本次题面以 pdf 和 html 两种格式下发，需要复制代码的同学请使用 html 格式的题面。

lab4-1-exam

创建并切换分支

```
cd ~/学号
git checkout lab4
git add .
git commit --allow-empty -m "save my lab4"
git checkout -b lab4-1-exam
```

题目背景

借助系统调用，用户进程可以向操作系统请求多种服务，访问各类资源和外部设备。但是，内核所持有的资源也需要相应的保护机制，以避免在多进程环境中产生竞争冲突。以下介绍一种称为自旋锁的锁机制。

> 自旋锁是计算机科学用于多线程同步的一种锁，线程反复检查锁变量是否可用。由于线程在这一过程中保持执行，因此是一种忙等待。一旦获取了自旋锁，线程会一直保持该锁，直至显式释放自旋锁。
>
> 获取、释放自旋锁，实际上是读写自旋锁的存储内存或寄存器。因此这种读写操作必须是原子的。
>
> —— 自旋锁，维基百科（有删改）

（以上材料仅用于帮助理解概念，与本题的要求可能不完全一致，其中提到的“线程”对应 MOS 中的进程）

锁机制的实现首先需要保证多进程并发的环境下获取锁和释放锁操作不会产生冲突，而系统调用机制恰好提供了这种**原子性**。在本题中，你需要添加新的系统调用，在 MOS 系统中实现简单的自旋锁，并对控制台输出设备加锁保护。

题目描述

在本题中，内核需要面向所有进程维护一个公用的锁（下称**锁**）。对于用户程序，锁在同一时刻要么被唯一一个进程持有，要么不被任何进程持有（此时称锁处于**空闲状态**）。内核初始化时，锁应当处于空闲状态。

为了在用户空间提供自旋锁机制，你需要实现以下两个用户函数：

“检查并设置”锁

```
/* user/lib.h */
int syscall_try_acquire_console(void);
```

若锁处于空闲状态，该函数设置锁由当前进程持有，并返回 `0`；否则，该函数返回 `-1`。

请注意：只要锁不处于空闲状态，即使锁已由当前进程持有，该函数也返回 `-1`。

释放锁

```
/* user/lib.h */
int syscall_release_console(void);
```

若锁由当前进程持有，该函数设置锁为空闲状态，并返回 `0`，从而使当前进程不再持有锁。否则，该函数返回 `-1`。

- 为了实现这两个用户函数，你需要设计并添加相应的系统调用，保证这两个锁操作具有原子性，即不会被其他进程打断，导致多个进程持有锁。

控制台输出锁保护

为了在内核中控制控制台输出设备的使用，你需要修改 `lib/syscall_all.c` 中 `sys_putchar` 系统调用函数的实现，使得只有持有锁的进程才能向控制台输出字符。如果当前进程不持有锁，该函数应直接返回，**不应输出任何内容。**

实现要求

- 在 `user/lib.h` 中声明函数 `int syscall_try_acquire_console();`
- 在 `user/lib.h` 中声明函数 `int syscall_release_console();`
- 在 `user/syscall_lib.c` 中实现以上两个用户函数，发起系统调用
- 修改相关文件，在内核中添加必要的系统调用，并维护必要的信息，以供用户函数使用。可能涉及的文件有：
  - `lib/syscall_all.c`：添加系统调用在内核中的实现函数
  - `lib/syscall.S`：将实现函数添加到系统调用入口向量表
  - `include/unistd.h`：定义系统调用号
- 修改 `lib/syscall_all.c` 中的 `sys_putchar` 函数

评测时，仓库中的以下文件可能被替换为标准版本：

- `Makefile`
- `user/Makefile`
- `init/init.c`

本地测试

测试程序

```
/* user/lktest.c */
#include "lib.h"

void umain() {
    u_int me = syscall_getenvid();
    while (syscall_try_acquire_console() != 0) {
        syscall_yield();
    }
    writef("I'm %x\n", me);
    syscall_release_console();

    while(1);
}
```

样例说明

用户程序首先进行忙等待，尝试通过 `syscall_try_acquire_console()` 向内核获取自旋锁；成功获取锁后，通过 `writef()` 函数输出自身的 envid。由于当前进程已持有锁，`writef()` 函数依赖的 `syscall_putchar()` 函数能够正常工作。最后，进程通过 `syscall_release_console()` 释放锁。

