Lab3: 基于方向的内核同步机制

Lab3的主要任务是实现一个新的内核同步机制，使得进程可以等待特定的方向事件，当设备被转动到该方向时自动解锁。

Lab3可以组队完成（不超过3人），Git repo的地址是https://classroom.github.com/g/WCH52QDf. 团队的所有成员都可以访问该私有库，并且每个成员应至少对团队的Git私有库进行五次提交修改。熟悉基于团队的共享私有库Git命令，例如 `git pull`， `git merge` 和 `git fetch` 。 重点是要进行增量式更改并使用迭代开发周期。

1. 用户空间守护进程：orientation daemon

Lab3的第一步是实现一个用户空间的守护进程(daemon)，将Android设备的方向传感器得到的值传给内核。

在Android平台上，设备的方向信息可以由其自带的方向传感器获得。 编写一个名为 `orientd` 的守护进程，该进程轮询方向传感器并将其返回值写入内核。使用下面的系统调用接口更新内核中的设备方向信息：

```
/*
 * Sets current device orientation in the kernel.
 * System call number 326.
 */
int set_orientation(struct dev_orientation *orient);

struct dev_orientation {
    int azimuth; /* rotation around the X-axis (-180<=azimuth<=180)*/
    int pitch;   /* rotation around the Y-axis: -90<=pitch<=90 */
    int roll;    /* rotation around Z-axis: +Y == -roll, -180<=roll<=180 */
};
```

你的私有库中的orientd目录下提供了一个基本的程序模板，该目录包括：

- `orientd.c`： daemon进程的主要功能驻留和轮询传感器数据的实现。
- `orientd.h`： 定义 `dev_orientation` 结构

编译模板程序只需输入 `make` 。要在模拟器上安装程序，请将使用 `adb push` 将其推送到模拟器。该模板程序会循环打印出设备方向数据。你需要把它变成一个守护(daemon)进程。并使daemon程序间歇运行，而不是让CPU一直循环读取设备。使用usleep添加计时器或使守护程序暂停，并在守护程序文件中定义间歇的时间间隔。所有相关的函数放在 `kernel/orientation.c` 和 `include/linux/orientation.h` 中。

需要注意的是，只有管理员（root用户）才能运行orientd并更新设备方向信息，且orientd须是一个守护进程，即它应该使用 `fork()` 将它自己放在后台，而不是使用shell（如＆符号或Ctrl-Z，bg命令）将程序强制进入后台序列。

提示： 可以在安装自定义内核之前先用默认4.4.124内核测试orientd，看其是否能够正确读取方向数据并打印，测试成功后再编写新的系统调用并编译安装新内核。

Android模拟器的方向传感器使用方法：启动仿真器后，单击右侧工具栏中的“...”，然后点击“Virtual sensor”。你可以拖动虚拟设备或下方的滚动条来操作模拟器。

在本地计算机中编译用户空间程序

如果你使用的是macOS或Linux，在本地计算机上编写和编译用户空间程序更方便。为此，你需要在本地计算机上安装Android NDK。你可以在这里下载NDK。 下载并解压缩NDK后，进入android-ndk-r18目录并运行：

```
build/tools/make_standalone_toolchain.py --arch x86_64 --api 28 --install-dir PATH_TO_NDK
```

将PATH_TO_NDK替换为要安装NDK的路径。然后，通过运行以下命令将$NDK_PATH环境变量设置为PATH_TO_NDK/bin：

```
echo export NDK_PATH=\"PATH_TO_NDK/bin\" >> ~/.bashrc
source ~/.bashrc
```

安装好Android NDK后，即可以在本地计算机中运行make来编译测试程序，并将其推送到模拟器中运行。

注意： 在macOS系统上，请勿使用 `git add --all` 或 `git commit -a` ，因为macOS系统文件名不区分大小写，而Linux内核源码的文件名是大小写敏感的，这样会导致内核不能正常编译。

2. 基于方向的内核同步机制

Lab3的第2步是设计并实现新的内核同步机制，该机制应允许一个或多个进程阻塞在一个方向事件 `P` 上，直到设备模拟器处于该特定方向 `P` 上。 在 `set_orientation` 中更新设备方向时，应解锁在包含新仿真器方向的方向事件上阻止的所有进程。 如果在内核中没有进程等待该特定方向，则该操作无效。

