FAQ

- barrier和条件变量等是实现在用户态的吗?
- mbox_send和mbox_recv需要返回什么值?
- 头文件明明include了，但还是报错undefined reference?
- qemu好像未能正确启动双核?
- 多核哪些是共享的？哪些是独立的?
- 如何在另一个处理器上跑进程?
- 主从核分别需要做哪些工作?
- 如何从内核输出到screen上?
- 多核后输出混乱怎么办?
- sbi_send_ipi给主核自己也发了中断怎么办?
- 我主核发送ipi以后，从核收不到有什么可能的原因?
- 多核情况下，如果kill的进程正在另一个核上运行怎么办?
- 多核有莫名其妙的错误，可能有什么原因?
- 如何获取当前核的id?
- exit和kill有什么区别?
- 我们的锁是否是用户态的？是否需要内核释放?
- C-CORE的任务需要几个mailbox?
- mailbox服务端比客户端多14个byte是什么原因?
- zombie进程应该什么时候清理掉?
- test_semaphore.c中semaphore_add_task1函数是否需要等待另外两个进程结束再销毁信号量?
- 双核如果只有一个进程应该怎么调度?
- 测试代码尝试输出到0,0位置可能导致出错?
- 如何实现退格的功能?

barrier和条件变量等是实现在用户态的吗?

tiny_libc下面的所有的东西都是用户态的。但barrier和条件变量的主要逻辑并不要求实现在用户态。换句话说，barrier和条件变量的逻辑建议按照前一个实验中的锁的设计模式实现。用户态只保留锁的id。内核态真正负责管理的结构体。

总之，和前面的实验一样，这里需要注意：不要将等待队列等内核数据结构暴露给用户态的进程。这一点是绝对不允许的。

mbox_send和mbox_recv需要返回什么值?

这两个函数的返回值代表mbox_send或mbox_recv的执行的过程中被阻塞了多少次。比如如果mbox_send因为邮箱满了被阻塞了一次，返回值就是1。这个阻塞次数具体和你的设计有关。有的设计中可能最大就可能被阻塞1次，要么就没被阻塞。也有些设计可能存在被唤醒后邮箱仍是满的的状态，再次被阻塞，那么就返回值就可能大于1。

头文件明明include了，但还是报错undefined reference?

首先强调一个点：函数是分为声明和定义两部分的。声明就是说明一下函数的名称、参数类型、返回值类型等。例如头文件中常见的：

```
int func_name(int a, long b);
// 或者全局变量
extern int ddd;
```

定义是指函数真正的实现，一般都在.c文件中：

```
int func_name(int a, long b)
{
    return a + (int)b;
}
```

C语言的编译流程是：以每个.c文件作为一个编译单元，编译器会先对.c文件的内容做预处理（也就是把include的文件直接嵌入进去，进行宏替换等等这类的），然后再编译出一个目标文件。但如果你在.c文件中调用了其他C文件中实现的函数，编译器此时无法生成真正的调用（因为不在一个文件里，不知道那个函数具体地址是啥）。等到最后所有的.c都编译完了，编译器会把编译出来的目标文件链接在一起。在链接的时候就会真正填入那些调用其他C文件中的函数的地址（因为链接的时候所有的目标文件就都已经生成了，链接器完全能够知道每个函数在哪个地址上）。

undefined reference是链接器在链接的过程中找不到某个函数/全局变量的定义，也就是说，它在你编译完的那堆.c文件中没有找到相应的符号。链接器的工作原理是，ELF文件中有特殊的section记录待填写的地址及其对应的符号。每链接一个库，链接器就会看当前有哪些符号是未定义的，然后试图从库中把相应符号的地址找出来填进去。

因此，undefined reference可能有两种情况：一种原因是，连接如果后面的库用到了前面的库的符号，那么链接器是不会再次跑去找前面的库里面的符号的（我们上面说过链接器是一个一个库依次往上链接的）。不过我们这个实验应该不涉及这种情况。另一种原因是，你的含有函数定义的.c文件没有加入到编译命令里面。因为本次实验是需要大家自己改写Makefile的，所以很可能是写的时候漏写了某个.c。或者是因为某种原因.c没有真正被编译到最后的可执行文件里。这个大家可以根据make输出的编译命令判断，看看含有函数定义的那个C文件到底有没有出现在编译命令中。

qemu好像未能正确启动双核?

