

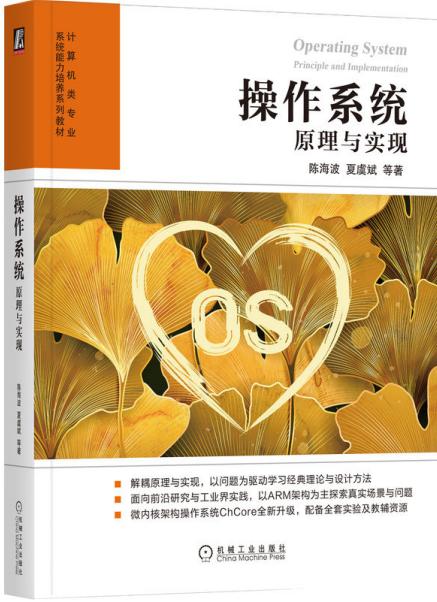
IPADS OS Course Lab Manual

 GitHub Pages passing license MulanPSL-2.0 release v25.03.1 issues 0 open

实验 文档 代码 解析 CREATIVE COMMONS NC-SA

本仓库包含上海交通大学IPADS实验室设计的操作系统课程系列实验。每个实验位于独立的目录，附有详细的[实验说明](#)和[源码解析](#)。

课程教材：



完成系列实验后，你可以在树莓派上用自己DIY的ChCore内核，运行宝可梦游戏、调用DeepSeek、本地运行Qwen-1.5b等等。

[GBA](#)

[Note](#)

如果你有任何建议或更正意见，欢迎提交 Pull Requests 或 Issues。让我们一起合作改进实验

Lab0: 拆炸弹 (ARM 汇编)

该实验受到CSAPP课程启发，CSAPP课程设计了一个针对x86/x86-64汇编的拆炸弹实验。不同之处在于，本实验目标是熟悉ARM汇编语言，并为后续的ARM/树莓派内核实验做好准备。

Tutorial: [https://www.bilibili.com/video/BV1q94y1a7BF/?](https://www.bilibili.com/video/BV1q94y1a7BF/)

vd_source=63231f40c83c4d292b2a881fda478960

Lab1: 内核启动

该实验的主要内容是关于 何在内核启动过程中设置CPU异常级别、配置内核页表并启用 MMU。在内核实验系列中，我们将使用 [ChCore 微内核](#) 的基础版本，并使用 Raspi3b+作为实验平台（无论是使用QEMU树莓派模拟器还是树莓派开发板都可以）。

Tutorial: https://www.bilibili.com/video/BV1gj411i7dh/?spm_id_from=333.337.search-card.all.click

Lab2: 内存管理

该实验主要内容是关于内核中的伙伴系统和slab分配器的实现，并为应用程序设置页表。

Tutorial: https://www.bilibili.com/video/BV1284y1Q7Jc/?vd_source=316867e8ad2c56f50fa94e8122dd7d38

Lab3: 进程与线程

该实验主要内容包括创建第一个用户态进程和线程，完善异常处理流程和系统调用，编写一个 Hello-World在实验内核上运行。

Tutorial: <https://www.bilibili.com/video/BV11N411j7bR/>

Lab4: 多核调度与IPC

该实验中可以看到多核是 何启动的、多线程 何调度、基于capability权限管控的进程间通信机制。

Tutorial: <https://www.bilibili.com/video/BV1AS421N7rU/>

Lab5: 虚拟文件系统

该实验关注虚拟文件系统（Virtual File System, VFS），VFS抽象层使得不同类型的文件系统可

以在应用程序层面以统一的方式进行访问。

Lab6: GUI (Optional)

该实验将详细介绍ChCore上基于Wayland的GUI系统的运行原理，包括Wayland通信协议和Wayland Compositor，并且要求读者在了解基于Wayland的GUI系统运行原理的基础上，基于ChCore的GUI框架编写自己的具有GUI界面的APP。

Last change: 2025-02-26, commit: [a7922cc](#)

如何开始实验

- 环境准备
 - 使用Dev-Container (推荐)
- 文档说明
- CI评分

环境准备

对于所有的操作系统，本实验必须依赖Docker环境且被期望用户只在Linux系统上进行运行(我们不主动维护Mac OS系统，如果有问题请主动发送issue)，请按照Docker官方指示为你运行的操作系统安装对应的Docker发行版。

关于docker

由于中国大陆地区的网络限制，请确保你的docker能够连接到DockerHub，测试方法可以使用 `docker pull nginx:latest`，如果无法访问，您可以依照[该文档](#)为你的docker daemon添加代理规则。如果你缺少代理，你可以使用这个以下的几个链接：[百度云\(提取uwuv\)](#) [交大云\(仅上海交通大学内部可以访问\)](#)。

下载压缩好的Docker镜像，镜像使用 `gzip` 解压缩镜像文件到标准输出流之后再由docker进行导入（下方的yy.mm请以当前的大版本号替换，版本tag为24.09.1则请将yy.mm替换成24.09）。

```
gzip -cd docker.ipads.oslab.yy.mm.tar.gz | docker load
```

关于虚拟机

如果你使用的是Windows/MacOS系统，不想手动安装docker以及下载镜像，我们也准备了基于VMWare 17的虚拟机镜像，你也可以使用上方的链接下载vmware虚拟机镜像，你可以在解压之后导入vmware即可使用。

用户名stu 密码为123456

Crlf

由于bash/GNU Make强制需要使用LF作为换行符，当你尝试保存脚本时请确保换行符为LF，而非CRLF。

Warning

小心使用sudo，请注意我们在编译的时候我们会同步的在编译文件夹时写入timestamp文

件，如果你使用root权限，会导致后续普通的正常构建返回错误。

使用Dev-Container (推荐)

果你使用的是带有支持Microsoft规范下[Dev-Container](#)插件的代码编辑器或者集成开发环境，亦或者你使用的是非Linux平台的开发环境，我们强烈建议你使用Dev-container直接进行开发，我们已经在其中已经预先安装好了你可能需要使用的所有工具链。并且针对vscode我们在每个Lab的分支目录下都已经配置好了合适的插件配置，简单安装即可以一键启用。安装完之后进入本实验的根目录，此时dev-container会识别到容器开发环境，重新进入后就可以直接使用了。

关于macos

MacOS 26后苹果使用了新的虚拟机引擎，其会影响qemu-user的兼容性，请按照以下的[指令](#)进行关闭。

关于windows

在MacOS平台，使用dev-container能够保证兼容性。但在Windows上面由于文件系统并非POSIX兼容，你需要执行以下的方式来正确拉取repo.

1. 请在设置确保打开Windows 10/11 的开发者模式
2. 运行git config --global core.autocrlf false
3. 使用git clone -c core.symlinks true 进行clone
4. 在完成lab3-5时请将 user/chcore-libc/libchcore/cmake/do_override_dir.sh 中所有的 ln -sf 替换成 cp -r
5. 对于所有 libchcore 下的所有文件，请都运行 make clean 确保其生效

果你是新用户，请使用以下命令进行拉取。

```
git clone -c core.symlinks=true https://github.com/SJTU-IPADS/OS-Course-Lab.git
```

文档说明

各实验文档除开 lab0 为单独的实验内容，其他都包含了以下的几种习题，请根据下方的指示完成对应实验报告以及对应的编程题。

思考题

思考题为需要在实验报告中书面回答的问题，需要用文字或者是图片来描述

练习题

练习题需在 ChCore 代码中填空，并在实验报告阐述实现过程，完成即可获得实验大多数的分数。

挑战题

挑战题为难度稍高的练习题，作为实验的附加题用于加深你对代码结构以及系统设计的理解。

CI评分

在新实验中，我们特意准备了支持github actions以及gitlab ci的CI配置，你可以在你所使用的代码托管平台上自动运行脚本，为了确保你不会篡改预编译的 .obj 文件，我们会根据本仓库的主线的各个Lab中的 `filelist.mk` (详见贡献指南)的定义自动提取提交中所需要的文件与当前仓库主线的文件树进行合并，最终进行评分。

Last change: 2025-09-27, commit: [a8321b7](#)

贡献指南

- [代码规范](#)
- [工具链使用](#)
- [何提交新实验](#)

代码规范

Commit

我们参照Conventional Commit构建了Pull Request的Blocker，并且关闭了对主分支的直接Push，请确保你的Commit符合Conventional Commit规范

Github

我们使用Github Issues跟踪所有的问题，如果你在实验过程中产生了任何预期以外的错误，欢迎提交Issues。

Mdbook

我们使用基于MdBook构建Markdown文档体系，你可以参照 `.github/github-pages.yaml` 中的下载指示，将所有的Mdbook及其所需要的所有**预处理器**，都安装到你的系统环境路径中。如果你对文档方面有任何的更正，你可以在 `Pages/SUMMARY.md` 中找到实验手册的文档结构以及对应的所有文件。更改后在仓库根目录你可以运行 `mdbook-mermaid install`，然后运行 `mdbook serve`，并访问 `localhost:3000` 查看编译后的文档。我们也使用 `markdownlint` 对所有文档开启的CI检查，请确保提交后能够通过CI。

工具链使用

由于工具链版本问题，可能会导致在不同版本工具链编译的情况下导致在不同Release版本所链接的系统镜像无法正常工作的情况，请确保开发过程中使用与 `.devcontainer/Dockerfile` 即 `ubuntu 20.04.6` 的相同版本的交叉编译链进行预编译源文件，本仓库对所有Lab的正确答案的构建同样也开启了CI检查，如果发现在不同Release版本下无法通过，请检查你所使用的工具链是否符合预期。

Important

对于所有源代码的预编译，请一定准备两份，并且对调试符号段的所有信息都使用

`aarch64-gnu-linux-strip` 进行删除。

如何提交新实验

对于新实验，我们使用两个文件进行定义实验规范，即 `filelist.mk` 以及 `scores.json`，其中 `scores.json` 用于定义给分点以及对应的分数，`filelist.mk` 则是提交文件列表，用于定义该Lab的提交文件，运行 `make submit` 后 `make` 会读取 `filelist.mk` 的文件定义，并按照一致的相对路径进行打包。文件定义样例在Lab1中可以查看。

Last change: 2024-09-06, commit: [668593f](#)

v25.03.1 更新内容

- 使用pexpect替换capturer.py
- 文档更新以及typo fix
- 删除gendedeps.sh
- 容器与虚拟机镜像更新
- 更新代码讲解

Last change: 2025-02-25, commit: [339eafd](#)

Lab0：拆炸弹

- 简介
- Makefile 讲解
- 评分与提交规则

简介

在实验 0 中，你需要通过阅读汇编代码以及使用调试工具来拆除一个二进制炸弹程序。本实验分为两个部分：第一部分介绍拆弹实验的基本知识，包括 ARM 汇编语言、QEMU 模拟器、GDB 调试器的使用；第二部分需要分析炸弹程序，推测正确的输入来使得炸弹程序能正常退出。

Warning

在完成本实验之前，请务必将你的学号填写在 `student-number.txt` 当中，否则本lab实验的成绩将计为0分

Makefile 讲解

- `make bomb`：使用`student-number.txt`提供的学号，生成炸弹，如果您不是上海交通大学的学生可以自行随意填写。
- `make qemu`：使用`qemu-aarch64`二进制模拟运行炸弹
- `make qemu-gdb`：使用`qemu-aarch64`提供的`gdb server`进行调试
- `make gdb`：使用`gdb`连接到`qemu-aarch64`的`gdb-server`进行调试

评分与提交规则

本实验你只需要提交 `ans.txt` 以及 `student-number.txt` 即可

Important

运行 `make grade` 来得到本实验的分数

运行 `make submit` 会在检查`student-number.txt`内容之后打包必要的提交文件

Last change: 2025-02-25, commit: [2d7a002](#)

基础知识

Info

本部分旨在熟悉 ARM 汇编语言，以及使用 QEMU 和 QEMU/GDB 调试

- 熟悉 AArch64 汇编
- 使用 QEMU 运行炸弹程序
- QEMU 与 GDB

熟悉 AArch64 汇编

AArch64 是 ARMv8 ISA 的 64 位执行状态。《ARM 指令集参考指南》是一个帮助入门 ARM 语法的手册。在 ChCore 实验中，只需要在提示下可以理解一些关键汇编和编写简单的汇编代码即可。你可以在 [Arm 的网站](#) 上搜索具体的指令，常备快速参考手册也有帮助，比 [这里](#)。

Tip

果你完全没接触过 ARM，这些提示可以帮助你更顺利进入实验：

- x0-x31 是 64 位通用寄存器
- x0-x7 用作传参，x0 还用作返回值
- x31 (sp) 是栈指针
- x30 (lr) 是返回地址
- x29 (fp) 是栈帧指针
- w0-w31 是 x0-x31 对应的 32 位寄存器
- [Xn] 和 [Xn, #imm] 是两种常用的取址模式，寄存器内的值解释为地址，加上可选的常量偏移

使用 QEMU 运行炸弹程序

我们在实验中提供了 bomb 二进制文件，但该文件只能运行在基于 AArch64 的 Linux 中。通过 QEMU，我们可以在其他架构上模拟运行。同时，QEMU 可以结合 GDB 进行调试（打印输出、单步调试等）

Tip

QEMU 不仅可以模拟运行用户态程序，也可以模拟运行在内核态的操作系统。在前一种模

式下，QEMU 会模拟运行用户态的汇编代码，同时将系统调用等翻译为对宿主机的调用。在后一种模式下，QEMU 将在虚拟硬件上模拟一整套计算机启动的过程。

在lab0目录下，输入以下命令可以在 QEMU 中运行炸弹程序

```
[user@localhost Lab0]$ make qemu
```

炸弹程序的标准输出将会显示在 QEMU 中：

```
Type in your defuse password:
```

QEMU 与 GDB

在实验中，由于需要在 x86-64 平台上使用 GDB 来调试 AArch64 代码，因此使用gdb-multiarch代替了普通的gdb。使用 GDB 调试的原理是，QEMU 可以启动一个 GDB 远程目标 (remote target)（使用-s或-S参数启动），QEMU 会在真正执行镜像中的指令前等待 GDB 客户端的连接。开启远程目标之后，可以开启 GDB 进行调试，它会在某个端口上进行监听。

打开两个终端，在bomb-lab目录下，输入make qemu-gdb和make gdb命令可以分别打开带有 GDB 调试的 QEMU 以及 GDB，在 GDB 中将会看到 下的输出：

```
...
0x0000000000400540 in ?? ()
...
(gdb)
```

Last change: 2025-02-14, commit: [773916e](#)

二进制炸弹拆除

我们在实验中提供了一个二进制炸弹程序bomb以及它的部分源码bomb.c。在 bomb.c 中，你可以看到一共有 6 个 phase。对每个 phase，bomb 程序将从标准输入中读取一行用户输入作为这一阶段的拆弹密码。若这一密码错误，炸弹程序将异常退出。你的任务是通过 GDB 以及阅读汇编代码，判断怎样的输入可以使得炸弹程序正常通过每个 phase。以下是一次失败尝试的例子：

```
[user@localhost lab0] $ make qemu  
qemu -aarch64 bomb  
Type in your defuse password:  
1234  
BOOM !!!
```

Tip

你需要学习gdb、objdump的使用来查看炸弹程序对应的汇编，并通过断点等方法来查看炸弹运行时的状态（寄存器、内存的值等）。以下是使用gdb来查看炸弹运行状态的例子。在这个例子中，我们在main函数的开头打了一个断点，通过continue让程序运行直至遇到我们设置的断点，使用info查看了寄存器中的值，最终通过x查看了x0寄存器中的地址指向的字符串的内容。以下是输入与输出。

```
add symbol table from file "bomb"
(y or n) y
Reading symbols from bomb ...
(gdb) break main
Breakpoint 1 at 0x4006a4
(gdb) continue
Continuing.
Breakpoint 1, 0x00000000004006a4 in main ()
(gdb) disassemble
Dump of assembler code for function main:
0x0000000000400694 <+0>:stp0x0000000000400698 <+4>:mov
x29 , x30 , [sp , # -16]!
x29 , sp
0x000000000040069c <+8>:adpx0 , 0x464000 <free_mem +64>
0x00000000004006a0 <+12>:addx0 , x0 , #0x778
=> 0x00000000004006a4 <+16>:bl0x413b20 <puts >
0x00000000004006a8 <+20>:bl0x400b10 <read_line >
0x00000000004006ac <+24>:bl0x400734 <phase_0 >
0x00000000004006b0 <+28>:bl0x400708 <phase_defused >
0x00000000004006b4 <+32>:bl0x400b10 <read_line >
0x00000000004006b8 <+36>:bl0x400760 <phase_1 >
0x00000000004006bc <+40>:bl0x400708 <phase_defused >
0x00000000004006c0 <+44>:bl0x400b10 <read_line >
0x00000000004006c4 <+48>:bl0x400788 <phase_2 >
0x00000000004006c8 <+52>:bl0x400708 <phase_defused >
0x00000000004006cc <+56>:bl0x400b10 <read_line >
0x00000000004006d0 <+60>:bl0x400800 <phase_3 >
0x00000000004006d4 <+64>:bl0x400708 <phase_defused >
0x00000000004006d8 <+68>:bl0x400b10 <read_line >
0x00000000004006dc <+72>:bl0x4009e4 <phase_4 >
0x00000000004006e0 <+76>:bl0x400708 <phase_defused >
0x00000000004006e4 <+80>:bl0x400b10 <read_line >
0x00000000004006e8 <+84>:bl0x400ac0 <phase_5 >
0x00000000004006ec <+88>:bl0x400708 <phase_defused >
0x00000000004006f0 <+92>:adpx0 , 0x464000 <free_mem +64>
0x00000000004006f4 <+96>:addx0 , x0 , #0x798
0x00000000004006f8 <+100>:bl0x413b20 <puts >
0x00000000004006fc <+104>:movw0 , #0x0
0x0000000000400700 <+108>:ldpx29 , x30 , [sp] , #16
0x0000000000400704 <+112>:ret
// #0
End of assembler dump.
(gdb) info registers x0
x0
0x464778
4605816
(gdb) x /s 0x464778
0x464778:
"Type in your defuse password!"
```

在破解后续阶段时，为了避免每次都输入先前阶段的拆弹密码，你可以通过重定向的方式来让炸弹程序读取文件中的密码：

```
[user@localhost lab0] $ make qemu < ans.txt
qemu -aarch64 bomb
Type in your defuse password:
5 phases to go
4 phases to go
3 phases to go
2 phases to go
1 phases to go
0 phases to go
Congrats! You have defused all phases!
```

Last change: 2024-09-06, commit: [76ff8c2](#)

Lab1: 机器启动

简介

本实验作为 ChCore 操作系统课程实验的第一个实验，分为三个部分。

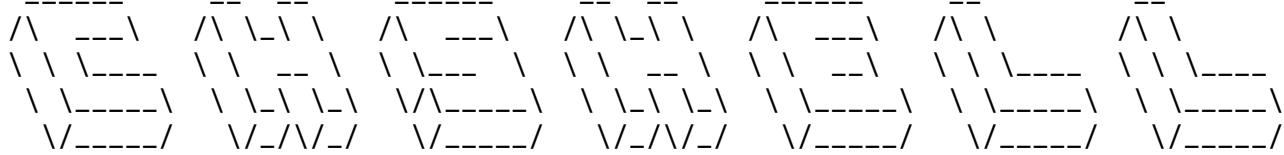
1. [RTFSC](#): 代码导读，由于是Lab1，我们主要注重于Chcore的构建系统，这部分没有习题。
2. [机器启动](#): 介绍aarch64结构启动时的关键寄存器以及关键的启动函数。
3. [页表配置](#): 介绍aarch64页表结构，以及针对树莓派3平台的内存布局编写页表配置。

调试指北

在开始实验之前，请务必读完[调试指北](#)，以帮助你快速上手调试。

本实验你可以在QEMU模拟器上完成实验，也可以在树莓派开发板上完成。本实验代码包含了基础的ChCore 微内核操作系统，除了练习题相关的源码以外，其余部分通过预先编译的二进制格式提供。

完成本实验的练习题之后，你可以进入 ChCore shell，运行命令或执行程序。例如，在 shell 中输入 `hello_world.bin` 运行一个简单的用户态程序；输入 `ls` 查看目录内容。



Welcome to ChCore shell!

\$

Last change: 2024-09-07, commit: [75616cc](#)

RTFSC(1)

Important

RTFSC = Read the FRIENDLY Source Code

Lab1的代码很多，在第一部分的代码架构解析的部分我们主要来讲解内核镜像是 何构建产生，以及评分基础设施是 何工作的。

- 构建系统
 - Makefile
 - CMake
 - 镜像定义生成
 - 定义多态设计
 - 镜像编译
- QEMU
- 评分系统

构建系统

Makefile

Makefile

果你对Makefile的语法有疑问的话，你可以参考这个网站的教程熟悉Makefile的写法。[1](#)

原有的Chcore的构建系统仅围绕着 `Scripts/chbuild` 这个脚本进行构建，但是由于 os Course Lab 需要增加评分的设施，我们为此在 `chbuild` 之外添加了Makefile的基础结构，以下为 Lab1/Makefile 的内容

```
LAB := 1
include $(CURDIR)/../Scripts/lab.mk
```

注意到我们仅仅我们定义了Lab的标识符，然后使用 `include` 将上层 `Scripts/lab.mk` 导入到当前的 `Makefile` 中。

```
# Note that this file should be included directly in every Makefile inside
# each lab's folder.
# This sets up the environment variable for lab's Makefile.

ifndef LABROOT
LABROOT := $(CURDIR)/..
endif

SCRIPTS := $(LABROOT)/Scripts

ifeq (,$(LAB))
$(error LAB is not set!)
endif

LABDIR := $(LABROOT)/Lab$(LAB)
SCRIPTS := $(LABROOT)/Scripts
GRADER ?= $(SCRIPTS)/grader.sh

# Toolchain Configuration
ifeq ($($shell command -v gdb-multiarch 2> /dev/null),)
# Default to gdb if gdb-multiarch is not available
# This is only the case on debian-based distros
    GDB := gdb
else
    GDB := gdb-multiarch
endif

DOCKER ?= docker
DOCKER_IMAGE ?= ipads/oslab:25.03
ifeq (,$(wildcard /docker.env))
DOCKER_RUN ?=
else
DOCKER_RUN ?= $(DOCKER) run -it --rm \
    -e SCRIPTS=$(SCRIPTS) \
    -e LABROOT=$(LABROOT) \
    -e LABDIR=$(LABDIR) \
    -e TIMEOUT=$(TIMEOUT) \
    -e LAB=$(LAB) \
    -u $($shell id -u $(USER)):$($shell id -g $(USER)) \
    -v $(LABROOT):$(LABROOT) -w $(CURDIR) \
    --security-opt=seccomp:unconfined \
    --platform=linux/amd64 \
    $(DOCKER_IMAGE)
endif
QEMU-SYS ?= qemu-system-aarch64
QEMU-USER ?= qemu-aarch64

# Timeout for grading
TIMEOUT ?= 10

ifeq ($($shell test $(LAB) -eq 0; echo $$?),1)
QEMU := $(QEMU-SYS)
ifeq ($($shell test $(LAB) -gt 4; echo $$?),0)
    include $(LABROOT)/Scripts/extras/lab$(LAB).mk
else
```

```
    include $(LABROOT)/Scripts/kernel.mk
endif
include $(LABROOT)/Scripts/submit.mk
else
QEMU := $(QEMU-USER)
endif
```

而 `lab.mk` 主要针对Lab环境进行检查，同时定义一些关键的变量，最终根据当前的 `$(LAB)` 的序号，再去导入不同的定义，在此处由于我们的 `$(LAB)` 变量为1，所以我们真正使用的Makefile定义为 `kernel.mk`

```
V ?= 0
Q := @
GRADER_V :=
ifeq ($(V), 1)
    Q :=
endif

ifeq ($(V), 2)
    Q :=
    GRADER_V := -v
endif

BUILDDIR := $(LABDIR)/build
KERNEL_IMG := $(BUILDDIR)/kernel.img
_QEMU := $(SCRIPTS)/qemu_wrapper.sh $(QEMU)
QEMU_GDB_PORT := 1234
QEMU_OPTS := -machine raspi3b -nographic -serial mon:stdio -m size=1G -kernel \
$(KERNEL_IMG)
CHBUILD := $(SCRIPTS)/chbuild
SERIAL := $(shell LC_ALL=C tr -dc A-Za-z0-9 </dev/urandom | head -c 13; echo)

export LABROOT LABDIR SCRIPTS LAB TIMEOUT

all: build

defconfig:
    $(Q)$(CHBUILD) defconfig

build:
    $(Q)test -f $(LABDIR)/.config || $(CHBUILD) defconfig
    $(Q)$(CHBUILD) build
    $(Q)find -L $(LABDIR) -path */compile_commands.json \
        ! -path $(LABDIR)/compile_commands.json -print \
    | $(SCRIPTS)/merge_compile_commands.py

clean:
    $(Q)$(CHBUILD) clean
    $(Q)find -L $(LABDIR) -path */compile_commands.json -exec rm {} \;

distclean:
    $(Q)$(CHBUILD) distclean

qemu: build
    $(Q)$(QEMU) $(QEMU_OPTS)

qemu-grade:
    $(SCRIPTS)/change_serial $(KERNEL_IMG) $(SERIAL)
    $(Q)$(QEMU) $(QEMU_OPTS)

qemu-gdb: build
    $(Q)echo "[QEMU] Waiting for GDB Connection"
    $(Q)$(QEMU) -S -gdb tcp::$(QEMU_GDB_PORT) $(QEMU_OPTS)

gdb:
    $(Q)$(GDB) --nx -x $(SCRIPTS)/gdb/gdbinit
```

```

grade:
    $(Q)$(MAKE) distclean &> /dev/null
    $(Q)(test -f $(LABDIR)/.config && cp $(LABDIR)/.config
$(LABDIR)/.config.bak) || :
    $(Q)$(MAKE) build
    $(Q)$(DOCKER_RUN) $(GRADER) -t $(TIMEOUT) -f $(LABDIR)/scores.json
$(GRADER_V) -s $(SERIAL) make SERIAL=$(SERIAL) qemu-grade
    $(Q)(test -f $(LABDIR)/.config.bak && cp $(LABDIR)/.config.bak
$(LABDIR)/.config && rm .config.bak) || :
.PHONY: qemu qemu-gdb gdb defconfig build clean distclean grade all

```

这里简述一下用法，其中我们定义了一个变量 v，当我们运行例 make v=1 时会将 Q 的定义重新设置，Q 的目的主要是为了做字符串的拼接。

Makefile怎么工作？

在Makefile中，其主要分为两种定义，全局定义以及规则定义。全局定义主要是定义变量以及Makefile宏或者是函数，规则定义则是根据变量或者字面字符串定义进行拼接，然后使用shell执行拼接后的命令，例 此处的Q在V不处于Verbose模式时就会被视作@，此时 Make 就不会打印下面的命令。

当我们在Lab1运行 make build，其就会转到 kernel.mk 的 build 这个规则下，此时 Make 会调用cmake完成进一步的构建

CMake

现代cmake教程

IPADS的前成员RC在Bilibili上分享过现代Cmake的pre，有兴趣的同学可以在这个链接进行观看²。

镜像定义生成

当我们运行 make build 之后，我们便进到了 chbuild 脚本中了，当开始时我们会使用 chbuild defconfig 这个bash函数调用cmake的其他脚本来生成镜像配置文件 .config，由于我们默认使用 raspi3 配置，我们会将 Scripts/defconfigs/raspi3.config 复制到当前Lab 的根目录下，这个是chcore的平台定义文件，之后则会调用 Scripts/build/cmake 添加到Lab 当中，之后则会运行 _config_default 这个函数主要负责递归读入Lab目录下的 config.cmake 文件并按照默认设置将平台无关的镜像配置文件，之后运行 _sync_config_with_cache 将镜像定义设置 .config 同步到CMakeCache中进行缓存，并返回到 Make 当中，之后 Make 继续运行 chbuild build ，按照 .config 定义进行构建镜像脚本。

```
defconfig() {
    if [ -d $cmake_build_dir ]; then
        _echo_err "There exists a build directory, please run
`$clean_command` first"
        exit 1
    fi

    if [ -z "$1" ]; then
        plat="raspi3"
    else
        plat="$1"
    fi

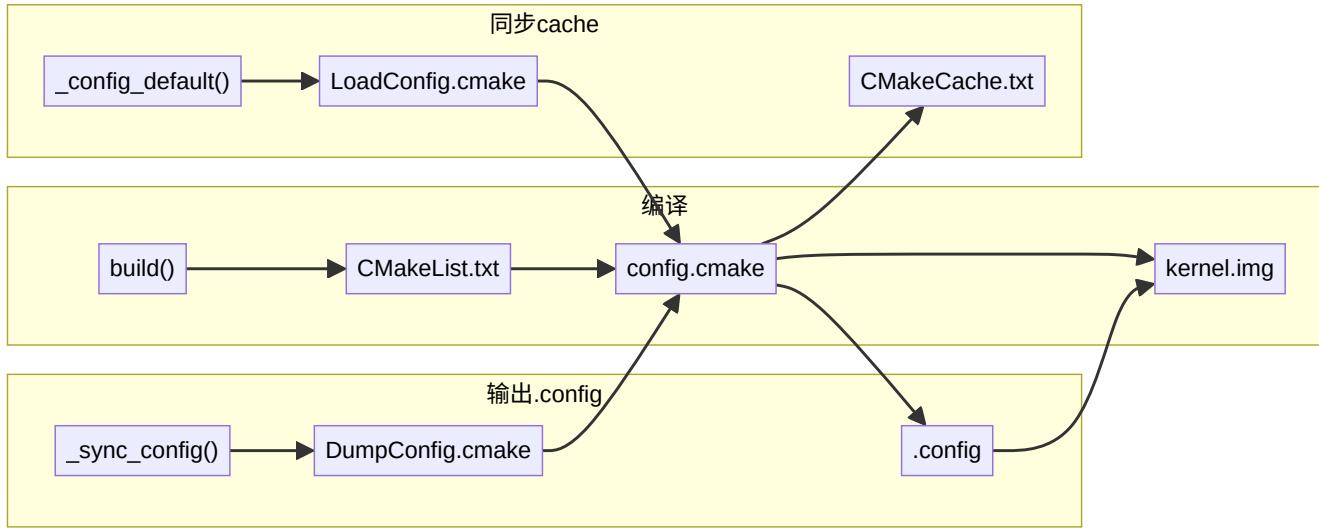
    _echo_info "Generating default config file for \`$plat\` platform..."
    cp $defconfig_dir/${plat}.config $config_file
    _config_default
    _sync_config_with_cache
    _echo_succ "Default config written to \`$config_file\` file."
}

_config_default() {
    _echo_info "Configuring CMake..."
    cmake -B $cmake_build_dir -C $cmake_init_cache_default
}

_sync_config_with_cache() {
    cmake -N -B $cmake_build_dir -C $cmake_init_cache_dump >/dev/null
}
```

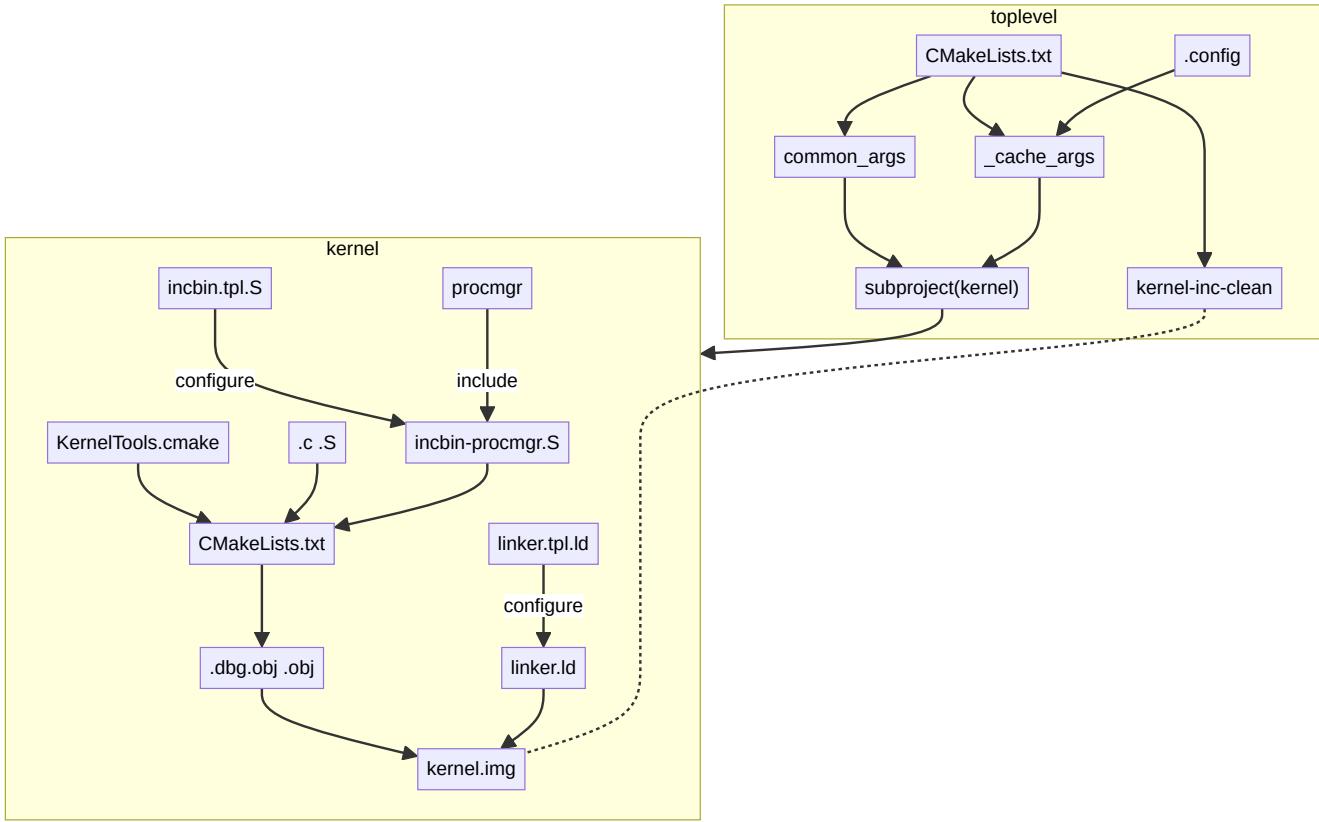
定义多态设计

Chcore通过 config.cmake 这个文件来定义规则的，但是我们单独去看的时候它使用了 chcore_config 这个宏，但是这个指令是不存在，实际上所有的 config.cmake 也是通过 include 指令来导入的，所以 chbuild 的每个指令都是去定义了 cmake 的 chcore_config 来执行不同的行为。大致的过程图 下， DumpConfig.cmake 主要是将 chcore_config 中的内容进行提取，并全部添加到 defconfig 生成的 .config 文件中，而 CMakeList.txt 构建时的 chcore_config 则是根据 .config 中的内容定向的配置子项目的编译选项。如果感兴趣你可以阅读 Scripts/build/cmake/ 下的 cmake 脚本文件。



镜像编译

Chcore的编译是从 `CMakeLists.txt` 的上层开始的，总的来说经过了 下的编译过程



首先上层的 `CMakeLists.txt` 会根据 `.config` 的内容构造 `_cache_args` 以及 `_common_args` 分别对应的是下层 CMake 的子项目的 CMake 构建参数以及变量参数，然后创建 `kernel-inc-clean` 用于删除 `kernel.img` 构建时的副产物，对应到最上层 `chbuild clean` 以及 `make clean` 时的清理选项，然后会递归进入 `kernel` 这个子项目。进入子项目后，CMake 首先会去导入 `KernelTools.cmake` 这个脚本去定义一些关键函数以及关键宏，同时会定义关键的工具链选项以及包含路径，最后再逐步地将每个子目录的 `CMakeLists.txt` 进行导入，对于源文件进行编译，对于预编译文件则是按照调试选项对应添加 `.dbg.obj` 或者是 `.obj` 文件进入文件列表，之后则是将 `user/procmgr` 这个文件利用 `incbin.tpl.s` 去生成对应的二进制汇编进行编译，最后

使用 `linker.tpl.ld` 所生成的 `linker.ld` 的 `linker script` 进行链接最后得到 `kernel.img` 的镜像。

Linker Script

果你对链接脚本感兴趣，你可以参考这个附录³.

QEMU

当 kernel 构建完成后，我们将使用 `qemu-system-aarch64` 进行模拟，当我们运行 `make qemu` 或者是 `make qemu-gdb` 时我们会进入 下面的规则，

```
Q :=  
GRADER_V := -v  
endif  
  
! -path $(LABDIR)/compile_commands.json -print \  
| $(SCRIPTS)/merge_compile_commands.py  
  
clean:  
$(Q)$(CHBUILD) clean  
$(Q)find -L $(LABDIR) -path */compile_commands.json -exec rm {} \  
;
```

此时 Make 会将 `QEMU_OPTS` 以及可能 `QEMU_GDB_PORT` 进行字符串的拼接，然后将参数传入 `qemu_wrapper.sh` 转到 `qemu` 程序中。

评分系统

我们使用 `make grade` 时会将 `TIMEOUT` 参数以及评分定义 `scores.json` 以及被评分的指令传入 `grader.sh`，

```
#!/usr/bin/env bash

if [[ -z $LABROOT ]]; then
    echo "Please set the LABROOT environment variable to the root directory
of your project. (Makefile)"
    exit 1
fi

SCRIPTS=${LABROOT}/Scripts
. ${SCRIPTS}/shellenv.sh

info "Grading lab ${LAB} ... (may take ${TIMEOUT} seconds)"

bold "====="
${SCRIPTS}/expect.py $@
score=$?

if [[ $score -eq 255 ]]; then
    error "Something went wrong. Please check the output of your program"
    exit 0
fi

info "Score: ${score}/100"
bold "====="

if [[ $score -lt 100 ]]; then
    exit $?
else
    exit 0
fi
```

在备份 .config 之后，其会调用 capturer.py 的内容，去动态捕捉命令的输出，并按照顺序与 scores.json 的内容进行比对，从而计算评分，如果提前退出或者接收到 SIGINT 信号，则整个程序会直接退出并返回 0分。

Bug

请注意我们是根据capturer.py的返回值来进行评分，如果有问题欢迎提交issues!

Last change: 2024-09-07, commit: [1f766de](#)

1. [Makefile Tutorial ←](#)
2. [Modern CMake ←](#)
3. [Kernel: Linker Script ←](#)

内核启动

- 树莓派启动过程
- 启动 CPU 0 号核
- 切换异常级别
- 跳转到第一行 C 代码
- 初始化串口输出
- 启用 MMU

树莓派启动过程

在树莓派 3B+ 真机上，通过 SD 卡启动时，上电后会运行 ROM 中的特定固件，接着加载并运行 SD 卡上的 `bootcode.bin` 和 `start.elf`，后者进而根据 `config.txt` 中的配置，加载指定的 `kernel` 映像文件（纯 binary 格式，通常名为 `kernel8.img`）到内存的 `0x80000` 位置并跳转到该地址开始执行。

而在 QEMU 模拟的 `raspi3b`（旧版 QEMU 为 `raspi3`）机器上，则可以通过 `-kernel` 参数直接指定 ELF 格式的 `kernel` 映像文件，进而直接启动到 ELF 头部中指定的入口地址，即 `_start` 函数（实际上也在 `0x80000`，因为 ChCore 通过 linker script 强制指定了该函数在 ELF 中的位置，有兴趣请参考[附录](#)）。

启动 CPU 0 号核

`_start` 函数（位于 `kernel/arch/aarch64/boot/raspi3/init/start.s`）是 ChCore 内核启动时执行的第一块代码。由于 QEMU 在模拟机器启动时会同时开启 4 个 CPU 核心，于是 4 个核会同时开始执行 `_start` 函数。而在内核的初始化过程中，我们通常需要首先让其中一个核进入初始化流程，待进行了一些基本的初始化后，再让其他核继续执行。

思考题 1

阅读 `_start` 函数的开头，尝试说明 ChCore 是如何让其中一个核首先进入初始化流程，并让其他核暂停执行的。

Hint

可以在 [Arm Architecture Reference Manual](#) 找到 `mpidr_el1` 等系统寄存器的详细信息。

切换异常级别

AArch64 架构中，特权级被称为异常级别（Exception Level, EL），四个异常级别分别为 EL0、EL1、EL2、EL3，其中 EL3 为最高异常级别，常用于安全监控器（Secure Monitor），EL2 其次，常用于虚拟机监控器（Hypervisor），EL1 是内核常用的异常级别，也就是通常所说的内核态，EL0 是最低异常级别，也就是通常所说的用户态。

QEMU raspi3b 机器启动时，CPU 异常级别为 EL3，我们需要在启动代码中将异常级别降为 EL1，也就是进入内核态。具体地，这件事是在 `arm64_elX_to_el1` 函数（位于 `kernel/arch/aarch64/boot/raspi3/init/tools.s`）中完成的。

为了使 `arm64_elX_to_el1` 函数具有通用性，我们没有直接写死从 EL3 降至 EL1 的逻辑，而是首先判断当前所在的异常级别，并根据当前异常级别的不同，跳转到相应的代码执行。

```
BEGIN_FUNC(arm64_elX_to_el1)
    /* LAB 1 TODO 1 BEGIN */
    /* BLANK BEGIN */
    /* BLANK END */
    /* LAB 1 TODO 1 END */

    // Check the current exception level.
    cmp x9, CURRENTEL_EL1
    beq .Ltarget
    cmp x9, CURRENTEL_EL2
    beq .Lin_el2
    // Otherwise, we are in EL3.

    // Set EL2 to 64bit and enable the HVC instruction.
    mrs x9, scr_el3
    mov x10, SCR_EL3_NS | SCR_EL3_HCE | SCR_EL3_RW
    orr x9, x9, x10
    msr scr_el3, x9

    // Set the return address and exception level.
    /* LAB 1 TODO 2 BEGIN */
    /* BLANK BEGIN */
    /* BLANK END */
    /* LAB 1 TODO 2 END */

.Lin_el2:
    // Disable EL1 timer traps and the timer offset.
    mrs x9, cnthctl_el2
    orr x9, x9, CNTHCTL_EL2_EL1PCEN | CNTHCTL_EL2_EL1PCTEN
    msr cnthctl_el2, x9
    msr cntvoff_el2, xzr

    // Disable stage 2 translations.
    msr vttbr_el2, xzr

    // Disable EL2 coprocessor traps.
    mov x9, CPTR_EL2_RES1
    msr cptr_el2, x9

    // Disable EL1 FPU traps.
    mov x9, CPACR_EL1_FPen
    msr cpacr_el1, x9

    // Check whether the GIC system registers are supported.
    mrs x9, id_aa64pfr0_el1
    and x9, x9, ID_AA64PFR0_EL1_GIC
    cbz x9, .Lno_gic_sr

    // Enable the GIC system registers in EL2, and allow their use in
EL1.
    mrs x9, ICC_SRE_EL2
    mov x10, ICC_SRE_EL2_ENABLE | ICC_SRE_EL2_SRE
    orr x9, x9, x10
    msr ICC_SRE_EL2, x9

    // Disable the GIC virtual CPU interface.
```

```

msr ICH_HCR_EL2, xzr

.Lno_gic_sr:
    // Set EL1 to 64bit.
    mov x9, HCR_EL2_RW
    msr hcr_el2, x9

    // Set the return address and exception level.
    adr x9, .Ltarget
    msr elr_el2, x9
    mov x9, SPSR_ELX_DAIF | SPSR_ELX_EL1H
    msr spsr_el2, x9

    isb
    eret

.Ltarget:
    ret
END_FUNC(arm64_elX_to_el1)

```

练习题 2

在 arm64_elX_to_el1 函数的 LAB 1 TODO 1 处填写一行汇编代码，获取 CPU 当前异常级别。

Hint

通过 CurrentEL 系统寄存器可获得当前异常级别。通过 GDB 在指令级别单步调试可验证实现是否正确。注意参考文档理解 CurrentEL 各个 bits 的意义。

eret 指令可用于从高异常级别跳到更低的异常级别，在执行它之前我们需要设置 elr_elx（异常链接寄存器）和 spsr_elx（保存的程序状态寄存器），分别控制 eret 执行后的指令地址（PC）和程序状态（包括异常返回后的异常级别）。

练习题 3

在 arm64_elX_to_el1 函数的 LAB 1 TODO 2 处填写大约 4 行汇编代码，设置从 EL3 跳转到 EL1 所需的 elr_el3 和 spsr_el3 寄存器值。

Hint

elr_el3 的正确设置应使得控制流在 eret 后从 arm64_elX_to_el1 返回到 _start 继续执行初始化。 spsr_el3 的正确设置应正确屏蔽 DAIF 四类中断，并且将 SP 正确设置为 EL1h。在设置好这两个系统寄存器后，不需要立即 eret。

练习完成后，可使用 GDB 跟踪内核代码的执行过程，由于此时不会有任何输出，可通过是否准确从 arm64_elX_to_el1 函数返回到 _start 来判断代码的正确性。

跳转到第一行 C 代码

降低异常级别到 EL1 后，我们准备从汇编跳转到 C 代码，在此之前我们先设置栈（SP）。因此，`_start` 函数在执行 `arm64_elx_to_el1` 后，即设置内核启动阶段的栈，并跳转到第一个 C 函数 `init_c`。

```
BEGIN_FUNC(_start)
    mrs x8, mpidr_el1
    and x8, x8, #0xFF
    cbz x8, primary

    /* Wait for bss clear */
wait_for_bss_clear:
    adr x0, clear_bss_flag
    ldr x1, [x0]
    cmp x1, #0
    bne wait_for_bss_clear

    /* Set cntkctl_el1 to enable cntvct_el0.
       * Enable it when you need to get current tick
       * at EL0, e.g. Running aarch64 ROS2 demos
    mov x10, #0b11
    msr cntkctl_el1, x10 */

    /* Turn to el1 from other exception levels. */
    bl arm64_elX_to_el1

    /* Prepare stack pointer and jump to C. */
    mov x1, #INIT_STACK_SIZE
    mul x1, x8, x1
    adr x0, boot_cpu_stack
    add x0, x0, x1
    add x0, x0, #INIT_STACK_SIZE
    mov sp, x0

wait_until_smp_enabled:
    /* CPU ID should be stored in x8 from the first line */
    mov x1, #8
    mul x2, x8, x1
    ldr x1, =secondary_boot_flag
    add x1, x1, x2
    ldr x3, [x1]
    cbz x3, wait_until_smp_enabled

    /* Set CPU id */
    mov x0, x8
    b secondary_init_c

    /* Should never be here */
    b .

primary:

    /* Turn to el1 from other exception levels. */
    bl arm64_elX_to_el1

    /* Prepare stack pointer and jump to C. */
    adr x0, boot_cpu_stack
    add x0, x0, #INIT_STACK_SIZE
    mov sp, x0

    b init_c
```

```
/* Should never be here */
b .
END_FUNC(_start)
```

思考题 4

说明为什么要在进入 C 函数之前设置启动栈。果不设置，会发生什么？

进入 `init_c` 函数后，第一件事首先通过 `clear_bss` 函数清零了 `.bss` 段，该段用于存储未初始化的全局变量和静态变量（具体请参考[附录](#)）。

思考题 5

在实验 1 中，其实不调用 `clear_bss` 也不影响内核的执行，请思考不清理 `.bss` 段在之后的何种情况下会导致内核无法工作。

初始化串口输出

到目前为止我们仍然只能通过 GDB 追踪内核的执行过程，而无法看到任何输出，这无疑是对于我们写操作系统的积极性的一种打击。因此在 `init_c` 中，我们启用树莓派的 UART 串口，从而能够输出字符。

在 `kernel/arch/aarch64/boot/raspi3/peripherals/uart.c` 已经给出了 `early_uart_init` 和 `early_uart_send` 函数，分别用于初始化 UART 和发送单个字符（也就是输出字符）。

```
void uart_send_string(char *str)
{
    /* LAB 1 TODO 3 BEGIN */
    /* BLANK BEGIN */
    /* BLANK END */
    /* LAB 1 TODO 3 END */
}
```

练习题6

在 `kernel/arch/aarch64/boot/raspi3/peripherals/uart.c` 中 `LAB 1 TODO 3` 处实现通过 UART 输出字符串的逻辑。

第一个字符串

恭喜！我们终于在内核中输出了第一个字符串！
感兴趣的同学请思考 `early_uart_send` 究竟是怎么输出字符的。

启用 MMU

在内核的启动阶段，还需要配置启动页表（`init_kernel_pt` 函数），并启用 MMU（`el1_mmu_activate` 函数），使可以通过虚拟地址访问内存，从而为之后跳转到高地址作准备（内核通常运行在虚拟地址空间 `0xffffffff0000000000` 之后的高地址）。

关于配置启动页表的内容由于包含关于页表的细节，将在本实验下一部分实现，目前直接启用 MMU。

在 EL1 异常级别启用 MMU 是通过配置系统寄存器 `sctlr_el1` 实现的（Arm Architecture Reference Manual D13.2.118）。具体需要配置的字段主要包括：

- 是否启用 MMU（`M` 字段）
- 是否启用对齐检查（`A SA0 SA nAA` 字段）
- 是否启用指令和数据缓存（`C I` 字段）

练习题7

在 `kernel/arch/aarch64/boot/raspi3/init/tools.s` 中 `LAB 1 TODO 4` 处填写一行汇编代码，以启用 MMU。

由于没有配置启动页表，在启用 MMU 后，内核会立即发生地址翻译错误（Translation Fault），进而尝试跳转到异常处理函数（Exception Handler），该异常处理函数的地址为异常向量表基址（`vbar_el1` 寄存器）加上 `0x200`。此时我们没有设置异常向量表（`vbar_el1` 寄存器的值是0），因此执行流会来到 `0x200` 地址，此处的代码为非法指令，会再次触发异常并跳转到 `0x200` 地址。使用 GDB 调试，在 GDB 中输入 `continue` 后，待内核输出停止后，按 `Ctrl-C`，可以观察到内核在 `0x200` 处无限循环。

Important

以上为Lab1 Part1 的内容

Last change: 2024-09-07, commit: [debf8d0](#)

页表映射

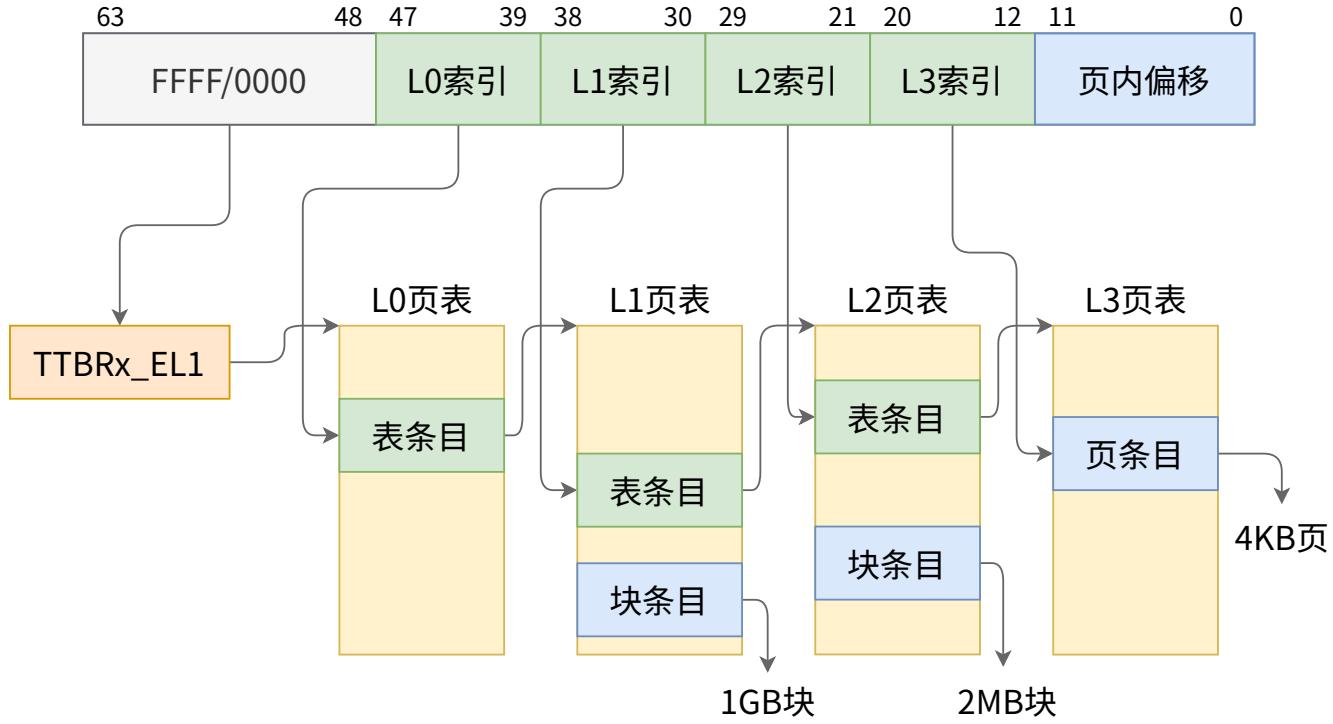
- AArch64 地址翻译
- 配置内核启动页表

AArch64 地址翻译

在配置内核启动页表前，我们首先回顾实验涉及到的体系结构知识。这部分内容课堂上已经学习过，如果你已熟练掌握则可以直接跳过这里的介绍（但不要跳过思考题）。

在 AArch64 架构的 EL1 异常级别存在两个页表基址寄存器： `ttbr0_el1`¹ 和 `ttbr1_el1`²，分别用作虚拟地址空间低地址和高地址的翻译。那么什么地址范围称为“低地址”，什么地址范围称为“高地址”呢？这由 `tcr_el1` 翻译控制寄存器³控制，该寄存器提供了丰富的可配置性，可决定 64 位虚拟地址的高多少位为 0 时，使用 `ttbr0_el1` 指向的页表进行翻译，高多少位为 1 时，使用 `ttbr1_el1` 指向的页表进行翻译⁴。一般情况下，我们会将 `tcr_el1` 配置为高低地址各有 48 位的地址范围，即，`0x0000_0000_0000_0000 ~ 0x0000_ffff_ffff_ffff` 为低地址，`0xffff_0000_0000_0000 ~ 0xffff_ffff_ffff_ffff` 为高地址。

了解了何决定使用 `ttbr0_el1` 还是 `ttbr1_el1` 指向的页表，再来看地址翻译过程何进行。通常我们会将系统配置为使用 4KB 翻译粒度、4 级页表（L0 到 L3），同时在 L1 和 L2 页表中分别允许映射 2MB 和 1GB 大页（或称为块）⁵，因此地址翻译的过程下图所示：



其中，当映射为 1GB 块或 2MB 块时，图中 L2、L3 索引或 L3 索引的位置和低 12 位共同组成块内偏移。

每一级的每一个页表占用一个 4KB 物理页，称为页表页（Page Table Page），其中有 512 个条目，每个条目占 64 位。AArch64 中，页表条目称为描述符（descriptor）⁶，最低位（bit[0]）为 1 时，描述符有效，否则无效。有效描述符有两种类型，一种指向下一级页表（称为表描述符），另一种指向物理块（大页）或物理页（称为块描述符或页描述符）。在上面所说的地址翻译配置下，描述符结构 下（“Output address”在这里即物理地址，一些地方称为物理页帧号（Page Frame Number, PFN））：

- L0、L1、L2 页表描述符

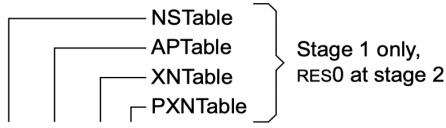
Block header fields:

- Upper block attributes
- RES0
- Output address[47:n]
 - n
 - $n-1$
- RES0
- nT
- RES0
- Lower block attributes
- 0
- 1

With the 4KB granule size, for the level 1 descriptor n is 30, and for the level 2 descriptor, n is 21.

