

|  |
| --- |
| Title |
| Subject |

Release V1.0

奕行智能科技有限公司

2024年3月24日

© 2024 EVAS Intelligence, all rights reserved

[1 引言 X](#_cfbbc5de8443c54348dc98f8eaa6e449)

[1.1 引言 X](#_ffb63d8d008bbfe18175381a67219a4a)

[1.2 实验内容 X](#_9aac764d22d7c94be58fc3f6fd296597)

[1.3 实验设计 X](#_1b28638c0c0698da109c5fed59e190da)

[1.4 实验环境配置 X](#_bdf5c23f815afe310d3b490db790c94a)

[1.4.1 虚拟机平台以及镜像 X](#_0959a08a76a595d6bf558b13245dd7e1)

[1.4.2 交叉编译器 X](#_79ca41d9d40eed2b1129d79639e137df)

[1.4.3 loongarch64仿真器 X](#_b2d46390a3c7d4207e0310950a492d37)

[2 实验一：内核、启动和printf X](#_4493145c3269ae826ed5468e91783950)

[2.1 实验目的 X](#_619d5a03dfb04719301b7e975b566ee1)

[2.2 操作系统的启动 X](#_9e30d0b2f9faa5365fef21a63707614d)

[2.2.1 硬件模拟器 X](#_c06746a714e27a725da80218bfd98c6e)

[2.2.2 硬件模拟器的启动流程 X](#_6e82ede5a6675d79fd24e936c9e59d77)

[2.3 内核，启动！ X](#_288e4434c508f4a291bfac95be6630cc)

[2.3.1 Makefile—内核代码的地图 X](#_7c73a2cc60b132f3a751b2491ba32fe2)

[2.3.2 ELF—操作系统内核的本质 X](#_f1e5309c9b7a79920969d56ecff6c4f8)

[2.3.3 LinkerScript—控制内核加载地址 X](#_af01f6fa54b4a2d5ff17b54b0b66937c)

[2.3.4 \_start—程序执行的第一条指令 X](#_35963b560d923e50ce9aca6568fb65da)

[2.4 实战printk X](#_cc5a5da3c461d8024392079cae8f9280)

[2.5 实验一正确结果 X](#_e81f7906381c9516d831176af378324a)

[3 实验二：内存管理 X](#_770ca3ccb7cb26170697471ea3588a39)

[3.1 实验目的 X](#_9844a15c30f381aa8f5957853da5bb13)

[3.2 loongarch64的访存流程 X](#_b2618db9cef1de9299592ccd2fc6e797)

[3.2.1 CPU发出地址 X](#_9176da038e5ccbffa0fb977dbf773c77)

[3.2.2 虚拟地址映射到物理地址 X](#_357519e329197fe9b4bb206966a73209)

[3.3 内核程序启动 X](#_4c6e44883e57e986abc9bcfc6d5839ea)

[3.3.1 la32r\_detect\_memory X](#_e784193be7a73cdde8326fd33d849e6b)

[3.3.2 la32r\_vm\_init X](#_20e5836e2f5a412cb847a6f54389306b)

[3.4 物理内存管理 X](#_e16dad7eb1ae3fc21877dcd233fcb0df)

[3.4.1 链表宏 X](#_ac13d4affff65dc38209413cd66db4b3)

[3.4.2 页控制块 X](#_1534e94a835db348dc96114878e46478)

[3.4.3 其他相关函数 X](#_7ad9d50c436746a43f29eac5c2cdd812)

[3.4.4 实验结果展示 X](#_1717e8d443dd23f5b5622fc89534e5de)

[3.5 虚拟内存管理 X](#_b14b1b60a5fd295438760fcfdbf3f2e3)

[3.5.1 二级页表结构 X](#_1362d6b811e76294dfbf16ae19dc3b6f)

[3.5.2 与页表相关的函数 X](#_86f1ea4e7021546498472f097b0ca108)

[3.6 访问内存与TLB重填 X](#_ca6a1567a9dfc0ae10f842a30de0988d)

[3.6.1 TLB相关的前置知识 X](#_936c72a31126ce293a8ae24aab390c13)

[3.6.2 TLB维护流程 X](#_47ddf950a1e76e531b35ffd3c5aba6f5)

[3.7 实验正确结果 X](#_cf5e4e63288d6a2c17e6df98289148da)

[3.7.1 实验代码功能解释 X](#_3367470c26f211378cb9d1ce7fa4ba8c)

[3.7.2 正确实验结果展示 X](#_5b7c95e3392790f4ee6aa8746d27fffb)

[4 实验三：进程与异常 X](#_30d846276930c68b29cb16adb580f4e8)

[4.1 实验目的 X](#_ac54576680941a4ba73924b996fc6494)

[4.2 进程 X](#_db152346d90df4f78cffee1e0c0aa61d)

[4.2.1 进程控制块 X](#_a4b61455935861135d6f1c3984679da1)

[4.2.2 段地址映射 X](#_99fb8b10ee5720d5bd98e6fb6903685d)

[4.2.3 进程的标识 X](#_b38eceadf86c4ac9f2c7df08321ae531)

[4.2.4 设置进程控制块 X](#_4a0af195030fdf958fb6efecfbedf293)

[4.2.5 加载二进制镜像 X](#_9dbe3e1d27b4f7e37f327cbde4a59c82)

[4.2.6 创建进程 X](#_3b06ce6ce442b57c2e3b71725dfb647c)

[4.2.7 进程运行与切换 X](#_404c084714639de616a6c0174736fd78)

[4.2.8 实验正确结果 X](#_e6c9bbcd59e584f6b8c66b33771c534b)

[4.3 中断与例外 X](#_6afa73e10162e588e340dba7d07f0772)

[4.3.1 例外的分发 X](#_b517c77a8c097dd05bb3fcd6fb023801)

[4.3.2 例外向量组 X](#_6bf723253e5f79c773b7d0027104e47f)

[4.3.3 时钟中断 X](#_b37b70078ff6d48c6189df0e44b7ca51)

[4.3.4 进程调度 X](#_8107177f5778704e0535b72778aa4db4)

[4.4 代码导读 X](#_233e91c272b1d7108552e6e30c439820)

[4.5 实验正确结果 X](#_6cc069a54dc28ae78bf13978b16c3d57)

[5 实验四：系统调用与FORK X](#_b823eaee3d6276e27a7a0676f64898ac)

[5.1 实验目的 X](#_a9982b566185dd55b2e7327cca0885eb)

[5.2 系统调用(System Call) X](#_1383e11b0706bf1fd295f536e7f81db6)

[5.2.1 用户态与内核态 X](#_e949f5f14ae57e6e21907bf51e6176d8)

[5.2.2 系统调用实例 X](#_b94e7b088c6a7de570d000dbaea3dd7f)

[5.2.3 系统调用机制的实现 X](#_e9eb0d8c4c16a22fd9140039f008491a)

[5.2.4 基础系统调用函数 X](#_82da0af679583948bea3cd0b49c5103a)

[5.3 进程间通信机制(IPC) X](#_8586aef8bf4b14d1adbe1dcc72a7f646)

[5.4 Fork X](#_90962697a7010ab893e75eacef652e0b)

[5.4.1 初窥 fork X](#_21811a036472ce7fe13c6374b678fcf3)

[5.4.2 写时复制机制 X](#_d94fda4b12d9d439ff7ec1660475bd4f)

[5.4.3 fork 的返回值 X](#_350ab4445486b4a754cbe8671348e511)

[5.4.4 地址空间的准备 X](#_e12ae97acc03125c193cf139c15364fb)

[5.4.5 页写入异常 X](#_8edfdc5aa12bc1e11860f086444160a3)

[5.5 实验正确结果 X](#_5e76ff9d727723a25acc5706257dbe39)

[6 实验五：文件系统 X](#_3fc521670700cb74b70f952b63857e49)

[6.1 实验目的 X](#_c631cd5c8df98aebb2478b15ec62b35b)

[6.2 文件系统概述 X](#_f2278ac3b476a004b108ec1c1b58e0e3)

[6.2.1 文件系统的设计与实现 X](#_8c08810262c0f88f582c30d592d798d1)

[6.3 IDE 磁盘驱动 X](#_e243179a005ec0a47e988416517f57ae)

[6.3.1 内存映射 I/O (MMIO) X](#_9f5566dd7279bbd333f410796c813157)

[6.3.2 IDE 磁盘 X](#_6a9a28d2e3b4eb2a3a49bf17aee61d46)

[6.3.3 驱动程序编写 X](#_e3df452dbf2fedf0552f701f5a4699e2)

[6.4 文件系统结构 X](#_ba0a29349c2c8d761574127e95f81c75)

[6.4.1 磁盘文件系统布局 X](#_693888363d59f7de52c7f4d46d8289e0)

[6.4.2 文件系统详细结构 X](#_64abc05922a22663211a10b1c2618df8)

[6.4.3 块缓存 X](#_0375c630971608d0d31457be7329f13b)

[6.5 文件系统的用户接口 X](#_3d439f6c23761a0ce58f7bfea02f0c5e)

[6.5.1 文件描述符 X](#_84048edc7c035dd617ca45c0c2a4b713)

[6.5.2 文件系统服务 X](#_205b14e53e9cf780604ef47d5fd574f3)

[6.6 实验结果展示 X](#_bc0a453c1ae41a5fd9086fa9d1a35d79)

[6.6.1 IDE 磁盘交互 X](#_ca3cc56424024fb8211959a2c37f7252)

[6.6.2 文件系统测试 X](#_b2c8e273056c01e717506234308e6b8c)

[6.6.3 文件系统服务测试 X](#_53ee481b6c164b5d0a3cd2a8cd83deef)

[7 实验六：管道与 SHELL X](#_be23c38b9aa07ed5b4b4ffd2d8882b7c)

[7.1 实验目的 X](#_f90008a5c378361325aae07ae685141c)

[7.2 管道 X](#_8eb5555138df27e8616c7d9ae75b65bd)

[7.2.1 初窥管道 X](#_4d7d6888a4ea0fcefc6a2dd9f7b5756f)

[7.2.2 la64 中 pipe 的使用与实现 X](#_06881a68098dfa8abde44509b76ad779)

[7.2.3 管道关闭的正确判断 X](#_187db720f9efb611cc380c13634e09eb)

[7.2.4 相关函数 X](#_6d14cd1421379e650325223f89aaaeb8)

[7.3 shell X](#_0678a52751729be97e7583b3a48bc5d5)

[7.3.1 学习 spawn 函数 X](#_afd00cee5d220aab0debd4b7c38449d5)

[7.3.2 解释 shell 命令 X](#_8749f7821b94e2263bb59a9197b870bf)

[7.3.3 相关函数 X](#_a8a1b7e46d7649cc5307876df7d43545)

[7.4 实验正确结果 X](#_ebc295a0c223288f452074992705f36f)

[7.4.1 管道测试 X](#_75f217b44b2f67c76d9eb1a31c5fc546)

[7.4.2 shell 测试 X](#_3a91eb132495752ada6ab076f48f2a73)

# 1 引言

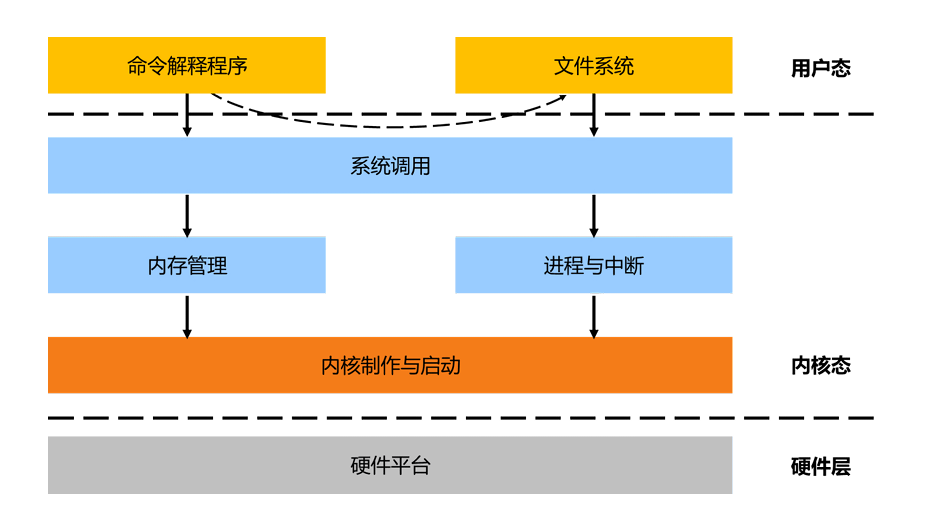
## 1.1 引言

本实验根据北航操作系统课程上提供的教学型小型操作系统mos改编而来。mos目前提供有mips和la32r两种架构的版本，经过详细分析代码和构建，发现la32r有着诸多不足之处，基本是mips架构的重编版本。于是为了学习loongarch架构和供后来者参考，在复现原本实验内容的前提下，将la32r版本改写为la64的版本，其中有些函数名字中的la32r未进行更改，但内容已经更新为la64架构，新版本充分体现了loongarch64特点。

## 1.2 实验内容

实验内容与原本mos完成的实验内容相同，分为6个实验（Lab1~Lab6），各个实验的相互关系如图1所示，具体实验内容如下：

1. 内核、启动和 printf ：通过 PC 启动的实验，掌握硬件的启动过程，理解链接地址、加载地址和重定位的概念，学习如何编写裸机代码。
2. 进程与异常：通过设置进程控制块和编写进程创建、进程中止和进程调度程序，实现进程管理；编写通用中断分派程序和时钟中断例程，实现中断管理。
3. 系统调用与fork：掌握系统调用的实现方法，理解系统调用的处理流程，实现本实验所需的系统调用。
4. 文件系统：通过实现一个简单的、基于磁盘的、微内核方式的文件系统，掌握文件系统的实现方法和层次结构。
5. 管道与shell：实现具有管道，重定向功能的命令解释程序shell，能够执行一些简单的命令。最后将6部分链接起来，使之成为一个能够运行的操作系统。



## 1.3 实验设计

由于开发一个实际的操作系统难度大、工作量繁重，为了保证教学效果，在核心能力部分采用微内核结构和增量式设计的原则，因此可以从最基本的硬件管理功能逐步扩充，最后完成一个完整的系统。实验内容的设计满足以下条件：

1. 每个实验可独立运行与测试，便于调试与评测，可获得阶段性成果。
2. 每个实验内容包含相对独立的知识点，并只依赖其前序实验。
3. 各个实验提交的代码一直伴随整个实验过程，可以不断改进、完善代码。整个系统结构如图2所示，蓝色部分是每次实验需要新增加的模块，绿色部分是需要修改完善的模块，灰色部分是不用修改的模块。在增量式设计下，可以从基本的功能出发，逐步完善整个系统，从而降低了学习操作系统的难度。

为了适应不同读者的学习要求，本书的实验采用分层的方式，从基础到复杂逐步实现实验的基本目标。因此，可将实验基本目标分为三个层次：

* 第一层次，掌握基本的系统使用与编程能力：包括 Linux、Makefile、git、vim 和仿真器的使用，基本的 shell 编程和使用系统调用编程;
* 第二层次，掌握操作系统核心能力：包括6个实验，从操作系统内核构造、内存管理、进程管理、系统调用、文件系统和命令解释程序，构成一个完整的小型操作系 统;
* 第三层次，锻炼操作系统提升能力：主要包括若干挑战性任务，学生需要独立在某一方面实现若干新的系统功能。

## 1.4 实验环境配置

所有实验须在linux系统上运行，因此需要搭建虚拟机平台，包含实验需要的开发环境，例如Linux环境、交叉编译器、loongarch64仿真器等。

### 1.4.1 虚拟机平台以及镜像

虚拟机平台使用VMware Workstation Pro，镜像使用的是ubuntu22.04，二者均可到官网下载，这里不再赘述安装过程。

### 1.4.2 交叉编译器

运行la64所需要的交叉编译器为loongarch64-unknown-linux-gnu-，可使用以下命令进行安装

wget https://github.com/loongson/build-tools/releases/download/2023.08.08/x86\_64-cross-tools-loongarch64-gcc-libc.tar.xz  
tar xvf ./x86\_64-cross-tools-loongarch64-gcc-libc.tar.xz

解压完成后使用sudo vim /etc/profile指令进入配置文件，在文件末尾添加如下两行代码：

export CROSS\_COMPILE=loongarch64-unknown-linux-gnu-  
export PATH=/path/of/your/cross/compile/cross-tools/bin:$PATH

保存退出后，重启虚拟机或使用如下指令重新应用配置文件：

source /etc/profile

重启系统后，使用以下命令检查软件安装情况：

$ loongarch64-unknown-linux-gnu-gcc --version  
*#出现以下版本信息则说明交叉编译器安装完成*   
loongarch64-unknown-linux-gnu-gcc (GCC) 14.0.0 20231018 (experimental)   
Copyright (C) 2023 Free Software Foundation, Inc.   
This is free software; see the source **for** copying conditions. There is NO   
warranty; not even **for** MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

### 1.4.3 loongarch64仿真器

实验中使用的硬件模拟器是QEMU，下面介绍如何进行安装下载QEMU源码包

$ wget https://download.qemu.org/qemu-9.0.2.tar.xz

解压

$ tar -xvf qemu-9.0.2.tar.xz

编译

$ cd qemu-9.0.2   
$ ../configure **\**    
 --target-list=loongarch64-softmmu **\**    
 --enable-slirp   
$ make -j24

完成后进行路径添加，使用vim ~/.bashrc指令进入配置文件，在文件末尾添加如下代码：

export PATH=/your/qemu/path/qemu-9.2.0/build:$PATH

保存退出后，重启虚拟机或使用如下指令重新应用配置文件：

source ~/.bashrc

重启系统后，使用以下命令检查QEMU安装情况：

$ qemu-system-loongarch64 -M ?   
*#出现以下信息则说明qemu安装成功*   
Supported machines are:   
none empty machine   
virt QEMU LoongArch Virtual Machine (default)

搭建好环境后，即可开始进行我们的实验。

# 2 实验一：内核、启动和printf

## 2.1 实验目的

1. 从操作系统角度理解la64体系结构
2. 掌握操作系统启动的基本流程
3. 掌握 ELF 文件的结构和功能
4. 理解 printk 函数的原理

## 2.2 操作系统的启动

### 2.2.1 硬件模拟器

计算机是由硬件和软件组成，如果仅有一个裸机什么工作也无法完成。另一方面，软件也必须运行在硬件之上才能实现其价值。由此可见，硬件和软件是相互依存、密不可分的。为了能较好的管理计算机系统的硬件资源，需要使用操作系统。那么在我们的操作系统实验中，需要管理的硬件在哪里呢？为了减少学习成本以及减少硬件不稳定带来的麻烦，在所有实验中使用 QEMU 硬件模拟器，模拟器可以模拟 CPU 等硬件环境。总的来说，在操作系统课程实验中，编写代码的环境是 Linux 系统，进行实验的硬件仿真平台是 QEMU 模拟器。实验编写的操作系统代码在 Linux 环境中通过 Makefile 来组织，通过交叉编译产生可执行文件，最后使用 QEMU 模拟器运行该可执行文件，实现 la64 操作系统的运行。

### 2.2.2 硬件模拟器的启动流程

有过操作系统基础知识的同学应该都知道，操作系统的启动是一个非常复杂的过程，需要经过 bootloader 引导才可以真正开始运行。不过，由于本次实验的 la64 操作系统的目标是在模拟器上运行，这个过程被大大简化了。模拟器支持直接加载 ELF格式的内核，也就是说，模拟器已经提供了 bootloader 的引导（启动）功能。la64 操作系统不需要再实现 bootloader 的功能。在 la64 操作系统的运行第一行代码前，我们就已经拥有一个正常的程序运行环境，内存和一些外围设备都可以正常使用。模拟器支持加载 ELF 格式内核，所以启动流程被简化为加载内核到内存，之后跳转到内核的入口，启动就完成了。这里要注意，之所以简单还有一个原因就在于QEMU本身是模拟器，是一种模拟硬件的软件而不是真正的硬件，所以就不需要面对传统的bootloader面对的那种非常纠结的情况了。

## 2.3 内核，启动！

接下来，我们就开始我们的la64操作系统内核的学习了。这一节中，我们将介绍如何修改内核并实现一些自定义的功能。

### 2.3.1 Makefile—内核代码的地图

当我们使用 ls 命令看看都有哪些实验代码时，会发现似乎文件目录有点多，各个不同的目录名称大致说明了他们各自的功用，但是挨个文件进行浏览还是有点难度。此时我们就可以进入到一个文件，叫做 Makefile。 相信有过操作系统学习的同学已经对 Makefile 有了初步的了解。下面这个 Makefile 文件即为构建我们整个操作系统所用的顶层 Makefil e。那么，我们就可以通过浏览这个文件来对整个操作系统的布局产生初步的了解：可以说，Makefile 就像源代码的地图，告诉我们源代码是如何一步一步成为最终的可执行文件的。以下代码是实验代码最顶层的 Makefile 节选，通过阅读它我们就能了解代码中很多宏观的东西。

include include.mk   
   
lab ?= **$(**shell cat .mos-this-lab 2>/dev/null || echo 6**)**   
   
target\_dir := target *# la64 构建目标所在目录*   
mos\_elf := **$(**target\_dir**)**/mos *# 最终需要生成的ELF可执行文件*   
user\_disk := **$(**target\_dir**)**/fs.img *# la64 文件系统使用的磁盘镜像文件*   
qemu\_pts := **$(**shell [ -f .qemu\_log ] && grep -Eo '/dev/pts/[0-9]+' .qemu\_log**)**   
link\_script := kernel.lds   
   
modules := lib init kern *# 需要生成的子模块*   
targets := **$(**mos\_elf**)**   
syms\_file := **$(**target\_dir**)**/prog.syms   
qemu\_files := **$(**mos\_elf**)**   
   
QEMU\_FLAGS := -M virt -m 4G -d in\_asm,mmu -D debug.log -nographic **\**

完整的 Makefile 文件内容在这里不做过多解释，请读者自行阅读理解。需要关注的是第16行的 QEMU\_FLAGS 参数 -d 和 -D ，-d 表示选择调试内容类型，in\_asm 代表输出 CPU 执行的每一条指令（反汇编），mmu 代表 MMU 地址转换，也就是 TLB 行为，-D 表示调试输出位置，二者经常搭配使用。比如在以上 Makefile 文件中，二者使用表示将 QEMU 中 CPU 执行的每一条反汇编指令以及涉及到的 TLB 行为输出到 Makefile 所在目录下的 debug.log 文件中，通过查看该日志即可观察 QEMU 的执行信息，经常用于调试。

### 2.3.2 ELF—操作系统内核的本质

通过阅读 Makefile 文件，我们大致把握了 la64 操作系统的代码组织结构。在这一小节里，我们来进一步探究 la64 代码最终编译链接得到“操作系统内核”究竟是什么。我们知道源代码文件需要经过编译和链接两个阶段，才能变成完整的可执行文件来运行。在编译阶段，每个源文件被翻译成二进制指令，得到目标文件，也就是我们通过-c编译选项生成的.o 文件。然而，由于完整程序的地址布局尚未确定，目标文件中的跳转等指令中的目标地址还是空的。在链接阶段，链接器会将所有的目标文件链接在一起，并填写具体的地址等信息，形成最终的可执行文件。接下来，我们提出我们的下一个问题：链接器通过哪些信息来链接多个目标文件呢？答案就在于在目标文件的结构中。在目标文件中，记录了代码各个段的具体信息。链接器通过这些信息 来将目标文件链接到一起。而 ELF(Executable and Linkable Format) 正是 Unix 上常用的一种目标文件格式。其实，不仅仅是目标文件，可执行文件也是使用ELF格式记录的。这一点通过 ELF 的全称也可以看出来。我们前面编译出来的la64内核本质上也是一个ELF文件。为了进一步了解编译出的内核文件，我们需要进一步探究ELF文件的功能以及格式。 ELF 是一种用于可执行文件、目标文件和库的文件格式。ELF格式是是 UNIX 系统实验室作为ABI(Application Binary Interface) 而开发和发布的，现在早已经是 Linux 下的标准格式了。我们在之前曾经看见过的.o文件就是ELF所包含的三种文件类型中的一种，称为可重定位 (relocatable) 文件，其它两种文件类型分别是可执行 (executable) 文件和共享对象 (shared object) 文件，这两种文件都需要链接器对可重定位文件进行处理才能生成。那么，ELF 文件中都包含有什么东西呢？简而言之，就是和程序相关的所有必要信息。下面我们可能进行一个小练习来观察ELF文件的结构。阅读 tools/readelf 目录下的 elf.h、readelf.c 和 main.c 文件，同时可以进入根目录下的 /usr/include/elf.h 文件进行阅读，这是 ubuntu 镜像所使用的elf.h文件，将其与 la64 使用的 elf.h 进行对比，看看我们使用的微内核与真正的操作系统使用的有何不同。在 tools/readelf 目录下执行 make 命令，即可生成可执行文件 readelf，它接受文件名作为参数，对 ELF 文件进行解析。可以执行 make hello 生成测试用的ELF文件 hello，然后运行 ./readelf hello 来测试 readelf。

### 2.3.3 LinkerScript—控制内核加载地址

生成内核文件之后，我们现在需要让内核被加载到正确的地址。之前在分析 ELF 文件时我们曾看到过，编译器在生成 ELF 文件时就已经记录了各节所需要被加载到的位置。同时，我们也发现，最终的可执行文件实际上是由链接器产生的（它将多个目标文件链接产生最终可执行文件）。因此，我们所需要做的，就是控制链接器的链接过程。接下来，我们就要引入一个神奇的东西：LinkerScript。链接器的设计者们在设计链接器的时候面临这样一个问题：不同平台的 ABI (ApplicationBinaryInterface) 都不一样，怎样做才能增加链接器的通用性，使得它能为各个不同的平台生成可执行文件呢？于是，就有了 Linker Script。LinkerScript 中记录了各个节应该如何映射到段，以及各个段应该被加载到的位置。使用命令 ld --verbose 可以输出默认的链接脚本，你可以在自己的机器上尝试这一条命令。因此，我们可以通过 LinkerScript 来尝试调整各节的位置。阅读 kernel.lds 文件，即可知道内核开始加载的位置。再补充一点：关于链接后的程序从何处开始执行。程序执行的第一条指令的地址称为入口地址（entrypoint）。我们的实验就在 kernel.lds 中通过 ENTRY(\_start) 来设置程序入口为 \_start。

### 2.3.4 \_start—程序执行的第一条指令

下面为 init/start.S 文件中内容，来大致阅读并分析一下 \_start 函数代码。

01 *#include <asm/asm.h>*  
02 *#include <mmu.h>*  
   
03 .section .text  
04 .**global** \_start  
05 \_start:  
06   
07 /\* initialize \*/  
08 *#if !defined(LAB) || LAB >= 3*  
09 la a0, exc\_gen\_entry  
10 csrwr a0, csr\_eentry  
11 *#endif*  
12   
13 *#if !defined(LAB) || LAB >= 2*  
14 la a1, tlb\_miss\_entry  
15 csrwr a1, csr\_tlbrentry  
16 *#endif*  
17   
18 li.d a0, 0x9000000000000001  
19 csrwr a0, csr\_dmw0  
20   
21 li.d a0, 0x0000000000000000  
22 csrwr a0, csr\_dmw1  
23 csrwr a0, csr\_dmw2  
24 csrwr a0, csr\_dmw3  
25   
26 li.d a0, 0xb0  
27 csrwr a0, csr\_crmd  
28   
29 li.d a0, 0x0c000000  
30 csrwr a0, csr\_tlbidx  
31   
32 li.d t0, ESTAT\_TI  
33 csrwr t0, csr\_ectl  
34   
35 li.d sp, KSTACKTOP  
36   
37 la.**global** t0, .bss  
38 la.**global** t1, bss\_end  
39 bss\_loop:  
40 beq t0, t1, bss\_loop\_end  
41 st.b zero, t0, 0  
42 addi.d t0, t0, 1  
43 b bss\_loop  
44 bss\_loop\_end:  
45 b la32r\_init  
46 nop

阅读以上代码，10~11，15~16行，作用是将我们后面要用到的函数起始地址写入寄存器中，如果要执行函数则会通过该寄存器记录的地址去进行函数执行。19-36行则是将一些数值写入寄存器中，后面会有详细的解释。36行则是将 sp 寄存器设置到内核栈空间的位置上，随后跳转到 la32r\_init 函数。内核栈空间的地址可以从include/mmu.h 中看到。 这里做一个提醒，请注意栈的增长方向。设置完栈指针后，我们就具备了执行C语言代码的条件，因此，接下来的工作就可以交给C代码来完成了。所以，在\_start的最后，我们调用C代码的主函数，正式进入内核的C语言部分。其中 la32r\_init 函数位于 init/init.c 中。

## 2.4 实战printk

了解了这么多的内容后，我们来进行一番实战，在内核中实现一个 printk 函数。在平时使用中，你可能觉得 printf 函数是由语言本身提供的，其实不是，printf 全都是由C语言的标准库提供的。而C语言的标准库是建立在操作系统基础之上的。所以，当我们开发操作系统的时候，我们就会发现，我们失去了C语言标准库的支持。我们需要用到的几乎所有东西，都需要我们自己来实现。要弄懂内核如何将信息输出到控制台，需要阅读以下三个文件：kern/printk.c，lib/print.c 和kern/machine.c。首先大家先对它们的大致的内容有个了解：

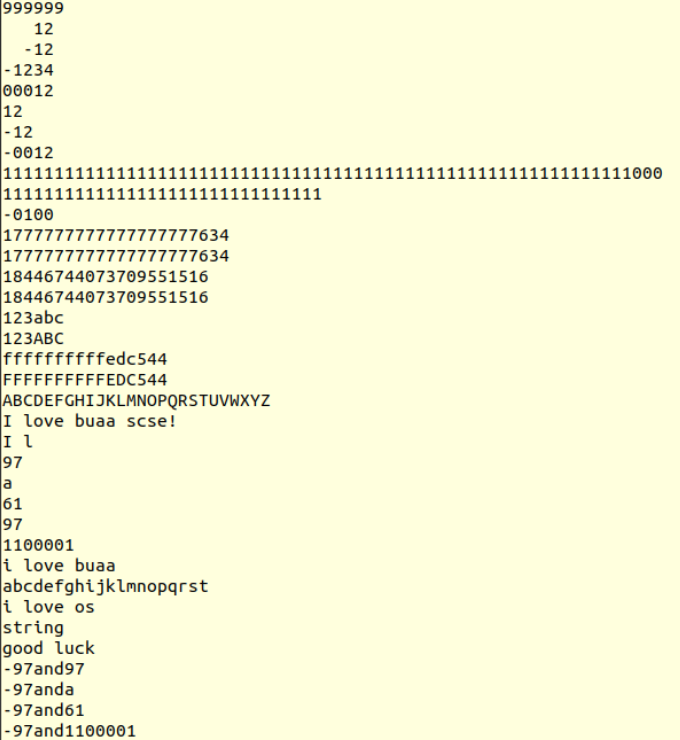
1. kern/machine.c:这个文件负责往 QEMU 的控制台输出字符，其原理为读写某一个特殊的内存地址。
2. kern/printk.c:此文件中，实现了printk，但其所做的，实际上是把输出字符的函数，接受的输出参数给传递给了 vprintfmt 这个函数。
3. lib/print.c:此文件中，实现了 vprintfmt 函数，其实现了格式化输出的主体逻辑。 接着为了便于理解，我们一起来梳理一下这几个文件之间的关系。kern/printk.c 中定义了我们的 printk 函数。仔细观察就可以发现，这个函数并没有直接实现输出，它只是把接受到的参数，以及outputk函数指针传入 vprintfmt 这个函数中。然后我们来看看 outputk 这个函数，这个函数实际上是用来输出一个字符串的，并且通过阅读可以发现，它实际上调用了一个叫做 printcharc 的函数。我们可以在 kern/machine.c 下面找到它的定义。最后可以发现，原来，想让控制台输出一个字符，实际上是对某一个内存地址写了一个字节。理解了以上之后，我们可以在 init.c 中直接调用 printcharc 来输出一个字符。

