

实验报告 L0

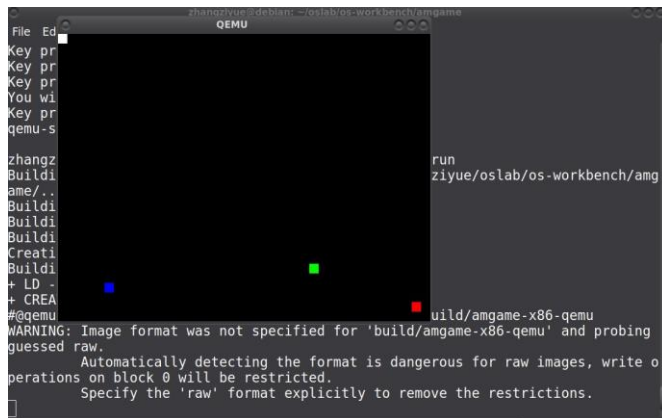
171860578 张梓悦

实验进度：

我完成了实验所要求的所有内容。

功能介绍：

需要使用上下左右键操作小白点，使其顺利到达红点的位置。其中蓝点和绿点随机移动，若触碰到这两个点则游戏失败。



代码框架设计：

```
int next_frame = 0;
while (1) {
    while (uptime() < next_frame) ; // 等待一帧的到来
    read_key();
    move();
    if(px == rx && py == ry) {
        printf("You win!\n");
        reset();
    }
    else if((px == x1 && py == y1) || (px == x2 && py == y2)) {
        printf("You lost!\n");
        reset();
    }
    //printf("lfps\n");
    next_frame += 1000 / FPS; // 计算下一帧的时间
}
```

在一个 while 循环中一帧一帧地接收键盘信息并移动小格点。由于本次实验是一个熟悉库函数的实验，故基本使用了全部库函数用以测试库函数的正确性。

其中使用 strcmp 来判断键盘信息：

```

if(strcmp(key_names[event.keycode], "UP") == 0) {
    if(py > 0)
        py--;
    clean();
    display();
}
else if(strcmp(key_names[event.keycode], "LEFT") == 0) {
    if(px > 0)
        px--;
    clean();
    display();
}
else if(strcmp(key_names[event.keycode], "RIGHT") == 0) {
    if(px < mw - 1)
        px++;
    clean();
    display();
}
}

```

遇到的 bug:

由于 git 没有添加库函数文件，导致回滚的时候 klib 直接变成了原始的那种直接 return 0 的状态，小游戏直接挂掉，找了半天才发现是这个问题。

实验报告 L1

171860578 张梓悦

实验进度:

我完成了实验所要求的所有内容。

代码设计(内存分配的方式参考了 R&K 中的实现):

```

void cli() {
    asm volatile ("cli");
}
void sti() {
    asm volatile ("sti");
}
void lock(intptr_t *lk) {
    cli();
    while (!_atomic_xchg(lk, 1));
}
void unlock(intptr_t *lk) {
    _atomic_xchg(lk, 0);
    sti();
}

```

首先定义了一把大锁防止由于多线程的问题干扰程序的测试。这把大锁将用在 hello cpu 的输出上（否则会输出重叠），也会用在 malloc 和 free 进行内存分配、释放的时候。

```
typedef union header {
    struct {
        union header *next;
        unsigned size; //real size = s
    };
    Align x;
}Header;
```

```
static Header base;
static Header *freep = NULL;
```

定义一个链表结构保存空闲块，双向链表并不能降低插入的时间，故我们使用单向链表即可。

```
char *sbrk(int size){ //we have written it in PA
    if(pm_start + size > pm_end)
        return (char*)-1;
    else {
        //printf("%d\n",pm_start);
        pm_start += size;
        return (char*) (pm_start - size);
    }
}
```

sbrk 虽然没有提供，但完全可以按照手册上对其的定义写一个功能一样的函数。

```
for (p = pre->next; pre = p, p=p->next) {
    if(p->size>=nu) { //this block is fit
        if(p->size==nu)
            pre->next=p->next;
        else {
            p->size-=nu;
            p+=p->size;
            p->size=nu;
        }
        freep=pre;
        return (void*) (p+1);
    }
}
```

由于我们为了使分配的块能够均匀分布，故每次 alloc 后表头 freep 都会向后移动一个块。

由于每一个块的头部是链表结构，故我们在返回时需要返回 head+1 的地址。

```
unsigned nu = (nbytes+sizeof(Header)-1)/sizeof(Header)+1;
```