请确认QEMU的参数设置中存在`-smp 2`

多核哪些是共享的？哪些是独立的?

多核处理器中，内存是共享的，每个核心上的代码访问到的都是同一块内存。这也就意味着，两个处理器其实执行的是同一套代码，比如在跳到0x50200000的时候，两个核心都在执行你的bootblock的代码。但两个处理器核心都有自己的一套寄存器。比如特权寄存器和通用寄存器，都各有一套。在进入bootblock的时候，a0寄存器会存放处理器的id号。核心的id号为从0开始编号一直到最大核心数。比如在我们的平台上，主核是0,从核是1。根据id号自行在代码中判断当前是运行在主核还是从核上的，然后根据自己的设计分别执行对应的操作。

注意，从核的时钟中断也是独立的，需要在从核上重新设置。

如何在另一个处理器上跑进程?

在做多核的时候，大家最经常想的问题是，如何在另一个处理器上运行进程。答案是利用中断。中断会触发处理器强制跳转到stvec寄存器处。因此，只要能够触发从核的中断，就可以让从核执行中断处理的代码。此时，代码已经是在从核上执行的了。接下来，你就可以让你的代码来调度起一个进程执行。这样，进程自然执行在了从核上面。

主从核分别需要做哪些工作?

以bootblock为例，bootblock原本的功能是从SD卡将内核搬运到内存中。当双核同时启动时，如果两个核全部去从SD卡搬运一遍数据，显然是不对的。因为内存是共享的，只需要其中一个核来把数据都搬进来就可以了，另一个核心同时搬运数据可能会造成一些冲突。因此，这里只需要主核搬运数据，从核需要等待主核搬完然后才能再执行。

同理，我们需要初始化中断、pcb、系统调用表等一系列动作，这些动作都只需要做一次即可。因此，这些都让主核来完成即可。从核只需要等主核把所有的事情都做完然后再进行必要的初始化即可。从核最主要的是需要设置自己的中断处理函数入口，之后设置上时钟中断就可以调度了。

大内核锁是什么？都需要在什么地方加？

由于主从核共享内存，所以显然两个核心同时处理一些全局变量的时候会发生多线程相关的一些问题。为了避免这些问题，我们需要加锁。为了实现方便，我们建议将整个内核全部锁起来。也就是你进入内核的时候直接加锁。出内核的时候再解锁。这样内核整个就全被锁保护起来了，内核的执行过程不会被另外一个核打断。这种方式使得我们在单核上的最重要的假设成立了：内核的执行过程不会被打断。在单核上，我们的很多实现依赖于内核不会被时钟中断打断这一特性。在多核上，虽然不会被时钟中断打断，但是有可能会被另一个核执行的内核代码扰乱。但在加了大内核锁以后，一切就和单核上很接近了。

大内核锁需要在进内核态和出内核态的时候进行加锁和解锁。进内核态包括启动的时候，还有例外处理的入口这两种情况。出内核态只有例外返回这一种情况。其中启动的时候是否需要加锁请根据自己的设计自行考虑。不同的设计不一样。

如何从内核输出到screen上？

printk是直接输出到串口的，但我们可能有时候想和用户态一样，只是输出到screen中。为了满足这种需求，可以单独制作一个prints函数，供内核调用，但是最后输出的时候调用`screen_write`输出到屏幕缓冲区，而不是输出到串口。