## 2.5 实验一正确结果

使用make && make run指令，若在qemu观察到输出

init.c: la32r\_init() is called

使用make test lab=1\_2 && make run指令，观察到现象和 tests/lab1\_2/init.c 中代码的输出一致，则代表顺利完成了Lab1实验。



# 3 实验二：内存管理

## 3.1 实验目的

1. 了解loongarch64的访存流程
2. 掌握与了解物理内存的管理方法（链表法）
3. 掌握与实现虚拟内存的管理方法（二级页表）
4. 掌握TLB清除与重填的流程

物理内存（Physical Memory）是计算机上重要的硬件资源之一，用于在计算机运行过程中存储代码和数据。la64支持对 56MB 物理内存进行申请和释放。内核的二进制可执行代码被加载到 0x200000 地址处。处理器中的内存管理单元（Memory Management Unit，MMU）将CPU访问的虚拟地址转换为物理地址。LoongArch架构中的MMU支持两种地址转换模式：直接地址翻译模式和映射地址翻译模式。其中，映射地址翻译模式又包括直接映射地址翻译模式（以下简称直接映射）和页表映射地址翻译模式（以下简称页表映射）。当 MMU 处于映射地址翻译模式时，会优先选择直接映射，若无法进行直接映射，则选择页表映射。la64采用映射地址翻译模式进行虚拟地址到物理地址的转换。在内核态下采用直接映射，即虚拟地址与物理地址是一一映射关系；在用户态下采用页表映射。在设计中，二级页表作为一种内核数据结构放置于内存中，在用户程序需要访问虚拟地址时候（通过中断机制）内核负责将对应的页表项填入TLB。在 Lab2 中，我们的核心任务是管理二级页表与填写TLB。通过二级页表保存和描述虚拟地址与物理地址的映射关系，通过TLB加速虚拟地址到物理地址的转换。在本次实验中，我们将学习为什么需要两种地址转换模式，并实现页表的初始化、页表项的填写、TLB的清除与重填等功能。

## 3.2 loongarch64的访存流程

### 3.2.1 CPU发出地址

CPU 运行程序时通过访存指令发出访存请求，进行内存读写操作。在计算机组成原理等硬件实验中，CPU 通常直接发送物理地址，这是为了简化内存操作，让大家关注CPU内部的计算与控制逻辑。而在实际程序中，访存、跳转等指令以及用于取指的PC寄存器中的访存目标地址都是虚拟地址。我们编写的C程序中也经常通过对指针解引用来进行访存，其中指针的值也会被视为虚拟地址，经过编译后生成相应的访存指令。

### 3.2.2 虚拟地址映射到物理地址

正常情况下，处理器中的内存管理单元将CPU访问的虚拟地址转换为物理地址采用的是页表的形式，通过页表将一个虚拟地址层层寻找到物理地址。而页表的初始化和创建发生在内核程序中，但内核程序的启动代码也需要被放置到物理地址中，所以此时需要绕过页表通过另一种方式来进行虚拟地址到物理地址的映射。Loongarch64提供了一种直接地址翻译模式，通过直接映射配置窗口机制完成虚实地址的直接映射。直接映射配置窗口共设置有四个，系统软件通过配置 CSR.DMW0~CSR.DMW3 寄存器来分别设置四个直接映射配置窗口。每个窗口除了地址范围信息外，还可以配置该窗口在哪些特权等级下可用，以及虚地址落在该窗口上的访存操作的存储访问类型。打开init/start.S，其中含有以下代码，即是直接映射窗口配置以及处理器核MMU的映射地址翻译模式配置。

01 li.d a0, 0x9000000000000011  
02 csrwr a0, csr\_dmw0  
   
03 li.d a0, 0x0000000000000000  
04 csrwr a0, csr\_dmw1  
05 csrwr a0, csr\_dmw2  
06 csrwr a0, csr\_dmw3  
   
07 li.d a0, 0xb0  
08 csrwr a0, csr\_crmd

init/start.S 文件在实验一中也有叫重点阅读过，以上节选部分就是关于内存管理的内容，下面将详细解读。以上代码的1、2行，将 DMW0 寄存器的值设置为 0x9000000000000001，从而将内核态下使用的虚拟地址到物理地址的地址转换模式配置为直接映射。依照图中的值，将 PLV0 字段设置为1，表示将特权级PLV0配置为直接映射；将 VSEG 字段设置为9，表示将直接映射窗口的虚拟地址的 [63:60] 位设置为9。这样，内核态下的虚拟地址空间0x9000000000000000～ 0x9000FFFFFFFFFFFF和物理地址空间0x0～0xFFFFFFFFFFFF就建立了一一映射。例如，在内核态下，CPU访问虚拟地址0x9000000000200000时，MMU将该虚拟地址转换为物理地址0x200000。现在回头看kernel.lds文件，即可知道为什么内核代码放置位置是在0x9000000000200000位置，却被写入了 0x200000 了。代码的9、10行，将 0xb0 写入 CSR.CRMD 中，其中 DA（第三位）= 0，PG（第四位）= 1，表示处理器核的 MMU 处于映射地址翻译模式，也就是采用直接映射地址翻译模式以及页表映射地址翻译模式。而当内核进行了页表的初始化以及创建之后，进入用户态之中，由于特权级的不同，并且用户程序代码存放位置大多不在直接映射窗口的虚拟地址空间范围之内，因此用户态采用页表映射地址翻译模式。

## 3.3 内核程序启动

Lab1已经成功跳转到 la32r\_init 函数，该函数实现位于 init/init.c 中。la32r\_init() 函数中需要分别调用如下三个函数：

* la32r\_detect\_memory():作用是探测硬件可用内存，并对一些内核内存管理相关的变量进行初始化。
* la32r\_vm\_init():作用是为内存管理机制作准备，建立一些用于管理的数据结构。
* page\_init():作用是初始化pages数组中的Page结构体以及空闲链表。 以上三个函数的实现都位于kern/pmap.c中，接下来的小节中将重点介绍前两个函数。

### 3.3.1 la32r\_detect\_memory

La32r\_detect\_memory()，实现位于kern/pmap.c，作用是探测硬件可用内存，并对一些和内存管理相关的变量进行初始化。在“开机”之后，操作系统首先会探测硬件的可用内存。bootloader 实际上已经帮助我们完成了这件事情，在启动时，系统可用物理内存大小参数通过 ram\_low\_size 被传入, 我们只需要直接使用即可。在该函数中初始化的变量包括 memsize，表示总物理内存对应的字节数，npage 表示总物理页数，其中的一些变量定义位于 include/mmu.h 中。

### 3.3.2 la32r\_vm\_init

la32r\_vm\_init()，实现位于 kern/pmap.c。在调用la32r\_detect\_memory()探测完可用内存后，将开始建立内存管理机制。为了建立内存管理机制，需要调用到 alloc 函数，此函数同样位于 kern/pmap.c 中，此时还没有页式内存管理机制，操作系统也需要建立一些数据结构来管理内存，这就会涉及到内存空间的分配。alloc函数的功能就是用于分配内存空间（在建立页式内存管理机制之前使用）。alloc 函数的内容请自行理解，其中函数开始第一行代码中 extern char end[] 是一个定义在该文件之外的变量，并不在一个C代码文件中，而是在Lab1重点要阅读的 kernel.lds 中：

. = 0x9000000001200000;  
end = . ;

也就是说该变量对应虚拟地址 0x9000000001200000，在建立内存管理机制时，本实验都是通过 DMW 访问内存。根据映射规则，0x9000000001200000 对应的物理地址是 0x1200000。 在物理地址 0x1200000 的前面，存放着操作系统内核的代码和定义的全局变量或数组（还额外保留了一些空间）。接下来将从物理地址 0x1200000 开始分配物理内存，用于建立管理内存的数据结构。回到 la32r\_vm\_init() 函数中。在函数中，完成了内存管理数据结构的空间分配。

## 3.4 物理内存管理

la64中的内存管理使用页式内存管理，采用链表法管理空闲物理页框。在实验中，内存管理的代码位于 kern/pmap.c 中。函数的声明位于 include/pmap.h 中。

### 3.4.1 链表宏

在后续的Lab中，也有一些地方需要用到链表，因此la64中使用宏对链表的操作进行了封装，这部分功能非常有用，设计技巧应用广泛，大家需要仔细阅读代码，深入理解。链表宏的定义位于 include/queue.h，其实现了双向链表功能，下面将对一些主要的宏进行解释:

* LIST\_HEAD(name, type)：创建一个名称为 name 链表的头部结构体，包含一个指向type 类型结构体的指针，这个指针可以指向链表的首个元素。
* LIST\_ENTRY(type)：作为一个特殊的类型出现，本质是一个链表项，包括指向下一个元素的指针 le\_next，以及指向前一个元素链表项 le\_next 的指针 le\_prev。le\_prev 是一个指针的指针，它的作用是当删除一个元素时，更改前一个元素链表项的 le\_next。
* LIST\_ENPTY(head)：判断head指针指向的头部结构体对应的链表是否为空。
* LIST\_FIRST(head)：将返回head对应的链表的首个元素。
* LIST\_INIT(head)：将head对应的链表初始化。
* LIST\_NEXT(elm, field)：返回指针elm指向的元素在对应链表中的下一个元素的指针。elm 指针指向的结构体需要包含一个名为 field 的字段，类型是一个链表项 LIST\_ENTRY(type)，下面出现的field含义均和此相同。
* LIST\_INSERT\_AFTER(listelm, elm, field)：将 elm 插到已有元素 listelm 之后。
* LIST\_INSERT\_BEFORE(listelm, elm, field)：将elm插到已有元素 listelm 之前。
* LIST\_INSERT\_HEAD(head, elm, field)：将elm插到head对应链表的头部。
* LIST\_REMOVE(elm, field)：将elm从对应链表中删除。

### 3.4.2 页控制块

la64中维护了 npage 个页控制块，也就是 Page 结构体。每一个页控制块对应一页的物理内存，la64用这个结构体来按页管理物理内存的分配。将空闲物理页对应的 Page 结构体全部插入一个链表中，该链表被称为空闲链表，即 page\_free\_list。当一个进程需要分配内存时，就需要将空闲链表头部的页控制块对应的那一页物理内存分配出去，同时将该页控制块从空闲链表中删去。当一页物理内存被使用完毕（准确来说，引用次数为0）时，将其对应的页控制块重新插入到空闲链表的头部。结合 Page 结构体来理解页控制块，它的定义位于 include/pmap.h 中。

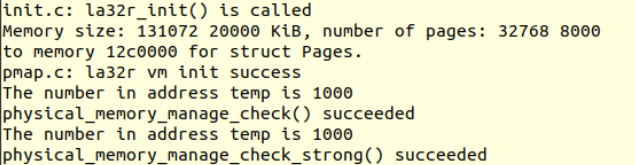
### 3.4.3 其他相关函数

在include/pmap.h中可以看到若干个以page\_开头的函数。其中，有关物理内存管理的函数有4个，分别用来初始化物理页面管理、分配物理页面、减少物理页面引用、回收物理页面到空闲页面链表。它们的实现均位于 kern/pmap.c 中：

* page\_init()：启动过程首先初始化了空闲页面链表，用于存储“没有被使用”的页控制块。需要申请内存空间时，就可以从没有被使用的页面中分配一页来使用。第二步按照页面对齐分配内存，接下接下来，将使用过的页面标记上引用次数，表示该页面被引用1次（可以通过已建立的页控制块数组访问）。将未被使用过的页面的页控制块加入空闲页面链表，供后续分配。
* page\_alloc(struct Page \*\*pp)：它的作用是将 page\_free\_list 空闲链表头部页控制块对应的物理页面分配出去，将其从空闲链表中移除，并清空此页中的数据，最后将pp指向的空间赋值为这个页控制块的地址。
* page\_decref(struct Page \*pp)：作用是令pp对应页控制块的引用次数减少1，如果引用次数为0则会调用 page\_free 函数将对应物理页面重新设置为空闲页面。
* page\_free(struct Page \*pp)：它的作用是将pp指向的页控制块重新插入到 page\_free\_list 中。此外需要先确保 pp 指向的页控制块对应的物理页面引用次数为0。

### 3.4.4 实验结果展示

执行 make test lab=2\_1 && make run 编译运行内核，出现如下输出即通过物理内存管理的单元测试：

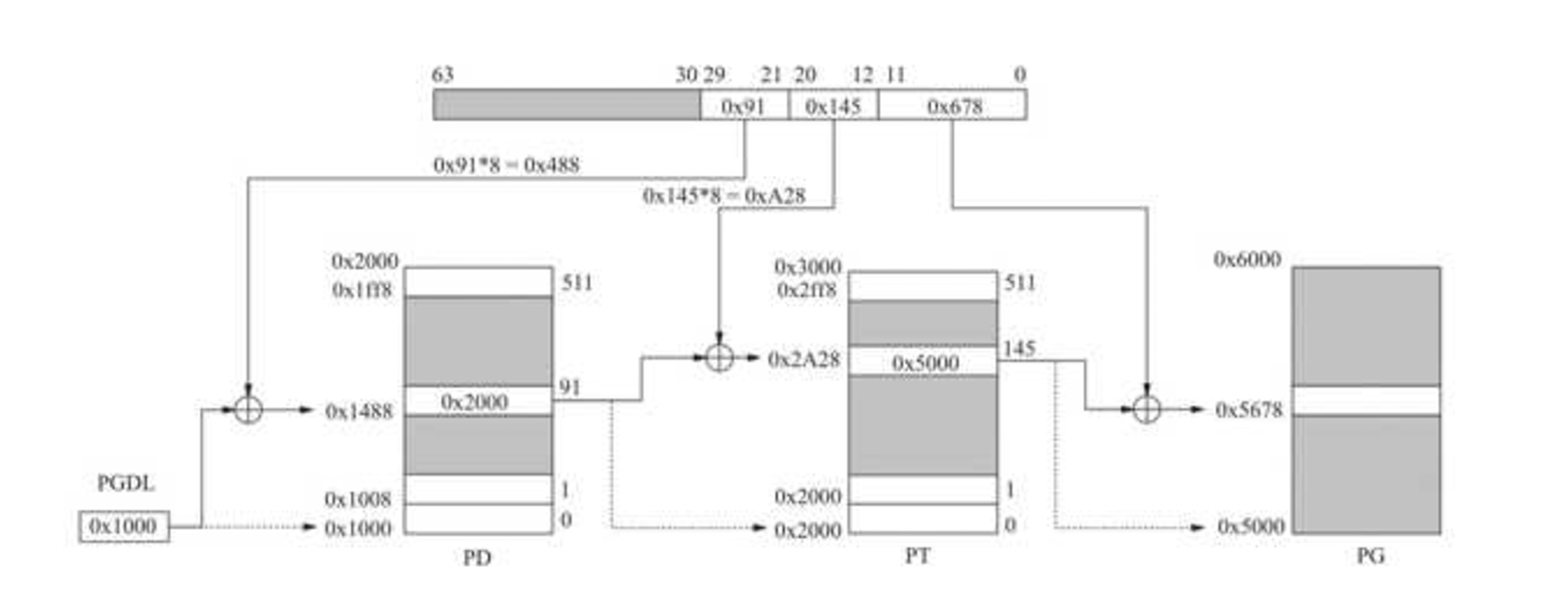


## 3.5 虚拟内存管理

通过阅读前面的代码，la64中使用 PADDR 与 KADDR 这两个宏可以对位于kseg0 （也就是dmw0范围内）的虚拟地址和对应的物理地址进行转换。但是，对于位于 kuseg （dmw0范围之外）的虚拟地址，la64中采用二级页表结构对其进行地址转换。

### 3.5.1 二级页表结构

在二级页表结构中，第一级表称为页目录 (Page Directory) ，第二级表称为页表 (Page Table) 。为避免歧义，下面用一级页表指代 Page Directory，二级页表指代 Page Table。二级页表机制相比单极页表机制，将虚拟页号分为了两部分。具体来说，对于一个32位的虚拟地址，从低到高从0开始编号，其29~21位表示的是一级页表项的偏移量，20~12 位表示的是二级页表项的偏移量，11~0位表示的是页内偏移量。include/mmu.h 中提供了两个宏以快速获取偏移量，PDX(va) 可以获取虚拟地址 va 的 30~21 位，PTX(va) 可以获取虚拟地址 va 的 20~12 位。访问虚拟地址时，先通过一级页表基地址和一级页表项的偏移量，找到对应的一级页表项，得到对应的二级页表的物理页号，再根据二级页表项的偏移量找到所需的二级页表项，进而得到该虚拟地址对应的物理页号。la64 发出的地址均为虚拟地址，因此如果程序想访问某个物理地址，需要通过映射到该物理地址的虚拟地址来访问。对页表进行操作时硬件处于内核态，因此使用宏 KADDR 获得其位于 kseg0 中的虚拟地址即可完成转换。下面以将虚拟地址 0x12345678 转换为物理地址 0x5678 为例，详细介绍 MMU 利用二级页表结构进行地址转换的过程。虚拟地址 0x12345678 的页目录项索引、页表项索引和页内偏移分别为 0x91、0x145 和 0x678。地址转换过程中涉及的页目录、页表和物理页的起始物理地址分别为 0x1000、0x2000 和 0x5000，这3个物理地址分别存放在 PGDL 寄存器、0x91号PDE和 0x145 号 PTE 中。MMU 的转换过程如下：



1. MMU 从 PGDL 寄存器中获取页目录的起始物理地址 0x1000，将其与 0x91 号页目录项的偏移 0x488 相加，得到页目录项的物理地址 0x1488。
2. MMU 通过物理地址 0x1488 获取页目录项中页表的起始物理地址 0x2000 ，将其与 0x145 号页表项的偏移 0xA28 相加，得到页表项的物理地址 0x2A28。
3. MMU 通过物理地址 0x2A28 获取页表项中物理页的起始物理地址 0x5000，将其与页内偏移 0x678 相加，最终得到虚拟地址 0x12345678 对应的物理地址 0x5678。可能会有人不理解为什么每次从虚拟地址中取出索引量后都要乘以8，原因是la64的地址是64位的，每一个偏移量需要占8个字节，因此相当于乘上8。在la64中，无论是一级页表还是二级页表，它们的结构都一样，只不过每个页表项记录的物理页号含义有所不同。每个页表均由512个页表项组成，每个页表项由30位组成，包括18位物理页号以及12位标志位。详细解释如下：

* 第0位，存在位（v），或者叫做有效位，如果是0，说明此页表项不存在，任何匹配此表项的地址都会引起TLBL/TLBS异常。
* 第1位，脏位（D），如果是0，则表示存储在TLB中中的页表没被改写过，如果为1，则代表被改写了，需要写回内存。
* 第2和第3位，特权等级位（PLV），在 la64 中仅 PLV0 级可对应核心态，以 PLV3 级对应用户态。
* 第4和第5位，存储访问类型位（MAT），0表示强序非缓存，1表示一致可缓存，2表示弱序非缓存，3保留。
* 第6位，全局标志位（G），执行 TLBFILL 和 TLBWR 指令时，仅当 TLBELO0 和 TLBELO1 中的 G 位均为1时，填入到 TLB 中的页表项的 G 位才为1。执行 TLBRD 指令时，当所读取的 TLB 表项的 G 位为1，则 TLBELO0 和 TLBELO1 中 的 G 位被同时置为1。
* 第7位，页标志位(P)，表示页表项为页。
* 第8位，可写位(W)，表示写入保护，如果为0则不可写，如果为1则允许写入。
* 第9到第11位，保留位，只读恒为0，写被忽略。

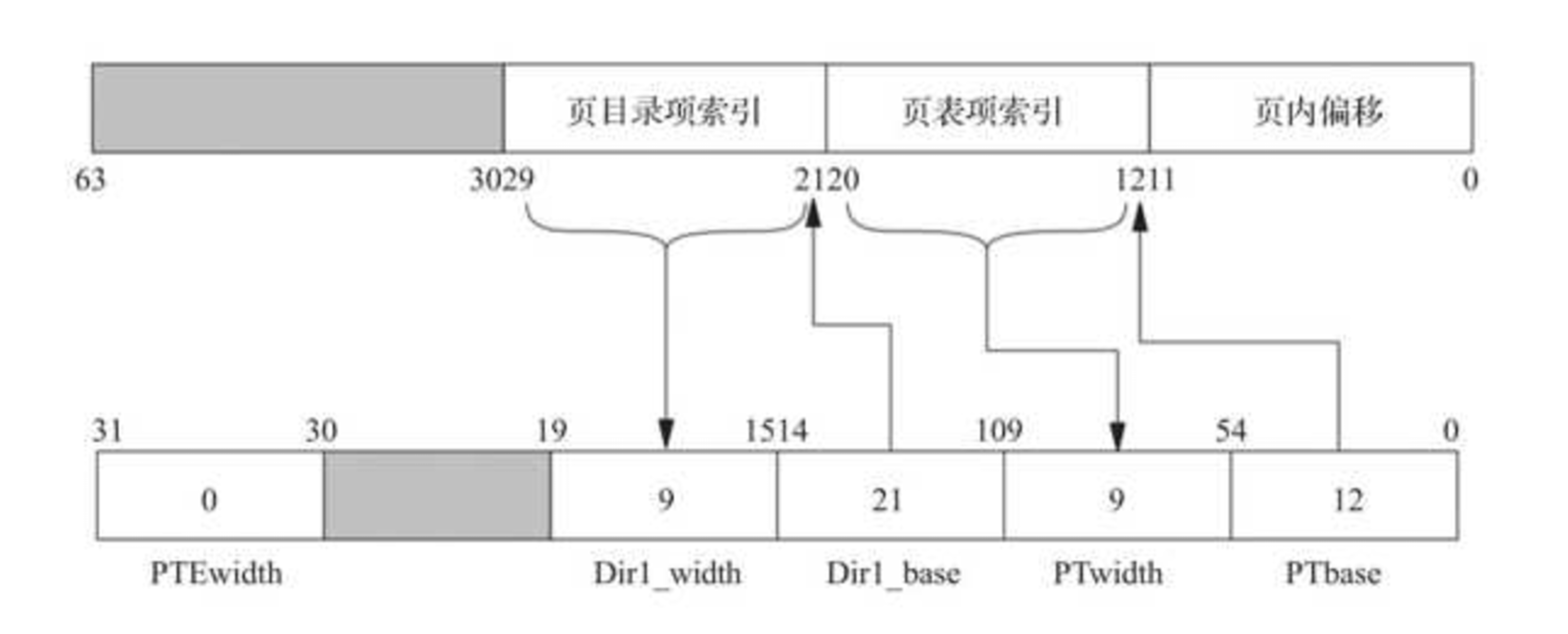
这里再复习一下 TLB 的功能。la64 中，通过页表进行地址变换时，硬件只会查询 TLB， 如果查找失败，就会触发 TLBMiss（TLB 缺失）异常，对应的异常中断处理程序就会对 TLB 进行重填。大家需要了解 loongarch64 虚拟地址映射的原理。需要特别注意的是，Lab2 中没有开启异常处理功能，我们仅对 TLB 重填过程进行模拟测试。这部分 TLB 相关的详细内容会在下面介绍。使用 tlb\_invalidate 函数可以实现删除特定虚拟地址的映射，每当页表被修改，就需要调用该函数以保证下次访问相应虚拟地址时一定触发 TLB 重填，进而保证访存的正确性。除此之外，一般的操作系统中，当物理页全部被映射（所有内存空间均被占用），此时还需要申请新的物理页，那么就需要将一些在内存中的物理页置换到硬盘中，选择哪个物理页的算法就称为页面置换算法，例如 FIFO 算法和 LRU 算法。然而在 la64 中，这一过程的实现被简化，一旦物理页全部被分配，进行新的物理页分配时并不会进行任何的页面置换，而是直接返回错误，即在对应 page\_alloc 函数中返回 -E\_NO\_MEM。

### 3.5.2 与页表相关的函数

下面将介绍几个与页表相关的函数： **页表初始化**void page\_init(void)，之前介绍过 page\_init 函数的功能，现在将目光聚焦到该函数的最后两行代码。

01 asm volatile("csrwr %0, 0x1c" : : "r" ((PTEWIDTH << 30)|(DIR2WIDTH << 25)|(DIR2BASE << 20)|(DIR1WIDTH << 15)|(DIR1BASE << 10)|(PTWIDTH << 5)|(PTBASE << 0)) );   
02 asm volatile("invtlb 0x0,$r0,$r0");

从功能上不难看出，第一行代码作用是将一些宏定义数值写入一个寄存器中，这些宏定义具体的值可以在 include/asm/regdef.h 进行查看和更改。通过查看手册可以知晓，0x1c 代表的是 PWCL 寄存器，因为我们使用的仅仅是二级页表，所以只需要通过将页表信息写入 PWCL 寄存器的方式来实现对二级页表结构的初始化，而不用涉及到PWCH寄存器。如第一行代码所示，将 PWCL 寄存器的 PTbase 字段设置为12（PWCL\_PBASE），表示虚拟地址中的页表项索引从第12位开始；将 PWCL 寄存器的 PTwidth 字段设置为9（PWCL\_PTWIDTH），表示虚拟地址中的页表项索引的长度为9，即页内偏移位于虚拟地址的第0～11位，页表项索引位于虚拟地址的第12～20位；将 PWCL 寄存器的 Dir1\_base 字段设置为21（PWCL\_PDBASE），表示虚拟地址中的页目录项索引从第21位开始；将 PWCL 寄存器的 Dir1\_width 字段设置为9（PWCL\_PDWIDH），表示虚拟地址中的页目录项索引的长度为9，即页目录项索引位于虚拟地址的第21～29位。因此，在 la64 中，进程地址空间大小为29×29×212=1GB。将 PWCL 寄存器的 PTEwidth 字段设置为 0（PWCL\_EWIDTH），表示页表项的大小为8B。具体对应如下图所示：



page\_init 函数的最后一行代码是在对用户态下使用的页表映射初始化完成后刷新 TLB。**二级页表检索函数**int pgdir\_walk(Pde \*pgdir, u\_long va, int create, Pte \*\*ppte)，该函数将一级页表基地址 pgdir 对应的两级页表结构中 va 虚拟地址所在的二级页表项的指针存储在 ppte 指向的空间上。如果 create 不为0且对应的二级页表不存在，则会使用 page\_alloc 函数分配一页物理内存用于存放二级页表，如果分配失败则返回错误码。**增加地址映射函数**int page\_insert(Pde \*pgdir, u\_int asid, struct Page \*pp, u\_longva, u\_intperm)，作用是将一级页表基地址 pgdir 对应的两级页表结构中虚拟地址 va 映射到页控制块 pp 对应的物理页面，并将页表项权限为设置为 perm。通过阅读可知，该函数实际上是调用了 pgdir\_walk 函数，第一次调用查找 va 是否存在了映射，如果存在映射，该映射是否有效，如果存在的映射是有效的，那现有映射的页和需要插入的页是否一样，如果不一样，那么先移除现有的映射，后续再插入新的页，如果一样，就只需要更新一下映射的权限位。如果不存在映射，那么第二次调用 pgdir\_walk 函数，并将 creat 传入参数为1，表示分配一页物理内存用于存放二级页表。**寻找映射的物理地址函数**struct Page \*page\_lookup(Pde \*pgdir, u\_long va, Pte \*\*ppte)，作用是返回一级页表基地址 pgdir 对应的二级页表页表结构中虚拟地址 va 映射的物理界面的页控制块，同时将 ppte 指向的空间设为对应的二级页表项地址。**取消地址映射函数**void page\_remove(Pde \*pgdir, u\_int asid, u\_long va)，作用是删除一级页表基地址 pgdir 对应的两级页表结构中虚拟地址 va 对物理地址的映射。如果存在这样的映射，那么对应物理页面的引用次数会减少一次。

## 3.6 访问内存与TLB重填

### 3.6.1 TLB相关的前置知识

**TLB组成**每个TLB表项都有两个组成部分，包括一组Key和两组Data。**TLBELO0、TLBELO1、TLBEHI**TLBELO0、TLBELO1、TLBEHI都是寄存器，他们只是分别对应到 TLB 的 Key 与两组 Data，并不是 TLB 本身。因龙芯架构下 TLB 采用双页结构，所以 TLB 表项的低位信息对应奇偶两个物理页表项，其中偶数页信息在 TLBELO0 中，奇数页信息在 TLBELO1 中，而 TLBELO0 和 TLBELO1 寄存器的格式定义完全相同。

### 3.6.2 TLB维护流程

通过之前的实验，大家可能仍然对软件访问内存的过程有所疑惑，这是由于还未涉及到用户进程相关的内容，所有代码、数据的虚拟地址均在dmw0段，无需通过页表的翻译便可直接获得其物理地址。因此本次实验中所完成的代码，大多是为之后的实验提供接口，本次实验中的一些内存管理功能只作为独立的函数存在。但由于内存访问将是之后实验中很重要的内容，在此次实验结束时，有必要将用户进程访问内存的流程解释清楚，既能帮助之后的实验，也能加深对本次实验的理解。本实验所使用的 MMU 硬件中只有 TLB，在用户地址空间访存时，虚拟地址到物理地址的转换均通过 TLB 进行。访问需要经过转换的虚拟内存地址时，首先要使用虚拟页号和当前进程的 ASID 在 TLB 中查询该地址对应的物理页号，如果虚页号和 ASID 组成的 Key 在 TLB 中存在对应的 TLB 表项（或虚页号在TLB中存在对应的 TLB 表项且表项权限位中的 G 位为1）时，则可取得物理地址；如果不能查询到，则产生 TLBMiss 异常，系统跳转到异常处理程序，在内核的两级页表结构中找到对应的物理地址，对 TLB 进行重填。操作系统可以修改页表中虚拟地址映射的物理页号或映射的权限位。如果 TLB 中已暂存了页表中某一虚拟地址对应的页表项内容，之后操作系统更新了该页表项，但没有更新 TLB，则访问该虚拟地址时实际可能会访问到错误的物理页面。所以我们需要维护 TLB 的表项，使得当 TLB 能够查询到虚拟地址相应的页号时，取得的物理页号和权限信息与实际在内核页表中对应的数据一致。因此，我们可以得到维护TLB的具体流程：

1. 更新页表中虚拟地址对应的页表项的同时，将TLB中对应的旧表项无效化。
2. 在下一次访问该虚拟地址时，硬件会触发 TLB 重填异常，此时操作系统对 TLB 进行重填。

**TLB旧表项无效化**我们通过位于 kern/tlbex.c 中的 do\_tlb\_invalid 函数实现删除特定虚拟地址在 TLB 中的旧表项，函数的主要逻辑依靠位于 kern/tlb\_asm.S 中的 tlb\_invalidate函数。当 tlb\_invalidate 被 do\_tlb\_invalid 调用时, a0 和 a1 寄存器中存放着传入的参数，其值为旧表项的 Key（由虚拟页号和 ASID 组成）。随后使用 invtlb 指令无效化 TLB 中的内容。**TLB重填**TLB的重填过程由 kern/tlb\_asm.S 中的 tlb\_miss\_entry 函数完成，具体代码如下：

01 tlb\_miss\_entry:  
02   
03 csrwr t0, csr\_tlbrsave  
04 csrrd t0, csr\_pgd  
05 lddir t0, t0, 1  
06 srli.d t0,t0,12  
07 slli.d t0,t0,12  
08 ldpte t0, 0  
09 ldpte t0, 1  
10 tlbfill  
11 csrrd t0, csr\_tlbrsave  
12   
13 ertn  
14   
15 .size tlb\_miss\_entry, .-tlb\_miss\_entry

首先保存 t0 寄存器的值，因为在 tlb\_miss\_entry 函数中使用了该寄存器。接着使用 csrrd 汇编指令，将 PGD 寄存器中保存的触发 TLB 重填例外的进程的页目录起始物理地址加载到 t0 寄存器中。随后使用 lddir 汇编指令，遍历页目录中触发 TLB 重填例外的虚拟地址对应的页目录项，并将该页目录项指向的页表起始物理地址加载到 t0 寄存器中。下面先使用右移指令，随后使用左移指令，每次位移位数相同，作用是将 t0 寄存器中的低12位清零。然后使用 ldpte 汇编指令，将页表中触发 TLB 重填例外的虚拟地址对应的2个页表项的内容写入 TLBRELO0 寄存器和 TLBRELO1 寄存器。后面使用 tlbfill 汇编指令，将 TLBRELO0 寄存器和 TLBRELO1 寄存器中存放的页表项信息填入到 TLB 中。最后恢复 t0 寄存器的值后，例外返回。中断、异常处理等过程将是之后实验的重点，现在可暂且将其理解为代码的跳转，到本节实验为止只需明白软件代码在访问内存时的处理过程即可。

