L2: 内核多线程 (kthreads)

截止日期

Soft Deadline: 5月19日23:59:59。

你的所有操作系统实验将在你之前的代码上完成，你只需要维护一份内容不断增加的实验报告(pdf格式)，其中每次实验的内容不建议超过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。

发布的tty driver等是experimental状态(很多功能不全/未实现)，因此很可能有bug，发现问题请随时和jyy报告。

一些FAQ/勘误：

- tty代码在申请内存时，会默认数据被清零。这个行为在L1中并没有document，因此如果希望跑起tty，请在你的实现中添加这个行为。
- teardown是设计为给大家Lab4时使用的，你可以不实现它，测试数据中没有teardown。

收到的作业

实现分时多线程

大家在kalloc的实验中已经熟悉了如何为一个多处理器的bare-metal编程。借助AbstractMachine提供的机制(C Runtime、物理内存、中断/异常等)，我们可以很容易地实现能够管理多个共享内存程序运行(线程)的“操作系统”。甚至我们的课堂上也演示过两份短小的代码：

- thread-os.c (/static/wiki/os/2019/demos/thread-os.c)：一个分时调度固定数量线程、round-robin调度的嵌入式操作系统；
- thread-os-mp.c (/static/wiki/os/2019/demos/thread-os-mp.c)：把刚才的代码扩展到多处理器。

大家在理解了框架代码以后，就能实现真正的“操作系统”啦！

背景

在之前的实验中，我们已经不知不觉为“实现操作系统”打下了很多基础。对第一次接触操作系统的同学们来说，虚拟化中的“虚拟CPU”，也就是让多个运行的程序能够分时共享处理器是操作系统中最神秘的部分之一。现在，从概念上我们已经

知道这是由“中断机制”实现的，在进程/线程执行时中断到来，操作系统代码开始执行并保存处理器运行的寄存器现场；在中断返回时，可以选择任何一个进程/线程已经保存的寄存器现场恢复，从而实现上下文切换。

在这个实验中，我们实现多处理器操作系统内核中的内核线程API (就像pthreads库，或是课堂上展示的 `threads.h`)。在完成这个实验后，你已经得到了一个真正的嵌入式操作系统，它可以运行在没有MMU的硬件上！

实验描述

在上一个实验中，我们已经实现了 `pmm` 模块，能够在多处理器中实现内存管理:

```
typedef struct {
  void (*init)();
  void *(*alloc)(size_t size);
  void (*free)(void *ptr);
} MODULE(pmm);
```

这个实验在 `pmm` 的基础上，进一步实现内核线程相关的操作系统内核API:

```
typedef _Context *(*handler_t)(_Event, _Context *);
typedef struct {
  void (*init)();
  void (*run)();
  _Context *(*trap)(_Event ev, _Context *context);
  void (*on_irq)(int seq, int event, handler_t handler);
} MODULE(os);

typedef struct task task_t;
typedef struct spinlock spinlock_t;
typedef struct semaphore sem_t;
typedef struct {
  void (*init)();
  int (*create)(task_t *task, const char *name, void (*entry)(void *arg
  void (*teardown)(task_t *task);
  void (*spin_init)(spinlock_t *lk, const char *name);
  void (*spin_lock)(spinlock_t *lk);
  void (*spin_unlock)(spinlock_t *lk);
  void (*sem_init)(sem_t *sem, const char *name, int value);
  void (*sem_wait)(sem_t *sem);
  void (*sem_signal)(sem_t *sem);
} MODULE(kmt);
```

其中有几个重要的数据结构，我们并没有定义，你需要在自己的头文件里定义它:

```
struct task; // 包含一个线程的所有资源
struct spinlock; // 互斥锁
struct semaphore; // 信号量
```

例如thread-os.c里的定义：

```
struct task {
  const char *name;
  _Context context;
  char stack[4096];
} tasks[] = ...
```