With the 16KB granule size, for the level 2 descriptor, n is 25.

With the 64KB granule size, for the level 2 descriptor, n is 29.



The diagram illustrates the memory hierarchy for a 64-bit system. The 64-bit address bus is divided into several fields:

- Bits 63-61: Labeled "Table".
- Bit 60: Labeled "IGNORED".
- Bits 59-58: Labeled "RES0".
- Bits 51-50: Labeled "Next-level table address[47:m]".
- Bits 48-47: Labeled "RES0†".
- Bits 12-11: Labeled "IGNORED".
- Bits 2-0: Labeled "2", "1", and "0".

With the 4KB granule size m is 12[‡], with the 16KB granule size m is 14, and with the 64KB granule size, m is 16.

A level 0 Table descriptor returns the address of the level 1 table.

A level 1 Table descriptor returns the address of the level 2 table.

A level 2 Table descriptor returns the address of the level 3 table.

‡ When m is 12, the RES0 field shown for bits[$(m-1):12$] is absent.

- L3 页表描述符

The diagram shows a register map with bit 63 labeled "Reserved" and bit 0 labeled "RES0".

The diagram shows the structure of the **Page, 4KB granule 48-bit OA** field. It consists of several fields: **Upper attributes**, **RES0**, **Output address[47:12]**, and **Lower attributes**. The **Output address[47:12]** field is further divided into two sub-fields: **63**, **50 48 47**, and **12 11**, with a final **2 1 0** at the end.

思考题 8

请思考多级页表相比单级页表带来的优势和劣势（果有的话），并计算在AArch64页表中分别以4KB粒度和2MB粒度映射0~4GB地址范围所需的物理内存大小（或页表页数量）。

页表描述符中除了包含下一级页表或物理页/块的地址，还包含对内存访问进行控制的属性（attribute）。这里涉及到太多细节，本文档限于篇幅只介绍最常用的几个页/块描述符中的属性字段：

字段	位	描述
UXN	bit[54]	置为 1 表示非特权态无法执行 (Unprivileged eXecute-Never)
PXN	bit[53]	置为 1 表示特权态无法执行 (Privileged eXecute-Never)
nG	bit[11]	置为 1 表示该描述符在 TLB 中的缓存只对当前 ASID 有效
AF	bit[10]	置为 1 表示该页/块在上一次 AF 置 0 后被访问过
SH	bits[9:8]	表示可共享属性 ⁷
AP	bits[7:6]	表示读写等数据访问权限 ⁸
AttrIndx	bits[4:2]	表示内存属性索引，间接指向 mair_el1 寄存器中配置的属性 ⁹ ，用于控制将物理页映射为正常内存 (normal memory) 或设备内存 (device memory)，以及控制 cache 策略等

配置内核启动页表

有了关于页表配置的前置知识，我们终于可以开始配置内核的启动页表了。

操作系统内核通常运行在虚拟内存的高地址（前所述，`0xffff_0000_0000_0000` 之后的虚拟地址）。通过对内核页表的配置，将虚拟内存高地址映射到内核实际所在的物理内存，在执行内核代码时，PC 寄存器的值是高地址，对全局变量、栈等的访问都使用高地址。在内核运行时，除了需要访问内核代码和数据等，往往还需要能够对任意物理内存和外设内存 (MMIO) 进行读写，这种读写同样通过高地址进行。

因此，在内核启动时，首先需要对内核自身、其余可用物理内存和外设内存进行虚拟地址映射，最简单的映射方式是一对一的映射，即将虚拟地址 `0xffff_0000_0000_0000 + addr` 映射到 `addr`。需要注意的是，在 ChCore 实验中我们使用了 `0xffff_ff00_0000_0000` 作为内核虚拟地址的开始（注意开头 f 数量的区别），不过这不影响我们对知识点的理解。

在树莓派 3B+ 机器上，物理地址空间分布 下¹⁰:

物理地址范围	对应设备
<code>0x00000000 ~ 0x3f000000</code>	物理内存 (SDRAM)
<code>0x3f000000 ~ 0x40000000</code>	共享外设内存
<code>0x40000000 ~ 0xffffffff</code>	本地 (每个 CPU 核独立) 外设内存

现在将目光转移到 `kernel/arch/aarch64/boot/rasp3/init/mmu.c` 文件，我们需要在 `init_kernel_pt` 为内核配置从 `0x00000000` 到 `0x80000000` (`0x40000000` 后的 1G, ChCore 只需使用这部分地址中的本地外设) 的映射，其中 `0x00000000` 到 `0x3f000000` 映射为 normal memory, `0x3f000000` 到 `0x80000000` 映射为 device memory, 其中

0x00000000 到 0x40000000 以 2MB 块粒度映射， 0x40000000 到 0x80000000 以 1GB 块粒度映射。

思考题 9

请结合上述地址翻译规则，计算在练习题 10 中，你需要映射几个 L2 页表条目，几个 L1 页表条目，几个 L0 页表条目。页表页需要占用多少物理内存？

练习题 10

在 `init_kernel_pt` 函数的 LAB 1 TODO 5 处配置内核高地址页表（`boot_ttbr1_l0`、`boot_ttbr1_l1` 和 `boot_ttbr1_l2`），以 2MB 粒度映射。

Hint

你只需要将 `addr` (0x00000000 到 0x80000000) 按照要求的页粒度一一映射到 `KERNEL_VADDR + addr (vaddr)` 上。 `vaddr` 对应的物理地址是 `vaddr - KERNEL_VADDR`。 Attributes 的设置请参考给出的低地址页表配置。

思考题11

请思考在 `init_kernel_pt` 函数中为什么还要为低地址配置页表，并尝试验证自己的解释。

完成 `init_kernel_pt` 函数后，ChCore 内核便可以在 `el1_mmu_activate` 中将 `boot_ttbr1_l0` 等物理地址写入实际寄存器(`ttbr1_el1`)，随后启用 MMU 后继续执行，并通过 `start_kernel` 跳转到高地址，进而跳转到内核的 `main` 函数（位于 `kernel/arch/aarch64/main.c`, 尚未发布，以 binary 提供）。

思考题12

在一开始我们暂停了三个其他核心的执行，根据现有代码简要说明它们什么时候会恢复执行。思考为什么一开始只让 0 号核心执行初始化流程？

Hint

`secondary_boot_flag` 将在 `main` 函数执行完时钟，调度器，锁的初始化后被设置。

Success

以上为Lab1 Part2 的内容 果顺利的话 运行make grade你会得到100/100

1. Arm Architecture Reference Manual, D13.2.144 ↵
2. Arm Architecture Reference Manual, D13.2.147 ↵
3. Arm Architecture Reference Manual, D13.2.131 ↵
4. Arm Architecture Reference Manual, D5.2 Figure D5-13 ↵
5. 操作系统：原理与实现 ↵
6. Arm Architecture Reference Manual, D5.3 ↵
7. Arm Architecture Reference Manual, D5.5 ↵
8. Arm Architecture Reference Manual, D5.4 ↵
9. Arm Architecture Reference Manual, D13.2.97 ↵
10. [bcm2836-peripherals.pdf](#) & [Raspberry Pi Hardware - Peripheral Addresses](#) ↵

Lab 2：内存管理

本实验主要目的在于让同学们熟悉内核启动过程中对内存的初始化和内核启动后对物理内存和页表的管理，包括三个部分。

1. [物理内存管理](#): 理解并完成伙伴系统以及SLAB系统
2. [虚拟页表管理](#): 深入理解页表分配机制以及页表项权限机制，并完成页表分配函数。
3. [缺页异常处理](#): 理解aarch64架构下的异常处理机制，并按照页表项的配置完成按需分配以及写时拷贝的缺页管理设置。

Warning

本Lab不包括代码导读

跟先前的Lab相同，本实验代码包含了基础的 ChCore 操作系统镜像，除了练习题相关部分的源码以外（指明需要阅读的代码），其余部分通过二进制格式提供。在正确完成本实验的练习题之后，你可以在树莓派3B+QEMU或开发板上进入 ChCore shell。

Last change: 2024-09-30, commit: [9684f5b](#)

物理内存管理

- 伙伴系统
- SLAB分配器
- Kmalloc

伙伴系统

内核初始化过程中，需要对内存管理模块进行初始化（`mm_init` 函数），首先需要把物理内存管起来，从而使内核代码可以动态地分配内存。

ChCore 使用伙伴系统（buddy system）¹ 对物理页进行管理，在 `mm_init` 中对伙伴系统进行了初始化。为了使物理内存的管理可扩展，ChCore 在 `mm_init` 的开头首先调用平台特定的 `parse_mem_map` 函数，该函数解析并返回了可用物理内存区域，然后再对各可用物理内存区域初始化伙伴系统。

伙伴系统中的每个内存块都有一个阶（order）表示大小，阶是从 0 到指定上限 `BUDDY_MAX_ORDER` 的整数。一个 n 阶的块的大小为 $2^n \times PAGE_SIZE$ ，因此这些内存块的大小正好是比它小一个阶的内存块的大小的两倍。内存块的大小是 2 次幂对齐，使地址计算变得简单。当一个较大的内存块被分割时，它被分成两个较小的内存块，这两个小内存块相互成为唯一的伙伴。一个分割的内存块也只能与它唯一的伙伴块进行合并（合并成他们分割前的块）。

ChCore 中每个由伙伴系统管理的内存区域称为一个 `struct phys_mem_pool`，该结构体中包含物理页元信息的起始地址（`page_metadata`）、伙伴系统各阶内存块的空闲链表（`free_lists`）等。

练习题1

完成 `kernel/mm/buddy.c` 中的 `split_chunk`、`merge_chunk`、`buddy_get_pages`、和 `buddy_free_pages` 函数中的 LAB 2 TODO 1 部分，其中 `buddy_get_pages` 用于分配指定阶大小的连续物理页，`buddy_free_pages` 用于释放已分配的连续物理页。

Hint

- 可以使用 `kernel/include/common/list.h` 中提供的链表相关函数和宏 `init_list_head`、`list_add`、`list_del`、`list_entry` 来对伙伴系统中的空闲链表进行操作
- 可使用 `get_buddy_chunk` 函数获得某个物理内存块的伙伴块
- 更多提示见代码注释

SLAB分配器

我们希望通过基于伙伴系统的物理内存管理，在内核中进行动态内存分配，也就是可以使用 `kmalloc` 函数（对应用户态的 `malloc`）。ChCore 的 `kmalloc` 对于较小的内存分配需求采用 SLAB 分配器²，对于较大的分配需求则直接从伙伴系统中分配物理页。动态分配出的物理页被转换为内核虚拟地址（Kernel Virtual Address, KVA），也就是在 LAB 1 中我们映射的 `0xfffff_ff00_0000_0000` 之后的地址。我们在练习题 1 中已经实现了伙伴系统，接下来让我们实现 SLAB 分配器吧。

练习题2

完成 `kernel/mm/slab.c` 中的 `choose_new_current_slab`、`alloc_in_slab_impl` 和 `free_in_slab` 函数中的 LAB 2 TODO 2 部分，其中 `alloc_in_slab_impl` 用于在 slab 分配器中分配指定阶大小的内存，而 `free_in_slab` 则用于释放上述已分配的内存。

Hint

- 你仍然可以使用上个练习中提到的链表相关函数和宏来对 SLAB 分配器中的链表进行操作
- 更多提示见代码注释

Kmalloc

有了伙伴系统和 SLAB 分配器，就可以实现 `kmalloc` 了。

练习题 3

完成 `kernel/mm/kmalloc.c` 中的 `_kmalloc` 函数中的 LAB 2 TODO 3 部分，在适当位置调用对应的函数，实现 `kmalloc` 功能

Hint

- 你可以使用 `get_pages` 函数从伙伴系统中分配内存，使用 `alloc_in_slab` 从 SLAB 分配器中分配内存
- 更多提示见代码注释

Kmalloc

现在内核中已经能够正常使用 `kmalloc` 和 `kfree` 了

Success

以上为Lab2 Part1的所有内容。

正确完成这一部分的练习题后，运行 `make grade`，你应当能够得到 30 分。注意，测试可能会遗漏你代码中的一些问题。因此即使通过这部分测试，代码中的隐藏问题也可能会影响后续实验产生影响导致无法通过最终的测试。不过，我们会按照 `make grade` 的结果为你计分。^_^

Last change: 2025-02-14, commit: [773916e](#)

1. 操作系统：原理与实现，5.1.3 伙伴系统原理 [←](#)
2. 操作系统：原理与实现，5.1.5 SLAB 分配器的基本设计 [←](#)

页表管理

在LAB 1 中我们已经详细介绍了 AArch64 的地址翻译过程，并介绍了各级页表和不同类型的页表描述符，最后在内核启动阶段配置了一个粗粒度的启动页表。现在，我们需要为用户态应用程序准备一个更细粒度的页表实现，提供映射、取消映射、查询等功能。

练习题4

完成 `kernel/arch/aarch64/mm/page_table.c` 中的 `query_in_pgtbl`、
`map_range_in_pgtbl_common`、`unmap_range_in_pgtbl` 和 `mprotect_in_pgtbl` 函数
中的 LAB 2 TODO 4 部分，分别实现页表查询、映射、取消映射和修改页表权限的操作，
以 4KB 页为粒度。

Hint

- 需要实现的函数内部无需刷新 TLB，TLB 刷新会在这些函数的外部进行
- 实现中可以使用 `get_next_ptp`、`set_pte_flags`、`virt_to_phys`、
`GET_LX_INDEX` 等已经给定的函数和宏
- 更多提示见代码注释

页表配错了怎么办？

在Aarch64的架构中，每当系统进入异常处理流程，寄存器 `ELR_EL1` 将保存错误发生的指令地址，而对于出错的虚拟内存地址，你可以通过查询 `FAR_EL1` 找到。

思考题5

阅读 Arm Architecture Reference Manual，思考要在操作系统中支持写时拷贝（Copy-on-Write，CoW）¹需要配置页表描述符的哪个/哪些字段，并在发生页错误时 何处理。
(在完成第三部分后，你也可以阅读页错误处理的相关代码，观察 ChCore 是 何支持 Cow 的)

思考题6

为了简单起见，在 ChCore 实验 Lab1 中没有为内核页表使用细粒度的映射，而是直接沿用了启动时的粗粒度页表，请思考这样做有什么问题。

挑战题7

使用前面实现的 `page_table.c` 中的函数，在内核启动后的 `main` 函数中重新配置内核页表，进行细粒度的映射。

Success

以上为Lab2 Part2的所有内容

正确完成该练习题后，运行 `make grade`，你应当能够得到 70 分。同样的，正确实现功能是通过测试的充分非必要条件。

Last change: 2025-02-14, commit: [773916e](#)

1. 操作系统：原理与实现，12.4 原子更新技术：写时拷贝 [←](#)

缺页异常处理

缺页异常（page fault）是操作系统实现延迟内存分配的重要技术手段。当处理器发生缺页异常时，它会将发生错误的虚拟地址存储于 `FAR_ELx` 寄存器中，并触发相应的异常处理流程。ChCore 对该异常的处理最终实现在 `kernel/arch/aarch64/irq/pgfault.c` 中的 `do_page_fault` 函数。本次实验暂时不涉及前面的异常初步处理及转发相关内容，我们仅需要关注操作系统是何处缺页异常的。

练习题8

完成 `kernel/arch/aarch64/irq/pgfault.c` 中的 `do_page_fault` 函数中的 LAB 2 TODO 5 部分，将缺页异常转发给 `handle_trans_fault` 函数。

在 ChCore 中，一个进程的虚拟地址空间由多段“虚拟地址区域”（VMR，又称 VMA）组成，一段 VMR 记录了这段虚拟地址对应的“物理内存对象”（PMO），而 PMO 中则记录了物理地址相关信息。因此，想要处理缺页异常，首先需要找到当前进程发生页错误的虚拟地址所处的 VMR，进而才能得知其对应的物理地址，从而在页表中完成映射。

练习题9

完成 `kernel/mm/vmspace.c` 中的 `find_vmr_for_va` 函数中的 LAB 2 TODO 6 部分，找到一个虚拟地址找在其虚拟地址空间中的 VMR。

Hint

- 一个虚拟地址空间所包含的 VMR 通过 `rb_tree` 的数据结构保存在 `vmspace` 结构体的 `vmr_tree` 字段
- 可以使用 `kernel/include/common/rbtree.h` 中定义的 `rb_search`、`rb_entry` 等函数或宏来对 `rb_tree` 进行搜索或操作

缺页处理主要针对 `PMO_SHM` 和 `PMO_ANONYM` 类型的 PMO，这两种 PMO 的物理页是在访问时按需分配的。缺页处理逻辑为首先尝试检查 PMO 中当前 fault 地址对应的物理页是否存在（通过 `get_page_from_pmo` 函数尝试获取 PMO 中 offset 对应的物理页）。若对应物理页未分配，则需要分配一个新的物理页，再将页记录到 PMO 中，并增加页表映射。若对应物理页已分配，则只需要修改页表映射即可。

练习题10

完成 `kernel/mm/pgfault_handler.c` 中的 `handle_trans_fault` 函数中的 LAB 2 TODO 7 部分（函数内共有 3 处填空，不要遗漏），实现 `PMO_SHM` 和 `PMO_ANONYM` 的按需物理页分配。你可以阅读代码注释，调用你之前见到过的相关函数来实现功能。

挑战题 11

我们在 `map_range_in_pgtbl_common`、`unmap_range_in_pgtbl` 函数中预留了没有被使用过的参数 `rss` 用来来统计map映射中实际的物理内存使用量¹，你需要修改相关的代码来通过 `Compute physical memory` 测试，不实现该挑战题并不影响其他部分功能的实现及测试。果你想检测是否通过此部分测试，需要修改 `kernel/config.cmake` 中 `CHCORE_KERNEL_PM_USAGE_TEST` 为ON

Challenge

为了防止你通过尝试在打印 `scores.json` 里的内容来逃避检查，我们在每一次评分前都会修改elf序列号段的信息并让其chcore在评分点进行打出，当且仅当评分程序捕捉到序列号输出之后，我们才会对检查进行打分，果没有通过序列号验证，你的评分将为0分。

Success

以上为Lab2 Part3。正确完成上述练习题后，运行 `make qemu` 后 ChCore 应当能正常进入 Shell；运行 `make grade`，你应当能够得到 100 分。果你无法通过测试，请考虑到也有可能是你前面两个部分的实现存在漏洞。

Last change: 2024-11-16, commit: [3c786f6](#)

1. https://en.wikipedia.org/wiki/Resident_set_size ↵

Lab 3：进程与线程

用户进程是操作系统对在用户模式运行中的程序的抽象。在Lab 1 和Lab 2 中，已经完成了内核的启动和物理内存的管理，以及一个可供用户进程使用的页表实现。现在，我们将一步一步支持用户态程序的运行。本实验包括五个部分：

1. [RTFSC\(2\)](#): 代码导读，了解Chcore微内核的核心机制以及用户态和内核态是如何进行交互的。
2. [线程管理](#): 支持创建第一个用户态进程和线程，分析代码如何从内核态切换到用户态。
3. [异常处理](#): 完善异常处理流程，为系统添加必要的异常处理的支持。
4. [系统调用](#): 正确处理部分系统调用，保证用户的正常输出。
5. [用户态程序编写](#): 编写一个简单用户程序，使用提供的 ChCore libc 进行编译，并加载至内核镜像中。

工具链准备

从Lab3开始我们开放了用户态的一些代码，你需要使用 下的命令下载libc.

```
git submodule update --init --recursive
```

关于超时

如果你发现时不时因为超时原因而无法正常通过测试，你可以尝试修改Lab文件夹的Makefile的TIMEOUT变量设置超时时间。

Last change: 2024-11-24, commit: [2367a29](#)

RTFSC (3)

Note

此为代码导读的第三部分，请仔细阅读。同之前的章节相同，本节不包含习题。

本次代码导读主要聚焦从main函数开始自上而下讲解Lab2 Lab3内核态的资源管理机制以及用户态和内核态的互相调用

Hint

你可能需要重新结合Lab2/Lab3的开放代码来理解本章

内核初始化

```
/*
 * @boot_flag is boot flag addresses for smp;
 * @info is now only used as board_revision for rpi4.
 */
void main(paddr_t boot_flag, void *info)
{
    u32 ret = 0;

    /* Init big kernel lock */
    ret = lock_init(&big_kernel_lock);
    kinfo("[ChCore] lock init finished\n");
    BUG_ON(ret != 0);

    /* Init uart: no need to init the uart again */
    uart_init();
    kinfo("[ChCore] uart init finished\n");

    /* Init per_cpu info */
    init_per_cpu_info(0);
    kinfo("[ChCore] per-CPU info init finished\n");

    /* Init mm */
    mm_init(info);

    kinfo("[ChCore] mm init finished\n");

    void lab2_test_buddy(void);
    lab2_test_buddy();
    void lab2_test_kmalloc(void);
    lab2_test_kmalloc();
    void lab2_test_page_table(void);
    lab2_test_page_table();
#ifndef CHCORE_KERNEL_PM_USAGE_TEST
    void lab2_test_pm_usage(void);
    lab2_test_pm_usage();
#endif
    /* Mapping KSTACK into kernel page table. */
    map_range_in_pgtbl_kernel((void*)((unsigned long)boot_ttbr1_l0 + KBASE),
        KSTACKx_ADDR(0),
        (unsigned long)(cpu_stacks[0]) - KBASE,
        CPU_STACK_SIZE, VMR_READ | VMR_WRITE);

    /* Init exception vector */
    arch_interrupt_init();
    timer_init();
    kinfo("[ChCore] interrupt init finished\n");

    /* Enable PMU by setting PMCR_EL0 register */
    pmu_init();
    kinfo("[ChCore] pmu init finished\n");

    /* Init scheduler with specified policy */
#ifndef CHCORE_KERNEL_SCHED_PBFIFO
    sched_init(&pbfifo);
#else
    sched_init(&pbrr);
#endif
```

```
#else
    sched_init(&rr);
#endif
    kinfo("[ChCore] sched init finished\n");

    init_fpu_owner_locks();

    /* Other cores are busy looping on the boot_flag, wake up those cores */
    enable_smp_cores(boot_flag);
    kinfo("[ChCore] boot multicore finished\n");

#ifndef CHCORE_KERNEL_TEST
    kinfo("[ChCore] kernel tests start\n");
    run_test();
    kinfo("[ChCore] kernel tests done\n");
#endif /* CHCORE_KERNEL_TEST */

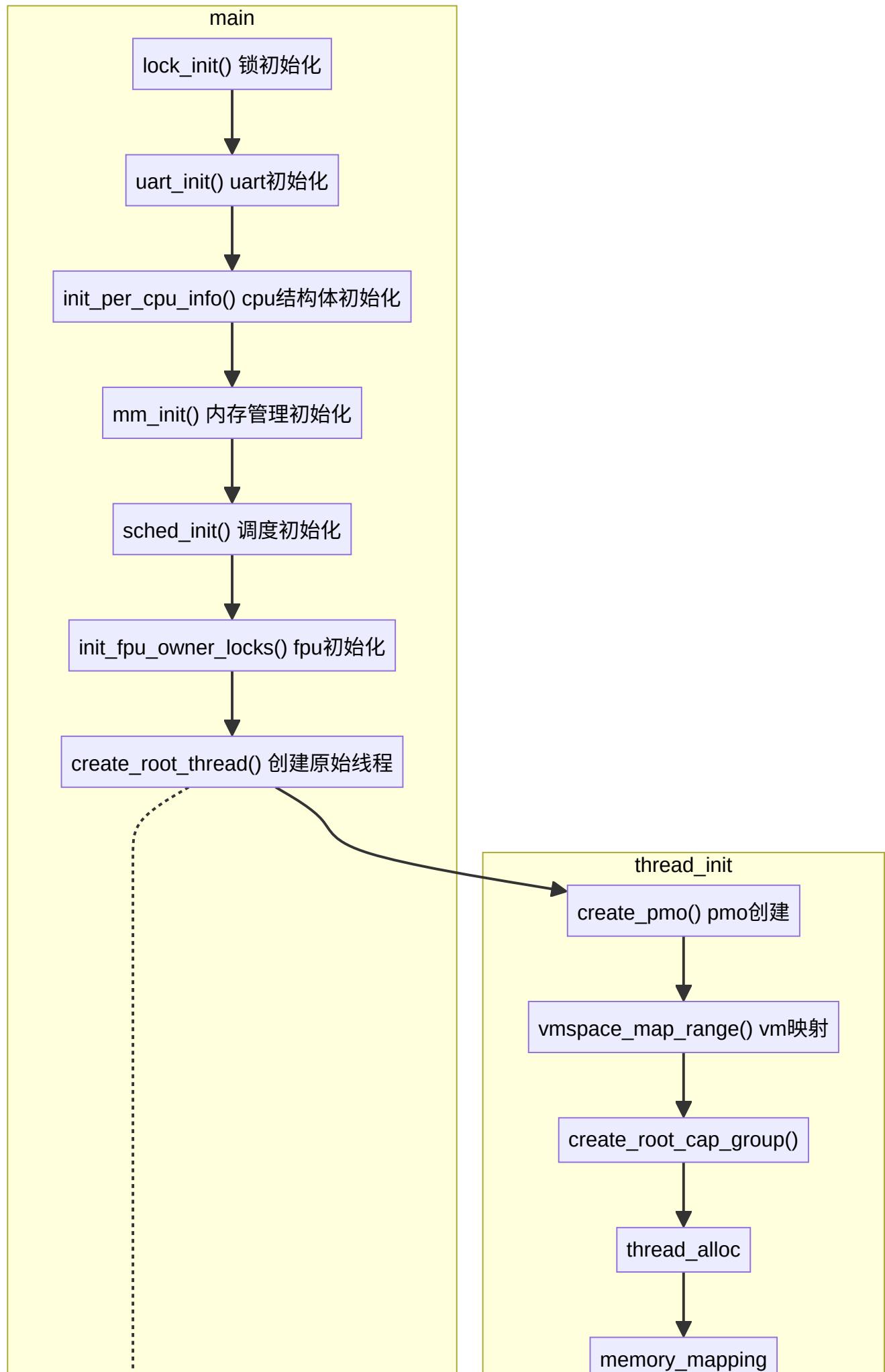
#if FPU_SAVING_MODE == LAZY_FPU_MODE
    disable_fpu_usage();
#endif

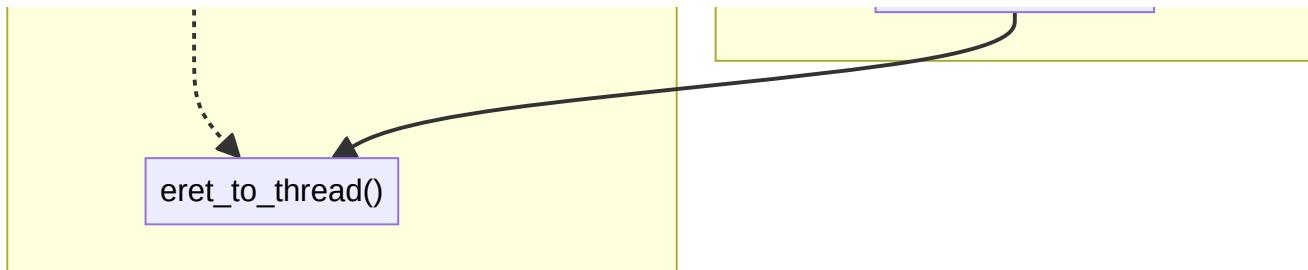
    /* Create initial thread here, which use the `init.bin` */
    create_root_thread();
    kinfo("[ChCore] create initial thread done\n");
    kinfo("End of Kernel Checkpoints: %s\n", serial_number);

    /* Leave the scheduler to do its job */
    sched();

    /* Context switch to the picked thread */
```

以下为Chcore内核初始化到运行第一个用户线程的主要流程图



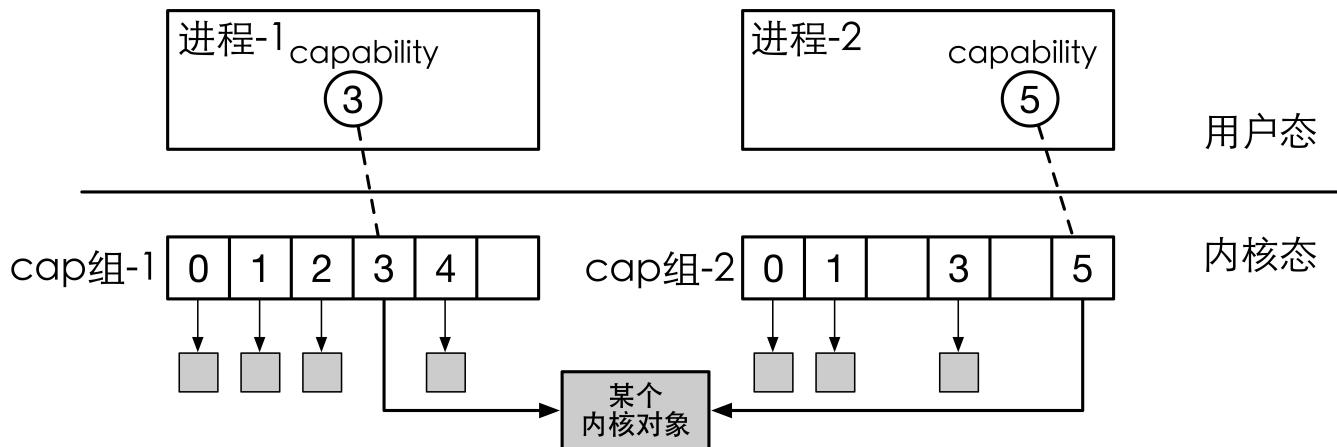


我们在 Lab2 中主要完成 mm_init 以及内存管理器与 vmspace 和 pmo 的互联，现在我们再从第一个线程创建的数据流来梳理并分析 Chcore 微内核的资源管理模式。

内核对象管理

在 Chcore 中所有的系统资源都叫做 object (对象)，用面向对象的方法进行理解的话，object 即为不同内核对象例 vmspace, pmo, thread(等等)的父类，Chcore 通过能力组机制管理所有的系统资源，能力组本身只是一个包含指向 object 的指针的数组

- 所有进程/线程都有一个独立的能力组，拥有一个全局唯一ID (**Badge**)
- 所有对象（包括进程或**能力组本身**）都属于一个或多个能力组当中，也就是说子进程与线程将属于父进程的能力组当中，在某个能力组的对象拥有一个能力组内的能力ID(**cap**)。
- 对象可以共享，即单个对象可以在多个能力组中共存，同时在不同cap_group中可以有不同的**cap**
- 对所有对象的取用和返还都使用引用计数进行追踪。当引用计数为0后，当内核垃圾回收器唤醒后，会自动回收。
- 能力组内的能力具有权限，表明该能力是否能被共享(CAP_RIGHT_COPY)以及是否能被删除(CAP_RIGHT_REVOCATION)



```
struct object {
    u64 type;
    u64 size;
    /* Link all slots point to this object */
    struct list_head copies_head;
    /* Currently only protect copies list */
    struct lock copies_lock;
    /*
     * refcount is added when a slot points to it and when get_object is
     * called. Object is freed when it reaches 0.
     */
    volatile unsigned long refcount;

    /*
     * opaque marks the end of this struct and the real object will be
     * stored here. Now its address will be 8-byte aligned.
     */
    u64 opaque[];
};

const obj_deinit_func obj_deinit_tbl[TYPE_NR] = {
    [0 ... TYPE_NR - 1] = NULL,
    [TYPE_CAP_GROUP] = cap_group_deinit,
    [TYPE_THREAD] = thread_deinit,
    [TYPE_CONNECTION] = connection_deinit,
    [TYPE_NOTIFICATION] = notification_deinit,
    [TYPE_IRQ] = irq_deinit,
    [TYPE_PMO] = pmo_deinit,
    [TYPE_VMSPACE] = vmspace_deinit,
#endif /* CHCORE_OPENTRUSTEE */
    [TYPE_CHANNEL] = channel_deinit,
    [TYPE_MSG_HDL] = msg_hdl_deinit,
#endif /* CHCORE_OPENTRUSTEE */
    [TYPE_PTRACE] = ptrace_deinit
};

void *obj_alloc(u64 type, u64 size)
{
    u64 total_size;
    struct object *object;

    total_size = sizeof(*object) + size;
    object = kzalloc(total_size);
    if (!object)
        return NULL;

    object->type = type;
    object->size = size;
    object->refcount = 0;

    /*
     * If the cap of the object is copied, then the copied cap (slot) is
     * stored in such a list.
     */
    init_list_head(&object->copies_head);
    lock_init(&object->copies_lock);

    return object->opaque;
```

```

}

void __free_object(struct object *object)
{
#ifndef TEST_OBJECT
    obj_deinit_func func;

    if (object->type == TYPE_THREAD)
        clear_fpu_owner(object);

    /* Invoke the object-specific free routine */
    func = obj_deinit_tbl[object->type];
    if (func)
        func(object->opaque);
#endif

    BUG_ON(!list_empty(&object->copies_head));
    kfree(object);
}

```

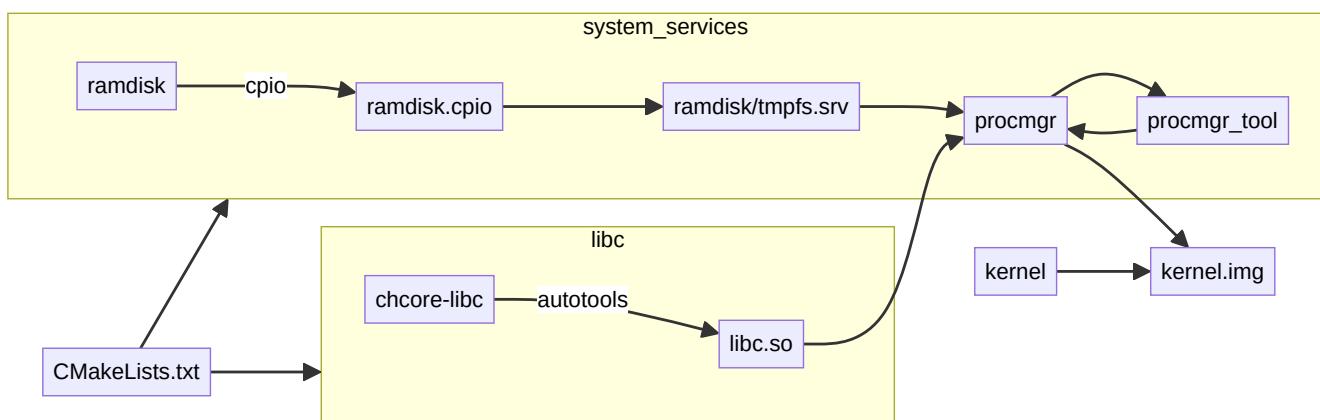
所有的对象都有一个公共基类，并定义了虚构函数列表，当引用计数归零即完全被能力组移除后内核会执行deinit代码完成销毁工作。

Note

你可以根据上述的描述来梳理根进程创建以及普通进程创建的异同，最后梳理出创建进程的标准模式。

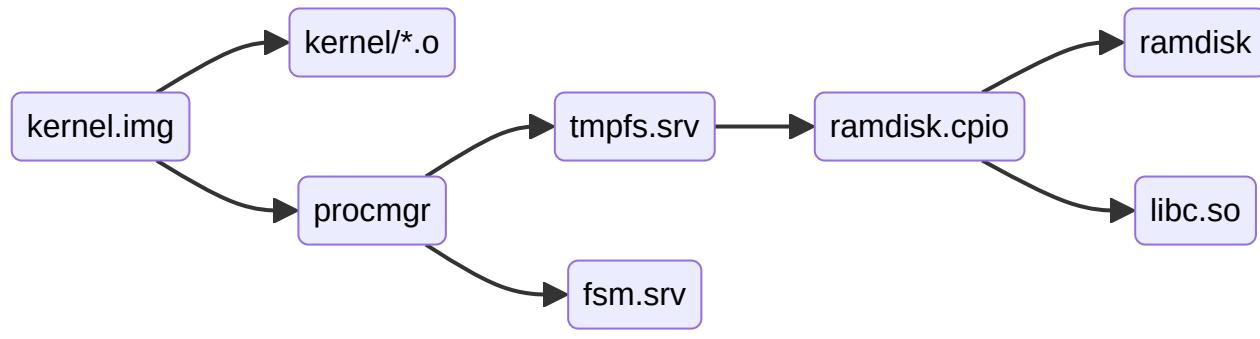
用户态构建

我们在 Lab1 的代码导读阶段说明了 kernel 目录下的代码是如何被链接成内核镜像的，我们在内核镜像链接中引入了 procmgr 这个预先构建的二进制文件。在 Lab3 中，我们引入了用户态的代码构建，所以我们将 procmgr 的依赖改为使用用户态的代码生成。下图为具体的构建规则图。



procmgr 是一个自包含的 ELF 程序，其代码在 procmgr 中列出，其主要包含一个 ELF 执行器以及作为Chcore微内核的 init 程序启动，其构建主要依赖于 fsm.srv 以及 tmpfs.srv，其中

fsm.srv 为文件系统管理器其扮演的是虚拟文件系统的角色用于桥接不同挂载点上的文件系统的实现，而 tmpfs.srv 则是 Chcore 的根文件系统其由 ramdisk 下面的所有文件以及构建好 libc.so 所打包好的 ramdisk.cpio 构成。当构建完 tmpfs.srv 后其会跟 libc.so 进行动态链接，最终 tmpfs.srv 以及 fsm.srv 会以incbin脚本的形式以二进制的方式被连接至 procmgr 的最后。在构建 procmgr 的最后一步，cmake 会调用 read_procmanager_elf_tool 将 procmgr 这个 ELF 文件的缩略信息粘贴至 procmgr 之前。此后 procmgr 也会以二进制的方式进一步嵌套进入内核镜像之后，最终会在 create_root_thread 的阶段通过其 elf 符号得以加载。最终，Chcore的Kernel镜像的拓扑结构 下



Last change: 2025-02-14, commit: [773916e](#)

线程生命周期管理

本实验的 OS 运行在 AArch64 体系结构，该体系结构采用“异常级别”这一概念定义程序执行时所拥有的特权级别。从低到高分别是 EL0、EL1、EL2 和 EL3。每个异常级别的具体含义和常见用法已在课程中讲述。

ChCore 中仅使用了其中的两个异常级别：EL0 和 EL1。其中，EL1 是内核模式，`kernel` 目录下的内核代码运行于此异常级别。EL0 是用户模式，`user` 目录下的用户库与用户程序代码运行在用户模式下。我们在之前的RTFSC中提到了，在Chcore中内核对用户态提供的所有的资源，

Lab2的内存对象，都围绕着`cap_group`以及`capability`展开。同目前所有的主流操作系统一样，ChCore 中的每个进程至少包含一个主线程，也可能有多个子线程，而每个线程则从属且仅从属于一个进程。在 ChCore 中，第一个被创建的进程是 `procmgr`，是 ChCore 核心的系统服务。本实验将以创建 `procmgr` 为例探索在 ChCore 中 何创建进程，以及成功创建第一个进程后 何实现内核态向用户态的切换。

在 ChCore 中，第一个被创建的进程是 `procmgr`，是 ChCore 核心的系统服务。本实验将以创建 `procmgr` 为例探索在 ChCore 中 何创建进程，以及成功创建第一个进程后 何实现内核态向用户态的切换。

权利组创建

创建用户程序至少需要包括创建对应的 `cap_group`、加载用户程序镜像并且切换到程序。在内核完成必要的初始化之后，内核将会跳转到创建第一个用户程序的操作中，该操作通过调用 `create_root_thread` 函数完成，本函数完成第一个用户进程的创建，其中的操作包括从 `procmgr` 镜像中读取程序信息，调用 `create_root_cap_group` 创建第一个 `cap_group` 进程，并在 `root_cap_group` 中创建第一个线程，线程加载着信息中记录的 elf 程序（实际上就是 `procmgr` 系统服务）。此外，用户程序也可以通过 `sys_create_cap_group` 系统调用创建一个全新的 `cap_group`。

练习题1

在 `kernel/object/cap_group.c` 中完善 `sys_create_cap_group`、
`create_root_cap_group` 函数。在完成填写之后，你可以通过 Cap create pretest 测试点。

Capgroup

完成 `create_root_cap_group` 函数后并通过测试后，你可以得到20分。

Hint

可以阅读 `kernel/object/capability.c` 中各个与 cap 机制相关的函数以及参考文档。

ELF加载

然而，完成 `cap_group` 的分配之后，用户程序并没有办法直接运行，因为 `cap_group` 只是一个资源集合的概念。线程才是内核中的调度执行单位，因此还需要进行线程的创建，将用户程序 ELF 的各程序段加载到内存中。(此为内核中 ELF 程序加载过程，用户态进行 ELF 程序解析可参考 `user/system-services/system-servers/procmgr/libs/libchcoreelf/libchcoreelf.c`，何加载程序可以对 `user/system-services/system-servers/procmgr/srvmgr.c` 中的 `procmgr_launch_process` 函数进行详细分析)

练习题2

在 `kernel/object/thread.c` 中完成 `create_root_thread` 函数，将用户程序 ELF 加载到刚刚创建的进程地址空间中。

Hint

- 程序头可以参考 `kernel/object/thread_env.h`。
- 内存分配操作使用 `create_pmo`，可详细阅读 `kernel/object/memory.c` 了解内存分配。
- 本练习并无测试点，请确保对 elf 文件内容读取及内存分配正确。否则有可能在后续切换至用户态程序运行时出错。

进程调度

完成用户程序的内存分配后，用户程序代码实际上已经被映射在 `root_cap_group` 的虚拟地址空间中。接下来需要对创建的线程进行初始化，以做好从内核态切换到用户态线程的准备。

练习题3

在 `kernel/arch/aarch64/sched/context.c` 中完成 `init_thread_ctx` 函数，完成线程上下文的初始化。

至此，我们完成了第一个用户进程与第一个用户线程的创建。接下来就可以从内核态向用户态进行跳转了。回到 `kernel/arch/aarch64/main.c`，在 `create_root_thread()` 完成后，分别调用了 `sched()` 与 `eret_to_thread(switch_context())`。`sched()` 的作用是进行一次调度，在此场景下我们创建的第一个线程将被选择。

`switch_context()` 函数的作用则是进行线程上下文的切换，包括`vmspace`、`fpu`、`tls`等。并且将`cpu_info`中记录的当前CPU线程上下文记录为被选择线程的上下文（完成后续实验后对此可以有更深的理解）。`switch_context()` 最终返回被选择线程的`thread_ctx`地址，即`target_thread->thread_ctx`。

`eret_to_thread` 最终调用了`kernel/arch/aarch64/irq/irq_entry.s`中的`__eret_to_thread` 函数。其接收参数为`target_thread->thread_ctx`，将`target_thread->thread_ctx` 写入`sp` 寄存器后调用了`exception_exit` 函数，`exception_exit` 最终调用`eret` 返回用户态，从而完成了从内核态向用户态的第一次切换。

注意此处因为尚未完成`exception_exit` 函数，因此无法正确切换到用户态程序，在后续完成`exception_exit` 后，可以通过`gdb` 追踪`pc` 寄存器的方式查看是否正确完成内核态向用户态的切换。

思考题4

思考内核从完成必要的初始化到第一次切换到用户态程序的过程是怎么样的？尝试描述一下调用关系。

无法继续执行

然而，目前任何一个用户程序并不能正常退出，也不能正常输出结果。这是由于程序中包括了`svc #0` 指令进行系统调用。由于此时 ChCore 尚未配置从用户模式（EL0）切换到内核模式（EL1）的相关内容，在尝试执行`svc` 指令时，ChCore 将根据目前的配置（尚未初始化，异常处理向量指向随机位置）执行位于随机位置的异常处理代码，进而导致触发错误指令异常。同样的，由于错误指令异常仍未指定处理代码的位置，对该异常进行处理会再次出发错误指令异常。ChCore 将不断重复此循环，并最终表现为 QEMU 不响应。后续的练习中将会通过正确配置异常向量表的方式，对这一问题进行修复。

