lab3-1-Extra-offline

lab3-1 Extra 课上测试已结束，若同学们未能完成课上题目且希望继续完善，可将代码推送到远程分支 `lab3-1-Extra-offline` 进行评测。我们对题目表述模糊的地方做了调整，也希望大家借此次题目引起对理论课内容的重视。

本任务没有截止时间，也不计入 OS 实验课的成绩。

创建并切换分支

如果**未做过**本次Extra：

```
git checkout lab3
git add .
git commit -m "xxxxx"
git checkout -b lab3-1-Extra
```

注意！请务必从lab3分支切换！！！

如果**做过**本次Extra：

```
git checkout lab3-1-Extra
```

题目描述

在理论课中，我们学习到了 PV 操作，它是一种实现进程同步和互斥的方法。本次题目我们将模拟 PV 操作。

下面对 PV 操作相关知识进行回顾，若自信理论课学习充分，可以跳过。

> PV 操作与信号量处理相关。对于一个信号量（将其记为 S ），在物理意义上其表示一种资源的个数，必须且只能设置一次初值。信号量的数值只能通过 P、V 操作改变。
>
> P 操作：物理意义上表示消耗资源。具体行为是检查信号量初值，若**大于 0**，表示有空闲资源，并分配一个资源；若**等于 0**，表示没有空闲资源，此时进程阻塞等待；资源的语义下，信号量值不会出现**小于 0** 的情况，但理论课中，我们用其相反数表示正在等待的进程个数。
>
> V 操作：物理意义上表示增加资源。具体行为是首先将信号量 S 加一，之后随机唤醒一个进程，也就是使其进入就绪队列准备被调度运行。
>
> 一个信号量维护一种资源，一个信号量对应一个等待队列，不同信号量维护的资源之间不能共享。
>
> 等待表示进程进入阻塞状态，除非得到自己想要的资源，否则不能进行任何其他活动。
>
> 一种资源可以有多个，一个进程也可以同时占有多个资源、多种资源。

由于 lab3-1 阶段进程无法运行与调度，因此我们将通过**模拟**的方式来完成。模拟的行为与实际运行有一定的差异，因此请认真阅读下面对本题机制的具体描述。

初始化信号量

在全局维护**两个**信号量，通过在所有 PV 操作前调用指定的初始化函数，将相关信号量初值设置为特定值 `x` ，在物理意义上其表示目前共有 `x` 个资源可供使用。

以下机制描述均针对**某一个信号量及其维护的某一种**资源。

P操作

物理意义上表示消耗资源。具体行为：检查信号量，**若大于 0**，表示有空闲资源，因此申请成功，进程获得资源，更新信号量大小并更新进程状态；**若等于 0**，表示没有空闲资源可供使用，此时进程更新状态，进入等待队列并**排在队尾**等待资源；资源的语义下，信号量值不会出现**小于 0**的情况。**你可能维护进程持有的资源个数**，需要在 P 操作执行时一并更新：若成功获得资源，则进程拥有资源个数加一。

V操作

物理意义上表示释放资源。具体行为：检查等待队列，若此时**有进程在等待资源**，则将资源分配给队首进程，本进程与获得资源的进程均更新状态；若此时**无进程等待资源**，则信号量加一并更新本进程状态。**你可能维护进程持有的资源个数**，需要在 V 操作执行时一并更新：若成功释放资源，则进程拥有资源个数减一，此处有一种未定义行为，当进程持有资源个数为 0 时，不继续减一，即保持进程持有资源个数**非负**。

错误检查

与实际运行不同，在模拟中，P、V 操作由顶层函数统一发出，但从实际运行逻辑看，进程可能无法执行这一操作。具体地，当进程处于等待资源状态时，其无法主动执行 P、V 的任何操作，若此时对此进程发出操作命令，将不按前述逻辑执行，而是直接返回错误码。

