

Lab4 Page table 页表

操作系统实验指导书 - 2024秋季 [实验概述 \(https://elearning.fudan.edu.cn/courses/78523/pages/shi-yan-gai-shu\)](https://elearning.fudan.edu.cn/courses/78523/pages/shi-yan-gai-shu) | [常见问题汇总 \(https://elearning.fudan.edu.cn/courses/78523/pages/chang-jian-wen-ti-hui-zong\)](https://elearning.fudan.edu.cn/courses/78523/pages/chang-jian-wen-ti-hui-zong) | [Lab1 UTIL \(https://elearning.fudan.edu.cn/courses/78523/pages/lab1-util-xv6yu-unixying-yong-cheng-xu\)](https://elearning.fudan.edu.cn/courses/78523/pages/lab1-util-xv6yu-unixying-yong-cheng-xu) | [Lab2 SYSCALL \(https://elearning.fudan.edu.cn/courses/78523/pages/lab2-syscall-xi-tong-diao-yong\)](https://elearning.fudan.edu.cn/courses/78523/pages/lab2-syscall-xi-tong-diao-yong) | [Lab3 Scheduling \(https://elearning.fudan.edu.cn/courses/78523/pages/lab3-scheduling-jin-cheng-diao-du\)](https://elearning.fudan.edu.cn/courses/78523/pages/lab3-scheduling-jin-cheng-diao-du) | **Lab4 Page table**

一、实验目的

1. 了解页表的实现原理。
2. 了解xv6物理地址和虚拟地址的转换和寻址, 内核态下的内存地址和用户态下的内存地址的差别。
3. 修改页表, 使内核更方便的进行用户虚拟地址翻译。

二、实验准备

2.1 切换分支

首先确保你已经拥有远程仓库的最新代码, 然后切换到 pgtbl 分支后进行开发。

```
$ git fetch
$ git checkout pgtbl
$ make clean
```

在切换分支之前, 记得通过 `git commit -m` 保存之前实验的内容。如果你希望舍弃之前实验的更改, 也可以使用-f选项强制切换分支, 例如 `git checkout -f pgtbl`。

2.2 参考资料

在做实验之前, 请同学们阅读以下材料:

1. [xv6手册](https://pdos.csail.mit.edu/6.828/2020/xv6/book-riscv-rev1.pdf)  (https://pdos.csail.mit.edu/6.828/2020/xv6/book-riscv-rev1.pdf) 的以下章节及相关源代码:

- xv6 book, Chapter 3 Page tables (页表)
- kernel/memlayout.h (定义内存的布局)
- kernel/vm.c (虚拟内存代码)
- kernel/kalloc.c (分配和释放物理内存代码)

2. [xv6中的虚拟内存管理](https://os-labs.pages.dev/lab4/part2/#3-xv6)  (https://os-labs.pages.dev/lab4/part2/#3-xv6)

三、实验内容

3.1 打印页表

3.1.1 实验目标

在 `exec` 中插桩一个打印函数，使得 `xv6` 启动时会打印首个进程的页表信息，来帮助你在之后的实验中进行debug。

3.1.2 实验流程

- 在 `kernel/vm.c` 中实现 `vmprint()`，并在 `exec()` 函数中插入语句 `if(p->pid==1) vmprint(p->pagetable)`，这条语句插在 `exec.c` 中 `return argc` 代码之前，即在第一个进程启动时打印页表信息。
- 在 `kernel/defs.h` 中定义 `vmprint()` 的接口，这样你才能在 `exec()` 中使用它。