Success

以上为Lab3 Part1 的所有内容，完成后你将获得40分

Last change: 2024-10-26, commit: [8f29e2d](#)

异常管理

由于 ChCore 尚未对用户模式与内核模式的切换进行配置，一旦 ChCore 进入用户模式执行就再也无法正常返回内核模式使用操作系统提供其他功能了。在这一部分中，我们将通过正确配置异常向量表的方式，为 ChCore 添加异常处理的能力。

在 AArch64 架构中，异常是指低特权级软件（ 用户程序）请求高特权软件（例 内核中的异常处理程序）采取某些措施以确保程序平稳运行的系统事件，包含**同步异常**和**异步异常**：

地址	异常类型	异常发生时处理器状态
+ 0x000	同步异常	ELx 使用 SP_EL0 作为 SP
+ 0x080	IRQ	
+ 0x100	FIQ	
+ 0x180	SError	
+ 0x200	同步异常	ELx 使用 SP_ELx 作为 SP
+ 0x280	IRQ	
+ 0x300	FIQ	
+ 0x380	SError	
+ 0x400	同步异常	EL0 运行于 AArch64 状态
+ 0x480	IRQ	
+ 0x500	FIQ	
+ 0x580	SError	
+ 0x600	同步异常	EL0 运行于 AArch32 状态
+ 0x680	IRQ	
+ 0x700	FIQ	
+ 0x780	SError	

- 同步异常：通过直接执行指令产生的异常。同步异常的来源包括同步中止（synchronous abort）和一些特殊指令。当直接执行一条指令时，若取指令或数据访问过程失败，则会产生同步中止。此外，部分指令（包括 `svc` 等）通常被用户程序用于主动制造异常以请求高特权级别软件提供服务（系统调用）。
- 异步异常：与正在执行的指令无关的异常。异步异常的来源包括普通中 IRQ、快速中断 FIQ 和系统错误 SError。IRQ 和 FIQ 是由其他与处理器连接的硬件产生的中断，系统错误

则包含多种可能的原因。本实验不涉及此部分。

发生异常后，处理器需要找到与发生的异常相对应的异常处理程序代码并执行。在 AArch64 中，存储于内存之中的异常处理程序代码被叫做异常向量（exception vector），而所有的异常向量被存储在一张异常向量表（exception vector table）中。可参考 `kernel/arch/aarch64/irq/irq_entry.s` 中的图表。

AArch64 中的每个异常级别都有其自己独立的异常向量表，其虚拟地址由该异常级别的异常向量基址寄存器（`VBAR_EL3`，`VBAR_EL2` 和 `VBAR_EL1`）决定。每个异常向量表中包含 16 个条目，每个条目里存储着发生对应异常时所需执行的异常处理程序代码。以上表格给出了每个异常向量条目的偏移量。

在 ChCore 中，仅使用了 EL0 和 EL1 两个异常级别，因此仅需要对 EL1 异常向量表进行初始化即可。在本实验中，ChCore 内除系统调用外所有的同步异常均交由 `handle_entry_c` 函数进行处理。遇到异常时，硬件将根据 ChCore 的配置执行对应的汇编代码，将异常类型和当前异常处理程序条目类型作为参数传递，对于 `sync_el1h` 类型的异常，跳转 `handle_entry_c` 使用 C 代码处理异常。对于 `irq_el1t`、`fiq_el1t`、`fiq_el1h`、`error_el1t`、`error_el1h`、`sync_el1t` 则跳转 `unexpected_handler` 处理异常。

练习题5

按照前文所述的表格填写 `kernel/arch/aarch64/irq/irq_entry.s` 中的异常向量表，并且增加对应的函数跳转操作

Success

以上为Lab2 Part2的全部内容，完成后你可以获得60分

Last change: 2024-12-05, commit: [073dba2](#)

系统调用

内核支持

系统调用是系统为用户程序提供的高特权操作接口。在本实验中，用户程序通过 `svc` 指令进入内核模式。在内核模式下，首先操作系统代码和硬件将保存用户程序的状态。操作系统根据系统调用号码执行相应的系统调用处理代码，完成系统调用的实际功能，并保存返回值。最后，操作系统和硬件将恢复用户程序的状态，将系统调用的返回值返回给用户程序，继续用户程序的执行。

通过异常进入到内核后，需要保存当前线程的各个寄存器值，以便从内核态返回用户态时进行恢复。保存工作在 `exception_enter` 中进行，恢复工作则由 `exception_exit` 完成。可以参考 `kernel/include/arch/aarch64/arch/machine/register.h` 中的寄存器结构，保存时在栈中应准备 `ARCH_EXEC_CONT_SIZE` 大小的空间。

完成保存后，需要进行内核栈切换，首先从 `TPIDR_EL1` 寄存器中读取到当前核的 `per_cpu_info`（参考 `kernel/include/arch/aarch64/arch/machine/smp.h`），从而拿到其中的 `cpu_stack` 地址。

练习题6

填写 `kernel/arch/aarch64/irq/irq_entry.S` 中的 `exception_enter` 与 `exception_exit`，实现上下文保存的功能，以及 `switch_to_cpu_stack` 内核栈切换函数。如果正确完成这一部分，可以通过 Userland 测试点。这代表着程序已经可以在用户态与内核态间进行正确切换。显示一下结果

```
Hello userland!
```

用户态libc支持

在本实验中新加入了 `libc` 文件，用户态程序可以链接其编译生成的 `libc.so`，并通过 `libc` 进行系统调用从而进行向内核态的异常切换。在实验提供的 `libc` 中，尚未实现 `printf` 的系统调用，因此用户态程序无法进行正常输出。实验接下来将对 `printf` 函数的调用链进行分析与探索。

`printf` 函数调用了 `vfprintf`，其中文件描述符参数为 `stdout`。这说明在 `vfprintf` 中将使用 `stdout` 的某些操作函数。

在 `user/chcore-libc/musl-libc/src/stdio/stdout.c` 中可以看到 `stdout` 的 `write` 操作

被定义为 `__stdout_write`，之后调用到 `__stdio_write` 函数。

最终 `printf` 函数将调用到 `chcore_stdout_write`。

思考题7

尝试描述 `printf` 何调用到 `chcore_stdout_write` 函数。

Hint

`chcore_write` 中使用了文件描述符，`stdout` 描述符的设置在 `user/chcore-libc/musl-libc/src/chcore-port/syscall_dispatcher.c` 中。

`chcore_stdout_write` 中的核心函数为 `put`，此函数的作用是向终端输出一个字符串。

练习题8:

在其中添加一行以完成系统调用，目标调用函数为内核中的 `sys_putstr`。使用 `chcore_syscallx` 函数进行系统调用。

至此，我们完成了对 `printf` 函数的分析及完善。从 `printf` 的例子我们也可以看到从通用 api 向系统相关 abi 的调用过程，并最终通过系统调用完成从用户态向内核态的异常切换。

Success

以上为Lab3 Part3的所有内容，完成后你可以获得80分

Last change: 2024-09-30, commit: [870cb79](#)

用户程序编写

我们完成了内核态向用户态的切换，以及用户态向内核态的异常切换。同时，我们拥有了一个完整的 `libc`，可以帮助我们进行系统调用。接下来，我们将尝试使用 ChCore 的 `libc` 及编译器进行简单的程序编译，并将其加载到内核镜像中运行。

练习题9

尝试编写一个简单的用户程序，其作用至少包括打印以下字符(测试将以此为得分点)。

```
Hello ChCore!
```

使用 `chcore-libc` 的编译器进行对其进行编译，编译输出文件名命名为 `hello_world.bin`，并将其放入 `ramdisk` 加载进内核运行。内核启动时将自动运行文件名为 `hello_world.bin` 的可执行文件。

Hint

- ChCore 的编译工具链在 `build/chcore-libc/bin` 文件夹中。
- 使用 `cmake` 进行编译，可以将工具链文件指定为 `build/toolchain.cmake`，将默认使用 ChCore 编译工具链。

到这里，你的程序应该可以通过所有的测试点并且获得满分。你可以编写一些更复杂的程序并尝试放入 ChCore 中运行。

Last change: 2025-02-14, commit: [773916e](#)

Lab 4：多核调度与IPC

在本实验中，我们将逐步实现ChCore的多核支持以及微内核系统的核心：进程间通信，本Lab包含四个部分：

1. 多核启动支持: 使ChCore通过树莓派厂商所提供的固件唤醒多核执行
2. 多核调度: 使ChCore实现在多核上进行round-robin调度。
3. IPC: 使ChCore支持进程间通信
4. IPC调优: 为ChCore的IPC针对测试的特点进行调优。

跟先前的Lab相同，本实验代码包含了基础的 ChCore 操作系统镜像，除了练习题相关部分的源码以外（指明需要阅读的代码），其余部分通过二进制格式提供。在正确完成本实验的练习题之后，你可以在树莓派3B+QEMU或开发板上进入 ChCore shell。注释 `/* LAB 4 TODO BEGIN (exercise #) */` 和 `/* LAB 4 TODO END (exercise #) */` 之间代表需要填空的代码部分。

Last change: 2024-10-21, commit: [8a420dc](#)

多核支持

Note

本部分实验没有代码题，仅有思考题。

• 启动多核

为了让ChCore支持多核，我们需要考虑一下问题：

- 何启动多核，让每个核心执行初始化代码并开始执行用户代码？
- 何区分不同核心在内核中保存的数据结构（比如状态，配置，内核对象等）？
- 何保证内核中对象并发正确性，确保不会由于多个核心同时访问内核对象导致竞争条件？

在启动多核之前，我们先介绍ChCore 何解决第二个问题。ChCore对于内核中需要每个CPU核心单独存一份的内核对象，都根据核心数量创建了多份（即利用一个数组来保存）。ChCore支持的核心数量为PLAT_CPU_NUM（该宏定义在 `kernel/common/machine.h` 中，其代表可用CPU核心的数量，根据具体平台而异）。比，实验使用的树莓派3平台拥有4个核心，因此该宏定义的值为4。ChCore会将CPU核心的核心ID作为数组的索引，在数组中取出对应的CPU核心本地的数据。为了方便确定当前执行该代码的CPU核心ID，我们在 `kernel/arch/aarch64/machine/smp.c` 中提供了 `smp_get_cpu_id` 函数。该函数通过访问系统寄存器 `tpidr_el1` 来获取调用它的CPU核心的ID，该ID可用作访问上述数组的索引。

```
#include <common/vars.h>

/* raspi3 config */
#define PLAT_CPU_NUM      4
#define PLAT_RASPI3
```

启动多核

在实验1中我们已经介绍，在QEMU模拟的树莓派中，所有CPU核心在开机时会被同时启动。在引导时这些核心会被分为两种类型。一个指定的CPU核心会引导整个操作系统和初始化自身，被称为CPU主核（primary CPU）。其他的CPU核心只初始化自身即可，被称为CPU从核（backup CPU）。CPU核心仅在系统引导时有所区分，在其他阶段，每个CPU核心都是被相同对待的。

思考题 1

阅读 Lab1 中的汇编代码 `kernel/arch/aarch64/boot/raspi3/init/start.S`。说明 ChCore 是何选定主CPU，并阻塞其他其他CPU的执行的。

然而在树莓派真机中，从还需要主C核手动指定每一个CPU核心的启动地址。这些CPU核心会读取固定地址的上填写的启动地址，并跳转到该地址启动。在 `kernel/arch/aarch64/boot/raspi3/init/init_c.c` 中，我们提供了 `wakeup_other_cores` 函数用于实现该功能，并让所有的CPU核心同在QEMU一样开始执行 `_start` 函数。

与之前的实验一样，主CPU在第一次返回用户态之前会在 `kernel/arch/aarch64/main.c` 中执行 `main` 函数，进行操作系统的初始化任务。在本小节中，ChCore 将执行 `enable_smp_cores` 函数激活各个其他CPU。

思考题 2

阅读汇编代码 `kernel/arch/aarch64/boot/raspi3/init/start.S`, `init_c.c` 以及 `kernel/arch/aarch64/main.c`，解释用于阻塞其他CPU核心的 `secondary_boot_flag` 是物理地址还是虚拟地址？是何传入函数 `enable_smp_cores` 中，又是何赋值的（考虑虚拟地址/物理地址）？

Success

以上为Lab4 part1 的所有内容

Last change: 2025-02-14, commit: [773916e](#)

多核调度

- 调度队列初始化
- 调度队列入队
- 调度队列出队
- 协作式调度
- 抢占式调度
 - 物理时钟初始化
 - 物理时钟中断与抢占

ChCore已经可以启动多核，但仍然无法对多个线程进行调度。本部分将首先实现协作式调度，从而允许当前在CPU核心上运行的线程主动退出或主动放弃CPU时，CPU核心能够切换到另一个线程继续执行。其后，我们将驱动树莓派上的物理定时器，使其以一定的频率发起中断，使得内核可以在一定时间片后重新获得对CPU核心的控制，并基于此进一步实现抢占式调度。

ChCore中与调度相关的函数与数据结构定义在 `kernel/include/sched/sched.h` 中。

```
struct sched_ops {
    int (*sched_init)(void);
    int (*sched)(void);
    int (*sched_periodic)(void);
    int (*sched_enqueue)(struct thread *thread);
    int (*sched_dequeue)(struct thread *thread);
    /* Debug tools */
    void (*sched_top)(void);
};
```

`sched_ops` 是用于抽象 ChCore 中调度器的一系列操作。它存储指向不同调度操作的函数指针，以支持不同的调度策略。`cur_sched_ops` 则是一个 `sched_ops` 的实例，其在内核初始化过程中（`main` 函数）调用 `sched_init` 进行初始化。ChCore 用在 `kernel/include/sched/sched.h` 中定义的静态函数封装对 `cur_sched_ops` 的调用。`sched_ops` 中定义的调度器操作 下所示：

- `sched_init`: 初始化调度器。
- `sched`: 进行一次调度。即将正在运行的线程放回就绪队列，然后在就绪队列中选择下一个需要执行的线程返回。
- `sched_enqueue`: 将新线程添加到调度器的就绪队列中。
- `sched_dequeue`: 从调度器的就绪队列中取出一个线程。
- `sched_top`: 用于 debug, 打印当前所有核心上的运行线程以及等待线程的函数。

在本部分将实现一个基本的 Round Robin (时间片轮转) 调度器，该程序调度在同一 CPU 核心上运行的线程，因此内核初始化过程调用 `sched_init` 时传入了 `&rr` 作为参数。该调度器的调度操作（即对于 `sched_ops` 定义的各个函数接口的实现）实现在 `kernel/sched/policy_rr.c` 中，这里简要介绍其涉及的数据结构：

`current_threads` 是一个数组，分别指向每个 CPU 核心上运行的线程。而 `current_thread` 则利用 `smp_get_cpu_id` 获取当前运行核心的 id，从而找到当前核心上运行的线程。

`struct queue_meta` 定义了 round robin 调度器使用的就绪队列，其中 `queue_head` 字段是连接该就绪队列上所有等待线程的队列，`queue_len` 字段是目前该就绪队列的长度，`queue_lock` 字段是用于保证该队列并发安全的锁。`kernel/sched/policy_rr.c` 定义了一个全局变量 `rr_ready_queue_meta`，该变量是一个 `struct queue_meta` 类型的数组，数组大小由 `PLAT_CPU_NUM` 定义，即代表每个 CPU 核心都具有一个就绪队列。运行的 CPU 核心可以通过 `smp_get_cpu_id` 获取当前运行核心的 id，从而在该数组中找到当前核心对应的就绪队列。

调度队列初始化

内核初始化过程中会调用 `sched_init` 初始化调度相关的元数据，`sched_init` 定义在 `kernel/sched/sched.c` 中，该函数首先初始化 `idle_thread`（每个 CPU 核心拥有一个 `idle_thread`，当调度器的就绪队列中没有等待线程时会切换到 `idle_thread` 运行），然后会初始化 `current_threads` 数组，最后调用 `struct sched_ops rr` 中定义的 `sched_init` 函数，即 `rr_sched_init`。

练习题 1

在 `kernel/sched/policy_rr.c` 中完善 `rr_sched_init` 函数，对 `rr_ready_queue_meta` 进行初始化。在完成填写之后，你可以看到输出“Scheduler metadata is successfully initialized!”并通过 Scheduler metadata initialization 测试点。

Tip

`sched_init` 只会在主 CPU 初始化时调用，因此 `rr_sched_init` 需要对每个 CPU 核心的就绪队列都进行初始化。

调度队列入队

内核初始化过程结束之后会调用 `create_root_thread` 来创建第一个用户态进程及线程，在 `create_root_thread` 最后会调用 `sched_enqueue` 函数将创建的线程加入调度队列之中。`sched_enqueue` 最终会调用 `kernel/sched/policy_rr.c` 中定义的 `rr_sched_enqueue` 函数。该函数首先挑选合适的 CPU 核心的就绪队列（考虑线程是否绑核以及各个 CPU 核心之间的负载均衡），然后调用 `_rr_sched_enqueue` 将线程插入到选中的就绪队列中。

练习 2

在 `kernel/sched/policy_rr.c` 中完善 `_rr_sched_enqueue` 函数，将 `thread` 插入到 `cpuid` 对应的就绪队列中。

Success

在完成填写之后，你可以看到输出“Successfully enqueue root thread”并通过 Schedule Enqueue 测试点。

调度队列出队

内核初始化过程结束并调用 `create_root_thread` 创建好第一个用户态进程及线程之后，在第一次进入用户态之前，会调用 `sched` 函数来挑选要返回到用户态运行的线程（虽然此时就绪队列中只有root thread一个线程）。`sched` 最终会调用 `kernel/sched/policy_rr.c` 中定义的 `rr_sched` 函数。该调度函数的操作非常直观，就是将现在正在运行的线程重新加入调度器的就绪队列当中，并从就绪队列中挑选出一个新的线程运行。由于内核刚刚完成初始化，我们还没有设置过 `current_thread`，所以 `rr_sched` 函数中 `old` 为 `NULL`，后面的练习中我们会考虑 `old` 不为 `NULL` 的情况。紧接着 `rr_sched` 会调用 `rr_sched_choose_thread` 函数挑选出下一个运行的线程，并切换到该线程。

`rr_sched_choose_thread` 内部会调用 `findRunnableThread` 从当前CPU核心的就绪队列中选取一个可以运行的线程并调用 `_rr_sched_dequeue` 将其从就绪队列中移除。

练习 3

在 `kernel/sched/sched.c` 中完善 `findRunnableThread` 函数，在就绪队列中找到第一个满足运行条件的线程并返回。在 `kernel/sched/policy_rr.c` 中完善 `_rr_sched_dequeue` 函数，将被选中的线程从就绪队列中移除。

Success

在完成填写之后，运行 ChCore 将可以成功进入用户态，你可以看到输出“Enter Procmgr Root thread (userspace)”并通过 Schedule Enqueue 测试点。

协作式调度

顾名思义，协作式调度需要线程主动放弃CPU。为了实现该功能，我们提供了 `sys_yield` 这一个系统调用(syscall)。该syscall可以主动放弃当前CPU核心，并调用上述的 `sched` 接口完成调度器的调度工作。`kernel/sched/policy_rr.c` 中定义的 `rr_sched` 函数中，如果当前运行线程的状态为 `TS_RUNNING`，即还处于可以运行的状态，我们应该将其重新加入到就绪队列当中，这样该线程在之后才可以被再次调度执行。

练习 4

在 `kernel/sched/sched.c` 中完善系统调用 `sys_yield`，使用户态程序可以主动让出CPU核心触发线程调度。此外，请在 `kernel/sched/policy_rr.c` 中完善 `rr_sched` 函数，将当前运行的线程重新加入调度队列中。

Success

在完成填写之后，运行 ChCore 将可以成功进入用户态并创建两个线程交替执行，你可以看到输出“Cooperative Schedluing Test Done!”并通过 Cooperative Schedluing 测试点。

抢占式调度

使用刚刚实现的协作式调度器，ChCore能够在线程主动调用 `sys_yield` 系统调用让出CPU核心的情况下调度线程。然而，若用户线程不想放弃对CPU核心的占据，内核便只能让用户线程继续执行，而无法强制用户线程中止。因此，在这一部分中，本实验将实现抢占式调度，以帮助内核定期重新获得对CPU核心的控制权。

ChCore启动的第一个用户态线程（执行 `user/system-services/system-servers/procmgr/procmgr.c` 的 `main` 函数）将创建一个“自旋线程”，该线程在获得CPU核心的控制权后便会执行无限循环，进而导致无论是该程序的主线程还是ChCore内核都无法重新获得CPU核心的控制权。就保护系统免受用户程序中的错误或恶意代码影响而言，这一情况显然并不理想，任何用户应用线程均可以 该“自旋线程”一样，通过进入无限循环来永久“霸占”整个CPU核心。

为了处理“自旋线程”的问题，ChCore内核必须具有强行中断一个正在运行的线程并夺回对CPU核心的控制权的能力，为此我们必须扩展ChCore以支持处理来自物理时钟的外部硬件中断。

物理时钟初始化

本部分我们将通过配置ARM提供的Generic Timer来使能物理时钟并使其以固定的频率发起中断。我们需要处理的系统寄存器 下([Refer](#)):

- CNTPCT_ELO: 它的值代表了当前的 system count。
- CNTFRQ_ELO: 它的值代表了物理时钟运行的频率，即每秒钟 system count 会增加多少。
- CNTP_CVAL_ELO: 是一个64位寄存器，操作系统可以向该寄存器写入一个值，当 system count 达到或超过该值时，物理时钟会触发中断。
- CNTP_TVAL_ELO: 是一个32位寄存器，操作系统可以写入 TVAL，处理器会在内部读取当前的系统计数，加上写入的值，然后填充 CVAL。
- CNTP_CTL_ELO: 物理时钟的控制寄存器，第0位ENABLE控制时钟是否开启，1代表 enable，0代表disable；第1位IMASK代表是否屏蔽时钟中断，0代表不屏蔽，1代表屏蔽。

对物理时钟进行初始化的代码位于 `kernel/arch/aarch64/plat/rasp3/irq/timer.c` 的 `plat_timer_init` 函数。

练习 5

请根据代码中的注释在 `kernel/arch/aarch64/plat/raspi3/irq/timer.c` 中完善 `plat_timer_init` 函数，初始化物理时钟。需要完成的步骤有：

- 读取 CNTFRQ_EL0 寄存器，为全局变量 `cntp_freq` 赋值。
- 根据 `TICK_MS`（由ChCore决定的时钟中断的时间间隔，以ms为单位，ChCore默认每10ms触发一次时钟中断）和 `cntpfrq_el0`（即物理时钟的频率）计算每两次时钟中断之间 `system count` 的增长量，将其赋值给 `cntp_tval` 全局变量，并将 `cntp_tval` 写入 `CNTP_TVAL_EL0` 寄存器！
- 根据上述说明配置控制寄存器 `CNTP_CTL_EL0`。

Hint

由于启用了时钟中断，但目前还没有对中断进行处理，所以会影响评分脚本的评分，你可以通过运行ChCore观察是否有 "[TEST] Physical Timer was successfully initialized!: OK" 输出来判断是否正确对物理时钟进行初始化。

物理时钟中断与抢占

我们在lab3中已经为ChCore配置过异常向量表（`kernel/arch/aarch64/irq/irq_entry.s`），当收到来自物理时钟的外部中断时，内核会进入 `handle_irq` 中断处理函数，该函数会调用平台相关的 `plat_handle_irq` 来进行中断处理。`plat_handle_irq` 内部果判断中断源为物理时钟，则调用 `handle_timer_irq`。

ChCore记录每个线程所拥有的时间片（`thread->thread_ctx->sc->budget`），为了能够让线程之间轮转运行，我们应当在处理时钟中断时递减当前运行线程的时间片，并在当前运行线程的时间片耗尽时进行调度，选取新的线程运行。

练习 6

请在 `kernel/arch/aarch64/plat/raspi3/irq/irq.c` 中完善 `plat_handle_irq` 函数，当中断号 `irq` 为 `INT_SRC_TIMER1`（代表中断源为物理时钟）时调用 `handle_timer_irq` 并返回。请在 `kernel/irq/timer.c` 中完善 `handle_timer_irq` 函数，递减当前运行线程的时间片 `budget`，并调用 `sched` 函数触发调度。请在 `kernel/sched/policy_rr.c` 中完善 `rr_sched` 函数，在将当前运行线程重新加入就绪队列之前，恢复其调度时间片 `budget` 为 `DEFAULT_BUDGET`。

Success

在完成填写之后，运行 ChCore 将可以成功进入用户态并打断创建的“自旋线程”让内核和主线程可以拿回CPU核心的控制权，你可以看到输出 "Hello, I am thread 3. I'm

spinning." 和 “Thread 1 successfully regains the control!” 并通过 Preemptive Scheduling 测试点。

Success

以上为Lab4 Part2的所有内容

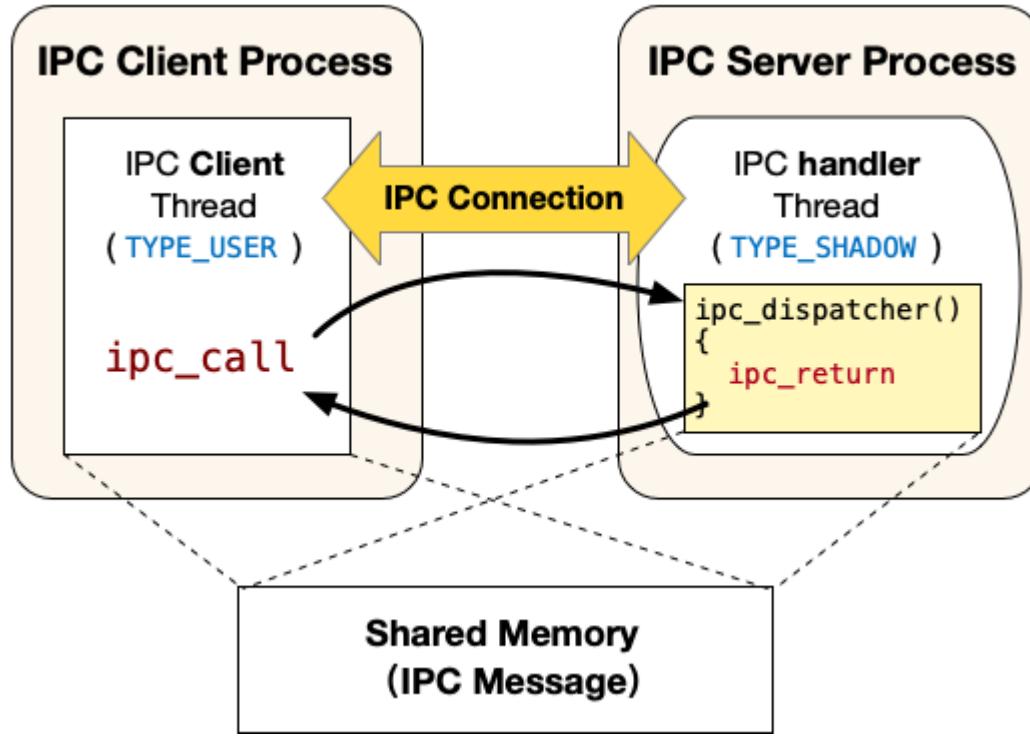
Last change: 2025-02-14, commit: [773916e](#)

进程间通信 (IPC)

- 进程间通讯概览
- 具体流程

在本部分，我们将实现ChCore的进程间通信，从而允许跨地址空间的两个进程可以使用IPC进行信息交换。

进程间通讯概览



ChCore的IPC接口不是传统的send/recv接口。其更像客户端/服务器模型，其中IPC请求接收者是服务器，而IPC请求发送者是客户端。服务器进程中包含三类线程：

- **主线程：**该线程与普通的线程一样，类型为 `TYPE_USER`。该线程会调用 `ipc_register_server` 将自己声明为一个IPC的服务器进程，调用的时候会提供两个参数：服务连接请求的函数 `client_register_handler` 和服务真正IPC请求的函数 `server_handler`（即图中的 `ipc_dispatcher`），调用该函数会创建一个注册回调线程；
- **注册回调线程：**该线程的入口函数为上文提到的 `client_register_handler`，类型为 `TYPE_REGISTER`。正常情况下该线程不会被调度执行，仅当有Client发起建立IPC连接的请求时，该线程运行并执行 `client_register_handler`，为请求建立连接的Client创建一个服务线程（即图中的IPC handler thread）并在服务器进程的虚拟地址空间中分配一个可以用来映射共享内存的虚拟地址。
- **服务线程：**当Client发起建立IPC连接请求时由注册回调线程创建，入口函数为上文提到的

server_handler，类型为 TYPE_SHADOW。正常情况下该线程不会被调度执行，仅当有 Client 端线程使用 ipc_call 发起IPC请求时，该线程运行并执行server_handler（即图中的 ipc_dispatcher），执行结束之后会调用 ipc_return 回到Client端发起IPC请求的线程。

注意

注册回调线程和服务线程都不再拥有调度上下文（Scheduling Context），也即不会主动被调度器调度到。其在客户端申请建立IPC连接或者发起IPC请求的时候才会被调度执行。为了实现该功能，这两种类型的线程会继承IPC客户端线程的调度上下文（即调度时间片 budget），从而能被调度器正确地调度。

具体流程

为了实现ChCore IPC的功能，首先需要在Client与Server端创建起一个一对一的IPC Connection。该Connection保存了IPC Server的服务线程（即上图中IPC handler Thread）、Client与Server的共享内存（用于存放IPC通信的内容）。同一时刻，一个Connection只能有一个Client接入，并使用该Connection切换到Server的处理流程。ChCore提供了一系列机制，用于创建Connection以及创建每个Connection对应的服务线程。下面将以具体的IPC注册到调用的流程，详细介绍ChCore的IPC机制：

1. IPC服务器的主线程调用: `ipc_register_server` (`user/chcore-libc/musl-libc/src/chcore-port/ipc.c` 中) 来声明自己为IPC的服务器端。
 - 参数包括server_handler和client_register_handler，其中server_handler为服务端用于提供服务的回调函数（比 上图中IPC handler Thread的入口函数 `ipc_dispatcher`）；client_register_handler为服务端提供的用于注册的回调函数，该函数会创建一个注册回调线程。
 - 随后调用ChCore提供的的系统调用： `sys_register_server`。该系统调用实现在 `kernel/ipc/connection.c` 当中，该系统调用会分配并初始化一个 `struct ipc_server_config` 和一个 `struct ipc_server_register_cb_config`。之后将调用者线程（即主线程）的 `general_ipc_config` 字段设置为创建的 `struct ipc_server_config`，其中记录了注册回调线程和IPC服务线程的入口函数（即图中的 `ipc_dispatcher`）。将注册回调线程的 `general_ipc_config` 字段设置为创建的 `struct ipc_server_register_cb_config`，其中记录了注册回调线程的入口函数和用户态栈地址等信息。
2. IPC客户端线程调用 `ipc_register_client` （定义在 `user/chcore-libc/musl-libc/src/chcore-port/ipc.c` 中） 来申请建立IPC连接。
 - 该函数仅有一个参数，即IPC服务器的主线程在客户端进程cap_group中的 capability。该函数会首先通过系统调用申请一块物理内存作为和服务器的共享内存

(即图中的Shared Memory)。

- 随后调用 `sys_register_client` 系统调用。该系统调用实现在 `kernel/ipc/connection.c` 当中，该系统调用会将刚才申请的物理内存映射到客户端的虚拟地址空间中，然后调用 `create_connection` 创建并初始化一个 `struct ipc_connection` 类型的内核对象，该内核对象中的 `shm` 字段会记录共享内存相关的信息（包括大小，分别在客户端进程和服务器进程当中的虚拟地址和 capability）。
- 之后会设置注册回调线程的栈地址、入口地址和第一个参数，并切换到注册回调线程运行。

3. 注册回调线程运行的入口函数为主线程调用 `ipc_register_server` 是提供的 `client_register_handler` 参数，一般会使用默认的 `DEFAULT_CLIENT_REGISTER_HANDLER` 宏定义的入口函数，即定义在 `user/chcore-libc/musl-libc/src/chcore-port/ipc.c` 中的 `register_cb`。

- 该函数首先分配一个用来映射共享内存的虚拟地址，随后创建一个服务线程。
- 随后调用 `sys_ipc_register_cb_return` 系统调用进入内核，该系统调用将共享内存映射到刚才分配的虚拟地址上，补全 `struct ipc_connection` 内核对象中的一些元数据之后切换回客户端线程继续运行，客户端线程从 `ipc_register_client` 返回，完成IPC建立连接的过程。

4. IPC客户端线程调用 `ipc_create_msg` 和 `ipc_set_msg_data` 向IPC共享内存中填充数据，然后调用 `ipc_call` (`user/chcore-libc/musl-libc/src/chcore-port/ipc.c` 中) 发起IPC请求。

- `ipc_call` 中会发起 `sys_ipc_call` 系统调用（定义在 `kernel/ipc/connection.c` 中），该系统调用将设置服务器端的服务线程的栈地址、入口地址、各个参数，然后迁移到该服务器端服务线程继续运行。由于当前的客户端线程需要等待服务器端的服务线程处理完毕，因此需要更新其状态为 `TS_WAITING`，且不要加入等待队列。

5. IPC服务器端的服务线程在处理完IPC请求之后使用 `ipc_return` 返回。

- `ipc_return` 会发起 `sys_ipc_return` 系统调用，该系统调用会迁移到IPC客户端线程继续运行，IPC客户端线程从 `ipc_call` 中返回。

练习题 7

在 `user/chcore-libc/musl-libc/src/chcore-port/ipc.c` 与 `kernel/ipc/connection.c` 中实现了大多数IPC相关的代码，请根据注释补全 `kernel/ipc/connection.c` 中的代码。之后运行ChCore可以看到 “[TEST] Test IPC finished!” 输出，你可以通过 Test IPC 测试点。

Warning

由于用户文件系统管理器(FSM)与服务管理器(Procmgr)需要通过IPC来进行数据传输，如果IPC链路实现错误则无法运行 `test_ipc.bin` 以及 `Chcore_Shell`

Hint

由于你已经在Lab3中了解到Printf的实现是系统调用并不经过IPC，所以你可以在所有的暴露代码(IPC链路上)中插入Printf/Printk指令进行Debug打印。

Success

以上为Lab4 Part3的所有内容

Last change: 2025-02-14, commit: [773916e](#)

实机运行与IPC性能优化

在本部分，你需要对IPC的性能进行优化。为此，你首先需要在树莓派3B实机上运行ChCore。

练习题 8

请在树莓派3B上运行ChCore，并确保此前实现的所有功能都能正确运行。

在ChCore启动并通过测试后，在命令行运行

```
./test_ipc_perf.bin
```

你会得到以下输出结果

```
[TEST] test ipc with 32 threads, time: xxx cycles
[TEST] test ipc with send cap, loop: 100, time: xxx cycles
[TEST] test ipc with send cap and return cap, loop: 100, time: xxx cycles
[TEST] Test IPC Perf finished!
```

练习题 9

尝试优化在第三部分实现的IPC的性能，降低test_ipc_perf.bin的三个测试所消耗的cycle数

IPC性能测试程序的测试用例包括：

1. 创建多个线程发起IPC请求（不传递cap），Server收到IPC后直接返回。记录从创建线程到所有线程运行结束的时间。
2. Client创建多个PMO对象，并发起IPC请求（传递PMO）；Server收到IPC后读取PMO，并依据读出的值算出结果，将结果写回随IPC传递的PMO中并返回；Client在IPC返回后读取PMO中的结果。将上述过程循环多次并记录运行时间。
3. Client创建多个PMO对象，并发起IPC请求（传递PMO）；Server收到IPC后读取PMO，并依据读出的值算出结果，然后创建新的PMO对象，将结果写入新创建的PMO中，并通过`ipc_return_with_cap`返回；Client在IPC返回后读取返回的PMO中的结果。将上述过程循环多次并记录运行时间。

在测试能够顺利通过的前提下，你可以修改任意代码。（测试程序所调用的函数位于`user/chcore-libc/libchcore/porting/overrides/src/chcore-port/ipc.c`）

Hint

我们所有的任务都要求多次创建ipc链接并进行操作，你需要具体理解ipc链接的创建过程并根据测试的单独场景进行优化。

Success

以上为Lab4 的所有内容

Last change: 2025-02-14, commit: [773916e](#)

Lab 5：虚拟文件系统

虚拟文件系统（Virtual File System, VFS）提供了一个抽象层，使得不同类型的文件系统可以在应用程序层面以统一的方式进行访问。这个抽象层隐藏了不同文件系统之间的差异，使得应用程序和系统内核可以以一致的方式访问各种不同类型的文件系统，ext4、tmpfs、FAT32等。在 ChCore 中，我们通过 FSM 系统服务以及 FS_Base 文件系统 wrapper 将不同的文件系统整合起来，给运行在 ChCore 上的应用提供了统一的抽象。

本Lab一共分为四个部分：

1. [Posix适配](#): 分析ChCore是如何实现兼容posix的文件接口的。
2. [FSM](#): FSM是ChCore的虚拟文件系统的实现层，其主要负责页缓存，挂载点管理，以及路径对接。我们在此部分实现这一文件系统转发层。
3. [FS_Base](#): FS_Base是文件系统实现层，由于在微内核系统中文件系统实际由一个个进程实现，所以我们统一包装标准的文件操作到通用库即为FS_Base，我们需要在这一个部分实现它。
4. [Poweraccess](#): FS_Page_Fault是文件系统PageFault的实现，本Lab需要同学们根据一个具体的任务使用页预取出来优化PageFault的触发。

跟先前的Lab相同，本实验代码包含了基础的 ChCore 操作系统镜像，除了练习题相关部分的源码以外（指明需要阅读的代码），其余部分通过二进制格式提供。在正确完成本实验的练习题之后，你可以在树莓派3B+QEMU或开发板上进入 ChCore shell。与之前的Lab不同的地方是，本Lab不涉及任何内核态的代码编写，你需要将所有的目光聚焦在 user 这个目录下面的文件。

注释 /* LAB 5 TODO BEGIN (exercise #) */ 和 /* LAB 5 TODO END (exercise #) */ 之间代表需要填空的代码部分。

Last change: 2025-05-14, commit: [82120fd](#)

Posix 适配

无论我们采用的是什么样的操作系统，如果希望能够对上用户态的程序的话，我们都希望其采用同一套的调用规范。相同的在我们开发用户态程序的时候，我们也希望下层的libc提供的接口保持一致，以便于开发者进行移植。而在现代操作系统中 Posix 是一个非常重要的规范。我们都可以在Windows, MacOS, Linux以及其他衍生系统上找到它的身影，Posix 针对文件系统提出了一系列的API规范。下面是一个简要的描述

1. mount, umount API：用于文件系统的挂载以及
2. open, close：用于打开以及关闭文件描述符
3. write, read：用于文件的读写
4. mkdir, rmdir, creat, unlink, link, symlink：用于文件以及目录的创建与删除
5. fcntl (byte range locks, etc.)：用于修改文件描述符的具体属性
6. stat, utimes, chmod, chown, chgrp：用于修改文件的属性
7. 所有的文件路径都以'/'开始

例 当我们在Linux系统中使用 `strace` 去追踪 `cat` 指令的系统调用时我们可以得到 下的系统调用序

```
openat(AT_FDCWD, "foo", O_RDONLY)      = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=4, ...}) = 0
fadvise64(3, 0, 0, POSIX_FADV_SEQUENTIAL) = 0
mmap(NULL, 139264, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7cb8b6fd3000
read(3, "foo\n", 131072)              = 4
write(1, "foo\n", 4)
)                                     = 4
read(3, "", 131072)                  = 0
```

撇去一些无关紧要的系统调用后，我们可以看到其首先调用了 `openat` 这个系统指令，其负责打开一个系统路径下的文件，并返回其一个文件描述符号。

练习题 1

阅读 `user/chcore-libc/libchcore/porting/overrides/src/chcore-port/file.c` 的 `chcore_openat` 函数，分析 ChCore 是如何处理 `openat` 系统调用的，关注 IPC 的调用过程以及 IPC 请求的内容。

Lab5 的所有代码都运行在用户态，不同应用间通过 IPC 进行通信，可调用的 IPC 相关函数定义在 `user/chcore-libc/libchcore/porting/overrides/include/chcore/ipc.h`。

果你感兴趣的话，你也可以继续阅读 `read` 以及 `write` 这些文件系统调用的实现，来看看 `chcore-libc` 是怎么将Chcore文件系统的实现对齐在Posix的API之上的。

Success

以上为Lab5 Part1的所有内容

Last change: 2025-04-15, commit: [90a02e2](#)

FSM

只要实现了 FSBase 和 FSWrapper 的接口的 IPC 服务，都可以成为一个文件系统示例。FSM 负责管理文件系统，为用户态建立文件系统连接并创建 IPC 的客户端，由于文件系统与其挂载点密切相关，所以 FSM 会处理以下类型的请求：

```
/* Client send fsm_req to FSM */
enum fsm_req_type {
    FSM_REQ_UNDEFINED = 0,
    FSM_REQ_PARSE_PATH,
    FSM_REQ_MOUNT,
    FSM_REQ_UMOUNT,
    FSM_REQ_SYNC,
};
```

当 FSM 收到 Client 的 FSM_REQ_MOUNT 类型的请求时，其会执行挂载文件系统的操作，**增加挂载的文件系统数量**，创建对应的 `mount_info_node` 添加到挂载信息表中，直至最后**与文件系统建立 IPC 连接**，并将创建完的 IPC 客户端保存在挂载信息中，总而言之，FSM 仅负责挂载和文件系统同步有关的工作，剩下的其他功能由每个具体的FS服务进行处理。

```
struct mount_point_info_node {
    cap_t fs_cap;
    char path[MAX_MOUNT_POINT_LEN + 1];
    int path_len;
    ipc_struct_t *_fs_ipc_struct; // fs_client
    int refcnt;
    struct list_head node;
};
```

练习题 2

实现 `user/system-services/system-servers/fsm/fsm.c` 的 `fsm_mount_fs` 函数。

提示：

你应当回顾Lab4的代码以查看ChCore是怎么基于IPC服务的cap来创建并维护连接的。

当 FSM 收到 Client 的 FSM_REQ_PARSE_PATH 类型的请求时，其首先会尝试解析 IPC 请求中访问文件的路径，通过遍历挂载信息链表，找到对应的最匹配的文件系统以及其挂载点路径。通过匹配的文件系统，获取到该文件系统的客户端 cap。如果 Client 已经获取到了文件系统的 cap，则直接返回解析后的 挂载点路径；否则 FSM 会把挂载路径以及其对应的文件系统的 cap 也一并返回给 Client，并记录该 Client 已获取的文件系统 cap 的信息（FSM 会记录所有已经发送给某个 Client 的文件系统的 cap，见 `user/system-services/system-servers/fsm/`

fsm_client_cap.h) 。

练习题 3

实现 user/system-services/system-servers/fsm/fsm.c 的 IPC 请求处理函数。

提示：

- 完成 user/system-services/system-servers/fsm/fsm_client_cap.c 中的相关函数。
- 所有关于挂载点有关的helper函数都在 user/system-services/system-servers/fsm/mount_info.c
- IPC handler 返回的 IPC msg 的数据类型为 struct fsm_request，其有关的含义在 user/chcore-libc/libchcore/porting/overrides/include/chcore-internal/fs_defs.h 有详细的解释。
- 使用 user/system-services/system-servers/fsm/mount_info.h 定义的函数来帮助你实现 IPC handler。
- 你应当回顾 Lab4 代码以查看 ChCore 是怎么将 cap 对象在进程间收发的，以及 ChCore 中是怎么使用共享内存完成 IPC 调用的。
- 由于 printf 并不经过FS所以你可以放心使用。

Hint

我们提供了所有需要实现的文件的 Obj 版本，你可以修改 CMakeLists.txt，将编译所需的源文件从未实现的 C 文件替换为包含了正确实现的 Obj 文件，以此验证某一部分练习的正确性。果你需要调试某一个部分，你可以将 Scripts/extras/lab5/cmake/ 下的 CMakeLists 对应复制到 FSM 以及 FS_Base 的目录下覆盖并重新编译，运行 make qemu 后你就可以查看到 printf 的调试信息。

Part1: fsm-full.txt/fs_base-part1.txt

Part2: fsm-part2.txt/fs_base-part2-vnode.txt

Part3: fsm-part2.txt/fs_base-part2-server_entry.txt

Part4: 原来的CMakeLists.txt

Success

以上为Lab5 Part2的所有内容

执行 make grade，可以得到 Scores: 20/100。

Last change: 2025-02-14, commit: [773916e](#)

FS_Base

- vnode
- server_entry
- fs_wrapper_ops

在 ChCore 中，FS_Base 是文件系统的一层 wrapper，IPC 请求首先被 FS_Base 接收，再由 FS_Base 调用实际的文件系统进行处理。

vnode

在 FS_Base wrapper 中，ChCore 实现了 vnode 抽象，为文件系统中的对象（文件、目录、符号链接等）提供一个统一的表示方式。

ChCore 中 vnode 的定义为：

```
struct fs_vnode {  
    ino_t vnode_id; /* identifier */  
    struct rb_node node; /* rbtree node */  
  
    enum fs_vnode_type type; /* regular or directory */  
    int refcnt; /* reference count */  
    off_t size; /* file size or directory entry number */  
    struct page_cache_entity_of_inode *page_cache;  
    cap_t pmo_cap; /* fmap fault is handled by this */  
    void *private;  
  
    pthread_rwlock_t rwlock; /* vnode rwlock */  
};
```

其中，private 表示文件系统特定的私有数据，例 对 inode 的引用，refcnt 代表该 vnode 被引用的次数，在下文的 server_entry 中会提到。

练习4

实现 user/system-services/system-servers/fs_base/fs_vnode.c 中 vnode 的 alloc_fs_vnode、get_fs_vnode_by_id、inc_ref_fs_vnode、dec_ref_fs_vnode 函数。

Tip

- 你可能需要回顾Lab2中的代码去了解红黑树的操作方法。

Success

完成练习4后，执行 `make grade`，可以得到 Scores: 35/100。

server_entry

文件描述符（File Descriptor，简称 fd）是操作系统用于管理文件和其他输入/输出资源（管道、网络连接等）的一种抽象标识符。我们来回顾一下计算机系统基础课中学习的 unix 文件系统抽象。在类 Unix 系统（Linux、macOS）中，文件描述符是一个非负整数，它指向一个内核中的文件表项，每个表项包含了文件的各种状态信息和操作方法。ChCore 将进程的 fd 保存在 chcore-libc 当中，同时在文件系统中通过 server_entry 维护了各个 Client 的 fd 的信息，把各个 Client 的 fd 和在文件系统侧的 fid 对应起来 (`(client_badge, fd) -> fid(server_entry)`)，也就是说 server_entry 对应着每个文件系统实例所对应的文件表项，其包含了对应文件表项的文件 offset 以及 vnode 引用。由于一个 vnode 可能会对应多个文件表项，所以 vnode 的引用数需要进行维护。

FS_Base 的 IPC handler 在处理 IPC 请求时，会先把 IPC 消息中包含的文件 fd 转换为 fid，所以我们需要把进程的 fd 和实际所对应的文件表项的映射建立起来，而在 ChCore 中对应的就是 server_mapping 链表。每当处理 IPC 请求时，文件系统都会通过进程发起的 badge 号找到与之对应的映射表，最终得到文件表项的 ID。

练习题 5

实现 `user/system-services/system-servers/fs_base/fs_wrapper.c` 中的 `fs_wrapper_set_server_entry` 和 `fs_wrapper_get_server_entry` 函数。

Tip

- 通过全局变量 `struct list_head server_entry_mapping` 遍历 `server_entry_node`。
- 你可以参考 `fs_wrapper_clear_server_entry` 来理解每一个变量的含义。

Success

完成练习5后，执行 `make grade`，可以得到 Scores: 50/100。

fs_wrapper_ops

当我们拥有了文件表项和VNode抽象后，我们便可以实现真正的文件系统操作了。

我们可以将 FS_Base 以及 FS_Wrapper 的所有逻辑看成一个 VFS 的通用接口，其暴露出的接口定义为 `struct fs_server_ops`。对于每一个文件系统实例，其都需要定义一个全局的名为

`server_ops` 的全局句柄，并将实际的文件系统操作的实现注册到该句柄中。你可以通过查看 `user/system-services/system-servers/tmpfs/tmpfs.c` 中查看 ChCore 的默认 `tmpfs` 文件系统是怎么将其注册到 `FS_Wrapper` 中的。而到了实际处理文件请求时，上层的 `FS_Wrapper` 在响应 IPC 请求的时候，只需要调用 `server_ops` 中的函数指针即可，不需要实际真正调用每一个文件系统实现的操作函数，这样便完成了一个统一的文件操作逻辑。例如在 `tmpfs` 中实际的读命令为 `tmpfs_read` 但在上层的 `fs_wrapper` 看来其调用只需要调用 `server_ops->read` 即可而不需要真正知晓 `tmpfs` 中的函数签名。

对于本 Lab 你只需要实现最基本的 Posix 文件操作即可，即 Open, Close, Read, Write 以及 LSeek 操作。而其下层每个文件系统除了 Open 操作，每当 FS_Base 尝试处理 Posix 文件请求时，其都会调用 `translate_fd_to_fid` 将对应的 fd 翻译成 fid 并重新写回 `struct fs_request` 中的 fd，所以请注意**不需要在实际的fs_wrapper_函数中再次调用该函数**。下面将简述一下每一个函数的语义。

