L0:amgame

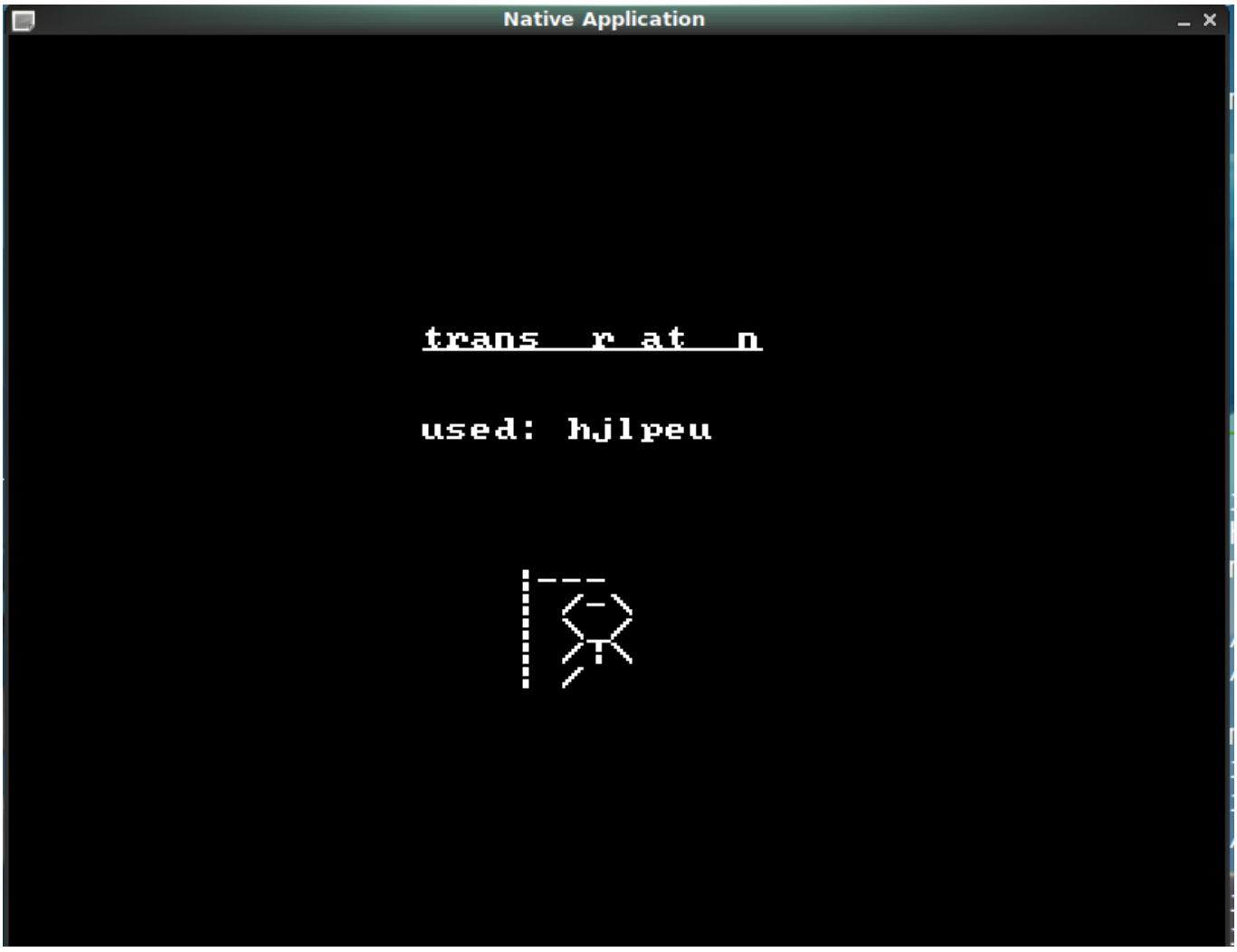
171860575 毛一鸣

实现游戏简介

hangman:猜词小游戏

规则：输入字母猜单词，如果当前字母是单词的一部分则该字母会在单词的对应位置显示；如果输入的字母不属于当前单词，则下方的“上吊小人”图像被更新，只有7次猜错的机会，当“上吊小人”图像完成时游戏失败，在图像完成之前猜出当前单词的所有字母则游戏成功。成功或失败之后均会自动进入下一轮。