多核后输出混乱怎么办？

如果发生输出混乱，请注意阅读screen.c。这是由于screen.c中，有一个共享的变量screen_cursor_x和screen_cursor_y。这个变量在单核的时候，我们是在切换进程的时候把它保存到current_running的cursor_x和cursor_y中的。通过这种方式我们维护了一个每个进程都独占屏幕的假象。但是在多核的情况下，可能会有这种情形发生：两个核都没有进行进程切换，但两个和都通过系统调用先后进入到内核，分别改变了screen_cursor_x和screen_cursor_y后退出内核。由于没有进程切换，所以这两个进程修改后的screen_cursor_x和screen_cursor_y都没有被保存下来。

为了避免这一情况的发生，我们建议直接去掉这两个全局变量，全部用当前核正在运行的进程的pcb中的cursor_x和cursor_y代替。唯一需要注意的一点是：screen_reflush会调用vt100_move_cursor改变物理坐标从而将光标移动到屏幕上发生变化的字符处，再输出新的字符。但该函数会顺手把cursor_x和cursor_y改掉。但screen_reflush中其实并不想真的改变用户的cursor。所以这里可能需要在函数的开头和结尾保存/恢复cursor_x和cursor_y的值。

最后，prints的逻辑有一部分和printk公用了。为了printk的cursor和screen中的逻辑cursor同步，我们在printk中模仿screen_write_ch中的逻辑做了一些维护动作。但prints中其实并不需要这些维护动作，因为它最后就是调用screen_write_ch进行输出的。在做了前面所述的修改后，这个逻辑问题会暴露出来。所以这里还需要让prints不做这些多余的维护动作（具体怎么实现就交给你设计了，这个不难）。

sbi_send_ipi给主核自己也发了中断怎么办？

处理器核所有收到的中断都会导致sip寄存器的对应位被置1。当中断屏蔽的情况下，处理器是不会产生中断的。当中断开启的状态下，只要sip中有某些位被置1的话，会自动产生中断。包括时钟中断在内，处理器是否产生中断全看sip寄存器以及中断是否屏蔽。sip寄存器中的位其实就代表了已经到来但还未处理的中断。

因此，如果主核收到了ipi，但并不想处理，那么可以直接在中断屏蔽的状态下给sip寄存器写0，从而把未处理的中断清理掉。这样中断开启后，自然就不会发生任何中断了。

我主核发送ipi以后，从核收不到有什么可能的原因？

请检查从核的sie寄存器和sstatus的sie位是否开启。只有当这两个全部打开的情况下，才能发生中断。

多核情况下，如果kill的进程正在另一个核上运行怎么办？

多核kill只要求，sys_kill返回的时候另一个进程确实已经被kill掉了，但不一定非要求这边一发kill命令那边立马kill。也就是说，如果另一个进程目前处于running状态，可以在pcb中先标记一下，然后阻塞发出kill指令的进程。等到另一个核时钟中断到了，检查到它已经被kill了，再执行kill动作，之后将发出kill指令的进程重新加入到ready 队列中调度。

或者，更简化一些的版本也是可以接受的，就是标记另一个核上正在运行的进程被杀掉了。然后直接返回。等另一个核时钟中断时再真正执行kill动作。

当然，如果有兴趣，也可以利用核间中断来实现实时的kill命令。但由于大内核锁设计的一些限制，实现起来也许会有一些难度

多核有莫名其妙的错误，可能有什么原因？

目前看到调出多核的同学有这么几个比较离奇的点：

(1) 有的同学把enable_preempt和disable_preempt都去掉就能正常跑通多核。这两个函数在我们当前的实验中暂时没有什么作用，可以安全地去除掉（简单的去除方法是一进入这俩函数就直接jr ra返回）。

(2) 有的同学是输出太慢，把screen.c中的screen_cursor_x/y替换成pcb中的cursor_x/y，也就是实时保存cursor的位置就能跑对。（这里注意screen_reflush中调用vt100改cursor的位置并不是用户想改的，所以那里可能需要提前保存一下用户原先的光标，后面退出函数前再恢复）