对于 Open 以及 Close 来说，其主要的目的就是创建以及回收 Server Entry 即文件表项。由于在 VFS 中 VNode 的创建是动态的，所以当进程尝试发出 Open 中，我们需要调用与之对应的 `server_ops` 并同时分配对应的文件表项。对于每一个新增的文件表项，我们需要将其关联到对应的内存 VNode 中。由于文件表项所对应的 VNode 可能不在内存中，所以当文件系统返回 inode 号时我们需要尝试查找相应的 vnode，果不存在则尝试分配并将其添加至对应的红黑树中。当完成 VNode 关联后，我们需要使用上一步实现的映射函数，将 `server_entry` 与用户 fd 映射，完成文件表项的创建。对于 Close，我们需要采取类似的逻辑，即回退所有的文件表项操作，减少引用计数，并尝试回收对应的系统资源。

针对 Read/Write/Lseek 操作，你需要参考 `man` 以及对应的 `tests/fs_test` 下的所有测试文件，按照 Posix 语义相应地维护 `server_entry` 以及 `vnode` 信息，并将数据返回给用户进程。

针对 mmap 操作，我们知道针对文件的 mmap 操作是采取 Demand Paging 的内存映射来实现的，当用户进程调用 `mmap` 时，FS 会首先为用户新增一个 pmo 即内存对象，并将其对应的类型设置为 `PMO_FILE`，并为其创建 `Page_Fault` 映射(`user/system-services/system-servers/fs_base/fs_page_fault.c`)，最后将该 pmo 对象发回用户进程并让其进行映射。当用户尝试访问该内存对象，并发生缺页异常时，内核会根据 pmo 的所有者(badge)将异常地址调用到对应FS处理函数进行处理，处理函数为每一个文件系统中的 `user_fault_handler`，此时 FS 服务器会根据缺页地址分配新的内存页，填充文件内容并完成缺页的处理，最终返回内核态，从而递交控制权到原来的用户进程。

练习题 6

实现 `user/system-services/system-servers/fs_base/fs_wrapper_ops.c` 中的 `fs_wrapper_open`、`fs_wrapper_close`、`__fs_wrapper_read_core`、`__fs_wrapper_write_core`，`fs_wrapper_lseek` 函数。

Tip

- `user/chcore-libc/libchcore/porting/overrides/include/chcore-internal/fs_defs.h` 中定义了 `struct fs_request`，其中定义了文件系统收到的 IPC 信息所包含的数据。
- 针对文件表项的 helper 函数 `alloc_entry` 和 `free_entry` 在 `user/system-services/system-servers/fs_base/fs_vnode.c` 中定义。
- `user/system-services/system-servers/tmpfs/tmpfs.c` 中定义了 tmpfs 文件系统提供的文件操作接口 `server_ops`, `fs_wrapper` 接口会调用到 `server_ops` 进行实际的文件操作。
- 用户态的所有针对文件的请求，首先会被路由到 `user/chcore-libc/libchcore/porting/overrides/src/chcore-port/file.c` 中，该文件包含了在调用 `ipc` 前后的预备和收尾工作。
- 你应当回顾 Lab2 的代码，去了解针对 PMO_FILE，内核是怎么处理缺页并将其转发到 FS 中的。同时你需要查看 `user/system-services/system-servers/fs_base/fs_page_fault.c` 中的 `page_fault` 处理函数，了解 FS 是如何处理 mmap 缺页异常的。

Success

完成练习题6后，执行 `make grade`，可以得到 Scores: 100/100。

思考题 7

思考 ChCore 当前实现 VFS 的方式有何利弊？如果让你在微内核操作系统上实现 VFS 抽象，你会如何实现？

Success

以上为Lab5 Part3的所有内容

Last change: 2025-04-15, commit: [90a02e2](#)

BowerAccess

Note

large language models (LLMs) 在当今时代快速发展，成为新一代科技发展创新浪潮的热点。

LLMs 通常具有数以亿计甚至数以百亿计的参数，这使得它们能够捕获语言中的复杂模式和语义关系，但也引入了极高的内存资源开销。以 GPT-3 举例，它的参数量为 175B，在未经优化的条件下，它的运行时内存开销大于 700GB。这远远超出了许多边缘设备的 GPU 显存大小（树莓派 4b 的 GPU 显存大小为 2GB）。

PowerInfer 是发表在 SOSP'24 上的工作，它可以使得 175B 的 LLM 部署在单个商用级 GPU 上，这得益于它观察到了 LLM 访存的稀疏性。即在 LLM 中，并不是所有的神经元都对于计算结果有影响，只有少部分神经元会影响最终的结果，通过只将重要的神经元加载到 GPU 显存中，并选择性地对神经元进行计算，来减少 LLM 的运行时内存开销。

Warning

这是一个我们制作的新Lab，欢迎大家测试并提出意见。

在 BowerAccess Lab 中，我们希望你参考 PowerInfer 的思想修改 chcore，进而在其上部署并优化一个与 LLM 类似的应用程序 `llm`（你可以在 shell 中通过 `./test_llm` 进行访问）。具体而言：

`llm` 是一个 CPU 内存开销极大的程序，它会使用 `mmap()` 映射一个 1MB 大小的**特殊内存空间**（用 `MAP_LLM` 进行标识）并访问其中的数据完成功能，这远远超过了 chcore 管理的 `MAP_LLM` 物理内存大小（出于题目设计考虑，我们将这种特殊内存空间的大小限制为 1MB），直接运行会产生大量的 Page Fault，引入了极大的运行时开销。

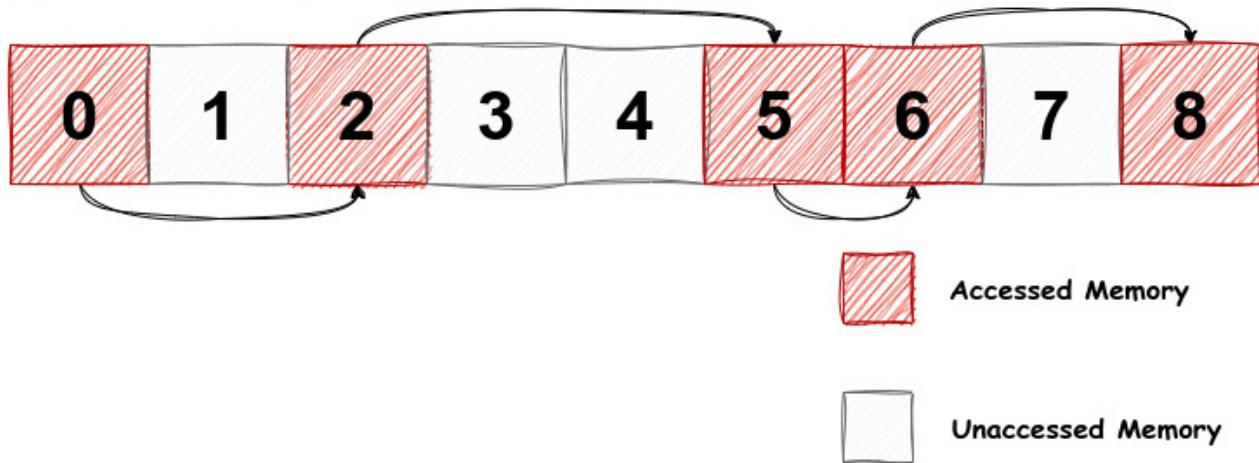
幸运的是，`llm` 的访存具有和 LLM 类似的稀疏性：

- 它虽然映射了非常大的内存空间，但是只会访问其中的一个特定的内存页面子集
- 只会按照单调递增的顺序访问内存页面

`llm` 的 `mmap()` 会传递一个特殊的特殊的 `MAP_LLM` 参数。

`llm` 的访存模式与下图相似（但访问的页面并不相同）：

Mapped Memory



虽然程序总共映射了 9 个页面，但是只访问了 0, 2, 5, 6, 8 这几个页面，且一定按照 0 -> 2 -> 5 -> 6 -> 8 的顺序访问，并没有访问 1, 3, 4, 7 这几个页面。

它的代码是（再次强调它只是示例代码，和真实的 llm 代码相似但不同）：

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>

#define PAGE_SIZE 4096

int main() {
    const char *filename = "weight.dat";
    int fd = open(filename, O_RDWR | O_CREAT, 0666);
    if (fd < 0) {
        perror("open");
        return EXIT_FAILURE;
    }

    // alloc 9 pages
    if (ftruncate(fd, 9 * PAGE_SIZE) == -1) {
        perror("ftruncate");
        close(fd);
        return EXIT_FAILURE;
    }

    // map the file, use the special `MAP_LLM` flag
    char *map = mmap(NULL, 9 * PAGE_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED
| MAP_LLM, fd, 0);
    if (map == MAP_FAILED) {
        perror("mmap");
        close(fd);
        return EXIT_FAILURE;
    }

    // access page 0, 2, 5, 6, 8
    int pages_to_access[] = {0, 2, 5, 6, 8};
    for (size_t i = 0; i < sizeof(pages_to_access) / sizeof(pages_to_access[0]); i++) {
        int page_number = pages_to_access[i];
        map[page_number * PAGE_SIZE] = 'A' + page_number;
        printf("Accessed page %d, first byte: %c\n", page_number,
map[page_number * PAGE_SIZE]);
    }

    if (munmap(map, 9 * PAGE_SIZE) == -1) {
        perror("munmap");
    }

    close(fd);
    return EXIT_SUCCESS;
}
```

根据 `llm` 所具有的**稀疏的、可预测的**访存模式，我们可以设计在 page fault 发生时，预取 (prefetch) 即将访存的多个页面，这样就可以减少 page fault 的次数，提高了运行性能。

Warning

我们只提供 `llm` 的二进制文件，并不提供源码文件。请自由发挥，判断出 `llm` 的具体访存模式。

思考题 8

阅读 `kernel/object/user_fault.c` 下的 `sys_user_fault_map_batched` 函数，并回答下问题

- “`MAP_LLM` 地址空间具有容量限制”这个特点在代码中是如何实现的？
- 对比它和 `sys_user_fault_map` 函数的区别，并回答我们为什么需要引入一个新的系统调用？

练习题 9

为了实现在 `page fault` 的时候预取页面的功能，我们需要修改 `user/system-services/system-servers/fs_base/fs_page_fault.c` 文件：

- 实现 `predict_prefetch_pages` 函数。
- 实现 `handle_one_fault` 函数中针对 `MAP_LLM` 的预取（prefetch）功能，实现需要 `predict_prefetch_pages` 和 `usys_user_fault_map_batched` 函数的配合。
- `user_fault` 中为处理用户地址缺页的Handler。

Success

以上为Lab5的所有内容

Last change: 2025-05-14, commit: [82120fd](#)

Last change: 2025-02-14, commit: [9fa9752](#)

Last change: 2025-02-14, commit: [23a4605](#)

Last change: 2025-02-14, commit: [23a4605](#)

源码解析

这部分内容是对机器启动部分的chcore源代码的详细解析，包括内核启动与页表映射两个部分。

Important

完成 Lab1 后，再阅读这部分内容。

Last change: 2025-02-14, commit: [9fa9752](#)

内核启动

目录

- 内核启动
 - 目录
 - 多内核启动及设置
 - 总览
 - 启动 CPU 0 号核
 - 何让内核依次启动?
 - 0号核
 - 非0号核
 - 后续操作
 - 内核启动时的设置
 - 关于栈的设置
 - bss段清零
 - 切换内核异常级别
 - 启用MMU
 - 初始化串口输出

Info

QEMU模拟器中，当kernel映像文件被bootloader加载到内存中后，内核会被直接带到预先设置好的地址，即 `_start` 函数（0x80000），我们将从这里逐步启动CPU的核心，并做一些必要的设置

Warning

这部分内容是源码解析的一部分，在完成Lab1后，再阅读这部分内容。

让我们把目光放到 `start.s` 文件上，这里是内核启动的开始：

多内核启动及设置

总览

对于多内核的chcore系统，我们在启动内核的时候通常会让一个内核进入启动流程，让其他内

核先进行等待，待该内核完成基本的初始化之后，再让其他核心进行这些流程

Note

通俗理解，就是“排好队，一个一个来”

启动 CPU 0 号核

既然是排队，那么总要有一个先后顺序，我们在chcore中的策略是让0号核心先启动，看代码下：

```
BEGIN_FUNC(_start)
    mrs x8, mpidr_el1
    and x8, x8, #0xFF
    cbz x8, primary
```

关于 `mpidr_el1` 这样的系统寄存器，可以在lab文档里给到的manual里查到相关信息（备注：更方便的手段是先询问llm，然后再在manual里面求证即可）：

Note

- In an implementation containing multiple PEs, each PE is identified by a unique *affinity* value reported by `MPIDR_EL1` {Aff3, Aff2, Aff1, Aff0}, where the value of affinity level 0 is the most significant for determining the PE behavior, and the values of higher affinity levels are less significant. Affinity level 3 is only supported in AArch64 state.
 - An implementation is described as multithreaded when the lowest level of affinity consists of logical PEs that are implemented using a multithreading type approach. In this section, when referring to a multithreaded implementation, *thread* is used to mean processing elements with:
 - `MPIDR_EL1.MT` or `MPIDR.MT` set to 1,
 - Different values for affinity level 0.
 - The same values for affinity level 1 and higher.
-

由此我们得知，`mpidr_el1`寄存器存储的是CPU核心的**唯一标识符**，这里我们使用它来区分不同的核心，逻辑 下：

- 读取系统寄存器的值到 `x8`
- 与 `0xFF` 进行与操作，即保留低8位，是一个mask操作，这样可以去除掉高位的不必要的信息
- 将得到的值与0比较，若相等，则跳转到 `primary` 标签，进行后续操作

如何让内核依次启动？

继续浏览start.S，根据上文的逻辑，在判断出当前CPU是否为0号核心之后，0号核心与非0号核心需要执行的操作是不同的

但是 何让0号核和其他核区别开来，做好自己的启动工作呢？这里给出一个大概的逻辑

0号核

注意到此时代码跳转到了primary标签

primary:

```
/* Turn to el1 from other exception levels. */
bl arm64_elX_to_el1

/* Prepare stack pointer and jump to C. */
adr x0, boot_cpu_stack
add x0, x0, #INIT_STACK_SIZE
mov sp, x0

b init_c

/* Should never be here */
b .
```

关于降低异常级别的部分会在下面提到，我们现在只需要站在宏观的视角理解0号核干了什么：

- 从其他的异常级别降低到1
- 为跳转到C语言部分代码做设置栈的准备
- 跳转到init_c
- 代码的最后是一个死循环，如果前面发生了故障可以将内核卡死在这里，注意到注释也提到了“Should never be here”

非0号核

非0号核在cbz指令判断失败后，会按照顺序继续执行下面的代码，下所示：

```
/* Wait for bss clear */
wait_for_bss_clear:
    adr x0, clear_bss_flag
    ldr x1, [x0]
    cmp x1, #0
    bne wait_for_bss_clear

...
/* Turn to el1 from other exception levels. */
bl arm64_elX_to_el1

/* Prepare stack pointer and jump to C. */
mov x1, #INIT_STACK_SIZE
mul x1, x8, x1
adr x0, boot_cpu_stack
add x0, x0, x1
add x0, x0, #INIT_STACK_SIZE
mov sp, x0

wait_until_smp_enabled:
/* CPU ID should be stored in x8 from the first line */
mov x1, #8
mul x2, x8, x1
ldr x1, =secondary_boot_flag
add x1, x1, x2
ldr x3, [x1]
cbz x3, wait_until_smp_enabled

/* Set CPU id */
mov x0, x8
b secondary_init_c

/* Should never be here */
b .
```

这里的代码采用了**轮询**的手段，通俗的讲，就是反复检查相关条件是否满足。CPU不断检查 `clear_bss_flag` 与 `secondary_boot_flag` 数组里的内容，若收到信号，则执行对应操作。

Tip

更多信息可以参考[轮询的维基百科](#)。

二者具体的操作逻辑不细讲，概括一下：

- bss段清零后，同样执行降低内存级别的操作，随后设置栈
- 这一段完成后继续等待信号，收到通知后即设置CPU id并跳转到这部分内核对应的c代码

后续操作

内核进行完毕初始设置后，即进入 `init_c.c` 部分的代码，在c代码的程序中继续完成相关设

置：

- 叫醒其他核
- 清理bss段数据
- 初始化串口
- 设置mmu
- 注意这里不同内核执行的函数不一样，有高低贵贱之分

```
void init_c(void)
{
    /* Clear the bss area for the kernel image */
    clear_bss();

    /* Initialize UART before enabling MMU. */
    early_uart_init();
    uart_send_string("boot: init_c\r\n");

    wakeup_other_cores();

    /* Initialize Kernel Page Table. */
    uart_send_string("[BOOT] Install kernel page table\r\n");
    init_kernel_pt();

    /* Enable MMU. */
    el1_mmu_activate();
    uart_send_string("[BOOT] Enable el1 MMU\r\n");

    /* Call Kernel Main. */
    uart_send_string("[BOOT] Jump to kernel main\r\n");
    start_kernel(secondary_boot_flag);

    /* Never reach here */
}

void secondary_init_c(int cpuid)
{
    el1_mmu_activate();
    secondary_cpu_boot(cpuid);
}
```

内核启动时的设置

上一部分我们对多内核启动的全部过程有了一个大概的了解，而这一部分则主要讲解内核在启动过程中的具体设置，包括汇编与C代码中的重要函数

事实上，它们是相互交错运行的，共同为新生伊始的CPU内核配置好相关设置

关于栈的设置

```
/* Prepare stack pointer and jump to C. */
adr  x0, boot_cpu_stack
add  x0, x0, #INIT_STACK_SIZE
mov  sp, x0
```

代码中的设置部分是将栈指针的内容准备（获取栈的基地址，计算栈顶地址）好后，直接移动到sp寄存器中，即完成了栈的设置

栈是系统用来存储局部变量、函数参数、返回地址、寄存器值的重要部分，若不设置这一部分，sp寄存器会指向随机地址，对系统的后续行为是毁灭性的打击

bss段清零

.bss段用于存储未初始化的全局变量和静态变量，将这部分值统一设置为0

若没有这一部分操作，则会让全局变量和静态变量的0初始值受到破坏

假 遇到程序或内核操作需要用到默认为0的全局变量，未初始化bss段数据的行为将会导致相应的操作出现bug

这一部分的代码在 `init_c.c` 中，可自行阅读

切换内核异常级别

上面分析 `start.s` 代码时，我们遇到了 `arm_elX_to_el1` 函数，其作用是将内核的异常级别从el3降低到el1。相关代码在同目录 `tool.s` 文件中，我们现在对其进行考察与分析：

```
BEGIN_FUNC(arm64_elX_to_el1)
    mrs x9, CurrentEL

    // Check the current exception level.
    cmp x9, CURRENTEL_EL1
    beq .Ltarget
    cmp x9, CURRENTEL_EL2
    beq .Lin_el2
    // Otherwise, we are in EL3.

    // Set EL2 to 64bit and enable the HVC instruction.

    ...

    // Set the return address and exception level.
    adr x9, .Ltarget
    msr elr_el3, x9
    mov x9, SPSR_ELX_DAIF | SPSR_ELX_EL1H
    msr spsr_el3, x9

.Lin_el2:
    // Disable EL1 timer traps and the timer offset.
    // Disable stage 2 translations.
    // Disable EL2 coprocessor traps.
    // Disable EL1 FPU traps.

    ...

    // Check whether the GIC system registers are supported.
    mrs x9, id_aa64pfr0_el1
    and x9, x9, ID_AA64PFR0_EL1_GIC
    cbz x9, .Lno_gic_sr

    // Enable the GIC system registers in EL2, and allow their use in EL1.
    // Disable the GIC virtual CPU interface.

    ...

.Lno_gic_sr: // No GIC System Registers

    // Set EL1 to 64bit.

    ...

    // Set the return address and exception level.
    adr x9, .Ltarget
    msr elr_el2, x9
    mov x9, SPSR_ELX_DAIF | SPSR_ELX_EL1H
    msr spsr_el2, x9

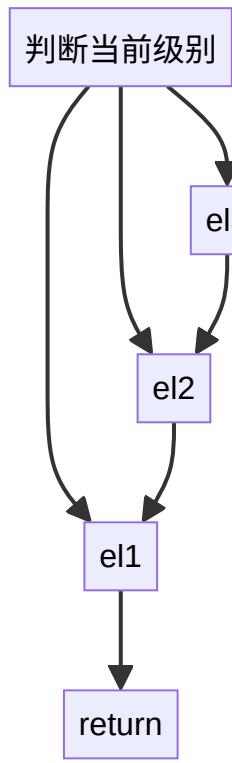
    isb
    eret

.Ltarget:
    ret
END_FUNC(arm64_elX_to_el1)
```

(部分细节处的琐碎设置代码已略去，看注释即可)

纵观全局，我们的源码符合lab文档里“没有直接写死从el3到el1”的逻辑，而是降低异常级别的行动分成了数个步骤来执行：

- 先获取当前异常级别
- 若级别是el3，则直接往下执行
- 若级别是el2/el1，则跳转到相应的部分，总体上是3→2→1的逻辑
- 在最后调用 `eret` 指令，正式调整内核级别



对于 `eret` 指令，这是一个用来从高级别跳转到低级别的指令，执行它需要我们设置两个寄存器：

- `elr_elx`：异常链接寄存器，保存跳转级别后执行的指令地址

Hint

在这里即为 `.target` 标签

- `spsr_elx`：保存的程序状态寄存器，包含异常返回后的异常级别

Hint

在这里即为 `SPSR_ELX_DAIF | SPSR_ELX_EL1H`。由于我们需要将异常级别控制在el1，这里我们的设置是直接将相关宏做或操作后赋值

关于代码的其他部分，可以阅读注释作初步了解，深入学习可以结合l1m与教材

启用MMU

果你看过 `init_c.c` 文件，会发现我们在内核启动时还需要进行启动页表的相关配置。关于页表的具体配置较为复杂，会在另一篇解析单独讲解，这里主要讲解启用MMU的部分

启用MMU部分的代码同样在 `tool.s` 文件中，相关代码 下：

```
BEGIN_FUNC(el1_mmu_activate)
    stp    x29, x30, [sp, #-16]!
    mov    x29, sp

    bl invalidate_cache_all

    /* Invalidate TLB */
    /* Initialize Memory Attribute Indirection Register */
    /* Initialize TCR_EL1 */
    /* set cacheable attributes on translation walk */
    /* (SMP extensions) non-shareable, inner write-back write-allocate */
    /* Write ttbr with phys addr of the translation table */

    ...

    mrs    x8, sctlr_el1
    /* Enable MMU */
    orr    x8, x8, #SCTLR_EL1_M
    /* Disable alignment checking */
    bic    x8, x8, #SCTLR_EL1_A
    bic    x8, x8, #SCTLR_EL1_SA0
    bic    x8, x8, #SCTLR_EL1_SA
    orr    x8, x8, #SCTLR_EL1_nAA
    /* Data accesses Cacheable */
    orr    x8, x8, #SCTLR_EL1_C
    /* Instruction access Cacheable */
    orr    x8, x8, #SCTLR_EL1_I
    /* Writable eXecute Never */
    orr    x8, x8, #SCTLR_EL1_WXN
    msr    sctlr_el1, x8

    ldp    x29, x30, [sp], #16
    ret
END_FUNC(el1_mmu_activate)
```

这时候我们的内核异常级别已经降低到el1，而启用MMU的操作同样是通过为系统寄存器进行相应的赋值（即硬件与软件的相互配合），代码中则是通过不断配置相关的字段来实现的，对于这里的源码，我们执行的操作 下：

- 启用MMU，即 M 字段，这个是必须的
- 禁用内存对齐检查，即 A, SA0, SA, nAA 字段
- 启用指令与数据缓存，即 C, I 字段
- 启用写保护，即 WXN 字段，可写页但不可执行

初始化串口输出

同样是 `init_c.c` 中的操作，我们需要对树莓派的UART串口进行初始化启用，从而使kernel能输出字符

具体的实现在 `uart.c` 文件中，代码结构 下所示：

```
#if USE_mini_uart == 1

// Mini UART代码
void early_uart_init(void) { ... }
static unsigned int early_uart_lsr(void) { ... }
static void early_uart_send(unsigned int c) { ... }

#else

// PL011代码
void early_uart_init(void) { ... }
static unsigned int early_uart_fr(void) { ... }
static void early_uart_send(unsigned int c) { ... }

#endif

void uart_send_string(char *str) {
int i;
for (i = 0; str[i] != '\0'; i++) {
    if (str[i] == '\n')
        early_uart_send('\r');
    early_uart_send(str[i]);
}
}
```

其中上半部分的代码内容涉及到硬件的操作，设置引脚、波特率等，我们无需了解。而这里的条件编译结构则为我们提供了两种uart——`mini uart` 与 `主uart`，同时二者对外的字符串发送接口是一样的，对外部保持了统一与抽象屏障

下半部分则是对字符串的具体发送工作，逻辑很简单——使用一个循环溜过去即可，遇到字符串结束符 `\0` 即停止

代码中在 `\n` 前方添加 `\r` 是为了兼容不同终端的换行处理。例，在早期的Mac OS中，使用的是Carriage Return (CR)，即 `\r` 作为换行符

Success

至此，内核启动部分的源码解析全部结束，页表映射的部分将在接下来的文章中讲述，希望对你的学习进步有所裨益！

Last change: 2025-02-14, commit: [9fa9752](#)

页表映射

目录

- 页表映射
 - 复习：页表结构
 - 页表基址寄存器
 - 页表地址翻译
 - Chcore的物理地址空间分布
 - Chcore页表映射
 - 总览
 - 宏定义与数据结构声明
 - 内存区域划分
 - 页表项数组定义
 - 页表控制属性描述符
 - 提取索引辅助函数
 - 页表地址映射
 - 低地址映射
 - 高地址映射
 - 配置本地外设内存映射

Info

chcore内核启动的最后一步是完成**页表的地址映射工作**。在前文中，我们于tool.S（被init_c.c调用）中启用了MMU以及相关配置，但具体的页表映射工作尚未提及，本节内容即为对chcore页表映射内容的源码解析

Warning

这部分内容是源码解析的一部分，在完成Lab1后，再阅读这部分内容。

参考源码文件：mmu.c，与init_c.c同目录

复习：页表结构

页表基址寄存器

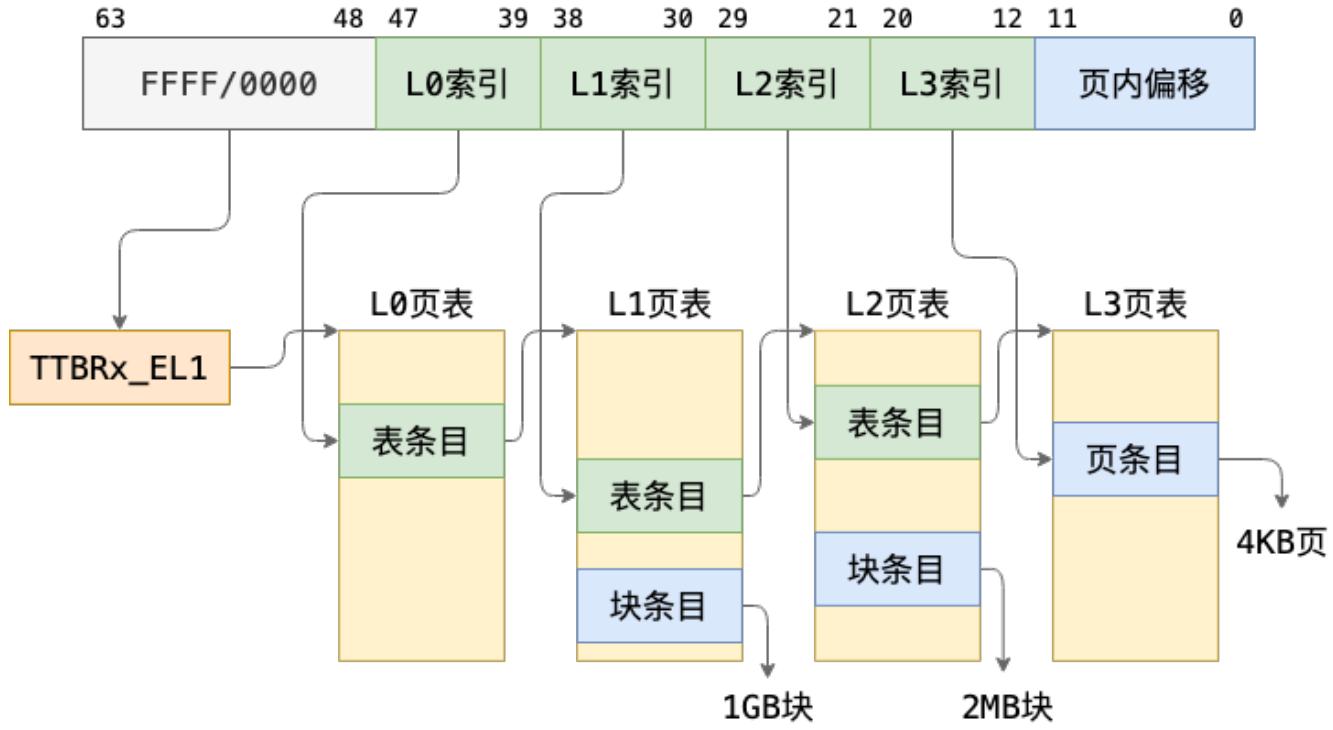
在 AArch64 架构的 EL1 异常级别存在两个页表基址寄存器：`ttbr0_el1` 和 `ttbr1_el1`，分别用作虚拟地址空间**低地址**和**高地址**的翻译。而关于高低地址的具体范围则由由 `tcr_el1` 翻译控制寄存器控制。

一般情况下，我们会将 `tcr_el1` 配置为高低地址各有 48 位的地址范围，即：

- `0x0000_0000_0000_0000 ~ 0x0000_ffff_ffff_ffff` 为低地址
- `0xffff_0000_0000_0000 ~ 0xffff_ffff_ffff_ffff` 为高地址

页表地址翻译

有了页表基址寄存器的知识，我们再来看 chcore 是如何翻译地址的。chcore 中页表的地址翻译采取了**多级页表**的形式，下图所示：



所谓**多级页表**，是一种内存管理技术，用于虚拟内存系统中将虚拟地址映射到物理地址。它通过**多层次结构**来减少页表所占用的内存空间，并提高页表的查找效率。

Note

在多级页表结构中，虚拟地址被分割成多个字段，每个字段对应不同级别的页表索引。最顶层的页表包含指向下一一级页表的指针，而每一层页表都包含指向更详细页表或物理内存页的指针。

在 Chcore 中，页表一共分为 4 级：从 L0-L2 都是对下一级别索引的指针，一直到最后一级 L3，才指向具体到页（以 4KB 粒度）

Chcore的物理地址空间分布

下图所示，这是我们后面页表映射时确定物理地址的重要信息

物理地址范围	对应设备
0x00000000 ~ 0x3f000000	物理内存 (SDRAM)
0x3f000000 ~ 0x40000000	共享外设内存
0x40000000 ~ 0xffffffff	本地 (每个 CPU 核独立) 外设内存

接下来，就让我们一起看看具体的页表映射源码吧！

Chcore页表映射

总览

先总览代码，获取一个对源码的大致印象与结构

```
#include <common/macro.h>
#include "image.h"
#include "boot.h"
#include "consts.h"

typedef unsigned long u64;
typedef unsigned int u32;

/* Physical memory address space: 0-1G */
#define PHYSMEM_START      (0x0UL)
#define PERIPHERAL_BASE   (0x3F000000UL)
#define PHYSMEM_END        (0x40000000UL)

/* The number of entries in one page table page */
#define PTP_ENTRIES 512
/* The size of one page table page */
#define PTP_SIZE 4096
#define ALIGN(n) __attribute__((__aligned__(n)))
u64 boot_ttbr0_l0[PTP_ENTRIES] ALIGN(PTP_SIZE);
u64 boot_ttbr0_l1[PTP_ENTRIES] ALIGN(PTP_SIZE);
u64 boot_ttbr0_l2[PTP_ENTRIES] ALIGN(PTP_SIZE);
u64 boot_ttbr0_l3[PTP_ENTRIES] ALIGN(PTP_SIZE);

u64 boot_ttbr1_l0[PTP_ENTRIES] ALIGN(PTP_SIZE);
u64 boot_ttbr1_l1[PTP_ENTRIES] ALIGN(PTP_SIZE);
u64 boot_ttbr1_l2[PTP_ENTRIES] ALIGN(PTP_SIZE);
u64 boot_ttbr1_l3[PTP_ENTRIES] ALIGN(PTP_SIZE);

#define IS_VALID (1UL << 0)
#define IS_TABLE (1UL << 1)
#define IS_PTE   (1UL << 1)

#define PNX          (0x1UL << 53)
#define UXN          (0x1UL << 54)
#define ACCESSED    (0x1UL << 10)
#define NG           (0x1UL << 11)
#define INNER_SHARABLE (0x3UL << 8)
#define NORMAL_MEMORY  (0x4UL << 2)
#define DEVICE_MEMORY  (0x0UL << 2)
#define RDONLY_S     (0x2UL << 6)

#define SIZE_2M (2UL * 1024 * 1024)
#define SIZE_4K (4UL * 1024)

#define GET_L0_INDEX(x) (((x) >> (12 + 9 + 9 + 9)) & 0x1ff)
#define GET_L1_INDEX(x) (((x) >> (12 + 9 + 9)) & 0x1ff)
#define GET_L2_INDEX(x) (((x) >> (12 + 9)) & 0x1ff)
#define GET_L3_INDEX(x) (((x) >> (12)) & 0x1ff)

extern int boot_cpu_stack[PLAT_CPU_NUMBER][INIT_STACK_SIZE];

void init_kernel_pt(void)
{
    u64 vaddr = PHYSMEM_START;

    /* TTBR0_EL1 0-1G */
```

```

boot_ttbr0_l0[GET_L0_INDEX(vaddr)] = ((u64)boot_ttbr0_l1) |
IS_TABLE
                                | IS_VALID | NG;
boot_ttbr0_l1[GET_L1_INDEX(vaddr)] = ((u64)boot_ttbr0_l2) |
IS_TABLE
                                | IS_VALID | NG;

boot_ttbr0_l2[GET_L2_INDEX(vaddr)] = ((u64)boot_ttbr0_l3) |
IS_TABLE
                                | IS_VALID | NG;

/* first 2M, including .init section */
for (; vaddr < SIZE_2M; vaddr += SIZE_4K) {
    boot_ttbr0_l3[GET_L3_INDEX(vaddr)] =
        (vaddr) /* Unprivileged execute never */
        | PXN /* Privileged execute never */
        | ACCESSED /* Set access flag */
        | NG /* Mark as not global */
        | INNER_SHARABLE /* Shareability */
        | NORMAL_MEMORY /* Normal memory */
        | IS_PTE | IS_VALID;

    /*
     * Code in init section(img_start~init_end) should be
     * mmaped as
     * RONLY_S due to WGN
     * The boot_cpu_stack is also in the init section, but
     * should
     * have write permission
     */
    if (vaddr >= (u64)(&img_start) && vaddr < (u64)
(&init_end)
        && (vaddr < (u64)boot_cpu_stack
            || vaddr >= ((u64)boot_cpu_stack)
                + PLAT_CPU_NUMBER
                    * INIT_STACK_SIZE))
{
    boot_ttbr0_l3[GET_L3_INDEX(vaddr)] &= ~PWN;
    boot_ttbr0_l3[GET_L3_INDEX(vaddr)] |=
        RONLY_S; /* Read Only*/
}
}

/* Normal memory: PHYSMEM_START ~ PERIPHERAL_BASE */
/* Map with 2M granularity */
for (; vaddr < PERIPHERAL_BASE; vaddr += SIZE_2M) {
    boot_ttbr0_l2[GET_L2_INDEX(vaddr)] =
        (vaddr) /* low mem, va = pa */
        | UXN /* Unprivileged execute never */
        | ACCESSED /* Set access flag */
        | NG /* Mark as not global */
        | INNER_SHARABLE /* Shareability */
        | NORMAL_MEMORY /* Normal memory */
        | IS_VALID;
}

/* Peripheral memory: PERIPHERAL_BASE ~ PHYSMEM_END */

```

```

/* Map with 2M granularity */
for (vaddr = PERIPHERAL_BASE; vaddr < PHYSMEM_END; vaddr += SIZE_2M) {
    boot_ttbr0_l2[GET_L2_INDEX(vaddr)] =
        (vaddr) /* low mem, va = pa */
        | UXN /* Unprivileged execute never */
        | ACCESSED /* Set access flag */
        | NG /* Mark as not global */
        | DEVICE_MEMORY /* Device memory */
        | IS_VALID;
}

/* TTBR1_EL1 0-1G */
/* BLANK BEGIN */
vaddr = KERNEL_VADDR + PHYSMEM_START;
boot_ttbr1_l0[GET_L0_INDEX(vaddr)] = ((u64)boot_ttbr1_l1) |
IS_TABLE
                                | IS_VALID;
boot_ttbr1_l1[GET_L1_INDEX(vaddr)] = ((u64)boot_ttbr1_l2) |
IS_TABLE
                                | IS_VALID;

/* Normal memory: PHYSMEM_START ~ PERIPHERAL_BASE
 * The text section code in kernel should be mapped with flag R/
X.
 * The other section and normal memory is mapped with flag R/W.
 * memory layout :
 * | normal memory | kernel text section | kernel data section
...
| normal memory |
*/
boot_ttbr1_l2[GET_L2_INDEX(vaddr)] = ((u64)boot_ttbr1_l3) |
IS_TABLE
                                | IS_VALID;

/* the kernel text section was mapped in the first
 * L2 page table in boot_ptd_l1 now.
 */
BUG_ON((u64)(&_text_end) >= KERNEL_VADDR + SIZE_2M);
/* _text_start & _text_end should be 4K aligned*/
BUG_ON((u64)(&_text_start) % SIZE_4K != 0
       || (u64)(&_text_end) % SIZE_4K != 0);

for (; vaddr < KERNEL_VADDR + SIZE_2M; vaddr += SIZE_4K) {
    boot_ttbr1_l3[GET_L3_INDEX(vaddr)] =
        (vaddr - KERNEL_VADDR) | UXN /* Unprivileged
execute
                                never */
        | PXN /* Privileged execute never*/
        | ACCESSED /* Set access flag */
        | INNER_SHARABLE /* Shareability */
        | NORMAL_MEMORY /* Normal memory */
        | IS_PTE | IS_VALID;
/* (KERNEL_VADDR + TEXT_START ~ KERNEL_VADDR + TEXT_END)
was
        * mapped to physical address (PHY_START ~ PHY_START +

```

```

TEXT_END)
    * with R/X
    */
    if (vaddr >= (u64)(&_text_start) && vaddr < (u64)
(&_text_end)) {
        boot_ttbr1_l3[GET_L3_INDEX(vaddr)] &= ~PXM;
        boot_ttbr1_l3[GET_L3_INDEX(vaddr)] |=
            RDONLY_S; /* Read Only*/
    }
}

for (; vaddr < KERNEL_VADDR + PERIPHERAL_BASE; vaddr += SIZE_2M)
{
    /* No NG bit here since the kernel mappings are shared */
    boot_ttbr1_l2[GET_L2_INDEX(vaddr)] =
        (vaddr - KERNEL_VADDR) /* high mem, va = kbase +
pa */
    | UXN /* Unprivileged execute never */
    | PXM /* Privileged execute never*/
    | ACCESSED /* Set access flag */
    | INNER_SHARABLE /* Shareability */
    | NORMAL_MEMORY /* Normal memory */
    | IS_VALID;
}

/* Peripheral memory: PERIPHERAL_BASE ~ PHYSMEM_END */
/* Map with 2M granularity */
for (vaddr = KERNEL_VADDR + PERIPHERAL_BASE;
     vaddr < KERNEL_VADDR + PHYSMEM_END;
     vaddr += SIZE_2M) {
    boot_ttbr1_l2[GET_L2_INDEX(vaddr)] =
        (vaddr - KERNEL_VADDR) /* high mem, va = kbase +
pa */
    | UXN /* Unprivileged execute never */
    | PXM /* Privileged execute never*/
    | ACCESSED /* Set access flag */
    | DEVICE_MEMORY /* Device memory */
    | IS_VALID;
}

/*
 * Local peripherals, e.g., ARM timer, IRQs, and mailboxes
 *
 * 0x4000_0000 .. 0xFFFF_FFFF
 * 1G is enough (for Mini-UART). Map 1G page here.
 */
vaddr = KERNEL_VADDR + PHYSMEM_END;
boot_ttbr1_l1[GET_L1_INDEX(vaddr)] = PHYSMEM_END | UXN /*

Unprivileged
execute
never
*/
| PXM /* Privileged execute
never*/
| ACCESSED /* Set access
flag */
| DEVICE_MEMORY /* Device

```

```
memory */  
| IS_VALID;  
}
```

结合注释信息可知，这部分代码主要分为两部分：

- 宏定义与数据结构声明：这部分定义了后面页表配置时相应属性对应的宏以及多级页表中会用到的数据结构；此外，宏定义中还包括内存区域划分与页表大小等信息
- 页表地址映射：即 `init_kernel_pt()` 函数，我们的页表映射工作即在此完成，也是我们源码解析的重点所在

宏定义与数据结构声明

这一部分主要介绍代码中的宏定义与页表配置相关的数据结构定义

内存区域划分

下方代码所示：

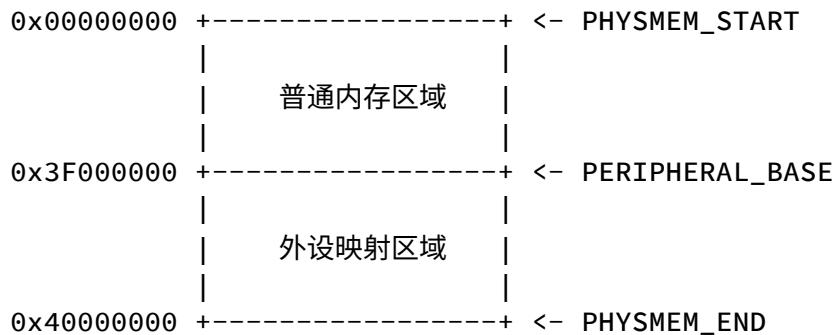
```
/* Physical memory address space: 0-1G */  
#define PHYSMEM_START (0x0UL)  
#define PERIPHERAL_BASE (0x3F0000000UL)  
#define PHYSMEM_END (0x400000000UL)
```

这三行代码声明的宏将我们要映射的物理地址（一共1G）分为了两部分：普通的RAM内存区域与外设映射区域，其中 `UL` 表示 `unsigned long`

其中前者很好理解，就是内核自身的RAM内存，关于后者“外设映射”，可以理解为是在这部分地址开始映射各种硬件外设，例如：

- GPIO控制器
- UART串口
- 中断控制器
- 定时器
- USB控制器等

总体的内存结构即下图所示：



页表项数组定义

下方代码所示：

```

/* The number of entries in one page table page */
#define PTP_ENTRIES 512
/* The size of one page table page */
#define PTP_SIZE 4096
#define ALIGN(n) __attribute__((__aligned__(n)))
u64 boot_ttbr0_l0[PTP_ENTRIES] ALIGN(PTP_SIZE);
u64 boot_ttbr0_l1[PTP_ENTRIES] ALIGN(PTP_SIZE);
u64 boot_ttbr0_l2[PTP_ENTRIES] ALIGN(PTP_SIZE);
u64 boot_ttbr0_l3[PTP_ENTRIES] ALIGN(PTP_SIZE);

u64 boot_ttbr1_l0[PTP_ENTRIES] ALIGN(PTP_SIZE);
u64 boot_ttbr1_l1[PTP_ENTRIES] ALIGN(PTP_SIZE);
u64 boot_ttbr1_l2[PTP_ENTRIES] ALIGN(PTP_SIZE);
u64 boot_ttbr1_l3[PTP_ENTRIES] ALIGN(PTP_SIZE);

```

其中数组部分比较好理解，看名称：ttbrx即表示页表基址寄存器，lx表示具体的页表级数（0-3），而数组大小即为定义好的512，是页表页的入口条数

这里再说说这一行代码：

```
#define ALIGN(n) __attribute__((__aligned__(n)))
```

这行代码定义了一个**对齐属性的宏**：`__attribute__((__aligned__(n)))` 是GCC编译器的一个特殊属性声明，它告诉编译器将变量或数据结构按照n字节边界对齐

例 下面这行声明代码：

```
u64 boot_ttbr0_l0[PTP_ENTRIES] ALIGN(PTP_SIZE);
```

这里 `ALIGN(PTP_SIZE)` 其中 `PTP_SIZE = 4096`，意味着这个数组将被对齐到4KB边界

而关于为什么需要对齐，这便涉及到硬件架构要求和性能优化的相关知识了，感兴趣的可以自己多查阅了解阅读

页表控制属性描述符

下方代码所示：

```
#define IS_VALID (1UL << 0)
#define IS_TABLE (1UL << 1)
#define IS_PTE (1UL << 1)

#define PNX (0x1UL << 53)
#define UXN (0x1UL << 54)
#define ACCESSED (0x1UL << 10)
#define NG (0x1UL << 11)
#define INNER_SHARABLE (0x3UL << 8)
#define NORMAL_MEMORY (0x4UL << 2)
#define DEVICE_MEMORY (0x0UL << 2)
#define RDONLY_S (0x2UL << 6)
```

这部分定义了页表的属性描述符，在配置页表的时候，我们可以通过将待配置的地址与之进行或运算（即 | ）即可

而页表属性的具体含义通常与内存访问权限等相关，具体见下，亦可以自行做更多了解：

- UXN：用户模式（非特权态）下不可执行
- PNX：特权模式（特权态）下不可执行
- RDONLY_S：只读访问
- INNER_SHARABLE：内部可共享
- NORMAL_MEMORY/DEVICE_MEMORY：内存类型标识
- NG：非全局页面标识

提取索引辅助函数

下方代码所示：

```
#define GET_L0_INDEX(x) (((x) >> (12 + 9 + 9 + 9)) & 0x1ff)
#define GET_L1_INDEX(x) (((x) >> (12 + 9 + 9)) & 0x1ff)
#define GET_L2_INDEX(x) (((x) >> (12 + 9)) & 0x1ff)
#define GET_L3_INDEX(x) (((x) >> (12)) & 0x1ff)
```

这部分是用于将虚拟地址提取出对应位置的索引的辅助函数，从L0到L3都有。其中各个数字的含义 下：

- 12: 页内偏移位数 (4KB页面 = 2^{12})
- 9: 每级页表索引的位数 (512个表项 = 2^9)
- 0x1ff: 9位掩码 (二进制: 111111111)，是一个mask操作

绝知此事要躬行，我们假设有一个虚拟地址：

位数:		9位		9位		9位		9位		12位	
内容:		L0索引		L1索引		L2索引		L3索引		页内偏移	

那么各辅助函数的功能即 下所述:

- GET_L0_INDEX : 右移39位， 获取最高的9位
- GET_L1_INDEX : 右移30位， 获取第二个9位
- GET_L2_INDEX : 右移21位， 获取第三个9位
- GET_L3_INDEX : 右移12位， 获取第四个9位

页表地址映射

工欲善其事，必先利其器。上面的介绍为我们解析配置页表部分的源码扫清了障碍，现在，我们正式进入 `init_kernel_pt()` 函数，来对chcore的页表映射逻辑一窥究竟

浏览代码不难发现，本函数主要分为两大块：低地址映射与高地址映射，前者为用户态，后者为内核态，分别由相应的页表基址寄存器控制。其中各自具体配置手段相似，区别在于内核态配置时需要加上相应的偏移量，否则配置就是乱的

低地址映射

我们将详细讲解这一部分，对于后面的高地址映射，我们将只说明不同的地方，其余大体上是相似的

- 首先是设置多级页表之间的链接关系

```

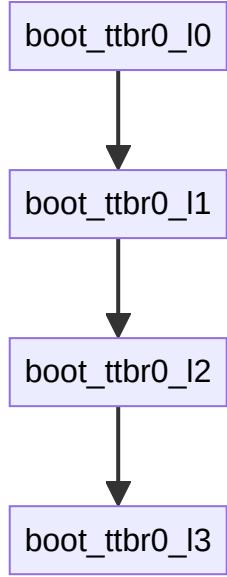
u64 vaddr = PHYSMEM_START;

/* TTBR0_EL1 0-1G */
boot_ttbr0_l0[GET_L0_INDEX(vaddr)] = ((u64)boot_ttbr0_l1) | IS_TABLE
                                         | IS_VALID | NG;
boot_ttbr0_l1[GET_L1_INDEX(vaddr)] = ((u64)boot_ttbr0_l2) | IS_TABLE
                                         | IS_VALID | NG;

boot_ttbr0_l2[GET_L2_INDEX(vaddr)] = ((u64)boot_ttbr0_l3) | IS_TABLE
                                         | IS_VALID | NG;

```

这里将vaddr设置为最开始的物理内存起始，然后进行了相应的链接关系配置



这里完成了初始化工作后，后面便开始了具体的配置

- 初始2M内存，以4KB粒度映射，注意这里包含 .init 部分，需要做特殊处理

```

/* first 2M, including .init section */
for (; vaddr < SIZE_2M; vaddr += SIZE_4K) {
    boot_ttbr0_l3[GET_L3_INDEX(vaddr)] =
        (vaddr) | UXN /* Unprivileged execute never */
        | PNX /* Privileged execute never */
        | ACCESSED /* Set access flag */
        | NG /* Mark as not global */
        | INNER_SHARABLE /* Shareability */
        | NORMAL_MEMORY /* Normal memory */
        | IS_PTE | IS_VALID;

