lab3-1-exam

创建并切换分支

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

题目描述

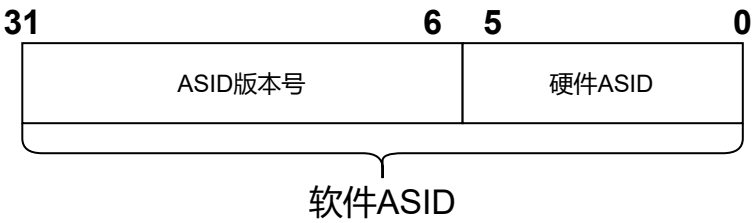
在指导书中，我们向大家介绍了ASID的概念以及它所带来的一系列限制。（详见指导书 113 - 114 页）

由于ASID的数量有限，我们需要建立一个管理机制来防止ASID冲突使得系统崩溃。在我们的MOS系统中，我们选择采用位图法来分配ASID从而保证不同的进程拥有不同的ASID。这种解决方法简单有效，但是这也导致MOS中只能同时运行64个进程。在我们实验课的小系统中，这个数量的进程绰绰有余，但是对于Linux这种商业系统，这个数量就显得太过小了。

那么Linux是怎么解决这个问题的呢?
答案是通过**版本号机制**来对ASID的**时效性**进行管理。

软件ASID

在进程的PCB中会存储软件ASID的值，其为一个32位无符号整型数，其低6位为指导书中介绍的TLB中使用的ASID，称为硬件ASID（实际上硬件ASID的位数随硬件而定，在我们的系统中为**6位**），其余的位为进程ASID的版本号，由操作系统管理。如下图所示：



