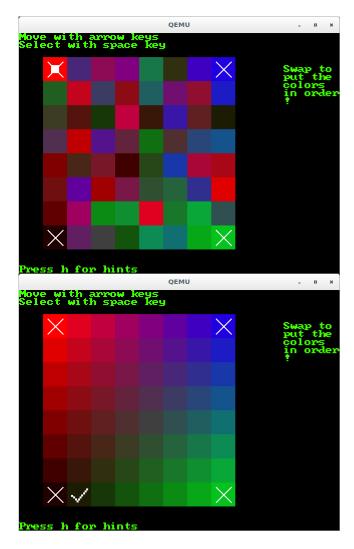
# L0 直接运行在硬件上的小游戏(amgame)

## 游戏截图



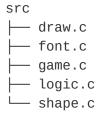
### 游戏内容

游戏会随机\*打乱GRID×GRID(在include/game.h中定义)个方块。其中,颜色为渐变色\*。 玩家通过上下左右键移动,按下空格选中方块。选中两个方块后会自动交换。玩家要使界面内的颜色重新 恢复原本的渐变色。

注1:随机指的是游戏文件中调用了srand(uptime()),基于rand函数的特性及uptime自身的不确定性,可以近似认为游戏达到了随机

注2:渐变色指,在初始的图中,同一行/列的方格,RGB值为三个等差数列

#### 文件结构



我的源代码分为5个文件。

game.c为游戏主循环,只包含主要的函数调用。

shape.c提供绘制基本图形的函数,如画叉、圆、箭头、打钩、光标及填充指定方格等。

font.c提供在图形界面上打印字符的函数,包含ASCII编码下所有字符的点阵,一个打印单个字符的函数 (都是静态),打印字符串的函数(非静态)

draw.c提供较复杂的绘图的接口,包括初始化屏幕和在光标选中格打印光标或提示(由print\_flag决定)logic.c提供复杂逻辑的判定,包括游戏初始化、交换两个方格、计算颜色渐变值、获得下一个输入按键、对输入按键进行处理。

#### 设计亮点

- 考虑到如果在主循环中进行判定while(uptime()< next\_frame),会导致按键的延迟,因此,我没有在主循环中使用时间相关的判定。只在少数需要体现出时间特性的部分单独加入了判断(如光标闪烁频率、长按按键的反馈)。这样可以做到,玩家按下按键后,第一时间\*处理该按键信息,且长按时表现出若干毫秒处理一次的特性。
- 考虑到native和qemu的分辨率不同(同时还要考虑兼容其它分辨率),include/game.h 中 SIDE 被定义成 16会导致在不同分辨率下的效果差很多。所以我经过计算和尝试,将其改成w/40。使代码能够适应大多数分辨率。同理,draw\_str中提供的size参数,我也使用了SIDE以实现自动适应屏幕的效果。但由于整型数精度问题,文字的适应效果有限。
- 考虑到游戏难度较大(作者自己都玩不过),我增加了Hint功能,按下h即可显示提示
- 为了节约空间,没有使用canvas数组,因此在打印时,为了表现出多层的效果,在进行判定时代码量稍多一点。
- 由于裸机本身不提供透明通道,alpha参数被我用来记录其它数值(该方格是否为固定方格)

### L1 kalloc

## 实现原理简述

先将从pm start到pm end的堆区分为若干个8 KB大小的页面。

其中,使用类似buddy system和区间树的方法进行管理,管理方式如下,

(由于线段树方式存在bug,后来改用了链表)

有一个全局free\_list链表和每个处理器自己的free list。当kalloc空间时,先上处理器单独的锁,若处理器自己有足够高的空间,则直接分配。

否则,调用global\_alloc函数,此时需要上全局锁,然后从全局链表中获取一段内存。考虑到小内存分配

的频繁,如果处理其内部内存不足时,在调用global\_alloc时,如果需要的内存小于PG\_SIZE,则申请PG\_SIZE的内存(否则多次小内存申请也会导致全局锁被阻塞)

kfree的时候,将内存且还给处理器独有的free list,因此,全局的free list可用内存会越来越少

#### 具体规则为

先将需要的空间大小向上取到16的倍数(减少日后合并的碎片数量)

从free list的头开始往后搜寻,找到第一个空间充裕的块(first bit)。如果该块除了提供所需空间之外,还足够提供头部的空间,则在尾部产生一个header,header的size记录本次分配的内存大小,便于kfree释放。如果该块不足以为新的header提供空间,则在维护好链表的指向关系后,直接返回当前header的space

header声明如下

```
struct header{
    struct header *next;
    uintptr_t size;
    struct{} space;//space doesn't take any storage
    //directly return &space
} static free_list[4]={};
```

其中, next指向下一个free list的节点, size用于标记目前区域的大小。

在申请空间时,先将需要的空间向上变为16的倍数,如果该header对应的区域大小大于所需空间,则从header的尾部划出所需空间加sizeof(header)大小的空间,并将尾部转化为一个header,将size置为分配的实际大小以便free函数解析,然后返回尾部的space的地址