    /*
     * Code in init section(img_start~init_end) should be mmaped
     * as
     * RONLY_S due to WGN
     * The boot_cpu_stack is also in the init section, but should
     * have write permission
     */
    if (vaddr >= (u64)(&img_start) && vaddr < (u64)(&init_end)
        && (vaddr < (u64)boot_cpu_stack
            || vaddr >= ((u64)boot_cpu_stack)
                + PLAT_CPU_NUMBER
                * INIT_STACK_SIZE)) {
        boot_ttbr0_l3[GET_L3_INDEX(vaddr)] &= ~PNX;
        boot_ttbr0_l3[GET_L3_INDEX(vaddr)] |=
            RONLY_S; /* Read Only*/
    }
}
  
```

主体的配置过程其实就是这样：

相应的页表级[辅助函数获取索引(vaddr)] = (vaddr) | 一大堆属性

这里还要注意一下 `for` 循环的末尾，有一个对 `.init` 部分内存的特殊设置—— `RONLY_S`，即只读

- 配置普通RAM内存与外设内存

```
/* Normal memory: PHYSMEM_START ~ PERIPHERAL_BASE */
/* Map with 2M granularity */
for (; vaddr < PERIPHERAL_BASE; vaddr += SIZE_2M) {
    boot_ttbr0_l2[GET_L2_INDEX(vaddr)] =
        (vaddr) /* low mem, va = pa */
        | UXN /* Unprivileged execute never */
        | ACCESSED /* Set access flag */
        | NG /* Mark as not global */
        | INNER_SHARABLE /* Shareability */
        | NORMAL_MEMORY /* Normal memory */
        | IS_VALID;
}

/* Peripheral memory: PERIPHERAL_BASE ~ PHYSMEM_END */
/* Map with 2M granularity */
for (vaddr = PERIPHERAL_BASE; vaddr < PHYSMEM_END; vaddr += SIZE_2M)
{
    boot_ttbr0_l2[GET_L2_INDEX(vaddr)] =
        (vaddr) /* low mem, va = pa */
        | UXN /* Unprivileged execute never */
        | ACCESSED /* Set access flag */
        | NG /* Mark as not global */
        | DEVICE_MEMORY /* Device memory */
        | IS_VALID;
}
```

不难发现总体上的代码逻辑是相似的，那 何区分不同的内存呢？——通过页表属性即可

在这一部分我们以2M的粒度对普通内存+外设内存完成了映射，注意在2M粒度下，我们的页表级是L2

高地址映射

总体上和低地址映射是相似的，不同之处在于二者之间有一个偏移量，因此我们在配置高地址映射的时候需要加上这一部分，它由相应的页表基址寄存器控制。在代码中，即为 `KERNEL_VADDR`，定义在 `image.h` 头文件中

这里我们只举一个例子来说明这一点，就不全部讲解了

```

/* TTBR1_EL1 0-1G */
/* BLANK BEGIN */
vaddr = KERNEL_VADDR + PHYSMEM_START;

...
for (; vaddr < KERNEL_VADDR + PERIPHERAL_BASE; vaddr += SIZE_2M) {
    /* No NG bit here since the kernel mappings are shared */
    boot_ttbr1_l2[GET_L2_INDEX(vaddr)] =
        (vaddr - KERNEL_VADDR) /* high mem, va = kbase + pa
*/
    | UXN /* Unprivileged execute never */
    | PXN /* Privileged execute never*/
    | ACCESSED /* Set access flag */
    | INNER_SHARABLE /* Shareability */
    | NORMAL_MEMORY /* Normal memory */
    | IS_VALID;
}

...

```

以配置普通RAM内存这一段为例，这里初始化 `vaddr` 时即加上了对应的偏移量，在 `for` 循环中也有相应的体现，这便是高地址映射时不同的地方

配置本地外设内存映射

注意到1G-4G这部分内存还没有用到，这部分是留给配置本地外设用的。在chcore中，我们只配置了1G，但是这是足够的，下方代码所示，这也是页表映射的最后一段：

```

/*
 * Local peripherals, e.g., ARM timer, IRQs, and mailboxes
 *
 * 0x4000_0000 .. 0xFFFF_FFFF
 * 1G is enough (for Mini-UART). Map 1G page here.
 */
vaddr = KERNEL_VADDR + PHYSMEM_END;
boot_ttbr1_l1[GET_L1_INDEX(vaddr)] = PHYSMEM_END | UXN /* Unprivileged
execute
never
*/
| PXN /* Privileged execute
never*/
| ACCESSED /* Set access flag */
| DEVICE_MEMORY /* Device memory
|
| IS_VALID;
*/

```

1G的内存配置是便直接使用L1级别的页表了，这也体现了多级页表的特点

Success

至此，页表映射部分的源码解析全部结束，希望对你学习进步有所裨益！

Last change: 2025-02-14, commit: [9fa9752](#)

教材补充

Info

本章节将从我们lab的配套教材《操作系统原理与实现》（机械工业出版社）入手，补充与lab相对应的相关知识，方便读者更好的吸收与理解。我们会列举出每一个知识点对应的原著章节，便于读者查阅。本章节并不会成为教材内容的简单复述，而是希望提取出与lab内容相关的知识点，阐述他们在ChCore中的体现，进行比对学习，更好地理解我们的ChCore。

一、ChCore启动与异常向量表初始化

Tip

推荐阅读：3.2 操作系统的硬件运行环境

1、特权级别与系统ISA

为了区分应用程序和操作系统的运行权限，CPU为两者提供了不同的特权级别：**用户态**和**内核态**。ISA（指令集架构）作为CPU向软件提供的接口，也对应地分为用户ISA和系统ISA，在用户态运行的软件只能使用**用户ISA**，在内核态运行的软件则可以同时使用**系统ISA**和**用户ISA**。

我们举例说明：

- (1) 通用寄存器、栈寄存器、条件码寄存器、运算指令等，都属于**用户ISA**
- (2) 系统ISA则包含系统状态、系统寄存器与系统指令。其中，系统状态包括当前CPU的特权级别、CPU发生错误时引发错误的指令地址、程序运行状态等。存储这些状态的寄存器称为**系统寄存器**，这些寄存器只能由运行在内核态的软件通过系统指令来访问。

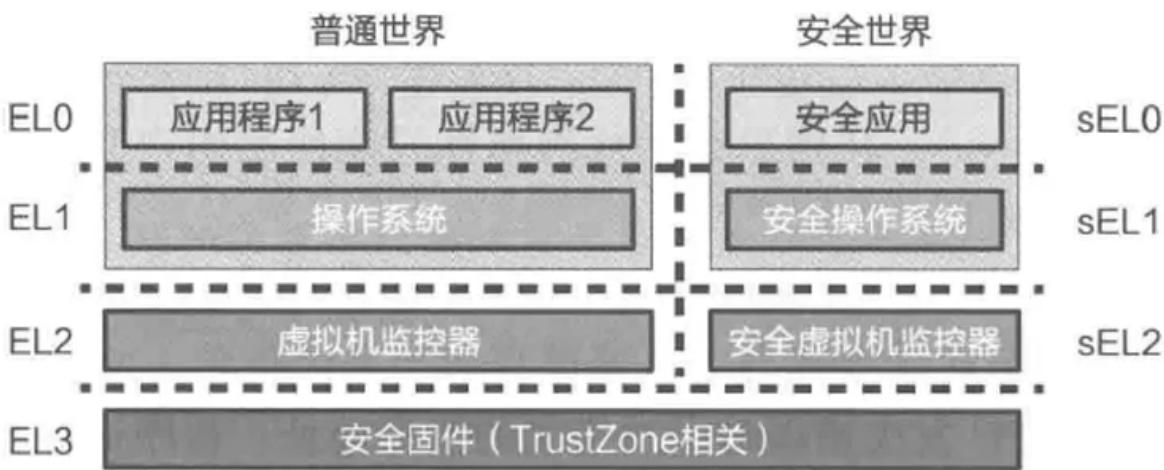
案例分析：ARM的特权级别和系统ISA

接下来我们以AArch64为例介绍CPU特权级别和系统ISA的一种具体实现。

在AArch64中的特权级别被称为**异常级别**（Exception Level, EL），共分为4个级别，具体下：

- EL0：用户态，**应用程序**通常运行在该特权级别。
- EL1：内核态，**操作系统**通常运行在该特权级别。
- EL2：用于虚拟化场景，**虚拟机监控器**通常运行在该特权级别。

- EL3：与安全特性TrustZone相关，负责普通世界和安全世界之间的切换。

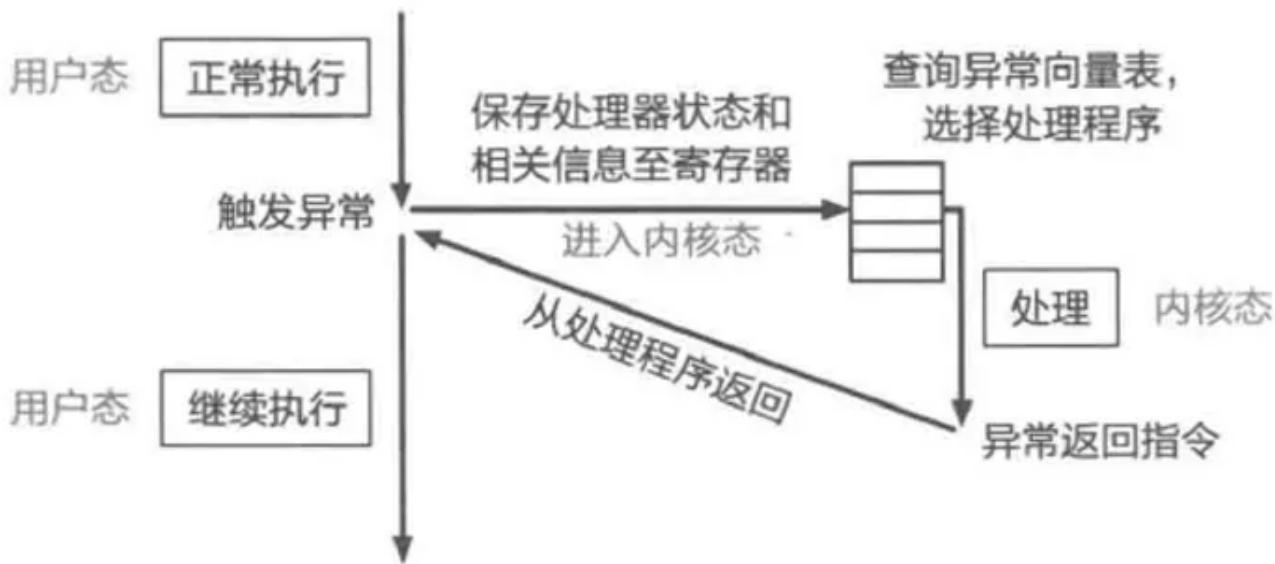


这里需要注意的一点在于，对于许多ISA来说，当CPU运行在内核态运行用户ISA，一般会使用用户ISA的寄存器（SP），这也是为什么从用户态切换到内核时，首先需要将用户态寄存器的值保存到内存。但是AArch64的做法是，为一些常用的用户态寄存器在不同特权级提供不同的硬件副本。例 对于栈寄存器SP，AArch64提供了SP_EL0（用户态）与SP_EL1（内核态）。其中，用户态在函数调用时使用SP_EL0，无法访问SP_EL1；内核态则使用SP_EL1。但也有权限读写SP_EL0。

AArch64的系统寄存器负责保存硬件系统状态，以及为操作系统提供管理硬件的接口。系统ISA提供了mrs和msr两条特权指令，其作用是从系统寄存器中读取值（获取系统信息）或向系统寄存器中写入值（控制系统状态）。系统ISA的指令只有在特权态才能运行，CPU在执行相关指令前会先根据PSTATE中的状态来判断是否合法。例，当PSTATE记录当前运行级别为EL0时，CPU运行的指令无权访问ELREL1系统寄存器。由于AArch64有多个特权级，因此对于系统寄存器，也需要通过类似的后缀来表明这些寄存器在哪一个特权级下使用，例 TTBR0EL1（一阶段页表基址寄存器）和TTBR0_EL2（用于虚拟化的二阶段页表基址寄存器）。

2、异常机制与异常向量表

经过ICS的学习，相信我们的读者对于异常机制已经非常熟悉了，这里我们用一张图回顾一下异常控制流



当操作系统发生异常事件导致“下陷”到内核态时，CPU只允许从固定的入口开始执行。为此，操作系统需要提前将代码的入口地址“告诉”处理器。对不同类型的异常事件，CPU通常支持设置不同的入口。这些入口通常以一张表的形式记录在内存中，也就是异常向量表，由操作系统负责构造。在系统启动后，操作系统会将**异常向量表的内存地址**写入CPU上的一个特殊寄存器——**异常向量表基址寄存器**（AArch64中的VBAR_EL1寄存器），然后开启中断，这样便完成了异常机制的初始化。

Caution

有些情况在x86架构中会触发异常，在ARM架构中则不会。例如在进行整数除法时，如果除数是零，就会引入通用寄存器无法表示的无穷大量或无穷小量，因此处理器往往会对除数为零的情况进行特别处理。在x86架构中，除零是异常情况，会下陷到操作系统进行处理；而在AArch64架构中，除零的结果还是零，被视为有效计算。在设计操作系统时，应当考虑到此类硬件设计细节上的区别。

3、将目光投向ChCore

经过我们前置知识的铺垫，我们了解到，异常向量表定义了操作系统的入口，因此异常向量表的初始化是启动后立即要做的重要步骤。

在计算机启动时，CPU执行的第一段代码其实并非ChCore的代码。以在树莓派上运行ChCore为例，在加电后，**主板中的固件和SD卡中的bootloader**将先后被加载到内存并运行，进行基本的初始化工作。之后，ChCore的二进制文件将被bootloader加载到内存中约定好的位置，这时才从头开始执行ChCore的代码。

这里我们的读者可能会疑惑：bootloader是什么？有什么作用呢？

Info

Bootloader就像是电脑的“**启动助手**”，它的工作是帮助电脑从关机状态启动到操作系统。想象一下，你有一个复杂的拼图游戏，Bootloader就是那个帮你把拼图的第一块放好的人。

Info

Bootloader的作用：Bootloader的主要任务是从存储设备（硬盘、SD卡）中加载操作系统到内存中，并开始执行操作系统的代码。它确保操作系统能够顺利启动。

Info

Bootloader的例子：在树莓派上，Bootloader就是那个从SD卡中读取ChCore操作系统的小程序。它把ChCore加载到内存中，然后告诉CPU从哪里开始执行ChCore的代码。

```

1 _start:
2 mrs x8, mpidr_el1
3 and x8, x8, #0xFF
4 cbz x8, primary
5
6 /* Code for secondary core */
7 ...
8 /* Init exception vector */
9 bl set_exception_vector
10 ...
11
12 primary:
13 /* Code for primary core */
14 /* init UART, Virtual Memory mapping in C */
15 ...
16 /* Init exception vector */
17 adr x0, el1_vector
18 msr vbar_el1,x0
19 ...
20
21 /* Exception Table */
22 el1_vector:
23 /* entry for other type of exception */
24...
25
26 /* entry for synchronous exception from ELO */
27 align 7 // 128 bytes for each entry
28 b sync_el0_64
29
30 /* entry for interrupt from ELO */
31 align 7 //128 bytes for each entry
32 b irq_el0_64
33
34 /* entry for other type of exception */
35 ...

```

代码片段所示，ChCore起始位置的代码为_start，这便是进入ChCore时执行的第一段代码。在多核机器中，所有核心都会同时开始执行_start，ChCore需要选择某个CPU核心作为主要核心（通常是第一个）来初始化操作系统，同时其他核心将被暂时阻塞。AArch64为我们提供了**MPIDR这个系统寄存器来获取当前CPU核心的编号。**

通过系统指令 mrs x8, mpidr_el1 来读取 MPIDR 的值，然后判断当前核心是否为主核。**主核进入初始化流程，其他核心等待主核完成初始化。**

这里主要关注其中对异常向量表的初始化。在上述代码中，异常向量表存放在el1_vector代表的位置。在AArch64中，异常向量表中的每一项都可以存放128字节的指令，但ChCore**只存放了一条跳转指令**，用于跳转到不同异常事件对应的处理函数。例 ，对于来源于用户态的同步异常，内核将跳转到sync_el0_64，并在其中保存通用寄存器，根据异常的详细信息进行不同的处理；对于异步异常（即中断），内核将跳转到irq_el0_64，并执行相应的处理。

在编译ChCore时，异常向量表的内容已按照AArch64的布局在二进制文件中构造好了。在启动阶段，异常向量表便作为二进制文件的一部分被bootloader直接加载到内存中。因此，进入内

核后，ChCore只需要将异常向量表的起始地址放入指定的系统寄存器，便可完成异常向量表的初始化工作。

上述代码中，ChCore将el1_vector代表的异常向量表起始地址，通过msr指令存储到VBAR_ELI寄存器中。之后发生异常事件时，处理器便能跳转到VBAR_ELI寄存器指向的异常向量表中对应的表项，并跳转到操作系统内核进行处理。

4、系统调用的优化

- 在上文中，我们讲述了通过异常机制实现系统调用，但是这种方法实现的系统调用需要执行特权级别切换、上下文保存等操作，十分费时。何绕过费时的异常机制来实现系统调用呢？
- 第一种方法是**将一部分数据以只读的形式共享给应用**，但如果系统调用需要修改内核中的变量或者在运行过程中需要读取更多内核数据，这种方法就不适用了。
- 第二种方法是应用以“向某一内存页写入请求”的方式发起系统调用，并通过**轮询**等待系统调用完成，内核同样通过轮询来等待用户的请求，然后执行系统调用，并将系统调用的返回值写入同一块内存页以表示完成。
- 关键点在于：**内核独占一个CPU核心**，这个核心一直在内核态运行，而其他CPU核心一直在用户态运行。

面临的问题及解决方法

- (1) 何处理顺序处理时延问题？——**让多个CPU核心同时运行在内核态并轮询用户的请求**
- (2) 只有一个CPU核心时该怎么办？——轮询改为**批处理**。即当CPU运行在用户态时，应用程序一次发起多个系统调用请求，同样将请求和参数写入共享内存页，然后CPU切换到内核态，内核一次性将所有系统调用处理完，把结果写入共享内存页，再切换回用户态运行。

批处理的优点不在于时延，而在于**系统的整体吞吐率**：即降低系统整体的特权级的切换次数从而增加单位时间内的有效工作量。

二、ChCore虚拟内存管理

Tip

推荐阅读：4.1-4.3 虚拟内存管理

1、CPU的职责：内存地址翻译

1、地址翻译

CPU中的**内存管理单元(MMU)**负责地址翻译：应用进程在CPU核心上运行期间，使用的虚拟地址会由MMU进行翻译。当需要访问物理内存时，MMU翻译出来的物理地址将会通过总线传到物理内存，从而完成物理内存读写请求。

现代CPU中常包含转址旁路缓存(Translation Lookaside Buffer,TLB)作为加速地址翻译的部件(TLB是MMU 内部的硬件单元）。

2、分页机制

页表：操作系统为每个应用进程构造的一张记录从虚拟页到物理页的映射关系表；

页表基址寄存器：存储页表的起始地址，在AArch64平台中称为TTBR。

在页表机制下，每个虚拟地址由两部分组成：第一部分标识虚拟地址的虚拟页号；第二部分标识虚拟地址的页内偏移。在地址翻译过程中，MMU首先解析得到虚拟地址中的虚拟页号，并通过虚拟页号去该应用进程的页表中找到对应条目，然后取出条目中存储的物理页号，最后用该物理页号对应的物理页起始地址加上虚拟地址的页内偏移，得到最终的物理地址。

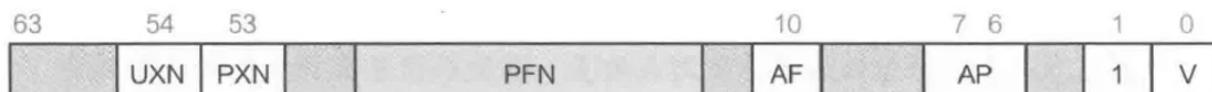
3、多级页表

为压缩页表大小，MMU采用了多级页表。前面提到，在使用简单的单级页表时，一个虚拟地址将被划分为两部分——虚拟页号和页内偏移。当使用k级页表时，一个虚拟地址的虚拟页号将被进一步地划分成k个部分（虚拟页号0, ……, 虚拟页号 i , $0 \leq i < k$ ），其中虚拟页号 i 对应该虚拟地址在第 i 级页表中的索引。当任意一级页表中的某一个条目为空时，该条目对应的下一级页表不需要存在，依次类推，接下来的页表同样不需要存在。因此，多级页表的设计极大减少了页表占用的空间大小。换句话说，多级页表允许整个页表结构中出现“空洞”，而单级页表则需要每一项都实际存在。通常，应用进程的虚拟地址空间中绝大部分的虚拟地址都不会被使用，所以多级页表通常具有很多“空洞”，从而能够极大地节约所占空间。

AArch64体系结构下一般为虚拟地址低48位参与地址翻译，页表级数为4级，虚拟页大小为4KB。

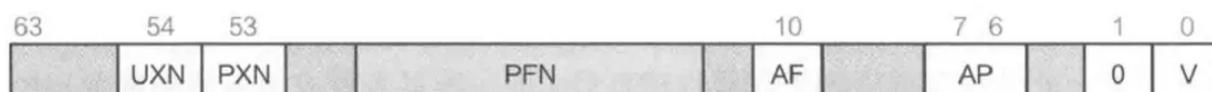
4、页表项与大页

中间级的页表项直接指向物理页时，其指向的是大页（比下一级页表项指向的物理页大小更大）。此外页表项也可以存储一些属性位，允许操作系统设置读写等权限。若实际访问所需权限和页表项中设置的权限不一致，则MMU会在地址翻译中出发访问异常。下图为页表项格式



页描述符：指向4K页

3级页表项



块描述符：指向大页



表描述符：指向下一级页表

0级、1级、2级页表项

5、TLB：页表的缓存

为了减少多级页表下地址翻译过程中的访存次数，MMU 引入转址旁路缓存

(TranslationLookaside Buffer, TLB) 部件来加速地址翻译的过程。具体来说，TLB缓存了虚拟页号到物理页号的映射关系。我们可以把 TLB 简化成存储着键值对的哈希表，其中键是虚拟页号，值是物理页号。MMU 会先把虚拟页号作为键去查询 TLB中的缓存项，若找到则可直接获得对应的物理页号而无须再查询页表。我们称通过TLB能够直接完成地址翻译的过程为TLB命中 (TLB Hit)，反之为TLB不命中 (TLB Miss)。

2、操作系统的职责：管理页表映射

CPU在上电启动后会默认使用物理地址，这是因为MMU的地址翻译功能还未开启，而操作系统则负责在初始化过程中启用该功能。一旦启用MMU地址翻译，CPU会根据页表对指令执行中涉及的地址进行翻译，即认为这些地址都是虚拟地址，因而操作系统和应用进程在后续运行中都是使用虚拟地址。因此，操作系統除了需要为每个应用进程设置页表外，也需要为自己配置页表。

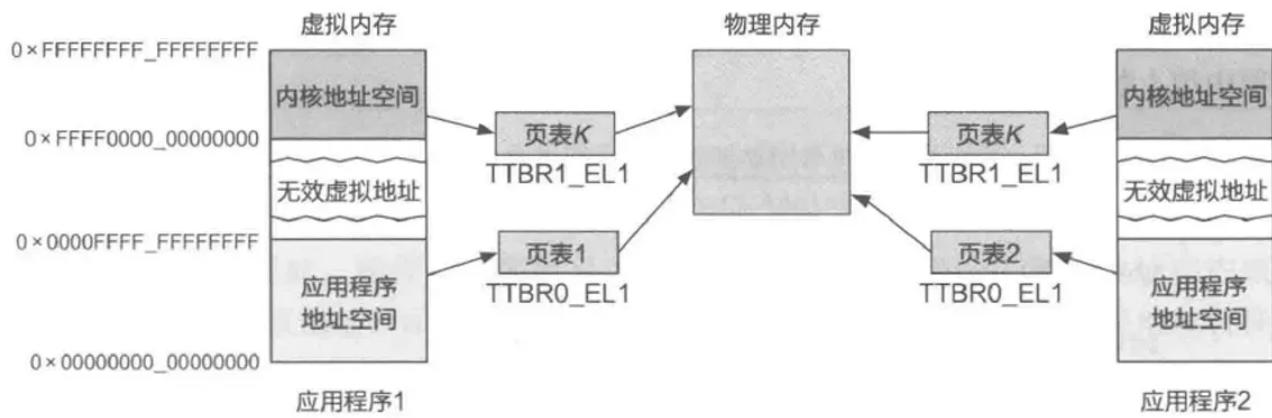
通常，操作系统为自己配置的页表具有两个特点：

第一，操作系统一般使用高虚拟地址，比 对于AArch64来说，操作系统使用高16位为1的虚拟地址；应用进程使用低虚拟地址，在AArch64中即为高16位为0的虚拟地址。

第二，操作系統一般会一次性将全部物理内存映射到虚拟地址空间中。映射方式为直接映射

(Direct Mapping)，即虚拟地址=物理地址+固定偏移。操作系統所使用的虚拟地址空间也称为**内核地址空间**。通过上述固定偏移的页表映射方式，操作系統能够在内核地址空间中很方便地在物理地址和虚拟地址之间进行转换。当操作系統需要访问一个物理地址时，仅需要访问该物理地址加上固定偏移的虚拟地址即可。

下图为AArch64体系结构下的操作系统页表和应用进程页表



3、ChCore虚拟内存管理

1、ChCore内核页表初始化

由于需要为应用程序提供虚拟内存抽象，操作系统启动期间会启用MMU。在启用后，MMU也会对操作系统执行期间使用的地址进行翻译，因此操作系统在启用MMU之前需要首先初始化自己的页表。

通常，操作系统对页表的配置分为两个阶段：第一阶段是启动初期，此时操作系统运行在低地址区域，页表将虚拟地址映射为完全相同的物理地址（VA=PA）；第二阶段是在跳转到高地址运行后，页表将虚拟地址映射为物理地址加上固定偏移（VA=PA+偏移量）。

```
1 BEGIN_FUNC(_start)
2 //内核运行的第一条指令：获取CPUID
3 mrs x8, mpidr_el1
4 and x8, x8, #0xFF
5 //若当前是0号CPU，则跳转到主CPU初始化处执行
6 cbz x8, primary
7 ...
8
9 primary:
10
11 //设置特权级别为EL1，即进入内核态执行
12 bl change_el_to_el1
13
14 //设置内核启动期间的栈桢
15 adr x0, boot_cpu_stack
16 add x0, x0, #INIT_STACK_SIZE
17 mov sp, x0
18
19 //调用C代码编写的初始化函数
20 bl init
21
22 //init函数不返回，控制流不会到这里
23 b .
24 END_FUNC(_start)
25
26
27 void init(void)
28{
29 ...
30
31 //初始化内核页表
32 init_kernel_pt();
33
34 //启用MMU
35 el1_mmu_activate();
36
37 //该函数位于高地址，且不返回
38 do_other_initialization();
39
40 //控制流不会到这里
41 }
```

延续上一节的代码，当判断为主核时，进入主核的初始化流程。

ChCore第一步会先把CPU特权级别设置为EL1（对应内核态），然后设置栈桢并调用C代码编写的初始化函数。

现在我们将目光放到下面的init函数上。在该函数中，ChCore首先配置自己的页表(init_kernel_pt函数)，其中会配置两份页表(boot_ttbr0_10负责低地址范围的映射，而boot_ttbr1_10对应高地址范围的映射)，然后启用MMU。

操作系统启动时第一条指令一般在较低的地址，这是来自硬件/固件的要求。应用程序通常使

用低地址，而操作系统使用高地址，所以操作系统启动期间需要跳转到高地址运行。因而ChCore在启用MMU后，会跳转到高地址继续执行剩余的初始化操作，此后将使用TTBR1_EL1中存储的页表进行翻译。

Hint

对于高地址范围（TTBR1_EL1对应的翻译），操作系统通常选择固定偏移（虚拟地址等于物理地址加上固定偏移）的映射方式，原因有两个：第一，两条在物理地址上相邻存储但跨越两个物理页的指令，要求对应的虚拟页也相邻，固定偏移映射是能满足该要求的最简单的方式；第二，操作系统通过简单的算术运算即可完成虚拟地址和物理地址之间的转换。

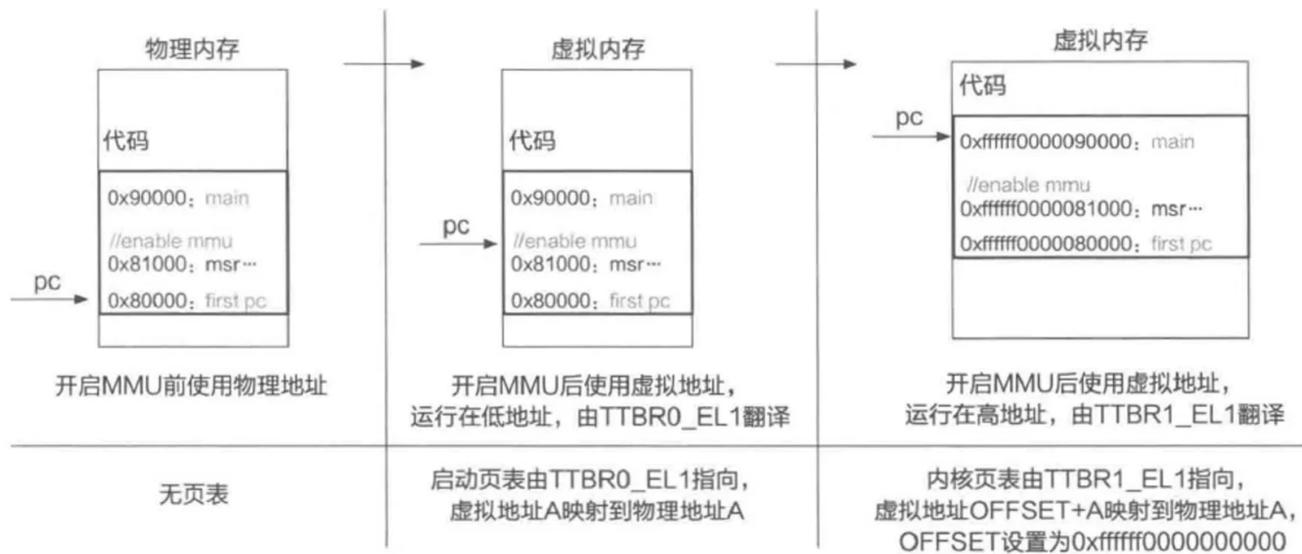
```
1 #define SCTRLR_EL1M (1)
2
3 BEGIN_FUNC(el1_mmu_activate)
4 ...
5
6 //设置页表基址寄存器
7 adrp x8, boot_ttbr0_10
8 msr ttbroel1, x8
9 adrp x8, boot_ttbr110
10 msr ttbr1el1, x8
11 ...
12
13 mrs x8, sctrlr_el1
14 ...
15 orr x8, x8, #SCTRLR_EL1_M
16 msr sctrlr_el1, x8
17 isb
18
19 ...
20 END_FUNC(el1_mmu_activate)
```

上面这段代码展示了何启用MMU，el1_mmu_activate函数首先在两个页表基址寄存器中写入页表基址，然后将系统寄存器sctrlr_el1的第0位设置为1，即SCTRLR_EL1_M位，从而开启MMU。

注意，从启动第一条指令到el1_mmu_activate函数中的isb指令之间，MMU都未开启，ChCore使用的是物理地址；而从isb指令开始，由于MMU已经开启，指令地址也会经过地址翻译。假设isb指令的上一条msr指令所在的物理地址是0x81000（即存放在物理内存中的位置），那么isb指令的物理地址则是0x81004。

CPU在执行msr指令时，PC中存放的指令地址是0x81000，由于MMU尚未启用，CPU是通过物理地址0x81000取出该指令的。CPU执行完msr指令后，PC中存放的指令地址是0x81004（准备执行isb指令），由于MMU已经启用，CPU中的MMU会对PC中的地址进行地址翻译（把0x81004作为虚拟地址），而isb指令的物理地址是0x81004，所以页表中的地址映射需要虚拟地址与物理地址完全相同。

下图展示了ChCore的启动过程：先使用物理过程，在使用经由启动页表翻译的虚拟地址，之后使用经由内核页表翻译的虚拟地址。



2、ChCore内存管理

```

1 struct vmspace
2 //vmregion链表
3 struct list_head vmr_list;
4 //页表基地址（虚拟地址）
5 vaddr_t *pgtbl;
6 //用于修改地址空间的并发控制
7 struct lock vmspace_lock
8
9 ...
10 };

```

上述代码片段给出了虚拟地址空间vmspace结构体中的三个成员变量。

vmr_list是链表头，指向一个由虚拟地址区域组成的链表。应用进程的虚拟空间最大可达 2^{48} ，但是往往其中大部分都是未被使用的区域。ChCore将一个应用地址空间内**有效使用**的虚拟地址区域保存在该链表里。

gtbl存储着这个虚拟地址空间对应的页表基地址。需要注意的是，pgtbl存储的地址是虚拟地址，由操作系统使用；对应的物理地址填在页表基址寄存器中，由MMU使用。ChCore在映射内核地址空间的时候，会保证内核虚拟地址和真正映射的物理地址之间相差一个固定的偏移量（直接映射）。因此，ChCore很容易根据pgtbl这个虚拟地址计算出真正应该写入页表基址寄存器中的物理地址（在vmspace中记录页表基地址的物理地址也是可行的，因为直接映射使虚拟地址和物理地址之间的转换变得很方便）。

```
1 struct vmregion {  
2 //链表中的一个节点，存储着prev和next指针  
3 struct list_head node;  
4 //区域起始地址  
5 vaddr_t start;  
6 //区域大小  
7 size_t size;  
8 //访问权限  
9 vmr_prop_t perm;  
10 //对应的物理内存对象  
11 struct pmo *pmo;  
12 };
```

上述代码给出了虚拟地址区域vmregion结构体中的成员变量。其中，成员变量node将不同的vmregion对象连成链表，成员变量start和size分别标识该虚拟内存区域的起始地址和区域大小，成员变量perm标识该区域的读、写、执行权限，最后一个成员变量pmo标识映射到该区域的物理内存对象。实验内核采用简单的映射关系，只有一个物理内存对象映射到一个虚拟内存区域。

ChCore实验内核以对象的形式进行物理内存资源的管理。物理内存对象pmobject分为不同的类型，由type成员标识，包含以下类型。

Info

PMO—DATA：课程实验中常用的类型，表示一段连续物理内存区域，即连续的物理页。在该类型中，成员变量start和size分别表示对应物理内存区域的起始地址和总大小。创建该类型的对象会立即进行物理内存分配，该类型的对象通常被用来映射给虚拟地址空间中的代码和数据区域，映射后不会发生缺页异常。

PMO—ANONYM：该类型的物理内存对象同样表示物理页的集合，但是不要求物理型页连续。分配该类型的对象不会立即触发物理内存分配，而是在访问时通过缺页型异常进行按需分配，其中成员变量radix将记录所有分配的物理页。虚拟地址空间中的堆区域适合映射该类型的物理内存对象。

PMO—SHM：该类型的物理内存对象用于实现应用进程之间的共享内存，即不同虚拟地址空间中的虚拟内存区域可以映射到同一个物理内存对象。成员变量refcnt表示对象的引用计数，当计数为0时，即可回收对应的物理内存资源。

PMO—USER_PAGER：表示对应的虚拟内存区域由用户态进行管理。

Success

至此，教材附加阅读内容部分到此结束，希望能对你的学习有所裨益！

拓展阅读与思考

前情提要

Lab 1 主要介绍Chcore 启动过程中的 工作，这些工作包括 启动cpu 0号核，切换异常级别，跳转到第一行C代码， 初始化串口输出，启用MMU，配置页表。

而在其他的系统之中，有不少的异同点，在下文之中我们将选取几个代表性的系统做对比分析

下文大纲

1. 我们将把 SJTU Chcore 系统 与 MIT xv6 系统进行比较，分析内核启动的相通点与不同点 并探讨可能的问题。
2. 我们将基于 Linux 介绍执行系统内核代码前，bootloader相关的粗略工作。

正文

Part 1 : Chcore 和 XV6 的对比

Part 1.1：基本分析

我们先看一下 xv6 book 关于启动过程中的工作：

To make xv6 more concrete, we'll outline how the kernel starts and runs the first process. The subsequent chapters will describe the mechanisms that show up in this overview in more detail.

When the RISC-V computer powers on, it initializes itself and runs a boot loader which is stored in read-only memory. The boot loader loads the xv6 kernel into memory. Then, in machine mode, the CPU executes xv6 starting at `_entry` (`kernel/entry.S:6`). The RISC-V starts with paging hardware disabled: virtual addresses map directly to physical addresses.

The loader loads the xv6 kernel into memory at physical address `0x80000000`. The reason it places the kernel at `0x80000000` rather than `0x0` is because the address range `0x0:0x80000000` contains I/O devices.

The instructions at `_entry` set up a stack so that xv6 can run C code. Xv6 declares space for an initial stack, `stack0`, in the file `start.c` (`kernel/start.c:11`). The code at `_entry` loads the stack pointer register `sp` with the address `stack0+4096`, the top of the stack, because the stack on RISC-V grows down. Now that the kernel has a stack, `_entry` calls into C code at `start` (`kernel/start.c:21`).

我们先不讨论内核代码在加载到内存之前系统做的工作，我们首先比较两个系统第一步执行内核代码的不同之处。对于Xv6而言，它的第一步是直接建立stack。而由Chcore文档我们知道进入内核第一步是通过一系列代码只让一个CPU Core工作，而其他CPU Core停止执行初始化工作 然后这两个CPU core切换异常级别接下来才是建立stack。那为什么xv6 会直接建立stack？我们接着看代码。两个系统在建立了stack都跳转到内核的第一行C代码进行初始化。

我们先看一下 xv6 book 关于这个过程中的工作：

The function `start` performs some configuration that is only allowed in machine mode, and then switches to supervisor mode. To enter supervisor mode, RISC-V provides the instruction `mret`. This instruction is most often used to return from a previous call from supervisor mode to machine mode. `start` isn't returning from such a call, and instead sets things up as if there had been one: it sets the previous privilege mode to supervisor in the register `mstatus`, it sets the return address to `main` by writing `main`'s address into the register `mepr`, disables virtual address translation in supervisor mode by writing `0` into the page-table register `satp`, and delegates all interrupts and exceptions to supervisor mode.

Before jumping into supervisor mode, `start` performs one more task: it programs the clock chip to generate timer interrupts. With this housekeeping out of the way, `start` "returns" to supervisor mode by calling `mret`. This causes the program counter to change to `main` (`kernel/main.c:11`).

After `main` (`kernel/main.c:11`) initializes several devices and subsystems, it creates the first process by calling `userinit` (`kernel/proc.c:212`). The first process executes a small program written

第一句话解释了为什么xv6不是在建立stack之前切换异常级别：因为start级别中的代码只能在 machine mode中执行，执行完之后再切换到supervisor mode。 machine mode 相当于EL3， supervisor mode 相当于 EL2。我们接着看，我们使用一个指令切换到supervisor mode同时 start 函数结束后将会返回到main.c 中（细节自行了解）。 main.c 也是进行初始化工作，我们接着看代码：

```

#include "types.h"
#include "param.h"
#include "memlayout.h"
#include "riscv.h"
#include "defs.h"

volatile static int started = 0;

// start() jumps here in supervisor mode on all CPUs.
void
main()
{
    if(cpuid() == 0){
        consoleinit();
        printinit();
        printf("\n");
        printf("xv6 kernel is booting\n");
        printf("\n");
        kinit();           // physical page allocator
        kvminit();        // create kernel page table
        kvminithart();   // turn on paging
        procinit();       // process table
        trapinit();       // trap vectors
        trapinithart();  // install kernel trap vector
        plicinit();       // set up interrupt controller
        plicinithart();  // ask PLIC for device interrupts
        binit();          // buffer cache
        iinit();          // inode table
        fileinit();       // file table
        virtio_disk_init(); // emulated hard disk
        userinit();       // first user process
        __sync_synchronize();
        started = 1;
    } else {
        while(started == 0)
            ;
        __sync_synchronize();
        printf("hart %d starting\n", cpuid());
        kvminithart();   // turn on paging
        trapinithart();  // install kernel trap vector
        plicinithart();  // ask PLIC for device interrupts
    }

    scheduler();
}

```

main.c代码的结构好像似曾相识，仔细一看，这不就是和Chcore中CPU Core启动一样吗！到此我们发现了内核代码的consistent design。而在真实世界中linux也是如此但也更复杂（链接：<https://xinqiu.gitbooks.io/linux-inside-zh/content/Initialization/linux-initialization-4.html>，我也未明白linux内核的mode change故未放上链接）。

我们接着看 main.c 代码大体逻辑与Chcore相同：先初始化console（类比Chcore中的uart_init），然后便开始启用MMU并初始化页表。这部分只拓展Lab1相关部分故不分析后面部分。

我们接下来看Xv6的页表初始化（地址映射），Xv6 内◆◆◆映射相关代码代码与Chcore 中设置高地址页表思路相同，相关代码放在文章末尾。

Part 1.2: 一个问题

看完之后我们可能会有问题，虽然虽然两个系统都会进行以上CPU core启动顺序相关工作，但代码位置却并不相同，我们下面分析一下这是为什么。那我们便要好好看下Chcore Lab1文档中的思考题12（在一开始我们暂停了三个其他核心的执行，根据现有代码简要说明它们什么时候会恢复执行。思考为什么一开始只让 0 号核心执行初始化流程？）

我们先看代码：

```
#include <common/asm.h>

.extern arm64_elX_to_el1
.extern boot_cpu_stack
.extern secondary_boot_flag
.extern secondary_init_c
.extern clear_bss_flag
.extern init_c

BEGIN_FUNC(_start)
    mrs x8, mpidr_el1
    and x8, x8, #0xFF
    cbz x8, primary

    /* Wait for bss clear */
    wait_for_bss_clear:
        adr x0, clear_bss_flag
        ldr x1, [x0]
        cmp x1, #0
        bne wait_for_bss_clear

    /* Set cntkctl_el1 to enable cntvct_el0.
       * Enable it when you need to get current tick
       * at EL0, e.g. Running aarch64 ROS2 demos
    mov x10, 0b11
    msr cntkctl_el1, x10 */

    /* Turn to el1 from other exception levels. */
    bl arm64_elX_to_el1

    /* Prepare stack pointer and jump to C. */
    mov x1, #0x1000
    mul x1, x8, x1
    adr x0, boot_cpu_stack
    add x0, x0, x1
    add x0, x0, #0x1000
    mov sp, x0

wait_until_smp_enabled:
    /* CPU ID should be stored in x8 from the first line */
    mov x1, #8
    mul x2, x8, x1
    ldr x1, =secondary_boot_flag
    add x1, x1, x2
    ldr x3, [x1]
    cbz x3, wait_until_smp_enabled

    /* Set CPU id */
    mov x0, x8
    bl secondary_init_c

primary:
    /* Turn to el1 from other exception levels. */
    bl arm64_elX_to_el1

    /* Prepare stack pointer and jump to C. */
```

```
adr  x0, boot_cpu_stack
add  x0, x0, #0x1000
mov  sp, x0

bl  init_c

/* Should never be here */
b .