题目要求

你需要在 `lib/env.c` 中完成以下函数：

全局初始化信号量 `S_init`

- 函数原型：`void S_init(int s, int num)`
- 测试时此函数仅会被调用两次，且调用时间在所有 PV 操作前，用于将编号为 s 的信号量初始值设置为 num 数值。两次调用时 s 的值分别为 1 和 2 ，表示两个不同信号量。
- 注意：函数名中 S 为大写字母，传入参数 s 为小写字母。

P操作 `P`

- 函数原型：`int P(struct Env* e, int s)`
- 调用此函数后，e 所指向的进程申请获得一个由 s 信号量管理的资源，这里 s 取值仅限于 1 和 2 ，具体逻辑见上述。
- 若操作能够执行（不论是否成功得到资源，都算能够执行），函数返回 0；反之（进程在执行此操作时已处于等待队列中），不执行操作，并返回 -1.

V操作 `V`

- 函数原型：`int V(struct Env* e, int s)`
- 调用此函数后，e 所指向的进程释放一个由s信号量管理的资源，这里 s 取值仅限于 1 和 2 ，具体逻辑见上述。
- 由于评测截取实际应用中的一部分PV操作，**其中可能出现一种未定义行为，即在进程手中没有此资源时仍执行了 V 操作，我们约定此时仍按照释放一个资源执行，但进程自己维护的本资源个数保持为 0，不再减一。**

- 若操作能够执行（不论是否成功增加资源，都算能够执行），函数返回 0；反之（进程在执行此操作时已处于等待队列中），不执行操作，并返回 -1.

进程状态查看 `get_status`

- 函数原型：`int get_status(struct Env* e)`
- 调用此函数，返回进程状态对应的数值（表述方便，记为b）：
    - 若进程正在队列中等待资源分配，b = 1
    - 若进程未处于等待状态且占有任一资源，b = 2
    - 若进程未处于等待状态且未占有任何资源，b = 3

进程创建函数`my_env_create`

- 函数原型：`int my_env_create()`
- 调用此函数，功能类似于 `env_create_priority`，由于本题目下进程不实际运行，无需加载二进制镜像，也无需设置进程优先级，因此本函数没有参数。为便于评测，请**返回创建进程的 env_id**，若创建失败，请返回 -1. 在此基础上你可以根据自己的实现机制增添其他初始化内容。
- 评测中此函数可能在任何时刻调用。

提示

PV操作及信号量机制的实现所依赖的数据结构一般为：一个整数（信号量）+ 一个队列（等待队列）。

等待队列的实现方法可自行选择，一种方法可以参考 env_free_list 的实现机制。

再次强调，不同信号量维护的资源不能共用，比如，若一个进程在等待 2 信号量的资源，1 信号量所维护的资源的释放不能解除此进程的等待状态。

注意

- 对于所有在 `lib/env.c` 中新增的函数，请在 `include/env.h` 中添加相应的函数声明。
- 由于评测指令数较多，请务必在提交评测前注释掉所有新增函数内的 `printf` 代码，以免超时。
- 如果你需要在进程控制块中维护相应内容，请在Env结构体中新增字段，不要借用其他 lab 的字段。同时，你需要保证 `sizeof(struct Env)` 不超过256.
- PV 操作部分我们不对算法复杂度有严格要求，但评测中 `get_status` 大量调用，此函数请务必使用 `O(1)` 算法，否则可能超时。

评测逻辑

在评测中，我们会替换 init.c 文件，在初始化操作系统后，首先调用 `S_init` 函数初始化信号量，之后按一定的顺序调用 P，V 两个函数，并在其中穿插 `my_env_create` 创建进程以及`get_status` 检查特定进程的状态。

样例说明

extra 部分分为基础测试和强测两部分，各占50分。评测会对错误信息给出一定反馈（包括期望结果与你的结果），如：

- "P func should return 0 but we got -1" 表示 P 操作返回值错误
- "V func should return 0 but we got -1" 表示 V 操作返回值错误
- "status should be 3 but we got 2" 表示进程状态错误
- "Other Error"表示除上述错误外的其他错误