#### 设计亮点:

- 在header的定义中,包含一个大小为0的space成员,通过取该成员的地址,可以直接得到分配的空间的起始地址(由于标准中说struct的成员顺序一定按照声明顺序,故在内存上排列方式一定如此),且不会浪费空间
- 每个处理器的free list的第一项是一个Sentinel,用于简化判断逻辑,由于每次free时会考虑该部分是 否能和前后的free space互相merge,所以,每个处理器提供一个sentinel,由于分配的时间不同, Sentinel和free list里的free space必然不相连,Sentinel一定不会和要free的space互相merge,极大 的简化了判断逻辑
- kalloc类似于一个装饰器,它调用kalloc\_real,但可以在调用前后做一些必要的处理,简化某些逻辑 (降低多个return维护难度)

#### 测试代码:

(免责声明:本人与其余同学互相交流/分享了测试函数,因而可能在查重时会出现重合,可以通过git记录的时间证明本人并非抄袭)

```
void test(){
    void *space[100];
    int i;
    for(i=0;i<100;++i){
        space[i]=pmm->alloc(rand()%((1<<10)-1));
    }
    for(i=0;i<1000;++i){
        int temp=rand()%10;
        pmm->free(space[temp]);
        space[temp]=pmm->alloc(rand()&((1<<10)-1));
    }
    for(i=0;i<100;++i){
        pmm->free(space[i]);
    }
}
```

该段代码中有一个大小为100的指针数组space,初始化后,space的每个元素指向一个kalloc返回的空间。

然后,每次随机选一个元素,将其空间free,并重新申请一段新的大小随机的空间。

经过若干次free、alloc循环后,再将space里的每个元素free,然后打印free list,如果实现无误,free list 应该只包含若干个很大的块,不应该存在碎片或者free list中互相重叠

```
static inline void fill(uint8_t *p,int a,int b,int len){
    for(int i=0;i<len;++i){</pre>
        p[i]=(a+=b);
    }
}
static inline void check(uint8_t *p,int a,int b,long long len){
    for(int i=0;i<len;++i){</pre>
        report_if(p[i]!=(uint8_t)(a+=b));
    }
}
void memory_test(void *dummy){
    int *a=pmm->alloc(MAXN*sizeof(int)),
        *b=pmm->alloc(MAXN*sizeof(int));
    int cpu_id=_cpu();
    long long len[MAXN];
    void *p[MAXN];
    for(int i=0;i<MAXN;++i){</pre>
        len[i]=(1<<((i&15)+5))+rand();
        a[i]=rand()+cpu_id;
        b[i]=rand()+cpu_id;
        p[i]=pmm->alloc(len[i]);
        fill(p[i], a[i], b[i], len[i]);
    }
    for(int j=0;j<MAXM;++j){</pre>
        int i=rand()%MAXN;
        check(p[i], a[i], b[i], len[i]);
        pmm->free(p[i]);
        int shift=13+(rand()&7);
        len[i]=rand()+((1<<shift)-1);</pre>
        if(i&1){
             len[i]+=1<<12;
        a[i]=rand()+cpu_id;
        b[i]=rand()+cpu_id;
        p[i]=pmm->alloc(len[i]);
        fill(p[i], a[i], b[i], len[i]);
    }
    for(int i=0;i<MAXN;++i)</pre>
        pmm->free(p[i]);
    printf("[cpu%d] finish memory test.\n",_cpu());
}
```

类似前一个测试,这个测试增加的是,会向申请的内存里填充一段内容,并且由于fill的性质,内存被写入相同的序列的概率非常低。因此,能通过这个测试基本可以判定内存分配没有重叠

#### 印象深刻的bug:

在nemu中,由于没有访问/读写权限的控制,如果声明了过大的局部变量,由于每个进程只分配了4KB的 栈空间,因此,会导致stack overflow,修改到cpuinfo甚至更低地址的重要系统内存,因此,我在先使用小 的局部数组跑了上面的测试后,使用pmm->alloc声明了大内存,并将局部数组改为一个指向申请的空间的指针。

## L2 内核多线程 (kthreads)

#### 代码组织架构

我在kernel/src下建了test文件夹,里面存放了针对不同部分(调度、信号量、自旋锁)的单元测试,并且通过相关的宏src/os.c:52-56,快速调用相关的测试。

为了防止过多测试函数影响可用内存大小,我增加了NO\_TEST宏。在完成测试后,#define NO\_TEST,则相关的测试代码和相关变量不会被生成,可以减少空间的占用

## 印象深刻的bug

之前在做的时候,发现task的属性(runable, running, slepping)的读、写都是与、或操作,于是使用lock add和lock or,希望能够略微提高效率(虽然影响不大,至少是难能可贵的尝试),但是发生的问题是,运行一段时间,就会有一些进程永远进入睡眠。

后来,我先花了一段时间重读了代码,删除一部分冗余代码,并且采用单元测试的思路,逐步缩小范围, 最终消除了bug(但是并不知道究竟为什么之前的写法会有错)

#### 学到的经验是:

- 单元测试比系统测试能够更好地定位bug所在,并且在有良好架构(如我写的那个宏)的情况下,单元测试的方便性非常之高。
- 遇到问题时,可以采用差量的方式来定位bug:我的信号量测试函数,是一个生产者和一个消费者互相唤醒,但总是运行一段时间之后就会发生死锁和\$eip"飞走"的情况,于是,我开始删除一些会影响信号量正确性的代码,比如wait之后不维护信号量的list,也不sleep,发现这样不会发生死锁。于是再针对将进程放入信号量的池的部分的代码进行修改