END_FUNC(_start)
```

以上代码有两部分我们应该注意：wait_for_bss_clear 和 wait_until_smp_enabled。当第一个CPU core执行到 init.c 的时候会调用clear_bss 将clear_bss_flag 设置为0 然后其他CPU core 会切换异常级别并建立stack但会停在wait_until_smp_enabled这部分。第一个CPU core 继续执行 init.c 中的代码（初始化串口输出，配置页表，启用MMU,然后调用start_kernel 跳转到 main.c ）第一个CPU Core 在main.c 中执行完时钟，调度器，锁的初始化后其他CPU Core 开始初始化。故答案应是在多核系统中，时钟、调度器和锁是多个核心共享的资源。在初始化时，果多个核心同时访问这些资源，可能会引发竞争条件或未定义的行为。

接着看XV6代码：

review : the order is entry.S -> init.c -> main.c

entry.S:

```
# qemu -kernel loads the kernel at 0x80000000
# and causes each hart (i.e. CPU) to jump there.
# kernel.ld causes the following code to
# be placed at 0x80000000.
```

.section .text

.global _entry

_entry:

```
# set up a stack for C.
# stack0 is declared in start.c,
# with a 4096-byte stack per CPU.
# sp = stack0 + (hartid * 4096)
la sp, stack0
li a0, 1024*4
csrr a1, mhartid
addi a1, a1, 1
mul a0, a0, a1
add sp, sp, a0
# jump to start() in start.c
call start
```

spin:

```
j spin
```

start.c:

```
// entry.S jumps here in machine mode on stack0.
```

void

start()

{

```
// set M Previous Privilege mode to Supervisor, for mret.
unsigned long x = r_mstatus();
x &= ~MSTATUS_MPP_MASK;
x |= MSTATUS_MPP_S;
w_mstatus(x);
```

```
// set M Exception Program Counter to main, for mret.
```

```
// requires gcc -mcmodel=medany
```

```
w_mepc((uint64)main);
```

```
// disable paging for now.
```

```
w_satp(0);
```

```
// delegate all interrupts and exceptions to supervisor mode.
```

```
w_medeleg(0xffff);
```

```
w_mideleg(0xffff);
```

```
w_sie(r_sie() | SIE_SEIE | SIE_STIE | SIE_SSIE);
```

```
// configure Physical Memory Protection to give supervisor mode
```

```
// access to all of physical memory.
```

```
w_pmpaddr0(0xfffffffffffffull);
```

```
w_pmpcfg0(0xf);
```

```
// ask for clock interrupts.
```

```
timerinit();
```

```
// keep each CPU's hartid in its tp register, for cpuid().  
int id = r_mhartid();  
w_tp(id);  
  
// switch to supervisor mode and jump to main().  
asm volatile("mret");  
}  
  
// ask each hart to generate timer interrupts.  
void  
timerinit()  
{  
    // enable supervisor-mode timer interrupts.  
    w_mie(r_mie() | MIE_STIE);  
  
    // enable the sstc extension (i.e. stimecmp).  
    w_menvcfg(r_menvcfg() | (1L << 63));  
  
    // allow supervisor to use stimecmp and time.  
    w_mcounteren(r_mcounteren() | 2);  
  
    // ask for the very first timer interrupt.  
    w_stimecmp(r_time() + 1000000);  
}
```

main.c:

```
#include "types.h"  
#include "param.h"  
#include "memlayout.h"  
#include "riscv.h"  
#include "defs.h"  
  
volatile static int started = 0;  
  
// start() jumps here in supervisor mode on all CPUs.  
void  
main()  
{  
    if(cpuid() == 0){  
        consoleinit();  
        printinit();  
        printf("\n");  
        printf("xv6 kernel is booting\n");  
        printf("\n");  
        kinit();           // physical page allocator  
        kvminit();        // create kernel page table  
        kvmminithart();   // turn on paging  
        procinit();        // process table  
        trapinit();        // trap vectors  
        trapminithart();  // install kernel trap vector  
        plicinit();        // set up interrupt controller  
        plicminithart();  // ask PLIC for device interrupts  
        binit();           // buffer cache  
        iinit();           // inode table  
        fileinit();        // file table
```

```
virtio_disk_init(); // emulated hard disk
userinit(); // first user process
__sync_synchronize();
started = 1;
} else {
while(started == 0)
;
__sync_synchronize();
printf("hart %d starting\n", cpuid());
kvminithart(); // turn on paging
trapinithart(); // install kernel trap vector
PLICinithart(); // ask PLIC for device interrupts
}

scheduler();
}
```

经以上分析后可知，在Chcore内核代码的start.S中设置CPU Core 启动顺序是因为BSS 和 时钟，调度器，锁的初始化。然后我们看Xv6 中的main.c 中的代码正好包含时钟，调度器，锁的初始化工作（Xv6 并没有与清理BSS相关的代码，故不考虑），故Xv6代码会在这里设置CPU Core 启动顺序 即通过 cpuid() == 0 与 while(started == 0)两个控制语句使得只有0号CPU Core 可以执行相关初始化并在初始化结束之后将started设置为1来让其他CPU Core并行开始初始化。而之前的entry.S 以及 init.c 中的代码是可以让CPU Core并行运行的，故没必要在之前设置CPU Core 启动顺序。

Part 1.3: 一个假设

按照Xv6的思路我们可以改写Chcore 代码：我们不必要在start.s 设置CPU Core 启动顺序，CPU Core可以并行进行切换异常级别，建立stack ，跳转到init.c 。init.c 中可以并行的代码依然放在init.c ,不能并行的放在main.c 中，然后main.c 采用Xv6 写法。反之我们可以改写Xv6。

以下是相关伪代码：

```
start.S:  
BEGIN_FUNC(_start)  
  
/* Turn to el1 from other exception levels. */  
bl arm64_elX_to_el1  
  
/* Prepare stack pointer and jump to C. */  
adr x0, boot_cpu_stack  
add x0, x0, #INIT_STACK_SIZE  
mov sp, x0  
  
b init_c  
  
/* Should never be here */  
b .  
END_FUNC(_start)
```

init.c:

```
void init_c(void)  
{  
    //maybe some other initialize work  
    early_uart_init();  
    uart_send_string("boot: init_c\r\n");  
  
    uart_send_string("[BOOT] Jump to kernel main\r\n");  
    start_kernel(secondary_boot_flag); //jump to main  
  
    /* Never reach here */  
}
```

main.c:

```
volatile static int started = 0;  
  
// start() jumps here in supervisor mode on all CPUs.  
void main()  
{  
    if(cpuid() == 0){  
        clear_bss();  
        init_kernel_pt();  
        el1_mmu_activate();  
        //other work is not listed here  
        started = 1;  
    } else {  
        while(started == 0)  
            ;  
        init_kernel_pt();  
        el1_mmu_activate();  
        //other work is not listed here  
    }  
}
```

需注意上述这是设计的想法，具体还与架构等底层有关，故上述想法可能错误。只是希望读者

可以从上述讲解中体会到操作系统设计的相通点并提出自己的思考。相关架构知识放在文章末尾。

Part 2 内核代码在被加载到内存之前到那些事

Part 2.1： BIOS之前的故事

由于Chcore文档并未过多讲述执行内核代码之前的事，我们将基于Linux讲述执行内核代码之前的工作，以下描述主要讲述大体流程，细节并未十分完善。

需要注意的一点是，内核代码很重要，但它并不是一开始就执行的，因为有一些非常基本的初始化工作要做，这主要与硬件有关。在Linux中，我们按下按钮，主板会从电池获取信号，然后启动CPU，CPU 则复位寄存器的所有数据，并设置每个寄存器的预定值，并且在 CPU 寄存器中定义了 下预定义数据：

```
IP          0xffff0
CS selector 0xf000
CS base     0xfffff0000
```

而且 $CS\ base + IP = 0xfffffffff0$ ，这个地方是 [复位向量\(Reset vector\)](#)。这是CPU在重置后期望执行的第一条指令的内存地址。它包含一个 [jump](#) 指令，这个指令通常指向BIOS入口点。

注：CPU将在实模式下执行，此模式下MMU并未启用，寻址方式为：`PhysicalAddress = Segment * 16 + Offset`。

Part 2.2： BIOS的故事

然后CPU会跳转到 BIOS 的入口，BIOS 开始工作（初始化和检查硬件和寻找可引导设备）

（BIOS 会检查配置并尝试根据设备在配置中的顺序找到设备中的引导扇区（通常是该设备的第一个扇区），而引导扇区始终包含主引导记录（MBR），MBR 通常有 446 个字节用于存储引导代码，64 个字节用于存储分区表，最后两个字节用于检查该设备是否为可引导设备），并且我们必须注意到，只要我们开始执行引导代码，BIOS 就已经将系统的控制权移交给引导代码了。

注：关于MBR的一些细节

果我们只考虑下面三者的相对顺序同时忽视其他可能的细节，通常disk分区大致 下：

- **扇区 0：存放 MBR（主引导记录）或 UEFI 引导程序。**
- **扇区 1 到 N：存放 核心映像，也就是操作系统的引导程序或内核映像。这个区域通常存储引导所需的文件，它在启动时会被加载到内存的高地址。**
- **扇区 N+1：开始存放 第一个可用分区，这个分区用于存储操作系统的文件系统，Windows 的 NTFS 或 Linux 的 EXT4 等。**

文章末尾将补充关于核心映像加载到内存的高地址的由来。

Part 2.3：引导代码

在现实世界中，要启动 Linux 系统，有多种引导程序可以选择。比 [GRUB 2](#) 和 [syslinux](#)。Linux 内核通过 [Boot protocol](#) 来定义应该 何实现引导程序。在这里我们只介绍 GRUB 2。现在我们已经选择一个可引导设备并将控制权移到引导扇区中的引导代码（通常称为 Boot.img），这个引导代码只做了一些非常基本的初始化工作，然后跳转到 GRUB 2 的核心映像并执行代码（将内核代码和文件系统驱动程序放入内存然后初始化控制台等），执行完代码后会调用 grub_menu_execute_entry 函数列出可用的系统供选择，当选择一个系统时，grub_menu_execute_entry 会调用 boot 命令来引导所选系统即将所选系统的内核代码加载到内存中，在内核被引导入内存后，内存使用情况将入下表所示：

通过阅读 kernel boot protocol，在内核被引导入内存后，内存使用情况将入下表所示：

	Protected-mode kernel
100000	+-----+
	I/O memory hole
0A0000	+-----+
	Reserved for BIOS Leave as much as possible unused
	~ ~
	Command line (Can also be below the X+10000 mark)
X+10000	+-----+
	Stack/heap For use by the kernel real-mode code.
X+08000	+-----+
	Kernel setup The kernel real-mode code.
	Kernel boot sector The kernel legacy boot sector.
X	+-----+
	Boot loader <- Boot sector entry point 0x7C00
001000	+-----+
	Reserved for MBR/BIOS
000800	+-----+
	Typically used by MBR
000600	+-----+
	BIOS use only
000000	+-----+

所以当 bootloader 完成任务，将执行权移交给 kernel，kernel 的代码从以下地址开始执行：

`0x1000 + X + sizeof(KernelBootSector) + 1`

个人以为应该是 `X + sizeof(KernelBootSector) + 1` 因为 X 已经是一个具体的物理地址了，不是一个偏移

上面的公式中， x 是 kernel bootsector 被引导入内存的位置。在我的机器上， x 的值是 $0x10000$ ，到这里，引导程序完成它的使命，并将控制权移交给了 Linux kernel。

Tip

本节参考链接：<https://xinqiu.gitbooks.io/linux-inside-zh/content/Booting/linux-bootstrap-1.html>

文章末尾补充内容

Part 3 :页表配置代码

Part 3.1:XV6 页表配置代码

```
// Make a direct-map page table for the kernel.
pagetable_t
kvmmake(void)
{
    pagetable_t kpgtbl;

    kpgtbl = (pagetable_t) kalloc();
    memset(kpgtbl, 0, PGSIZE);

    // uart registers
    kvmmmap(kpgtbl, UART0, UART0, PGSIZE, PTE_R | PTE_W);

    // virtio mmio disk interface
    kvmmmap(kpgtbl, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);

    // PLIC
    kvmmmap(kpgtbl, PLIC, PLIC, 0x4000000, PTE_R | PTE_W);

    // map kernel text executable and read-only.
    kvmmmap(kpgtbl, KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R | PTE_X);

    // map kernel data and the physical RAM we'll make use of.
    kvmmmap(kpgtbl, (uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext, PTE_R |
PTE_W);

    // map the trampoline for trap entry/exit to
    // the highest virtual address in the kernel.
    kvmmmap(kpgtbl, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);

    // allocate and map a kernel stack for each process.
    proc_mapstacks(kpgtbl);

    return kpgtbl;
}

// Initialize the one kernel_pagetable
void
kvminit(void)
{
    kernel_pagetable = kvmmake();
}

// add a mapping to the kernel page table.
// only used when booting.
// does not flush TLB or enable paging.
void
kvmmmap(pagetable_t kpgtbl, uint64 va, uint64 pa, uint64 sz, int perm)
{
    if(mappages(kpgtbl, va, sz, pa, perm) != 0)
        panic("kvmmmap");
}
```

Part 3.2:Chcore 页表配置代码

这部分内容可以参考源码解析部分的文章

Part 4: 架构补充知识

Part 4.1: 细粒度的汇编控制与编译器的自动优化

- **ARM架构的汇编:** ARM汇编指令在某些情况下比x86更加细粒度，特别是在多核并发编程中，对于缓存一致性和内存屏障的控制。ARM架构可能需要开发者在汇编层面显式地插入内存屏障指令，以确保不同线程之间的操作顺序（例 dmb、dsb等）。而在x86架构上，由于它的强一致性（即大多数情况下，内存操作的顺序已经得到硬件的保证），开发者通常无需关心指令的顺序，编译器和处理器会自动处理。
- **编译器支持:** 为了应对这些低级别的内存操作，编译器往往提供一些内置函数（`__sync_synchronize()`）来强制执行内存屏障操作，确保在多线程环境下不同线程之间的可见性和顺序性。在ARM平台上，编译器需要支持这些内置的同步原语，或者开发者需要手动插入相应的内存屏障指令。

Part 4.2: 内存模型：强一致性 vs. 弱一致性

- **x86的强一致性:** x86架构（尤其是较新版本）通常采用强一致性的内存模型，也就是说，指令的执行顺序对于程序员来说是更直观的，硬件会确保内存操作的顺序与程序代码的顺序尽量一致。在这种架构下，许多并发相关的问题（例 数据竞争）能够通过简单的同步原语或内存屏障得到有效解决。
- **ARM的弱一致性:** 与x86不同，ARM架构通常采用弱一致性模型，这意味着在没有显式同步指令的情况下，**处理器的内存访问可能会乱序执行，这对于多核处理器尤其重要。**在ARM上，果你不显式地插入内存屏障，可能会遇到数据的"可见性"问题。例，某些写操作可能不会立即对其他核心可见，从而导致程序出现潜在的并发错误。因此，ARM汇编通常需要开发者更加关注 何通过内存屏障指令来保证不同线程之间的操作顺序。

Part 4.3: `__sync_synchronize()`与兼容性

- `__sync_synchronize()`是一个编译器内置的内存屏障函数，主要用来确保在它之前的所有操作在它之后的操作之前完成。它可以帮助开发者避免一些由于内存重排引起的问题。在x86架构上，这个函数通常会被翻译成适当的内存屏障指令（`mfence`或`sfence`）。然而，ARM架构可能会有不同的实现方式，依赖于不同的编译器（GCC或Clang）以及ARM的具体实现（例 v7、v8或更高版本）来选择合适的指令（例 dmb、dsb）。
- 果编译器没有自动插入正确的同步原语，开发者可能需要手动插入ARM汇编指令来保证多线程同步。这就涉及到开发者对平台底层细节的了解，特别是在不同架构上实现内存屏障的细节。

Part 5: 核心映像加载到内存的高地址的由来

Part 5.1: 内核映像加载位置的历史变化

- **早期的内核加载位置：**在早期的 Linux 系统中，内核映像通常被加载到 **低地址**，大约在 1MB 左右的地方。这是因为当时的硬件架构（比 x86）内存较小，CPU 的物理地址空间和虚◆◆◆地址空间通常都相对较小，且操作系统通常从较低的内存地址开始加载。内核从低内存加载的方式也较为简单，因为这时候没有复杂的内存管理机制。
- **现代 Linux 内核加载位置：**随着硬件技术的发展，尤其是 64 位系统的普及和内存容量的增大，现代的 Linux 系统通常将内核加载到 **高地址**。这种做法使得内核能够充分利用现代处理器的 **虚拟内存机制**，同时也能避免与用户空间的内存发生冲突。具体的高地址位置通常是由硬件架构、内核配置以及引导加载程序来决定。

Part 5.2: 内核加载位置的当前做法

在 **现代 64 位 Linux 系统** 中，内核映像通常被加载到一个 **高位地址**，例：

- 在 **x86_64** 架构上，内核通常被加载到 **0x1000000 (16MB)** 或更高的位置。这个位置是在内存的 **高端**，远离用户空间和其他内核空间，避免了地址冲突，并允许操作系统更灵活地管理内存。
- 在 **ARM64** 或其他架构上，内核的加载位置也类似，通常会在系统启动时由引导程序（**U-Boot**）和系统固件（**UEFI**）决定，并根据内存大小和硬件要求来分配。

Part 5.3: 为什么从低地址到高地址变化？

- **支持更大的内存空间：**现代的计算机通常具有大量的 RAM，使用高位地址加载内核可以让内核直接与硬件提供的高地址空间对接，避免内核与低位内存（用户空间和系统堆栈等）发生冲突。
- **虚拟内存支持：**现代系统的虚拟内存支持允许内核在虚拟地址空间中管理和映射更大的内存区域。高位地址加载使得内核可以在虚拟地址中灵活地划分内存，并减少碎片化问题。
- **更高效的内存管理：**将内核映像加载到高位地址有助于现代内存管理技术（**PAE**、**x86_64**、**NUMA** 等）能够更好地工作，尤其是在需要管理大内存和多核处理器的环境下。
- **避免地址冲突：**将内核映像加载到高位地址可以防止与用户空间的内存区域发生冲突。特别是对于需要动态分配内存的内核，这种方法有助于减少发生冲突的风险。

Part 5.4: Linux 内核加载位置的可配置性

虽然现代 Linux 系统的内核通常会被加载到高位地址，但这并不是固定不变的，也不是硬件强制的。实际上，Linux 内核的加载位置是可以通过引导程序来配置的。例：

- **引导加载程序（如 GRUB）** 会提供一个位置，用于将内核映像加载到内存中。这个位置可以根据需要进行配置，◆◆别是在某些嵌入式系统中，可能会有特殊的要求。
- **内核配置：**Linux 内核的配置中可以指定内核映像的加载地址。这通常是通过内核的启动选项或者配置文件来决定的。

- **物理和虚拟内存管理：**内核的加载地址与物理和虚拟内存的管理密切相关，可能会根据硬件架构、内存管理策略等不同而有所不同。

Part 5.5: Linux 文档中的内核加载地址

根据 **Linux 文档** 和内核源代码（ **Documentation/x86/boot.rst** 和 **arch/arm/boot/dts** 中的文档），现代系统中内核加载的地址是有标准的。例如，Linux 在启动时会读取引导加载程序指定的地址，并且根据系统的配置，使用一个默认的加载地址。

在 **x86_64** 架构中，常见的加载地址是 `0x1000000` (16MB)，而对于 **ARM64** 系统，内核加载的地址可能是 `0x40008000` (64MB)。但这也取决于引导加载程序和系统配置，具体情况可以通过内核的启动配置和引导程序的选项进行调整。

Part 6:自己动手

Tip

果你想要自己写boot sector 或 了解更多内容推荐[os-tutorial](#)，大概需要30mins - 45 mins。

Success

到这里，你已经完成了机器启动部分全部附加内容的学习！请继续保持你的学习热情和探索精神，ChCore的世界还等待你去探索。

Last change: 2025-02-14, commit: [9fa9752](#)

Last change: 2025-02-14, commit: [23a4605](#)

源码解析

这部分内容是对内存管理部分的chcore源代码的详细解析，包括物理内存管理与虚拟内存管理两个部分。

Important

完成 Lab2 后，再阅读这部分内容。

Last change: 2025-02-14, commit: [9fa9752](#)

Buddy System 伙伴系统

- 回顾：为什么需要伙伴系统
 - 外部碎片与内部碎片
 - 伙伴系统的设计理念
- Chcore伙伴系统的设计
 - 核心数据结构
 - page
 - phys_mem_pool
 - 核心函数接口
- Chcore源码对伙伴系统的具体实现
 - init_buddy
 - get_buddy_chunk
 - buddy_get_pages
 - buddy_free_pages
 - spilt_chunk
 - merge_chunk

SLab分配器

- 回顾：SLab分配器的设计理念
 - 基本结构
 - SLab的内部组织
- Chcore中SLab分配器的设计
 - 核心数据结构
 - slab_pointer
 - slab_header
 - slab_slot_list
 - 核心函数接口
- Chcore源码对SLab功能的具体实现
 - init_slab
 - alloc_in_slab
 - free_in_slab

Kmalloc & Kfree

- kmalloc
- kfree

Buddy System 伙伴系统

Buddy System，即**伙伴系统**，被广泛应用于分配连续的物理内存页。本节内容将：

- 回顾伙伴系统的基础知识与设计理念

- 分析Chcore源码对伙伴系统的代码实现与架构
- 解析Chcore源码对伙伴系统操作相关函数的具体实现

为什么需要伙伴系统

面对进程多种多样的内存分配请求，自然需要一个精良的分配器设计来承担这样宏伟的责任。许多分配器无不在**内存碎片（无法被利用的内存）的困扰下捉襟见肘，而伙伴系统则在一定程度上解决了普通简易分配器会带来的内存碎片问题，尤其是在分配连续内存的情况下。**

外部碎片与内部碎片

- 外部碎片：单个空闲内存部分小于分配请求的内存大小
- 内部碎片：分配内存大于实际内存导致的内存碎片

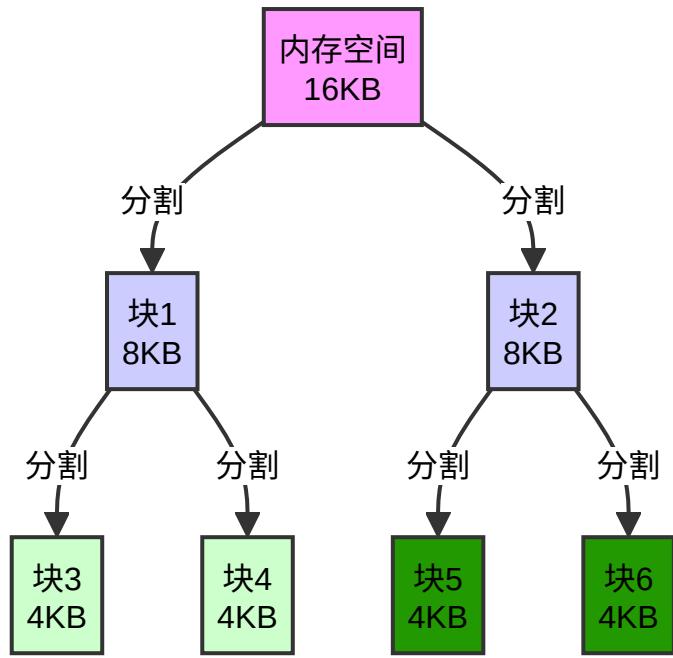
Buddy System将通过合并与级联的设计来避免这些困扰

伙伴系统的设计理念

伙伴系统设计理念即基于**伙伴块进行分裂与合并**，主要可概括为以下几点：

- 物理内存划为块为单位分配内存，每个块由连续物理页组成，大小为 2^n 个物理页（便于分裂与合并）
- 大块可根据分配需求分裂为小块，两个相同的小块可合并为大块，大块分裂出的两个小块即称作伙伴
- 分配时根据需求选择块，若这样的空闲块不存在，则找大块分裂；释放内存时查看其伙伴，若伙伴也空闲则直接合并

下图所示，相同颜色的块即表示伙伴块



由此一来，基于Buddy System的分配模式便可确保每时每刻的空闲块都是最大的，这样可以避免外部碎片；同时基于分配请求来寻找空闲块分配的操作又尽可能减小了内部碎片的大小。尽量避免了较大内存碎片的产生

Chcore伙伴系统的设计

核心数据结构

核心数据结构，即 `page` 和 `phys_mem_pool`，下所示为源码：

```
#define N_PHYS_MEM_POOLS 3

/* The following two are defined in mm.c and filled by mmparse.c. */
extern paddr_t physmem_map[N_PHYS_MEM_POOLS][2];
extern int physmem_map_num;

/* All the following symbols are only used locally in the mm module. */

/* `struct page` is the metadata of one physical 4k page. */
struct page {
    /* Free list */
    struct list_head node;
    /* Whether the correspond physical page is free now. */
    int allocated;
    /* The order of the memory chunck that this page belongs to. */
    int order;
    /* Used for ChCore slab allocator. */
    void *slab;
    /* The physical memory pool this page belongs to */
    struct phys_mem_pool *pool;
};

struct free_list {
    struct list_head free_list;
    unsigned long nr_free;
};

/*
 * Supported Order: [0, BUDDY_MAX_ORDER].
 * The max allocated size (continous physical memory size) is
 * 2^(BUDDY_MAX_ORDER - 1) * 4K.
 * Given BUDDY_MAX_ORDER is 14, the max allocated chunk is 32M.
 */
#define BUDDY_PAGE_SIZE      (0x1000)
#define BUDDY_MAX_ORDER       (14)

/* One page size is 4K, so the order is 12. */
#define BUDDY_PAGE_SIZE_ORDER (12)

/* Each physical memory chunk can be represented by one physical memory pool.
 */
struct phys_mem_pool {
    /*
     * The start virtual address (for used in kernel) of
     * this physical memory pool.
     */
    vaddr_t pool_start_addr;
    unsigned long pool_mem_size;

    /*
     * The start virtual address (for used in kernel) of
     * the metadata area of this pool.
     */
    struct page *page_metadata;

    /* One lock for one pool. */
}
```

```

struct lock buddy_lock;

/* The free list of different free-memory-chunk orders. */
struct free_list free_lists[BUDDY_MAX_ORDER];

/*
 * This field is only used in ChCore unit test.
 * The number of (4k) physical pages in this physical memory pool.
 */
unsigned long pool_phys_page_num;
};

/* Disjoint physical memory can be represented by several phys_mem_pools. */
extern struct phys_mem_pool global_mem[N_PHYS_MEM_POOLS];

```

page

在chcore中，一个 `page` 结构体即表示一个实际的4KB大小的物理内存页，我们来看它的具体成员组成：

- `struct list_head node`：是用于维护空闲链表的节点，当我们需要把这个页移入/移出空闲链表时，就会使用到这个节点
- `int allocated`：表示该页是否被分配
- `int order`：表示该页所属的块的阶数（即 2^n 的n）
- `struct phys_mem_pool *pool`：表示该页所隶属的内存池

另外还有slab，这个会在下文讲解slab分配器的时候具体解析

phys_mem_pool

chcore里的内存池即表示一整块连续的物理内存，也就是伙伴系统概念中所提到的整个空闲链表背后所属于的大的连续内存块。所以相应的，它就需要维护链表数组以及其所包含的所有页表元数据。我们还是来看看它的具体成员组成：

- `vaddr_t pool_start_addr`：内存池表示内存的起始地址
- `unsigned long pool_mem_size`：内存池表示的内存大小
- `struct page *page_metadata`：一个页表数组，包含了这个内存块里的所有页表
- `struct lock buddy_lock`：内存池的锁，用于并行化
- `struct free_list free_lists[BUDDY_MAX_ORDER]`：空闲链表数组，这里的order最大为14，故有14个这样的链表，可以依次表示4KB到32MB的内存

核心函数接口

即伙伴系统用于实际操作这些数据结构的函数，这里先大概介绍一下接口，下文再深入研究其实现：

```

void init_buddy(struct phys_mem_pool *, struct page *start_page,
                 vaddr_t start_addr, unsigned long page_num);

static struct page *get_buddy_chunk(struct phys_mem_pool *pool,
                                     struct page *chunk)

static struct page *split_chunk(struct phys_mem_pool *pool, int order,
                                 struct page *chunk)

static struct page *merge_chunk(struct phys_mem_pool *pool, struct page
*chunk)

struct page *buddy_get_pages(struct phys_mem_pool *, int order);

void buddy_free_pages(struct phys_mem_pool *, struct page *page);

void *page_to_virt(struct page *page);

struct page *virt_to_page(void* ptr);

unsigned long get_free_mem_size_from_buddy(struct phys_mem_pool *);

unsigned long get_total_mem_size_from_buddy(struct phys_mem_pool *);

```

从上到下的函数接口依次的功能为：

- `init_buddy`：初始化伙伴系统，包括其链表数组，页表元数据数组等
- `get_buddy_chunk`：根据指定内存池和内存块（的起始页表项）找到其伙伴块
- `split_chunk`：分裂内存块
- `merge_chunk`：合并伙伴内存块
- `buddy_get_pages`：查询并获取指定内存池和阶数的内存块
- `buddy_free_pages`：释放指定内存池的内存块
- `page_to_virt`：获取给定内存块（`page_metadata`）的虚拟地址，这是因为`page`类的指针指向的是 `page_metadata` 而非实际地址
- `virt_to_page`：根据给定虚拟地址找到相应的内存块（`page_metadata`），和上面的函数互为逆操作
- `get_free_mem_size_from_buddy`：获取给定内存池中当前可用的内存总量
- `get_total_mem_size_from_buddy`：获取给定内存池的总大小

Chcore源码对伙伴系统的具体实现

上一部分我们从上层的视角讲解了伙伴系统的原理以及Chcore中伙伴系统的架构，这一部分我们则从下层的视角，来分析上文提到的函数接口以及Buddy System核心功能的具体实现

本节主要着重分析对伙伴系统的初始化以及对伙伴块的操作函数的讲解，而受限于篇幅，辅助函数本处不做解析，感兴趣者可自行阅读

init_buddy

源码

```
/*
 * The layout of a phys_mem_pool:
 * | page_metadata are (an array of struct page) | alignment pad | usable
memory
 *
 * The usable memory: [pool_start_addr, pool_start_addr + pool_mem_size].
*/
void init_buddy(struct phys_mem_pool *pool, struct page *start_page,
                 vaddr_t start_addr, unsigned long page_num)
{
    int order;
    int page_idx;
    struct page *page;

    BUG_ON(lock_init(&pool->buddy_lock) != 0);

    /* Init the physical memory pool. */
    pool->pool_start_addr = start_addr;
    pool->page_metadata = start_page;
    pool->pool_mem_size = page_num * BUDDY_PAGE_SIZE;
    /* This field is for unit test only. */
    pool->pool_phys_page_num = page_num;

    /* Init the free lists */
    for (order = 0; order < BUDDY_MAX_ORDER; ++order) {
        pool->free_lists[order].nr_free = 0;
        init_list_head(&(pool->free_lists[order].free_list));
    }

    /* Clear the page_metadata area. */
    memset((char *)start_page, 0, page_num * sizeof(struct page));

    /* Init the page_metadata area. */
    for (page_idx = 0; page_idx < page_num; ++page_idx) {
        page = start_page + page_idx;
        page->allocated = 1;
        page->order = 0;
        page->pool = pool;
    }

    /* Put each physical memory page into the free lists. */
    for (page_idx = 0; page_idx < page_num; ++page_idx) {
        page = start_page + page_idx;
        buddy_free_pages(pool, page);
    }
}
```

解析

这个函数主要负责初始化我们的伙伴系统，它将接受的参数用于初始化一个特定的内存池，并为之设置相应页表数组和空闲链表数组等。你可能会好奇，这些参数本身又是怎么来的？这便涉及到对整个物理内存系统的初始化，可以移步 `mm.c` 文件的 `mm_init` 函数及相关辅助函数作进一步的学习

根据抽象屏障原则，我们这里只需要关心它是何时初始化一个内存池即可

我们根据注释分析得到实际上的内存池内存布局：

内存池的布局：

页面元数据区域 (struct page 数组)	对齐填充 (未使用)	实际可用内存区域 (用于分配的物理内存)

源码的工作可以分成三个阶段：

- 初始化物理内存池：根据传入的参数设置物理内存池的相应数据成员，这里的 `BUDDY_PAGE_SIZE` 是 `0x1000`，也即 `4KB`
- 初始化空闲链表：用一个 `for` 循环将每个阶的空闲链表均初始化，这里首先将每个链表的空闲数设为 `0`，同时用初始化函数 `init_list_head` 将这个链表节点的首尾都连到自己身上
- 初始化页表元数据：先用 `memset` 将这块内存全部清零；再用 `for` 循环溜一遍，利用指针运算初始化每个页表相应数据；最后再依次 `free` 每个物理页表，这里自然会涉及到后面的 `merge` 操作，并最终得到一个全部空闲的内存池

可以看见，`chcore` 中实际的设计和书上提到的基本一致，不过书上的介绍做了一定的简化

get_buddy_chunk

找呀找呀找朋友，找到一个伙伴块

源码

```

static struct page *get_buddy_chunk(struct phys_mem_pool *pool,
                                    struct page *chunk)
{
    vaddr_t chunk_addr;
    vaddr_t buddy_chunk_addr;
    int order;

    /* Get the address of the chunk. */
    chunk_addr = (vaddr_t)page_to_virt(chunk);
    order = chunk->order;
    /*
     * Calculate the address of the buddy chunk according to the address
     * relationship between buddies.
     */
    buddy_chunk_addr = chunk_addr
        ^ (1UL << (order + BUDDY_PAGE_SIZE_ORDER));

    /* Check whether the buddy_chunk_addr belongs to pool. */
    if ((buddy_chunk_addr < pool->pool_start_addr)
        || ((buddy_chunk_addr + (1 << order) * BUDDY_PAGE_SIZE)
            > (pool->pool_start_addr + pool->pool_mem_size))) {
        return NULL;
    }

    return virt_to_page((void *)buddy_chunk_addr);
}

```

解析

由伙伴块的定义可知，互为伙伴的两个块，它们的内存地址仅有一位不同，且该位由order决定
于是这块代码的核心便在于 何由位运算得到伙伴块的地址，首先，我们有：

块A的伙伴是块B = 块A的地址 ^ size
块B的伙伴是块A = 块B的地址 ^ size

于是我们可以由order和BUDDY_PAGE_SIZE_ORDER (这里是12，因为一个页的大小是4KB)
得到size的大小：

(1UL << (order + BUDDY_PAGE_SIZE_ORDER))

再做一个异或，就得到了相应的伙伴块地址。接下来的事情，就交给辅助函数！

buddy_get_pages

现在让我们看看一次完整的向伙伴系统申请内存的流程是怎样的！

源码

```

struct page *buddy_get_pages(struct phys_mem_pool *pool, int order)
{
    int cur_order;
    struct list_head *free_list;
    struct page *page = NULL;

    if (unlikely(order >= BUDDY_MAX_ORDER)) {
        kwarn("ChCore does not support allocating such too large "
              "contious physical memory\n");
        return NULL;
    }

    lock(&pool->buddy_lock);

    /* Search a chunk (with just enough size) in the free lists. */
    for (cur_order = order; cur_order < BUDDY_MAX_ORDER; ++cur_order) {
        free_list = &(pool->free_lists[cur_order].free_list);
        if (!list_empty(free_list)) {
            /* Get a free memory chunck from the free list */
            page = list_entry(free_list->next, struct page,
node);
            list_del(&page->node);
            pool->free_lists[cur_order].nr_free -= 1;
            page->allocated = 1;
            break;
        }
    }

    if (unlikely(page == NULL)) {
        kdebug("[OOM] No enough memory in memory pool %p\n", pool);
        goto out;
    }

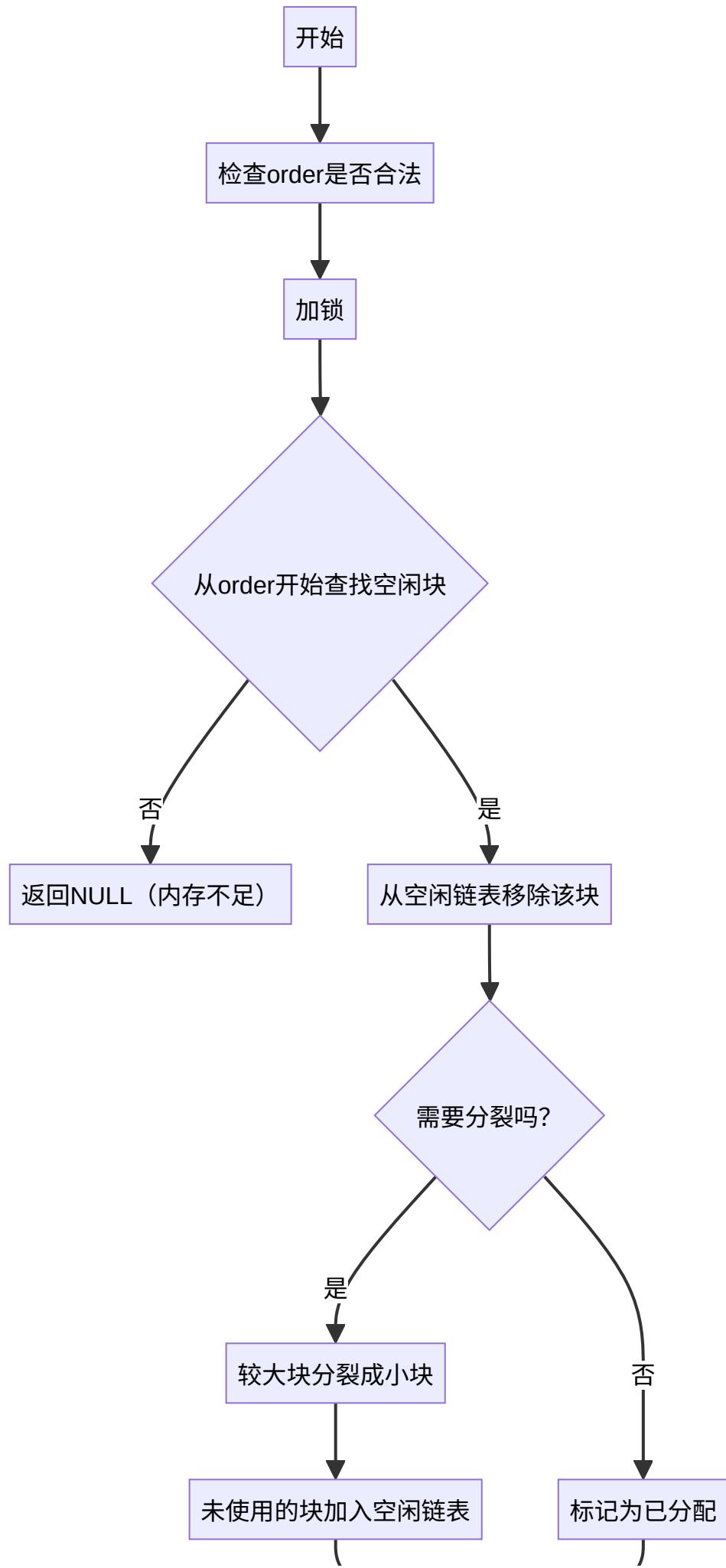
    /*
     * Split the chunk found and return the start part of the chunck
     * which can meet the required size.
     */
    page = split_chunk(pool, order, page);

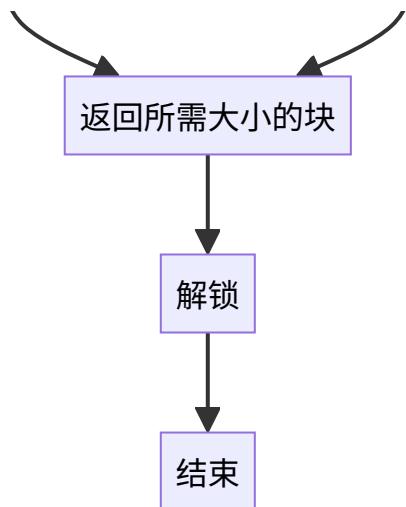
out:
    unlock(&pool->buddy_lock);
    return page;
}

```

解析

纵览代码，我们可以绘制该函数执行的流程图：





重点可以看看伙伴系统是如何查找满足要求的空闲块的：

```

/* Search a chunk (with just enough size) in the free lists. */
for (cur_order = order; cur_order < BUDDY_MAX_ORDER; ++cur_order) {
    free_list = &(pool->free_lists[cur_order].free_list);
    if (!list_empty(free_list)) {
        /* Get a free memory chunk from the free list */
        page = list_entry(free_list->next, struct page,
node);
        list_del(&page->node);
        pool->free_lists[cur_order].nr_free -= 1;
        page->allocated = 1;
        break;
    }
}
    
```

从查询要求提供的order开始，一级一级向上查找，如果没有则向上加一级，如果有则操作链表，取出这块空闲块，并同时设置空闲链表的相应参数，退出循环

一个可能的例子 下：

假设要申请 order=2 (16KB) 的内存：

1. 先查找 order=2 的空闲链表
2. 果没有，查找 order=3 的空闲链表
3. 果没有，继续查找更大的空闲链表
4. 直到找到可用的内存块或达到最大阶数

至于裂开的操作，我们马上就会讲到

buddy_free_pages

和get操作类似，我们现在来看看free操作是如何进行的

源码

```
void buddy_free_pages(struct phys_mem_pool *pool, struct page *page)
{
    int order;
    struct list_head *free_list;

    lock(&pool->buddy_lock);

    BUG_ON(page->allocated == 0);
    /* Mark the chunk @page as free. */
    page->allocated = 0;
    /* Merge the freed chunk. */
    page = merge_chunk(pool, page);

    /* Put the merged chunk into the its corresponding free list. */
    order = page->order;
    free_list = &(pool->free_lists[order].free_list);
    list_add(&page->node, free_list);
    pool->free_lists[order].nr_free += 1;

    unlock(&pool->buddy_lock);
}
```

解析

get还需要查询是否有空闲块，而free则简单多了——直接设置参数后merge即可！最后不要使用链表操作将这个块放进内存池的空闲链表里

同样的，加锁和去锁在这里也少不了，它们是并发安全性的保证

下面我们就讲解裂开和合并的具体实现，也是伙伴系统最关键的地方

spilt_chunk

源码

```

static struct page *split_chunk(struct phys_mem_pool *pool, int order,
                                struct page *chunk)
{
    struct page *buddy_chunk;
    struct list_head *free_list;

    /*
     * If the @chunk's order equals to the required order,
     * return this chunk.
     */
    if (chunk->order == order)
        return chunk;

    /*
     * If the current order is larger than the required order,
     * split the memory chunck into two halves.
     */
    chunk->order -= 1;

    buddy_chunk = get_buddy_chunk(pool, chunk);
    /* The buddy_chunk must exist since we are splitting a large chunk. */
    if (buddy_chunk == NULL) {
        BUG("buddy_chunk must exist");
        return NULL;
    }

    /* Set the metadata of the remaining buddy_chunk. */
    buddy_chunk->order = chunk->order;
    buddy_chunk->allocated = 0;

    /* Put the remaining buddy_chunk into its correspondint free list. */
    free_list = &(pool->free_lists[buddy_chunk->order].free_list);
    list_add(&buddy_chunk->node, free_list);
    pool->free_lists[buddy_chunk->order].nr_free += 1;

    /* Continue to split current chunk (@chunk). */
    return split_chunk(pool, order, chunk);
}

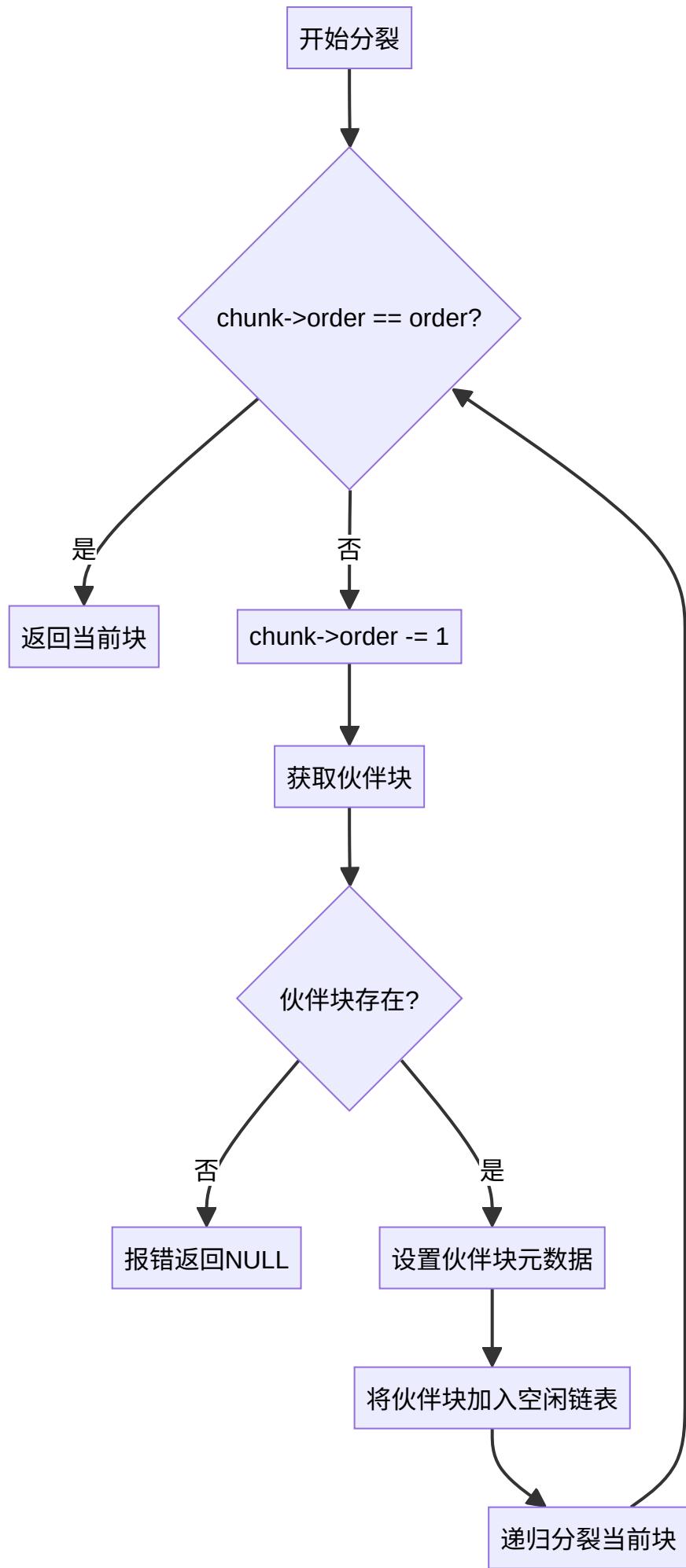
```

解析

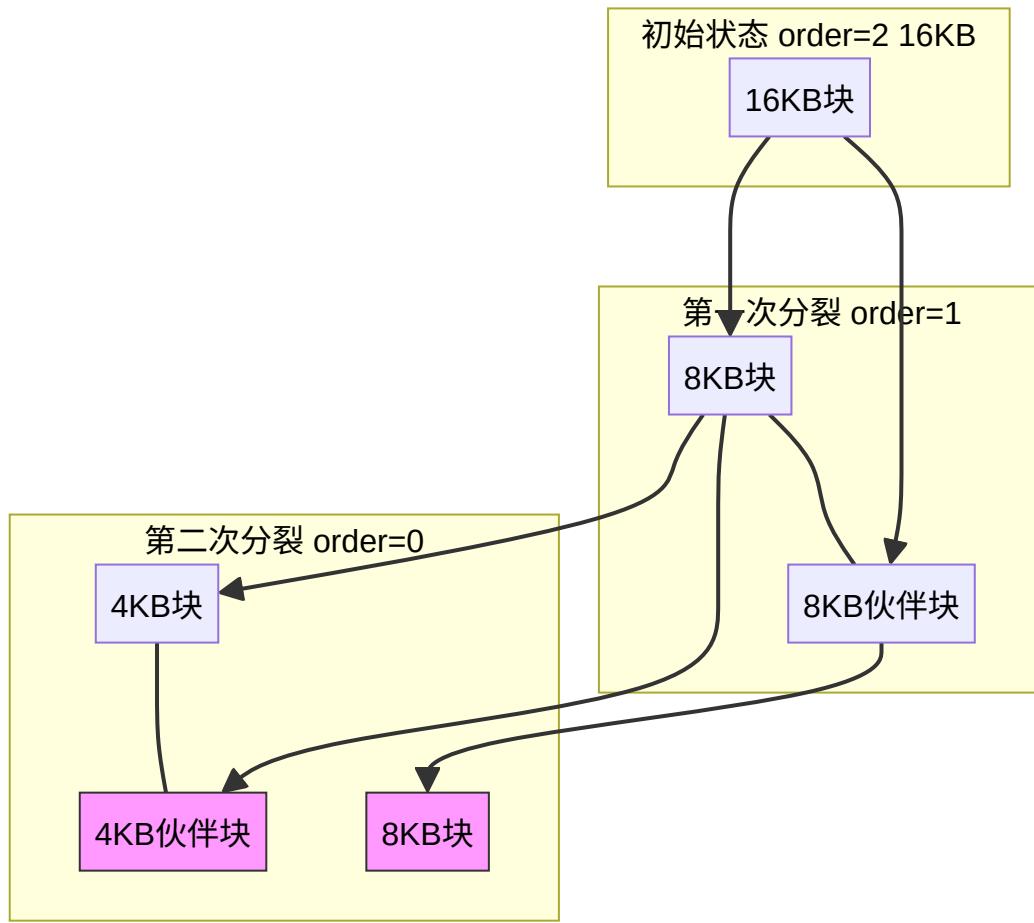
注意该函数接受的参数： order 指的是最后要得到的内存块的order，这一点也可以在上面申请内存的函数中对该函数的调用发现

函数整体上采用尾递归的设计，每次执行时先判断边界条件，随后再进行具体的单次分裂的操作

下面是对于其逻辑的具体分析：



以将16KB分裂为4KB为例，我们可以用一个示意图来描述这个过程。此处粉色块表示被加入空闲链表的伙伴块，白色块表示继续递归分裂的块，直到拿到我们想要的块



最后我们再来看合并块的实现

merge_chunk

源码

```

/* The most recursion level of merge_chunk is decided by the macro of
 * BUDDY_MAX_ORDER. */
static struct page *merge_chunk(struct phys_mem_pool *pool, struct page
*chunk)
{
    struct page *buddy_chunk;

    /* The @chunk has already been the largest one. */
    if (chunk->order == (BUDDY_MAX_ORDER - 1)) {
        return chunk;
    }

    /* Locate the buddy_chunk of @chunk. */
    buddy_chunk = get_buddy_chunk(pool, chunk);

    /* If the buddy_chunk does not exist, no further merge is required.
 */
    if (buddy_chunk == NULL)
        return chunk;

    /* The buddy_chunk is not free, no further merge is required. */
    if (buddy_chunk->allocated == 1)
        return chunk;

    /* The buddy_chunk is not free as a whole, no further merge is
required.
 */
    if (buddy_chunk->order != chunk->order)
        return chunk;

    /* Remove the buddy_chunk from its current free list. */
    list_del(&(buddy_chunk->node));
    pool->free_lists[buddy_chunk->order].nr_free -= 1;

    /* Merge the two buddies and get a larger chunk @chunk (order+1). */
    buddy_chunk->order += 1;
    chunk->order += 1;
    if (chunk > buddy_chunk)
        chunk = buddy_chunk;

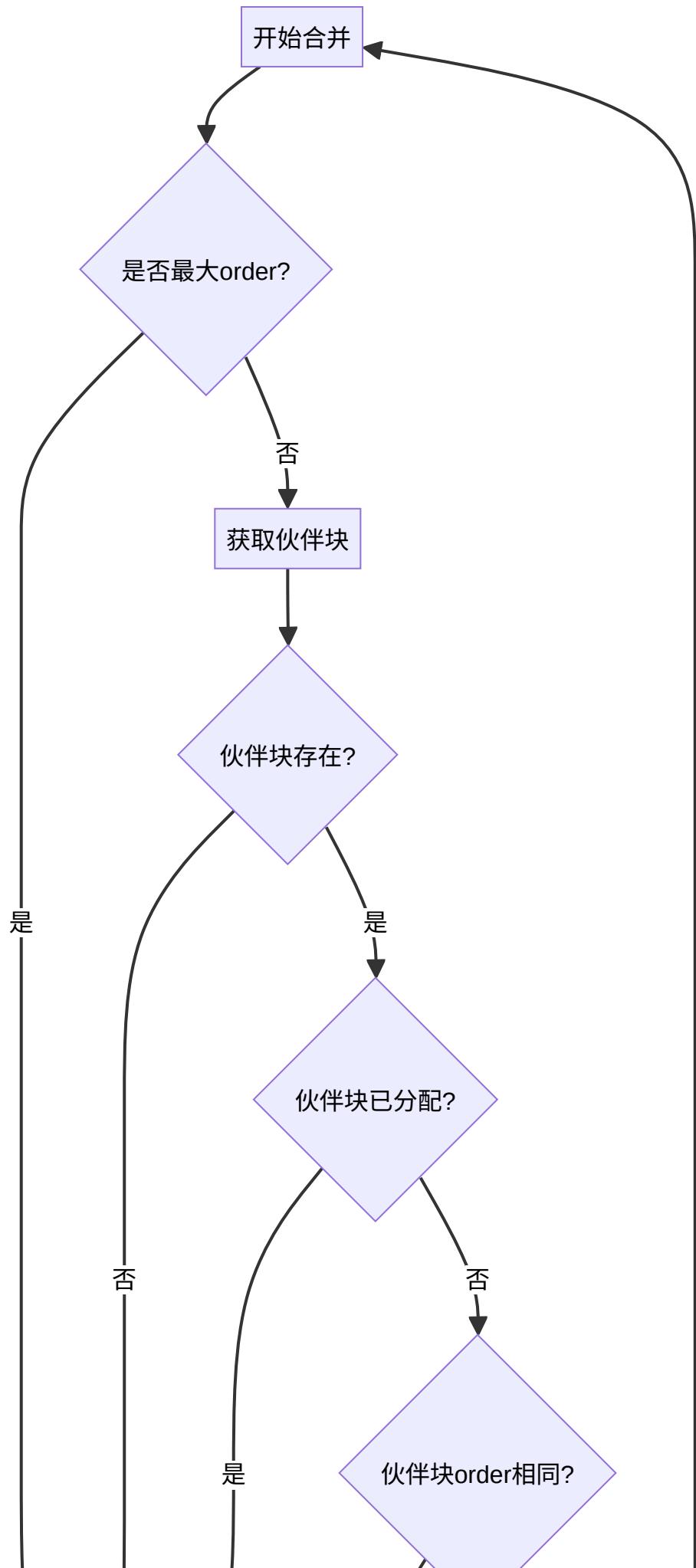
    /* Keeping merging. */
    return merge_chunk(pool, chunk);
}