3.1.3 预期结果

当`xv6`启动的时候，它自身会调用 `exec()` 启动第一个进程 `init`，这个时候我们的函数会得到以下的输出：

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
page table 0x0000000087f25000
llidx: 0: pa: 0x0000000087f21000, flags: ----
|| llidx: 0: pa: 0x0000000087f20000, flags: ----
|| || llidx: 0: va: 0x0000000000000000 -> pa: 0x0000000087f22000, flags: rwxu
|| || llidx: 1: va: 0x0000000000000100 -> pa: 0x0000000087f1f000, flags: rwx-
|| || llidx: 2: va: 0x0000000000000200 -> pa: 0x0000000087f1e000, flags: rwxu
llidx: 255: pa: 0x0000000087f24000, flags: ----
|| llidx: 511: pa: 0x0000000087f23000, flags: ----
|| || llidx: 510: va: 0x00000003ffffff00 -> pa: 0x0000000087f76000, flags: rw--
|| || llidx: 511: va: 0x00000003ffffff00 -> pa: 0x0000000080007000, flags: r-x-
init: starting sh
$
```

- 第一行打印的是 `vmprint` 的参数，即获得的页表参数具体的值。
- 在之后打印的则是页表项。RISC-V的页表被设计成了三层，每一个“||”都代表一层。
- 打印的格式为：（注意 冒号后面都接一个空格）
 - 如果是非叶节点，则为：`idx: [索引编号]: pa: [物理地址], flags: [四个权限位(r/w/x/u)]`
 - 如果是叶子节点，则为：`idx: [索引编号]: va: [虚拟地址] -> pa: [物理地址], flags: [四个权限位(r/w/x/u)]`
- 上述的详细含义为：
 - `索引编号`：指示了该页表项在当前等级页表内的序号（取值范围：0-511）；
 - `物理地址`：指示了这个页表项对应的十六进制物理地址；

- **flags**的四个权限

位：指示了这个页表项的flags，包括读 (R)、写 (W)、执行 (X)、用户态 (U)

- 只打印有效的pte。在上面的示例中，根页表项只有第0项和第255项的映射是有效的，其中第0项的次页表只映射了索引0，该索引0映射了叶子页表的0、1和2。你的代码输出的物理地址与上述示例可能不相同，但显示项数和虚拟地址应相同。
- 测试**：运行 `make grade`，其中的 `pte printout` 测试就是该任务的测试结果


3.1.4 提示

- 使用 `printf()` 打印页表数据中的指针时，你可以直接使用 `%p` 标示。
- 由于xv6不支持%c, 因此打印字符的时候请通过将字符转化为字符串的方式，使用%s格式化字符串。
- 参考 `kernel/riscv.h` 文件末的宏定义。
- `kernel/vm.c` 中的函数 `freewalk()` 能帮助你理解遍历页表的过程。如果是遍历到叶子节点，需要打印虚拟地址va。

```
// Recursively free page-table pages.
// All leaf mappings must already have been removed.
void
freewalk(pagetable_t pagetable)
{
    // there are 2^9 = 512 PTEs in a page table.
    // 遍历一个页表的PTE表项
    for(int i = 0; i < 512; i++){
        pte_t pte = pagetable[i]; //获取第i条PTE

        /* 判断PTE的Flag位，如果还有下一级页表(即当前是根页表或次页表)，
           则递归调用freewalk释放页表项，并将对应的PTE清零 */
        if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){
            // this PTE points to a lower-level page table.
            uint64 child = PTE2PA(pte); // 将PTE转为为物理地址
            freewalk((pagetable_t)child); // 递归调用freewalk
            pagetable[i] = 0; // 清零
        } else if(pte & PTE_V){

            /* 如果叶子页表的虚拟地址还有映射到物理地址，报错panic。
               因为调用freewalk之前应该会先uvmunmap释放物理内存 */
            panic("freewalk: leaf");
        }
    }
    kfree((void*)pagetable); // 释放pagetable指向的物理页
}
```

- 建议进入QEMU界面的console，输入 `info mem` 获取页表，可用于对比你自己打印的页表与系统中的页表是否一致（参考[系统调用](https://os-labs.pages.dev/remote_env_gdb2)  (https://os-labs.pages.dev/remote_env_gdb2))。

3.2 任务二：独立内核页表

目前，xv6的每个进程都有自己独立的 **用户页表**（只包含该进程用户内存的映射，从虚拟地址0开始），但是每个进程进入内核的时候，会使用唯一的一个 **全局共享内核页表**。我们需要 **将全局共享内核页表改成独立内核页表**，使得每个进程拥有自己独立的内核页表，也就是全局共享内核页表的副本。

3.2.1 独立页表的背景

共享内核页表中，所有物理地址都和与之完全相等的虚拟地址建立映射，也就是直接映射。这是让内核能够直接以物理地址访问内存的数据，不需要使用任何的虚拟地址。

但是，由于用户地址的映射并未存储于内核页表，如果我们需要处理用户程序传来的虚拟地址（比如系统调用传入的指针），我们需要先找到用户页表，逐个页表项地找到能够翻译对应虚拟地址的页表项后，才可以获取实际的物理地址并进行访问，这叫做软件模拟翻译。软件模拟翻译的实现很复杂，同时，因为需要复杂的查找，还降低了性能。

所以我们将 **用户页表中的内存映射** 和 **原来共享内核页表中的内存映射** 进行合并，这样内核也能够直接对用户的虚拟地址进行访问，而不需要软件模拟翻译。

3.2.2 独立页表的要求

共享内核页表的映射：虚实地址相同，也就是直接映射。

独立内核页表的映射：虚实地址相同的映射应该要保留。

同时还需要修改有关的操作。

3.2.3 测试

首先，在xv6运行 `kvmtest`，如果你确实使用了独立内核页表，会看到以下结果：