如何获取当前核的id？

bootblock.S被加载时，a0中存放了当前的mhartid。这个id代表当前的核心号。在我们的实验中可以假定这个核心号就是0或1，0是主核，1是从核。

start code中也提供了一个`get_current_cpu_id`函数，可以用于获取当前核心的id。这个函数中通过读取mhartid寄存器来获取当前核心id。理论上mhartid只能在m态读取，但是我们的实验为了简化，对处理器和QEMU都做了修改，允许大家在s态读取该寄存器。

```
ENTRY(get_current_cpu_id)
  csrr a0, CSR_MHARTID
  jr ra
ENDPROC(get_current_cpu_id)
```

exit和kill有什么区别？

exit基本上就相当于kill了自己。这两者的核心实现甚至可以提取到一个函数里面。但对于exit来说，有个很麻烦的问题是如何释放自己的内核栈。因为`do_exit`是运行在自己的内核栈上的，所以没法在`do_exit`的时候释放自己的内核栈。否则从逻辑上讲，释放了一段正在使用的内存空间是很容易出问题的（虽然在我们的实验中，因为释放分配内存不频繁，出错概率不高）。

一般来讲，处理上述问题有三种方法：

- 由父进程负责释放：一般的类Unix设计的系统中，进程结束后会进入一个zombie态。父进程在wait/waitpid的时候会将它的内核栈释放掉。因为此时是跑在父进程的内核栈上的，所以可以安全

释放已经退出的子进程的内核栈。

- 标记释放：可以先做一个标记。在下次时钟中断时或者其他合适的时机释放。总之，找一个别的进程陷入到内核的时机，由于此时的内核栈使用的已经是别的进程的内核栈了，所以可以安全进行释放。
- 与PCB一起回收复用：也有一种取巧的方式是，在第一次分配完PCB后，内核栈/用户栈就固定地归属于相应的PCB。等到下次`do_spawn`开启新进程时，直接复用之前的PCB和内核栈/用户栈，而不是重新分配新的内核栈/用户栈。简单的说就是给每个PCB固定分配好内核栈/用户栈，后面只要用到该PCB就始终使用固定分配的这组用户栈内核栈，不再重新进行释放/分配一类的动作。

我们的锁是否是用户态的？是否需要内核释放？

我们的锁不是所谓的用户态锁，验收时需要演示内核释放锁的功能。

讲义的某些段落我们会再做修正。需要释放锁的原因和今年的框架代码设计，以及P2实验的实现方式有关。最早设计框架的时候，因为种种原因所以锁的设计有点像pthread的mutex那种锁。在往年的教学中也曾建议过同学尝试实现类似于pthread的线程锁。所以讲义说的是对于用户态的锁内核没有办法释放。用户态的锁特指类似于pthread这种锁的状态完全由用户维护，内核只负责挂起/唤醒进程的模式(因为这种多用于线程中，所以以下简称为线程锁模式)。

线程锁的模式是：用户态的库负责维护锁没锁上这个状态（一般需要用原子操作完成）。如果锁没有锁上，就直接返回（注意，由于是用户态直接维护锁的状态，所以无需通过系统调用进入到内核）。如果发现用户态已上锁，则通过内核的某些系统调用将自己挂起。释放锁时，如果知道没有其他进程在等待，则直接用原子操作把锁的状态置为未锁定。否则，调用系统调用唤醒其他等待锁的进程。

可以看到，由于线程锁“锁没锁上”的状态是由用户维护的，内核毫不知情。内核只负责按照用户的要求挂起/唤醒进程。所以即使线程被kill了或者因为其他原因挂掉了，内核也无法把锁释放掉。因为锁的状态本身本就不是由内核管理的。

线程锁的设计的好处是，在竞争不激烈的情况下，无需频繁进入到内核态。因为代表锁没锁上的相关变量都放在了用户态，直接用原子操作维护，无需内核参与。只有当发生竞争时，才会进入到内核态挂起线程。这样的设计能够提升系统整体性能。但用原子操作维护“有无进程等待锁”这个状态的操作相对复杂(可以参考这个https://akkadia.org/drepper/futex.pdf)，所以出于简化实验的目的考虑，今年统一改为了进程锁的实现方式。