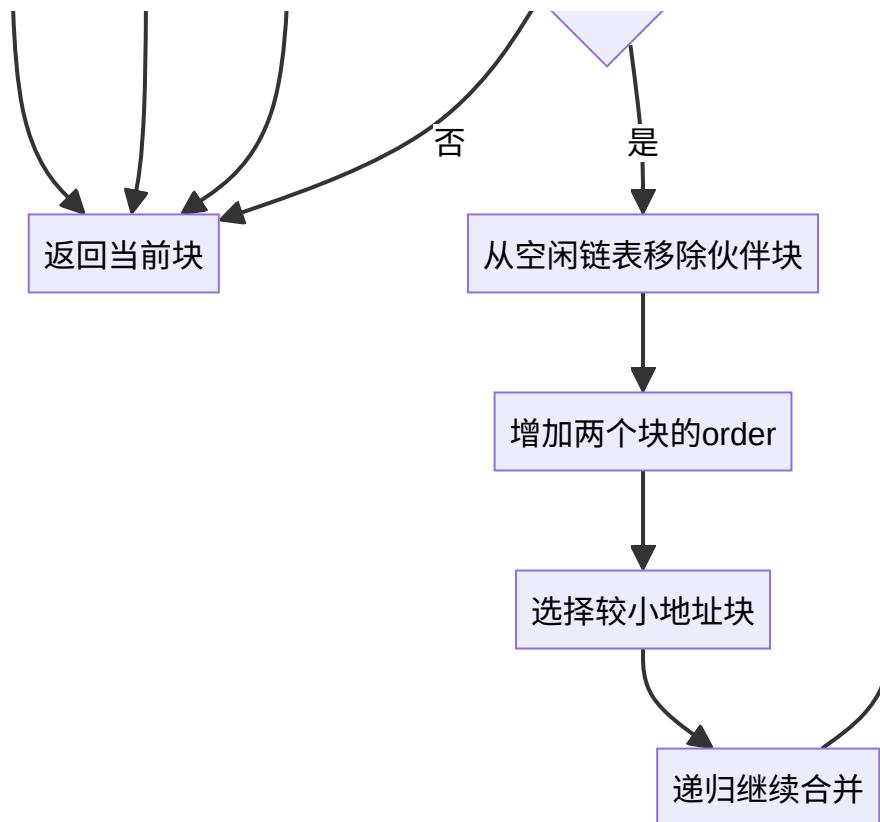
```

解析

有了对分裂块过程的了解，那么合并块理解起来就更简单了：同样是基于尾递归的实现，每次执行时先判断当前块是否已经达到最大的阶数or它的伙伴已经不在了，然后再执行相应操作，最后递归处理

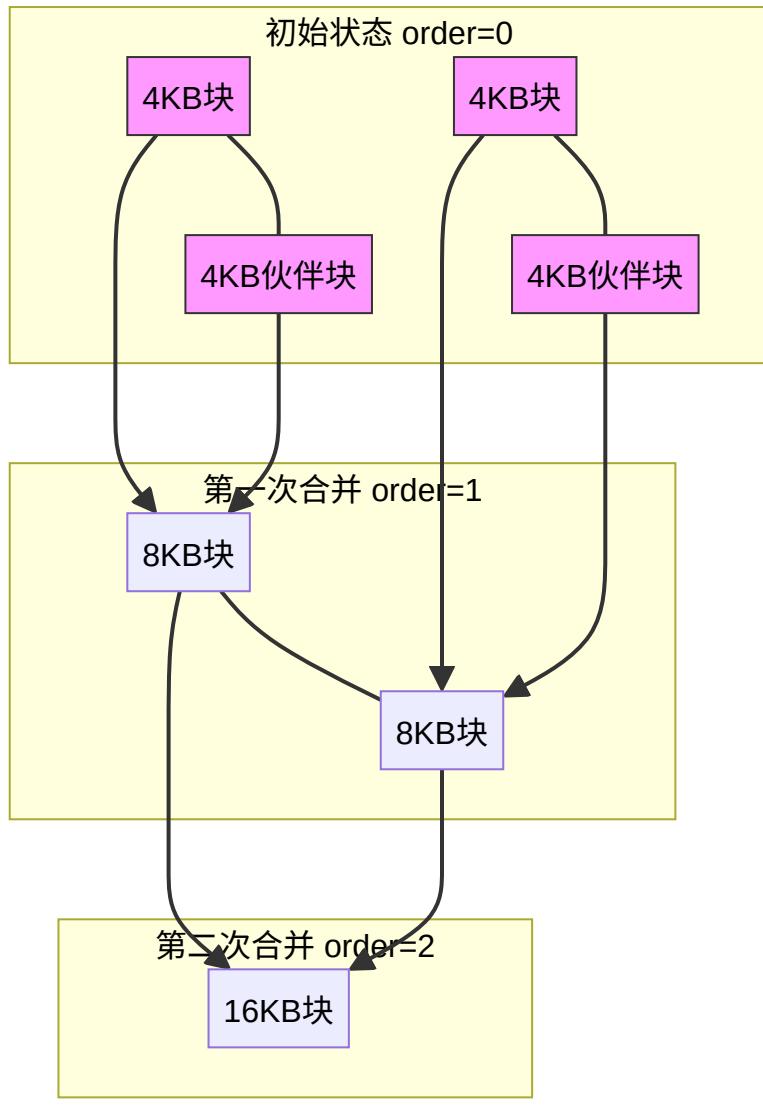
我们以一张示例图结束伙伴块之间的羁绊





注意这里判断伙伴块阶数相同的操作，实际上是看伙伴块是否依然作为一个整体空闲着存在，而不是已经裂开了

同样以4KB——16KB为例，介绍一次典型的合并过程



这样便完成了对merge操作的实现的介绍与解析

SLab分配器

Buddy System负责分配连续的物理内存页，但是如果系统遇到了小内存分配的需求，比如仅仅数十个字节的结构体等，这时候若继续使用伙伴系统，由于其最少只能分配4KB的内存，便会造成严重的内部碎片问题。而SLab分配器（SLub）便是专用于小内存分配的，它和Buddy System一起构成了Chcore的物理内存管理系统

本节内容承接上文，为大家介绍Chcore中SLab分配器的设计与具体实现，主要还是包括三个部分：

- 回顾SLab分配器的基础知识与设计理念
- 分析Chcore源码对SLab的代码实现与架构
- 解析Chcore源码对SLab操作相关函数的具体实现

SLab分配器的设计理念

SLab专用于小内存的分配。和伙伴系统不同，SLab本身不涉及对内存块的合并与分裂，而是通过一个个不同的SLab池，来分配不同固定大小的内存块，并且在SLab池内部使用链表结构串起一个个SLab，SLab内部又使用空闲链表的结构，便可以对所有的空闲内存块进行管理和分配

基本结构

- SLab是一个内存分配器，主要用于分配小块内存（32字节到2048字节之间）
- 整个系统维护了多个不同大小的 `slab_pool`，每个池对应特定大小的内存块
- 内存大小按2的幂次方划分：32, 64, 128, 256, 512, 1024, 2048字节（从 2^5 到 2^{11} ）

SLab的内部组织

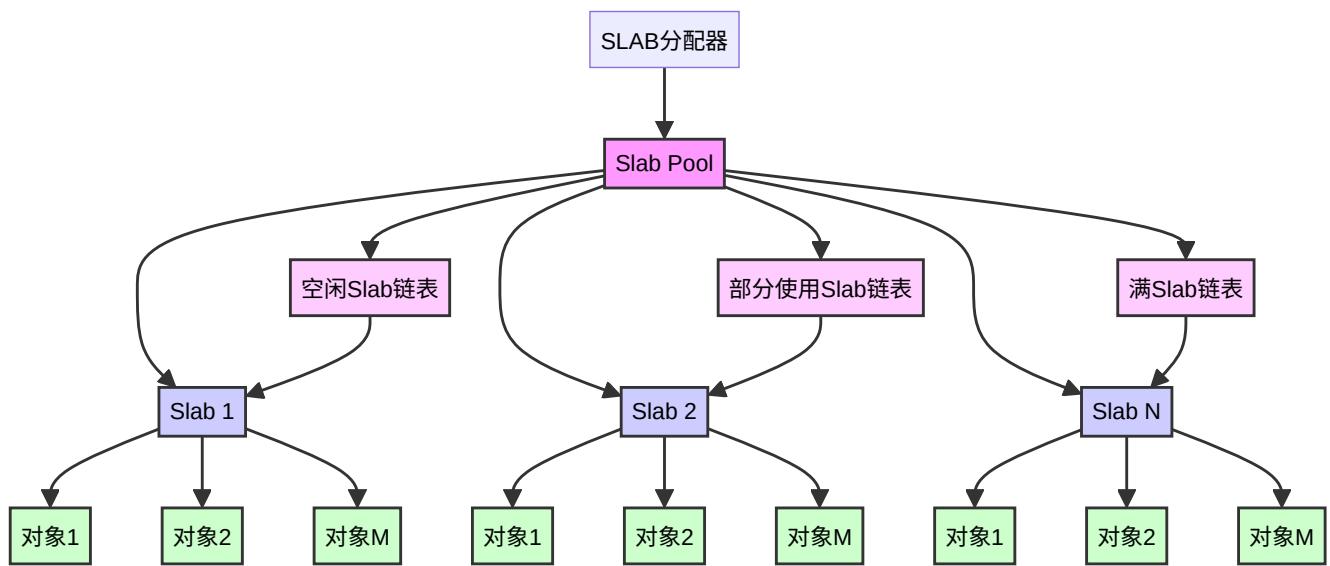
每个 `slab_pool` 由多个slab组成，每个slab又具有以下特点：

- 固定大小为128KB（`SIZE_OF_ONE_SLAB`）
- 包含一个 `slab_header`（位于slab开始处）
- 剩余空间被划分为大小相等的对象槽（`slots`）
- 使用空闲链表（`free_list`）管理未分配的对象槽
- 每个 `slab_pool` 有current和partial指针，分别指向当前slab以及总的slab链表组

SLab的工作流程

- 当请求分配内存时，SLAB分配器会首先从**部分使用Slab链表**中查找是否有可用的对象
- 果没有，它会从**空闲Slab链表**中取出一个slot，并将其移动到**部分使用Slab链表**
- 果空闲链表也为空（即partial为空），则会从内存中（即伙伴系统）分配一个新的slab，然后进行上一步操作
- 当释放内存块时，果一个slab中的所有slots都被释放，则该slab会被移动到**空闲Slab链表**

有 下示意图以供参考



这里的对象即为内存块，在Chcore实现中用slot表示，下面会提到。实际的SLab分配器的设计可能用到full_slab链表也可能不会，具体根据其实现确定

查看本机Slab示例：

```
sudo cat /proc/slabinfo
```

kmalloc-4k		569	584	4096	8	8 : tunables	0	0	0 :
slabdata	73	73	0						
kmalloc-2k		752	752	2048	16	8 : tunables	0	0	0 :
slabdata	47	47	0						
kmalloc-1k		704	704	1024	32	8 : tunables	0	0	0 :
slabdata	22	22	0						
kmalloc-512		2112	2112	512	32	4 : tunables	0	0	0 :
slabdata	66	66	0						
kmalloc-256		768	768	256	32	2 : tunables	0	0	0 :
slabdata	24	24	0						
kmalloc-192		1050	1050	192	42	2 : tunables	0	0	0 :
slabdata	25	25	0						
kmalloc-128		1376	1376	128	32	1 : tunables	0	0	0 :
slabdata	43	43	0						
kmalloc-96		1134	1134	96	42	1 : tunables	0	0	0 :
slabdata	27	27	0						
kmalloc-64		3072	3072	64	64	1 : tunables	0	0	0 :
slabdata	48	48	0						
kmalloc-32		3072	3072	32	128	1 : tunables	0	0	0 :
slabdata	24	24	0						
kmalloc-16		6144	6144	16	256	1 : tunables	0	0	0 :
slabdata	24	24	0						
kmalloc-8		12288	12288	8	512	1 : tunables	0	0	0 :
slabdata	24	24	0						

Chcore中SLab分配器的设计

同伙伴系统一样，这部分主要介绍Chcore中SLab分配器实现的核心数据结构和函数接口

核心数据结构

先上代码，看看有哪些数据结构

```
/*
 * order range: [SLAB_MIN_ORDER, SLAB_MAX_ORDER]
 * ChCore prepares the slab for each order in the range.
 */
#define SLAB_MIN_ORDER (5)
#define SLAB_MAX_ORDER (11)

/* The size of one slab is 128K. */
#define SIZE_OF_ONE_SLAB (128*1024)

/* slab_header resides in the beginning of each slab (i.e., occupies the
first slot). */
struct slab_header {
    /* The list of free slots, which can be converted to struct
slab_slot_list. */
    void *free_list_head;
    /* Partial slab list. */
    struct list_head node;

    int order;
    unsigned short total_free_cnt; /* MAX: 65536 */
    unsigned short current_free_cnt;
};

/* Each free slot in one slab is regarded as slab_slot_list. */
struct slab_slot_list {
    void *next_free;
};

struct slab_pointer {
    struct slab_header *current_slab;
    struct list_head partial_slab_list;
};

/* slab_pool is also static. We do not add the static modifier due to unit
test.
*/
struct slab_pointer slab_pool[SLAB_MAX_ORDER + 1];
static struct lock slabs_locks[SLAB_MAX_ORDER + 1];
```

Chcore中的SLab分配器设定大体服从书上的安排，以内存池为单元，先看宏定义：

- SLAB_MIN_ORDER & SLAB_MAX_ORDER：表示SLab分配器可以操作的内存块大小，从 2^{11} 到 2^{11} 字节
- SIZE_OF_ONE_SLAB：表示每个slab的大小，在Chcore中是128KB

接下来我们再分析其核心数据结构的实现：

slab_pointer

即Chcore中表示slab池的数据结构，这个也可以在下面对 `slab_pool` 的定义中看见。其数据成员即我们之前介绍的current和partial指针，在这里以 `slab_header` 和 `list_head` 的形式出现

slab_header

是Chcore中代表一个个具体的slab的对象，其成员包含：

- `void *free_list_head`：内部空闲slot的链表
- `struct list_head node`：partial中表示自身的节点

何由节点反过来得到其处于的slab？

在chcore之中采用的是内存对齐+元数据的方式，具体而言，元数据为：

- 在每一个slab块的开头，维护一个其中内存槽slot的free list head
- 在每一个内存槽slot之中，存一个next_free的指针
- slab由buddy sys分配，保证slab header地址按 page 对齐

这样，当我想要free addr时，可以找到对应的page，进而找到slab header，进而得到free_list，最后在free_list之中插入这个slot，然后由这个slab的空闲slot个数判断是否是 full → partial，partial→free，从而进行插入partial list或者把slab空间归还给buddy sys的操作

- `int order`：该slab的阶数
- `total_free_cnt & current_free_cnt`：总的空闲块数 & 当前空闲块数

最后是slot自身

slab_slot_list

仅包含一个 `void *next_free` 指针，它们串起来得到每个slab内部的空闲块链表

核心函数接口

Chcore的SLab分配器的实现包含了较多的辅助函数，这里只介绍重要的几个接口：

```
void init_slab(void);
void *alloc_in_slab(unsigned long, size_t *);
void free_in_slab(void *addr);
```

从上到下函数接口的功能依次为：

- `init_slab`：初始化SLab分配器
- `alloc_in_slab`：在slab中申请分配内存
- `free_in_slab`：释放相应的内存

Chcore源码对SLab功能的具体实现

同伙伴系统一样，这部分解析Chcore源码对SLab功能函数的具体实现，主要内容即为上面部分提到的三个函数，其中又会涉及到其他辅助函数，根据重要程度串起来讲解

`init_slab`

源码

```
void init_slab(void)
{
    int order;

    /* slab obj size: 32, 64, 128, 256, 512, 1024, 2048 */
    for (order = SLAB_MIN_ORDER; order <= SLAB_MAX_ORDER; order++) {
        lock_init(&slabs_locks[order]);
        slab_pool[order].current_slab = NULL;
        init_list_head(&(slab_pool[order].partial_slab_list));
    }
    kdebug("mm: finish initing slab allocators\n");
}
```

解析

初始化函数和伙伴系统比起来简单了许多，直接用一个for循环去过一遍所有slab池即可：

- 初始化 32~2048 字节为slot的各个slab, 每一个slab为 `SIZE_OF_ONE_SLAB=128K` 大小
- 初始化对应的锁
- 对全局变量的slab array的 `current_slab` `partial_slab_list` 初始化

`alloc_in_slab`

源码

```
void *alloc_in_slab(unsigned long size, size_t *real_size)
{
    int order;

    BUG_ON(size > order_to_size(SLAB_MAX_ORDER));

    order = (int)size_to_order(size);
    if (order < SLAB_MIN_ORDER)
        order = SLAB_MIN_ORDER;

#ifndef ENABLE_MEMORY_USAGE_COLLECTING
    if (real_size)
        *real_size = 1 << order;
#endif

    return alloc_in_slab_impl(order);
}

static void *alloc_in_slab_impl(int order)
{
    struct slab_header *current_slab;
    struct slab_slot_list *free_list;
    void *next_slot;

    lock(&slabs_locks[order]);

    current_slab = slab_pool[order].current_slab;
    /* When serving the first allocation request. */
    if (unlikely(current_slab == NULL)) {
        current_slab = init_slab_cache(order, SIZE_OF_ONE_SLAB);
        if (current_slab == NULL) {
            unlock(&slabs_locks[order]);
            return NULL;
        }
        slab_pool[order].current_slab = current_slab;
    }

    free_list = (struct slab_slot_list *)current_slab->free_list_head;
    BUG_ON(free_list == NULL);

    next_slot = free_list->next_free;
    current_slab->free_list_head = next_slot;

    current_slab->current_free_cnt -= 1;
    /* When current_slab is full, choose a new slab as the current one.
 */
    if (unlikely(current_slab->current_free_cnt == 0))
        choose_new_current_slab(&slab_pool[order]);

    unlock(&slabs_locks[order]);

    return (void *)free_list;
}

static void choose_new_current_slab(struct slab_pointer *pool)
{
```

```
struct list_head *list;

list = &(pool->partial_slab_list);
if (list_empty(list)) {
    pool->current_slab = NULL;
} else {
    struct slab_header *slab;

    slab = (struct slab_header *)list_entry(
        list->next, struct slab_header, node);
    pool->current_slab = slab;
    list_del(list->next);
}
}
```

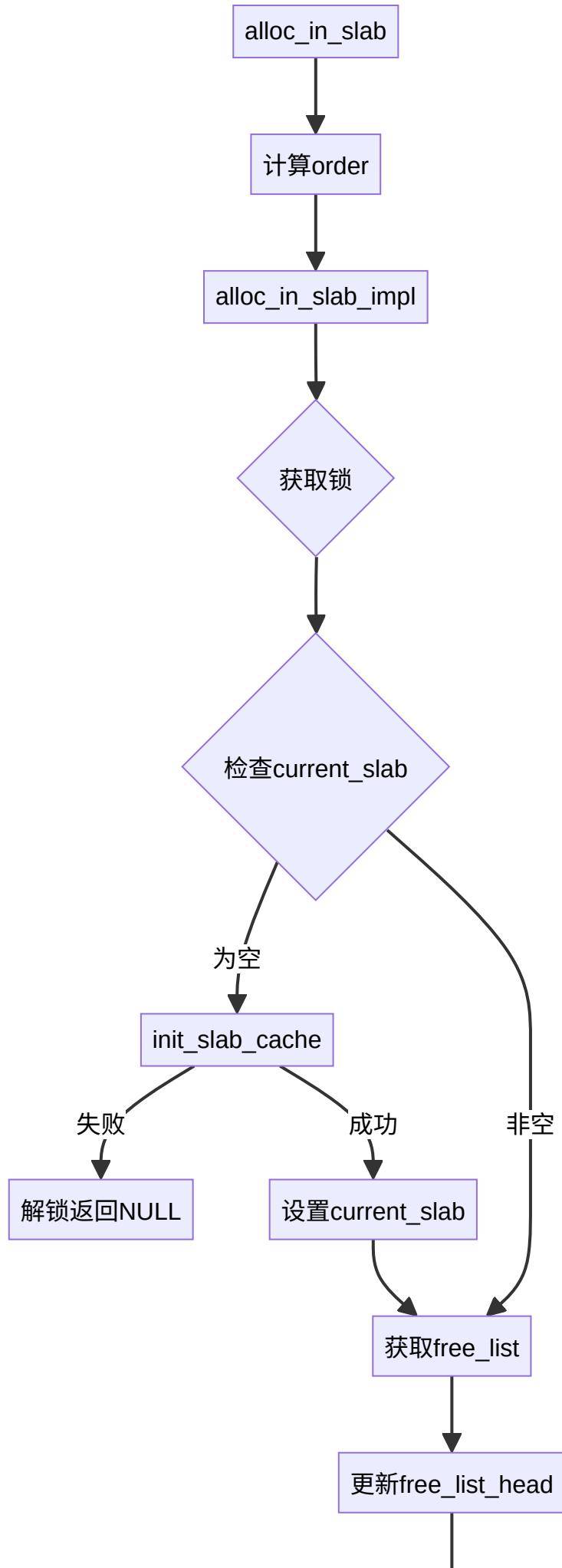
解析

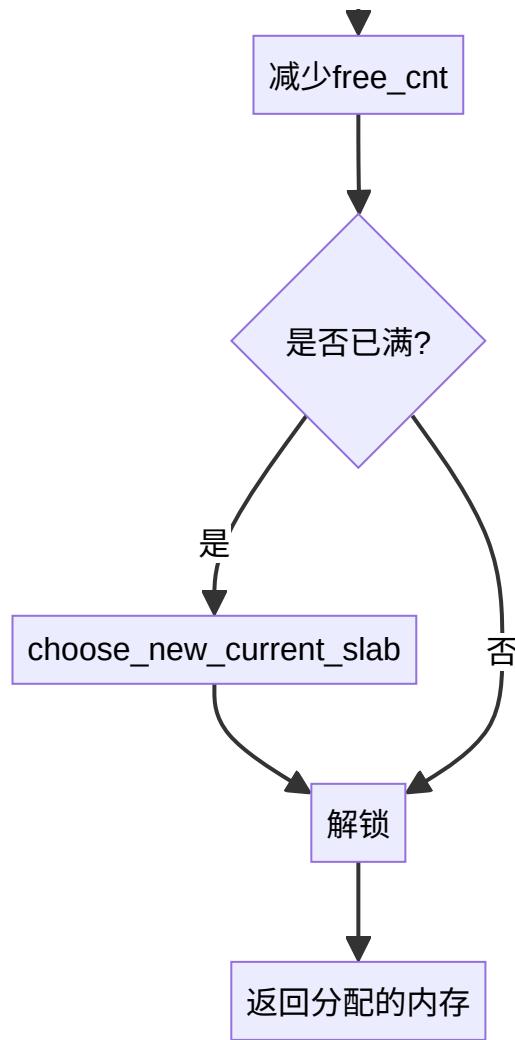
注意到源码将参数检查和核心实现分为了两部分，其中核心实现单独用了一个辅助函数，对外的接口则为 `alloc_in_slab` 本身

二者视作一起看，其核心逻辑 下：

- 参数检查，确保order符合范围，太大则报错，太小则向上补到最小值
- 加锁，确保并发安全性
- 若current为空，则为首次分配，需要先init一下（注意这里是直接使用的内部接口，保持了外界的无感知，也简化了这里的代码逻辑）
- 随后直接从current的slab的空闲链表中取出一块slot作为返回值，同时处理相应字段，空闲块数量减一，修改list_head等
- 若当前slab已满，则换新的slab，通过 `choose_new_current_slab` 进行操作，方式依然是通过定义好的list操作宏直接操作相应的链表
- 解锁，完成分配

逻辑图可以作 下参考，便于理解阅读：





free_in_slab

最后是free的操作，为slab拼上最后一块拼图

源码

```

void free_in_slab(void *addr)
{
    struct page *page;
    struct slab_header *slab;
    struct slab_slot_list *slot;
    int order;

    slot = (struct slab_slot_list *)addr;
    page = virt_to_page(addr);
    if (!page) {
        kdebug("invalid page in %s", __func__);
        return;
    }

    slab = page->slab;
    order = slab->order;
    lock(&slabs_locks[order]);

    try_insert_full_slab_to_partial(slab);

#if ENABLE_DETECTING_DOUBLE_FREE_IN_SLAB == ON
    /*
     * SLAB double free detection: check whether the slot to free is
     * already in the free list.
     */
    if (check_slot_is_free(slab, slot) == 1) {
        kinfo("SLAB: double free detected. Address is %p\n",
              (unsigned long)slot);
        BUG_ON(1);
    }
#endif

    slot->next_free = slab->free_list_head;
    slab->free_list_head = slot;
    slab->current_free_cnt += 1;

    try_return_slab_to_buddy(slab, order);

    unlock(&slabs_locks[order]);
}

```

解析

整体上还是走的“获取信息——转到相应object——根据情况分类讨论作对应操作”

梳理一下 `free_in_slab` 的实现逻辑：

- 处理参数，通过地址转换拿到slot、page、slab等对象
- 上锁
- 检查现在的slab是否是full的，果是，那么在free前还得把它放回partial
- 进行free操作，即把链表头插回来、设置slab空闲链表、以及空闲数加一
- 检查现在的slab是否是空的，果是，那么物归原主——还给buddy大人

- 解锁，完成free全部操作

至此，SLab分配器源码解析结束

Kmalloc与Kfree

有了Buddy System和SLab分配器，我们便能够去做真正的分配内存与释放内存的操作了——也即malloc和free。本部分源码解析即讲解Chcore的内核中kmalloc/kfree，这里的k表示kernel，即专用于内核空间分配/释放的函数

直接上源码！

源码

kmalloc

```
void *kmalloc(unsigned long size)
{
    size_t real_size;
    void *ret;
    ret = _kmalloc(size, true, &real_size);
    return ret;
}

/* Currently, BUG_ON no available memory. */
void *_kmalloc(size_t size, bool is_record, size_t *real_size)
{
    void *addr;
    int order;

    if (unlikely(size == 0))
        return ZERO_SIZE_PTR;

    if (size <= SLAB_MAX_SIZE) {
        addr = alloc_in_slab(size, real_size);
#if ENABLE_MEMORY_USAGE_COLLECTING == ON
        if(is_record && collecting_switch) {
            record_mem_usage(*real_size, addr);
        }
#endif
    } else {
        if (size <= BUDDY_PAGE_SIZE)
            order = 0;
        else
            order = size_to_page_order(size);
        addr = get_pages(order);
    }

    return addr;
}
```

kfree

```

void kfree(void *ptr)
{
    _kfree(ptr, true);
}

void _kfree(void *ptr, bool is_revoke_record)
{
    struct page *page;

    if (unlikely(ptr == ZERO_SIZE_PTR))
        return;

    page = virt_to_page(ptr);
#ifndef ENABLE_MEMORY_USAGE_COLLECTING
    if (collecting_switch && is_revoke_record) {
        revoke_mem_usage(ptr);
    }
#endif
    if (page && page->slab)
        free_in_slab(ptr);
    else if (page && page->pool)
        buddy_free_pages(page->pool, page);
    else
        kwarn("unexpected state in %s\n", __func__);
}

```

解析

有了之前Buddy System和SLab提供的接口，实现这两个函数便简单多了，参数检查过后，直接用一个分支结构即可解决

kmalloc

- 参数检查
- 看需要分配的size大小，若小于 SLAB_MAX_SIZE 则使用SLab分配器的接口分配内存
- 若大于，则使用伙伴系统的接口分配内存（包装在 get_pages 函数里的，这里没有列出）
- 返回分配得到的地址指针

kfree

- 参数检查
- 根据地址拿到它所属的page
- 看page的属性，果它有slab，则说明是slab里的page，交由slab接口处理
- 果它有pool，则说明是伙伴系统的page，交由伙伴系统接口处理

总结

至此，物理内存管理部分的源码解析已经全部结束，希望能对你的学习有所裨益！

Last change: 2025-07-05, commit: [b9c54c8](#)

目录

- 页表管理
 - 核心数据结构
 - 函数功能实现
 - 接口定义
 - 源码解析
- 缺页管理
 - ARM异常机制
 - 异常类型
 - 处理逻辑
 - 缺页异常源码解析
 - VMR & PMO

页表管理

本节内容讲解Chcore页表管理部分的源码（不包括缺页管理，缺页管理单独讲解），我们将从Chcore页表管理的核心数据结构讲起，并进一步解析其页表管理的函数实际实现

核心数据结构

Chcore中表示页表的核心数据结构 下方源码所示：

```
/* page_table_entry type */
typedef union {
    struct {
        u64 is_valid : 1,
        is_table : 1,
        ignored1 : 10,
        next_table_addr : 36,
        reserved : 4,
        ignored2 : 7,
        PXNTable : 1, // Privileged Execute-never for
next level
        XNTable : 1, // Execute-never for next level
        APTable : 2, // Access permissions for next
level
        NSTable : 1;
    } table;
    struct {
        u64 is_valid : 1,
        is_table : 1,
        attr_index : 3, // Memory attributes index
        NS : 1, // Non-secure
        AP : 2, // Data access permissions
        SH : 2, // Shareability
        AF : 1, // Accesss flag
        nG : 1, // Not global bit
        reserved1 : 4,
        nT : 1,
        reserved2 : 13,
        pfn : 18,
        reserved3 : 2,
        GP : 1,
        reserved4 : 1,
        DBM : 1, // Dirty bit modifier
        Contiguous : 1,
        PXN : 1, // Privileged execute-never
        UXN : 1, // Execute never
        soft_reserved : 4,
        PBHA : 4; // Page based hardware attributes
    } l1_block;
    struct {
        u64 is_valid : 1,
        is_table : 1,
        attr_index : 3, // Memory attributes index
        NS : 1, // Non-secure
        AP : 2, // Data access permissions
        SH : 2, // Shareability
        AF : 1, // Accesss flag
        nG : 1, // Not global bit
        reserved1 : 4,
        nT : 1,
        reserved2 : 4,
        pfn : 27,
        reserved3 : 2,
        GP : 1,
        reserved4 : 1,
        DBM : 1, // Dirty bit modifier
    } l2_block;
} page_table_entry;
```

```

        Contiguous      : 1,
        PNX            : 1,    // Privileged execute-never
        UXN            : 1,    // Execute never
        soft_reserved  : 4,
        PBHA           : 4;    // Page based hardware attributes
    } l2_block;
    struct {
        u64 is_valid      : 1,
        is_page          : 1,
        attr_index       : 3,    // Memory attributes index
        NS               : 1,    // Non-secure
        AP               : 2,    // Data access permissions
        SH               : 2,    // Shareability
        AF               : 1,    // Accesss flag
        nG               : 1,    // Not global bit
        pfn              : 36,
        reserved         : 3,
        DBM              : 1,    // Dirty bit modifier
        Contiguous       : 1,
        PNX              : 1,    // Privileged execute-never
        UXN              : 1,    // Execute never
        soft_reserved    : 4,
        PBHA             : 4,    // Page based hardware attributes
        ignored          : 1;
    } l3_page;
    u64 pte;
} pte_t;

/* page_table_page type */
typedef struct {
    pte_t ent[PTP_ENTRIES];
} ptp_t;

```

共有两种数据结构，分别是：

- pte_t：即页表项，表示一个具体的页表
- ptp_t：即页表页，我们可以看见它是由一个 pte_t 的数组构成的结构体

我们这里重点看一下 pte_t 的定义——它采用了bit-fields和union的语法

所谓bit-fields，即

```
type member_name : bit_width
```

每个字段后面的数字表示该字段占用的位数，编译器会自动将这些字段打包到一个u64中，字段的总位数不能超过基础类型（这里是u64）的大小，由此我们可以总结该数据结构的特点：

- 使用union来表示不同类型的页表项
- 支持4种格式： table （指向下级页表）、 l1_block、 l2_block （大页） 和 l3_page （4KB页）
- 通过bit-field精确控制每个控制位的位置
- 允许以不同方式解释同一块内存，可以直接访问原始值或者结构化的字段

这时候还有一个问题，不同架构所用到的页表项是不一样的，所以我们需要一个通用页表项来处理：

```
/**  
 * @brief Architecture-independent PTE structure, containing some useful  
 * information which is shared by all architectures.  
 *  
 * This struct can be used to write architecture-independent code to parse  
 * or manipulate PTEs.  
 */  
struct common_pte_t {  
    /** Physical Page Number */  
    unsigned long ppn;  
    /** ChCore VMR permission, architecture-independent */  
    vmr_prop_t perm;  
    unsigned char  
        /** This PTE is valid or not */  
        valid : 1,  
        /** This PTE is accessed by hardware or not */  
        access : 1,  
        /** This PTE is written by hardware or not */  
        dirty : 1, _unused : 4;  
};
```

它是一个架构无关的页表项抽象，主要作用是提供一个统一的接口来处理不同架构的页表项

在函数实现中，我们会有相应的辅助函数来提供将页表项和通用页表项之间转换的功能，下例所示

```

// 从架构相关的PTE转换为通用PTE
void parse_pte_to_common(pte_t *pte, unsigned int level, struct common_pte_t
*ret)
{
    switch (level) {
        case L3:
            ret->ppn = pte->l3_page.pfn;           // 提取物理页号
            ret->perm = 0;
            ret->perm |= (pte->l3_page.UXN ? 0 : VMR_EXEC); // 转换执行权限
            ret->perm |= __ap_to_vmr_prot(pte->l3_page.AP); // 转换访问权限
            ret->perm |= (pte->l3_page.attr_index == DEVICE_MEMORY ?
                            VMR_DEVICE : 0); // 转换内存属性
            ret->access = pte->l3_page.AF;           // 转换访问标志
            ret->dirty = pte->l3_page.DBM;           // 转换脏页标志
            ret->valid = pte->l3_page.is_valid;       // 转换有效位
            break;
    }
}

// 从通用PTE更新到架构相关PTE，不过这里目前还只支持L3级别的更新
void update_pte(pte_t *dest, unsigned int level, struct common_pte_t *src)
{
    switch (level) {
        case L3:
            dest->l3_page.pfn = src->ppn;           // 更新物理页号
            dest->l3_page.AP = __vmr_prot_to_ap(src->perm); // 更新访问权限
            dest->l3_page.UXN = ((src->perm & VMR_EXEC) ? AARCH64_MMU_ATTR_PAGE_UX :
                                  AARCH64_MMU_ATTR_PAGE_UXN);
            dest->l3_page.is_valid = src->valid;       // 更新有效位
            dest->l3_page.AF = src->access;           // 更新访问标志
            dest->l3_page.DBM = src->dirty;           // 更新脏页标志
            break;
    }
}

```

函数功能实现

正如我们的Lab文档所提到的，内核启动阶段所做的事情只是配置了一个粗粒度的页表系统。而实际操作系统所需要的页表管理则远不止于此。我们需要一个更细粒度的页表实现，提供映射、取消映射、查询等功能。而这些功能在源码中则以各种接口（接口在 `mmu.h` 中）呈现，并在 `page_table.c` 中实现。

还是先看看接口是怎么定义的，再来看实现。

接口定义

```

int map_range_in_pgtbl_kernel(void *pgtbl, vaddr_t va, paddr_t pa,
    size_t len, vmr_prop_t flags);
int map_range_in_pgtbl(void *pgtbl, vaddr_t va, paddr_t pa,
    size_t len, vmr_prop_t flags, long *rss);
int unmap_range_in_pgtbl(void *pgtbl, vaddr_t va, size_t len, long *rss);
int query_in_pgtbl(void *pgtbl, vaddr_t va, paddr_t *pa, pte_t **entry);
int mprotect_in_pgtbl(void *pgtbl, vaddr_t va, size_t len, vmr_prop_t prop);

```

从上到下介绍一遍；

- `map_range_in_pgtbl`：页表映射函数，又分为内核态与用户态，但是实现逻辑基本是一样的，因此在源码中会用一个common辅助函数来实现
- `unmap_range_in_pgtbl`：取消页表映射函数
- `query_in_pgtbl`：页表查询函数
- `mprotect_in_pgtbl`：页表权限修改函数

源码解析

`map_range_in_pgtbl`

我们先对比一下两个函数的接口以及源码

```

int map_range_in_pgtbl_kernel(
    void *pgtbl,           // 页表基地址
    vaddr_t va,            // 要映射的虚拟地址起始位置
    paddr_t pa,            // 要映射的物理地址起始位置
    size_t len,             // 映射长度
    vmr_prop_t flags       // 映射属性（读/写/执行权限等）
)

int map_range_in_pgtbl(
    void *pgtbl,           // 页表基地址
    vaddr_t va,            // 虚拟地址起始位置
    paddr_t pa,            // 物理地址起始位置
    size_t len,             // 映射长度
    vmr_prop_t flags,      // 映射属性
    long *rss              // 常驻集大小计数器
)

```

可以看到，非内核的页表映射函数还多了一个rss计数器，它的作用是跟踪用户进程的内存使用情况

再来看其实现，会发现都用到了一个common函数：

```
/* Map vm range in kernel */
int map_range_in_pgtbl_kernel(void *pgtbl, vaddr_t va, paddr_t pa,
    size_t len, vmr_prop_t flags)
{
    return map_range_in_pgtbl_common(pgtbl, va, pa, len, flags,
        KERNEL_PTE, NULL);
}

/* Map vm range in user */
int map_range_in_pgtbl(void *pgtbl, vaddr_t va, paddr_t pa,
    size_t len, vmr_prop_t flags, long *rss)
{
    return map_range_in_pgtbl_common(pgtbl, va, pa, len, flags,
        USER_PTE, rss);
}
```

那么关键就在这个 `map_range_in_pgtbl_common` 函数，我们学习学习它的源码实现

```
static int map_range_in_pgtbl_common(void *pgtbl, vaddr_t va, paddr_t pa,
size_t len,
                                     vmr_prop_t flags, int kind, long *rss)
{
    s64 total_page_cnt;
    ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp;
    pte_t *pte;
    int ret;
    int pte_index; // the index of pte in the last level page table
    int i;

    BUG_ON(pgtbl == NULL); // alloc the root page table page at first
    BUG_ON(va % PAGE_SIZE);
    total_page_cnt = len / PAGE_SIZE + (((len % PAGE_SIZE) > 0) ? 1 : 0);

    l0_ptp = (ptp_t *)pgtbl;

    l1_ptp = NULL;
    l2_ptp = NULL;
    l3_ptp = NULL;

    while (total_page_cnt > 0) {
        // l0
        ret = get_next_ptp(l0_ptp, L0, va, &l1_ptp, &pte, true, rss);
        BUG_ON(ret != 0);

        // l1
        ret = get_next_ptp(l1_ptp, L1, va, &l2_ptp, &pte, true, rss);
        BUG_ON(ret != 0);

        // l2
        ret = get_next_ptp(l2_ptp, L2, va, &l3_ptp, &pte, true, rss);
        BUG_ON(ret != 0);

        // l3
        // step-1: get the index of pte
        pte_index = GET_L3_INDEX(va);
        for (i = pte_index; i < PTP_ENTRIES; ++i) {
            pte_t new_pte_val;

            new_pte_val.pte = 0;
            new_pte_val.l3_page.is_valid = 1;
            new_pte_val.l3_page.is_page = 1;
            new_pte_val.l3_page.pfn = pa >> PAGE_SHIFT;
            set_pte_flags(&new_pte_val, flags, kind);
            l3_ptp->ent[i].pte = new_pte_val.pte;

            va += PAGE_SIZE;
            pa += PAGE_SIZE;
            if (rss)
                *rss += PAGE_SIZE;
            total_page_cnt -= 1;
            if (total_page_cnt == 0)
                break;
        }
    }
}
```

```

    dsb(ishst);
    isb();

    /* Since we are adding new mappings, there is no need to flush TLBs.
 */
    return 0;
}

```

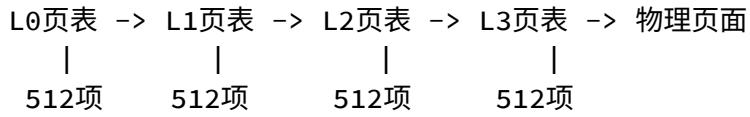
总体上就是参数检查——计算需要映射的总页数——开while循环开始映射

```

// 一次循环可以映射的最大页数
最大映射数 = PTP_ENTRIES - pte_index
= 512 - (va >> 12 & 0x1FF)

```

数据结构关系 下：



下面是详细的函数执行逻辑：

- 参数检查，并计算需要映射的总页数，这里的方式是向上取整
- 初始化页表指针，为后面的大循环做准备
- 进入while循环，依次获取四级页表的页表页，其本质就是位运算，可以回顾一下机器启动部分关于页表映射的讲解
- 获得到L3页表的索引，并尽可能多的去映射，映射时需要设置其页表项字段以及更新rss和页表页数组
- 重复while循环直到映射完毕，并建立数据和指令同步屏障

这里再来明晰一下while循环的作用：因为每个L3级别的页表页只能映射 $2^9=512$ 个页表项，因此当映射需求较大的时候就需要多轮循环才能映射完毕

我们假设某次映射需求有2000个页表项需要被映射，那么会发生 下事情：

```
// 假设要映射2000个页面 (约8MB)
初始: total_page_cnt = 2000
```

第一次外层循环:

- 找到第一个L3页表
- 从pte_index开始映射
- 假设pte_index = 100
- 可以映射412个页面(512-100)
- total_page_cnt = 1588

第二次外层循环:

- 找到/创建下一个L3页表
- 从索引0开始映射
- 可以映射512个页面
- total_page_cnt = 1076

... 循环继续 ...

循环会一直执行，直到完成所有映射需求或者遇到错误（内存不足）

unmap_in_range_pgtbl

还是先上源码

```
int unmap_range_in_pgtbl(void *pgtbl, vaddr_t va, size_t len, long *rss)
{
    s64 total_page_cnt; // must be signed
    s64 left_page_cnt_in_current_level;
    ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp;
    pte_t *pte;
    vaddr_t old_va;

    int ret;
    int pte_index; // the index of pte in the last level page table
    int i;

    BUG_ON(pgtbl == NULL);

    l0_ptp = (ptp_t *)pgtbl;

    total_page_cnt = len / PAGE_SIZE + (((len % PAGE_SIZE) > 0) ? 1 : 0);
    while (total_page_cnt > 0) {
        old_va = va;
        // l0
        ret = get_next_ptp(l0_ptp, L0, va, &l1_ptp, &pte, false,
NULL);
        if (ret == -ENOMAPPING) {
            left_page_cnt_in_current_level =
                (L0_PER_ENTRY_PAGES
                 - GET_L1_INDEX(va) * L1_PER_ENTRY_PAGES);
            total_page_cnt -=
                (left_page_cnt_in_current_level
                 > total_page_cnt ?
                     total_page_cnt :
                     left_page_cnt_in_current_level);
            va += left_page_cnt_in_current_level * PAGE_SIZE;
            continue;
        }
        // l1
        ret = get_next_ptp(l1_ptp, L1, va, &l2_ptp, &pte, false,
NULL);
        if (ret == -ENOMAPPING) {
            left_page_cnt_in_current_level =
                (L1_PER_ENTRY_PAGES
                 - GET_L2_INDEX(va) * L2_PER_ENTRY_PAGES);
            total_page_cnt -=
                (left_page_cnt_in_current_level
                 > total_page_cnt ?
                     total_page_cnt :
                     left_page_cnt_in_current_level);
            va += left_page_cnt_in_current_level * PAGE_SIZE;
            continue;
        }
        // l2
        ret = get_next_ptp(l2_ptp, L2, va, &l3_ptp, &pte, false,
NULL);
        if (ret == -ENOMAPPING) {
            left_page_cnt_in_current_level =
```

```

        (L2_PER_ENTRY_PAGES
         - GET_L3_INDEX(va) * L3_PER_ENTRY_PAGES);
    total_page_cnt -=
        (left_page_cnt_in_current_level
         > total_page_cnt ?
            total_page_cnt :
            left_page_cnt_in_current_level);
    va += left_page_cnt_in_current_level * PAGE_SIZE;
    continue;
}

// l3
// step-1: get the index of pte
pte_index = GET_L3_INDEX(va);
for (i = pte_index; i < PTP_ENTRIES; ++i) {
    if (l3_ptp->ent[i].l3_page.is_valid && rss)
        *rss -= PAGE_SIZE;

    l3_ptp->ent[i].pte = PTE_DESCRIPTOR_INVALID;
    va += PAGE_SIZE;
    total_page_cnt -= 1;
    if (total_page_cnt == 0)
        break;
}
recycle_pgtable_entry(l0_ptp, l1_ptp, l2_ptp, l3_ptp, old_va,
rss);
}

dsb(ishst);
isb();

return 0;
}

```

整体逻辑和map时候的逻辑相似，只是在while大循环里，多了一个“跳过未映射区域”的操作，这样可以避免不必要的页表遍历，减少unmap的用时和资源开销

```

// 举例： 果L2级别未映射
// L2_PER_ENTRY_PAGES = 512 * 512 = 262144
// 可以一次跳过大未映射页面
left_page_cnt_in_current_level = L1_PER_ENTRY_PAGES - ...

```

而在L3部分真正解除映射的代码，又会涉及到 下操作：

- 判断页面是否有效，有效则更新为无效，并更新rss计数器
- 回收掉无效的页表项，即 recycle_pgtable_entry 函数，其实现即用到了之前提到的 kfree等

同样的，假设我们有解除映射的页表需求，可能的工作流程示例 下：

```
// 假设要解除映射1000000个页面
初始: total_page_cnt = 1000000

// 情况1: 遇到未映射区域
if (L2未映射) {
    跳过整个L2范围
    更新total_page_cnt
    继续下一个区域
}

// 情况2: 找到映射区域
在L3页表中:
    解除映射
    更新RSS
    尝试回收页表页
```

query_in_pgtbl

和上面两个函数一样，查询函数的实现逻辑是换汤不换药的，但是需要添加相应的特色内容，以满足查询本身的需求

注意后两个参数是输出

```
int query_in_pgtbl(
    void *pgtbl,           // 页表基地址
    vaddr_t va,             // 要查询的虚拟地址
    paddr_t *pa,            // 输出: 对应的物理地址
    pte_t **entry           // 输出: 对应的页表项 (可选)
)
```

```

/*
 * Translate a va to pa, and get its pte for the flags
 */
int query_in_pgtbl(void *pgtbl, vaddr_t va, paddr_t *pa, pte_t **entry)
{
    /* On aarch64, l0 is the highest level page table */
    ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp;
    ptp_t *phys_page;
    pte_t *pte;
    int ret;

    // L0 page table
    l0_ptp = (ptp_t *)pgtbl;
    ret = get_next_ptp(l0_ptp, L0, va, &l1_ptp, &pte, false, NULL);
    if (ret < 0)
        return ret;

    // L1 page table
    ret = get_next_ptp(l1_ptp, L1, va, &l2_ptp, &pte, false, NULL);
    if (ret < 0)
        return ret;
    else if (ret == BLOCK_PTP) {
        *pa = virt_to_phys((vaddr_t)l2_ptp) + GET_VA_OFFSET_L1(va);
        if (entry)
            *entry = pte;
        return 0;
    }

    // L2 page table
    ret = get_next_ptp(l2_ptp, L2, va, &l3_ptp, &pte, false, NULL);
    if (ret < 0)
        return ret;
    else if (ret == BLOCK_PTP) {
        *pa = virt_to_phys((vaddr_t)l3_ptp) + GET_VA_OFFSET_L2(va);
        if (entry)
            *entry = pte;
        return 0;
    }

    // L3 page table
    ret = get_next_ptp(l3_ptp, L3, va, &phys_page, &pte, false, NULL);
    if (ret < 0)
        return ret;

    *pa = virt_to_phys((vaddr_t)phys_page) + GET_VA_OFFSET_L3(va);
    if (entry)
        *entry = pte;
    return 0;
}

```

整体上除了一级一级页表往下查询+错误处理之外，还添加了一个额外的步骤——支持大页映射查询，如果发现相应的高级页表页是一个大页，那么则直接通过大页地址+页内偏移得到最终返回值

同样的，如果一路成功到了L3页表页，那么说明我们要查询的就是个基本的4KB小页，直接同样操作获取物理地址就行。相当于该函数一共支持三种页面大小：

- 1GB (L1 block)
- 2MB (L2 block)
- 4KB (L3 page)

mprotect_in_pgtbl

最后我们再来看看修改页表权限的操作是如何实现的，上源码

```
int mprotect_in_pgtbl(void *pgtbl, vaddr_t va, size_t len, vmr_prop_t flags)
{
    s64 total_page_cnt; // must be signed
    ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp;
    pte_t *pte;
    int ret;
    int pte_index; // the index of pte in the last level page table
    int i;

    BUG_ON(pgtbl == NULL);
    BUG_ON(va % PAGE_SIZE);

    l0_ptp = (ptp_t *)pgtbl;

    total_page_cnt = len / PAGE_SIZE + (((len % PAGE_SIZE) > 0) ? 1 : 0);
    while (total_page_cnt > 0) {
        // l0
        ret = get_next_ptp(l0_ptp, L0, va, &l1_ptp, &pte, false,
NULL);
        if (ret == -ENOMAPPING) {
            total_page_cnt -= L0_PER_ENTRY_PAGES;
            va += L0_PER_ENTRY_PAGES * PAGE_SIZE;
            continue;
        }

        // l1
        ret = get_next_ptp(l1_ptp, L1, va, &l2_ptp, &pte, false,
NULL);
        if (ret == -ENOMAPPING) {
            total_page_cnt -= L1_PER_ENTRY_PAGES;
            va += L1_PER_ENTRY_PAGES * PAGE_SIZE;
            continue;
        }

        // l2
        ret = get_next_ptp(l2_ptp, L2, va, &l3_ptp, &pte, false,
NULL);
        if (ret == -ENOMAPPING) {
            total_page_cnt -= L2_PER_ENTRY_PAGES;
            va += L2_PER_ENTRY_PAGES * PAGE_SIZE;
            continue;
        }

        // l3
        // step-1: get the index of pte
        pte_index = GET_L3_INDEX(va);
        for (i = pte_index; i < PTP_ENTRIES; ++i) {
            /* Modify the permission in the pte if it exists */
            if (!IS_PTE_INVALID(l3_ptp->ent[i].pte))
                set_pte_flags(
                    &(l3_ptp->ent[i])), flags, USER_PTE);

            va += PAGE_SIZE;
            total_page_cnt -= 1;
            if (total_page_cnt == 0)
                break;
        }
    }
}
```

```
        }  
    }  
  
    return 0;  
}
```

看过了上面的源码后，这部分源码就显得很亲切了，和unmap 出一辙的跳过未映射区域的操作

最后唯一更改的地方仅仅是L3的循环，将之前的unmap操作换成了调用 `set_pte_flag` 函数，即完成了对页表权限的修改

```

static int set_pte_flags(pte_t *entry, vmr_prop_t flags, int kind)
{
    BUG_ON(kind != USER_PTE && kind != KERNEL_PTE);

    /*
     * Current access permission (AP) setting:
     * Mapped pages are always readable (No considering XOM).
     * EL1 can directly access EL0 (No restriction like SMAP
     * as ChCore is a microkernel).
     */
    entry->l3_page.AP = __vmr_prot_to_ap(flags);

    if (kind == KERNEL_PTE) {
        // kernel PTE
        if (!(flags & VMR_EXEC))
            // 果没有指定flags任意权限（读，写,...）或者此段虚拟内存没有执行权限
            // pte权限设置为内核不可执行
            entry->l3_page.PXN = AARCH64_MMU_ATTR_PAGE_PXN;
        // 内核kind下pte user不可执行
        entry->l3_page.UXN = AARCH64_MMU_ATTR_PAGE_UXN;
    } else {
        // User PTE
        if (!(flags & VMR_EXEC))
            entry->l3_page.UXN = AARCH64_MMU_ATTR_PAGE_UXN;
        // EL1 cannot directly execute EL0 accessible region.
        // user kind下内核亦不可执行
        entry->l3_page.PXN = AARCH64_MMU_ATTR_PAGE_PXN;
    }