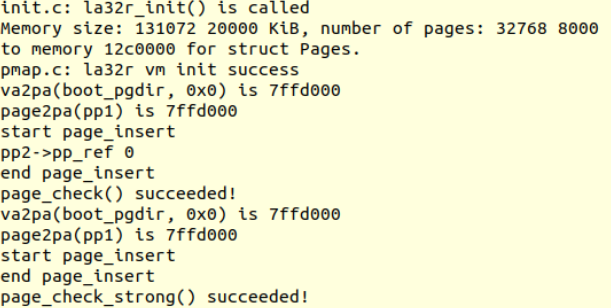
## 3.7 实验正确结果

### 3.7.1 实验代码功能解释

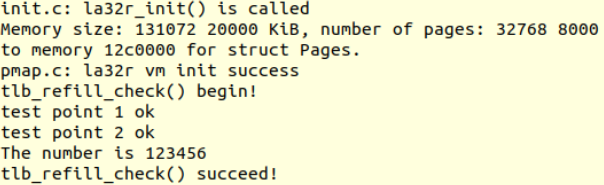
大家需要自己仔细阅读测试代码，充分理解测试代码都做了什么，给本章学习做一个好的收尾。**lab2\_2**代码整体上是一个对物理页管理与页表映射机制进行强一致性、边界条件和引用计数验证的自检流程，用于在 la64 架构的内核初始化阶段严格校验 page\_alloc / page\_free / page\_insert / page\_remove 等核心内存管理原语是否实现正确。具体而言，page\_strong\_check() 先分配一页作为页目录并连续分配多页物理页，验证基本分配的唯一性；随后通过“暂时夺取”空闲页链表来人为制造内存耗尽场景，检查在无空闲页时页表分配与映射是否能正确失败，并在释放单个物理页后确认该页会被优先用作页表页。接着，该函数系统性地测试多种映射与重映射情况，包括同一物理页被映射到多个虚拟地址、重复插入同一映射、不同行为下引用计数的增减是否准确、页表页是否被正确创建和复用，以及在移除映射后物理页是否能被正确回收并重新分配。最后，它强制回收页目录和页表页，恢复空闲页链表并释放所有临时占用的页面，确保系统状态完全可逆、无资源泄漏。la32r\_init() 则在系统启动时依次完成内存探测、虚拟内存初始化和页管理初始化，并通过先后调用 page\_check() 与 page\_strong\_check() 来验证内存子系统的正确性，若全部断言通过则表明分页与物理页管理实现是健壮且符合设计预期的。**lab2\_3**代码整体实现了一个针对 la64 架构 TLB refill 与缺页异常处理路径的功能性自检，用于在内核初始化阶段验证页表建立、TLB 失效处理以及缺页时按需分配物理页的机制是否正确工作。代码首先分配并初始化一个新的页目录，将其同时写入相关 CSR 以作为当前活动页表基址，然后分配若干物理页并刻意清空空闲页链表，构造受控的内存环境；在已有页表页的前提下，测试基本的页映射与 page\_lookup、va2pa 的一致性。随后，通过人为访问尚未建立映射的虚拟地址，显式调用 do\_tlb\_invalid 来模拟 TLB 无效或缺页异常，验证异常处理程序能够正确创建页表项、分配新的物理页并建立映射关系。最后，代码通过对新映射地址执行真实的读写操作，确认 TLB refill 机制已经生效、页表项被正确加载进 TLB 且内存访问行为符合预期。整体而言，该测试覆盖了从页表初始化、TLB 失效处理到缺页后可读写验证的完整路径，确保虚拟内存子系统在运行时具备正确的按需映射与 TLB 同步能力。

### 3.7.2 正确实验结果展示

在完成内存管理的学习后，执行 make test lab=2\_2 && make run 编译运行内核，显示如下图所示信息即通过内存管理的单元测试（其中地址7ffd000在不同实现下的值可能不同），也就是虚拟内存管理实验：



执行 make test lab=2\_3 && make run 编译运行内核，显示如下图所示信息即通过内存管理的 TLB 重填实验测试。



# 4 实验三：进程与异常

## 4.1 实验目的

1. 创造一个进程并成功运行。
2. 实现时钟中断，通过时钟中断内核可以再次获得执行权。
3. 实现进程调度，创建两个进程，并且通过时钟中断切换进程执行。

在Lab3 中将运行一个用户模式的进程。本实验需要使用数据结构进程控制块 Env 来跟踪用户进程，并建立一个简单的用户进程，加载一个程序镜像到指定的内存空间，然后让它运行起来。同时，实验实现的 la64 内核具有处理异常的能力。

## 4.2 进程

整个 la64 并没有实现线程，因此本实验中进程既是基本的分配单元，也是基本的执行单元。每个进程都是一个实体，有其自己的地址空间，通常包括代码段、数据段和堆栈。程序是一个没有生命的实体，只有被操作系统赋予生命时，它才能成为一个活动的实体，而执行中的程序，就是进程。

### 4.2.1 进程控制块

进程控制块(PCB)是系统专门设置用来管理进程的数据结构，它可以记录进程的外部特征，描述进程的变化过程。系统利用 PCB 来控制和管理进程，所以 PCB 是系统感知进程存在的唯一标志。进程与 PCB 是一一对应的。在 la64 中，PCB 由一个 Env 结构体实现，主要包含如下一些信息：

// Control block of an environment (process).  
struct Env {  
 struct Trapframe env\_tf; // saved context (registers) before switching  
 LIST\_ENTRY(Env) env\_link; // intrusive entry **in** 'env\_free\_list'  
 u\_long env\_id; // unique environment identifier  
 u\_long env\_asid; // ASID of this env  
 u\_long env\_parent\_id; // env\_id of this env's parent  
 u\_long env\_status; // status of this env  
 Pde \*env\_pgdir; // page directory  
 TAILQ\_ENTRY(Env) env\_sched\_link; // intrusive entry **in** 'env\_sched\_list'  
 u\_long env\_pri; // schedule priority  
  
 // Lab 4 IPC  
 u\_long env\_ipc\_value; // the value sent to us  
 u\_long env\_ipc\_from; // envid of the sender  
 u\_long env\_ipc\_recving; // whether this env **is** blocked receiving  
 u\_long env\_ipc\_dstva; // va at which the received page should be mapped  
 u\_long env\_ipc\_perm; // perm **in** which the received page should be mapped  
  
 // Lab 4 fault handling  
 u\_long env\_user\_tlb\_mod\_entry; // userspace TLB Mod handler  
  
 // Lab 6 scheduler counts  
 u\_long env\_runs; // number of times we've been env\_run'ed  
};

我们只对Lab3相关部分进行介绍，剩下的自行结合注释阅读：

* env\_tf : 用于保存当前进程的上下文信息对应的结构体 Trapframe，该结构体的定义在 include/trap.h 中，在发生进程调度，或当陷入内核时，会将当时的进程上下文环境保存在 env\_tf 变量中。
* env\_link : env\_link 的机制类似于 Lab2 中的 pp\_link，使用它来构造空闲进程链表 env\_free\_list。
* env\_id : 每个进程的 env\_id 都不一样，它是进程独一无二的标识符。
* env\_parent\_id : 在之后的实验中，我们将了解到进程是可以被其他进程创建的，创建本进程的进程称为父进程。此变量记录父进程的进程 id，进程之间通过此关联可以形成一棵进程树。
* env\_status : 该字段只能有以下三种取值：-ENV\_FREE : 表明该进程控制块处于空闲状态，其没有被任何进程使用，即该进程控制块处于进程空闲链表中。-ENV\_NOT\_RUNNABLE : 表明该进程处于阻塞状态，处于该状态的进程需要在一定条件下变成就绪状态从而被 CPU 调度。（比如因进程通信阻塞时变为 ENV\_NOT\_RUNNABLE， 收到信息后变回 ENV\_RUNNABLE）。-ENV\_RUNNABLE : 该进程处于执行状态或就绪状态，即其可能是正在运行的，也可能正在等待被调度。
* env\_pgdir : 这个字段保存了该进程页目录的内核虚拟地址。
* env\_sched\_link : 这个字段用来构造调度队列 env\_sched\_list。
* env\_pri : 这个字段保存了该进程的优先级。

在实验中，存放进程控制块的物理内存在系统启动后就已经分配好，就是 envs 数组。和 Lab 2对页控制块数组的管理类似，我们使用链表管理进程控制块数组。struct Env 中的链表项共涉及调度队列 env\_sched\_list 和空闲队列 env\_free\_list 两个队列：

* env\_sched\_list : 为了便于管理已经被分配的进程控制块和对进程的调度，我们用链表将这些控制块串联起来，形成 env\_sched\_list。在进程创建时需要为其分配进程控制块 并加入 env\_sched\_list，在进程被释放时需要将其对应的进程控制块从 env\_sched\_list 移出。
* env\_free\_list : 为了快速分配空闲的进程控制块，我们需要像 Lab2 一样将空闲的 Env 控制块按照链表形式串联起来，形成 env\_free\_list。一开始所有进程控制块都是空闲的，所以要把它们都串联到 env\_free\_list 上去。

其中 env\_free\_list 使用了 Lab2 中涉及的 LIST 结构，而 env\_sched\_list 则使用了相似但不同的另一种结构 TAILQ，它同样定义在 include/queue.h 中，实现了一个双端队列，既支持在头部插入和取出，也支持在尾部插入和取出。你需要使用以 TAILQ 开头的宏来操作该队列。

### 4.2.2 段地址映射

在 env\_init 函数的最后，使用 page\_alloc 函数为模板页表 base\_pgdir 分配了一页物理内存，将其转换为内核虚拟地址，并使用 map\_segment 函数在该页表中将内核数组 pages 和 envs 映射到了用户空间的 UPAGES 和 UENVS 处。在之后的 env\_setup\_vm 函数中，我们会将这部分模板页表复制到每个进程的页表中。段地址映射函数 void map\_segment(Pde \*pgdir, u\_int asid, u\_long pa, u\_long va, u\_long size, u\_int perm) ，功能是在一级页表基地址 pgdir 对应的两级页表结构中做段地址映射，将虚拟地址段 [va,va+size) 映射到物理地址段 [pa,pa+size) ，因为是按页映射，要求 size 必须是页面大小的整数倍。同时为相关页表项的权限为设置为 perm。它在这里的作用是将内核中的 Page 和 Env 数据结构映射到用户地址，以供用户程序读取。

### 4.2.3 进程的标识

现代计算机系统中经常有很多进程同时存在，每个进程执行不同的任务，它们之间也经常需要相互协作、通信，那么操作系统是如何识别每个进程呢？在本实验中我们依靠进程标识符来实现这一点。struct Env 进程控制块中的 env\_id 域，是每个进程独一无二的标识符，需要在进程创建的时候就被赋予。env.c 文件中实现了一个叫做 mkenvid 的函数，作用就是生成一个新的进程env\_id。la64 系统的进程控制块，还有一个 env\_asid 域记录进程的 ASID，这是进程虚拟地址空间的标识。env\_id 已经可以唯一标识进程，包括进程虚拟地址空间，为什么 la64 还需要引入额外的域作为虚拟地址空间的标识？系统中并发执行多个拥有不同虚拟地址空间的进程，具有不同的页表。而 CPU 的 MMU 使用 TLB 缓存虚拟地址映射关系，不同页表拥有不同虚拟地址映射，显然，不同进程的虚拟地址 可以对应相同的虚拟页号。当 CPU 切换页表，TLB 中仍可能缓存有之前页表的虚拟地址映射关系，这些映射关系可能与当前页表所描述的不一致，可以将其称为无效映射关系。为了避免 TLB 缓存的无效映射关系在页表切换后导致错误的地址翻译发生，早期操作系统实现在 CPU 每次切换页表时，无效化所有 TLB 表项。但这种实现会导致频繁的 TLB Miss，影响处理器性能。现代的 CPU 及操作系统，采用 ASID 解决上述问题。ASID 用于标识虚拟地址空间，同时并发执行的多个进程具有不同 ASID 以方便 TLB 标识其虚拟地址空间。正如 Lab2 中提到，TLB 实质上构建了一个 <VPN, ASID>—><PFN, P, D, V, G> 的映射。这里的 VPN 表示虚拟页号，PFN 表示物理页框号。TLB 存储进程的 ASID，作为 Key 的一部分，用于区别不同的虚拟地址空间中的映射。相当于每一个进程，都有自己虚拟地址空间（使用 ASID标识）下的一套独立 TLB 缓存，每次切换页表后，操作系统不必再清空所有 TLB 表项。由于 ASID 的唯一标识性，在实验中直到进程被销毁或 TLB 被清空时，才可以把其 ASID 分配给其他进程。

### 4.2.4 设置进程控制块

了解以上基础知识后，可以开始利用空闲进程链表 env\_free\_list 创建进程。下面具体介绍如何创建一个进程。进程创建的流程如下：

1. 申请一个空闲的 PCB（也就是Env结构体），从 env\_free\_list 中索取一个空闲 PCB 块，这时候的 PCB 就像张白纸一样。
2. “纯手工打造”打造一个进程。在这种创建方式下，由于没有模板进程，所以进程拥有的所有信息都是手工设置。而进程的信息又都存放于进程控制块中，所以需要手工初始化进程控制块。
3. 进程光有 PCB 的信息还没法跑起来，每个进程都有独立的地址空间。所以，要为新进程初始化页目录。
4. 此时 PCB 已经被填写了很多东西，不再是一张白纸，把它从空闲链表里拿出，就可以使用。

第二步的信息设置是本次实验的关键，阅读 kern/env.c 中的 env\_alloc 函数，env\_setup\_vm 函数，我们需要初始化的字段已经在 env\_alloc 函数的注释中给出，这个函数的重点在于我们已经给出的这个赋值 e->env\_tf.crmd = CRMD\_IE | CRMD\_PLV。通过查看手册，可以知道 IE 位表示中断是否开启，为1表示开启，否则不开启。我们需要将 IE 设置为1，表示中断使能。此外，将 CRMD\_PLV 写入 tf 中，因为每次触发中断后，程序会自动进入特权模式处理异常，设置 PLV 后，使用 ertn 指令可以回到用户模式中。之后第一个进程成功以用户模式运行，这时操作系统也可以正常响应中断了。

### 4.2.5 加载二进制镜像

在这一小节中，我们需要将程序加载到新进程的地址空间中。在 Lab1 中我们曾详细学习了 ELF 文件，它的类型有三种，一是可重定位文件，二是可执行文件，三是可被共享的对象文件。本节中的程序是指可执行文件。要想正确加载一个 ELF 文件到内存，只需将 ELF 文件中所有需要加载的程序段（program segment）加载到对应的虚拟地址上即可。我们已经写好了用于解析 ELF 文件的代码中的内容，你可以直接调用相应函数获取 ELF 文件的各项信息，并完成加载过程。相关函数和类型的声明如下：

// lib/elfloader.c  
const Elf64\_Ehdr \*elf\_from\_64b(const void \*binary, size\_t size);  
int elf\_load\_seg(Elf64\_Phdr \*ph, const void \*bin, elf\_mapper\_t map\_page, void \*data);  
  
// kern/env.c  
static void load\_icode(struct Env \*e, const void \*binary, size\_t size);  
static int load\_icode\_mapper(void \*data, u\_long va, size\_t offset, u\_long perm, const void \*src, size\_t len);

在 Lab3 阶段，我们还没有实现文件系统，因此无法直接操作磁盘中的 ELF 文件。在这里我们已经将 ELF 文件内容转化成了C数组的形式（可以在make后查看 user/bare/loop.b.c文件），这样可以通过编译到内核中完成加载。load\_icode 函数负责加载可执行文件 binary 到进程 e 的内存中。它调用的 elf\_from 函数完成了解析 ELF 文件头的部分，elf\_load\_seg 负责将 ELF 文件的一个 segment 加载到内存。为了达到这一目标，elf\_load\_seg 的最后两个参数用于接受一个自定义的回调函数 map\_page，以及需要传递给回调函数的额外参数 data。每当 elf\_load\_seg 函数解析到一个需要加载到内存中的页面，会将有关的信息作为参数传递给回调函数，并由它完成单个页面的加载过程，而这里 load\_icode\_mapper 就是 map\_page 的具体实现。load\_icode 函数会从 ELF 文件中解析出每个 segment 的段头 ph，以及其数据在内存中的起始位置 bin，再由 elf\_load\_seg 函数将参数指定的程序段（program segment）加载到进程的地址空间中。elf\_load\_seg 函数会从 ph 中获取 va（该段需要被加载到的虚地址）、sgsize（该段在内存中的大小）、bin\_size（该段在文件中的大小）和 perm（该段被加载时的页面权限），并根据这些信息完成以下两个步骤：

1. 加载该段的所有数据（bin）中的所有内容到内存（va）。
2. 如果该段在文件中的内容的大小达不到为填入这段内容新分配的页面大小，即分配了新的页面但没能填满（如.bss区域），那么余下的部分用 0 来填充。

load\_icode 函数通过调用 elf\_load\_seg 函数来将 ELF 文件真正加载到内存中，将 load\_icode\_mapper 这个函数作为参数传入。在该函数中出现了 e->env\_tf.era = ehdr->e\_entry 字段，该字段指示了进程恢复运行时 PC 应恢复到的位置。我们要运行的进程的代码段预先被载入到了内存中，且程序入口为 e\_entry，当我们运行进程时，CPU 将自动从 PC 所指的位置开始执行二进制码。

### 4.2.6 创建进程

这里需要指出，“创建进程”是指在操作系统内核初始化时直接创建进程，而不是在通过 fork 等系统调用来创建进程。在 Lab4 中将介绍 fork 这一种进程创建的方式。创建进程的过程很简单，就是实现对上述个别函数的封装，分配一个新的 Env 结构体，设置进程控制块，并将程序载入到目标进程的地址空间即可完成。在 include/env.h 中，可以看到两个宏定义函数，#define ENV\_CREATE\_PRIORITY(x, y) 以及 #define ENV\_CREATE(x) ，两个函数将二进制 ELF 文件的起始地址以及 ELF 文件大小传入 env\_create 函数中，并可选定优先级。最后，可以在 init/init.c 中去掉注释，增加下面两句代码，在内核初始化时创建两个进程。

ENV\_CREATE\_PRIORITY(user\_bare\_loop, 1);   
ENV\_CREATE\_PRIORITY(user\_bare\_loop, 2);

这里的 user\_bare\_loop 用于变量命名，对应的用户程序位于 user/bare/loop.S。经过 ENV\_CREATE 宏的拼接后，得到内核中的 binary\_user\_bare\_loop\_start 数组和 binary\_user\_bare\_loop\_size 变量，我们可以在 make 时构建出的user/bare/loop.b.c文件中找到它们的定义。在创建进程前，记得调用 env\_init 初始化进程管理。

### 4.2.7 进程运行与切换

在 kern/env.c 中可以找到 env\_run 函数，该函数是进程运行使用的基本函数，它包括两部分：

* 保存当前进程上下文（如果当前没有运行的进程就跳过这一步）
* 恢复要启动的进程的上下文，然后运行该进程。

其实我们这里运行一个新进程往往意味着是进程切换，而不是单纯的进程运行。进程切换，顾名思义，就是当前进程停下工作，让出 CPU 来运行另外的进程。那么要理解进程切换，我们就要知道进程切换时系统需要做些什么。实际上进程切换的时候，为了保证下一次进入这个进程的时候我们不会再“从头来过”，而是有记忆地从离开的地方继续往后走，我们要保存一些信息， 那么，需要保存什么信息呢？事实上，我们只需要保存进程的上下文信息，包括通用寄存器、HI、 LO和 CRMD、BadVAddr 寄存器。进程控制块除了env\_tf其他的字段在进程切换后还保留在原本的进程控制块中，并不会改变，因此不需要保存。在 Lab3 中，我们在本实验里的寄存器状态保存的地方是 KSTACKTOP 以下的一个 sizeof(TrapFrame) 大小的区域中。

curenv->env\_tf = \*((structTrapframe \*)KSTACKTOP-1)

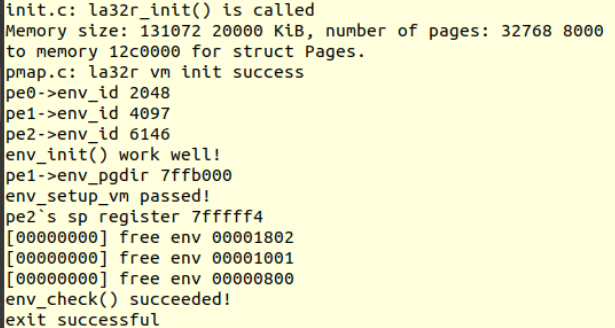
代码中的 curenv->env\_tf 就是当前进程的上下文所存放的区域。我们将把 KSTACKTOP 之下的 Trapframe 拷贝到当前进程的 env\_tf 中，以达到保存进程上下文的效果。总结以上说明，我们不难看出 env\_run 的执行流程：

1. 保存当前进程的上下文信息。
2. 切换curenv为即将运行的进程。
3. 设置全局变量 cur\_pgdir 为当前进程页目录地址，在 TLB 重填时将用到该全局变量。
4. 调用 env\_pop\_tf 函数，恢复现场、异常返回。

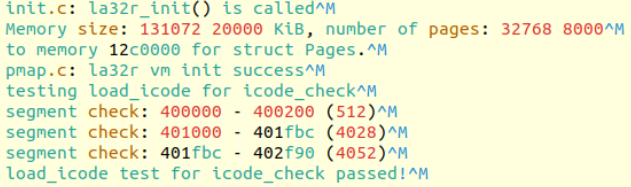
这里用到的 env\_pop\_tf 是定义在 kern/env\_asm.S 中的一个汇编函数，建议大家去自己阅读并理解其含义。这个函数也呼应了我们前文提到的，进程每次被调度运行前一定会执行的 ertn 汇编指令。至此，第一部分学习已经完成。

### 4.2.8 实验正确结果

大家要自行阅读和理解测试代码，搞清楚测试代码中的函数调用关系，理解函数功能以及作用。执行 make test lab=3\_1 && make run 编译运行内核，显示如下信息即通过 env\_init 的单元测试：

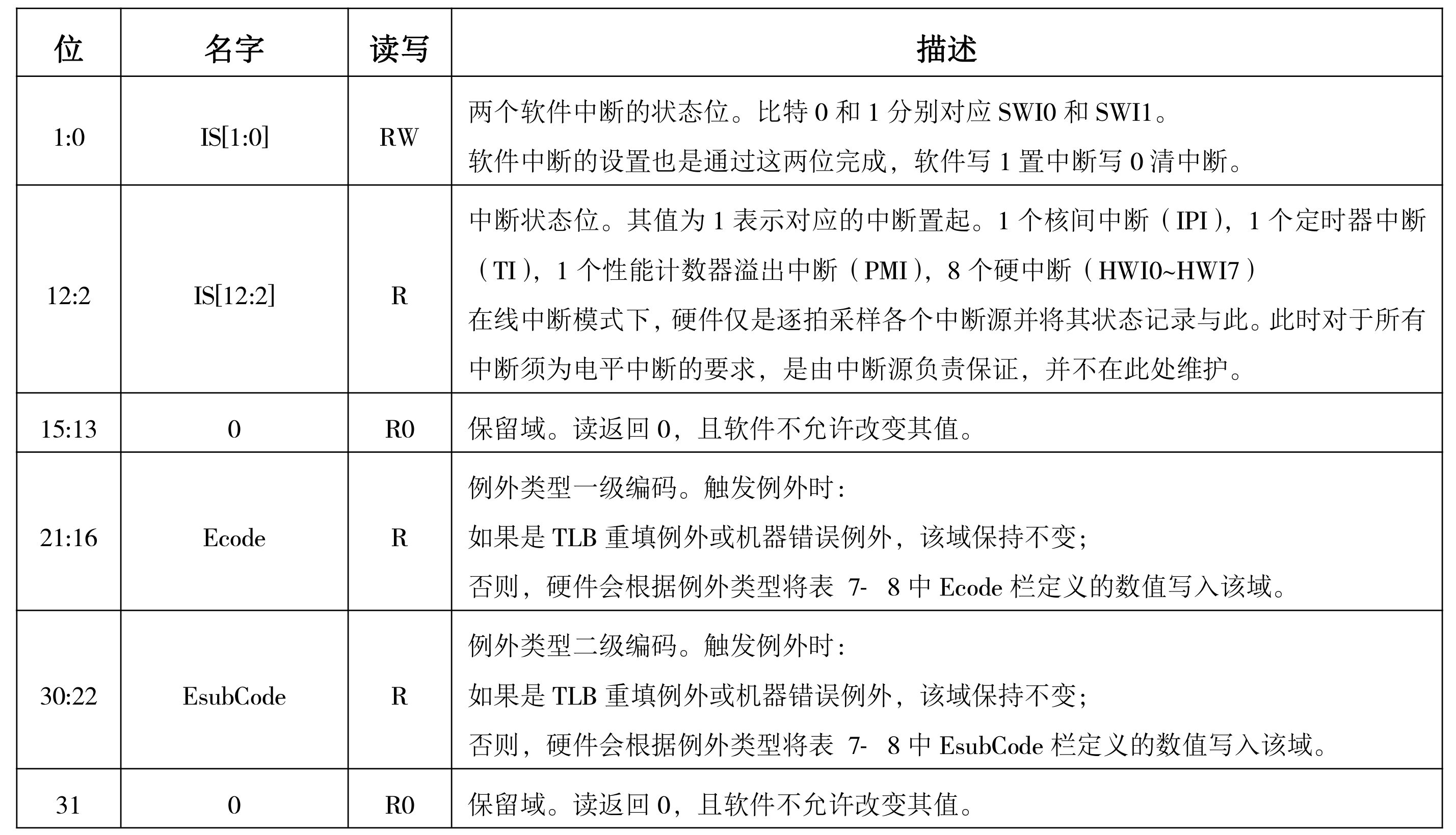


执行 make test lab=3\_2 && make run 编译运行内核,显示如下信息即通过 load\_icode 的单元测试：



## 4.3 中断与例外

龙芯架构下的中断采用线中断的形式。每个处理器核内部可记录13个线中断，分别是：1个核间中断（IPI），1 个定时器中断（TI），1个性能监测计数溢出中断（PMI），8个硬中断（HWI0~HWI7），2个软中断（SWI0~SWI1）。所有的线中断都是电平中断，且都是高电平有效。在龙芯架构下，控制状态寄存器中的 0x5，ESTAT 例外状态寄存器，记录了中断状态位以及例外类型编码，如下图所示：



由图表可知，寄存器的第0~13位记录了发生的中断情况，第16~21位则记录了发生的例外情况，其中如果 Ecode 值为0x0，则表示为中断，这也就是两者的联系。我们实验里认为中断是例外的一种，并且是仅有的一种异步例外（请读者自行去了解同步和异步的区别）。la64处理一个例外时大致要完成四项工作：

1. 设置 ERA，指向从异常返回的地址。
2. 设置 CRMD，强制 CPU 进入内核态（行使更高级的特权）并禁止中断和例外。
3. 设置 ESTAT 例外状态寄存器，用于记录中断以及例外发生的原因。
4. CPU 开始从异常入口位置取指，此后一切交给软件处理。

而这个“一切交给软件处理”，就是我们操作系统要完成的事情。

### 4.3.1 例外的分发

当发生例外时，处理器会进入一个用于分发例外的程序，这个程序的作用就是检测发生了哪种例外，并调用相应的例外处理程序。一般来说，例外分发程序会被要求放在固定的某个物理地址上（根据处理器的区别有所不同），以保证处理器能在检测到异常时正确地跳转到那里。这个分发程序可以认为是操作系统的一部分。下述代码就是例外分发代码，处在 kern/entry.S 中，下面分析其功能。

exc\_gen\_entry:  
 SAVE\_ALL  
 csrrd t0, csr\_estat  
 srli.d t0, t0, 16  
 andi t0, t0, 0x3f  
 slli.d t0, t0, 3  
 la t1, exception\_handlers  
 add.d t1, t0, t1  
 ld.d t0, t1, 0  
 jr t0

这段例外分发代码的作用流程如下：

1. 使用 SAVE\_ALL 宏将当前上下文保存到内核的异常栈中。
2. 将 ESTAT 寄存器的内容拷贝到t0寄存器中。
3. 取得 ESTAT 寄存器的16~21位，也就是对应的例外码，这是区别不同例外的重要标志。
4. 以取得的例外码作为索引在 exception\_handlers 数组中找到对应的中断处理函数，后文中会有涉及。
5. 跳转到对应的例外处理函数中，也就是响应了例外，并将例外交给了对应的例外处理函数去处理。

这里出现了一个新的宏 SAVE\_ALL，该宏在后面的 Lab 中也会使用，用于将当前的 CPU 现场（上下文）保存到内核的异常栈中，我们可以在 include/stackframe.h 文件中查看这个宏的定义。在 SAVE\_ALL 的执行过程中，我们首先判断当前异常是否是在内核态触发的，即是否发生了例外重入。具体而言，代码首先通过 csrrd 指令将 csr\_prmd（例外前的处理器模式寄存器）的值读入临时寄存器 $r21（即 x， 定义位于 include/asm/regdef.h 中），并使用 andi 指令提取其最低两位，用于判断例外发生前的特权级。如果该值为 0，则表示例外是在内核态触发的，属于例外重入的情况。对于例外重入的情况，此时 sp 寄存器已经指向当前的内核异常栈，因此无需切换栈指针。代码将当前的 sp 保存到即将构建的 trapframe 中对应的 TF\_REG3 位置，随后继续执行后续保存现场的流程。而对于非例外重入的情况，即例外是在用户态触发的，sp 此时仍指向用户栈。代码会先将 sp 切换为内核例外栈顶 KSTACKTOP - TF\_SIZE，并在切换之前，将原用户态的 sp 值保存到 trapframe 中的 TF\_REG3 字段，从而保证用户栈指针不会丢失。完成例外重入判断与栈指针处理后，代码继续将关键的控制状态寄存器（包括 csr\_prmd、csr\_estat、csr\_era 和 csr\_badv）以及所有通用寄存器依次保存到 trapframe 中，为后续的异常处理与上下文恢复提供完整的处理器状态。.text.exc\_gen\_entry 段和 .text.tlb\_miss\_entry 段需要被链接器放到特定的位置，它们是异常处理程序的入口地址。在我们的系统中，CPU 发生例外（除了用户态地址的 TLBMiss 例外）后，就会自动跳转到 exc\_gen\_entry 函数所在地址处；发生用户态地址的 TLBMiss 例外时，会自动跳转到 tlb\_miss\_entry 函数所在地址处。开始执行。我们可以在 kernel.lds 中看到这两个函数是如何被放置到特定位置的，其具体实现位于 kern/entry.S 以及 kern/tlb\_asm.S，这两个函数为操作系统增加了例外分发功能。

### 4.3.2 例外向量组

例外分发程序通过 exception\_handlers 数组定位中断处理程序，而 exception\_handlers 就称作例外向量组。下面我们来看一下 kern/traps.c 中的 exception\_handlers 数组，来了解异常向量组里存放了什么。

extern void handle\_int(void);  
extern void handle\_tlb(void);  
extern void handle\_sys(void);  
extern void handle\_mod(void);  
extern void handle\_reserved(void);  
  
void (\*exception\_handlers[64])(void) = {  
 [0 ... 63] = handle\_reserved, [0] = handle\_int, [1 ... 3] = handle\_tlb,  
*#if !defined(LAB) || LAB >= 4*  
 [0x4] = handle\_mod, [0xb] = handle\_sys,  
*#endif*  
};

上述代码使用了 GNU C 的拓展语法 [first ... last] = value 来对数组某个区间上的元素赋成同一个值。举例来说，

int arr[5] = { [0 ... 3] = 1, [2 ... 4] = 2 };

这一句等价于下面的代码：

int arr[5] = { 1, 1, 2, 2, 2 };

