# L0: 直接运行在硬件上的小游戏(amgame)

**截止日期**

Soft Deadline: 3月24日23:59:59。

需要提交以学号命名的.pdf格式实验报告。除非特殊情况，实验报告**不建议超过2页A4纸**。请在实验报告中描述你在实验中遇到的特别值得一提的事件，例如你代码的架构设计、特别精巧的实现、遇到印象深刻的bug等。

如果你的qemu在 `-accel tcg` 上报错，可以去掉这个运行选项。

## 收到的作业

---

**引子：如何为“裸机”编程?**

对于初学者来说，实现操作系统最大的困难就在于，“操作系统就是个C程序”这件事在细节上实在难以理解。因此“写个操作系统”好像一直都是个朝圣一般的事情，在知乎上也被很多人吹过。

但退一步说，一旦理解“操作系统就是个C程序”，写操作系统也就一点也不神秘了，我们的C程序从 `main` 开始按照我们写的代码运行，我们编写硬件相关的代码管理中断、I/O、虚拟存储……仅此而已。

这个实验就用实际行动告诉大家：真的可以在硬件上运行C程序。不信？写个小游戏试试，它就是操作系统的原型——你甚至可以很容易地把它改造成一个“批处理”操作系统。

## 背景

因此你总是想去找点资料，比如《Orange's OS》，就会发现这个书大部分时间都不在讲操作系统，而是在讲与操作系统毫无关系的什么GDT是什么，IDT是什么，TSS是什么……当年学《计算机系统基础》的时候被迫读过手册，但过了点时间以后，好像也忘记得差不多了。在花了很多时间把这些概念搞清楚以后，还没开始施展拳脚写真正的操作系统，书就结束了。当然更多的人是直接倒在了前面：无论讲解得多么好，都不可能在短时间内让你彻底理清x86手册。x86就是x86。

为什么大家会遇到这样的困难呢？虽然教科书上说了一万遍“**机制与策略分离**”，但在“写个操作系统”方面，机制和策略却没有很好地被分开。对着x86讲操作系统实在是没有必要：操作系统，说白了就一个C程序，我为什么非要去理解

GDT, IDT, TSS？**x86的确提供了足够的机制用来实现操作系统，但这不代表它提供了初学者实现操作系统的最佳机制**。尤其是ISA通常都是硬件设计者兼顾上层软件需求和物理硬件实现难度时做出各种权衡得来的，我们的确没必要这样嘛。所以Top2的T大就做了个不错的选择：uCore系统运行在MIPS32上，这个体系结构学起来简单多了，没有什么GDT，有非常优雅的MMU。最近他们在大动干戈把系统移植到RISC-V上。

操作系统实验到底该做什么呢？为了把“操作系统就是个C程序”的理念贯彻到底，我们设计一个硬件抽象层(hardware abstraction layer)，在这个抽象层上能实现现代操作系统，同时这个抽象层也很容易在各个硬件上实现。

第一个实验里我们就熟悉一下这个抽象层，顺便复习一下计算机硬件，编写一个小游戏。之后在体系结构实验中，你自己实现了MIPS或RISC-V的CPU，你写的游戏就可以不经过任何修改在你的CPU上运行起来，很酷不是吗?

## AbstractMachine

请阅读AbstractMachine文档 (AM)。之后你会在编程的过程中不断熟悉它，所以你需要仔细阅读它的代码，我们主要关注参考实现：QEMU模拟的x86系统，但AbstractMachine可以实现在更多的CPU上：MIPS, RISC-V, 甚至是NEMU模拟器。

我们在C程序中调用AbstractMachine API就能实现操作系统，就正如我们在应用程序里调用操作系统API一样。这体现了计算机系统设计与实现的**抽象**，底层系统提供**机制** (mechanism)，上层系统利用这些机制实现**策略** (policy)。

## 实验描述

### 获取实验代码

本学期的所有代码(minilab, OSlab)都在同一个目录中完成。请参考代码获取与提交 (OS2019_Code)。

在你获得os-workbench之后，执行

```
git pull origin L0
```

将Lab0所需的带代码(abstract-machine)和更新后的 `amgame/Makefile` 拉去到本地。在 `amgame/` 执行 `make` 即可编译； `make run` 即可执行:

- 不必配置 `AM_HOME` ， Makefile中已经进行了指定;
- Makefile指定了 `ARCH=x86-qemu` ;
- `make run` 会启动一个两个处理器的虚拟机运行。

### 实验提交

在 `amgame/` 执行 `make submit` 提交。需要提交以学号命名的pdf实验报告。

编写一个直接运行在AbstractMachine上(仅仅启用IOE扩展，不启用其他扩展)小游戏：

- 完成klib的实现(你们已经实现过了，因此复制过来即可)。klib里已经实现了访问I/O设备的API，阅读它们的实现，以及x86-qemu中的代码。
- 兼容很简单的处理器：即小游戏运行中只调用TRM和IOE API，而不使用其他API，且游戏使用的内存(代码、静态数据、堆区总和不超过1MB)——这样大家的游戏就可以在各个CPU上广为流传下去啦！从下一个实验开始我们将开启多处理器之旅。
- 你可以做任何小游戏，哪怕只有几个像素点在屏幕上乱动也可以(就可以获得全部正确性分数，哪怕有bug)，只要满足：(1) 有肉眼可辨认的图形输出；(2) 能使用键盘与游戏交互。
- 你的游戏应当是可以在多个硬件体系结构之间移植的。