游戏界面：



成功猜词：

未猜出：

思路

借鉴了am中打字小游戏的c语言实现，上学期数电大实验实现打字小游戏的经验也提供了有益的帮助。

游戏主界面被分为4个部分：gameboard(填词的区域)，usedboard(显示已经用过的不正确的字母)，hangman(“上吊小人”图像区域)，finishboard(游戏结束后显示"success"或"GG"的区域)。游戏初始化时，根据屏幕的宽度与长度，在`init_screen()`中设置上述4个部分的左上角顶点的坐标。接下来通过`reset_game()`初始化一局游戏，包括重置相关变量，绘制初始图像等。其中，绘制图像都是通过函数`draw_character()`和`redraw()`实现的，而`redraw()`函数内部调用am提供的`draw_rect()`和`draw_sync()`，图像的绘制在此处与底层硬件层相接。

游戏的主循环中，每隔`1000 / FPS`毫秒检测一次键盘输入，如果检测到键盘按键抬起，则对该按键对应字母进行检查，若与单词中的字母匹配，则更新gameboard，并将剩余字母数left减小；若不匹配，则更新hangman区域内的图像，并将剩余机会数chance减1.若chance或left为0，则游戏结束，在finishboard区域绘制"success!"或"GG".等待1秒后，调用`reset_game()`，重新开始游戏。

遇到的问题和解决方案

调用klib时提示冲突

原因在于`game.c`中默认提供的函数与klib中定义的函数原型冲突了。经过权衡后，决定删去提供的默认函数，因为klib中除了默认函数，还有包括`stdio`、`stdlib`等其他库的内容，兼容klib无疑是更好的选择。

字符画数组无法初始化

编译时报错，报得五花八门，最后发现是用字符`\`初始化数组时，其默认被识别为转义符而非原字符造成的，将其改为`'\\'`解决了问题。如果是一个单独的字符，也许这个问题很容易被发现，然而当它混在字符画中时，事情就变得没那么简单了orz

开始新的游戏之后，屏幕上会残留之前的字符

显然是由于屏幕没有被擦除所致，因此每次开始新的一局游戏之前，首先将gameboard、usedboard、hangman、finishboard四个区域对应的数组清零，并调用`redraw()`进行一次重绘，这样就会在清屏之后进行新的绘制了。

游戏中多次输入正确字符，会被多次判定有效

逻辑上，当单词中的一个字母被正确输入后，再次输入该字母就不再有任何效果了。然而在代码实现时，每次检测到正确的字母时剩余字母数都会减少，导致最后剩余字母数变为负数。为此，在检测到正确的字母后，将该位置的字符改为'*'，就避免了这一问题。

L1:kalloc

架构

本次实验的代码架构参考了著名的《The C Programming Language》,尽管经典而精简，但是仍然用了一整天才能参透其中每一行的深意orz

由于`kalloc`和`kfree`是动态更新的，因此便于动态修改的链表自然是最佳的数据结构，而链表的每个结点都代表一个内存块。从这一视角来说，`kalloc`的工作本质上只有两项：若当前free链表中没有结点有足够的空间满足所申请的大小，则向`am`申请空间，若当前空间满足申请大小，则从free链表中“挖”去一块。“挖”的过程也分为两种情况，若free链表中的某个结点的大小正好等于所申请的大小，则直接在链表中删去这个结点；若某个结点的大小大于所申请的大小，则将这个结点的可用空间减去申请的大小。而`kfree`的工作则是将被释放的地址所在结点插入回free链表（因为被释放的地址首先是通过`kalloc`申请的，因此本质上它仍是一个结点，只是暂时脱离了整个free链表），若这个结点左右两侧有地址相邻的结点，则将它们合并以避免空间的碎片化。

为了维护链表，定义结构体HEADER存储当前结点的大小以及下一个结点的地址，并将HEADER放置在每个结点的首地址处，结点p被`kalloc`申请时，实际返回地址为`(HEADER*)p + 1`,从而避免`HEADER`被修改，在被`kfree`时仍然可以利用其信息。（这个技巧和蒋神上课讲的HACK技巧有异曲同工之妙）

我们可以将整个堆区视为free链表的可用空间，但是为了高效的搜索和空间的优化，free链表不会一开始就使用堆区的所有空间，而是在当前空间不足时才通过`morecore`函数向堆区申请空间。在`am`中，堆区的申请可以直接通过增加`pm_start`的值，并将修改前的`pm_start`地址作为返回值。若`pm_start`加上申请的空间大小超过了堆区的最大范围`pm_end`，则返回NULL，代表向`am`申请堆区空间失败。在原书中，设置了`NALLOC`值作为`morecore`向系统申请大小的最小值，若申请空间大小小于`NALLOC`则将空间大小赋值为`NALLOC`，但是这是由于在正常情况下，通过系统调用申请空间是十分耗时的，因此需要设置这样一个缓冲区的大小。然而在`am`中，申请空间只是一个加法的过程，几乎不需要时间，这样一个缓冲区反而增加了内存的碎片化，因此我取消了这一缓冲区的优化。

而多线程并发的问题要如何解决呢？根据KISS法则以及OSTEP第322页提到的`Knuth's Law`:

> Premature optimization is the root of all evil.