这些API实现的要求将在之后详细解释。

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 L2
```

会更新 kernel/ 目录下的一些文件，包括刚才的 kmt 模块的声明。但模块没有定义，因此你需要新建一个文件并定义它。此外，你需要在klib中实现 mem-move() 函数才能正确执行测试用的设备驱动程序。

得到一个真的操作系统

实现了这些API，我们就得到了真的操作系统吗？没错！框架代码里附带了一些“设备驱动程序”的代码，它们会调用大家实现的操作系统API，在操作系统中初始化设备、分配内存、创建线程……然后你就可以为我们的“操作系统”编程了！

为“操作系统”编程

现在，我们的操作系统还不支持进程、文件系统等，但的确有了完善的物理内存管理和线程管理API。换句话说，我们实现的是一个支持线程的“嵌入式操作系统”，它能运行在没有MMU的硬件上。

为这样的一个操作系统编写代码，就是直接在操作系统代码中静态链接一些函数，这些函数可以作为线程的入口，并且函数可以任意访问内核数据、直接以函数调用的形式调用操作系统内的API。

例如，系统中有4个tty设备，我们可以为每一个设备创建一个“echo”线程：

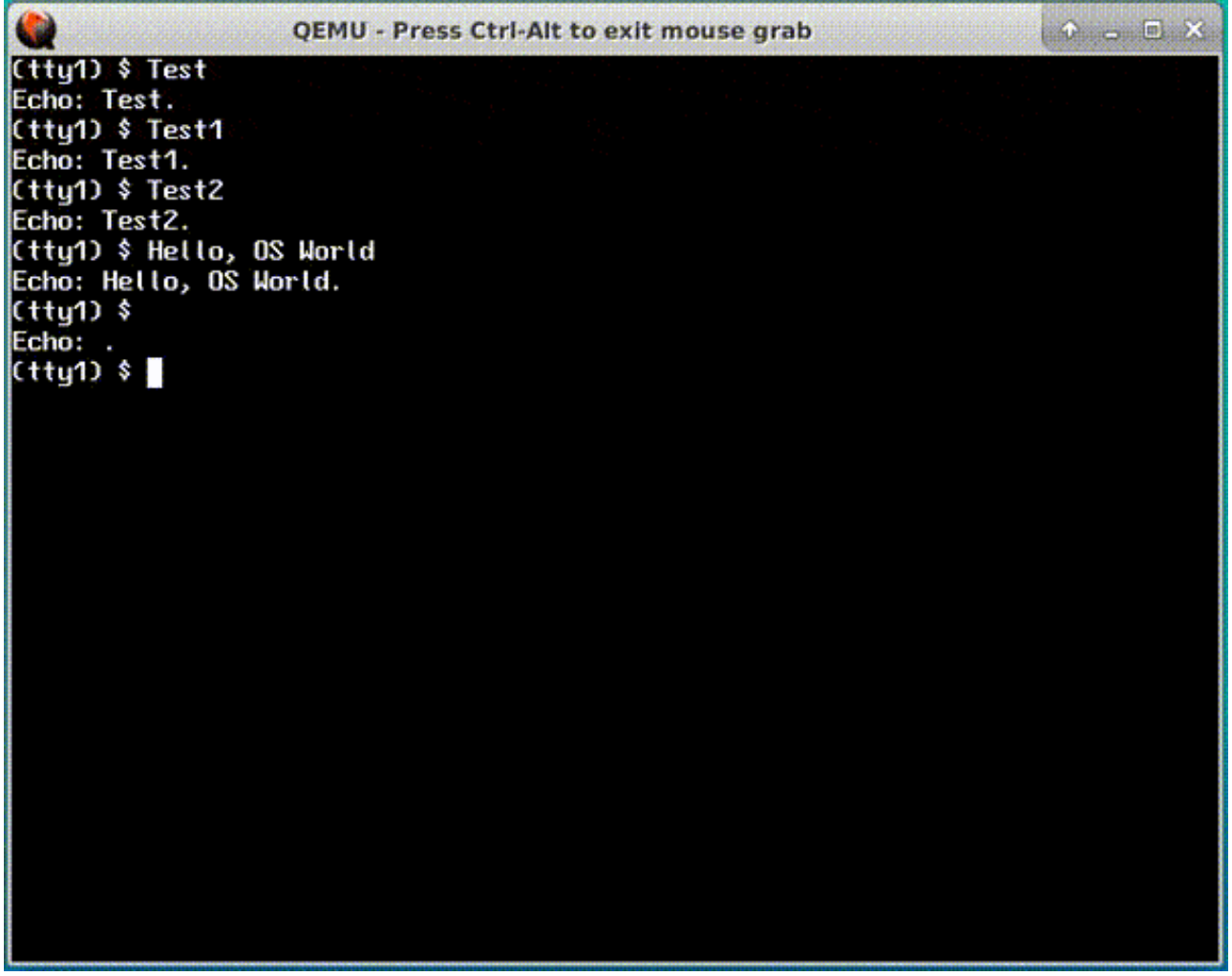
```
void os_init() {
  ...
  kmt→create(pmm→alloc(sizeof(task_t)), "print", echo_task, "t
  kmt→create(pmm→alloc(sizeof(task_t)), "print", echo_task, "t
  kmt→create(pmm→alloc(sizeof(task_t)), "print", echo_task, "t
  kmt→create(pmm→alloc(sizeof(task_t)), "print", echo_task, "t
}
```

每个echo线程就是一个“迷你”的shell，它从tty中读取输入，然后翻译成操作系统内API的序列。在下面这个简单的实现里，我们并不实现shell的功能，而是在读取到输入之后把它回显到屏幕上(你能做到这一点，也就可以实现shell了，不是吗？)：