```
$ kvmtest
kvmtest: start
kvmtest: OK
$
```

然后，请在xv6运行 `usertests`，确保所有测试通过（显示"ALL TESTS PASSED"）。

3.2.4 （一种可参考的）流程

Step 1： 修改 `kernel/proc.h` 中的 `struct proc`，增加两个新成员：`pagetable_t k_pagetable;` 和 `uint64 kstack_pa;`，分别用于给每个进程中设置一个内核独立页表 and 内核栈的物理地址。

Step 2： 仿照 `kvminit()` 函数重新写一个创建内核页表的函数。

- **为进程分配内核页表的一种解决方案：** 不要修改全局的内核页表（`kernel/vm.c`中的 `pagetable_t kernel_pagetable`），而是直接创建一个新的内核页表，并将其地址 `k_pagetable` 返回。实现的时候不要映射CLINT，否则会在任务三发生地址重合问题。

Step 3： 修改 `procinit` 函数。`procinit()` 是在系统引导时（见 `kernel/main.c` 的 `main` 函数），用于给进程分配内核栈的物理页并在页表建立映射。

- **参考优化方法：**把 `procinit()` 中内核栈的物理地址 `pa` 拷贝到PCB新增的成员 `kstack_pa` 中，同时还需要保留内核栈在全局页表 `kernel_pagetable` 的映射，然后在Step 4 `allocproc()` 中再把它映射到进程的内核页表里。关于内核栈说明，见[实验原理 3.4](https://os-labs.pages.dev/lab4/part2/#34) [↗\(https://os-labs.pages.dev/lab4/part2/#34\)](https://os-labs.pages.dev/lab4/part2/#34)。

为什么要保留初始内核页表？

保留原有的 `kvminit()` 以及 `kernel/vm.c` 中的 `kernel_pagetable`，因为有些时候CPU可能并未执行用户进程。

Step 4：修改 `allocproc` 函数。`allocproc()` 会在系统启动时被第一个进程 `userinit()` 和 `fork()` 调用。在 `allocproc` 函数里调用Step 2 创建的函数设置内核页表，并且参考借鉴 `kvmmmap` 函数将Step 3 设置的内核栈映射到页表 `k_pagetable` 里。

`allocproc` 函数功能说明

在进程表中查找空闲 `PCB`，如果找到，初始化在内核中运行所需的状态（如初始化PID、trapframe、用户页表等），并保持 `p->lock` 返回。如果没有空闲 `PCB`，或者内存分配失败，则返回0。