本地测试

编写完成后，将 init/init.c 中的 `mips_init` 函数替换为如下内容：

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

    pv_check(); // for lab3-1-Extra local test

    *((volatile char*)(0xB0000010)) = 0;
}
```

之后在 init/init.c 文件内新增函数 `pv_check()`，并在其中编写测试代码，示例如下：

```c
void pv_check() {
    S_init(1, 1);
    S_init(2, 1);
    struct Env* e1, *e2, *e3;
    envid2env(my_env_create(), &e1, 0);
    envid2env(my_env_create(), &e2, 0);
    envid2env(my_env_create(), &e3, 0);
    printf("%d\n", P(e1, 1));
    printf("envid: %d, status: %d\n", e1->env_id, get_status(e1));
    printf("%d\n", P(e2, 1));
    printf("envid: %d, status: %d\n", e2->env_id, get_status(e2));
    printf("%d\n", P(e3, 1));
    printf("%d\n", P(e3, 2));
    printf("envid: %d, status: %d\n", e3->env_id, get_status(e3));
    printf("%d\n", P(e1, 2));
    printf("%d\n", V(e1, 1));
    printf("%d\n", V(e1, 2));
    printf("envid: %d, status: %d\n", e1->env_id, get_status(e1));
}
```

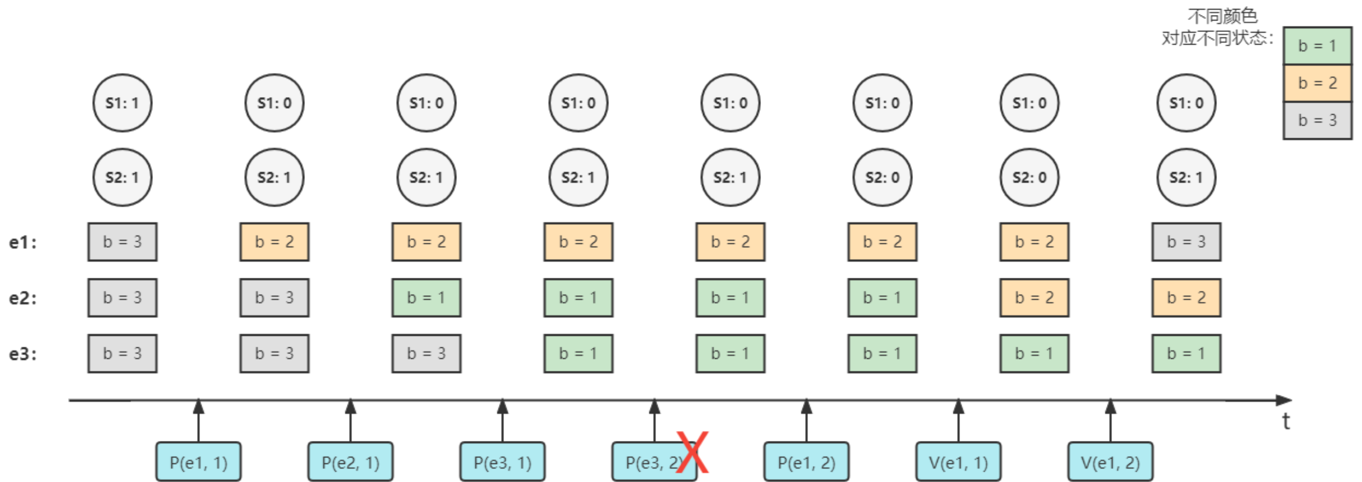
编译&运行后，得到正确输出如下：

```
0

envid: 1024, status 2

0

envid: 3073, status 1

0

-1

envid: 5122, status 1

0
```

```

0

0

envid: 1024, status 3
```

这里我们给出此样例的图解。

初始信号量 S1 和 S2 各一个，最上两行表示本时刻信号量数值；3 至 5 行不同颜色表示各进程本时刻不同状态，对应关系见右上角；最下一行表示该时刻进行的 PV 操作；红叉表示此操作失败。



代码提交

```
git add .
git commit -m "xxxxx"
git push origin lab3-1-Extra:lab3-1-Extra-offline
```