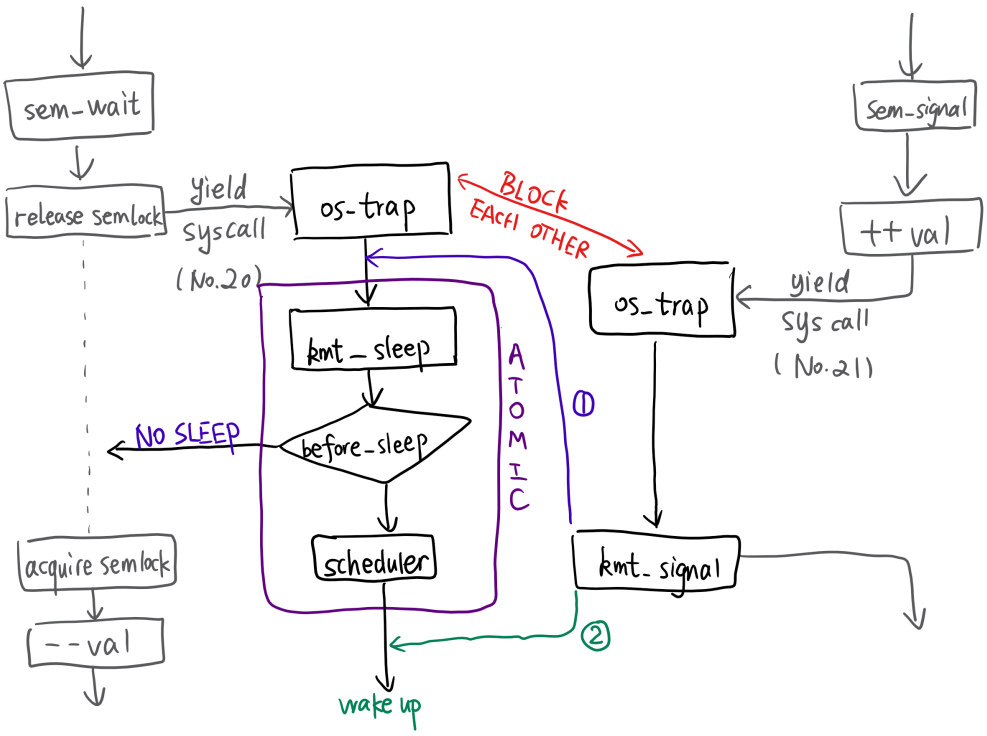

图 1:

放弃直接睡眠改用系统调用的好处：

- 没有复杂的开关锁顺序，逻辑更加简单，~~代码更加优美~~；
- 一把大锁锁住其他的 CPU 不能进入中断，阻止了各种奇葩睡眠竞争的发生。

祖传 bug

本次实验总共发现并修复了很多个 PA 祖传 bug：

- 提供了 `memmove` 函数，不使用额外内存。
- 修正了 `printf` 打印-2147483648 会因为取反整数溢出，输出 10 个符号的 bug。
- 修正了 `memset` 赋值为负会因为有符号数移位右边补 1 导致赋值错误的 bug。

L1 内存分配

数据结构

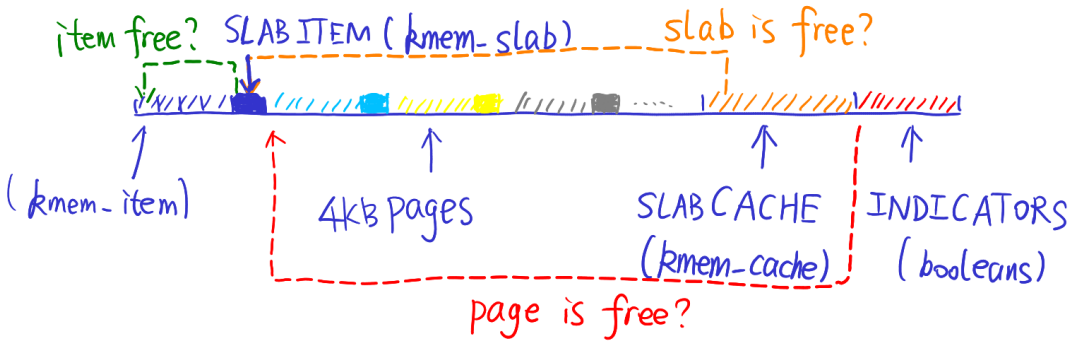

图 2: 内存区域划分与指针链接

为了防止在各种坑人的内存测试中挂掉，我采用了 Slab 内存分配方式，每种内存大小对应一个 cache，每个 cache 有多个 slab 对象，每个 slab 对象对应一个 4KB（或者更大的）内存块，包含多个内存对象。

整个内存被分为三个区域（对应图中从左往右）：

- 页面区：可分配的内存空间
- 缓存区：分配 slab cache 对象
- 指示器：用 bool 类型指示对应的内存块是否可用

每个 slab cache 为 `kmem_cache` 类型，包含两个链表，分别对应可用的 Slab 和已经全部占用（满的）slab。每个 slab 为 `kmem_slab` 类型，也分别保存两个链表，指向可用的内存对象和不可用的内存对象。为了回收方便，我在每个内存对象的前面贴了一个 `kmem_item` 类型的数据，指向 slab。

分配、回收过程

内存分配时，先计算所需 `size + kmem_item` 的大小，找到对应大小的 cache。如果对应的 cache 有空闲的 slab，直接使用，否则进行增长操作。如果所需大小小于 512B 则分配 1 个页面，否则分配 8 个对应对象所需要的内存页面数。在空闲的 slab 中取出一个空闲的内存对象，将指针加上 `kmem_item` 对应的偏移量返回给用户。

回收内存时，将指针减去对应的偏移量，得到 `kmem_item` 的指针，然后取出 `kmem_slab` 的地址，将该内存对象重新标记为可用即可。

接口封装与自旋锁

内存的分配、回收在进入 `alloc` 和 `free` 函数时直接上自旋锁，先计算所需大小，然后调用具体执行的函数 `kmem_alloc` 与 `kmem_free`，内部调用返回后再解开自旋锁，返回获得的结果。

存在的问题与解决方案

在某位超级大佬提供的测试代码中我的~~无懈可击的完美~~程序挂了，因为大佬的测试代码分配的内存大小随机，而我的代码每种大小都会单独生成 cache，很快 cache 分区就爆了。

解决方案很简单，对于每次内存分配，都找到不小于所需要的大小的 2 的次幂。这样 cache 的种类就非常少了（10 种即可覆盖 1KB 以下的所有需求）。而由于对 `alloc` 和 `free` 函数进行了封装，修改起来也非常简单。只需要在进行内部调用前修改所需大小的值，把修改后的值传入调用即可。例如：

```
void* alloc(size_t size) {
    lock();
    size_t real_size = cal_real_size();
void *ret = kmem_alloc(real_size);
    unlock();
return ret;
}
```