由于此程序正确使用了自旋锁机制，即使同时运行在多个进程中，输出的文本也不会交错。

测试流程

将以上的用户程序保存为 `user/lktest.c`，并修改 `user/Makefile`，在构建目标 `all:` 后添加 `lktest.x` 和 `lktest.b`，最后将 `init/init.c` 中的`mips_init()`函数修改为以下内容，以在多个进程中运行测试程序：

```
void mips_init() {
    printf("init.c:\tmips_init() is called\n");
    mips_detect_memory();
    mips_vm_init();
    page_init();
    env_init();

    int i;
    for (i = 0; i < 10; ++i) {
        ENV_CREATE(user_lktest);
    }

    trap_init();
    kclock_init();
    while(1);
    panic("init.c:\tend of mips_init() reached!");
}
```

编译并运行：

```
make clean && make
/OSLAB/gxemul -E testmips -C R3000 -M 64 gxemul/vmlinux
```

参考输出

```
main.c: main is start ...
init.c: mips_init() is called
Physical memory: 65536K available, base = 65536K, extended = 0K
to memory 80401000 for struct page directory.
to memory 80431000 for struct Pages.
pmap.c:  mips vm init success
I'm 4c09
I'm 4408
I'm 3c07
I'm 3406
I'm 2c05
I'm 2404
I'm 1c03
I'm 1402
I'm c01
I'm 400
```

此处**略去了输出中的空行，以及 pageout 的输出。**该测试中各进程间的输出顺序也不是唯一确定的。

你还可以将用户程序中 `syscall_release_console();` 一行注释掉再编译运行（即获取锁后不再释放锁），可能观察到仅有第一个获取锁的进程成功输出。哪个进程成功获取锁同样是不确定的。评测使用的进程不会在持有锁时不释放锁直接退出。

实现提示

- 请回顾在 MOS 操作系统中添加系统调用的具体流程，包括需要修改哪些文件
- 可以在内核中记录锁的相关状态，使内核中的系统调用函数能够确定当前持有锁的进程
- 无需处理死锁、进程销毁前未释放锁等错误行为

提交评测

```
git add .
git commit -m "finish exam"
git push origin lab4-1-exam:lab4-1-exam
```

lab4-1-Extra

创建并切换分支

```
git checkout lab4
git checkout -b lab4-1-Extra
```

题目背景

通过使用系统调用，我们在 MOS 系统中实现了进程间通信（IPC）机制，允许进程之间传递数据和共享页面。阅读 `user/ipc.c` 后我们可以发现，通过 IPC 机制发送数据的用户函数 `ipc_send` 是使用轮询实现的。在接收方调用 `ipc_recv` 进入接收状态之前，发送方会不断使用 `syscall_ipc_can_send` 发起系统调用，尝试发送数据，这种忙等待的方式会产生较高的 CPU 和系统调用开销。

反观接收函数 `ipc_recv` ，它只需要进行一次系统调用即可完成接收，这是因为我们在 `sys_ipc_recv` 函数中通过修改进程的 `env_status` 实现了阻塞机制，让接收进程在数据发来前处于阻塞状态，不被调度运行。在 `sys_ipc_can_send` 函数进行发送时，接收进程才被唤醒。

题目描述

在本题中，你需要调整 IPC 机制的实现，修改内核中的 `sys_ipc_can_send` 和 `sys_ipc_recv` 这两个函数，使得用户进程使用 `ipc_send` 时只需要调用一次 `syscall_ipc_can_send` 就能完成发送，不进行多次系统调用。

实现要求

- 修改 `lib/syscall_all.c` 中 `sys_ipc_can_send` 和 `sys_ipc_recv` 函数的实现（根据需要，可定义其他辅助函数和变量）
- 如下修改 `user/ipc.c` 中 `ipc_send` 函数的实现：

```
void ipc_send(u_int whom, u_int val, u_int srcva, u_int perm) {
    syscall_ipc_can_send(whom, val, srcva, perm);
}
```

- **不要**修改用户空间中 `syscall_ipc_can_send`、`syscall_ipc_recv` 和 `msyscall` 函数的原有实现

评测说明

- 评测满足以下约束：

约束	数值
创建的最大进程数目	25
单个进程进行 IPC 发送的最大次数	50
单个进程进行 IPC 接收的最大次数	50

- 评测程序进行的 IPC 操作一定合法，即参数中传入的 envid 一定属于已创建且未退出的进程，传入的虚拟地址一定在用户空间内且页对齐，地址非 0 时的 `perm` 一定带有 `PTE_V`，且接收进程一定与发送进程不同，也不会涉及死锁等错误行为。
- 评测**不会**检查 `sys_ipc_can_send` 的返回值。
- 评测时，仓库中的以下文件或目录可能被替换为标准版本。标准版本中的 `ipc_send` 函数与“实现要求”中所提供代码的行为一致。
  - `Makefile`
  - `user/Makefile`
  - `user/ipc.c`
  - `user/syscall_lib.c`
  - `user/syscall_wrap.S`
  - `init/init.c`