```
void echo_task(void *name) {
  device_t *tty = dev_lookup(name);
  while (1) {
    char line[128], text[128];
    sprintf(text, "(%s) $ ", name); tty_write(tty, text);
    int nread = tty→ops→read(tty, 0, line, sizeof(line));
    line[nread - 1] = '\0';
    sprintf(text, "Echo: %s.\n", line); tty_write(tty, text);
  }
}
```

上面这段屏幕录像中，键盘按键后，中断处理程序将会唤醒tty对应的线程，tty线程再通过终端驱动程序、显示驱动程序，在VGA控制器上绘制字符。`echo_task`通过调用tty驱动的`read()`函数，在没有数据时阻塞，在读取到数据后执行相应的处理代码。这个流程和真正操作系统中发生的事情是十分类似的——你看，操作系统的确就是个C程序。

因此，如果你实现得一切正确，你就已经获得了“操作系统”的体验：



甚至对于后面的实验来说，你不过是在编写一些功能性的代码而已，“操作系统”所有的神秘感都已经不再存在了。

运行框架代码

想要获得上图的效果，需要pmm, kmt模块全部实现正确，然后初始化dev模块(调用 dev→init())，一个示例 os_init 代码：

```
static void os_init() {
  pmm→init();
  kmt→init();
  dev→init();
  // 创建你的线程，线程可以调用`tty→ops→read`或`tty→ops→write`
}
```

dev模块支持以下设备：

```
#define DEVICES(_) \
  _(0, rd_t,    "ramdisk0", 1, &rd_ops) \
  _(1, rd_t,    "ramdisk1", 2, &rd_ops) \
  _(2, input_t, "input",    1, &input_ops) \
  _(3, fb_t,    "fb",       1, &fb_ops) \
  _(4, tty_t,   "tty1",     1, &tty_ops) \
  _(5, tty_t,   "tty2",     2, &tty_ops) \
  _(6, tty_t,   "tty3",     3, &tty_ops) \
  _(7, tty_t,   "tty4",     4, &tty_ops)
```

其中有两个ramdisk、一个输入设备、一个fb、若干虚拟终端。dev模块会创建两个线程：