具体来说，该同步机制的API需要实现的以下一组新的系统调用。

```
/*
 * Create a new orientation event using the specified orientation range.
 * Return an event_id on success and appropriate error on failure.
 * System call number 327.
 */
int orientevt_create(struct orientation_range *orient);

/*
 * Destroy an orientation event and notify any processes which are
 * currently blocked on the event to leave the event.
 * Return 0 on success and appropriate error on failure.
 * System call number 328.
 */
int orientevt_destroy(int event_id);

/*
 * Block a process until the given event_id is notified. Verify that the
 * event_id is valid.
 * Return 0 on success and appropriate error on failure.
 * System call number 329.
 */
int orientevt_wait(int event_id);

struct orientation_range {
    struct dev_orientation orient;  /* device orientation */
    unsigned int azimuth_range;     /* +/- degrees around X-axis */
    unsigned int pitch_range;       /* +/- degrees around Y-axis */
    unsigned int roll_range;        /* +/- degrees around Z-axis */
};
```

```
/* Helper function to determine whether an orientation is within a range. */
static __always_inline bool orient_within_range(struct dev_orientation *orient,
                        struct orientation_range *range)
{
    struct dev_orientation *target = &range->orient;
    unsigned int azimuth_diff = abs(target->azimuth - orient->azimuth);
    unsigned int pitch_diff = abs(target->pitch - orient->pitch);
    unsigned int roll_diff = abs(target->roll - orient->roll);

    return (!range->azimuth_range || azimuth_diff <= range->azimuth_range
            || 360 - azimuth_diff <= range->azimuth_range)
        && (!range->pitch_range || pitch_diff <= range->pitch_range)
        && (!range->roll_range || roll_diff <= range->roll_range
            || 360 - roll_diff <= range->roll_range);
}
```

修改 `set_orientation` 系统调用，使其通知打开的方向事件中范围包含当前方向的所有事件。事件被通知后，等待该事件的所有进程都应被解锁。如果没有进程等待该事件，则不做任何反应。

首先应该考虑解决此问题所需的数据结构。系统需要支持多个进程同时等待不同的方向范围，因此可能需要一组方向范围的描述符数据结构，每个结构都标识一个事件。这些数据结构需要放在一个列表中，从该列表中可以找到你需要的范围描述符。范围描述符所使用的内存空间应该由动态分配获得，即使用内核函数 `kmalloc()` 和 `kfree()` 。

你的同步机制需要支持多处理器同步对数据结构的并发访问。此外，需要在保证正确的前提下提高访问效率。例如，在持有锁时可能进入睡眠状态的情况下需要使用信号量而不是自旋锁。了解如何同步对内核中缓冲区的访问，可以尝试使用内核结构 `kfifo` 。

你可以选择使用底层的等待队列方法，如 `add_wait_queue()` ， `remove_wait_queue()` ， 或使用更高级的 `prepare_to_wait()` ， `finish_wait()` ）等方法。在Linux kernel development第三版的第58-61页中可以找到代码示例。 `interrupttible_sleep_on()` 和 `sleep_on()` 等函数在某些情况下也可以使用。

提示： 在存在并发创建，访问和删除操作的情况下保证数据结构有效性的一种有用方法是维护对象的**引用计数**。此时，对象仅由最后一个使用的用户释放。例如，文件系统使用了inode结构的引用计数：当进程打开文件时，inode的引用计数递增。此时如果另一个进程删除该文件，则该inode在系统中依然有效（尽管不可见），因为其计数为正。当进程关闭相应的文件描述符时，计数递减，如果引用技术等于0，则释放该inode。由于存在并发访问，引用计数也需要使用显式同步方法或使用原子类型 `atomic_t` 进行保护。

你应该正确处理可能发生的错误并报告相应的错误代码，例如如果内存分配失败，则返回-ENOMEM。鼓励参考完成类似任务的现有内核代码。

3. 系统测试

写两个C程序分别进行系统测试。每个程序fork出n个child（n>2），然后所有child等待同一个方向事件，并执行以下操作：

- `faceup` ：当模拟器屏幕朝向你时每隔一秒打应 `"%d: facing up!"` ，其中 `"%d"` 是该进程在所有child集合中的序号。
- `facedown` ：当模拟器屏幕背向你时每隔一秒打应 `"%d: facing down!"` ，其中 `"%d"` 是该进程在所有child集合中的序号。

同时父进程等待60秒，然后通过关闭打开的方向事件来关闭所有子进程（而不是通过发送信号或其他此类方法）。 通过旋转仿真器验证系统是否正常运行。将测试程序放在repo中的tests目录中。

4. 注意事项

1. 遵循Linux内核编码风格 https://www.kernel.org/doc/html/v4.15/translations/zh_CN/coding-style.html 并使用checkpatch.pl检查你的代码。
2. Android NDK 下载地址和详细信息可见：https://developer.android.com/ndk/downloads/.