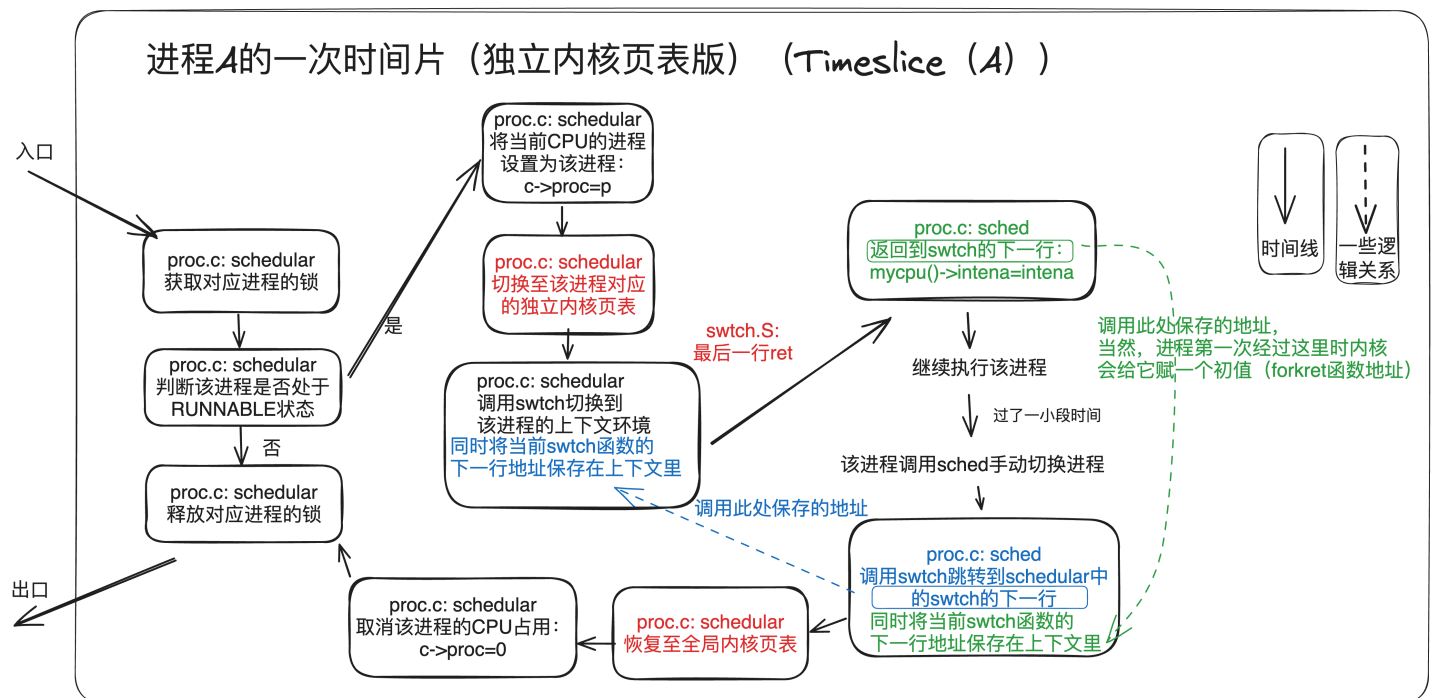
关于内核栈映射

注意，要保证在每一个进程的内核页表中映射该进程的内核栈。xv6本会在 `procinit()` 中分配内核栈的物理页并在页表建立映射。但是现在，应该在 `allocproc()` 中实现该功能，因为执行 `procinit()` 的时候进程的内核页表还未被创建。你可以在 `procinit()` 中只保留内存的分配，但在 `allocproc()` 中完成映射。

Step 5：修改调度器（`scheduler`），使得切换进程的时候切换内核页表。

- **参考方法：**在进程切换的同时也要切换页表将其放入寄存器 `satp` 中，请借鉴 `kvminithart()` 的页表载入方式)，在调用 `w_satp()` 之后调用 `sfence_vma()`，具体原理可查阅实验原理的[3.3 内核对用户空间的访问](https://os-labs.pages.dev/lab4/part2/#33) [↗\(https://os-labs.pages.dev/lab4/part2/#33\)](https://os-labs.pages.dev/lab4/part2/#33)。
- **无进程运行的适配：**当目前没有进程运行的时候，`scheduler()` 应该要 `satp` 载入全局的内核页表 `kernel_pagetable` (`kernel/vm.c`)。
- 关于 `scheduler` 调度器，可以参考[HITSZ操作系统课程组讲解XV6（三）内存管理](https://www.bilibili.com/video/BV1Te4y1i77z?share_source=copy_web&vd_source=225a99017e082147ac525beeddd6e3e2) [↗\(https://www.bilibili.com/video/BV1Te4y1i77z?share_source=copy_web&vd_source=225a99017e082147ac525beeddd6e3e2\)](https://www.bilibili.com/video/BV1Te4y1i77z?share_source=copy_web&vd_source=225a99017e082147ac525beeddd6e3e2)

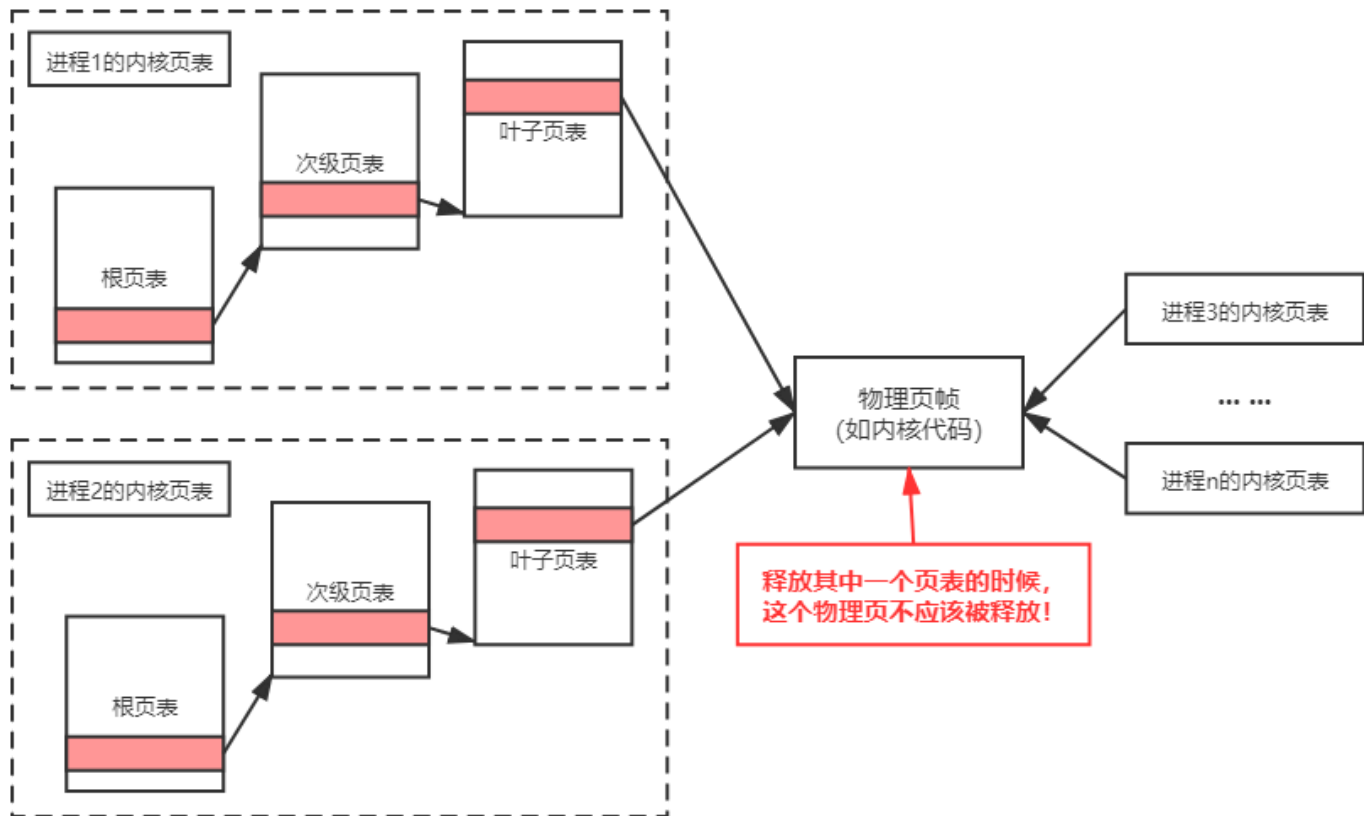
这里我们给出一个调度器内的页表切换流程图，同学们可以对照[实验原理部分的原版xv6的调度器流程](https://os-labs.pages.dev/lab4/part2/#32) [↗\(https://os-labs.pages.dev/lab4/part2/#32\)](https://os-labs.pages.dev/lab4/part2/#32) 比对观察区别：