```
kmt→create(pmm→alloc(sizeof(task_t)), "input-task", input_task
kmt→create(pmm→alloc(sizeof(task_t)), "tty-task", tty_task, NU
```

其中会用到信号量和自旋锁，因此请确保它们实现正确:)

Don't Panic

你的代码如果有微小的bug，可能无法正确运行设备驱动程序。不要紧，我们有很简单温和的测试用例，实现正确即可得分。

API解析

OS (operating systems)模块

os 模块是操作系统主循环的代码，主要负责系统的初始化和中断响应。

- os→init() 会在系统启动时被启动的第一个处理器调用，负责完成系统的初始化；
- os→run() 是操作系统的主循环。我们假设 os→run() 时已经完成了所有初始化工作，每个处理器都会调用同一份 os→run 。
- os→trap(ev, context)：中断/异常处理程序的唯一入口。中断后，AbstractMachine完成寄存器现场保存后，就会调用 os→trap() ，并且在函数返回后，将 os→trap() 返回的寄存器现场恢复到CPU上。
- os→on_irq(seq, event, handler)：注册一个在中断时调用的callback，我们马上就解释。

理解 os 模块，要从程序执行的入口开始。以下是上一个实验中的框架代码（ framework/main.c)。在这个实验中，我们不改变这段代码:

```
#include <kernel.h>
#include <klib.h>

int main() {
  _ioe_init();
  _cte_init(os→trap);

  // call sequential init code
  os→init();
  _mpe_init(os→run); // all cores call os→run()
  return 1;
}
```

直到 _mpe_init() 之前，都只有一个处理器在执行(CPU0，如果此时调用 _cpu() 会返回 0)。_cte_init(os→trap) 指定了 os→trap 是唯一的中断处理程序，所有的处理器发生中断都会调用 os→trap() 。

小心多处理器中断

这让 os→trap() 变成了被并行执行的代码。因此，如果里面有任何共享变量，你需要用自旋锁保护好——还记得“可见性”的例子吗？

调用 _mpe_init() 之后，所有的处理器都开始执行 os→run()，操作系统正式启动。通常，os→run() 的工作就是打开中断然后死循环，你不必修改 os→run() 的代码:

```
static void os_run() {
  hello();
  _intr_write(1);
  while (1) {
    _yield();
  }
}
```

我们如何测试你的代码?

你提交的代码需要确保**已经删除了你所有的测试代码**，没有任何多余的输出:你提交的代码就好像是一个“什么也没有”的死循环操作系统内核，但中断不断到来，你的操作系统在每次中断到来后都调度idle线程用死循环消耗处理器。

我们的测试代码会替换 main 的实现，在调用 os→run() 之前创建若干线程，完成测试。因此**请不要修改 framework/ 下的任何代码**。大约，我们会把 main 函数替换成:

```
static void producer(void *arg) { ... }
static void consumer(void *arg) { ... }
static void create_threads() {
  ...
  kmt→create(pmm→alloc(sizeof(task_t)),
             "test-thread-1", producer, xxx);
  kmt→create(pmm→alloc(sizeof(task_t)),
             "test-thread-2", consumer, yyy);
  ...
}
int main() {
  _ioe_init();
  _cte_init(os→trap);
  os→init();
  create_threads();
  _mpe_init(os→run); // all cores call os→run()
  return 1;
}
```

我们会观察操作系统是否正确实现了producer/consumer。你的自旋锁/信号量等可能有bug，但我们有很多足够简单的测试用例(类似thread-os.c里打印字符)，通过就能得到很多分数；此外自旋锁/信号量也有简单和复杂的测试用例。此外，我们在测试时会链接一个我们实现正确的klib，所以大家即便klib有bug，也不必太过担心。

最后，如果你忘记注释掉框架代码(例如tty)，也没有关系。我们只增加我们的workloads (例如生产者-消费者问题)。

os模块的另一个重要功能是管理系统中的中断。我们已经知道，中断发生后会调用 `os→trap()`，这是你需要实现的。`os→trap()` 中另外需要完成的一个功能是执行 `os→on_irq` 注册的回调函数。