### 编写可移植的代码

你需要兼容以下情况：

- 适配不同的屏幕大小。不同的体系结构中，屏幕大小可能是320x200、640x480、800x600等，你的游戏最好能在所有分辨率下都获得不错的体验。
- 32/64bit，因此你应当使用 `intptr_t` 或 `uintptr_t` 来保存指针数值。
- 大/小端，因此你不能把不同大小的指针类型强制转换。

我们会在native (64bit Linux)和x86-qemu (32bit)两个环境下运行你的游戏，你的游戏必须在两个环境下都能正确编译并运行良好。

AbstractMachine的项目中自带了两个游戏：打字游戏和LiteNES，大家可以参考它们的代码。

### 考虑游戏性能

你的游戏可能运行在频率仅有10MHz的自制处理器上。因此，如果你希望你的游戏能在未来你自己写的处理器上流畅运行，减少每一帧的运算量和图形绘制就显得比较重要了。你可以计算出一个时钟周期大致允许运行的代码量(假设绘制一个像素也需要若干条指令)，相应设计游戏的逻辑。

我们认为优秀的游戏会被收录到游戏池中，永久保存下来！

# 实验指南

参考AbstractMachine文档 (AM)和项目中附带的代码，学习如何在AM上编程。

## AbstractMachine上的程序

建立一个在AbstractMachine上的项目，我们首先需要配置 `AM_HOME` 和 `ARCH` 环境变量。如果你第一次使用Linux，这会是一个比较痛苦的过程——你需要学习很多shell有关的知识、修改 `.bashrc` 脚本等等……刚开始你可能会碰到挫折，不过不要慌，慢慢就会好起来的。

我们可以参考AM项目里的Hello World程序。它由一个.c文件和一个Makefile组成：

```
#include <am.h>

void print(const char *s) {
  for (; *s; s ++) {
    _putc(*s);
  }
}
int main() {
  print("Hello World from " __ISA__ " program!\n");
  return 0;
}
```

真的就是一个C程序耶，而且这个程序可以直接在裸机上运行。Makefile也非常简单：

```
NAME := hello
SRCS := hello.c
include $(AM_HOME)/Makefile.app
```

看来大部分事情AM都帮我们搞定了，可以直接在裸机上编写C程序。现在我们来编译运行它。在终端中执行

```
make ARCH=x86-qemu
```

会自动完成编译，而执行

```
make ARCH=x86-qemu run
```

可以看到QEMU启动，并在QEMU的调试终端上打印出“Hello World from x86 program”。就这么简单，我们就实现了为x86硬件编程！

思考题： **__ISA__**

这个宏不是C标准里定义的，那么是谁定义的？当我们设置不同的ARCH，就会生成不同硬件的代码和镜像。想知道这是怎么办到的，当然需要自己动手啦！

UNIX世界早就给我们提供了足够的工具来搞定这些日常琐事：

```
find . -type f | xargs grep -ne __ISA__
```

(请RTFM理解这个命令的含义)。我们能定位到Makefile.compile中的一行代码，然后就豁然开朗了。

## 实现游戏之前：实现库函数

将你们在《计算机系统基础》课程中实现的klib复制到你们的repo(`abstract-machine/`)中。

如果没有实现过？那你最好实现一下。

## 实现游戏：简单的任务

游戏不过是个死循环：

```
next_frame = 0;
while (1) {
  while (uptime() < next_frame) ; // 等待一帧的到来
  while ((key = readkey()) != _KEY_NONE) {
    kbd_event(key);           // 处理键盘事件
  }
  game_progress();            // 处理一帧游戏逻辑，更新物体的位置等
  screen_update();            // 重新绘制屏幕
  next_frame += 1000 / FPS; // 计算下一帧的时间
}
```

大家见过的LiteNES和仙剑奇侠传，都是这样的循环。

再比如(你想实现一个简单的)弹球游戏，你只需要维护弹球的坐标$x, y$和两个方向的速度$v_x, v_y$，然后每一帧更新即可。配上碰撞检测，那就是个很酷的弹球啦！

**不要使用浮点数**

在x86-qemu里用是没问题的，但如果你想在你的MIPS32里实现IEEE754，我保证你会想吐血的。

# 小游戏与操作系统

一个小游戏的复杂程度，也许已经可以和简单的批处理操作系统相比了。批处理系统管理很多jobs，类似游戏也可以看成是一个大循环：

```
while (1) {
  Job *job = get_job();
  if (job) {
    load(job);
  }
}
```

我们不妨可以假设“批处理”系统是这样工作的：

1. 批处理系统执行的命令由键盘输入，因此 `get_job()` 就是从键盘读取按键并解析成命令，和游戏读取按键类似。
2. 执行的命令(job)是保存在磁盘上的ELF格式的二进制文件(虽然批处理系统时代还没有ELF)，那么 `load()` 函数的功能就和加载AM程序的loader类似，从磁盘中读取ELF头，然后把相应的节加载到内存的正确位置，然后执行

```
((void(*)(void))elf_header→entry)();
```

把控制权交给job，job结束后调用函数返回指令，就会自然返回到大循环中——这样我们可以很容易地把全班同学的游戏做成一个“合集”，批处理系统就是“游戏菜单”，就像NES的合集游戏那样：


### 开启操作系统之旅

现在我们已经学会怎么在“裸机”上编程啦！很简单，真的就是写C程序——我也已经理解应该如何实现批处理操作系统。从下一个实验开始，我们将正式开启多处理器和现代操作系统的旅程。