需要注意的是：调度器作为一个永不返回的程序，其运行也需要页表的支持，所以同学们需要实现方框中红色字体的部分，这样可以使得 **每个进程都运行在自己的内核独立页表的支持下**，调度器运行在全局内核页表的支持下，不会出现地址映射混乱的情况。

Step 6：当进程结束的时候，你需要修改 `freeproc()` 函数来释放对应的内核页表。你需要找到 **释放页表但不释放叶子页表指向的物理页帧** 的方法。你可参考 `kernel/vm.c` 中的 `freewalk`，其用于释放整个页表，但要求叶子页表的页项已经被清空。

- 虽然我们为每个进程引入了内核页表，但是内核的代码和数据都还是唯一的。这意味着，在各个内核页表的叶子页表中，页表项指向了共享的物理页，如下图所示：



- 因此，在我们释放某个进程的内核页表的时候，不应该把这个共享物理页帧释放，否则会产生非常严重的后果！

Step 7： 通过 `usertests` 和 `kvmtest` 测试。

3.2.5 提示

这里有必要提醒一下同学，要以最简单的方法以及工程量最小的方法实现，最好不要因为优化等问题，大改内核机制。

- 好好利用 `vmprint()` 来帮助debug。
- 页表的bug通常会导致映射缺失的traps，访存失败，指令运行错误等报错。如果 内核 缺失了地址映射造成了页缺失（page fault），通常会打印个 `sepc=0x00000000XXXXXXXX`，这代表的是出错时pc的值，你可以查 `kernel/kernel.asm` 看看 对应地址 的代码的含义。
- 你的实现可以修改原有的函数或者是新增函数，最好放在 `kernel/vm.c + kernel/proc.c` 中，但是 千万不要修改 `kernel/vmcopyin.c + kernel/stats.c + user/usertests.c + user/stats.c`。

3.3 问答题

- 阅读参考材料 `xv6 book Chapter 3 Page tables` 以及 [xv6中的虚拟内存管理](https://os-labs.pages.dev/lab4/part2/#3-xv6) (https://os-labs.pages.dev/lab4/part2/#3-xv6)，阐述SV39标准下，给定一个64位虚拟地址为 `0xFFFFFFE789ABCDEF` 的时候，是如何一步一步得到最终的物理地址的？（页表内容可以自行假设）

- 我们注意到, SV39标准下虚拟地址的L2, L1, L0 均为9位。这实际上是设计中的必然结果, 它们只能是9位, 不能是10位或者是8位, 你能说出其中的理由吗? (提示: 一个页目录的大小必须与页的大小等大)

四、实验结果提交

将实验报告和实验代码打包为压缩文件提交到eLearning平台。