```
typedef _Context *(*handler_t)(_Event, _Context *);
typedef struct {
  ...
  void (*on_irq)(int seq, int event, handler_t handler);
} MODULE(os);
```

`on_irq(seq, event, handler)` 的含义是，在 `os→trap(_Event ev, _Context *ctx)` 执行时，当 `ev.event`（事件编号）和 `event` 匹配时，调用 `handler(event, ctx);`。其中：

- `seq` 决定了handler被调用的顺序，如果两个handler将会先调用 `seq` 小的。`seq` 相同的按任意顺序调用即可；
- 当 `event == _EVENT_NULL` 时，会在任何中断/异常时调用 `handler`；
- 我们允许handler返回一个 `_Context`，在中断返回时恢复到这个conetxt。

这个设计的好处是使得 os 模块就变得非常简单，它甚至完全不需要知道其他任何模块的存在，即便实现分时多线程，也不再需要修改os模块。在参考代码(JYY的实现，你们看不到啦！)中， os→trap() 是这样实现的：

```
static _Context *os_trap(_Event ev, _Context *context) {
  _Context *ret = NULL;
  for_each (handler in handlers) {
    if (handler→event == _EVENT_NULL || handler→event == ev.event) {
      _Context *next = handler→handler(ev, context);
      if (next) ret = next;
    }
  }
  return ret;
}
```

借助 os→on_irq ，我们可以“注册”若干中断处理程序，在适当的时机做适当的事情——这类似于“面向切面编程” (Aspected-Oriented Programming, AOP (https://en.wikipedia.org/wiki/Aspect-oriented_programming))的设计。例如，我们的参考代码中寄存器现场保存、寄存器现场恢复、中断管理都是通过 on_irq 实现的：

```
// thread.c, 线程管理
void kmt_init() {
  ...
  os→on_irq(INT_MIN, _EVENT_NULL, kmt_context_save); // 总是最先调用
  os→on_irq(INT_MAX, _EVENT_NULL, kmt_context_switch); // 总是最后调用
  ...
}

// input.c, 输入设备

static _Context *input_notify(_Event ev, _Context *context) {
  kmt→sem_signal(&sem_kbdirq); // 在IO设备中断到来时，执行V操作唤醒一个线程
  return NULL;
}

void input_init() {
  ...
  os→on_irq(0, _EVENT_IRQ_IODEV, input_notify); // 并不在意何时调用
  ...
}
```

大家在实验时，不必严格遵守 on_irq 的约定，只要能在 input_init() 中的 on_irq 能正确在IO设备中断到来时调用 input_notify() 即可。

KMT (kernel multithreading)模块

`kmt→init()` 负责初始化必要的数据，例如分配一些重要的数据结构。我们预期你会在 `os→init()` 时调用 `kmt→init()` 。在我们的参考实现中， `os_init()` 也是相当简单的，供大家参考(有些部分暂时不会用到):

```
static void os_init() {
  pmm→init(); // 初始化物理内存分配
  kmt→init(); // 初始化内核多线程
  _vme_init(pmm→alloc, pmm→free); // 虚拟存储，仅在CPU0执行，不过你不需要
  dev→init(); // 初始化设备
  vfs→init(); // 初始化虚拟文件系统
}
```

除此之外，KMT模块的API分为几组

线程管理

```
// KMT 线程管理
int (*create)(task_t *task, const char *name, void (*entry)(void *arg)
void (*teardown)(task_t *task);
```

其中 `create` 在系统中创建一个线程(`task_t` 应当事先被分配好)，这个线程立即就可以被调度执行(但调用create时中断可能处于关闭状态，在打开中断后它才获得被调度执行的权利)。

`teardown` 相应回收为线程分配的资源——`task_t` 中部分内存可能是动态分配的:

```
int kmt_create(task_t *task, const char *name, void (*entry)(void *arg
  ...
  task→stack = pmm→alloc(STACK_SIZE); // 动态分配内核栈
  ...
}
```

这部分的内存需要在 `teardown()` 时被回收。

在任意时刻，操作系统中都可能有多个线程，你需要设计调度的策略在多个处理器中调度这些线程，使系统中能够被执行的线程尽可能不发生饥饿。

自旋锁

```
// KMT 自旋锁
void (*spin_init)(spinlock_t *lk, const char *name);
void (*spin_lock)(spinlock_t *lk);
void (*spin_unlock)(spinlock_t *lk);
```

lock-unlock实现保护一段强原子性(任何其他线程、中断处理程序、其他处理器都不能同时得到同一把锁):

- 允许在中断处理程序中调用自旋锁。
- 允许任意在任意处理器的任意线程中调用自旋锁。
- `spin_lock` 将会关闭处理器的中断，因此对一个处理器而言，持有任何一个自旋锁之后就不会再发生线程切换。
- `spin_unlock` 在解除最后一个当前处理器持有的自旋锁之后，需要将处理器的中断状态恢复。例如在中断处理程序中，中断是关闭的，因此 `spin_unlock` 不应该打开中断；但在一般的线程中， `spin_unlock` 后应当恢复处理器的中断。

信号量

```
// KMT 信号量
void (*sem_init)(sem_t *sem, const char *name, int value);
void (*sem_wait)(sem_t *sem);
void (*sem_signal)(sem_t *sem);
```

在信号量初始化时， `value` 指定了它初始的数值。初始时 `value == 1` 可以把信号量当互斥锁；初始时 `value == 0` 可以把信号量作为生产者-消费者缓冲区管理实现。 `sem_wait` 和 `sem_post` 分别对应了P/V操作。

- 允许在线程中执行信号量的 `sem_wait` 操作。在P操作执行没有相应资源时，线程将被阻塞(不再被调度执行)。中断没有对应的线程、不能阻塞，因此不能在中断时调用 `sem_wait` 。
- 允许在任意状态下任意执行 `sem_signal` ，包括任何处理器中的任何线程和任何处理器的任何中断。

在信号量实现时，大约需要做以下几件事(任何一本操作系统教材上都会提到类似的实现):

```
void sem_wait(sem_t *sem) {
  spin_lock(&sem→lock); // 获得自旋锁
  sem→count--; // 自旋锁保证原子性
  if ( ... ) {
    // 没有资源，需要等待
    ...
    mark_as_not_runnable(current); // 当前线程不能再执行
    _yield(); // 引发一次上下文切换
  }
  spin_unlock(&sem→unlock);
}
```

但这里有个很有意思的问题——在持有自旋锁的前提下 `int $0x80` 是非常危险的——那该怎么解决呢？当然是靠你的聪明才智啦！

额外注意事项

我们的测试用例

我们会用线程API创建若干线程，并且使用信号量、锁完成各种同步任务。这部分代码你是看不见的。我们至多会创建16个线程，请保证你的操作系统支持16个以上的线程，测试线程永远不会退出。

比较重要的是，你的代码应当在多处理器、中断都存在的情况下保持正确，具体来说，你需要小心地理解以下需求是为什么：

保证多处理器/中断上的正确性

- 所有的代码都可以在多个处理器上被**同时**调用。因此你需要小心地保证原子性、顺序、可见性。万小心：`kmt→create()`，`kmt→sem_signal()` 等所有函数都可能同时在多个处理器上被调用。你现在觉得这句话很可笑，但你调过bug就知道厉害了。
- 在中断处理程序中可以调用自旋锁。实际上，一个CPU的中断处理程序可以和另一个CPU访问同一个共享数据结构(例如在中断中向链表里插入一个元素，这个链表被另一个线程读取)，因此自旋锁是保证正确性的重要手段。

实验指南

坐下来，读一读代码

不知道实验要求在说什么?

不妨阅读一下代码。操作系统内核是从 `main` 函数开始执行的，然后完成一些初始化任务，之后启动多处理器。每个处理器再完成per-cpu的初始化之后，会进入“等待中断”的循环——这就是操作系统。

带着这个理解再去看每个API，就不会觉得太困难了。

以及，其实课堂上已经讲解过了操作系统内核的工作原理：

- thread-os.c (/static/wiki/os/2019/demos/thread-os.c)：一个分时调度固定数量线程、round-robin调度的嵌入式操作系统；
- thread-os-mp.c (/static/wiki/os/2019/demos/thread-os-mp.c)：把刚才的代码扩展到多处理器。

大家不妨在自己的机器上运行一下(你需要编写一个Makefile，用AM编译它)。以thread-os为例，比较关键的代码是它用