通过把相应处理函数的地址填到对应数组项中，我们初始化了如下例外：**0号例外** 的处理函数为 **handle\_int**，表示中断，由时钟中断、控制台中断等中断造成**1号例外** 的处理函数为 **handle\_mod**，表示存储例外，进行存储操作时该页被标记为只读**2号例外** 的处理函数为 **handle\_tlb**，表示 TLB load 例外**3号例外** 的处理函数为 **handle\_tlb**，表示 TLB store 例外**8号例外** 的处理函数为 **handle\_sys**，表示系统调用，用户进程通过执行 syscall 指令陷入内核一旦初始化结束，有例外产生，那么其对应的处理函数就会得到执行。而我们在本 Lab 中，主要使用0号例外，即中断例外的处理函数（对于2、3号异常，在Lab2的相关内容中有所介绍）。而我们接下来要做的，就是产生并处理时钟中断，利用时钟中断进行抢占式进程调度。

### 4.3.3 时钟中断

在前面的介绍中我们已经知道 ESTAT 寄存器中有11个独立的中断位。其中8位是硬件中断，另外3位是软件中断，且不同中断处理起来也会有差异。所以在完成这一部分内容之前，我们首先来介绍一下中断处理的流程。

1. 通过例外分发，判断出当前异常为中断例外，随后进入相应的中断处理程序。在 la64 中即对应 handle\_int 函数。
2. 在中断处理程序中进一步判断 ESTAT 寄存器中是由几号中断位引发的中断，然后进入不同中断对应的中断服务函数。
3. 中断处理完成，通过 ret\_from\_exception 函数恢复现场，继续执行。

上面涉及到的函数定义在 kern/genex.S 中。下面我们来简单介绍一下时钟中断的概念，为什么 ‘la64’ 系统需要时钟中断。时钟中断与 ‘la64’ 系统的时间片轮转调度算法是紧密相关的。时间片轮转调度是一种进程调度算法，每个进程被分配一个时间段，称作它的时间片，即该进程允许运行的时间。如果在时间片结束时进程还在运行，则该进程将挂起，切换到另一个进程运行。如果该进程在时间片结束前阻塞或者结束，则立即切换到另一个进程运行。‘la64’ 是如何知晓一个进程的时间片结束的呢？就是通过硬件定时器产生的时钟中断。在 ‘la64’ 中，时间片的长度是用时钟中断衡量的。比如设定某个进程的时间片的长度为200倍的 ‘TIMER\_INTERVAL’（时钟中断间隔），那么当 ‘la64’ 记录到该进程的执行中发生了200个时钟中断时，’la64’ 就知晓该进程的时间片结束了。当时钟中断产生时，当前运行的进程被挂起，’la64’ 需要在调度队列中选取一个合适的进程运行。‘LoongArch’ 架构提供了用于产生时钟中断的恒定频率定时器（以下简称定时器）。在使用定时器前，需要为其设定1个自减初始值，定时器会按照固定频率进行自减操作，当自减到0时，产生时钟中断信号。’include/kclock.h’ 中的 ‘RESET\_KCLOCK’ 宏完成了对定时器的配置。下面将对时钟初始化相关函数进行解释，首先看到 ‘start.S’ 函数中的以下代码：

li.d t0, ESTAT\_TI  
 csrwr t0, csr\_ectl

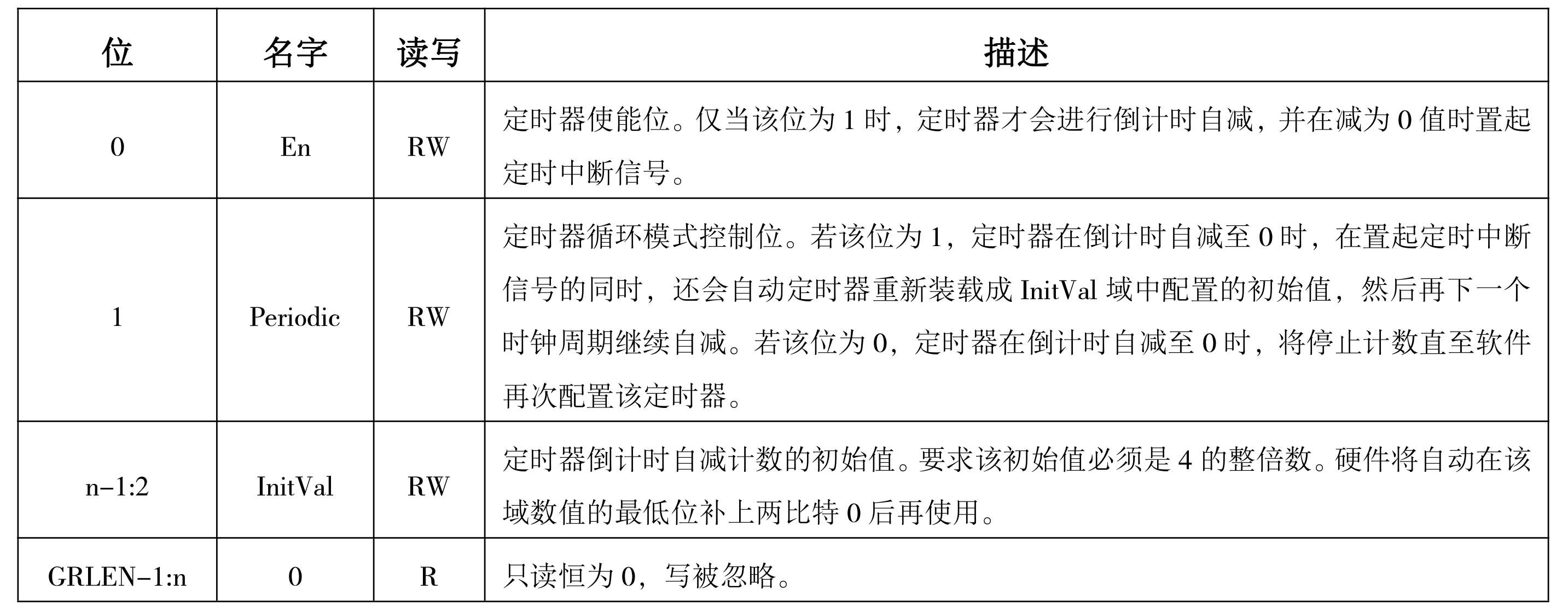
然后是 ‘include/kclock.h’ 中的代码：

*#define TIMER\_CONFIG (0xfffd)*  
.macro RESET\_KCLOCK  
 li.d t0, TIMER\_CONFIG  
 csrwr t0, csr\_tcfg  
.endm

实现时钟中断的使能需要配置2个寄存器：’ECFG’ 寄存器和 ‘CRMD’ 寄存器。其中，’ECFG’ 寄存器用于使能局部中断，’CRMD’ 寄存器用于使能全局中断。如前所述LoongArch架构支持13个中断，每个中断都有一个局部中断使能位，总共13位，分别对应 ‘ECFG’ 寄存器的 ‘LIE’ 字段中的13位，其中，定时器中断对应第11位。而通过写 ‘TCFG’ 寄存器中的字段，就可以进行定时器的配置。 ‘TCFG’寄存器需要配置的字段包括定时器倒计时自减的初始值 ‘InitVal’、循环模式控制位 ‘Periodic’ 和定时器使能位 ‘EN’。

* InitVal：定时器倒计时自减计数的初始值。硬件将该值左移2位后的值作为最终的定时器的自减初始值。例如，若将 ‘InitVal’ 字段设置为1，则定时器的自减初始值为4，即定时器在自减4次后，产生时钟中断信号。
* Periodic：循环模式控制位。若该位为1，则定时器在倒计时自减至0时，在产生时钟中断信号的同时，将定时器的自减初始值重新加载为InitVal字段中的值左移2位后的值，然后在下一个时钟周期重新自减。若该位为0，则自减至0时，在产生定时器中断信号后停止自减。
* EN：定时器使能位。只有当该位为1时，定时器才会进行倒计时自减，并在减至0时，产生定时器中断信号。

详细位数如下图所示:



总的来说，RESET\_KCLOCK 宏完成了对定时器的配置。在设定好的时钟周期后，时钟中断将被触发。la64 中，时钟中断的初始化发生在调度执行每一个进程之前。从代码角度，就是在 env\_pop\_tf 中调用了宏 RESET\_KCLOCK，随后又在宏 RESTORE\_ALL 中恢复了 CRMD 寄存器，开启了中断。一旦时钟中断产生，就会触发硬件的异常中断处理流程。系统将 PC 指向 exc\_gen\_entry 函数所在地址，跳转到 .text.exc\_gen\_entry 代码段执行。对于时钟引起的中断，通过 .text.exc\_gen\_entry 代码段的分发，最终会调用 handle\_int 函数进行处理。handle\_int 函数根据 ESTAT 寄存器的值判断是否是 Timer 对应的11号中断位引发的时钟中断，如果是，则执行中断服务函数 timer\_irq ，跳转到 schedule 中执行。这个函数就是我们将要学习的调度函数。

### 4.3.4 进程调度

handle\_int 函数的最后跳转到了 schedule 函数。这个函数在 kern/sched.c 中所定义，它就是我们本次实验最后要学习的调度函数。调度的算法很简单，就是时间片轮转的算法。上文提到，la64 中的时间片的长度是用时钟中断衡量的，即时间片长度被量化为 N × TIMER\_INTERVAL。具体的，env 中的优先级即为这里的 N，规定了该进程的时间片长度。不过寻找就绪状态进程不是简单遍历所有进程，而是用一个调度链表存储所有就绪（可运行）的进程，即一个进程在调度链表中当且仅当这个进程是就绪（ENV\_RUNNABLE）状态的。当内核创建新进程时，将其插入调度链表的头部；在其不再就绪（被阻塞）或退出时，将其从调度链表中移除。调度函数 schedule 被调用时，当前正在运行的进程被存储在全局变量 curenv 中（在第一个进程被调度前为NULL），其剩余的时间片长度被存储在静态变量 count 中。我们考虑是否需要进行进程切换，包括以下几种情况：

* 尚未调度过任何进程（curenv为空指针）；
* 当前进程已经用完了时间片；
* 当前进程不再就绪（如被阻塞或退出）；
* yield 参数指定必须发生切换。

无需进行切换时，我们只需要将剩余时间片长度 count 减去1，然后调用 env\_run 函数，继续运行当前进程 curenv。在发生切换的情况下，我们还需要判断当前进程是否仍然就绪，如果是则将其移动到调度链表的尾部。之后，我们选中调度链表首部的进程来调度运行，将剩余时间片长度设置为其优先级。至此，我们的Lab3就算是圆满完成了。

## 4.4 代码导读

Lab3 涉及到了内核态以及用户态的切换，还有不同进程寄存器之间的切换，需要大家仔细学习、理解，为了帮助理清 Lab3 代码逻辑和执行流程，我们在这里给出代码导读部分。Lab3 在 Lab2 基础上增加了两个代码段，叫做 .tlb\_miss\_entry 和 .exc\_gen\_entry，其代码定义在 kern/entry.S，实现了分发处理异常的过程，在异常的分发小节有所详述，这里不再赘述。下面将对 Lab3 中的代码进行详细的分析：

1. env\_run(1). 将正在执行的进程（如果有）的现场保存到对应的进程控制块中。(2). 选择一个可以运行的进程，恢复该进程上次被挂起时候的现场。这个过程主要通过 env\_pop\_tf 来完成，而 env\_pop\_tf 调用了 ret\_from\_exception 来完成从异常返回的过程。env\_pop\_tf 函数的部分内容如下：

LEAF(env\_pop\_tf)  
csrwr a1, csr\_asid  
move sp, a0  
RESET\_KCLOCK  
b ret\_from\_exception  
END(env\_pop\_tf)

我们在 env.c 中将进程控制块的 env\_tf.crmd 初始化为 CRMD\_IE | CRMD\_PLV。这样设置是为了让 la64 可以正常响应时钟中断，从而可以在时钟中断发生时，由异常分发程序调用 handle\_int 函数，经过 timer\_irq、schedule、env\_run来完成进程调度。

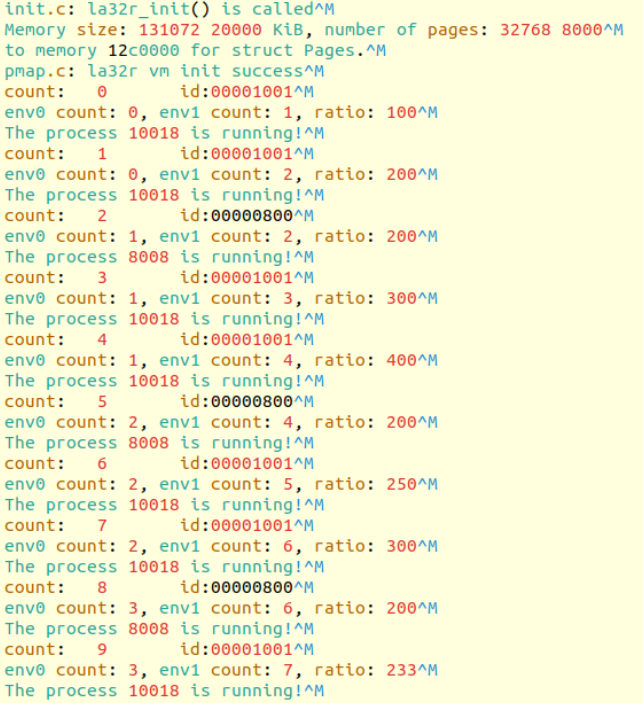
1. ret\_from\_exception从名字中就可以看出，这个函数作用是用例外返回，其具体实现如下：

ret\_from\_exception:  
 RESTORE\_SOME  
 ld.d sp, sp, TF\_REG3 /\* Deallocate stack \*/  
 ertn

首先是将保存到内核初始化栈上的中断现场恢复到对应的寄存器中，然后是恢复栈指针，将不同进程对应的栈指针加载到使用的栈指针中，最后是调用 ertn 指令，从例外处理返回。3. ertnERTN 指令用于从例外处理返回。如果所处理的例外是 Debug 例外，将 CSR.DBG 中的 DS 位清0，同时跳转到 CSR.DERA 所存放的地址处开始取指。如果所处理的例外是 Debug 例外之外的其它例外，将例外对应的 PPLV、PIE、PWE 等信息更新至 CSR.CRMD 中，同时跳转到例外所对应的返回地址处开始取指。如果所处理的例外是 Error 类例外，例外对应的 PPLV、PIE 和 PWE 信息来自于 CSR.MERRCTL，例外对应的返回地址来自于 CSR.MERRERA。除此之外，Error 类例外还要将 CSR.MERRCTL 中的 PDA、PPG、PDCAF、PDCAM 信息更新到 CSR.CRMD 中。如果所处理的例外是 TLB 重填例外，例外对应的 PPLV、PIE 和 PWE 信息来自于 CSR.TLBRSAVE，例外对应的返回地址来自于 CSR.TLBRERA。除此之外，还要将 CSR.CRMD 中的 DA 位清0、PG 位置1。如果所处理的例外不是 Debug 例外、Error 类例外和 TLB 重填例外，那么例外对应的 PPLV、PIE 和 PWE 信息来自于 CSR.PRMD，例外对应的返回地址来自于 CSR.ERA。执行 ERTN 指令时，如果 CSR.LLBCTL 中的 KLO 位不等于1，则将 LLbit 置0，否则 LLbit 不修改。

## 4.5 实验正确结果

执行 CFLAGS=-DMOS\_SCHED\_MAX\_TICKS=100 make 或 make test lab=3\_3 构建内核，再执行 make run，显示如下信息即通过 Lab3 的测试：



# 5 实验四：系统调用与FORK

## 5.1 实验目的

1. 掌握系统调用的概念及流程
2. 实现进程间通信机制
3. 实现 fork 函数
4. 掌握页写入异常的处理流程

在用户态下，用户进程不能访问系统的内核空间，也就是说它一般不能存取内核使用的内存数据，也不能调用内核函数，这一点是由体系结构保证的。然而，用户进程在特定的场景下往往需要执行一些只能由内核完成的操作，如操作硬件、动态分配内存，以及与其他进程进行通信等。允许在内核态执行用户程序提供的代码显然是不安全的，因此操作系统设计了一系列内核空间中的函数，当用户进程需要进行这些操作时，会引发特定的异常以陷入内核态，由内核调用对应的函数，从而安全地为用户进程提供受限的系统级操作，我们把这种机制称为系统调用。在 Lab4 中，我们需要实现上述的系统调用机制，并在此基础上实现进程间通信（IPC）机制和一个重要的进程创建机制 fork。在 fork 部分的实验中，我们会介绍一种被称为写时复制（COW）的特性，以及与其相关的页写入异常处理。

## 5.2 系统调用(System Call)

本节中，我们着重讨论系统调用的作用，并完成其实现。

### 5.2.1 用户态与内核态

在之前各个 Lab 的学习中，相信大家对用户态、用户空间、内核态等概念已经不陌生了。随着 ‘la64’ 操作系统功能的不断完善，我们也需要扩充实现完整的用户态机制，这就包括操作系统中用户进程与内核进行通信的关键机制—系统调用。我们首先回顾以下几组概念：

1. 用户态和内核态（也称用户模式和内核模式）：它们是 CPU 运行的两种状态。根据 Lab3 的说明，在 loongarch 架构中，该状态由 CRMD 寄存器中 PLV 位的值标志。
2. 用户空间和内核空间：它们是虚拟内存（进程的地址空间）中的两部分区域。根据 Lab2 的说明，la64 中的用户空间包括 kuseg，而内核空间主要包括 kseg1 。每个进程的用户空间通常通过页表映射到不同的物理页，而内核空间则直接映射到固定的物理页1以及外部硬件设备。CPU 在内核态下可以访问任何内存区域，对物理内存等硬件设备有完整的控制权，而在用户态下则只能访问用户空间。
3. （用户）进程和内核：进程是资源分配与调度的基本单位，拥有独立的地址空间，而内核负责管理系统资源和调度进程，使进程能够并发运行。与前两对概念不同，进程和内核并不是对立的存在，可以认为内核是存在于所有进程地址空间中的一段代码。

### 5.2.2 系统调用实例

puts() 函数是 C 标准库中用于输出一个字符串的函数，我们以其在 Linux 下的执行过程为例，展示其完整的调用过程如下：

puts函数完整调用过程

|  |  |
| --- | --- |
| Step1 | 用户调用 puts 函数 |
| Step2 | 在一系列的函数调用后，最终调用了 write 函数 |
| Step3 | write 函数在寄存器中设置了对应的系统调用号以及相应的参数，并执行了 syscall 指令 |
| Step4 | 进入内核态，内核中相应的函数或服务被执行 |
| Step5 | 回到用户态的 write 函数中，将结果从相关的寄存器中取回，并返回 |
| Step6 | 再次经过一系列的返回过程后，回到了 puts函数中 |
| Step7 | puts 函数返回 |

在 loongarch 中，syscall 是一条用于执行系统调用的自陷指令，它使得进程陷入到内核的异常处理程序中，由内核根据系统调用时的上下文执行相应的内核函数，完成相应的功能，并最终返回到 syscall 的后一条指令。从以上步骤，我们可以看到：

1. 存在一些只能由内核来完成的操作（如读写设备、创建进程、IO 等）。
2. C 标准库中一些函数的实现须依赖于操作系统（如我们所探究的 puts 函数）。
3. 通过执行 syscall 指令，用户进程可以陷入到内核态，请求内核提供的服务。
4. 通过系统调用陷入到内核态时，需要在用户态与内核态之间进行数据传递与保护。

综合以上内容，我们可以知道，内核将自己所能够提供的服务以系统调用的方式提供给用户空间，以供用户程序完成一些特殊的系统级操作。由于用户程序只能将服务相关的参数交予操作系统由操作系统执行，这样就保证了所有的特殊操作受操作系统掌控，而这些操作实际执行时也将由内核进行重重检查，因此系统调用保证了系统的安全性。进一步，由于直接使用这些系统调用较为麻烦，于是产生了一系列用户空间的 API 定义，如 POSIX 和 C 标准库等，它们在系统调用的基础上，实现更多更高级的常用功能，使得用户在编写程序时不用再处理这些繁琐而复杂的底层操作，而是直接通过调用高层次的 API 就能实现各种功能。通过这样巧妙的层次划分，使得程序更为灵活，也具有了更好的可移植性。对于用户程序来说，只要自己所依赖的 API 不变，无论底层的系统调用如何变化，都不会对自己造成影响，更易于在不同的系统间移植。整个结构如下表所示。

API、系统调用层次结构

|  |  |
| --- | --- |
| 用户程序 User Program |  |
| 应用程序编程接口 API | POSIX, CStandard Library, etc. |
| 系统调用 | read, write, etc. |

### 5.2.3 系统调用机制的实现

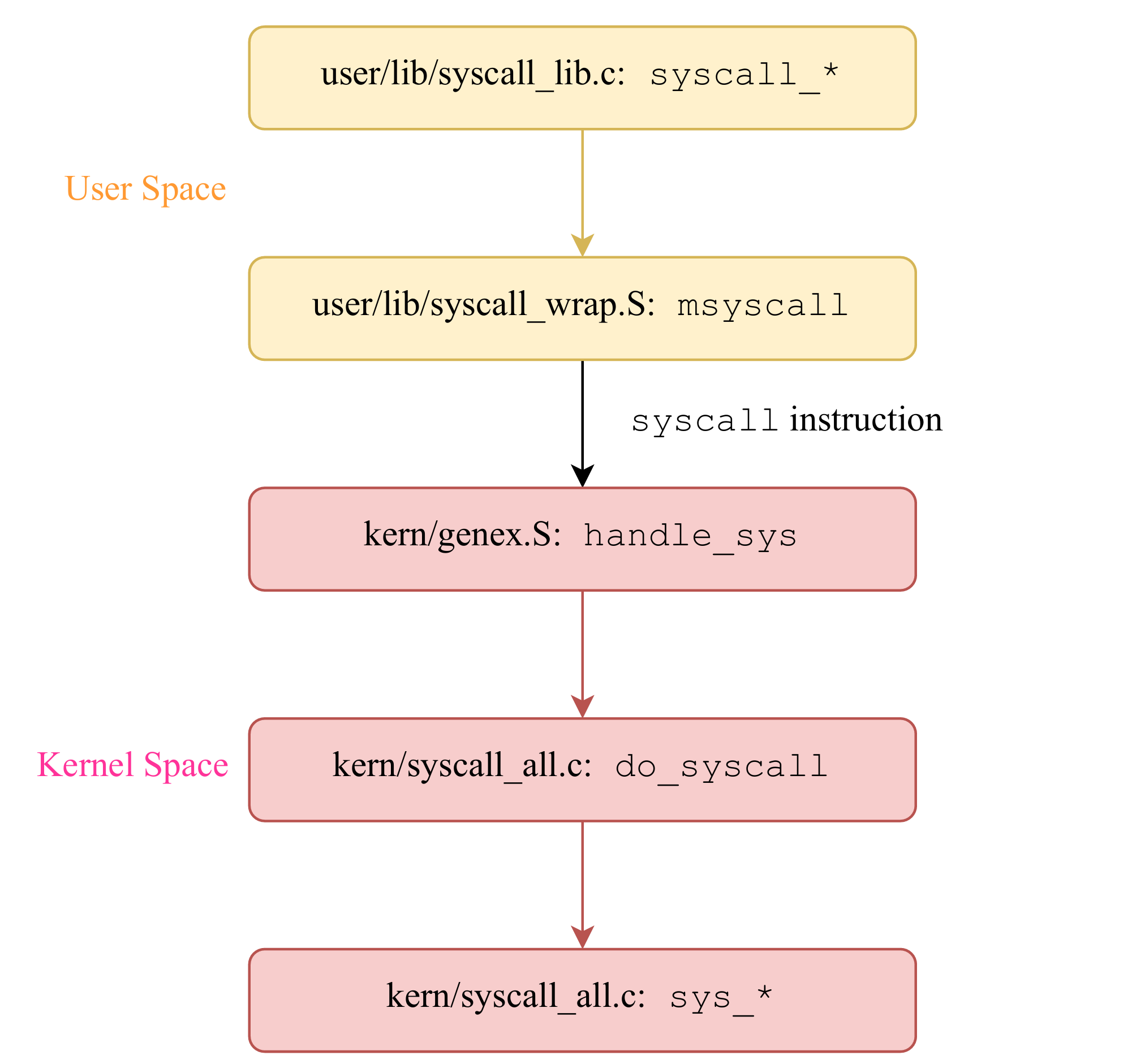
发现了系统调用的本质之后，我们就要着手在我们的 la64 操作系统中实现一套系统调用机制了。为了使得后面的实现思路更清晰，这里我们先回顾一下 Lab3 里面中断异常处理的行为：

1. 处理器跳转到异常分发代码处
2. 进入异常分发程序，根据 ESTAT 寄存器值判断异常类型并跳转到对应的处理程序
3. 处理异常，并返回

大家一定还记得异常分发向量组中的8号异常，它就是我们的操作系统处理系统调用时的异常。我们观察已有代码并跟随用户态 user/lib/debugf.c 中的 debugf 函数来学习其具体流程：

1. debugf 函数内部的逻辑可分为两部分，一部分负责将参数解析为字符串，一部分负责将字符串输出（debug\_output 函数）
2. debug\_output 函数调用了用户空间的 syscall\_\* 函数
3. syscall\_\* 函数调用了 msyscall 函数，系统由此陷入内核态
4. 内核态中将异常分发到 handle\_sys 函数，将系统调用所需要的信息 (在此处是需要输出的字符串 s) 传递入内核
5. 内核取得信息，执行对应的内核空间的系统调用函数（sys\_\*）
6. 系统调用完成，并返回用户态，同时将返回值“传递”回用户态
7. 从系统调用函数返回，回到用户程序 debugf 调用处

按照如上流程阅读代码的过程中，相信你已经发现了系统调用的流程，如下图所示：



在用户空间的程序中，我们定义了许多的函数，以 debugf 函数为例，这一函数实际上并不是最接近内核的函数，它最后会调用一个名为 syscall\_print\_cons 的函数，这个函数在 user/lib/syscall\_lib.c 中。下面我们来引入**系统调用号**。系统调用号的概念：

1. 在我们的 la64 操作系统实验中，这些 syscall\_\* 的函数与内核中的系统调用函数（ sys\_\* 的函数）是一一对应的。(a) syscall\_\* 的函数是我们在用户空间中最接近的内核的函数，在用户态调用。(b) sys\_\* 的函数是内核中系统调用的具体实现部分，在内核态执行。
2. 在所有 syscall\_\* 的函数的实现中，都调用了 msyscall 函数。
3. msyscall 函数的第一个参数是一个与调用名相似的宏。例：SYS\_print\_cons。
4. 这个参数就是系统调用号，定义于 include/syscall.h。

系统调用号的作用：类似于不同异常类型对应不同异常号，系统调用号是内核区分不同系统调用的唯一依据。msyscall 函数的参数：msyscall 一共有 6 个参数，除系统调用号之外 msyscall 函数还有 5 个参数，这些参数是系统调用时需要传递给内核的参数，因为最多参数的系统调用所需要的参数数量（syscall\_mem\_map 函数需要 5 个参数）。进一步的问题是，这些参数究竟是如何从用户态传递入内核态的呢？这里就需要用 loongarch 的调用规范来说明了。这里引入**栈帧**。栈帧的概念：

1. 在 loongarch 的调用规范中，进入函数体时会通过对栈指针做减法（压栈）的方式为该函数自身的局部变量、返回地址、调用函数的参数分配存储空间。
2. 而在函数调用结束之后会对栈指针做加法（弹栈）来释放这部分空间。
3. 这部分空间称为栈帧（stack frame）。

栈帧的使用方法：

1. 调用方：在自身栈帧的底部预留被调用函数的参数存储空间。
2. 被调用方：从调用方的栈帧中读取参数。

根据 loongarch 寄存器使用规范，让我们来学习一下栈帧的实现细节：

1. 寄存器 $a0-$a3 用于存放函数调用的前四个参数。
2. 其余的参数存放在栈中。(a) 但是前四个参数还需要在栈上预留空间，就像它们需要从栈上传递一样。

一个在 loongarch 中的例子：

1. msyscall 函数一共有 6 个参数。(a) 前 4 个参数会被 syscall\_\* 的函数分别存入 $a0-$a3 寄存器（寄存器传参的部分）同时栈帧底部保留 16 字节的空间（不要求存入参数的值） (b) 后 2 个参数只会被存入在预留空间之上的 8 字节空间内（没有寄存器传参）
2. 总共 24 字节的空间用于参数传递。
3. C 代码中的这些调用过程会由编译器自动编译为汇编代码。
4. 而我们在内核中则需要显式地从保存的用户上下文中获取函数的参数值。

当 user/lib/syscall\_lib.c 中定义的用户包装函数 syscall\_\* 调用 msyscall 时，根据以上的调用规范，需要传递给内核的系统调用号以及其他参数已经被合理安置。接下来我们只需要编写用户空间中的msyscall 函数：

1. 这个叶函数没有局部变量。
2. 也就是说这个函数不需要分配栈帧。
3. 只需执行自陷指令 syscall 来陷入内核态并在处理结束后正常返回即可。
4. 请注意，syscall 指令是不允许在延迟槽中使用的。

在通过 syscall 指令陷入内核态后，处理器将 PC 寄存器指向一个内核中固定的异常处理入口。在异常向量表中，系统调用这一异常类型的处理入口为 handle\_sys 函数，它是在 kern/genex.S 中定义的对 do\_syscall 函数的包装，我们需要首先在 kern/syscall\_all.c 中实现 do\_syscall 函数。关于陷入内核态，有一些点需要注意，具体如下：

1. 陷入内核态的操作并不是从一个函数跳转到了另一个函数，代码使用的栈指针 $sp 是内核空间中的栈指针。
2. 系统从用户态切换到内核态后，内核首先需要将原用户进程的运行现场保存到内核空间。(a) 在 kern/entry.S 中通过 SAVE\_ALL 宏完成。
3. 随后的栈指针则指向保存的 Trapframe，因此我们正是借助这个保存的结构体来获取用户态中传递过来的值。(a) 例如：用户态下 $a0 寄存器的值保存在内核栈的 TF\_REG4(sp) 处。
4. 当内核在 handle\_ 开头的包装函数中调用实际进行异常处理的 C 函数时，这个栈指针将作为参数传递给这个 C 函数。
5. 因此我们可以在 C 语言中通过这个 structTrapframe \* 来获取用户态现场中的参数。

建议大家去阅读 kern/syscall\_all.c 中的 do\_syscall 函数，详细理解内核部分的系统调用机制是如何工作的。做完这一步，整个系统调用的机制已经可以正常工作，接下来我们要来实现几个具体的系统调用。

### 5.2.4 基础系统调用函数

在了解系统调用机制后，接下来我们实现几个系统调用。kern/syscall\_all.c 中定义了一系列系统调用，它们就是 loongarch 系统的基础系统调用，后续的 IPC 与 fork 机制都以这些系统调用作为支撑。sys\_mem\_alloc 函数：

1. 这个函数的主要功能是分配内存。(a) 通过这个系统调用，用户程序可以给该程序所允许的虚拟内存空间显式地分配实际的物理内存。(b) 对于我们程序员的视角而言，是我们编写的程序在内存中申请了一片空间。(c) 对于操作系统内核来说，是一个进程请求将其运行空间中的某段地址与实际物理内存进行映射。i. 此处请同学们回顾进程虚拟页面映射机制、物理内存申请机制。ii. 我们将通过 page\_alloc，page\_insert 来实现内存分配和映射页面。iii. 完成时请注意检查虚拟地址的合法性。iv. 向系统调用传入的虚拟地址应当是有效的用户地址，否则不进行内存分配操作，返回错误代码 -E\_INVAL。(d) 从而可以通过该虚拟页面来对物理内存进行存取访问。
2. 接受一个进程的标识符（envid）作为参数，来确定发出请求的进程。
3. 通过调用 envid2env 函数获得对应的进程控制块。

sys\_mem\_map 函数：

1. 这个函数的作用是将源进程地址空间中的相应内存映射到目标进程的相应地址空间的相应虚拟内存中去。(a) 此时两者共享一页物理内存。
2. 这个函数有如下操作逻辑。(a) 找到需要操作的两个进程。(b) 获取源进程的虚拟页面对应的实际物理页面。(c) 将该物理页面与目标进程的相应地址完成映射。(d) 完成时请注意检查虚拟地址的合法性。i. 传入的虚拟地址应当是有效的用户地址。ii. 否则不进行页面映射操作，返回错误代码 -E\_INVAL，表示传入非法（Invaild）用户地址。