同时也要分配 size+Header 结构体大小的内存。

```
if(bp +bp->size == p->next) { //merge with next
    bp->size += p->next->size;
    bp->next = p->next->next;
}
else
    bp->next = p->next;

if(p + p->size == bp) { //merge with p
    p->size += bp->size;
    p->next = bp->next;
}
else
    p->next = bp;
```

free 的时候相邻的块能合并就将他们合并。

```
static void *kalloc(size_t size) {
    lock(&alloc_lock);
    void *ret = my_alloc(size);
    unlock(&alloc_lock);
    return ret;
    //return NULL;
}

static void kfree(void *ptr) {
    lock(&alloc_lock);
    my_free(ptr);
    unlock(&alloc_lock);
}
```

最后一把大锁杜绝一切 bug。

精巧的设计：

Header 结构中 size 的单位为 sizeof(Header)，这样的设计不仅解决了对齐问题，同时在插入和合并计算新节点的头的时候只需执行 $p + p \rightarrow \text{size}$ 即可。由于 size 不是真实的 size 大小，故不必将 p 强制转换成 char* 再进行加法，使得代码看起来简洁美观。

遇到的问题：

- ①起初我使用的是一个链表结构体的数组来模拟内存池，但是由于数组的大小是固定的，过多次数的 malloc 可能在堆区还未放满的时候内存池已经满了，同时真正的操作系统不可能像这样外挂一个存储模块来管理内存。于是查阅资料发现真正的 malloc 实现都是将每一个块的头部设为链表的结点，于是参考 R&K 的实现方式写了一个新的内存管理函数。
- ②free 函数中需要判断一下 p 是否在空闲链表中扫了一整遍后是否找到，否则会因为 free 相同的地址而陷入死循环。

实验报告 L2

实验进度：

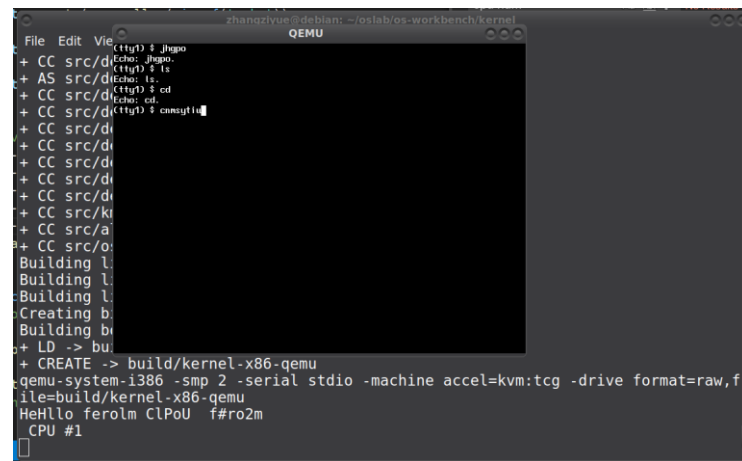
我完成了实验所要求的所有内容。

可以保证线程的创建，以及生产者消费者运行正常

生产者消费者代码：

```
static void producer(void *arg){
    while(1){
        for(volatile int i = 0; i < 1000000; i++){
            kmt->sem_wait(&empty);
            kmt->sem_wait(&mutex);
            cunt++;
            printf("%d ", cunt);
            kmt->sem_signal(&mutex);
            kmt->sem_signal(&full);
        }
    }
}

static void consumer(void *arg){
    while(1){
        for(volatile int i = 0; i < 1000000; i++){
            kmt->sem_wait(&full);
            kmt->sem_wait(&mutex);
            cunt--;
            printf("%d ", cunt);
            kmt->sem_signal(&mutex);
            kmt->sem_signal(&empty);
        }
    }
}
```



tty 也可正常使用，只是加载进去的速度非常慢

代码设计：

锁的实现参考了 xv6，信号量以及生产者消费者按照课本给出的定义进行设计。信号量的设计精髓在于使用数组作为挂起线程的线程池，简化了代码，并且没使用链表从而降低了找 bug 难度。线程的保存及切换，参考了讲义里给的代码，并自己定义了一个链表来保存异常处理时需要使用的结构体。

遇到的问题：

运行一定时间就会出现死锁，即不能把每个函数都加上锁，否则大概率撞车。故只要在 ostrap 里加一把大锁，而其中调用的上下文保存切换就无需再上锁。Cpu 不与线程绑定，虚拟机就会不炸，但是我发现这是因为在运行的线程上下文突然被修改而造成。故我们可以添加一个 runnable 的状态，只准上下文切换的时候修改 runnable 的线程，从而可以规避该 bug。还有就是之前一直在用虚拟机进行实验，虚拟机对多核的调度不知道使用了什么