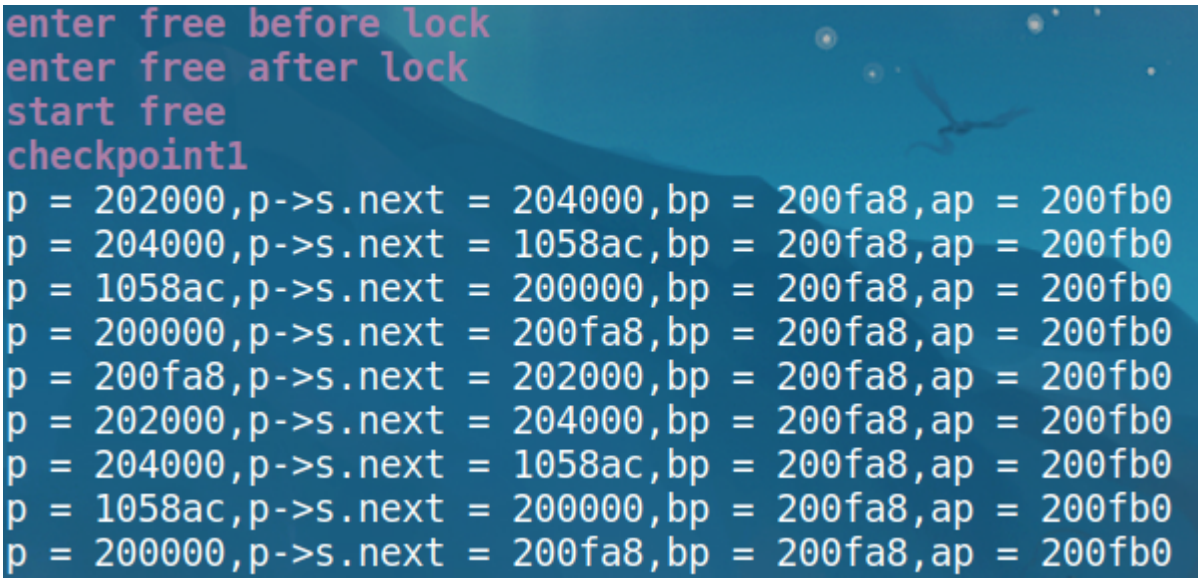
首先用最简单粗暴，也必然正确的方法：为`kalloc`和`kfree`函数直接上一把大锁。如果性能欠佳，再考虑优化。目前来看，即使上了一把大锁，`kalloc`和`kfree`的时间性能还是完全能够满足需求的。

遇到印象深刻的bug

第一次尝试多线程并发编程，碰到什么bug都会印象深刻（因为真的很难下手啊）。最棘手的一个，还是在调用`kfree`后出现的程序卡住的问题。

程序卡住不动了，第一反应是就是是否出现了死锁？然而我的`kfree`和`kalloc`使用的是同一把锁，按理不可能出现死锁的情况，但是以我目前可怜的并发知识水平，我不敢妄加确定。怎么办呢？做PA和蒋神的名言告诉我们，打Log可以解决所有问题。于是我在`kfree`的`lock`前后，主函数前后以及`unlock`前后各打了一个Log，结果发现：在一次执行时，主函数前的Log成功打印，而主函数后的Log却迟迟没有出现。

这说明我的推理还是正确的：问题没有出在锁上，而是`kfree`函数本身。乘胜追击，通过简单的Log夹逼，我发现问题出在了一个`while`循环上。打印一下相关变量的值，bug终于浮出水面：



然而事情并没有这么简单，我检查了好久`kfree`的代码，发现其判断逻辑并无问题。而最终bug的来源，居然藏在`kalloc`中的一处逻辑错误：申请内存空间时，更新后的结点地址的偏移量应该等于申请的空间大小，而我却设置成原结点分配空间后的剩余空间大小了。

修正后的代码。bug在于将`p += p->s.size`写成了`p += nunits`：

```
            //suppose size of p is 70,nunits = 50,p = a(an address)
            //after executing the following codes, prevp->s.next =
a, a->s.size = 20
            //p = a+20(an address), p->s.size = 50, now it has
nothing to do with the free linked list
            //return p+1 = a+21, as the available space.
            else{
                p->s.size -= nunits;
                p += p->s.size;
                p->s.size = nunits;
            }
```