实现提示

- 可以参考当前对接收进程实现的阻塞机制，在接收进程进入接收状态前，让发送进程进入阻塞状态从而不被调度。在接收进程开始接收时，需要唤醒发送进程，并在接收过程中为其完成发送操作。这种情况下，接收进程不再需要被阻塞。
- 发送进程进行发送时，如果接收进程已处于接收状态，则不需要阻塞发送进程，可如同课下实现直接发送。
- 你可以将完成发送时传递数据的过程封装为单独的函数，并在发送和接收过程中调用，以处理以上的两种情况。
- 在接收进程开始接收时，可能有多个进程都向其进行过发送，此时 `sys_ipc_recv` 仍应保证每次接收只选择一个发送进程，接收其数据并将其唤醒，而其余发送进程仍应处于阻塞状态，等待接收进程下一次调用 `ipc_recv`。我们对这一接收顺序没有要求（可以与发送顺序不一致），但不应有发送的数据被遗漏。
- 在阻塞一个发送进程时，你可能需要将描述本次发送的相关数据信息存储在内核中的缓冲区中，包括发送方和接收方的 envid，以及被发送的 `value`、`srcva` 和 `perm`。开始接收时，你需要找到该接收进程对应的数据信息。
- 你可以为每个未进入接收状态的进程准备一个接收队列，将来自每个发送进程的数据放入其中暂存。开始接收时，需要检查接收进程的接收队列是否为空。
- 为了保证内存映射的正确性，请避免在 `struct Env` 中新增字段。你可以使用 `ENVX` 宏等方式获取进程控制块在 `envs` 中的下标，并使用独立的静态数组为每个进程在内核中维护必要的信息。

以上提示仅供参考，在满足题目要求，正确实现 IPC 机制的前提下，你的实现可以不遵循以上提及的实现细节。

本地测试

首先，将 `user/syscall_lib.c` 中的 `syscall_ipc_can_send` 函数修改如下：

```
int syscall_ipc_can_send(u_int envid, u_int value, u_int srcva, u_int perm) {
    writef("%x: sending %d to %x\n", env->env_id, value, envid);
    msyscall(SYS_ipc_can_send, envid, value, srcva, perm, 0);
    return 0;
}
```

然后创建如下两个用户程序：

```
// user/ipcsend.c
#include "lib.h"

void umain() {
    u_int me = syscall_getenvid();
    u_int dst = envs[3].env_id;
    u_int val = ENVX(me);
    ipc_send(dst, val, 0, 0);

    u_int who;
    int *buf = (int *) 0x60000000;
    ipc_recv(&who, buf, 0);
    writef("%x: got %d from %x\n", me, buf[0], who);

    while (1);
}
```

```
// user/ipcrecv.c
#include "lib.h"

void umain() {
    u_int me = syscall_getenvid();
    u_int i, sum = 0, whos[3];
    for (i = 0; i < 10000; ++i)
        syscall_yield();
    for (i = 0; i < 3; ++i)
        sum += ipc_recv(&whos[i], 0, 0);

    int *buf = (int *) 0x60000000;
    syscall_mem_alloc(0, buf, PTE_V | PTE_R);
    buf[0] = sum;

    for (i = 0; i < 3; ++i)
        ipc_send(whos[i], i, buf, PTE_V | PTE_R);
    while (1);
}
```

将以上两个用户程序分别保存为 `user/ipcsend.c` 和 `user/ipcrecv.c`，最后参照 lab4-1-exam 或指导书中的相关描述，修改 `user/Makefile` 中的构建目标，并将 `init/init.c` 中的 `mips_init()` 修改为以下内容：

```
void mips_init() {
```

```
    printf("init.c:\tmips_init() is called\n");
    mips_detect_memory();
    mips_vm_init();
    page_init();
    env_init();

    int i;
    for (i = 0; i < 3; ++i) {
        ENV_CREATE(user_ipcsend);
    }
    ENV_CREATE(user_ipcrecv);

    trap_init();
    kclock_init();
    while(1);
    panic("init.c:\tend of mips_init() reached!");
}
```

参考输出

```
1402: sending 2 to 1c03
c01: sending 1 to 1c03
400: sending 0 to 1c03
1c03: sending 0 to 400
1c03: sending 1 to c01
1c03: sending 2 to 1402
1402: got 3 from 1c03
c01: got 3 from 1c03
400: got 3 from 1c03
```

此处略去了部分空行、内核初始化和 pageout 的输出。各行输出的顺序不是唯一确定的，且可能出现交错，但前三行输出应当与你一致（内部顺序可能不同）。

提交评测

```
git add .
git commit -m "finish extra"
git push origin lab4-1-Extra:lab4-1-Extra
```