奇怪的机制，tty 加载出来需要半分钟，生产者消费者跑几秒钟就挂了。之后用 github 将代码转移到真机上运行，这些问题就都不存在了。

实验报告 L3

实验进度：

我完成了实验所要求的所有内容。

已知 bug：

cat 操作只支持 proc/文件夹中的文件，不支持 mnt/里面 a.txt 以及 hello.cpp

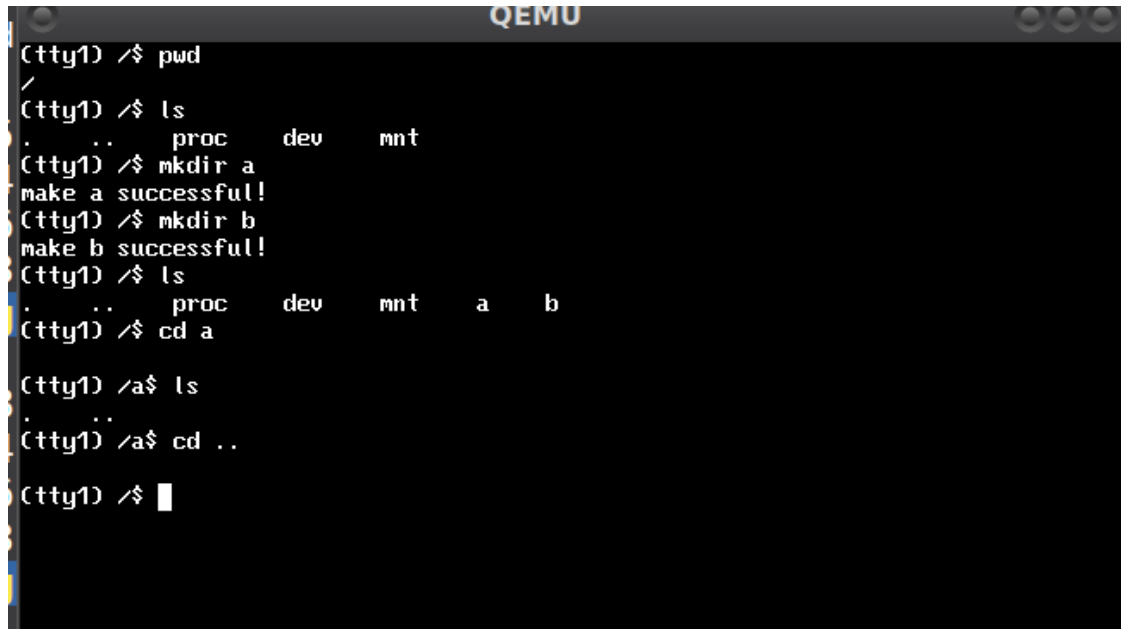
支持的操作：

pwd:查看目录

ls: 查看文件夹内容

mkdir: 建立新文件夹

cd: 进入文件夹，相对路径绝对路径均可



```
(tty1) /$ pwd
/
(tty1) /$ ls
.  ..  proc  dev  mnt
(tty1) /$ mkdir a
make a successful!
(tty1) /$ mkdir b
make b successful!
(tty1) /$ ls
.  ..  proc  dev  mnt  a  b
(tty1) /$ cd a

(tty1) /a$ ls
.  ..
(tty1) /a$ cd ..

(tty1) /$
```

rmdir: 删除文件夹

link: 链接两个文件(硬链接，删除其中一个两个均会消失)

unlink:解除链接

cat: 打印(仅支持在/proc 文件夹中打印对应信息, 打印 ext2 文件系统中的文件会出 bug)

```
(tty1) /$ mkdir c
make c successful!
(tty1) /$ ls
.  ..  proc  dev  mnt  a  b  c
(tty1) /$ rmdir c
remove c successful!
(tty1) /$ ls
.  ..  proc  dev  mnt  a  b
(tty1) /$ link a b
link /a -> /b successful!
(tty1) /$ ls
.  ..  proc  dev  mnt  a -> b  b
(tty1) /$ unlink a
unlink /a successful!
(tty1) /$ ls
.  ..  proc  dev  mnt  a  b
(tty1) /$ link a b
link /a -> /b successful!
```

```
(tty1) /$ ls
.  ..  proc  dev  mnt  a -> b  b
(tty1) /$ rmdir b
remove b successful!
(tty1) /$ ls
.  ..  proc  dev  mnt
(tty1) /$ cd proc

(tty1) /proc$ ls
.  ..  cpuinfo  meminfo  idle1  idle2  input-task  tty-task
(tty1) /proc$ cat cpuinfo
cpuinfo:
pid:6
name:print1
cpu_number:

pid:
name:idle1
cpu_number:1

(tty1) /proc$ cat meminfo
meminfo:
using mem: 8650808 b
free mem: 123469768 b
tot mem: 132120576 b
(tty1) /proc$
```