关于内存还有一个函数：sys\_mem\_unmap，这个系统调用的功能是解除某个进程地址空间虚拟内存和物理内存之间的映射关系。用到的函数：page\_remove。除了与内存相关的函数外，另外一个常用的系统调用函数是 sys\_yield。这个函数的功能是实现用户进程对 CPU 的放弃，从而调度其他的进程。可以利用我们之前已经编写好的函数 schedule。至此，我们能够进一步理解进程与内核间的关系并非对立：在内核处理进程发起的系统调用时，我们并没有切换地址空间（页目录地址），也不需要将进程上下文（Trapframe）保存到进程控制块中，只是切换到内核态下，执行了一些内核代码。可以说，处理系统调用时的内核仍然是代表当前进程的，这也是系统调用、TLB缺失等同步异常与时钟中断等异步异常的本质区别，实现系统调用后，我们已经可以编写并运行用户程序，利用系统调用让用户程序在控制台上输出文本了。

## 5.3 进程间通信机制(IPC)

进程间通信机制 (IPC) 是微内核最重要的机制之一。IPC 机制的实现远远没有我们想象得那样神秘，特别是在我们这个被简化了的 la64 操作系统中。IPC 机制的实现使得我们系统中的进程之间拥有了相互传递消息的能力，为后续实现 fork、文件系统服务、管道和 shell 均有着极大的帮助。根据之前的讨论，我们能够确定的是:

* IPC 的目的是使两个进程之间可以通信
* IPC 需要通过系统调用来实现
* IPC 还与进程的数据、页面等信息有关

所谓通信，最直观的一种理解就是交换数据。假如我们能够将让一个进程有能力将数据传递给另一个进程，那么进程之间自然具有了相互通信的能力。由于这些进程的地址空间之间是相互独立的，要想传递数据，我们就需要想办法把一个地址空间中的东西传给另一个地址空间。我们知道，所有的进程都共享同一个内核空间（主要为kseg0）。因此，想要在不同空间之间交换数据，我们就可以借助于内核空间来实现。发送方进程可以将数据以系统调用的形式存放在进程控制块中，接收方进程同样以系统调用的方式在进程控制块中找到对应的数据，读取并返回.

struct Env {  
 u\_long env\_ipc\_value; // the value sent to us  
 u\_long env\_ipc\_from; // envid of the sender  
 u\_long env\_ipc\_recving; // whether this env **is** blocked receiving  
 u\_long env\_ipc\_dstva; // va at which the received page should be mapped  
 u\_long env\_ipc\_perm; // perm **in** which the received page should be mapped  
};

在进程控制块中我们看到了我们想要的内容：

|  |  |
| --- | --- |
| env\_ipc\_value | 进程传递的具体数值 |
| env\_ipc\_from | 发送方的进程ID |
| env\_ipc\_recving | 1：等待接受数据中； 0：不可接受数据 |
| env\_ipc\_dstva | 接收到的页面需要与自身的哪个虚拟页面完成映射 |
| env\_ipc\_perm | 传递的页面的权限位设置 |

知道了这些，我们就不难实现 IPC 机制了。请结合下方讲解完成学习：**sys\_ipc\_recv(u\_int dstva)** 函数用于接受消息。在该函数中：

1. 首先要将自身的 env\_ipc\_recving 设置为 1，表明该进程准备接受发送方的消息
2. 之后给 env\_ipc\_dstva 赋值，表明自己要将接受到的页面与 dstva 完成映射
3. 阻塞当前进程，即把当前进程的状态置为不可运行（ENV\_NOT\_RUNNABLE）
4. 最后放弃 CPU（调用相关函数重新进行调度），等待发送方将数据发送过来

**sys\_ipc\_try\_send(u\_int envid, u\_int value, u\_int srcva, u\_int perm)** 函数用于发送消息：

1. 根据 envid 找到相应进程，如果指定进程为可接收状态(考虑 env\_ipc\_recving)，则发送成功
2. 否则，函数返回 -E\_IPC\_NOT\_RECV，表示目标进程未处于接受状态
3. 清除接收进程的接收状态，将相应数据填入进程控制块，传递物理页面的映射关系
4. 修改进程控制块中的进程状态，使接受数据的进程可继续运行(ENV\_RUNNABLE)

建议大家仔细阅读 IPC 相关函数，把执行过程中的内核态用户态状态跳转，以及数据传输等等地方搞清楚，加深对系统调用的理解。值得一提的是，由于在我们的用户程序中，会大量使用 srcva 为 0 的调用来表示只传value 值，而不需要传递物理页面，换句话说，当 srcva 不为 0 时，我们才建立两个进程的页面映射关系。因此在学习相关函数时也需要注意此种情况。

## 5.4 Fork

在 Lab3 我们曾提到过，内核通过 env\_create 函数创建一个进程。但如果要让一个进程创建一个进程，我们就需要基于系统调用，引入新的 fork 机制了。

### 5.4.1 初窥 fork

fork 函数简介：

1. fork 这个函数名有如下内涵。(a) 直接的翻译是叉子的意思，而在操作系统中是表示分叉的意思。(b) 就好像一条河流动着，遇到一个分叉口，分成两条河一样。(c) fork 就是那个分叉口。
2. 一个进程在调用 fork() 函数后，将从此分叉成为两个进程运行。(a) 其中新产生的进程称为原进程的子进程。i. 子进程开始运行时的大部分上下文状态与原进程相同,包括程序和 fork 运行时的现场（包括通用寄存器和程序计数器PC等）ii. 在子进程中，fork() 调用的返回值为0。(b) 其中旧的进程称为子进程的父进程。i. 在父进程中，fork() 调用的返回值为子进程的 env\_id。+ env\_id 一定大于0。+ fork 失败的情况下，子进程不会被创建，且父进程将得到小于0的返回值。

fork 在父子进程中产生不同返回值这一特性，让我们能够在代码中调用 fork 后判断当前在父进程还是子进程中，以执行不同的后续逻辑，也使父进程能够与子进程进行通信。你可能会想，fork 执行完为什么不直接生成一个空白的进程块，生成一个几乎和父进程一模一样的子进程有什么用呢？事实上，fork 是 Linux 操作系统中创建新进程最主要的方式。相比独立开始运行的两个进程，父子进程间的通信要方便的多。因为子进程中仍能读取原属于父进程的部分数据，父进程也可以根据 fork 返回的子进程 id，通过调用其他系统接口控制其行为。在 Linux 中，与 fork 经常一起使用的，是名为 exec 的一系列系统调用。它会使进程抛弃现有的程序和运行现场，执行一个新的程序。若在进程中调用 exec，进程的地址空间（以及在内存中持有的所有数据）都将被重置，新程序的二进制镜像将被加载到其代码段，从而让一个从头运行的全新进程取而代之。fork 的一种常见应用就被称作 fork-exec，指在 fork 出的子进程中调用 exec，从而在创建出的新进程中运行另一个程序。为了让读者对 fork 的认识不只是停留在理论层面，我们下面来做一个小实验，复制到您的 Linux 环境下运行一下吧。

#include *<stdio.h>*  
#include *<unistd.h>*  
  
int main() {  
 int var= 1;  
 long pid;  
 printf("Beforefork,var=%d.**\n**", var);  
 pid = fork();  
 printf("Afterfork, var=%d.**\n**",var);  
 **if**(pid == 0) {  
 var = 2;  
 sleep(3);  
 printf("child got %ld, var = %d", pid, var);  
 } **else** {  
 sleep(2);  
 printf("parent got %ld, var = %d", pid, var);  
 }  
 printf(", pid: %ld**\n**", (long) getpid());  
 **return** 0;  
}

使用 gcc fork\_test.c && ./a.out 运行一下，你得到的输出应该如下所示（pid 可能不同）

Before fork, var = 1.  
After fork, var = 1.  
After fork, var = 1.  
parent got 16903, var = 1, pid: 16902  
child got 0, var = 2, pid: 16903

我们从这段简短的代码里可以获取到很多的信息，比如以下几点：

* fork 之前只有父进程存在。
* fork 之后，父子进程同时开始执行 fork 之后的代码段。
* fork 在不同的进程中返回值不一样，在子进程中返回值为 0，在父进程中返回值不为 0，而为子进程的 pid（Linux 中进程专属的 id，类似于 la64 中的 envid）。
* 父进程和子进程虽然很多信息相同，但他们的进程控制块是不同的。从上面的小实验我们也能看出来，子进程实际上就是按父进程的代码段等内存数据，以及进程上下文等状态作为模板而雕琢出来的。但即使如此，父子进程也还是有很多不同的地方。我们简要概括一下整个 fork 实现过程中需要仔细阅读的文件，包括：
* kern/syscall\_all.c：sys\_exofork 函数，sys\_set\_env\_status 函数，sys\_set\_tlb\_mod\_entry 函数是我们这次需要完成的函数。
* kern/tlbex.c：do\_tlb\_mod 函数负责完成写时复制处理前的相关设置，也是我们这次需要完成的函数。
* user/lib/fork.c：fork 函数是我们这次实验的重点函数，我们将分多个步骤来学习这个函数。
* user/lib/fork.c：cow\_entry 函数是写时复制处理的函数，也是内核会从 do\_tlb\_mod 返回到的函数，负责对带有 PTE\_COW 标志的页面进行处理，是我们这次需要完成的主要函数之一。
* user/lib/fork.c：duppage 函数是父进程对子进程页面空间进行映射以及相关标志设置的函数，是我们这次需要完成的主要函数之一。
* user/lib/entry.S：用户进程的入口，是我们这次需要了解的函数之一。
* user/lib/libos.c：用户进程入口的 C 语言部分，负责完成执行用户程序 main 前后的准备和清理工作，是我们这次需要了解的函数之一。
* kern/genex.S：该文件实现了 la64 的异常处理流程，虽然不是我们需要实现的重点，但是建议读者认真阅读，理解中断处理的流程。

### 5.4.2 写时复制机制

前文中，我们知道了在调用 fork 后，子进程会继承父进程地址空间中的代码段和数据段等内容。由于在 fork 后，父子进程将成为相互独立的两个进程，因此两个进程对于其内存的修改应该是互不影响的。如果我们在 fork 时将父进程地址空间中的内容全部复制到新的物理页，将会消耗大量的物理内存。而这些物理内存中，如代码段部分，父子进程通常不会对其进行写入。对于这样的页面，我们希望能够避免对它们进行复制，从而可以节省物理内存。为了父子进程能够共用尽可能多的物理内存，我们希望引入一种写时复制（Copy-on-write，COW）机制：

1. 在 fork 时，只需将地址空间中的所有可写页标记为写时复制页面。
2. 根据标记，在父进程或子进程对写时复制页面进行写入时，能够产生一种异常。
3. 操作系统处理异常如下。(a) 为当前进程试图写入的虚拟地址分配新的物理页面。(b) 新的页面复制原页面的内容。(c) 返回用户程序。
4. 处理完成后即可对新分配的物理页面进行写入。
5. 这种机制使得我们可以最大限度的节省物理内存，减少了不必要的复制。

为了实现写时复制机制，我们需要借助硬件异常来实现。在 loongarch 中，当程序尝试写入的虚拟页对应的 TLB 项没有 PTE\_D 标志时，会触发 TLBMod 异常，使系统陷入到内核中。由于对应的异常处理函数是由内核设置的，因此我们可以使用它来实现写时复制机制。我们可以将写时复制界面的 PTE\_D 标志置为 0。当进程读这个页面时，不会出现问题。但当进程尝试写这个页面时，由于 PTE\_D 为 0，所以会触发 TLB Mod 异常。此时，我们就可以在异常处理函数中，将虚拟页映射到一个新的物理页，然后将旧的物理页的内容复制到新的物理页中。这样，两个进程的虚拟页就会各自映射到不同的物理页了。之后进程再对这个虚拟页进行修改，就不会有任何问题了。同时，为了区分真正的“只读”页面和“写时复制”页面，我们需要利用 TLB 项中的之前没有使用过的标志位，引入新的标志位 PTE\_COW，在 loongarch 中，也就是 PTE\_W， 写位。在 TLBMod 的异常处理函数中，如果触发该异常的页面的 PTE\_COW 为 1，我们就需要进行上述的写时复制处理，即分配一页新的物理页，将写时复制页面的内容拷贝到这一物理页，然后映射给尝试写入该页面的进程。在我们的 la64 操作系统实验中，进程调用 fork 时，需要对其所有的可写入的内存页面，设置页表项标志位 PTE\_COW 并取消可写位 PTE\_D，以实现写时复制保护。在这样的保护下，用户程序可以在逻辑上认为 fork 时父进程中内存的状态被完整复制到了子进程中，此后父子进程可以独立操作各自的内存。

### 5.4.3 fork 的返回值

在我们的 la64 操作系统实验中，需要强调的一点是我们实现的 fork 是一个用户态函数，fork 函数中需要若干个原子的系统调用来完成所期望的功能。其中最核心的一个系统调用就是新进程的创建 syscall\_exofork。在 fork 的实现中，我们是通过判断 syscall\_exofork 的返回值来决定 fork 的返回值以及后续动作，所以会有类似这样结构的代码：

envid = syscall\_exofork();   
**if** (envid == 0) {  
 // 子进程  
 ...  
} **else** {  
 // 父进程  
 ...  
}

既然 fork 的目的是使得父子进程处于几乎相同的运行状态，我们可以认为在返回用户态时，父子进程应该经历了同样的恢复运行现场的过程，只不过对于父进程是从系统调用中返回时恢复现场，而对于子进程则是在进程被调度时恢复现场。在现场恢复后，父子进程都会从内核返回到 msyscall 函数中，而它们的现场中存储的返回值（即 $v0 寄存器的值）是不同的。这一返回值随后再被返回到 syscall\_exofork 和 fork 函数，使 fork 函数也能区分两者。为了实现这一特性，你可能需要先学习 sys\_exofork 的几个任务，在它分配一个新的进程控制块后，还需要用一些当前进程的信息作为模版来填充这个控制块：**运行现场** 要复制一份当前进程的运行现场（进程上下文） Trapframe 到子进程的进程控制块中。**返回值有关** 这个系统调用本身是需要一个返回值的，我们希望系统调用在内核态返回的 envid 只传递给父进程，对于子进程，则需要将其现场中的 v0 寄存器修改为 0。**进程状态** 我们当然不能让子进程在父进程的 syscall\_exofork 返回后就直接被调度，因为这时候它还没有做好充分的准备，所以我们需要将它的状态设为 ENV\_NOT\_RUNNABLE 且避免它被加入调度队列。在解决完返回值的问题之后，父与子就能够分别各自从 syscall\_exofork 中返回。

### 5.4.4 地址空间的准备

la64 允许进程访问自身的进程控制块，而在 user/lib/libos.c 的实现中，用户程序在运行时入口会将一个用户空间中的指针变量 struct Env \*env 指向当前进程的控制块。对于 fork 后的子进程，它具有了一个与父亲不同的进程控制块，因此在子进程第一次被调度的时候（当然这时还是在 fork 函数中）需要对 env 指针进行更新，使其仍指向当前进程的控制块。这一更新过程与运行时入口对 env 指针的初始化过程相同，具体步骤如下：

1. 通过一个系统调用来取得自己的 envid，因为对于子进程而言 syscall\_exofork 返回的是一个 0 值。
2. 根据获得的 envid，计算对应的进程控制块的下标，将对应的进程控制块的指针赋给 env。

此外，父进程还需要将地址空间中需要与子进程共享的页面映射给子进程，这需要我们遍历父进程的大部分用户空间页，并使用将要实现的 duppage 函数来完成这一过程。duppage 时，对于可以写入的页面的页表项，在父进程和子进程都需要加上 PTE\_COW 标志位，同时取消 PTE\_D 标志位，以实现写时复制保护。在 duppage 函数中，唯一需要强调的一点是，要对具有不同权限位的页使用不同的方式进行处理。学习时需要注意这几种情况：**只读页面** 对于不具有 PTE\_D 权限位的页面，按照相同权限（只读）映射给子进程即可。**写时复制页面** 即具有 PTE\_COW 权限位的页面。这类页面是之前的 fork 时 duppage 的结果，且在本次 fork 前必然未被写入过。**共享页面** 即具有 PTE\_LIBRARY 权限位的页面。这类页面需要保持共享可写的状态，即在父子进程中映射到相同的物理页，使对其进行修改的结果相互可见。在文件系统部分的实验中，我们会使用到这样的页面。 **可写页面** 即具有 PTE\_D 权限位，且不符合以上特殊情况的页面。这类页面需要在父进程和子进程的页表项中都使用 PTE\_COW 权限位进行保护。在完成写时复制的保护机制后，还不能让子进程处于能被调度的状态，因为作为父亲它还有其他的责任——为写时复制特性的页写入异常处理做好准备。

### 5.4.5 页写入异常

内核在捕获到一个常规的缺页中断（TLB 缺失异常）时会陷入异常，跳转到异常处理函数 handle\_tlb 中，这一汇编函数的实现在 kern/genex.S 中，通过调用 do\_tlb\_refill 函数，在页表中进行查找，将物理地址填入 TLB 并返回到用户程序中的异常地址，再次执行访存指令。前文中我们提到了写时复制（COW）特性，这种特性也是依赖于异常处理的。当用户程序写入一个在 TLB 中被标记为不可写入（无 PTE\_D）的页面时，loongarch 会陷入页写入异常（TLBMod），我们在异常向量组中为其注册了一个处理函数 handle\_mod，这一函数会跳转到 kern/tlbex.c 中的 do\_tlb\_mod 函数中，这个函数正是处理页写入异常的内核函数。对于需要写时复制（COW）的页面，我们只需取消其 PTE\_D 标记，即可在它们被写入时触发 do\_tlb\_mod 中的处理逻辑。你可能会发现，do\_tlb\_mod 函数似乎并没有进行页面复制等 COW 的处理操作。事实上，我们的 la64 操作系统按照微内核的设计理念，尽可能地将功能实现在用户空间中，其中也包括了页写入异常的处理，因此主要的处理过程是在用户态下完成的。如果需要在用户态下完成页写入异常的处理，是不能直接使用正常情况下的用户栈的（因为发生页写入异常的也可能是正常栈的页面），所以用户进程就需要一个单独的栈来执行处理程序，我们把这个栈称作异常处理栈，它的栈顶对应的是内存布局中的 UXSTACKTOP。此外，内核还需要知晓进程自身的处理函数所在地址，它的地址存在于进程控制块的 env\_user\_tlb\_mod\_entry 域中，这个地址也需要事先由父进程通过系统调用设置。因此，概括一下上述内容，在我们的 la64 操作系统中，处理页写入异常的大致流程可以概括为：

1. 用户进程触发页写入异常，陷入到内核中的 handle\_mod，再跳转到 do\_tlb\_mod 函数。
2. do\_tlb\_mod 函数负责将当前现场保存在异常处理栈中，并设置 a0 和 ERA 寄存器的值，使得从异常恢复后能够以异常处理栈中保存的现场（Trapframe）为参数，跳转到 env\_user\_tlb\_mod\_entry 域存储的用户异常处理函数的地址。
3. 从异常恢复到用户态，跳转到用户异常处理函数中，由用户程序完成写时复制等自定义处理。

处理 COW 时，我们需要注册的页写入异常用户处理函数是 fork.c 中定义的 cow\_entry 函数。这个函数进行写时复制处理之后，使用系统调用 syscall\_set\_trapframe 恢复事先保存好的现场，其中也包括 sp 和 PC 寄存器的值，使得用户程序恢复执行。让我们回到 fork 函数，在调用 syscall\_exofork 之前，我们需要使用 syscall\_set\_tlb\_mod\_entry 函数来注册自身的页写入异常处理函数，也就是我们上文提到的 env\_user\_tlb\_mod\_entry 域指向的用户处理函数。这里需要通过系统调用告知内核自身的处理程序是 cow\_entry。你还需要完成内核中的系统调用处理函数 sys\_set\_tlb\_mod\_entry，将进程控制块的 env\_user\_tlb\_mod\_entry 域设为传入的参数。我们现在知道了页写入异常处理时会返回到用户空间的 cow\_entry 函数，我们再来看这个函数会做些什么。

static void \_\_attribute\_\_((noreturn)) cow\_entry(struct Trapframe \*tf) {  
 u\_int va = tf->cp0\_badvaddr;  
 // ...  
 int r = syscall\_set\_trapframe(0, tf);  
 user\_panic("syscall\_set\_trapframe returned *%d*", r);  
}

从内核返回后，此时的第一个参数是由内核设置的，处于异常处理栈中，且指向一个由内核复制好的 Trapframe 结构体的底部。该函数从 Trapframe 中读取了 TF\_BADV 字段的值，这个值也正是 CPU 设置的发生页写入异常的地址 va，我们可以根据这个地址进行写时复制处理。在函数的最后，使用系统调用 syscall\_set\_trapframe 恢复了保存的现场。说到这里，我们就要来学习 cow\_entry 中真正进行处理的部分了。你需要学习以下这些任务相关的内容：

1. 根据 vpt 中 va 所在页的页表项，判断其标志位是否包含 PTE\_COW，是则进行下一步，否则调用 user\_panic() 报错。
2. 分配一个新的临时物理页到临时地址 UCOW，使用 memcpy 将 va 页的数据拷贝到刚刚分配的页中。
3. 将发生页写入异常的地址 va 映射到临时页面上，注意设定好对应的页面标志位（即去除 PTE\_COW 并恢复 PTE\_D），然后解除临时地址 UCOW 的内存映射。

父进程还需要使用 syscall\_set\_tlb\_mod\_entry，设置子进程的页写入异常处理函数为 cow\_entry。最后，父进程通过系统调用 syscall\_set\_env\_status 设置子进程为可以运行的状态。在内核中实现sys\_set\_env\_status 函数时，不仅需要设置进程控制块的 env\_status 域，还需要在 env\_status 从 ENV\_NOT\_RUNNABLE 转换为 ENV\_RUNNABLE 时将控制块加入到可调度进程的链表中，反之则从链表中移除。说到这里我们需要整理一下思路，fork 中父进程在 syscall\_exofork 后还需要做的事情有：

1. 遍历父进程地址空间，进行 duppage。
2. 设置子进程的异常处理函数，确保页写入异常可以被正常处理。
3. 设置子进程的 env\_status，允许其被调度。

最后再将子进程的 envid 返回，fork 函数就完成了。

## 5.5 实验正确结果

本次实验下有多个测试程序，最简单的单元测试是 envid2env\_check()，在完成 envid2env 后，将 envid2env\_check() 加入 init/init.c 即可测试，也可以使用 make test lab=4\_1直接构建测试。其参考输出如下：

envid2env()workwell!

用户态的第一个测试程序是 user/tltest.c，在系统调用部分完成后，将 ENV\_CREATE(user\_tltest) 加入 init/init.c，使用 make && make run即可测试，参考输出如下：

Smashing some kernel codes...   
If your implementation **is** correct, you may see unknown exception here:  
...  
panic at traps.c:24 (do\_reserved): Unknown ExcCode 5

完成fork后，单独测试fork的程序是user/fktest.c，使用方法同上。其参考输出如下：

this **is** father: a:1  
this **is** father: a:1  
this **is** father: a:1  
...  
 this **is** child :a:2  
 this **is** child :a:2  
 this **is** child :a:2  
...  
 this **is** child2 :a:3  
 this **is** child2 :a:3  
 this **is** child2 :a:3

其中三种文本段应当交替出现且永不停止。另一个测试程序 user/pingpong.c 主要测试 fork 和进程间通信，参考输出如下：

@@@@**@send** 0 **from** 800 to 1001  
1001 am waiting.....  
800 am waiting.....  
1001 got 0 **from** 800  
  
@@@@**@send** 0 **from** 1001 to 800  
1001 am waiting.....  
800 got 1 **from** 1001

# 6 实验五：文件系统

## 6.1 实验目的

1. 了解文件系统的基本概念和作用。
2. 了解普通磁盘的基本结构和读写方式。
3. 了解实现设备驱动的方法。
4. 掌握并实现文件系统服务的基本操作。
5. 了解微内核的基本设计思想和结构。

在之前的实验中，我们把所有的程序和数据都存放在内存中。然而内存空间的大小是有限的，而且内存中的数据存在易失性问题。因此有些数据必须保存在磁盘、光盘等外部存储设备上, 这些外部存储设备能够长期保存大量的数据，且便于将数据装载到不同进程的内存空间进行共享。为了便于管理和访问存放在外部存储设备上的数据，在操作系统中引入了文件系统。在文件系统中，文件是数据存储和访问的基本单位。对于用户而言，文件系统可以屏蔽访问外存数据的复杂性。

## 6.2 文件系统概述

计算机文件系统是一种存储和组织数据的方法，便于访问和查找数据。文件系统使用文件和树型目录的逻辑抽象屏蔽了底层硬盘和光盘等物理设备基于数据块进行存储和访问的复杂性。用户不必关心数据实际保存在硬盘（或者光盘）的哪个数据块上，只需要记住这个文件的所属目录和文件名。在写入新数据之前，用户不必关心硬盘上的哪个块是空闲的，硬盘上的存储空间管理（分配和释放）由文件系统自动完成，用户只需要记住数据被写入到了哪个文件中即可。常见的文件系统通常基于硬盘和光盘等块存储设备，并维护文件在设备中的物理位置；而此外的某些文件系统仅仅是一种访问数据的接口，实际的数据在内存中或者通过网络协议（如NFS、SMB、FTP 等）提供。广义上，一切带标识的、在逻辑上有完整意义的字节序列都可以称为“文件”。文件系统将外部设备中的资源抽象为文件，从而可以统一管理外部设备，实现对数据的存储、组织、访问和修改等操作。在本实验中，我们拟实现一个精简的文件系统，其中需要对三种设备进行统一管理，即文件设备（file，即狭义的“文件”）、控制台（console）和管道（pipe）。其中，后两者将在下一个实验“管道与 Shell”中进行使用。

### 6.2.1 文件系统的设计与实现

在本次实验中，我们将要实现一个简单但结构完整的文件系统。整个文件系统包括以下几个部分：

1. 外部存储设备驱动：通常，我们需要按一定顺序读写设备寄存器，来实现对外部设备的操作。为了将这种操作转化为具有通用、明确语义的接口，必须实现相应的驱动程序。在本部分，我们将实现IDE 磁盘的用户态驱动程序，该驱动程序将通过系统调用的方式陷入内核，对磁盘镜像进行读写操作。
2. 文件系统结构：在本部分，我们会实现模拟磁盘的驱动程序以及磁盘上和操作系统中的文件系统结构，并实现文件系统操作的相关函数。在我们的操作系统中，fs目录下存放的是文件系统服务程序的代码；而 user/lib 目录中的 file.c、fd.c、fsipc.c 等文件则存放了文件系统的用户库，这部分代码会被一同链接到用户程序中，允许用户程序调用其中的函数来操作文件系统。文件系统服务进程和其他用户进程之间使用 Lab4 中实现的 IPC 机制进行通信。
3. 文件系统的用户接口：在本部分，要求为用户提供接口和机制使得用户程序能够使用文件系统，这主要通过一个用户态的文件系统服务来实现。同时，我们将引入文件描述符等结构来抽象地表示一个进程所打开的文件，而不必关心文件实际的物理表示。

整个文件系统所涉及的代码文件比较多, 此处我们概要介绍一些核心代码文件的主要功能，希望有助于理解文件系统实现的总体框架。这些代码实现在三个目录中：

* tools：目录中存放的是构建时辅助工具的代码。在之前的 Lab 中，我们实现了解析 ELF 文件的 readelf 工具; 本 Lab 中，我们将在其中实现 fsformat 工具，并借助它来创建磁盘镜像。请注意，tools 目录下的代码仅用于 la64 的构建，在宿主 Linux 环境（而非 loongarch 模拟器）中运行，也不会被编译进 la64 的内核、用户库或用户程序中。
* fs：目录中存放的是文件系统服务进程的代码。我们在 fs.c 中实现文件系统的基本功能函数，在 ide.c 中通过系统调用与磁盘镜像进行交互。该进程的主干函数在 serv.c 中，通过 IPC 通信与用户进程 user/lib/fsipc.c 内的通信函数进行交互。
* user/lib：目录下存放了用户程序的库函数。在本 Lab 中，该目录下的 fsipc.c 实现了与文件系统服务进程的交互，file.c 实现了文件系统的用户接口；fd.c 中实现了文件描述符，允许用户程序使用统一的接口，抽象地操作磁盘文件系统中的文件，以及控制台和管道等虚拟的文件。

整个文件系统体现了 la64 的微内核设计，包括下面三个部分：

1. 将传统操作系统的文件系统移出内核，使用用户态的文件系统服务程序以及一系列用户库来实现。即使它们崩溃，也不会影响到整个内核的稳定。其他用户进程通过进程间通信 (IPC) 来请求文件系统的相关服务。因此，在微内核中进程间通信 (IPC) 是一个十分重要的机制。
2. 操作系统将一些内核数据暴露到用户空间，使得进程不需要切换到内核态就能访问。la64 将进程页表映射到用户空间，此处文件系统服务进程访问自身进程页表即可判断磁盘缓存中是否存在对应块。
3. 将传统操作系统的设备驱动移出内核，作为用户程序来实现。微内核体现在，内核在此过程中仅提供读写设备物理地址的系统调用。

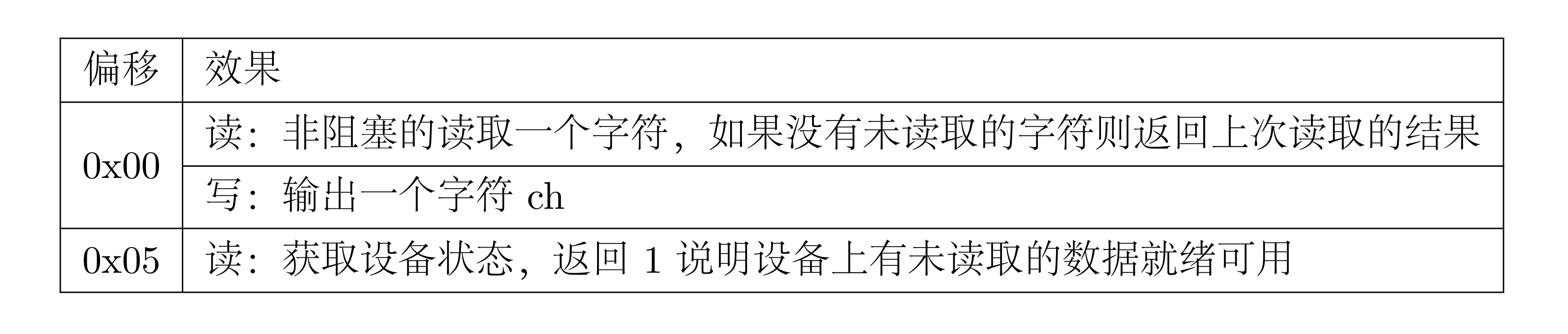
接下来我们一一详细解读这些部分的实现。

## 6.3 IDE 磁盘驱动

为了在磁盘等外部设备上实现文件系统，我们必须为这些外部设备编写驱动程序。实际上，la64 已经在 kern/machine.c 中实现了一个简单的控制台驱动程序，其中在内核态下使用了内存映射 I/O（MMIO）技术，通过控制台实现了字符的输入输出。本次要实现的硬盘驱动程序与已经实现的串口驱动，都采用 MMIO 技术编写驱动，不同之处在于，我们需要驱动的物理设备— IDE 磁盘功能更加复杂，并且本次要编写的驱动程序完全运行在用户空间。下面我们首先介绍内存映射 I/O，之后再了解 IDE 磁盘的结构和操作，最后介绍磁盘驱动程序的编写。