修改之后，`kfree`终于可以正常工作啦。

L2:kthread

感想

Concurrency considered harmful

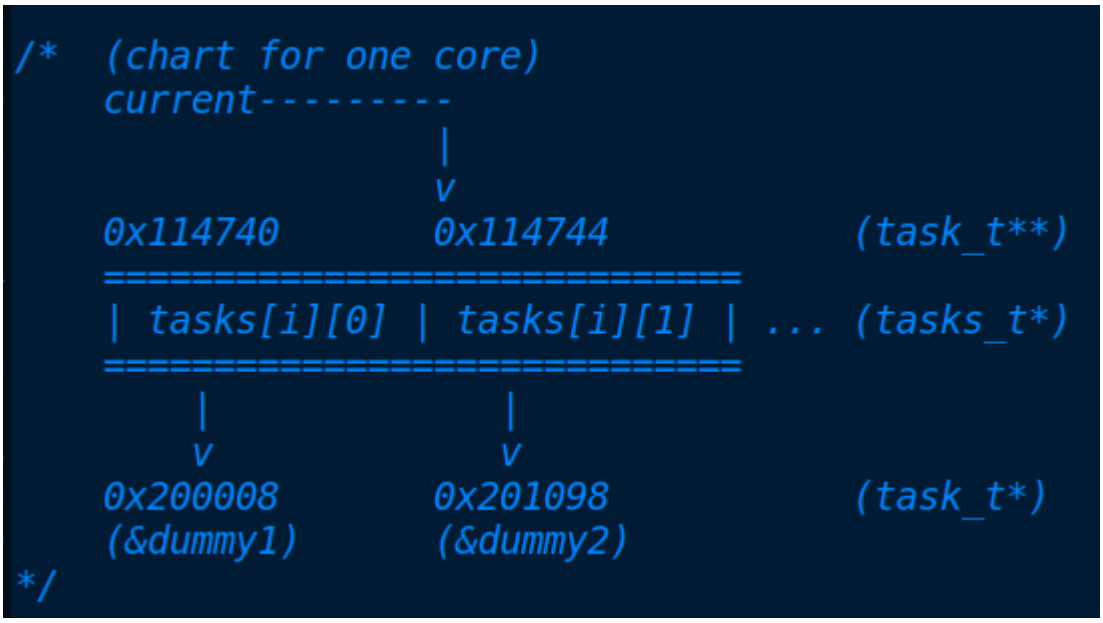
Think twice, code once

到目前为止做过的最hardcore的Lab，没有之一。PA4.2拼死拼活好歹花了三天给整出来了，这次的L2从五一开始整整做了七天，中间没有碰任何其他作业，每天起床第一件事就是debug，晚上睡前也在想怎么debug，实在是收(tong)获(ku)良(wan)多(fen)

这次做L2的过程让我意识到了一个很不好的习惯，可能是长期做OJ所致，觉得写完了代码就是胜利，先跑跑看，能过最好，不能过就慢慢debug吧。实际上，很多花了很长时间才找到的bug其实都是拜前期思考不周所赐，如果一开始就抱着对代码十二分负责的态度而非面向bug编程的思想，一定能省下更多宝贵的时间吧。

架构

总体架构基于讲义信息和供参考的`thread-os-mp.c`，与后者的区别之处在于`thread-os-mp.c`中的线程是静态分配的，而L2需要动态分配，因此实现时采用了二重指针，并使用二维数组，将线程与cpu绑定，二维数组`tasks`的第i行中存储CPU #i的线程结构`task_t *`的地址。`tasks`的任意一行的示意图如下:



线程切换时，按`tasks[_cpu()]`中的存储顺序循环调度线程。

遇到印象深刻的bug

1. 我还没有执行os->run(),谁提前把中断给开了？

   在调试过程中，意外发现明明初始化工作尚未完成，却已经有提示中断的Log出现了。提出猜想：是否初始化过程中的代码已经打开了中断？通过在初始化过程中打印`_intr_read()`的值，发现其值为1，印证了这一猜想。那么是谁把中断给开了？通过`objdump`反汇编`kernel-x86-qemu`并查找`sti`发现，`sti`只在`_intr_write()`中出现，而我当时还暂时没有实现锁和信号量，L2的代码一次`_intr_write()`都没有调用。可是中断就是被打开了，这简直是天方夜谭。

   没有办法，只能通过一系列的调试先确定改变IF的代码位置，发现在`kmt_create()`前后，中断由关闭变为打开状态。这时我注意到，`kmt_create()`中有一个`task`参数，而这个参数需要调用L1中实现的`alloc()`来分配内存...