```
_cte_init(interrupt);
```

注册了一个callback function，在每次中断/异常时调用 `interrupt()` 函数执行，而中断处理的流程也非常简单：

```
_Context *interrupt(_Event ev, _Context *ctx) {
  if (current) current→context = *ctx; // 把当前运行程序的寄存器现场保存到c

  // "调度"下一个执行的线程
  if (!current || current + 1 == &tasks[LENGTH(tasks)]) {
    current = &tasks[0]; // back to the first task
  } else {
    current++;
  }

  // 把current中保存的寄存器现场恢复到处理器上执行
  return &current→context;
}
```

在多处理器下，你会遇到一些额外的问题，但总体来说原理是相似的——实现操作系统没啥困难的，可以立即就动手啦！

其实我在骗你!

你们在编程的时候，会遇到无数诡异的bug，跑着跑着虚拟机就挂了，而且错得非常离谱，完全不知道发生了什么。对于并发程序，更难受的是可能要运行很多次，或者在某个特定的条件下bug才会触发。所以继续读下去吧。

调试操作系统内核

先回顾一下课上的内容

课上已经介绍过一些调试操作系统内核 (OS2019_C7.slides)的经验，没上过课的同学，只能说你错过了一些很重要的内容。

很快写着写着你就会发现自己的代码出bug了，有可能是并发的，有可能只是顺序的逻辑实现错了。如果出现了莫名其妙的异常、虚拟机神秘重启等情况不要惊慌，**机器永远是对的**，坐下来调代码吧。

要想快速知道代码出了什么问题，实现 `printf` 和 `assert` 就是非常重要的：

```
#ifdef NDEBUG
  #define assert(ignore) ((void)0)
#else
  #define assert(cond) \
    do { \
      if (!(cond)) { \
        printf("Assertion fail at %s:%d\n", __FILE__, __LINE__); \
        _halt(1); \
      } \
    } while (0)
#endif
```

以及 panic 能帮你及时检测操作系统中不预期出现的状态:

```
void *ptr = pmm→alloc(size);
if (!ptr) {
  panic("memory allocation failed");
}
```

在适当的地方加上printf和assert能帮助你快速定位到程序中出错的状态(比如在中断到来时打印寄存器的现场，能快速帮你定位出现异常的位置)，缩小bug的检查范围。

更好的 printf 方式

相信大家都有过不停地加 printf，删掉 printf 的调试体验吧。这感觉可不太好，找了一大圈，最后发现一开始删掉的 printf 打印的信息才是最有用的，不过一整天就已经过去了。

下面的记录模式也许能帮到你:

```
#define TRACEME

// #include <trace.h>
#ifdef TRACEME
  #define TRACE_ENTRY printf("[trace] %s:entry\n", __func__)
  #define TRACE_EXIT printf("[trace] %s:exit\n", __func__)
#else
  #define TRACE_ENTRY ((void)0)
  #define TRACE_EXIT ((void)0)
#endif

void f() {
  TRACE_ENTRY;
  printf("This is f.\n");
  TRACE_EXIT;
}
```

可以在重要的函数上面加上这些日志，通过 TRACEME 宏来决定到底是否打印日志。

如果各种 printf 都没办法帮你找到问题，那就只能上gdb了。框架代码里故意没有提供 make debug 的选项，但这是做得到的，只需要用 qemu 提供的命令行选项。

- -gdb 能启动调试模式。
- -S 能让虚拟机在收到调试命令前不执行执行。
- 在 gdb 中可以用 target 连接远程调试。
- 用 .gdbinit 能把连接、加载调试信息、断点设置等一并自动化，这样 make debug 就能实现一键调试。

> **拖延症?**
>
> 你在读完上面那些文字的时候，也许拖延症已经犯了：这么麻烦，关我×事。对中枪的同学：当你实在找不到bug的时候，回来求gdb也许能帮你挽回几天的调试时间。这时候你就真正体会到**基础设施的重要性**了。

总的来说，如果只是写几十上百行的OJ程序，无论你习惯多么的坏，多多少少总是能调试出来的。但如果要维护更大的程序，用上正确的工具就能帮上大忙了。

保护自己不受伤害

有些bug可能会非常难调试，这里举一个例子：栈溢出。也许已经有同学在Lab 0的时候吃过苦头了：栈空间不是无限大的，而如果我们带着很大的局部变量或者递归很多层，栈就悄悄溢出了……在操作系统内核中，内核线程栈的溢出就显得更危险了，因为你可能会定义：

```
struct thread_t {
  int32_t id;
  ...
  uint8_t stack[STK_SZ];
};
```

而栈溢出(x86的栈从高地址向低地址生长)的后果就是线程的信息可能被覆盖，出现各种诡异(难以理解)的情况，而且有可能bug若隐若现，加一条 printf 也许就不触发了。

如果想避免这种情况的发生，可以给栈的前后加一些栅栏缓冲(fences)：

```
struct task_t {
  int32_t id;
  ...
  uint8_t fence1[32];
  uint8_t stack[STK_SZ];
  uint8_t fence2[32];
};
```

然后给这些fences赋上一些“magic numbers”，比如 `0×cc` 。随时我们检查这些fences是否被修改过(刚好全部写入相同数值的可能性几乎为0)，就能知道有没有栈溢出(under/overflow)了。

烫烫烫和屯屯屯的故事

mscc在Debug模式编译时，未初始化的栈内存会填入 `0×cc` ，未初始化的堆区填入 `0×cd` ，堆内存的fences填入 `0×fd` ，被free的内存填入 `0×dd` 。这就是大家为什么会看到烫烫烫和屯屯屯了——刚好这些内存被解码成了中文的字符。

以下笑话来自互联网：

手持两把锟斤拷，
口中疾呼烫烫烫。
脚踏千朵屯屯屯，
笑看万物锘锘锘。

同理，我们可以在kalloc/free的时候做一些hacking：可以悄悄多分配一些内存作为fences，这样可以帮助你检查出很多类型的bugs，例如你可以在一些关键的pointer deference的地方插入

```
*ptr ... // important pointer, e.g., current
if (0×cccccccc == *(uint32_t *)ptr) {
  panic("dereference a freed memory");
}
```

最后，这几节中介绍的知识都是**完全没有用**的——如果你不写够大的代码，做够系统的项目，你永远都不会用到这里的知识，永远都不会想把C语言的能力发挥到极致。这里的知识对考试也几乎完全没有任何帮助。但正是在种种细节上追求完美才成就了大型软件系统的成功。在阅读优雅的代码时总有赏心悦目的快乐。关于操作系统内核，xv6 (http://pdos.csail.mit.edu/6.828/xv6)绝对是一份了不起的佳作，看似平淡无奇的代码里参透着系统设计的智慧，值得大家品位。