进程锁的设计参考的是Unix的XSI Semaphore的设计模式。XSI Semaphore是进程间同步所使用的信号量。XSI Semaphore的使用方式是进程通过`semget(key_t key, int nsems, int semflg)`获取信号量集合(XSI的这个信号量支持一次创建包含多个信号量的集合)的id。后面用户要执行增加/减少信号量的操作时，可以以id作为参数调用`semop`，实现对信号量的增/减（当然，如果尝试减少信号量时，如果信号量已经等于0了就会阻塞）。XSI Semaphore由于是内核管理的，所以提供undo功能。也就是如果持有信号量的进程退出，它所持有的信号量都会被释放。

在我们今年的实验中，我们的锁实现都是锁的对象本身在内核中，用户只是通过内核提供的接口申请锁的相关服务。用户并不负责锁对象本身的管理。回忆一下，按照今年建议的设计方案，用户得到的只是锁这个资源的id。实际上锁的对象本身是由内核管理的。用户只是申请了这种资源的使用。因此，如果持有这种资源的用户程序因为某种原因退出，那么负责管理资源的内核当然有义务将用户持有的资源回收。而且与线程锁在用户态自行维护锁的状态不同，进程锁的状态全部是由内核维护的。内核完全有能力知道这个锁是否锁上，有多少进程在等待它等信息。综上，我们认为内核既有能力也有义务完成对于锁这一内核对象的维护。所以今年也在验收时会要求演示出这种释放锁/回收资源的能力。

C-CORE的任务需要几个mailbox?

C-CORE任务需要3个mailbox，每个进程一个。每次向随机选择的另外某一个进程的mailbox中发送随机字符串，同时收取发给自己的字符串。

mailbox服务端比客户端多14个byte是什么原因?

mailbox客户端启动后，会先向服务端发送“clientInitReq”（这个字符串连带上结束符\0共计14byte），之后服务端会在另一个邮箱里回复一个整数。客户端会收取这个整数作为打印位置的行号。后面客户端的输出都会打印在该行。

服务端会记录所有收到的字节数，所以客户端发送的clientInitReq也会被计算在内。而客户端只会记录后面随机生成的那些字符串。所以每多一个客户端，服务端收到的数据会比客户端多14个byte，这个是正常现象。

这里额外提醒一下，如果邮箱实现正确的话，客户端应该是可以启动多个的。建议大家多开几个跑久一点试试别出现卡死的情况。

zombie进程应该什么时候清理掉？

一些同学对于zombie态存在一些理解上的困惑，这里统一说明一下。我们首先假设这样一种场景，父进程创建了一个子进程，之后父进程做了一些其他的工作，然后等待子进程结束并获取子进程的返回值。

```
int child()
{
    int a = some_calculation();
    return a;
}
int parent()
{
    pid_t pid = sys_spawn(child); // 假设这里创建了子进程，子进程的入口为child函数
    // do something
    int retval = wait(pid); // 假设这个wait会等待子进程结束并返回子进程的返回值a。
    return 0;
}
```

那么等父进程调用wait的时候可能有两种情况：

- 子进程还未执行完：这是一种比较简单的情况。内核只需要把父进程挂在等待队列上，等子进程执行完以后，将子进程的返回值传给父进程即可。
- 子进程已执行完：这种情况就存在一个麻烦，父进程调用wait的时候子进程已经退出了，那么内核怎么获得子进程的返回值呢？为了解决这一问题，内核在子进程退出时，必须保留子进程的必要的信息（例如返回值、退出状态等）。这样等到父进程调用wait的时候，内核才能将相应的信息传递给父进程。

上述的第二种情况，虽然子进程已退出，但是内核仍然保留了子进程的部分信息的状态，就是所谓的zombie态。我们的实验中场景更加简化，但是原理和上面讲的例子是差不多的。所以大家可以认为，zombie进程其实就是一个已经退出的进程，只是内核为了等父进程wait所以保留了某些必要的信息。在进入zombie态时，内核可以将进程的大部分资源都释放掉，仅保留需要的信息。等待父进程wait调用结束后，再将所有资源都释放掉。