### 6.3.1 内存映射 I/O (MMIO)

在“内存管理”实验中，我们已经了解了 loongarch 存储器地址映射的基本内容。几乎每一种外设都是通过读写设备上的寄存器来进行数据通信，外设寄存器也称为 I/O 端口，主要用来访问 I/O 设备。外设寄存器通常包括控制寄存器、状态寄存器和数据寄存器，这些寄存器被映射到指定的物理地址空间。例如，在 MALTA 中，console 设备被映射到 0x180003F8，IDE 控制器被映射到0x180001F0，等等。实验中使用的 loongarch 体系结构并没有复杂的 I/O 端口的概念，而是统一使用内存映射 I/O 的模型。在 loongarch 的内核地址空间中（kseg1段）实现了硬件级别的物理地址和内核虚拟地址的转换机制，其中，对kseg1 段地址的读写不经过MMU映射，且不使用高速缓存，这正是外部设备驱动所需要的。由于我们是在模拟器上运行操作系统，I/O 设备的物理地址是完全固定的，因此我们可以通过简单地读写某些固定的内核虚拟地址来实现驱动程序的功能。在之前的实验中，我们曾经使用 KADDR 宏把一个物理地址转换为 kseg1 段的内核虚拟地址，实际上是给物理地址加上 kseg1 的偏移值（即 0x9000000000000000）。在编写设备驱动的时候，我们需要将物理地址转换为kseg1 段的内核虚拟地址，也就是同样给物理地址加上 kseg1 的偏移值(0x9000000000000000)。以我们编写完成的串口设备驱动为例，MALTA 提供的 console 设备是一个典型的 NS16550 设备，其基地址为 0x180003F8，设备寄存器映射如图所示。



现在，我们通过往内存的(0x180003F8 + 0x9000000000000000) 地址写入字符，就能在 shell 中看到对应的输出。kern/machine.c 中的 printcharc 函数的实现如下所示：

void printcharc(char ch) {  
 \*((**volatile** char \*)(KSEG1 + MALTA\_SERIAL\_DATA)) = ch;  
}

这个函数首先通过 KSEG1 + MALTA\_SERIAL\_DATA 定位到了字符需要被写入到的地址，为了对这个地址的数据进行操作需要将其强制转化为类型 char 的指针。注意，此处的类型 char 要使用 volatile 限定。因为对于 I/O 设备的操作，我们的预期是每次操作都直接与设备进行交互。如果不使用 volatile，编译器可能会对该地址的数据的访问做优化，导致对于 I/O 设备的地址的访问不直接与设备交互。例如，对于这个地址进行多次连续读取，编译器可能会优化为只读取一次，并复用这个结果，进而引发错误。转化为使用 volatile 修饰的指针之后，通过解引用，可以对这个地址的数据进行修改。而在本次实验中，我们需要编写的 IDE 磁盘驱动程序位于用户空间，用户态进程若是直接读写内核虚拟地址将会由处理器引发一个地址错误（ADEL/S）。所以对于设备的读写必须通过系统调用来实现。这里我们引入了 sys\_write\_dev 和 sys\_read\_dev 两个系统调用来实现设备的读写操作。这两个系统调用以用户虚拟地址、设备的物理地址和读写的长度（按字节计数），作为参数，在内核空间中完成 I/O 操作。

### 6.3.2 IDE 磁盘

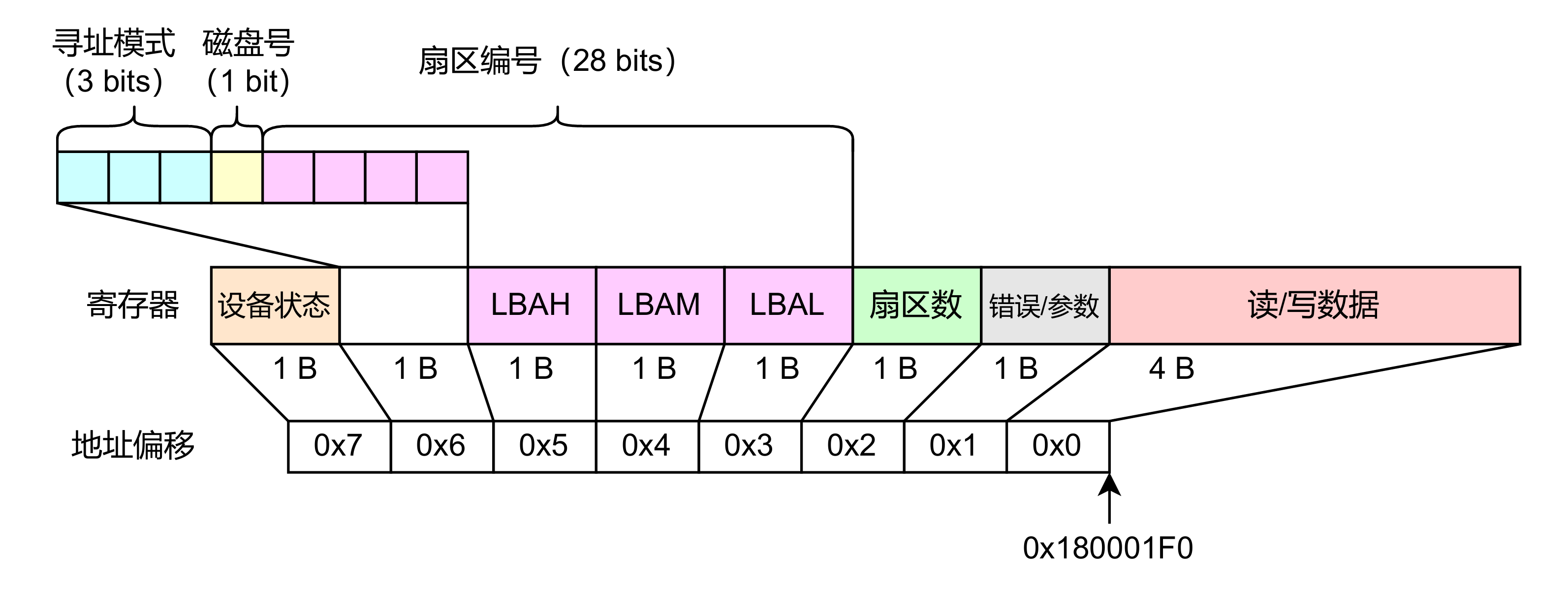
在我们的实验中，MALTA 提供的“磁盘”是一个 Intel PIIX4 IDE 控制器，其上连接 IDE 磁盘供操作系统使用。具体关于此 IDE 设备的信息，可以参考 Intel 82371AB PCI ISA IDEXcelerator (PIIX4)（第 192 页 9.2 节）。在此基础上，我们就可以实现 la64 的文件系统了。接下来，我们将了解一些读写 IDE 磁盘的基础知识。**磁盘的物理结构**下面简单介绍与磁盘相关的基本知识，首先是几个基本概念：

1. 扇区（sector）: 磁盘盘片被划分成很多扇形的区域，这些区域叫做扇区。扇区是磁盘执行读写操作的单位，一般是 512 字节。扇区的大小是一个磁盘的硬件属性。
2. 磁道（track）: 盘片上以盘片中心为圆心，不同半径的同心圆。
3. 柱面（cylinder）: 硬盘中，不同盘片相同半径的磁道所组成的圆柱面。
4. 磁头（head）: 每个磁盘有两个面，每个面都有一个磁头。当对磁盘进行读写操作时，磁头在盘片上快速移动。

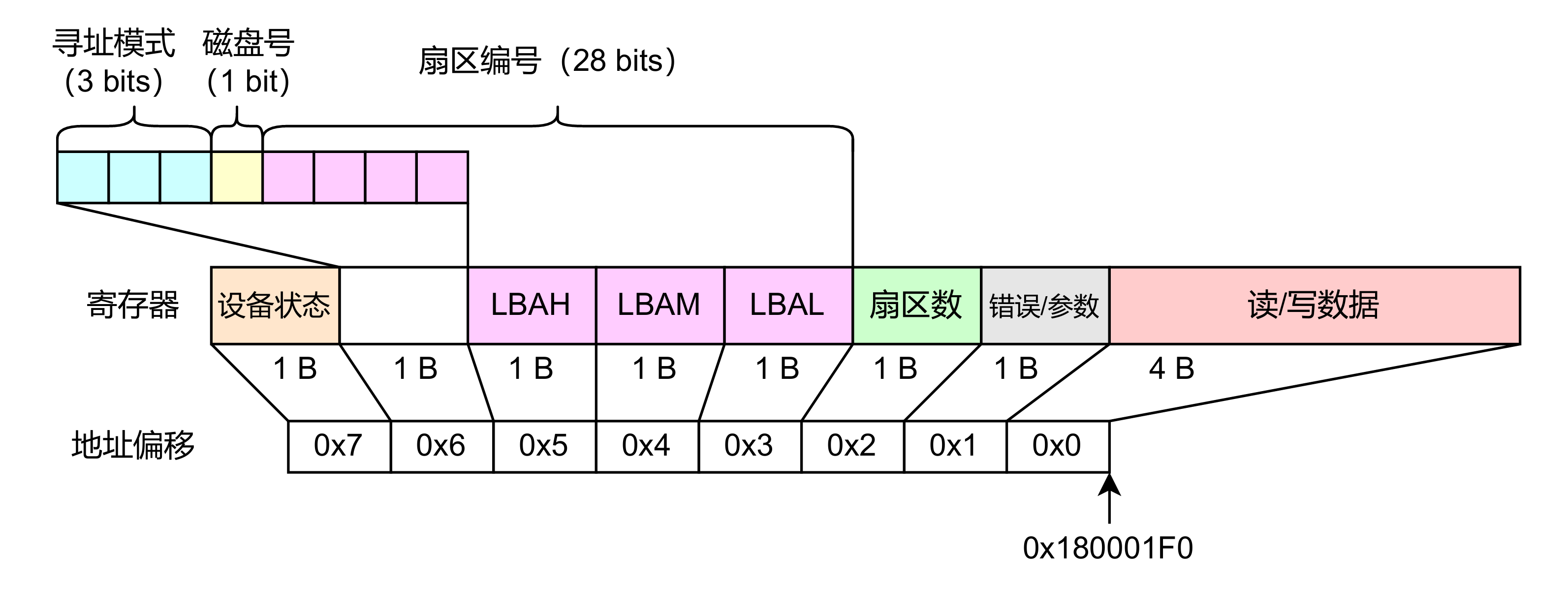
典型磁盘的基本结构如下图所示：



**IDE 磁盘操作**前文中我们提到过，扇区（sector）是磁盘读写的基本单位，MALTA 上的 PIIX4 也提供了对扇区进行操作的基本方法。通过读写 PIIX4 的特定寄存器，我们可以实现以扇区为最小单元的读写。MALTA 平台上的 PIIX4 磁盘控制器基地址为 0x180001F0，其 I/O 相关寄存器相对于该地址的偏移和对应的功能如下图所示。



在 la64 中，我们可以挂载两块 IDE 磁盘（磁盘编号为0和1），但实际上我们只使用到了磁盘编号为0的一块磁盘。对于磁盘寻址，我们可以按照柱面-磁头-扇区（Cylinder-Head-Sector，CHS）的方式来定位一个扇区。这种寻址方式符合磁盘的物理结构，但是显然三个参数导致了寻址的复杂性。在该模式下，PIIX4 I/O 的寄存器映射含义与上图中不完全相同，具体可以参考前文给出的文档。由于 CHS 模式下不方便进行寻址，因此在实验中我们采用逻辑块寻址（LogicalBlock Addressing，LBA）的方式来进行扇区寻址。在 LBA 模式下，IDE 设备将磁盘看作一个线性的字节序列，每个扇区都有一个唯一的编号，只需要设置目标扇区编号，就可以完成磁盘的寻址。在我们的实验中，扇区编号有 28 位，因此最多可以寻址 228 个扇区，即 128 GB 的磁盘空间。该模式下，磁盘操作参数和寄存器映射的关系如下图所示。



### 6.3.3 驱动程序编写

通过对 printcharc 函数的实现的分析，我们已经掌握了 I/O 操作的基本方法，那么，读写 IDE 磁盘的相关代码也就不难理解了。当我们在磁盘的指定位置读取或写入一个扇区时，需要调用 read\_sector 函数将磁盘中对应扇区的数据读到设备缓冲区中，或 write\_sector 函数来缓冲区中的数据写入磁盘。注意，与控制台串口相同，磁盘操作中所有的地址操作都需要将物理地址转换成虚拟地址。此处设备基地址对应的 kseg1 的内核虚拟地址是 0xB80001F0。在磁盘读写的流程中，我们需要反复检查 IDE 设备是否已经就绪。这是由于 IDE 外设一般不能立即完成数据操作，需要 CPU 检查 IDE 状态并等待操作完成。为此我们构建了一个检查 IDE 状态的帮手函数 wait\_ide，用于等待 IDE 上的操作就绪。

u\_int wait\_ide() {  
 u\_int flag;  
 **while** (1) {  
 flag = \*((**volatile** u\_char\*)(MALTA\_IDE\_STATUS + 0xA0000000));  
 **if** ((flag & MALTA\_IDE\_BUSY) == 0) {  
 **break**;  
 }  
 }  
 **return** flag;  
}

在 IDE 设备就绪后，我们就可以对其进行读写操作了。首先，设置操作扇区的数目，这里我们只操作一个扇区，因此设置为 1。接下来，我们依次设置操作扇区号。本实验中使用的 IDE 设备无法一次性写入操作扇区号，因此需要单独设置扇区号的各位。特别地，在设置操作扇区号的[27:24]位时，还需要同时设置扇区寻址模式和磁盘编号，因此需要通过位运算将各值组合，并一齐写入对应地址。完成一系列设置后，再次等待 IDE 设备准备就绪，并在此之后，通过系统调用读取或写入扇区。由于本实验中使用的 IDE 设备每次仅能读取或写入 4 字节，因此需要通过一个循环完成整个扇区的读取或写入，即连续向相同的地址读取或写入 4 字节。最后，检查 IDE 设备状态，确认扇区读取和写入成功。具体地，对于读取扇区，我们可以实现 read\_sector 函数，读取 diskno 号磁盘上的 secno 号扇区到 dst 指向的地址。

void read\_sector(int diskno, int secno, void\* dst)  
{  
 wait\_ide();  
  
 \*((volatileu\_char\*)(MALTA\_IDE\_NSECT +0xA0000000)) = 1;  
 \*((volatileu\_char\*)(MALTA\_IDE\_LBAL +0xA0000000)) = secno & 0xff;  
 \*((volatileu\_char\*)(MALTA\_IDE\_LBAM +0xA0000000)) = (secno >> 8) & 0xff;  
 \*((volatileu\_char\*)(MALTA\_IDE\_LBAH +0xA0000000)) = (secno >> 16) & 0xff;  
 \*((volatileu\_char\*)(MALTA\_IDE\_DEVICE +0xA0000000)) =   
 ((secno >> 24) & 0x0f) | MALTA\_IDE\_LBA | (diskno << 4);  
 \*((volatileu\_char\*)(MALTA\_IDE\_STATUS +0xA0000000)) = MALTA\_IDE\_CMD\_PIO\_READ;  
 wait\_ide();  
 **for**(inti = 0; i <SECT\_SIZE; i +=4) {  
 memcpy(dst + i,(void\*)(MALTA\_IDE\_DATA+0xA0000000),4);  
 }  
}

类似的，我们也可以实现 write\_sector 函数，将 src 指向的一个扇区的数据写入到 diskno 号磁盘的 secno 号扇区。

void write\_sector(int diskno, int secno, void\* src)  
{  
 wait\_ide();  
  
 \*((volatileu\_char\*)(MALTA\_IDE\_NSECT + 0xA0000000)) = 1;  
 \*((volatileu\_char\*)(MALTA\_IDE\_LBAL + 0xA0000000)) = secno & 0xff;  
 \*((volatileu\_char\*)(MALTA\_IDE\_LBAM + 0xA0000000))= (secno >> 8) & 0xff;  
 \*((volatileu\_char\*)(MALTA\_IDE\_LBAH + 0xA0000000))= (secno >> 16) & 0xff;  
 \*((volatileu\_char\*)(MALTA\_IDE\_DEVICE + 0xA0000000)) = ((secno >> 24) & 0x0f) | MALTA\_IDE\_LBA | (diskno << 4);  
 \*((volatileu\_char\*)(MALTA\_IDE\_STATUS + 0xA0000000)) = MALTA\_IDE\_CMD\_PIO\_WRITE;  
 wait\_ide();  
  
 **for**(int i = 0; i < SECT\_SIZE; i += 4) {  
 memcpy((void\*)(MALTA\_IDE\_DATA + 0xA0000000), src + i, 4);  
 }  
}

实现了磁盘驱动，我们可以尝试对磁盘进行读写测试。QEMU 允许通过启动参数来指定 IDE 磁盘镜像。编译生成的磁盘镜像文件位于 target/fs.img，我们可以在运行命令中加上 -driveid=ide0,file=target/fs.img,if=ide,format=raw 挂载该磁盘镜像。实验中我们挂载了两个磁盘镜像，其中一个是我们编译生成的磁盘镜像，另一个是空的磁盘镜像，完整命令如下：

qemu-system-mipsel \  
 -cpu 4Kc \  
 -m 64 -nographic\  
 -M malta \  
 -drive id=ide0,file=target/fs.img,**if**=ide,format=raw \  
 -drive id=ide1,file=target/empty.img,**if**=ide,format=raw \  
 -no-reboot \  
 -kernel target/mos

我们通过 Makefile 简化了这个过程，可以直接运行make run来执行上述命令。

## 6.4 文件系统结构

实现了 IDE 磁盘的驱动，我们就有了在磁盘上实现文件系统的基础。接下来我们设计整个文件系统的结构，并在磁盘和操作系统中分别实现对应的结构。

### 6.4.1 磁盘文件系统布局

磁盘空间的基本布局如下图所示。



图中出现的 Block 是磁盘块。不同于扇区，磁盘块是一个虚拟概念，是操作系统与磁盘交互的最小单位；操作系统将相邻的扇区组合在一起，形成磁盘块进行整体操作，减小了因扇区过多带来的寻址困难；磁盘块的大小由操作系统决定，一般由2的幂次个扇区构成。而扇区是真实存在的，是磁盘读写的基本单位，与操作系统无关。从图中可以看到，la64 操作系统把磁盘最开始的一个磁盘块（4096字节）当作引导扇区和分区表使用。接下来的一个磁盘块作为超级块（SuperBlock），用来描述文件系统的基本信息，如Magic Number、磁盘大小以及根目录的位置。la64 操作系统中超级块的结构如下：

**struct** **Super** {  
 u\_int s\_magic;  
 u\_int s\_nblocks;  
 *// Magic number: FS\_MAGIC*  
 **struct** **File** s\_root; *// Root directory node*  
};

其中每个域的意义如下：

* s\_magic：魔数，为一个常量，用于标识该文件系统。
* s\_nblocks：记录本文件系统有多少个磁盘块，在本文件系统中为1024。
* s\_root：根目录，其 f\_type 为 FTYPE\_DIR，f\_name 为“/”。

通常采用两种数据结构来管理可用的资源：链表和位图。在 Lab2 “内存管理”和 Lab3 “进程与异常”实验中，我们使用了链表来管理空闲内存资源和进程控制块。在文件系统中，我们将使用位图(Bitmap) 法来管理空闲的磁盘资源，用一个二进制位 bit 标识磁盘中的一个磁盘块的使用情况（1表示空闲，0表示占用）。这里我们参考 tools/fsformat.c 来介绍文件系统标记空闲块的机制。tools/fsformat.c 是用于创建符合我们定义的文件系统镜像的工具，可以将多个文件按照内核所定义的文件系统写入到磁盘镜像中。通过观察头文件和 tools/Makefile ，我们可以看出，tools/fsformat.c 的编译过程与其他文件有所不同，其使用的是 Linux 下的 gcc 编译器，而非交叉编译器。编译生成的 fsformat 运行于 Linux 宿主机上，专门用于创建磁盘镜像文件。生成的镜像文件 fs.img 可以模拟与真实的磁盘文件设备的交互。在写入文件之前，tools/fsformat.c 的 init\_disk 函数，将所有的块都标为空闲块：

nbitblock = (NBLOCK + BLOCK\_SIZE\_BIT- 1) / BLOCK\_SIZE\_BIT;  
 **for** (i = 0; i < nbitblock; ++ i)   
 {  
 disk[2 + i].type = BLOCK\_BMAP;  
 }  
 **for** (i = 0; i < nbitblock; ++i)   
 {  
 memset(disk[2 + i].data, 0xff, BLOCK\_SIZE);  
 }  
 **if** (NBLOCK != nbitblock \* BLOCK\_SIZE\_BIT)   
 {  
 diff = NBLOCK % BLOCK\_SIZE\_BIT / 8;  
 memset(disk[2 + (nbitblock- 1)].data + diff, 0x00, BLOCK\_SIZE- diff);  
 }

nbitblock 表示为了用位图标识整个磁盘上所有块的使用情况所需要的磁盘块（bitblock，位图块）的数量。紧接着，我们使用 memset 将位图中的每一个字节（Byte）都设成 0xff，即将所有位图块的每一位都设为 1，表示磁盘块处于空闲状态。如果位图还有剩余，不能将最后一块位图块中靠后的一部分内容标记为空闲，因为这些位所对应的磁盘块并不存在，是不可使用的。因此，将所有的位图块的每一位都置为 1 之后，还需要根据实际情况，将位图不存在的部分设为 0。相应地，在 la64 操作系统中，文件系统也需要根据位图来判断和标记磁盘的使用情况。fs/fs.c 中的 block\_is\_free 函数就用来通过位图中的特定位来判断指定的磁盘块是否被占用。

int block\_is\_free(u\_int blockno)  
{  
 **if**(super == 0 || blockno >= super->s\_nblocks)  
 {  
 **return** 0;  
 }  
 **if**(bitmap[blockno / 32] & (1 << (blockno % 32)))  
 {  
 **return** 1;  
 }  
 **return** 0;  
}

### 6.4.2 文件系统详细结构

操作系统要想管理一类资源，就得有相应的数据结构。对于描述和管理文件来说，一般使用文件控制块（File结构体）。其定义如下：

**struct** **File** {  
 char f\_name[MAXNAMELEN];*//filename*  
 uint32\_t f\_size; *//filesizeinbytes*  
 uint32\_t f\_type; *//filetype*  
 uint32\_tf\_direct[NDIRECT];  
 uint32\_tf\_indirect;  
  
 **struct** **File** \*f\_dir;*//thepointertothedirwherethisfileisin,validonlyinmemory.*  
 char f\_pad[FILE\_STRUCT\_SIZE - MAXNAMELEN - (3 + NDIRECT) \* 4 - **sizeof**(void \*)];  
}\_\_attribute\_\_((aligned(4),packed));

结合文件控制块的示意图，我们对各个域进行解读：f\_name 为文件名称，文件名的最大长度 MAXNAMELEN 值为 128。f\_size 为文件的大小，单位为字节。f\_type 为文件类型，有普通文件(FTYPE\_REG)和目录(FTYPE\_DIR)两种。f\_direct[NDIRECT] 为文件的直接指针，每个文件控制块设有 10 个直接指针，用来记录文件的数据块在磁盘上的位置。每个磁盘块的大小为 4 KB，也就是说，这十个直接指针能够表示最大 40 KB的文件，而当文件的大小大于 40 KB时，就需要用到间接指针。f\_indirect 指向一个间接磁盘块，用来存储指向文件内容的磁盘块的指针。为了简化计算，我们不使用间接磁盘块的前十个指针。f\_dir 指向文件所属的文件目录。f\_pad 则是为了让整数个文件结构体占用一个磁盘块，填充结构体中剩下的字节。



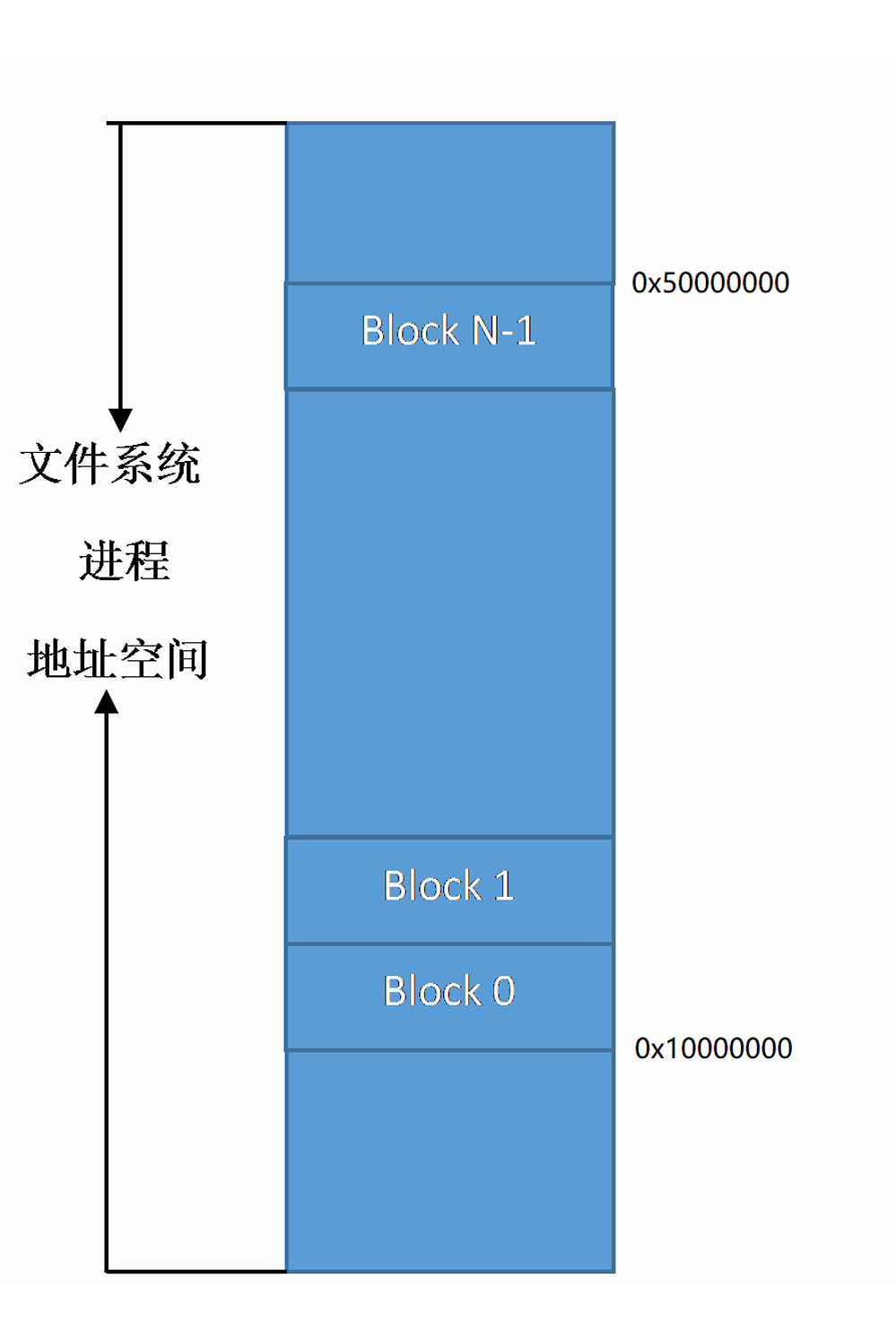
对于普通的文件，其指向的磁盘块存储着文件内容，而对于目录文件来说，其指向的磁盘块存储着该目录下各个文件对应的文件控制块。当我们要查找某个文件时，首先从超级块中读取根目录的文件控制块，然后沿着目标路径，挨个查看当前目录所包含的文件是否与下一级目标文件同名，如此便能查找到最终的目标文件。请阅读 tools/fsformat.c 和 fs/Makefile，掌握如何将文件和目录按照文件系统的格式写入磁盘，了解文件系统结构的具体细节，学会添加自定义文件进入磁盘镜像。（fsformat.c 中的主函数十分灵活，可以通过修改命令行参数来生成不同的镜像文件）

### 6.4.3 块缓存

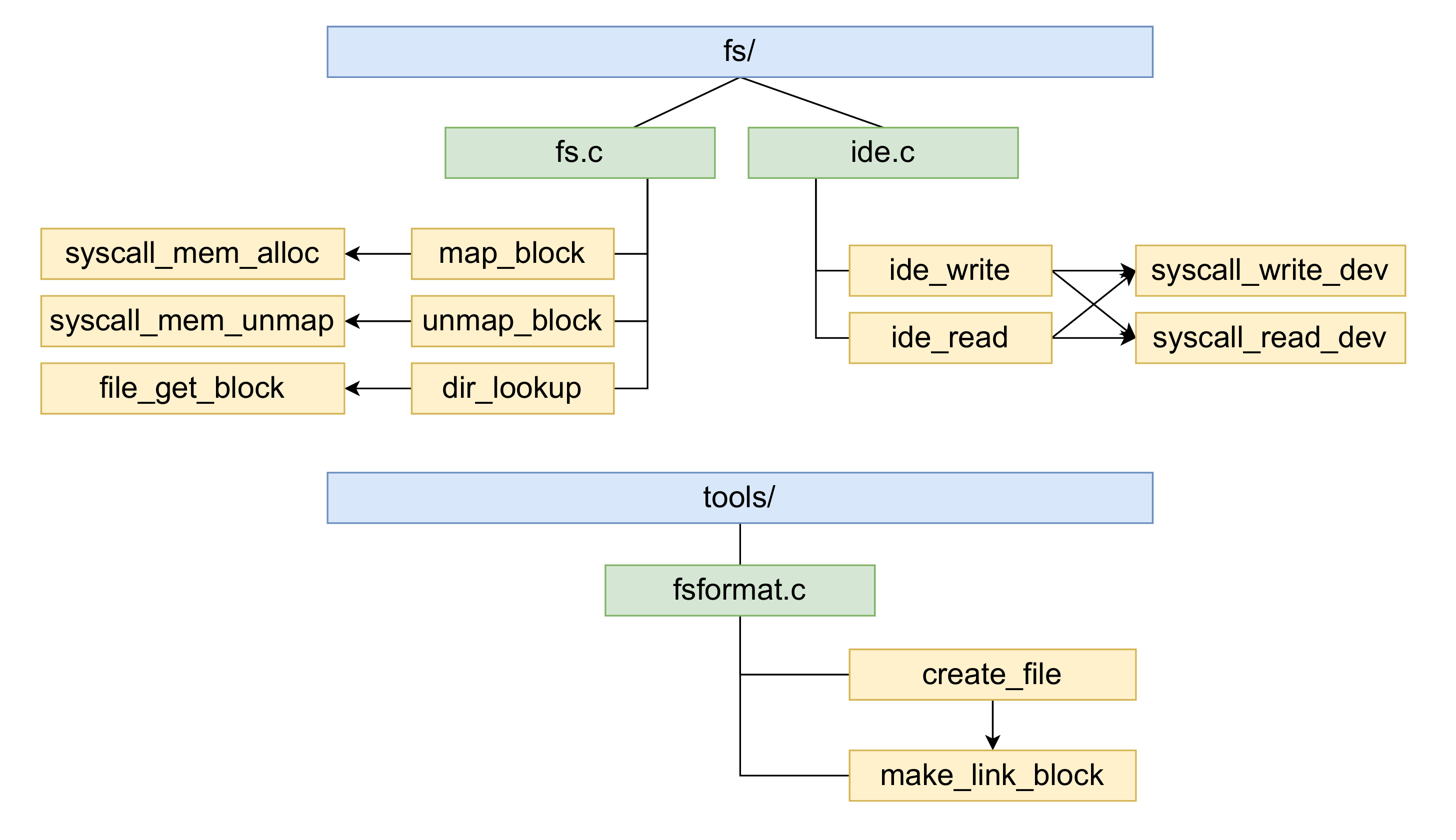
块缓存指的是借助虚拟内存来实现磁盘块缓存的设计。由于对于硬盘 I/O 操作一次只能读取少量数据，同时频繁读写磁盘会拖慢系统，故常用块缓存将数据暂存于内存，减少磁盘访问次数，从而加快数据读取。la64 操作系统中，文件系统服务是一个用户进程（将在下文介绍），一个进程可以拥有 4 GB的虚拟内存空间，将 DISKMAP 到 DISKMAP + DISKMAX 这一段虚存地址空间 (0x10000000-0x4FFFFFFF) 作为缓冲区，当磁盘读入内存时，用来映射相关的页。DISKMAP 和 DISKMAX 的值定义在 fs/serv.h 中：