为了避免引起歧义，我们再详细阐述一下上述定义：

软件ASID是一个 `u_int` 型的变量，保存在 `Env` 块中，假设其名为 `env_asid`。

硬件ASID为 `env_asid` 的低 6 位，可以通过 `env_asid & 0x3f` 获得。

ASID版本号是 `env_asid` 的高 26 位，可以通过 `env_asid >> 6` 获得。

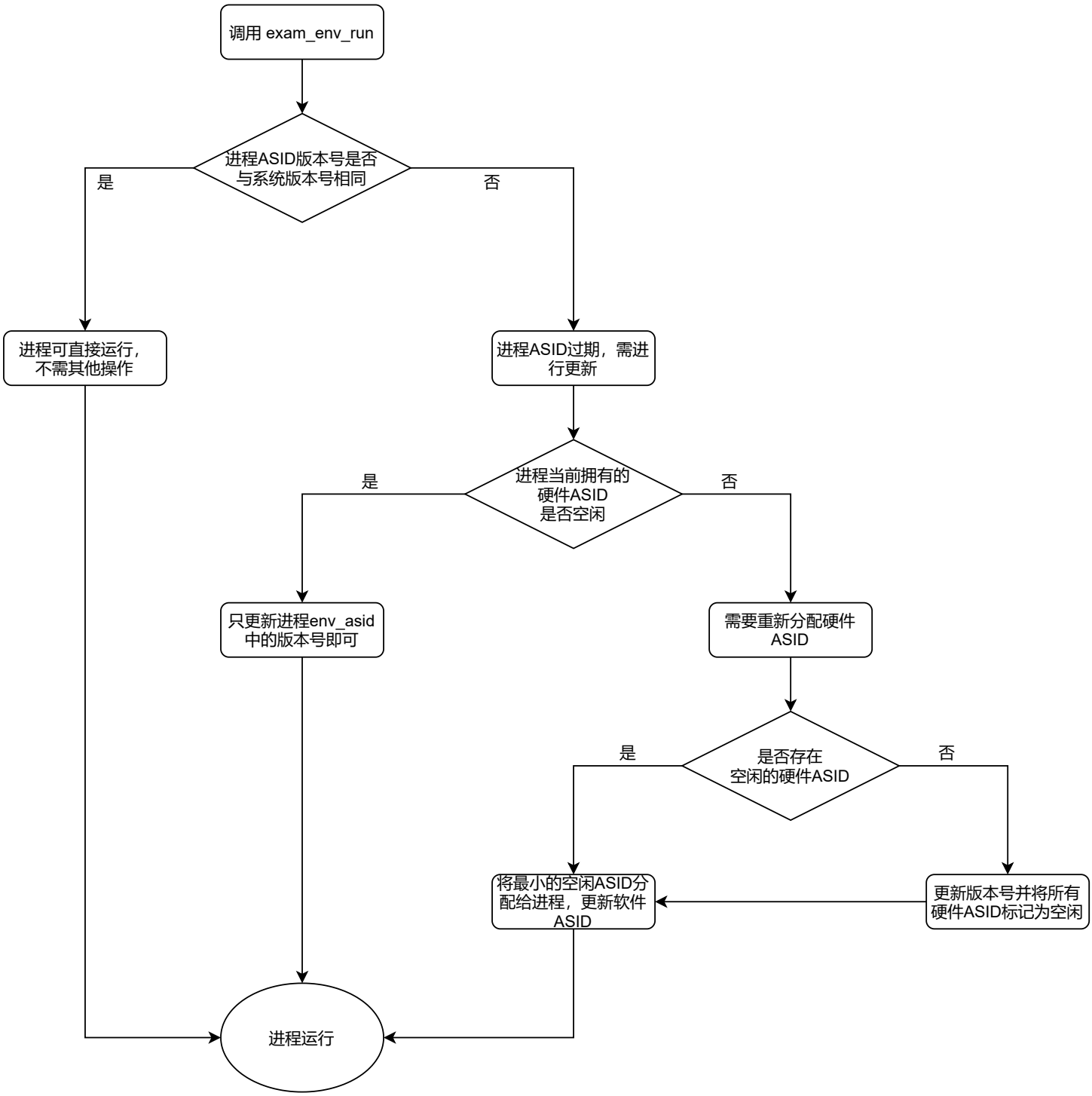
版本号机制的具体实现

1. 操作系统自身维护一个当前的系统 ASID 版本号和一个用于管理已分配的硬件 ASID 的数据结构。
2. 进程控制块中保存进程自身的软件 ASID，其在进程创建时初始化为 0.
3. 进程运行时首先检测进程的 ASID 版本号是否和系统版本号相同，
    - 如果相同，说明该进程当前的 ASID 有效，可以直接运行。
    - 如果不同，说明该进程当前的 ASID 已过期，需要为进程分配新的ASID。

首先，系统判断进程`env_asid`的硬件ASID部分在当前系统版本号下是否已被分配，如果未被分配，只更新`env_asid`中的版本号。否则需要为其分配新的硬件ASID：

- 如果当前系统版本号下存在未分配的硬件 ASID，则分配出最小的空闲硬件ASID，将系统版本号与新分配到的硬件ASID拼接并写入进程控制块中的ASID字段。
- 如果不存在未分配的硬件 ASID，则更新系统版本号（将版本号加一），将所有硬件ASID置为空闲状态，并将最小的空闲ASID分配给当前进程

具体流程如下图：



题目要求

由于lab3-1阶段进程无法运行与调度，因此我们将通过**模拟**的方式来完成上述机制，具体如下：

- 我们规定，软件 asid 的格式如下：
  0-5位为硬件ASID，6-31位为版本号（与上面介绍相同）
- 修改 mkenvid() 为如下内容：(函数代码已随题目下发，如与下面函数有偏差，可能导致评测无法通过)

```c
u_int mkenvid(struct Env *e)
{
    /*Hint: lower bits of envid hold e's position in the envs array. */
    u_int idx = (u_int)e - (u_int)envs;
    idx /= sizeof(struct Env);

    /*Hint: avoid envid being zero. */
    return (1 << (LOG2NENV)) | idx;  //LOG2NENV=10
}
```

- 修改 include/env.h 中 `Env` 结构体的定义，增加字段 `u_int env_asid` ，并在创建进程时将其初始化为 0.

  请注意，**我们在测试时会读取该字段，所以请务必保证命名相同**

- 在 `lib/env.c` 中新增函数 `u_int exam_env_run(struct Env *e)`，**并在 include/env.h 中添加函数声明**
  在其中实现前文所述的 ASID 的分配逻辑，更新进程控制块中的 `env_asid` 字段，要求：

  1. 在有多于一个空闲的 ASID 时，**分配最小的。**
  2. 版本号从 `0x4` 开始，当硬件ASID耗尽时，版本号增加 1。
  3. 如果在此次调用过程中更新了系统版本号，函数返回 1，否则返回 0。

  注意：不要进行除上述要求外的其他操作

- 在 `lib/env.c` 中新增函数 `void exam_env_free(struct Env *e)`，**并在 include/env.h 中添加函数声明**
  这个函数用于释放该进程占用的 ASID，要求：

  1. 如果进程 ASID 的版本号和系统版本号相同，需要释放该进程占用的 硬件ASID，使该 ASID 可以参与接下来的分配
  2. 对于没有分配 ASID 或 ASID 过期的进程，不需进行任何操作

  注意：不要进行除上述要求外的其他操作

- 在 **lib/env.c** 的 `env_init()` 函数中添加相应内容使得每次调用此函数时系统版本号复原为 0x4，且所有硬件ASID均重置为可分配状态，**注意原有代码保持不变。**（因为评测中我们会在一次编译后模拟多次完整过程）

提示：我们的 *mos* 系统使用了位图法管理 *ASID*，*lib/env.c* 中的 *asid_alloc, asid_free* 函数稍作修改即可用于此次测试，当然你可以选择采用其它方式

样例说明

样例名称	调用 exam_env_run 次数	样例特点	分数
basic test 1	< 100	测试 `exam_env_run` 的基本功能，包括： 是否能够给进程的 `env_asid` 字段赋值 是否能够在 ASID 耗尽时更新版本号 **注：该样例保证不调用 `exam_env_free`**	20
basic test 2	< 100	在 basic test 1 的基础上增加了 `exam_env_free` 的调用	20
basic test 3	< 100	对于边界情况的弱测，如： 释放版本号过期的进程	20
strong test	约 2000	强测	每个测试点20分 共40分

