操作系统设计与实现 **2020 春**

实验报告

指导老师：蒋炎岩 姓名：王晨渊, 学号：181220057

1 L1

通过反复刷分，利用 OJ 的性能起伏，我成功在某次提交中得到的所有 hard 测试 accept 的结果，美滋滋。

1.1 实现方法

我尝试了空闲链表和 slab 两种方法。

测试发现空闲链表效率偏低，因此我最终使用了 slab，因此下文也仅介绍 slab。

slab 的核心在于同种类型（指页面中存储的单个内存块的大小）的页面串成链表，内存不够时获取全局的大锁来分配新页面。

首先将需要分配的内存大小上取整到最近的 2 的幂。

接着，分配一定大小 (0x8000 字节) 的一个 page 用于存储大量的该大小的内存块。其中 page 的头部存放了 bitmap 用于索引 page 内部的大量相同大小的内存块。

最后将 page 串在该内存大小对应类型的链表中。

slab 体现了 fast path 与 slow path 相结合的设计思路。

在 alloc 时：

对于某个 page 的访问，需要获取对应 page 的锁才能进行，这是 fast path。

当某类型的所有 page 都满的时候，获取全局的大锁分配新的 page。这是 slow path。

在 free 时：通过 free 的内存地址计算出对应的 page 的起始位置，获取对应 page 的锁然后将 bitmap 进行修改即可。

1.2 学会的奇技淫巧

1.2.1 上取整的正确写法

```
#define F(x,y) ((((x)-1)/(y)+1)*(y))
```

1.2.2 上取整到 2 的幂的倍数

```
#define Align(num, align) (((num)+((align)-1))&~((align)-1))
```

其中 num 是需要被对齐的数，align 是 2 的非负整数次幂

1.2.3 取出从右往左第一个 1

#define MASK(x) ((x)&(~(x)+1))

1.2.4 取出第一个 1 的位置

Built-in Function: int __builtin_clz (unsigned int x)

Returns the number of leading 0-bits in x, starting at the most significant bit position. If x is 0, the result is undefined.

Built-in Function: int __builtin_ctz (unsigned int x)

Returns the number of trailing 0-bits in x, starting at the least significant bit position. If x is 0, the result is undefined.

Built-in Function: int __builtin_clzl (unsigned long)

Similar to __builtin_clz, except the argument type is unsigned long.

Built-in Function: int __builtin_ctzl (unsigned long)

Similar to __builtin_ctz, except the argument type is unsigned long.

1.2.5 快速对齐 2 的幂

#if __SIZEOF_POINTER__==8
#define UP(x) 1«(64-__builtin_clzl((x)-1))
#else
#define UP(x) 1«(32-__builtin_clz((x)-1))
#endif

1.3 经验教训

C 代码中的字面的十进制整数默认是 int，如果移位操作符移的位数超过了 int 的范围会导致 UB。

如果希望字面的十进制整数为其他类型，可以使用 UL 等后缀。

2 L2

经过艰苦卓绝的一周 debug 之旅，我终于 accept 了。

2.1 实现方式

我深入研究了 XV6 后，采用了以下的实现方式。

定义一个 cpu_local 结构数组，用于记录 cpu 私有的一些信息，用于实现锁、task 的调度与信号量。

task 链表的设计。我将链表设计为了一个带锁的 queue，只支持 deque 与 enque 两种操作，且操作必须获得锁。

值得注意的是，idle 的 task 是额外定义的全局变量，原因如下：idle 的栈不是用 kalloc 分配的内核栈；懒线程其具有特殊性，对完成任务没有帮助，调度时应当进行特殊处理，不应该放在 task 链表中。

关于自旋锁的实现，我高度与 xv6 相似，这里不再赘述了。而 on_irq 是一个非常 trival 的为函数指针链表添加节点的函数，我也不再赘述。我将重点放在我的 kmt_context_save 与 kmt_schedule 函数的实现上。

我在 cpu_local 结构中放了 last 与 current 指针用于指向上一次的任务与当前任务，保证了第一次 trap 选取的新线程，会在第 3 次 trap 才放回到 task queue 中，从而保证不会发生 stack race。为了提高代码的优美程度，我使用了函数 mycpu() 返回对应 cpu 私有的结构体。

具体调度的工作流程如下：

如果 cpu 不是第一次中断，且现在运行的任务不是 idle，则把 last 对应的线程 enque 到 task queue 中。再把寄存器现场保存到当前的任务中，即 mycpu()->current 的 context 中。第二个判断条件是因为 idle 线程是不应该出现在 task queue 中的。

接着从 task queue 里找到第一个可以运行的 task，将当前任务设置为 last，将选取的新任务设置为 current，返回新任务的寄存器现场即可。

值得注意的是，每次选取新任务是真的把任务从 task queue 里拿了出来，这是为了防止同一个寄存器现场被另一个 cpu 恢复。

信号量的实现也不复杂，见代码即可。值得注意的是，我的 P 操作有可能因为数据竞争发生多余的 yield，这是因为解锁之后另一个线程可能已经进行了 V 操作，使得那次 P 操作的 yield 是不必要的。但是我认为这个仅仅是不严重地影响了性能，对正确性没有影响，因此没有进行修改。

2.2 debug 检查

我利用 cs 寄存器为 8，rip、eip 大于 0x10000，小于 &etext 进行了对 context 检查。此外，我还在 on_trap 中对 event 的类型进行了检查。

2.3 经验教训

2.3.1 stack race

2.3.2 手滑毁一生

kalloc 了指针的大小，而不是指针解引用的类型的大小。

2.3.3 死锁

在 sem_signal 中如果获取 task queue 的锁会导致如果在 handler 中调用 sem_signal 会死锁。