#define DISKMAP 0x10000000  
#define DISKMAX 0x40000000

为了建立起磁盘地址空间和进程虚存地址空间之间的缓存映射，我们采用的设计如下图所示。



当把一个磁盘块中的内容载入到内存中时，需要为之分配对应的物理内存；当结束使用这一磁盘块时，需要释放对应的物理内存以回收操作系统资源。fs/fs.c 中的 map\_block 函数和 unmap\_block 函数实现了这一功能。read\_block 函数和 write\_block 函数用于读写磁盘块。read\_block 函数将指定编号的磁盘块读入到内存中，首先检查这块磁盘块是否已经在内存中，如果不在，先分配一页物理内存，然后调用 ide\_read 函数来读取磁盘上的数据到对应的虚存地址处。file\_get\_block 函数用于将某个指定的文件指向的磁盘块读入内存。其主要分为 2 个步骤：首先为即将读入内存的磁盘块分配物理内存，然后使用 read\_block 函数将磁盘内容以块为单位读入内存中的相应位置。这两个步骤对应的函数都借助了系统调用来完成。在完成块缓存部分之后我们就可以实现文件系统中的一些文件操作了。这里我们给出了文件系统结构中部分函数可能的调用参考，如下图所示，希望同学们认真理解每个文件、函数的作用和之间的关系。



## 6.5 文件系统的用户接口

在文件系统建立之后，还需要向用户提供相关的使用接口。la64 操作系统内核符合一个典型的微内核的设计，文件系统属于用户态进程，以服务的形式供其他进程调用。这个过程中，不涉及了不同进程之间通信的问题，也涉及了文件系统如何隔离底层的文件系统实现，抽象地表示一个文件的问题。首先，我们引入文件描述符（filedescriptor）作为用户程序管理、操作文件的基础。

### 6.5.1 文件描述符

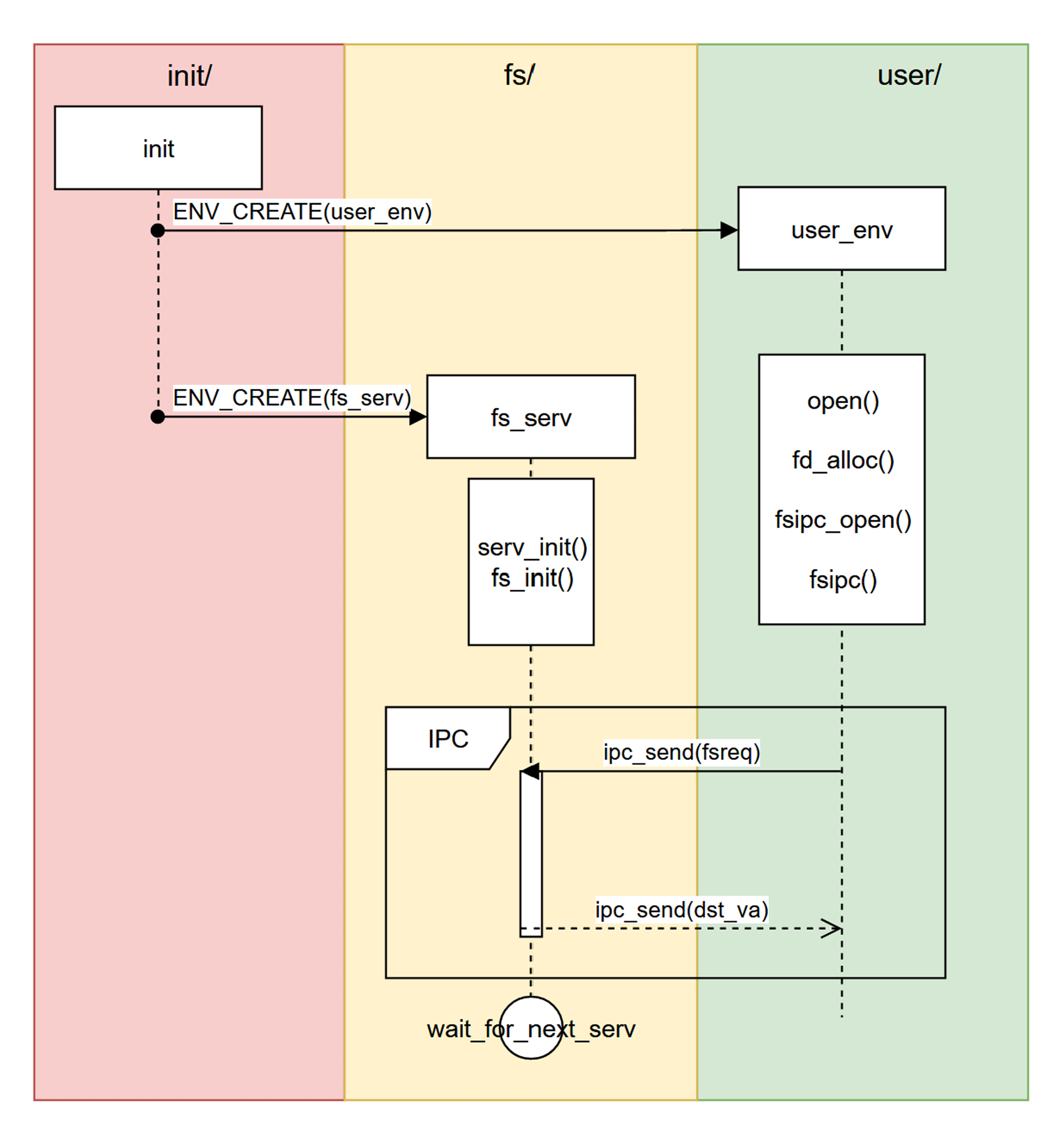
当用户进程试图打开一个文件时，文件系统服务进程需要一个文件描述符来存储文件的基本信息和用户进程中关于文件的状态；同时，文件描述符也起到描述用户对于文件操作的作用。当用户进程向文件系统发送打开文件的请求时，文件系统进程会将这些基本信息记录在内存中，然后由操作系统将用户进程请求的地址映射到同一个存储了文件描述符的物理页上（此部分代码位于 serv.c 的 serve\_open 函数内），因此一个文件描述符至少需要独占一页的空间。当用户进程获取了文件大小等基本信息后，再次向文件系统发送请求将文件内容映射到指定内存空间中。阅读 user/lib/file.c 时，我们会发现很多函数中都会将一个 struct Fd \* 型的指针转换为 struct Filefd\* 型的指针，这是基于 C 语言中关于指针的强制类型转换来实现的。该类型转换并不改变指针的值，仅改变程序对地址处数据的解释方式两个结构体的定义如下：

**struct** **Fd**{  
 u\_int fd\_dev\_id;  
 u\_int fd\_offset;  
 u\_int fd\_omode;  
};  
  
**struct** **Filefd**{  
 **struct** **Fd** f\_fd;  
 u\_int f\_fileid;  
 **struct** **File** f\_file;  
};

对于类型为 Fd 的一段内存，该段内存开头被解释为三个 u\_int 类型的数据。将其强制转换为 Filefd 类型之后，由于 Filefd 第一个成员是 Fd 类型，对这段内存开始部分的解释依旧可以是三个 u\_int 类型的数据（通过f\_fd成员即可访问到）。同时第二个成员开始的内存将会按照 FileFd 的定义进行解释。若将 Filefd 类型强制转换为 Fd 将会遮盖掉不同细分类别的文件描述符的不同，有利于统一操作。当要读取一个大文件中间的一小部分内容时，一个简单的做法是从头开始查找，但这样的开销很大；此外，在多次读写同一文件描述符期间，我们也希望能够从文件中前一次读写完毕的位置开始，继续读写数据。因此，文件描述符中需要维护一个指针，以帮助我们在文件中定位；在实现 read、write 和 seek 等操作时，也需要更新该指针的值。

### 6.5.2 文件系统服务

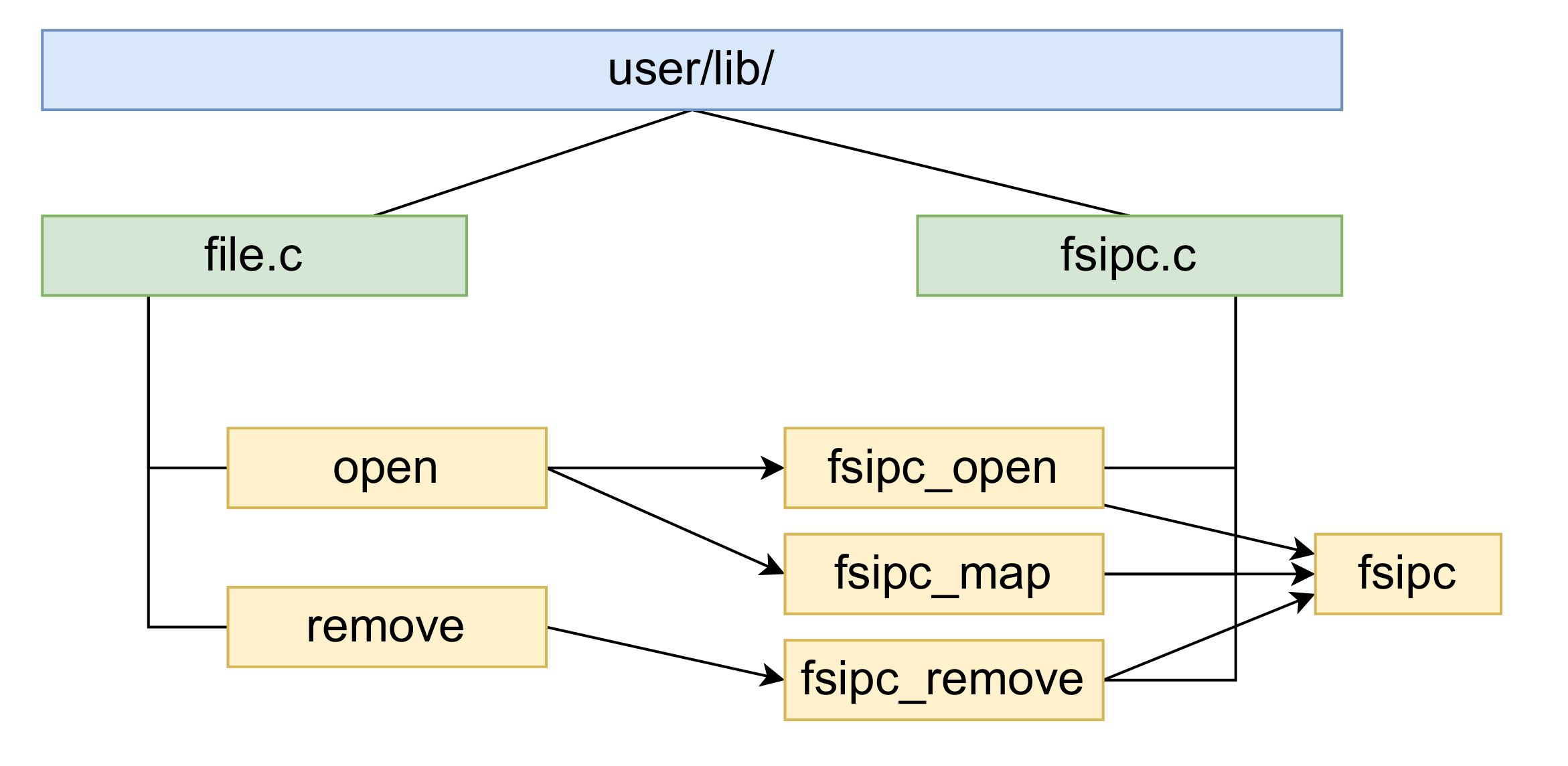
la64 操作系统中的文件系统服务通过 IPC 的形式供其他进程调用，进行文件读写操作。具体来说，在内核开始运行时，就启动了文件系统服务进程 ENV\_CREATE(fs\_serv)，用户进程需要进行文件操作时，使用 ipc\_send、ipc\_recv 与 fs\_serv 进行交互，完成操作。fs/serv.c 中服务进程的主函数首先调用了 serve\_init 函数准备好全局的文件打开记录表 opentab，然后调用 fs\_init 函数来初始化文件系统。fs\_init 函数首先通过读取超级块的内容获知磁盘的基本信息，然后检查磁盘是否能够正常读写，最后调用 read\_bitmap 函数检查磁盘块上的位图是否正确。执行完文件系统的初始化后，调用 serve 函数，文件系统服务开始运行，等待其他程序的请求。下图以 UML 时序图的形式在宏观层面上展示了一个用户进程请求文件系统服务的过程（以 open 为例）。其中 user\_env 所加载的程序不仅可以是实验源码已给出的 fstest.c, 也可以是其他以 main 为入口函数的用户程序，你可以通过这种方式对文件系统服务进行测试。其中 IPC 系统调用的细节请参考 Lab4 的相关内容。（三种颜色不仅区分三个不同的程序，也表示进程执行的代码在我们的操作系统被载入内存前所处的文件位置）



文件系统支持的请求类型定义在 user/include/fsreq.h 中，包含以下几种：

**enum** {  
 FSREQ\_OPEN,  
 FSREQ\_MAP,  
 FSREQ\_SET\_SIZE,  
 FSREQ\_CLOSE,  
 FSREQ\_DIRTY,  
 FSREQ\_REMOVE,  
 FSREQ\_SYNC,  
 MAX\_FSREQNO,  
};

用户程序在发出文件系统操作请求时，将请求的内容放在对应的结构体中进行消息的传递，fs\_serv 进程收到其他进行的 IPC 请求后，IPC 传递的消息包含了请求的类型和其他必要的参数，根据请求的类型执行相应的文件操作（文件的增、删、改、查等），将结果重新通过 IPC 反馈给用户程序。文件 user/lib/fsipc.c 中定义了请求文件系统时用到的 IPC 操作，user/lib/file.c 文件中定义了用户程序读写、创建、删除和修改文件的接口。完成以下练习，实现删除指定路径的文件的功能。这里我们给出了文件系统的用户接口中部分函数可能的调用参考图，希望同学们体会函数之间的调用关系、理解文件系统中用户接口的实现过程。



## 6.6 实验结果展示

本实验有三个阶段性测试可以检验你实现的文件系统的正确性。

### 6.6.1 IDE 磁盘交互

在 init/init.c 中仅启动 devtst 进程，使用 make && make run 编译运行内核，将对 Console 和 IDE 磁盘交互进行测试

ENV\_CREATE(user\_devtst);

也可以使用 make test lab=5\_1直接构建。make run 后，输入一个字符串 01234567890123 并按下回车，正确结果如下：

devtst begin  
01234567890123  
syscall\_read\_dev **is** good  
end of devtst  
dev address **is** ok

### 6.6.2 文件系统测试

在 init/init.c 中仅启动文件系统服务进程，该进程在正式开始服务前，会执行 fs/check.c 中的单元测试。

ENV\_CREATE(fs\_serv);

正确结果如下：

FS **is** running  
superblock **is** good  
read\_bitmap **is** good

### 6.6.3 文件系统服务测试

完成全部学习后，启动一个 fstest 进程和文件系统服务进程：

ENV\_CREATE(user\_fstest);  
 ENV\_CREATE(fs\_serv);

就能开始对文件系统的检测，运行文件系统服务，等待应用程序的请求。注意：我们此时必须将文件系统进程作为第二个进程启动，其原因是我们在 user/lib/fsipc.c 中定义的文件系统 IPC 请求的目标指定为第二个创建的进程。

FS **is** running  
superblock **is** good  
read\_bitmap **is** good  
serve\_open 00000800 ffff000 0x2  
open **is** good  
read **is** good  
serve\_open 00000800 ffff000 0x0  
open again: OK  
read again: OK  
file rewrite **is** good  
serve\_open 00000800 ffff000 0x0  
file remove: OK

# 7 实验六：管道与 SHELL

## 7.1 实验目的

1. 掌握管道的原理与底层细节
2. 实现管道的读写
3. 复述管道竞争情景
4. 实现基本 shell
5. 实现 shell 中涉及管道的部分

## 7.2 管道

在 Lab4 中，我们已经学习过一种进程间通信(IPC,Inter-ProcessCommunication)的方式—共享内存。而今天我们要学的管道，其实也是进程间通信的一种方式。

### 7.2.1 初窥管道

管道是一种典型的进程间单向通信的方式。管道分有名管道和匿名管道两种，匿名管道只能在具有公共祖先的进程之间使用，且通常使用在父子进程之间。在 la64 中，我们要实现匿名管道。本章后续所称的管道，没有特别说明，均指匿名管道。在 Unix 中，管道由 int pipe(intfd[2]) 函数创建，成功创建管道返回 0，参数中的 fd 用来保存读写端的文件描述符，fd[0] 对应读端，fd[1] 对应写端。为了更好地理解管道实现的原理，我们先来做一个小实验体会一下。

#include *<stdlib.h>*  
#include *<unistd.h>*  
  
int fildes[2];  
char buf[200];  
int status;  
  
int main()  
{  
 status = pipe(fildes);  
 **if** (status == -1)  
 {  
 printf("error**\n**");  
 }  
  
 **switch** (fork())  
 {  
 **case** -1:  
 **break**;  
   
 **case** 0: */\*子进程-作为管道的读者\*/*  
 close(fildes[1]); */\*关闭不用的写端\*/*  
 read(fildes[0],buf, 100); */\*从管道中读数据\*/*  
 printf("child-processread:%s",buf); */\*打印读到的数据\*/*  
 close(fildes[0]); */\*读取结束，关闭读端\*/*  
 exit(EXIT\_SUCCESS);  
  
 **default**:  
 close(fildes[0]); */\*关闭不用的读端\*/*  
 write(fildes[1],"Helloworld**\n**",12); */\*向管道中写数据\*/*  
 close(fildes[1]); */\*写入结束，关闭写端\*/*  
 exit(EXIT\_SUCCESS);  
 }  
}

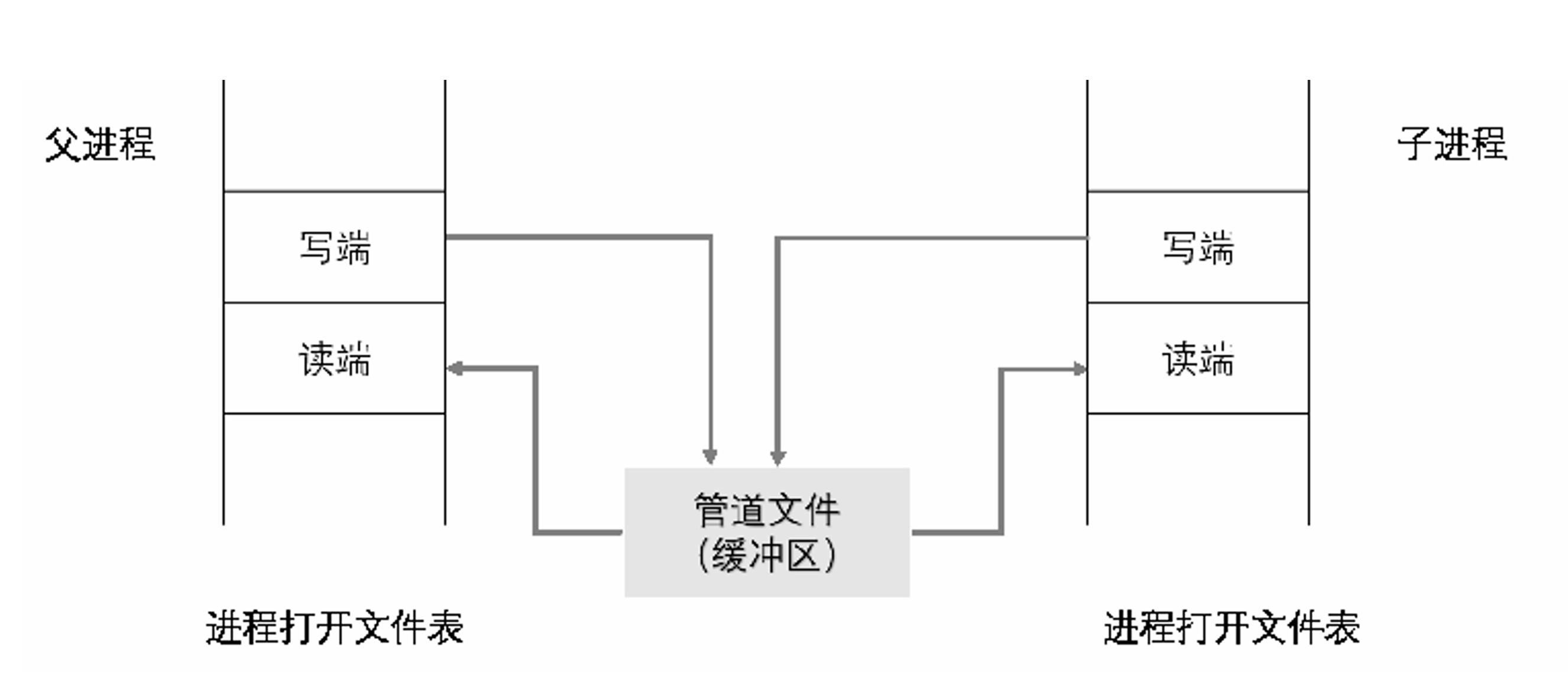
你可以将示例代码复制到 Linux 环境，使用命令 gcctest\_pipe.c && ./a.out 来执行。示例代码实现了从父进程向管道中写入消息 Hello,world，子进程从管道中读出数据并打印到屏幕上。它演示了管道在父子进程之间通信的基本用法：父进程在 pipe 函数之后，调用 fork 来产生一个子进程，之后在父子进程中各自执行不同的操作：关掉自己不会用到的管道端，然后进行相应的读写操作。在示例代码中，父进程操作写端，而子进程操作读端。从本质上说，管道是一种只存在于内存中的文件。在 UNIX 以及 la64中，父进程调用 pipe 函数时，会打开两个新的文件描述符：一个表示只读端，另一个表示只写端，两个描述符都映射到了同一片内存区域。在 fork 的配合下，子进程复制父进程的两个文件描述符，从而在父子进程间形成了四个（父子各拥有一读一写）指向同一片内存区域的文件描述符，父子进程可根据需要关掉自己不用的一个，从而实现父子进程间的单向通信管道，这也是匿名管道只能用在具有亲缘关系的进程间通信的原因。

### 7.2.2 la64 中 pipe 的使用与实现

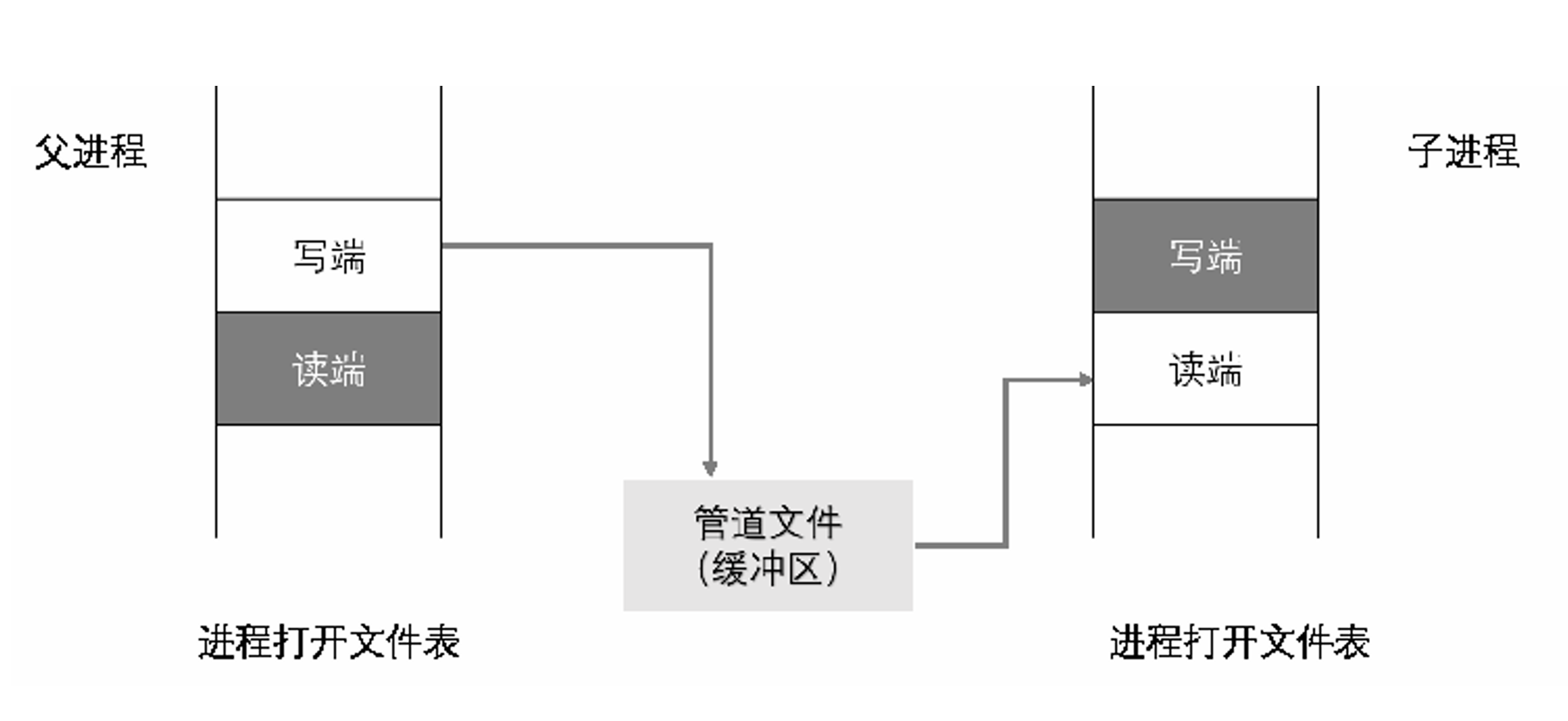
在 la64 中，管道的使用和 UNIX 系统的示例代码逻辑相同。la64 中使用管道的示例代码为 user/testpipe.c。示例代码中先使用函数 pipe(int p[2]) 创建了管道，读端的文件描述符（fd）编号为 p[0]，写端的文件描述符编号为 p[1]。之后使用 fork() 创建子进程，注意这时父子进程使用自己的 p[0] 和 p[1] 访问到的内存区域是一致的。之后子进程关闭了自己的 p[1]，从 p[0] 读；父进程关闭了自己的 p[0],从 p[1] 写入管道。在管道读写的过程中，我们需要保证写端对管道的写入对读端可见，也就是保证父子进程通过管道访问的内存相同。下面我们通过对 user/lib/pipe.c 中的 pipe 函数的具体实现来介绍如何实现这一需求。

int pipe(int pfd[2]) {  
 int r;  
 void \*va;  
 **struct** **Fd** \*fd0, \*fd1;  
  
 */\* Step 1: Allocate the file descriptors. \*/*  
 **if** ((r = fd\_alloc(&fd0)) < 0 || (r = syscall\_mem\_alloc(0, fd0, PTE\_D | PTE\_LIBRARY)) < 0) {  
 **goto** err;  
 }  
  
 **if** ((r = fd\_alloc(&fd1)) < 0 || (r = syscall\_mem\_alloc(0, fd1, PTE\_D | PTE\_LIBRARY)) < 0) {  
 **goto** err1;  
 }  
  
 */\* Step 2: Allocate and map the page for the 'Pipe' structure. \*/*  
 va = fd2data(fd0);  
 **if** ((r = syscall\_mem\_alloc(0, (void \*)va, PTE\_D | PTE\_LIBRARY)) < 0) {  
 **goto** err2;  
 }  
 **if** ((r = syscall\_mem\_map(0, (void \*)va, 0, (void \*)fd2data(fd1), PTE\_D | PTE\_LIBRARY)) <  
 0) {  
 **goto** err3;  
 }

在 pipe 中，首先分配两个文件描述符 fd0 和 fd1 并为其分配空间，然后给 fd0 对应的虚拟地址分配一页物理内存，再将 fd1 对应的虚拟地址映射到这一页物理内存。我们曾在 Lab4 填写 duppage 函数时介绍过这类页面，即“共享页面”。共享页面是具有权限位 PTE\_LIBRARY 的页面，需要保持共享可写的状态，使得父子进程对其进行修改的结果相互可见。当父子进程试图写共享页面时，直接在该页面上进行写操作即可。仔细观察 pipe 中出现的权限位 PTE\_LIBRARY，根据上述提示检查你在 Lab4 实现的 user/lib/fork.c 中的 duppage 函数，保证它能根据父页面的不同权限位，为子页面设置对应的权限位，以此区分实现 Lab4 所需的写时复制机制或者 Lab6 中管道所需的共享页面机制。下面我们使用下图来表示父子进程与管道的数据缓冲区的关系：



实际上，在父子进程中各自 close 掉不再使用的端口后，父子进程与管道缓冲区的关系如图所示。



下面我们来讲一下 struct Pipe，并开始着手填写操作管道端的函数。

**struct** **Pipe** {  
 u\_int p\_rpos; *// read position*  
 u\_int p\_wpos; *// write position*  
 u\_char p\_buf[PIPE\_SIZE]; *// data buffer*  
};

在 Pipe 结构体中,p\_rpos 给出了下一个将要从管道读的数据的位置，而 p\_wpos 给出了下一个将要向管道写的数据的位置。只有读者可以更新 p\_rpos，同样，只有写者可以更新 p\_wpos，读者和写者通过这两个变量的值进行读写的协调。一个管道有 PIPE\_SIZE(32 Byte) 大小的缓冲区。这个 PIPE\_SIZE 大小的缓冲区发挥的作用类似于环形缓冲区，所以下一个要读或写的位置 i 实际上是 i % PIPE\_SIZE。读者在从管道读取数据时，要将 p\_buf[p\_rpos%PIPE\_SIZE] 中的数据拷贝走，然后读指针自增 1。但是需要注意的是，管道的缓冲区此时可能还没有被写入数据。所以如果管道数据为空，即当 p\_rpos >= p\_wpos 时，应该进程切换到写者运行。类似于读者，写者在向管道写入数据时，也是将数据存入 p\_buf[p\_wpos%PIPE\_SIZE]，然后写指针自增 1。需要注意管道的缓冲区可能出现满溢的情况，所以写者必须得在 p\_wposp\_rpos < PIPE\_SIZE 时方可运行，否则要一直挂起。上面这些还不能保证读者写者一定能顺利完成管道操作。假设这样的情景：管道写端已经全部关闭，读者读到缓冲区有效数据的末尾，此时有 p\_rpos = p\_wpos。按照上面的做法，我们这里应当切换到写者运行。但写者进程已经结束，进程切换就造成了死循环，这时候读者进程如何知道应当退出了呢？为了解决上面提出的问题，我们必须知道管道的另一端是否已经关闭。不论是读者还是写者进程，我们都需要对管道另一端的状态进行判断：当出现缓冲区空或满的情况时，要根据另一端是否关闭来判断是否要返回。如果另一端已经关闭，进程返回0即可；如果没有关闭，则切换进程运行。管道的关闭涉及到以下几个函数：fd.c 中的 close，fd\_close 以及 pipe.c 中的 pipe\_close，请自行阅读并结合分析。在 la64 中，我们使用 static int \_pipe\_is\_closed(struct Fd \*fd, struct Pipe \*p) 函数来判断管道的另一端是否已经关闭。这个函数的核心，就是下面要讲的恒成立等式。在之前的图中我们没有明确画出文件描述符所占的页，但实际上，对于每一个匿名管道而言，我们分配了三页空间：一页是读数据的文件描述符 rfd，一页是写数据的文件描述符 wfd，剩下一页是被两个文件描述符共享的管道数据缓冲区 pipe。既然管道数据缓冲区是被两个文件描述符所共享的，我们很直观地就能得到一个结论：如果有 1 个读者，1 个写者，那么管道将被引用 2 次（正如之前所给图所示）。pageref 函数能得到页的引用次数，所以有下面这个等式成立：

pageref(rfd) + pageref(wfd) = pageref(pipe)