4.1 实验报告

参照[课程文件](#)

(<https://elearning.fudan.edu.cn/courses/78523/files/folder/%E5%AE%9E%E9%AA%8C%E7%9B%B8%E5%85%B3%E6%96%87%E6%A1%A3>) 中的《OS实验报告模板》

(<https://elearning.fudan.edu.cn/files/4915949/>), 按如下要求书写实验报告, 力争规范、简洁。

- 对于每个实验, 详细描述实验过程, 对于你认为的关键步骤附上必要的截图。
- 有需要写代码的实验, 必须配有代码、注释以及对代码功能的说明。
- 鼓励实验报告中包括但不限于以下内容: 实验过程中碰到了什么问题? 如何解决这些问题? 实验后还存在哪些疑问或者有什么感想?
- 如果实验附有练习, 请在每个练习之后作答, 这是实验报告评分的重要部分。

4.2 实验代码

不需要提交完整的代码包, 只需要提交 `commit.patch` 文件即可, 操作步骤如下:

- 在完成实验之后, 将当前分支上的所有更改进行提交 (`commit`, 具体方法参考[git使用教程](#) <https://os-labs.pages.dev/lab1/part4/#3-git>)。
- 在仓库的目录下使用 `make diff` 命令导出更改文件 `commit.patch`。

4.3 提交平台

请将生成的 `commit.patch` 文件与实验报告一起打包提交到eLearning平台Lab4

操作系统实验指导书 - 2024秋季 [实验概述](#) (<https://elearning.fudan.edu.cn/courses/78523/pages/shi-yan-gai-shu>) | [常见问题汇总](#) (<https://elearning.fudan.edu.cn/courses/78523/pages/chang-jian-wen-ti-hui-zong>) | [Lab1 UTIL](#) (<https://elearning.fudan.edu.cn/courses/78523/pages/lab1-util-xv6yu-unixying-yong-cheng-xu>) | [Lab2 SYSCALL](#) (<https://elearning.fudan.edu.cn/courses/78523/pages/lab2-syscall-xi-tong-diao-yong>) | [Lab3 Scheduling](#) (<https://elearning.fudan.edu.cn/courses/78523/pages/lab3-scheduling-jin-cheng-diao-du>) | **Lab4 Page table**