这里补充一点知识。对于Unix类的系统来说，如果子进程进入了zombie，而父进程在没有wait它的情况下就退出了，那么子进程会过继给“爷爷进程”或者“祖先进程”。如果“爷爷进程”也挂了，那么会再往上过继。直至最后，会过继给init进程（因为init是所有进程的祖先）。最终会由init调用wait来将其彻底释放掉。

test_semaphore.c中semaphore_add_task1函数是否需要等待另外两个进程结束再销毁信号量

确实需要，这里是测试程序写得有些不严谨。建议仿照barrier测试程序的模式，修改为

```
void semaphore_add_task1(void)
```

```
{
    int i;
    int print_location = 1;
    // int sum_up = 0;

    mthread_semaphore_init(&semaphore, 1);

    struct task_info subtask1 = {(uintptr_t)&semaphore_add_task2, USER_PROCESS};
    struct task_info subtask2 = {(uintptr_t)&semaphore_add_task3, USER_PROCESS};
    pid_t pids[2];
    pids[0] = sys_spawn(&subtask1, NULL, ENTER_ZOMBIE_ON_EXIT);
    pids[1] = sys_spawn(&subtask2, NULL, ENTER_ZOMBIE_ON_EXIT);

    for (i = 0; i < 10; i++)
    {
        mthread_semaphore_down(&semaphore); // semaphore.value--
        // semaphore = 0
        global_count++;
        mthread_semaphore_up(&semaphore);

        sys_move_cursor(1, print_location);
        printf("> [TASK] current global value %d. (%d)", global_count, i + 1);

        sys_sleep(1);
    }

    for (int i = 0; i < 2; ++i) {
        sys_waitpid(pids[i]);
    }

    mthread_semaphore_destroy(&semaphore);

    sys_exit();
}
```

双核如果只有一个进程应该怎么调度?

建议在没有进程调度的情况下，可以跑pid0。相当于跑内核主函数末尾处那个死循环。

测试代码尝试输出到0,0位置可能导致出错?

一些测试代码的`sys_move_cursor`会尝试把光标移动到0,0的位置，但在start code给的screen.c中：

```
/* write a char */
static void screen_write_ch(char ch)
{
    if (ch == '\n')
    {
        screen_cursor_x = 1;
        screen_cursor_y++;
    }
    else
    {
        // 这里会认为screen_cursor_x和screen_cursor_y都是从1开始的
        new_screen[(screen_cursor_y - 1) * SCREEN_WIDTH + (screen_cursor_x - 1)]
= ch;
        screen_cursor_x++;
```

```
    }
    current_running->cursor_x = screen_cursor_x;
    current_running->cursor_y = screen_cursor_y;
}
```

因此，可能会导致无法正常输出。大家可以修改测试程序，或者调整这个`screen_write_ch`。推荐方式是调整screen.c中的代码，因为添加多核支持本身就需要修改这个文件。

如何实现退格的功能？

用户程序中的退格通过输出`\b`来实现。但因为我们是输出到屏幕缓冲区的，所以需要`screen.c`中的相关代码配合。建议是修改screen.c中的`screen_write_ch`函数：

```
static void screen_write_ch(char ch)
{
    if (ch == '\n')
    {
        screen_cursor_x = 1;
        screen_cursor_y++;
    }
    else if (ch == 8 || ch == 127)
    {
        // 添加对退格的处理，把屏幕缓冲区相应的位置设置为' '即可。
        // 以下是示意代码，根据你的多核的修改方式的不同，可能有所区别。
        screen_cursor_x--;
        new_screen[(screen_cursor_y - 1) * SCREEN_WIDTH + (screen_cursor_x - 1)]
= ' ';
    }
    else
    {
        new_screen[(screen_cursor_y - 1) * SCREEN_WIDTH + (screen_cursor_x - 1)]
= ch;
        screen_cursor_x++;
    }
    current_running->cursor_x = screen_cursor_x;
    current_running->cursor_y = screen_cursor_y;
}
```