   介于L1中我实现的锁十分辣鸡，我开始意识到我可能是被上一个实验的遗留问题给坑了。果不其然，在L1中，我的`unlock()`函数每次都会打开中断，然而在初始化阶段，中断一开始就是关闭的，即使上锁再解锁之后，仍然应保持关闭状态。这就解释了中断为何会被提前打开。这一惨痛教训也使我理解了xv6的spinlock中`mycpu()->intena`的含义：intena即为INTerrupt-ENAble，如果其值为false，说明上锁时中断已被关闭，`unlock`时也不应打开中断。

2. 上下文保存、切换之后，为什么线程不能执行？

   一开始我创建了两个线程进行测试，程序始终处于卡死的状态，让我百思不得其解。机缘巧合之下，我把线程数增加到了五个，惊奇发现`make run2`时有三个线程可以运行了，而`make run4`只有一个线程可以运行。我立即感觉到我发现了一些不得了的东西，继续增加线程数后发现，`make run2`始终有2个线程无法正常运行，而`make run4`则保持在4个。

   继续探究，通过打印保存上下文时`os_trap()`传入的`ctx`值发现，第一次切换时传入的`ctx`是`os->run()`对应的上下文，然而我在`tasks`中并未为其分配存储位置，而是只为通过`kmt_create()`创建的线程存储了信息，这就导致第一次上下文切换时，`os->run()`的上下文被存入通过`kmt->create()`创建的线程的上下文中，导致`kmt_switch()`切换到后者时，实际上切换到了`os->run()`中的`while(1) _yield()`,而非创建时指向的函数，因此看起来就好像什么反应也没有，不能执行。

3. 运行时大概率触发“重复获得锁”的panic

   刚刚成功运行驱动设备，喜悦立即被触发的panic无情浇灭。panic的信息是在尝试获得锁时，提示该cpu已经获得锁。为了解决问题，我在触发panic前打印了触发panic的锁的名称，结果是`fb` —— 一个设备驱动程序中的内部锁。这就让人无从下手了，因为我不能修改设备驱动程序，理顺其中的逻辑与调用关系同样困难重重。

   但是bug还是要解决的，我和同学讨论时，发现有一些同学也碰到了类似的问题，不少同学指出了可能的问题或者他们自己的解决方案，如`os_trap()`中触发了多重中断，或者调度算法中存在bug等。遗憾的是，这些问题都不是这个bug的原因。最后与一位同学讨论后才发现，原因竟然是我的`spinlock`在初始化以及`unlock()`时，照搬xv6的代码，将`cpu`初始化为0。然而估计xv6中cpu是从1开始计数的，而am中`_cpu()`的返回值却是从0开始的。这就导致一旦0号cpu试图访问一把上锁的锁，就会触发这个panic！把初始化值由0改为-1，这个困扰了我一天多的问题就解决了...

4. 程序运行一半突然卡住不能继续执行

   玄学的问题还在继续，自己写了一个producer（输出loop次左括号）和consumer（输出loop次右括号）进行测试，当consumer成功输出loop次右括号之后，程序突然卡住不动，连log都一条也不输出了。尝试在`os_trap`中输出接收到的`event`编号，结果一个`event`也没有收到。这说明中断又在该打开的时候给关上了。

   为了解决这个bug，我对`spinlock`中执行`_intr_write(1)`（即`sti`）处进行观察。 执行`_intr_write(1)`需要满足两个条件:`!cpu_ncli[_cpu()] && cpu_intena[_cpu()]`,打log发现，`cpu_intena[_cpu()]`居然一直都为0，也就是说其值并未被正确初始化。顺藤摸瓜，初始化处的代码如下：

```
_intr_write(0);  //cli
if(cpu_ncli[_cpu()] == 0){
        cpu_intena[_cpu()] = _intr_read();
}
cpu_ncli[_cpu()] += 1;
```

   这一看，先执行`cli`再读IF，IF当然一直都是0，所以`cpu_intena[_cpu()]`也是0，把cli移到最后，问题就解决了。可是我为什么会在一开始写代码的时候犯下如此低级的错误而没有想到呢？又是参考xv6的`spinlock`时麻痹大意了。xv6中的对应代码如下：

```
int eflags;

eflags = readeflags();
cli();
if(mycpu()->ncli == 0)
    mycpu()->intena = eflags & FL_IF;
mycpu()->ncli += 1;
```

   写`spinlock`前看过am中`_intr_read()`的实现，就是返回`get_efl() & FL_IF`,可万万没想到，xv6中的`eflags`是先于IF读取好的，而`_intr_read()`是同时读的，这一进一出，差的就是一个`cli`的时间，然后把自己给坑了。