我们将借助这个恒等式判断管道另一端是否已经关闭。假设现在读者进程正在运行，而管道写者进程已经结束了，那么此时就应该有：pageref(wfd) = 0。所以就有 pageref(rfd) = pageref(pipe)。因此，只要判断这个等式是否成立就可以得知写端是否关闭。对写者来说同理。

### 7.2.3 管道关闭的正确判断

la64 操作系统采用的是时间片轮转调度的进程调度算法，有关这点，你应在 Lab3 中就深有体会了。这种抢占式的进程管理意味着，用户进程随时可能会被打断。当然，如果进程间是孤立的，随时打断也没有关系。但当多个进程共享同一个变量时，不同的进程执行顺序有可能产生完全不同的结果，造成运行结果的不确定性。而进程通信需要共享同一块内存（不论是管道还是共享内存），所以我们要对进程中共享变量的读写操作有足够高的警惕。因此，因为管道本身的共享性质，在当前这种不加锁控制的情况下，无法保证 \_pipe\_is\_closed 用于管道另一端关闭的判断一定返回正确的结果。进程通过 pipe\_close 函数来关闭管道的端口，该函数的实质是通过两次系统调用 unmap 解除文件描述符 fd 和数据缓存区 pipe 的映射。但是由于进程切换的存在，并不能保证两次系统调用可以在同一进程时间片内被执行，两次系统调用之间可能因为进程切换而被打断。所以，fd 和对 pipe 的 pp\_ref 也不能保证同步被写入，这将影响我们判断管道是否关闭的正确性。结合下文的代码，考虑以下场景：

pipe(p);  
 **if**(fork() == 0 )  
 {  
 close(p[1]);  
 read(p[0], buf, **sizeof** buf);  
 }  
 **else**  
 {  
 close(p[0]);  
 write(p[1], "Hello", 5);  
 }

* 假设 fork 结束后，子进程先执行。时钟中断产生在 close(p[1]) 与 read 之间，父进程开始执行。
* 父进程在 close(p[0]) 过程中，已经解除了 p[0] 对 pipe 的映射(unmap)，还没有来得及解除对 p[0] 的映射。假设这时时钟中断产生，进程调度后子进程接着执行。
* 注意此时各个页的引用情况：pageref(p[0]) = 2(因为父进程还没有解除对 p[0] 的映射)，而 pageref(p[1]) = 1(因为子进程已经关闭了 p[1])。但注意，此时 pipe 的 pageref 是 2，子进程中 p[0] 引用了 pipe，同时父进程中 p[0] 刚解除对 pipe 的映射，所以在父进程中也只有 p[1] 引用了 pipe。
* 子进程执行 read，read 中首先判断写者是否关闭。比较 pageref(pipe) 与 pageref(p[0]) 之后发现它们都是 2，说明写端已经关闭，于是子进程退出。

从上述场景中可以发现，由于时间中断发生在父进程 unmap pipe 和 unmap fd 之间，子进程根据 pageref(pipe) == pageref(p[0]) 这一等式错误判断了写端已经关闭。同时，我们也可发现，父进程 pipe\_close 的运行逻辑是首先解除 pipe 的映射，再解除 fd 的映射。但若我们在 pipe\_close 函数中调整 unmap 的顺序，即首先解除 fd 的映射，再解除 pipe 的映射，可以避免上述场景的错误情况，接下来，我们将解释调整顺序后可行的原因。\_pipe\_is\_closed 函数返回正确结果的条件其实只是：

* 写端关闭当且仅当 pageref(p[0]) == pageref(pipe);
* 读端关闭当且仅当 pageref(p[1]) == pageref(pipe);

比如说第一个条件，写端关闭时，当然有 pageref(p[0]) == pageref(pipe)。但是由于进程切换的存在，我们无法确保当 pageref(p[0]) == pageref(pipe) 时，写端关闭。正面如果不好解决问题，我们可以考虑从其逆否命题着手，即我们要确保：当写端没有关闭的时候，pageref(p[0]) ̸= pageref(pipe)。我们考虑之前那个预想之外的情景，它出现的最关键原因在于：pipe 的引用次数总比 fd 要高。当管道的 close 进行到一半时，若先解除 pipe 的映射，再解除 fd 的映射，就会使得 pipe 引用次数的 -1 先于 fd。这就导致在两个 unmap 的间隙，会出现 pageref(pipe) == pageref(fd) 的情况。若调换 fd 和 pipe 在 close 中的 unmap 顺序，使得 fd 引用次数的 -1 先于 pipe。在两个 unmap 的间隙，pageref(pipe) > pageref(fd) 仍成立，即使此时发生中断，也不会影响判断管道是否关闭的正确性。根据上面的描述我们其实已经能够得出一个结论：控制 fd 与 pipe 的 map/unmap 的顺序可以解决进程竞争导致非同步写入的问题。我们通过控制修改 pp\_ref 的前后顺序避免了非同步写入的问题，但是我们还得解决第二个问题：读取 pp\_ref 的同步问题。同样是上面之前给出的代码，我们思考下面的情景：

* 假设 fork 结束后，子进程先执行。执行完 close(p[1]) 后，执行 read，要从 p[0] 读取数据。但由于此时管道数据缓冲区为空，所以 read 函数要判断父进程中的写端是否关闭，进入到 \_pipe\_is\_closed 函数，pageref(fd) 值为 2(父进程和子进程都打开了 p[0])，此时时钟中断产生。
* 内核切换到父进程执行，父进程 close(p[0])，之后向管道缓冲区写数据。要写的数据较多，假设写到一半时钟中断产生，进程调度后切换到子进程运行。
* 子进程继续运行，获取到 pageref(pipe) 值为 2(父进程打开了 p[1], 子进程打开了p[0])，引用值相等，于是认为父进程的写端已经关闭，子进程退出。

fd 是一个父子进程共享的变量，但子进程中的 pageref(fd) 没有随父进程对 fd 的修改而同步，这就造成了子进程读到的 pageref(fd) 成为了“脏数据”。为了保证读的同步性，子进程应当重新读取 pageref(fd) 和 pageref(pipe)，并且要在确认两次读取之间进程没有切换后，才能返回正确的结果。为了实现这一点，我们要使用到之前一直都没用到的变量：env\_runs。env\_runs 记录了一个进程 env\_run 的次数，这样我们就可以根据某个操作 do() 前后进程 env\_runs 值是否相等，来判断在 do() 中进程是否发生了切换。根据上面的表述，修改 \_pipe\_is\_closed 函数，使得它满足“同步读”的要求。注意 env\_runs 变量是需要维护的。

### 7.2.4 相关函数

下面我们介绍 6 个与管道相关的函数。**创建管道函数**user/lib/pipe.c 中的 int pipe(int pfd[2])。这个函数的作用是创建一个管道。大致可以分为三步：

* 首先，创建并分配两个文件描述符 fd0、fd1，并为这两个文件描述符自身分配相应的空间。
* 然后给 fd0 对应的数据区域分配一页空间，并将 fd1 对应的数据区域映射到相同的物理页，这一页的内容为 pipe 结构体。
* 最后，将作为读端的 fd0 的权限设置为只读，作为写端的 fd1 的权限设置为只写，并通过函数的传入参数将这两个文件描述符的编号返回。

值得注意的是，目前我们一共分配了三个页面，权限位均需要包含 PTE\_LIBRARY，这是因为这些页面的数据对于父子进程是共享的，修改页面内容时不触发写时复制，这样才能顺利的在父子进程之间传递信息。**两个查询管道是否关闭的函数**user/lib/pipe.c 中的 static int \_pipe\_is\_closed(struct Fd \*fd, struct Pipe \*p) 与 int pipe\_is\_closed(int fdnum)。这两个函数的作用是判断管道是否已经被关闭，其中，函数的主要逻辑存在于 \_pipe\_is\_closed 中，pipe\_is\_closed 是对于 \_pipe\_is\_closed 的重新封装。通过创建管道时，分配的页面以及映射关系可知，文件描述符所在页面 rfd、wfd 被读端写端各自映射一次，管道页面 pipe 被读端写端同时映射，被映射两次，因此则有如下等式

pageref(rfd) + pageref(wfd) = pageref(pipe)

当管道某一端被关闭后，这一端映射的页面将被解除，假设当前调用此函数的进程为读进程，写端关闭，则 pageref(wfd) = 0，当前进程为写进程时类似，因此，当管道另一端关闭时，以下等式成立：

pageref(fd) = pageref(pipe)

需要注意的是，在 \_pipe\_is\_closed 中，我们需要利用两次 pageref 函数分别获取 fd 和 pipe 对应页面的引用数，由于这两次操作不是原子的，之间可能被时间片调度打断，导致 pageref(fd) 和 pageref(pipe) 非同步获取，先获取的数据失效。为了解决这个问题，我们需要确保两次获取 pageref 时进程没有切换，也就是两次获取 pageref 前后的 env->env\_run 变量没有变化时，再根据等式判断管道另一端是否被关闭。若等式成立，则说明管道另一端已经关闭，返回 1,否则返回 0。**读管道函数**user/lib/pipe.c 中的 static int pipe\_read(struct Fd \*fd, void \*vbuf, u\_int n,u\_int offset)。这个函数的作用是从 fd 对应的管道数据缓冲区中，读取至多 n 字节到 vbuf 对应的虚拟地址中，并返回本次读到的字节数。读取的过程中会遇到两类可能的情况：

* 缓冲区不为空：则按顺序读取缓冲区中的内容，直到缓冲区为空或达到读取上限，并返回读到的字节数。
* 缓冲区为空：则使用 \_pipe\_is\_closed 函数查询管道的写端是否已经关闭，若已经关闭，则说明读入完成，函数返回0；若没有关闭，则使用 syscall\_yield() 进行等待，直到管道关闭或缓冲区不为空。

**写管道函数**user/lib/pipe.c 中的 static int pipe\_write(struct Fd \*fd, const void \*vbuf, u\_intn, u\_int offset)。与读管道函数类似，这个函数的作用是从 vbuf 对应的虚拟地址，向 fd 对应的管道数据缓冲区中写入 n 字节，并返回本次写入的字节数。此处的参数 n 与读取时有所不同，表示需要写入恰好 n 个字节，而非缓冲区的长度上限。因此，在完成全部 n 个字节的写入之前，若管道缓冲区已满，则需要使用 syscall\_yield() 进行等待，而非提前返回，直到缓冲区不满或管道关闭。**关闭管道函数**user/lib/pipe.c 中的 static int pipe\_close(struct Fd \*fd)。这个函数的作用是关闭管道的一端。将待关闭的一端对应的文件描述符传入，并 unmap 文件描述符自身的页面以及 unmap 文件描述符对应的数据页面—pipe结构体，完成关闭操作。

## 7.3 shell

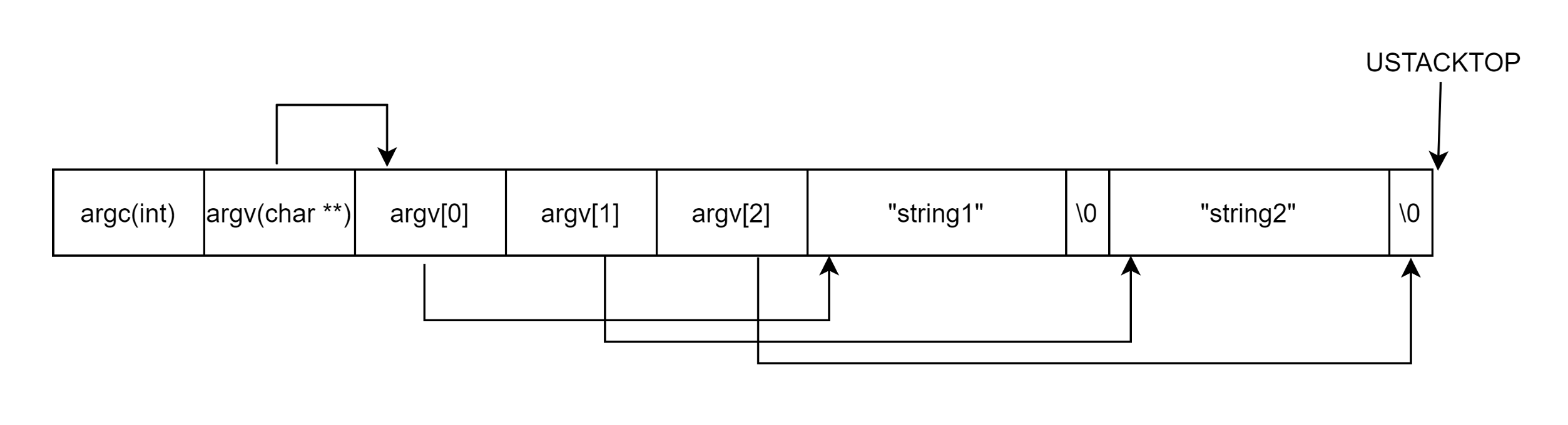
在计算机科学中，shell 是指“为使用者提供操作界面”的软件（命令解析器）。它接收用户命令，然后调用相应的应用程序。基本上 shell 分两大类：一是图形界面 shell（Graphical User Interface shell 即 GUI shell）。例如：应用最为广泛的是微软 Windows 系列操作系统的 Windows Explorer，也包括广为人知的 Linux 操作系统的XWindow Manager（BlackBox 和 FluxBox），以及功能更强大的 CDE、GNOME、KDE 和 XFCE 等。二是命令行式 shell（Command Line Interface shell ，即 CLI shell），也就是我们 la64 操作系统最后即将实现的 shell 模式。

### 7.3.1 学习 spawn 函数

spawn 的作用是帮助我们调用文件系统中的可执行文件并执行。spawn 的流程可以分解如下：

* 从文件系统打开对应的文件（二进制 ELF，在我们的OS里是 \*.b）；
* 申请新的进程控制块；
* 将目标程序加载到子进程的地址空间中，并为它们分配物理页面；
* 为子进程初始化地址空间。对于栈空间，由于 spawn 需要将命令行参数传递给用户程序，所以要将参数也写入用户栈中；
* 设置子进程的寄存器（栈指针 sp 和用户程序入口 EPC）；
* 将父进程的共享页面映射到子进程的地址空间中；
* 这些都做完后，设置子进程可执行。

关于如何为子进程初始化栈空间，请仔细阅读 init\_stack 函数。因为我们无法直接操作子进程的栈空间，所以该函数首先将需要准备的参数填充到本进程的 UTEMP 这个页面处，然后将 UTEMP 映射到子进程的栈空间中。首先将 argc 个字符串填到栈上，并且不要忘记在每个字符串的末尾要加上 '\0'表示结束，然后将 argc + 1 个指针填到栈上，第 argc + 1 个指针指的是一个空字符串表示参数的结束。最后将 argc 和 argv 填到栈上，argv 将指向那 argc + 1 个字符指针。这里给出一张 spawn 准备的栈空间的示意图。



### 7.3.2 解释 shell 命令

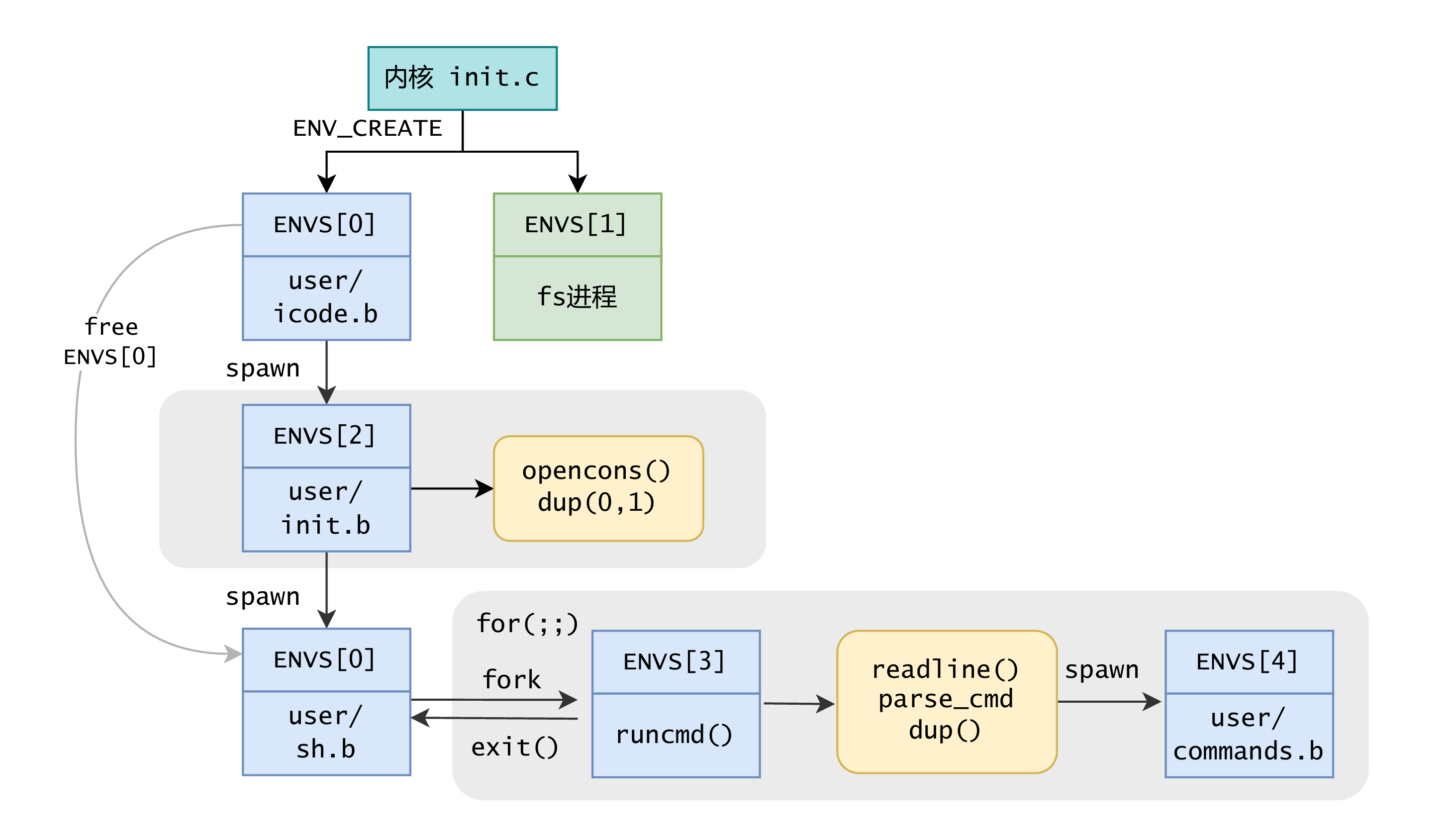
在 Lab5 中我们实现了文件系统，Lab6 中我们在 user 目录下提供了 ls.c、cat.c、echo.c等几个用户程序，模拟了 Linux 下的同名命令，而 Shell 程序 sh.c 调用了上面提到的 spawn 函数，其能够读取相应的可执行文件，并加载到新进程中运行。请阅读 fs/Makefile，思考我们是如何将这些用户程序编译得到的可执行文件“烧录”到 la64 的文件系统中的。接下来，我们需要在 shell 进程里实现对管道和重定向的解释功能。解释 shell 命令时：

1. 如果碰到重定向符号 ‘<’ 或者 ‘>’,则读下一个单词，打开这个单词所代表的文件，然后将其复制给标准输入或者标准输出。
2. 如果碰到管道符号 ‘|’,则首先需要建立管道 pipe，然后 fork。

* 对于父进程，需要将管道的写者复制给标准输出，然后关闭父进程的读者和写者，运行 ‘|’ 左边的命令，获得输出，然后等待子进程运行。
* 对于子进程，将管道的读者复制给标准输入，从管道中读取数据，然后关闭子进程的读者和写者，继续读下一个单词。

在这里可以举一个使用管道符号的例子来方便大家理解，我们学习过的 Linux 中的管道命令 cat my.sh | grep "Hello" 这就是使用管道的例子，cat my.sh 命令会将 my.sh 中的内容原文写入到管道，而 grep "Hello" 命令作为子进程执行，将管道中的内容读出并作为 grep 的输入，进行查找字符串操作并将查找结果输出到标准输出。在 spawn 函数中通过设置 PTE\_LIBRARY 权限位，将父进程所有的共享页面映射给了子进程。想一下，进程空间中的哪些内存是共享内存？在进程空间中，文件、管道、控制台以及文件描述符都是以共享页面的方式存在的。有几处通过 spawn 产生新进程的位置。

* 如下图所示，内核启动的进程 user/icode.b 调用了 spawn，创建了 init.b 进程。init.b 进程先打开控制台（console）作为 0 和 1 号文件描述符（fd），也就是进程的标准输入和输出，然后调用 spawn 创建了 sh.b 进程，也就是我们的 shell。通过共享页面（PTE\_LIBRARY）机制，fork 和 spawn 创建的子进程继承了父进程持有的 fd，所以 shell 进程仍能通过标准输入输出操作控制台，与用户进行交互。



* 子 shell 进程负责解析命令行命令，并通过 spawn 生成可执行程序进程（对应\*.b文件）。在解析命令行的命令时，子 shell 会将重定向的文件及管道等 dup 到子 shell 的标准输入或输出，然后 spawn 时将标准输入和输出通过共享内存映射给可执行程序，所以可执行程序可以从控制台、文件和管道等位置输入和输出数据。

### 7.3.3 相关函数

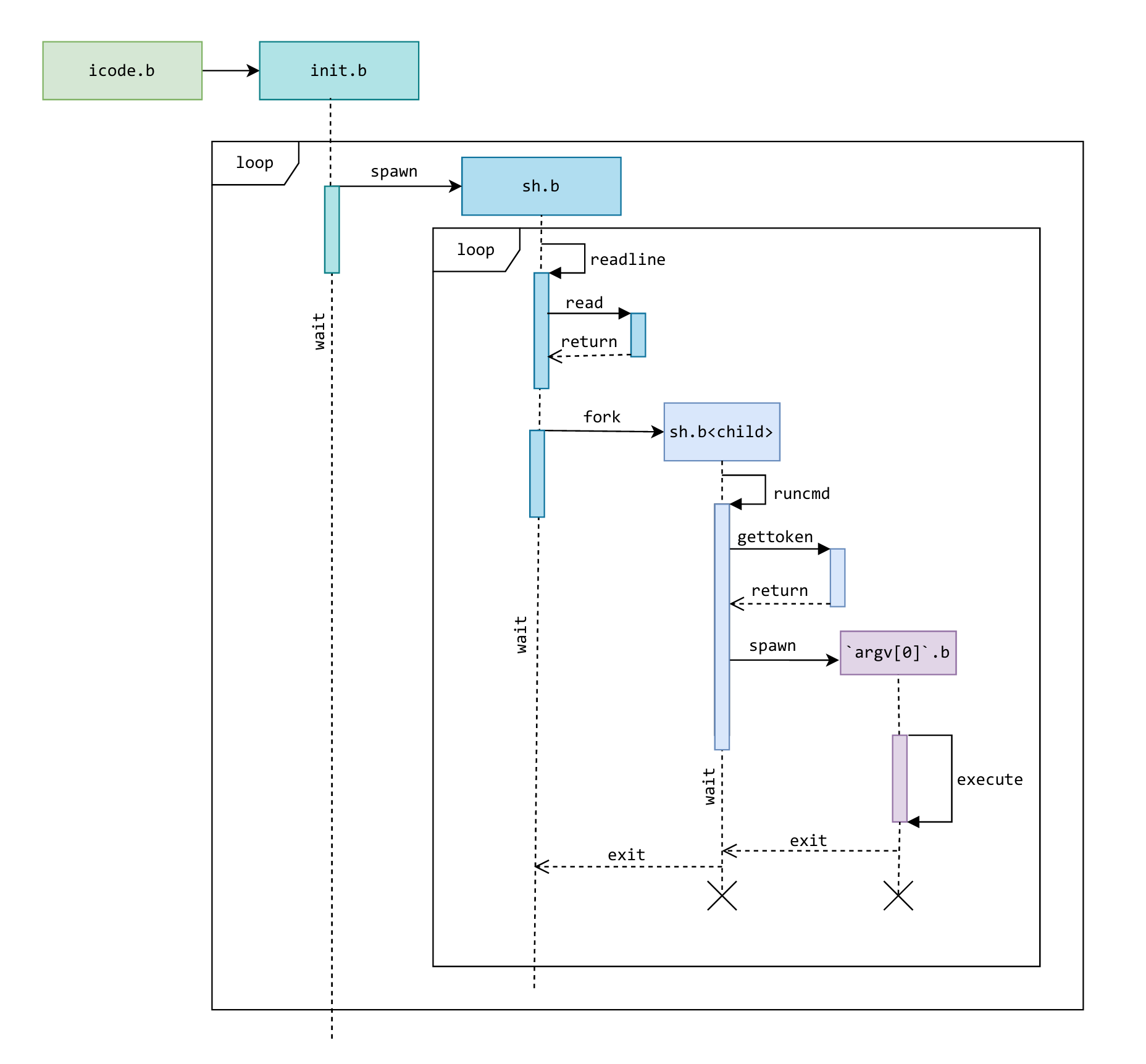
下面我们介绍 8 个与 shell 相关的函数。**初始化栈空间函数**user/lib/spawn.c 中的 int init\_stack(u\_int child, char \*\*argv, u\_int \*init\_esp)。这个函数的作用是初始化子进程的栈空间，达到向子进程的主函数传递参数的目的。由于父进程无法直接操作子进程的栈空间，因此需要将参数填充到当前进程的 TMPPAGE 页面处，将 TMPPAGE 映射到子进程的栈空间中。具体参数填充方式可以参考之前所给图。**spawn 函数**user/lib/spawn.c 中的 int spawn(char \*prog, char \*\*argv)。这个函数与 fork 函数类似，其最终效果都是产生一个子进程，不过与 fork 函数不同的是，spawn 函数产生的子进程不再执行与父进程相同的程序，而是装载新的 ELF 文件，执行新的程序。spawn 函数的大致流程如下：

1. 使用文件系统提供的 open 函数打开即将装载的 ELF 文件 prog。
2. 使用系统调用 syscall\_exofork 函数为子进程申请一个进程控制块。
3. 使用 init\_stack 函数为子进程初始化栈空间，将需要传递的参数 argv 传入子进程。
4. 使用 elf\_load\_seg 将 ELF 文件的各个段加载进子进程。
5. 设置子进程的运行现场寄存器，将 tf->era 设置为程序入口点，tf->regs[29] 设置为装载参数后的栈顶指针，从而在子进程被唤醒时以正确的状态开始运行。
6. 将父进程的共享页面映射给子进程，与 fork 不同的是，这里只映射共享页面。
7. 使用系统调用 syscall\_set\_env\_status 唤醒子进程。

其中，第1步、第3步和第4步是相比 fork 函数新增的部分，第5步和第6步与 fork 略有不同，第2步和第7步与 fork 中的步骤几乎一致。**sh.c 中的主函数**user/sh.c 中的 int main(int argc, char \*\*argv)。与 pipe.c 这样的用户库不同，sh.c 是一个完整的用户程序，也就是 shell，其主函数即为启动 shell 进程时第一步进入的函数。函数的主体是一个死循环，循环中大致流程如下：

* 调用 readline 读入用户输入的命令。
* fork 出一个子进程。
* 子进程执行用户的命令，执行结束后子进程结束。父进程在此等待子进程。
* 父进程等待子进程结束后，返回循环开始，读入用户的下一个命令。

**命令读入函数**user/sh.c 中的 void readline(char \*buf, u\_int n)。这个函数的作用是从标准输入（控制台），读入一行用户读入的命令，保存在 char\* buf 中。**命令解析函数**user/sh.c 中的 int \_gettoken(char \*s, char \*\*p1, char \*\*p2) 和 int gettoken(char\*s, char \*\*p1)。gettoken 函数接收 readline 读入的命令字符串作为传入参数 char\* s。这个两个函数的作用是将命令字符串分割，提取命令中基本单元——特殊符号或单词，并过滤空白字符。其中，gettoken 是对于 \_gettoken 的封装，调用 gettoken 时，每次调用返回上一次调用解析到的基本单元，并解析下一个基本单元。特殊符号包括重定向符号 < 和 >、管道符号 |、命令分隔符号 ; 和 & 。**cmd 命令解析与执行函数**user/sh.c 中的 void parsecmd(char \*\*argv, int \*rightpipe) 和 user/sh.c 中的 void runcmd(char \*s)。void parsecmd(char \*\*argv, int \*rightpipe) 这个函数的作用是解析用户输入的命令。通过调用 gettoken，将命令解析成单个词法单元，保存在 argv 中，并在遇到重定向、管道等特殊符号时做相应的特殊处理。当解析完一条命令包含的所有词法单元后，结束 parsecmd 函数，进入 runcmd 函数的命令执行阶段。特别地，当命令中包含管道操作时，则需要创建管道并 fork 出一个子 shell。子进程调用 parsecmd 函数继续解析管道右侧的命令，父进程返回 runcmd 函数执行管道左侧命令，此时 rightpipe 记录子进程的进程号也就是解析并执行管道右侧命令的进程号，用于管道左侧命令进程等待管道右侧命令进程执行完毕后再释放。void runcmd(char \*s) 这个函数的作用是执行解析出的每一条命令。通过 argv 判断命令的种类，调用 spawn 产生子进程并装载命令对应的 ELF 文件，子进程执行命令，父进程在此处等待子进程结束后，结束进程。随后，上文提到的在 main 循环结尾处等待的进程，也就是此处父进程的父进程，结束等待，返回循环开始，读入新一行命令。具体进程间的层次关系可以参照下图。**shell 相关函数调用关系**



## 7.4 实验正确结果

本节将通过一些测试用例评测管道和 Shell 的正确性，同时给出了测试结果的截图，同学们可以自行核对。

### 7.4.1 管道测试

管道测试对应的测试点为 tests/lab6\_1，测试中有三个文件，分别是 user/testpipe.c、user/testpiperace.c 和 user/testelibrary.c，以合适的次序建好进程后，在 testpipe 的测试中若出现两次 pipe tests passed 即说明测试通过。testpipe 本地测试部分运行结果如下。

[00001802] pipecreate  
.................................  
pipe write closed properly  
pipe tests passed  
[00004005] destroying 00004005  
[00004005] free env 00004005  
i am killed ...  
pipe tests passed  
[00001802] destroying 00001802  
[00001802] free env 00001802  
i am killed ...

在 testpiperace 中，子进程多次查询管道是否关闭。而父进程不停地执行 dup 函数，dup 操作主要包含两个操作，分别是关闭映射目标原有的管道（close ）和将旧文件映射到新文件（包括文件描述符和文件内容）。如果这两个操作中有任意一个操作产生竞争，将可能导致子进程认为写入端关闭。在 testpiperace 的测试中应当出现 race didn’t happen 是正确的。testpiperace 本地测试部分运行结果如下。

race didn't happen  
[00002003] destroying 00002003  
[00002003] free env 00002003  
i am killed ...

在 user/testptelibrary.c 的测试中，如果 fork 和 spawn 对于共享页面的处理正确，测试才能通过。

### 7.4.2 shell 测试

在 init/init.c 中按照如下顺序依次启动 shell 和文件服务：

ENV\_CREATE(user\_icode);  
 ENV\_CREATE(fs\_serv);

如果正常会看到如下效果。

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::  
:: ::  
:: MOS Shell 2023 ::   
:: ::  
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

使用不同的命令会有不同的效果：

* 执行 ls.b，会显示一些文件和目录名。

$ ls.b  
testarg.b cat.b pingpong.b testbss.b newmotd testpiperace.b testpipe.b  
motd init.b num.b testfdsharing.b ls.b echo.b sh.b halt.b testptelibrary.b

* 执行 cat.b 并输入任意字符，会有回显现象出现，按下 Ctrl-D 可结束输入。
* 执行 cat.b motd，会显示文件 motd 中的内容。

$ cat.b  
aabbcc

$ cat.b motd  
This is /motd, the message of the day.  
  
Welcome to the MOS kernel, now with a file system!

* 执行 ls.b | cat.b，和 ls.b 的现象应当一致。

$ ls.b | cat.b  
testarg.b cat.b pingpong.b testbss.b newmotd testpiperace.b testpipe.b  
motd init.b num.b testfdsharing.b ls.b echo.b sh.b halt.b testptelibrary.b