测试样例

编写完成后，将 `init/init.c` 中的 `mips_init()` 函数删除，并加入如下代码。

```
void lab3_1_exam_test(){
    struct Env *envs[100];
    struct Env *e;
    int i;

    /* alloc 100 envs for testing */
    for (i = 0; i < 100; i++) {
        if (env_alloc(&e, 0) != 0) {
            return;
        }
        envs[i] = e;
    }

    /* run 64 env to use all asid */
    for (i = 0; i < 64; i++) {
        exam_env_run(envs[i]);
    }

    /* run env with valid asid, should not flush TLB */
    printf("ret = %d\n", exam_env_run(envs[10]));

    /* first free a env, so we get a free asid */
    exam_env_free(envs[0]);
    envs[0]->env_asid = 0;

    /* run a env, it should have hardware asid of 0 */
    printf("ret = %d\n", exam_env_run(envs[64]));
    printf("envid = %x, asid = %x\n", envs[64]->env_id, envs[64]->env_asid);

    /* run a env, there should be no free asid, so we should flush TLB */
```

```
    printf("ret = %d\n", exam_env_run(envs[0]));
    printf("envid = %x, asid = %x\n", envs[0]->env_id, envs[0]->env_asid);

    /* run a env with invalid asid, but its hardware asid is free, so only update its generation */
    printf("ret = %d\n", exam_env_run(envs[5]));
    printf("envid = %x, asid = %x\n", envs[5]->env_id, envs[5]->env_asid);

    /* run a env with invalid asid, and its hardware asid is not free, so alloc a new asid */
    printf("ret = %d\n", exam_env_run(envs[64]));
    printf("envid = %x, asid = %x\n", envs[64]->env_id, envs[64]->env_asid);
}

void mips_init(){
    mips_detect_memory();
    mips_vm_init();
    page_init();

    env_init();

    lab3_1_exam_test();

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

运行如下指令：

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

正确输出如下：

```
ret = 0

ret = 0

envid = 440, asid = 100

ret = 1

envid = 400, asid = 140

ret = 0

envid = 405, asid = 145

ret = 0

envid = 440, asid = 141
```

代码提交

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

lab3-1-extra

创建并切换分支

```
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！！！

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；显然，信号量值不会出现**小于0**的情况。
>
> 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。

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。

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；显然，信号量值不会出现**小于0**的情况。

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。

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。
- 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。

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.

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：

```
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：

```
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));
```

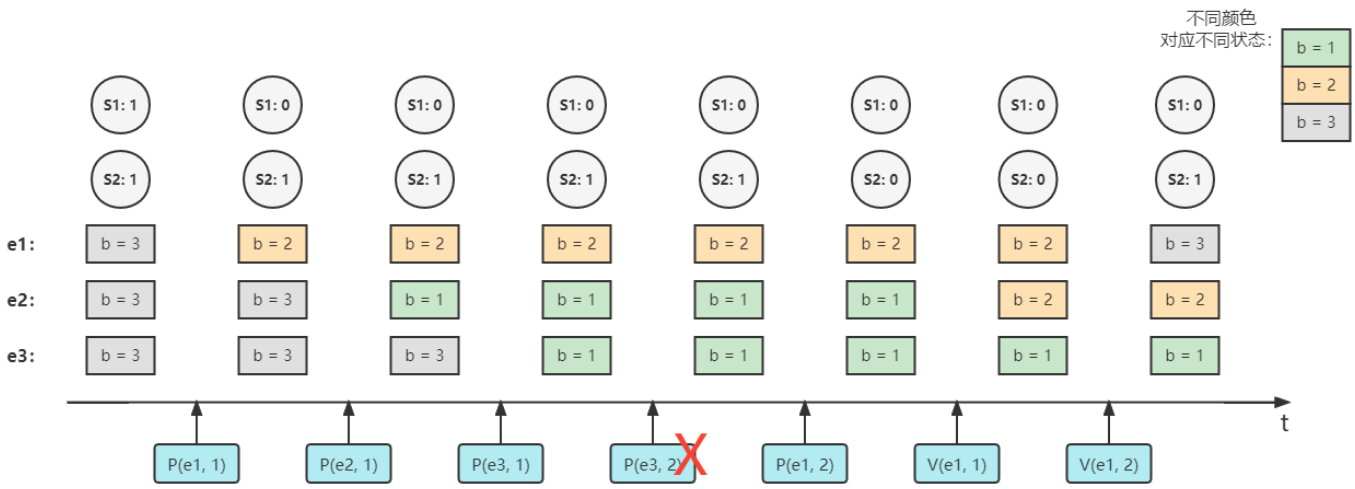
```
    printf("%d\n", P(e3, 1));
    printf("%d\n", P(e3, 2));
    printf("envid: %d, status: %d\n", e3->env_id, get_status(e3));
    printf("%d\n", P(e1, 2));
    printf("%d\n", V(e1, 1));
    printf("%d\n", V(e1, 2));
    printf("envid: %d, status: %d\n", e1->env_id, get_status(e1));
}
```

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
```