    // Set AF (access flag) in advance.
    entry->l3_page.AF = AARCH64_MMU_ATTR_PAGE_AF_ACCESSED;
    // Mark the mapping as not global
    entry->l3_page.nG = 1;
    // Mark the mapping as inner sharable
    entry->l3_page.SH = INNER_SHAREABLE;
    // Set the memory type
    if (flags & VMR_DEVICE) {
        entry->l3_page.attr_index = DEVICE_MEMORY;
        entry->l3_page.SH = 0;
    } else if (flags & VMR_NOCACHE) {
        entry->l3_page.attr_index = NORMAL_MEMORY_NOCACHE;
    } else {
        entry->l3_page.attr_index = NORMAL_MEMORY;
    }

#endif CHCORE_OPENTRUSTEE
    if (flags & VMR_TZ_NS) {
        entry->l3_page.NS = AARCH64_MMU_ATTR_PAGE_NS_NON_SECURE;
    }
#endif /* CHCORE_OPENTRUSTEE */

    return 0;
}

```

其中几个安全名词

SMAP (Supervisor Mode Access Prevention) : **SMAP**是ARM架构中的一项安全功能，可防止管理模式（类似于EL1）在未经适当检查的情况下直接访问用户模式内存（类似于EL0）。这有助于防止特权提升攻击，其中在EL0处运行的恶意程序可能尝试访问属于EL1处的OS的敏感数据。声明“No restriction like SMAP”意味着，在这种配置中，EL1可以直接访问EL0内存，而不受通常的SMAP限制。这是另一个重大的安全风险。

XOM (仅执行内存) : **XOM**是一种内存保护功能，可防止代码从标记为仅数据的内存区域执行。这是一种重要的安全机制，可以缓解缓冲区溢出攻击和其他攻击者可能试图将恶意代码注入数据区的漏洞。声明“No considering XOM”意味着，在这个特定的配置中，系统不强制执行仅限内存的限制。这是一个重大的安全风险。

之后

`query_in_pgtbl`、`map_range_in_pgtbl_common`、`unmap_range_in_pgtbl`

就是软件遍历pagetable的n重循环（n=levels），在中间判断 valid 和是否 超级块（block/table）即可

缺页管理

本节内容讲解Chcore中对缺页异常的处理，同时也会拓展ARM异常相关的知识。由于缺页异常会涉及到进程的虚拟地址区域（VMR/VMA）和物理内存对象（PMO）的相关知识，所以我们也会对这部分作相应的解析，以帮助大家学习。本节内容顺序下：

- ARM异常基础知识
- 缺页异常函数的源码解析
- VMR和PMO相关源码解析

ARM异常机制

异常类型

ARM将异常分为同步异常和异步异常两大类。同步异常是由指令执行直接引发的，例如系统调用、页面错误或非法指令等，这类异常具有**确定性**，每次执行到特定指令时都会触发。而异步异常包括硬件中断（IRQ）、快速中断（FIQ）和错误（ERROR），它们与当前指令无关，通常由**外部事件或硬件故障引起**

arm 把异常分成几种 SYNC, IRQ, FIQ, ERROR

这几个名字都很抽象，先讲SYNC同步异常

什么是同步异常? arm的手册给出的定义是：确定性的(每次执行到那个指令就会产生), 由执行或尝试执行指令引发的, 按照预期产生的; 而异步异常则是非确定性, 非指令性的, 预期之外的

例 电源被踹了一脚断电了, 或者时钟定时触发中断, 这是异步异常

而访问了不该访问的指令, 没有权限或者EL不对, (捕获浮点错误时的) 除0, 这些是可以溯源到某条指令上的, 是同步异常

EL1t和EL1h分别用于实际内核和虚拟机hypervisor模式

对于异步异常, 又可以分成中断和错误, 最终的概述 下

- **sync:** 同步异常, 系统调用或页面错误。
- **irq:** 硬件中断请求 (IRQ) , 由外部设备生成的中断。
- **fiq:** 快速中断请求 (FIQ) , 用于更高优先级的中断处理。
- **error:** 处理其他类型的错误, 未定义指令或故障。

linux ref: <https://www.cnblogs.com/charliechen114514-blogs/p/18455517>

处理逻辑

缺页异常说到底也是ARM异常的一个子集, 在学习缺页异常之前, 不妨先看看整体的异常管理设计逻辑是什么样的

根据异常类型和当前的执行模式 (内核态或用户态) , 设计相应的处理逻辑:

- **同步异常处理逻辑:**
 - **用户态 (EL0) 触发的同步异常:**
 - **目标:** 不能直接让内核崩溃, 必须妥善处理
 - **处理方式:** 根据异常类型回调对应的处理逻辑。例 , 对于页面错误, 可以实现需求分页或COW机制
 - **内核态 (EL1) 触发的同步异常:**
 - **目标:** 尝试修复一些提前设计的机制和可以处理的操作, 其他情况应导致系统崩溃
 - **处理方式:** 对于可修复的异常, 执行相应的修复逻辑; 对于不可修复的异常, 记录错误信息并触发系统崩溃
- **中断处理逻辑:**
 - **目标:** 快速响应并处理外部设备的中断请求
 - **处理方式:** 调用中断处理逻辑, 完成中断处理后返回到中断发生前的状态
- **错误处理逻辑:**
 - **目标:** 处理不可恢复的错误, 确保系统稳定性
 - **处理方式:** 记录错误信息并触发系统崩溃 (panic) , 以便进行后续的调试和分析

落实到实现本身, 则需要根据当前的异常级别寄存器 (EL0或EL1) 来区分内核态和用户态, 并在异常发生时保存上下文信息。

于是对于同步异常，就可以通过特定寄存器（FAR_ELx）获取产生异常的指令地址，从而进行调试和分析

缺页异常源码解析

有了对ARM异常机制与处理逻辑的基本了解，我们就可以进一步分析 `do_page_fault` 的源码了

可以参考代码中的注释，英文为自带的，中文为附加的助于理解的

```
void do_page_fault(u64 esr, u64 fault_ins_addr, int type, u64 *fix_addr)
{
    vaddr_t fault_addr;
    int fsc; // fault status code
    int wnr;
    int ret;

    // 从far_el1寄存器读取汇编
    fault_addr = get_fault_addr();
    // #define GET_ESR_EL1_FSC(esr_el1) (((esr_el1) >> ESR_EL1_FSC_SHIFT)
& FSC_MASK)
    fsc = GET_ESR_EL1_FSC(esr);
    switch (fsc) {
        case DFSC_TRANS_FAULT_L0:
        case DFSC_TRANS_FAULT_L1:
        case DFSC_TRANS_FAULT_L2:
        case DFSC_TRANS_FAULT_L3: {
            // 地址转换错误，根据vma进行进一步处理，也就是缺页异常
            ret = handle_trans_fault(current_thread->vm_space,
fault_addr);
            if (ret != 0) {
                // 没有正确处理
                /* The trap happens in the kernel */
                if (type < SYNC_EL0_64) {
                    // EL1 的 type，表示内核态的异常，跳转到no_context标签
                    goto no_context;
                }
            }
            // 用户态的异常处理失败，打印信息后退出
            kinfo("do_page_fault: faulting ip is 0x%lx (real
IP)",
                  "faulting address is 0x%lx,"
                  "fsc is trans_fault (0b%b),"
                  "type is %d\n",
                  fault_ins_addr,
                  fault_addr,
                  fsc,
                  type);
            kprint_vmr(current_thread->vm_space);

            kinfo("current_cap_group is %s\n",
                  current_cap_group->cap_group_name);

            sys_exit_group(-1);
        }
        break;
    }
    case DFSC_PERM_FAULT_L1:
    case DFSC_PERM_FAULT_L2:
    case DFSC_PERM_FAULT_L3:
        // 权限错误
        wnr = GET_ESR_EL1_WnR(esr);
        // WnR, ESR bit[6]. Write not Read. The cause of data abort.
        if (wnr) {
            //写权限错误
            ret = handle_perm_fault(
                  current_thread->vm_space, fault_addr,
```

```
VMR_WRITE);
    } else {
        //读权限错误
        ret = handle_perm_fault(
            current_thread->vmspace, fault_addr,
VMR_READ);
    }

    if (ret != 0) {
        /* The trap happens in the kernel */
        if (type < SYNC_EL0_64) {
            goto no_context;
        }
        sys_exit_group(-1);
    }
    break;
case DFSC_ACCESS_FAULT_L1:
case DFSC_ACCESS_FAULT_L2:
case DFSC_ACCESS_FAULT_L3:
// Access faults: 没有access bit的pte, 此处还不支持处理, 仅打印信息
    kinfo("do_page_fault: fsc is access_fault (0b%b)\n", fsc);
    BUG_ON(1);
    break;
default:
//默认处理流程, 指遇到了奇奇怪怪的的错误, 且系统当前还不支持处理它们
//因此这里的处理流程就是打印错误相关的信息, 然后触发内核panic终止之
    kinfo("do_page_fault: faulting ip is 0x%lx (real IP),"
          "faulting address is 0x%lx,"
          "fsc is unsupported now (0b%b)\n",
          fault_ins_addr,
          fault_addr,
          fsc);
    kprint_vmr(current_thread->vmspace);

    kinfo("current_cap_group is %s\n",
          current_cap_group->cap_group_name);

    BUG_ON(1);
    break;
}

return;
// no_context 这一名称来源于内核的异常处理流程。
// 当内核检测到异常发生在内核态时, 它发现没有“用户态上下文”
// (即不是用户程序引发的异常), 因此称之为 no_context
// 这只是一个逻辑分支, 用于区分内核态异常的处理流程
no_context:
    kinfo("kernel_fault: faulting ip is 0x%lx (real IP),"
          "faulting address is 0x%lx,"
          "fsc is 0b%b\n",
          fault_ins_addr,
          fault_addr,
          fsc);
    __do_kernel_fault(esr, fault_ins_addr, fix_addr);
}
```

```

static void __do_kernel_fault(u64 esr, u64 fault_ins_addr, u64 *fix_addr)
{
    kdebug("kernel_fault triggered\n");
    // 内核态page fault的时候，查表尝试修复，修复不了就终止内核
    if (fixup_exception(fault_ins_addr, fix_addr)) {
        return;
    }

    BUG_ON(1);
    sys_exit_group(-1);
}

```

对于这里好多种的switch分支，再统一分类说明一下：

转换错误 (Translation Fault)

```

case DFSC_TRANS_FAULT_L0:
case DFSC_TRANS_FAULT_L1:
case DFSC_TRANS_FAULT_L2:
case DFSC_TRANS_FAULT_L3:

```

这组错误表示**页表项不存在**，也就是本文的核心**缺页异常**，常见场景：

- 第一次访问堆区新分配的内存
- 访问未映射的内存区域
- 栈增长时的新页面访问

权限错误 (Permission Fault)

```

case DFSC_PERM_FAULT_L1:
case DFSC_PERM_FAULT_L2:
case DFSC_PERM_FAULT_L3:

```

这组错误表示**访问权限不足**，常见场景：

- 写入只读内存（代码段）
- 执行不可执行的内存
- 用户态访问内核内存

访问错误 (Access Fault)

```

case DFSC_ACCESS_FAULT_L1:
case DFSC_ACCESS_FAULT_L2:
case DFSC_ACCESS_FAULT_L3:

```

这组错误表示**硬件级别的访问失败**，常见场景：

- 访问未对齐的地址

- 硬件级别的内存访问限制
- TLB（页表缓存）相关错误

后面几种不必过多了解，感兴趣的可以进一步阅读相关源码

接下来我们学习VMR和PMO的相关代码，之后再回过头来梳理一遍我们的缺页异常处理流程

VMR & PMO

回顾Lab文档，我们知道：

在 ChCore 中，一个进程的虚拟地址空间由多段“虚拟地址区域”（VMR，又称 VMA）组成，一段 VMR 记录了这段虚拟地址对应的“物理内存对象”（PMO），而 PMO 中则记录了物理地址相关信息。因此，想要处理缺页异常，首先需要找到当前进程发生页错误的虚拟地址所处的 VMR，进而才能得知其对应的物理地址，从而在页表中完成映射。

我们先来看看VMR的数据结构是 何设计的（已经添加了详细的注释）：

```
/*
 * vmregion表示一个虚拟地址空间中的连续内存区域
 * 例：代码段、数据段、堆、栈等
 */
struct vmregion {
    /* 作为vmospace.vmr_list的节点，用于顺序遍历所有vmregion */
    struct list_head list_node;

    /* 作为vmospace.vmr_tree的节点，用于按地址快速查找vmregion */
    struct rb_node tree_node;

    /* 作为PMO的mapping_list的节点，用于跟踪所有映射到此PMO的vmregion */
    struct list_head mapping_list_node;

    /* 指向此vmregion所属的虚拟地址空间 */
    struct vmospace *vmspace;

    /* 此内存区域的起始虚拟地址 */
    vaddr_t start;

    /* 此内存区域的大小（字节数）*/
    size_t size;

    /* 在对应物理内存对象（PMO）中的偏移量 */
    size_t offset;

    /* 访问权限标志（：可读、可写、可执行等）*/
    vmr_prop_t perm;

    /* 指向此区域对应的物理内存对象 */
    struct pmoobject *pmo;

    /* 写时复制（CoW）机制中的私有页面链表 */
    struct list_head cow_private_pages;
};

/*
 * vmspace表示一个完整的虚拟地址空间
 * 通常对应一个进程的整个地址空间
 */
struct vmospace {
    /* vmregion链表的头节点，用于顺序遍历所有内存区域 */
    struct list_head vmr_list;

    /* vmregion红黑树的根节点，用于快速查找特定地址所在的内存区域 */
    struct rb_root vmr_tree;

    /* 指向此地址空间的页表根节点 */
    void *pgtbl;

    /* 进程上下文ID，用于避免TLB冲突 */
    unsigned long pcid;
};
```

```
/* 用于保护vmregion操作（增删改）的锁 */
struct lock vmspace_lock;

/* 用于保护页表操作的锁 */
struct lock pgtbl_lock;

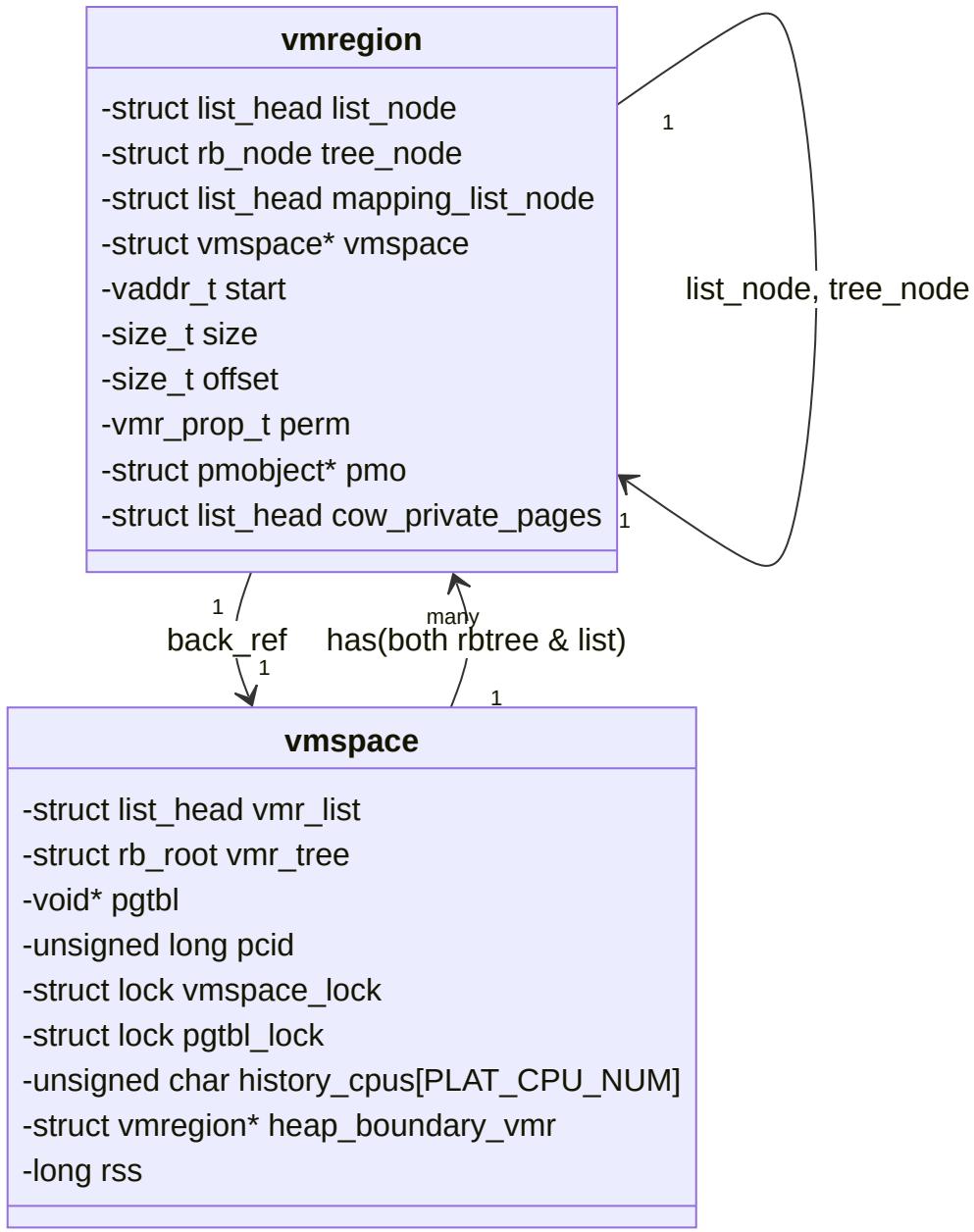
/*
 * TLB刷新相关：
 * 记录此vmspace在哪些CPU核心上运行过
 * 用于确定需要在哪些CPU上进行TLB刷新
 */
unsigned char history_cpus[PLAT_CPU_NUM];

/* 指向堆区域的边界vmregion，用于堆的动态扩展 */
struct vmregion *heap_boundary_vmr;

/*
 * 记录已映射的物理内存大小 (Resident Set Size)
 * 受pgtbl_lock保护
 */
long rss;
};
```

其实就是 vmregion 包含在 vmspace 里的关系，类比一本书和书里的不同章节

以下是示意图便于理解：



观察其设计不难发现一个“奇怪”的现象，那就是它同时维护了双向链表和红黑树的数据结构

这两种数据结构各有优劣，但若同时出现，则是一种“以空间换时间”的策略，以集众数据结构之长。虽然维护两套数据结构需要额外的内存空间和更新开销，但却能够在不同场景下都获得最优的性能表现。从缓存的视角上看，在扫描时，list能保证新插入的项被优先遍历，有更强的TLB亲和性，而红黑树的设计则保证了在查找特定元素时稳定的时间发挥。这种设计在Linux中同样被广泛采用

举两个例子分别说明两种情况，以下代码均出自chcore中vma的操作函数的源码

链表的情况

```

static void free_vmr_region(struct vmregion *vmr)
{
    struct cow_private_page *cur_record = NULL, *tmp = NULL;

    // 使用链表遍历所有CoW私有页面
    for_each_in_list_safe (cur_record, tmp, node, &vmr->cow_private_pages) {
        free_cow_private_page(cur_record);
    }
    list_del(&vmr->mapping_list_node);
    kfree((void *)vmr);
}

```

例 在freeVMR时，这种遍历的操作需求就很适合用链表来实现

红黑树的情况

```

struct vmregion *find_vmr_for_va(struct vmspace *vmspace, vaddr_t addr)
{
    struct vmregion *vmr;
    struct rb_node *node;

    // 使用红黑树快速查找地址对应的VMR
    node = rb_search(
        &vmspace->vmr_tree,
        (const void *)addr,
        cmp_vmr_and_va);

    if (unlikely(node == NULL))
        return NULL;

    vmr = rb_entry(node, struct vmregion, tree_node);
    return vmr;
}

```

涉及到查找的时候，就该红黑树大显神通了，利用封装好的函数和宏，兼具效率与代码可读性

PMO

在地址空间vmr中，还需要保证虚拟内存地址和物理地址的映射，从而避免查进程自身空间的页表，不需要保持内核页表和每个进程页表的项的对应。这就轮到PMO发挥作用的时候了，我们看看其数据结构的设计

```
/* This struct represents some physical memory resource */
struct pmobject {
    paddr_t start;
    size_t size;
    pmo_type_t type;
    /* record physical pages for on-demand-paging pmo */
    struct radix *radix;
    /*
     * The field of 'private' depends on 'type'.
     * PMO_FILE: it points to fmap_fault_pool
     * others: NULL
     */
    void *private;
    struct list_head mapping_list;
};
```

这里使用了start和size的结构，支持copy-on-writing和on demand paging。具体而言，声明时，只需要记录pmo 的start+size，在pmo之中维护访问过的/没访问的物理地址集合，在出现pagefault的时候分配，并更新这个集合就行。而对于 何维护这个集合的问题，chcore采用了 radix-tree的形式，在Linux中也有相似应用

在 Linux 内核中，radix tree（或其改进版本 xarray）被用于管理 page cache 和内存对象（ PMO，Physical Memory Object）时的地址到页面映射。这种选择的背后是对性能、功能和扩展性的权衡

似乎只用bitmap也能达到一样的效果，那么为什么不用bitmap呢？这是因为radix tree管理的pmo的地址空间通常是很**大一段稀疏的**（启用on demand paging）。这对bitmap非常不友好，而radix tree对稀疏和懒分配有很好的支持。此外，bitmap只能标记存在与否，而radix tree可以存指针，从而达到更灵活的元数据管理

回顾trans_fault的处理

有了VMR&PMO的知识，我们就可以进一步研究之前处理地址转换错误（也就是缺页异常）的时候的细节操作了：

```
int handle_trans_fault(struct vmspace *vmspace, vaddr_t fault_addr)
{
    struct vmregion *vmr;
    struct pmo *pmo;
    paddr_t pa;
    unsigned long offset;
    unsigned long index;
    int ret = 0;

    /*
     * Grab lock here.
     * Because two threads (in same process) on different cores
     * may fault on the same page, so we need to prevent them
     * from adding the same mapping twice.
     */
    lock(&vmspace->vmspace_lock);
    vmr = find_vmr_for_va(vmspace, fault_addr);

    if (vmr == NULL) {
        kinfo("handle_trans_fault: no vmr found for va 0x%lx!\n",
              fault_addr);
        dump_pgfault_error();
        unlock(&vmspace->vmspace_lock);
    }

#ifndef CHCORE_ARCH_AARCH64 || !defined(CHCORE_ARCH_SPARC)
    /* kernel fault fixup is only supported on AArch64 and Sparc
     */
    return -EFAULT;
#endif
    sys_exit_group(-1);
}

pmo = vmr->pmo;
/* Get the offset in the pmo for faulting addr */
offset = ROUND_DOWN(fault_addr, PAGE_SIZE) - vmr->start + vmr-
>offset;
vmr_prop_t perm = vmr->perm;
switch (pmo->type) {
case PMO_ANONYM:
case PMO_SHM: {
    /* Boundary check */
    BUG_ON(offset >= pmo->size);

    /* Get the index in the pmo radix for faulting addr */
    index = offset / PAGE_SIZE;

    fault_addr = ROUND_DOWN(fault_addr, PAGE_SIZE);

    pa = get_page_from_pmo(pmo, index);
    if (pa == 0) {
        /*
         * Not committed before. Then, allocate the physical
         * page.
         */
        void *new_va = get_pages(0);
        long rss = 0;
```

```
if (new_va == NULL) {
    unlock(&vmspace->vmspace_lock);
    return -ENOMEM;
}
pa = virt_to_phys(new_va);
BUG_ON(pa == 0);
/* Clear to 0 for the newly allocated page */
memset((void *)phys_to_virt(pa), 0, PAGE_SIZE);
/*
 * Record the physical page in the radix tree:
 * the offset is used as index in the radix tree
 */
kdebug("commit: index: %ld, 0x%lx\n", index, pa);
commit_page_to_pmo(pmo, index, pa);

/* Add mapping in the page table */
lock(&vmspace->pgtbl_lock);
map_range_in_pgtbl(vmspace->pgtbl,
                    fault_addr,
                    pa,
                    PAGE_SIZE,
                    perm,
                    &rss);
vmspace->rss += rss;
unlock(&vmspace->pgtbl_lock);
} else {
/*
 * pa != 0: the faulting address has been committed to
 * a physical page.
 *
 * For concurrent page faults:
 *
 * When type is PMO_ANONYM, the later faulting
threads
faulting
its
*
* of the process do not need to modify the page
* table because a previous faulting thread will do
* that. (This is always true for the same process)
* However, if one process map an anonymous pmo for
* another process (e.g., main stack pmo), the
*
* thread (e.g., in the new process) needs to update
* page table.
* So, for simplicity, we just update the page table.
* Note that adding the same mapping is harmless.
*
* When type is PMO_SHM, the later faulting threads
* needs to add the mapping in the page table.
* Repeated mapping operations are harmless.
*/
if (pmo->type == PMO_SHM || pmo->type == PMO_ANONYM)
{
    /* Add mapping in the page table */
    long rss = 0;
    lock(&vmspace->pgtbl_lock);
    map_range_in_pgtbl(vmspace->pgtbl,
                        fault_addr,
```

```

                pa,
                PAGE_SIZE,
                perm,
                &rss);
        vmspace->rss += rss;
        unlock(&vmspace->pgtbl_lock);
    }
}

if (perm & VMR_EXEC) {
    arch_flush_cache(fault_addr, PAGE_SIZE,
SYNC_IDCACHE);
}

break;
}

case PMO_FILE: {
    unlock(&vmspace->vmspace_lock);
    fault_addr = ROUND_DOWN(fault_addr, PAGE_SIZE);
    handle_user_fault(pmo, ROUND_DOWN(fault_addr, PAGE_SIZE));
    BUG("Should never be here!\n");
    break;
}
case PMO_FORBID: {
    kinfo("Forbidden memory access (pmo->type is PMO_FORBID).
\n");
    dump_pgfault_error();

    unlock(&vmspace->vmspace_lock);
    sys_exit_group(-1);
    break;
}
default: {
    kinfo("handle_trans_fault: faulting vmr->pmo->type"
          "(pmo type %d at 0x%lx)\n",
          vmr->pmo->type,
          fault_addr);
    dump_pgfault_error();

    unlock(&vmspace->vmspace_lock);
    sys_exit_group(-1);
    break;
}
}

unlock(&vmspace->vmspace_lock);
return ret;
}
}

```

结合这个处理函数，我们就可以构建出chcore对缺页异常整体的处理了：

- 发生缺页异常，触发 `do_page_fault` 函数
- 异常在switch分支中被归类为 `DFSC_TRANS_FAULT_LX`，且会根据是否是内核态的错误作进一步的处理
- 函数获取当前的虚拟地址和vmspace，将异常转发给 `handle_trans_faults` 函数

- 有了上述二物，处理函数用红黑树查找到具体的vmregion，并得到对应的PMO
 - 根据PMO的类型作进一步的处理： 匿名页面和共享内存 PMO_ANONYM, PMO_SHM、文件映射 PMO_FILE、禁止访问的内存 PMO_FORBID
-

这里同时需要注意可能出现并发的pagefault,其语义处理根据不同type发生变化： 果是同个进程的多个线程，且类型为 PMO_ANONYM，那只需要第一个线程更新radix即可； 果是跨进程的线程，则需要各自更新

- 对于匿名/共享内存，需要查询其在radix tree中是否已经存在记录。 果存在，就只需要在自己的页表中设置页表映射; 否则为这个on demand paging的页面分配空间并更新
-

在PHO_SHM共享内存的时候，多个进程的物理页面是相同的，即各自的vma引用同一个pmo。所以并发场景下后来的线程会出现pa已经在radix之中存在的情况

- 刷新指令缓存，处理完毕

至此，缺页管理部分的源码解析也到此结束，希望能对你的学习有所裨益！

Last change: 2025-02-14, commit: [9fa9752](#)

教材补充

1. 操作系统是如何管理物理内存资源的？

- 1.1 内存碎片
- 1.2 基于位图的连续物理页分配方法
- 1.3 伙伴系统原理
- 1.4 ChCore中伙伴系统的实现
- 1.5 SLAB分配器的基本设计
- 1.6 常用的空闲链表

2. 操作系统如何获得更多物理内存资源？

- 2.1 换页机制
- 2.2 页替换策略
- 2.3 页表项中的访问位与页替换策略实现
- 2.4 利用虚拟内存抽象节约物理内存资源

3. 性能导向的内存分配扩展机制

- 3.1 缓存结构
- 3.2 物理内存分配与CPU缓存
- 3.3 多核与内存分配
- 3.4 非一致内存访问 (NUMA架构)

操作系统是如何管理物理内存资源的？



在本节中，我们先要了解操作系统的物理分配内存设计的两个评价维度，即减少资源浪费和追求优秀性能，优秀的物理内存分配器要做到两方面兼顾。有了这样的评价标准，接下来，我们介绍基于位图的连续物理页分配方法，这是一种我们方便理解的简单分配器，从中我们可以初窥物理页分配的具体实现。但是这种方法存在着诸多缺点，显然无法满足我们的需求，于是我们需要寻求更加高效的物理页分配方法，这就引出了我们本节的重点。

——伙伴系统原理。我们会详细向你介绍伙伴系统原理的具体内容，并结合代码分析 ChCore 中对于伙伴系统原理的实现。但是走到这里并没有万事大吉，因为伙伴系统的最小分配单位是一个物理页（4KB），大多数情况下会产生较多的内部碎片，浪费很多内存，因此为了分配小内存，开发人员设计了另外一套内存分配机制——SLAB 分配器。除了上述两种内存分配方法，还有其他基于不同空闲链表的内存分配方法，我们将在最后为你介绍三种常见的空闲链表。

内存碎片

我们知道操作系统的物理内存分配需要兼顾优秀的性能和减少资源浪费。前者主要是尽可能降低分配延迟和节约CPU资源，后者主要是考虑内存碎片问题。内存碎片又分为外部碎片和内部碎片。

- 外部碎片：多次分配和回收之后，物理内存上的空闲部分处于离散分布的状态，可能导致系统存在足够的空闲内存，却无法满足内存分配请求。
- 内部碎片：分配内存大于实际使用内存。二者都会造成内存浪费现象。

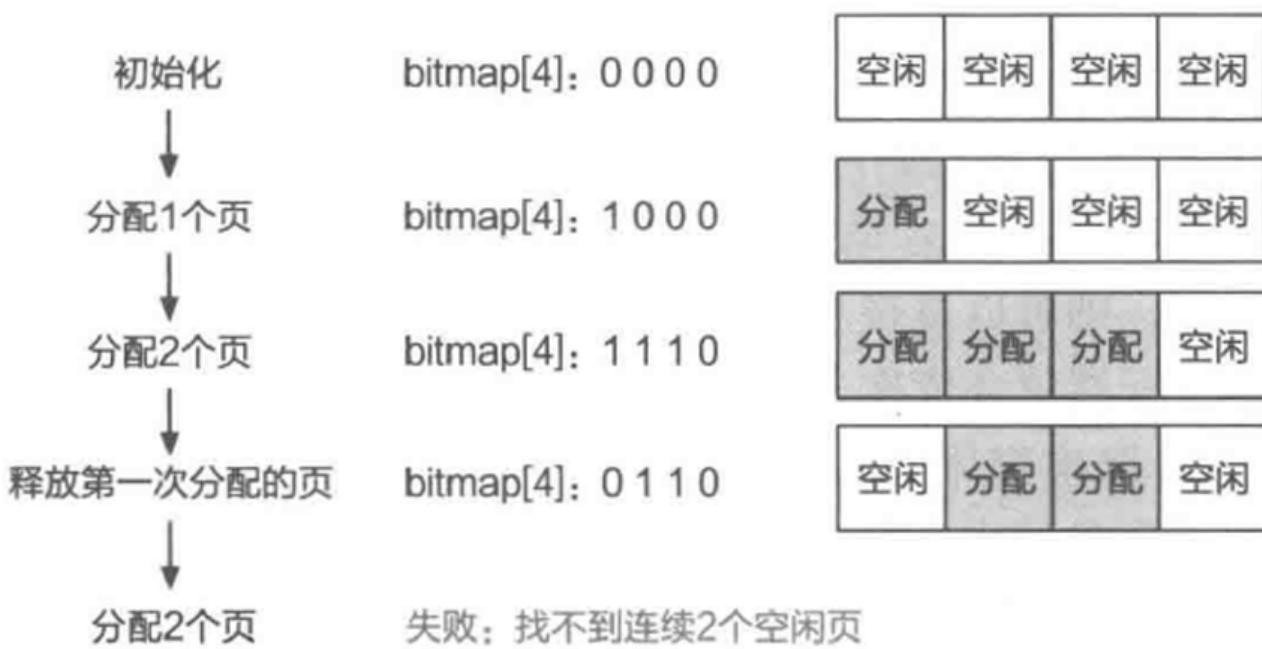
基于位图的连续物理页分配方法

- 基本思想：初始化一个位图，每一位对应一个物理页，若为0则表明相应的物理页空闲，反之则为已分配；在分配时查找位图，找到为0的位，分配相应的物理页，并且把该位设置为1。若分配连续的物理页，则只需要查找连续n位为0的位置，在位图中把相应的物理页标记为1，并返回其中起始物理页的地址即可。

```
1 // 共有 N 个 4K 物理页，位图中的每一个对应一个页
2 bit bitmap[N];
3
4 void init_allocator(void)
5 {
6     int i;
7     for (i = 0; i < N; ++i)
8         bitmap[i] = 0;
9 }
10
11 // 分配 n 个连续的物理页
12 u64 alloc_pages(u64 n)
13 {
14     int i, j, find;
15
16     for (i = 0; i < N; ++i) {
17         find = 1;
18
19         // 从第 i 个物理页开始判断连续 n 个页是否空闲
20         for (j = 0; j < n; ++j) {
21             if (bitmap[i+j] != 0) {
22                 find = 0;
23                 break;
24             }
25         }
26
27         if (find) {
28             // 将找到的连续 n 个物理页标记为已分配
29             for (j = i; j < i+n; ++j)
30                 bitmap[j] = 1;
31             // 返回第 i 个物理页的起始地址
32             return FREE_MEM_START + i * 4K;
33         }
34     }
35
36 // 分配失败
37 return NULL;
38 }
39
40 // 释放 n 个连续的物理页
41 void free_pages(u64 addr, u64 n)
42 {
43     int page_idx;
44     int i;
45
46     // 计算待释放的起始页索引
47     page_idx = (addr - FREE_MEM_START) / 4K;
48
49     for (i = 0; i < n; ++i) {
50         bitmap[page_idx + i] = 0;
51     }
52 }
```

这种简单分配器可以实现我们的基本需求，但是在前文介绍的两个评价维度上都存在不足。

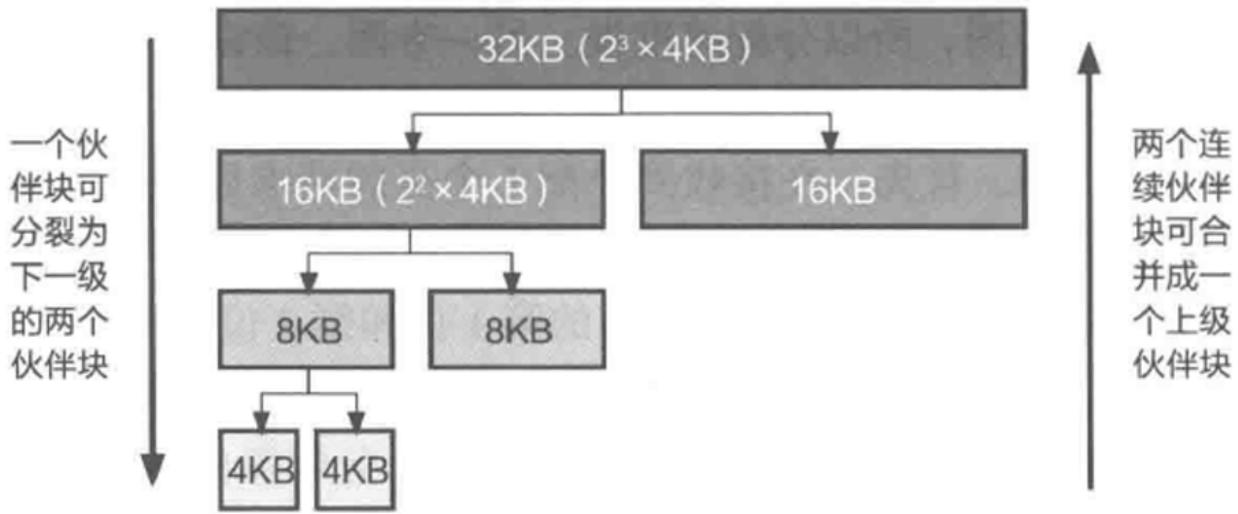
- 分配速度慢，需要依次查询整个位图。
- 导致外部碎片问题



图所示，如果先分配一个页，再分配连续的两个页，此时回收第一次分配的一个页，就会导致出现两块独立的空闲块，当有一个2个页的内存请求到达时就会产生失败。

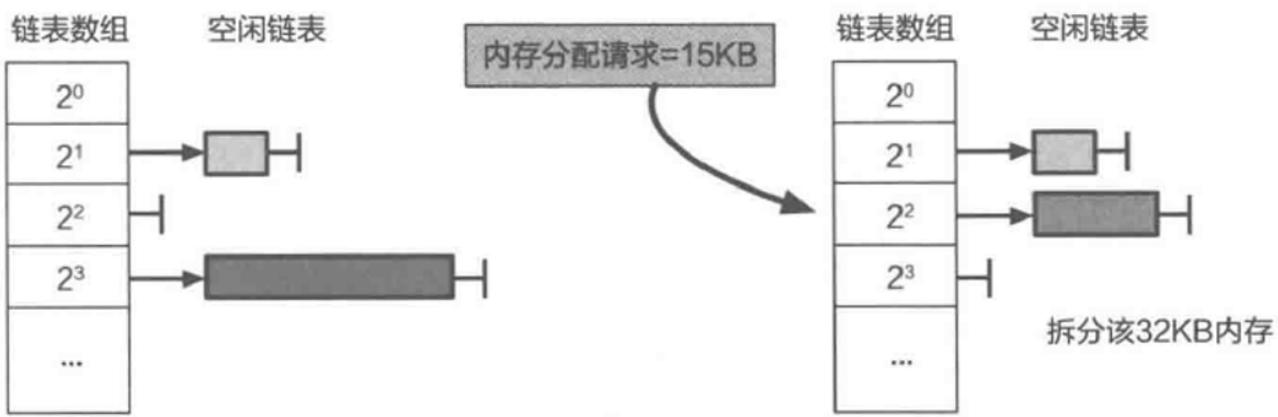
伙伴系统原理

伙伴系统原理在现代操作系统中被广泛地用于分配连续的物理内存页。



- 基本思想将物理内存划分成连续的块，以块作为基本单位进行分配。每个块都有一个或多个连续的物理页组成，物理页的数量必须是2的n次幂。当一个请求需要分配m个物理页时，伙伴系统将寻找一个大小合适的块，该块包含 2^n 个物理页，且满足 $2^{n-1} < m \leq 2^n$ 。在处理分配请求的过程中，如果没有合适大小的块，我们可以找到更大的块，将其分裂成两半，即两个小一号的块，这两个块互为伙伴。分裂得到的块可以继续分裂，直到得到

一个大小合适的块去服务相应的分配请求。在一个块释放后，分配器会找到其伙伴块，若伙伴块也处于空闲状态，则将这两个伙伴块进行合并，形成一个大一号的空闲块，然后继续尝试向上合并。由于分裂操作和合并操作都是级联的，因此能够很好地缓解外部碎片的问题。



伙伴系统的实现需要用到上图所示的空闲链表数组。全局有一个有序数组，数组中的每一项指向一个空闲链表，每条链表将其对应大小的空闲块连接起来（一个链表中的空闲块大小相同）。当接收到分配请求之后，伙伴分配器首先算出应该分配多大的空闲块，然后查找对应的空闲链表。

- 在上述例子中，我们有一个15KB的内存分配请求，我们计算出应该取一个16KB的块，我们定位到对应的链表，发现该链表为空，因此我们查找下一个更大的链表，于是取出空闲块进行分裂操作，获得两个16KB大小的空闲块，其中一个用于服务请求，另一个依然作为空闲块插入空闲的链表中。

当接收到释放块的请求时，分配器首先找到待释放块的伙伴块。如果伙伴块处于非空闲状态，则将被释放的块直接插入对应大小的空闲链表中，即完成释放；如果伙伴块处于空闲状态，则将两个块进行合并，当成一个完整的块释放，并重复该过程。值得注意的是，在合并过程中寻找伙伴块的方法非常高效。

- 互为伙伴的两个块，它们的内存地址仅有一位不同，且该位由块大小决定。所以在已知一个内存块地址的前提下，只需要翻转该地址中的一位就可以得到其伙伴块的地址，从而能够快速判断是否需要合并。举例来说，块 A (0~8KB) 和块 B (8~16KB) 互为伙伴块，它们的物理地址分别是 0x0 和 0x2000，根据直接映射方式，它们在内核地址空间中的虚拟地址分别为固定偏移 +0x0 和固定偏移 +0x2000，仅有第 13 位不同，而块大小是 8KB。

ChCore中伙伴系统的实现

```
struct physical_page {
    // 是否已经分配
    int allocated;
    // 所属伙伴块大小的幂次
    int order;
    // 用于维护空闲链表，把该页放入/移出空闲链表时使用
    list_node node;
};

// 伙伴系统的空闲链表数组
list free_lists[BUDDY_MAX_ORDER];
```

上面展示了表示物理页的结构体，其中每个结构体对应一个物理页。

```
// 伙伴系统初始化
void init_buddy(struct physical_page *start_page, u64 page_num)
{
    int order;
    int index;
    struct physical_page *page;

    // 初始化物理页结构体数组
    for (index = 0; index < page_num; ++index) {
        page = start_page + index;
        // 标记成已分配
        page->allocated = 1;
        page->order = 0;
    }

    // 初始化伙伴系统的各空闲链表
    for (order = 0; order < BUDDY_MAX_ORDER; ++order) {
        init_list(&(free_lists[order]));
    }

    // 通过释放物理页的接口把物理页插入伙伴系统的空闲链表
    for (index = 0; index < page_num; ++index) {
        page = start_page + index;
        buddy_free_pages(page);
    }
}
```

上面展示了伙伴系统的初始化函数 `init_buddy`，他会依次初始化每个 `physical_page` 结构体。在这个过程中，每个结构体会被标记为已分配（`allocated` 设置为 1）且所属伙伴块仅有 一个物理页（`order` 设置为 0）。之后，调用 `init_list` 来初始化各个大小的空闲链表。然后，`init_buddy` 函数将利用物理页释放接口依次释放每个物理页（函数中最后一个 `for` 循环），在释放每个物理页的过程中，空闲的伙伴块会被级联地合并，并且插入伙伴系统相应的空闲链表中。

```

// 分配伙伴块: 2^order 数量的连续 4K 物理页
struct page *buddy_alloc_pages(u64 order)
{
    int cur_order;
    struct list_head *free_list;
    struct page *page = NULL;

    // 搜索伙伴系统中的各空闲链表
    for (cur_order = order; cur_order < BUDDY_MAX_ORDER; ++cur_order) {
        free_list = &(free_lists[cur_order]);
        if (!list_empty(free_list)) {
            // 从空闲链表中取出一个伙伴块
            page = get_one_entry(free_list);
            break;
        }
    }

    // 若取出的伙伴块大于所需大小，则进行分裂
    page = split_page(order, page);
    // 标记已分配。示意代码忽略分配失败的情况
    page->allocated = 1;
    return page;
}

// 释放伙伴块
void buddy_free_pages(struct page *page)
{
    int order;
    struct list_head *free_list;

    // 标记成空闲
    page->allocated = 0;
    // (尝试)合并伙伴块
    page = merge_page(page);

    // 把合并后的伙伴块放入对应大小的空闲链表
    order = page->order;
    free_list = &(free_lists[order]);
    add_one_entry(free_list, page);
}

```

上述代码介绍了 `buddy_alloc_pages` 和 `buddy_free_pages` 的实现逻辑。

`buddy_alloc_pages` 接收 `order` 参数，用于分配 2^{order} 个连续物理页。

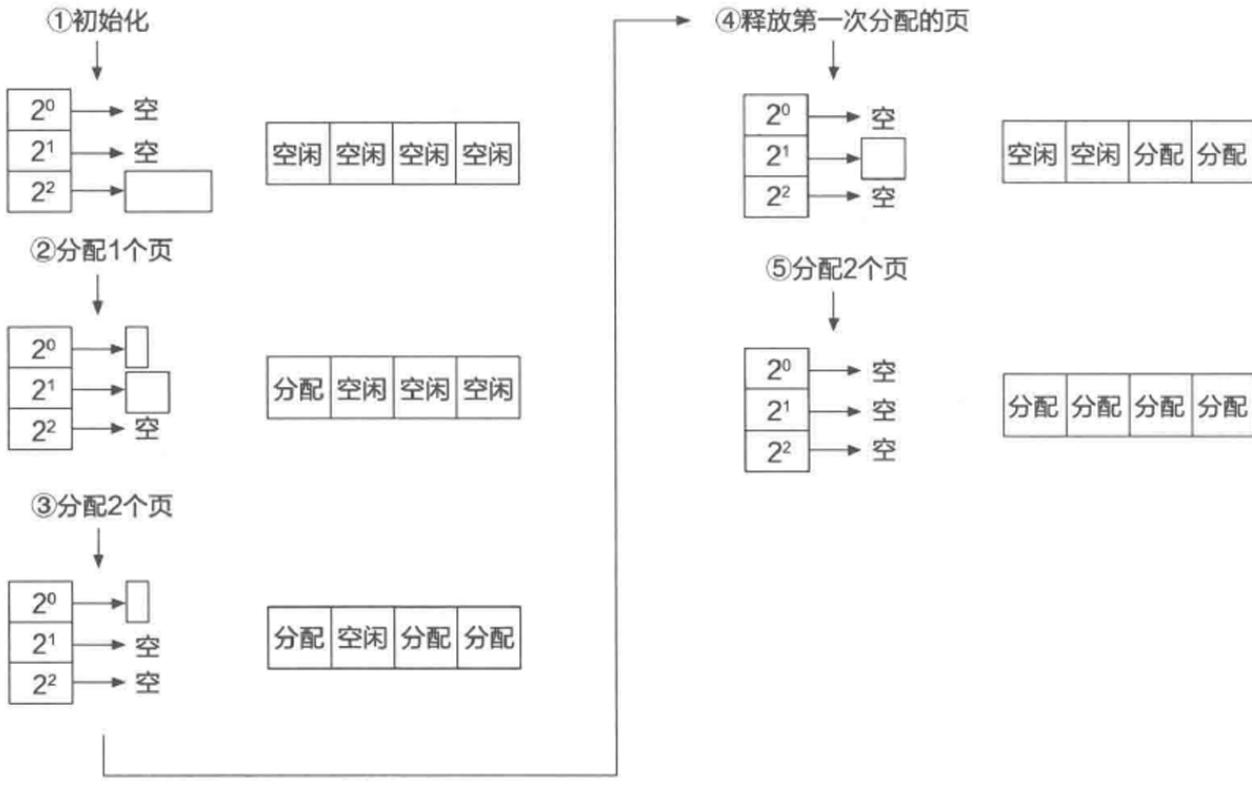
- `order`：表示需要分配的块大小为 2^{order} 个物理页（例 `order=3` 表示分配 8 个页）。
- `free_list`：指向不同大小空闲链表的指针。
- `page`：最终返回的分配块。
- 第一个 `for` 循环的作用是从 `order` 对应的链表开始查找，若为空则尝试更大的块 (`order+1`)。找到非空链表后，调用 `get_one_entry` 取出一个空闲块。如果取出

的块大小大于所需（例 分配 2 个页但取出的是 8 个页的块），则递归分裂成更小的块。
分裂后，将多余的块插入对应链表（例 分裂出 4 个页的块插入 order=2 的链表）。

- 仅将分配块的第一个物理页标记为已分配，而不修改其余页对应的 `physical_page` 结构体。这能够减少内存修改操作从而提升性能，而这么做可行的原因是这些页在被分配后不位于伙伴系统的任一空闲链表中（因此不会被再次分配）。

函数 `buddy_free_pages` 释放内存块，并尝试合并伙伴块以减少碎片。

- 将被释放块的第一个页标记为空闲，然后检查伙伴块（相邻且大小相同的块）是否空闲，若空闲则合并成更大的块。合并过程可能递归进行，直到无法合并为止。根据合并后块的大小（`order`），将其插入对应的空闲链表。



在我们学习了上述实现之后，我们对比一下我们在“基于位图的连续物理页分配方法”小节中给出的例子。

当接收到分配 **1 个物理页** 的请求时，`buddy_alloc_pages` 函数会对上述 4 页的空闲块进行级联拆分：首先将 4 页块拆分为两个 **2 页块**，然后进一步将其中一个 2 页块拆分为两个 **1 页块**。分配其中一个 1 页块后，空闲链表中剩余一个 **2 页块** 和一个 **1 页块**。

接下来，当收到分配 **2 个连续物理页** 的请求时，伙伴系统直接返回剩余的 2 页块。

当释放首次分配的 1 页块时，系统会检查其伙伴块：若伙伴块（另一个 1 页块）也处于空闲状态，则将两者合并为一个 **2 页块**，并插入对应链表。

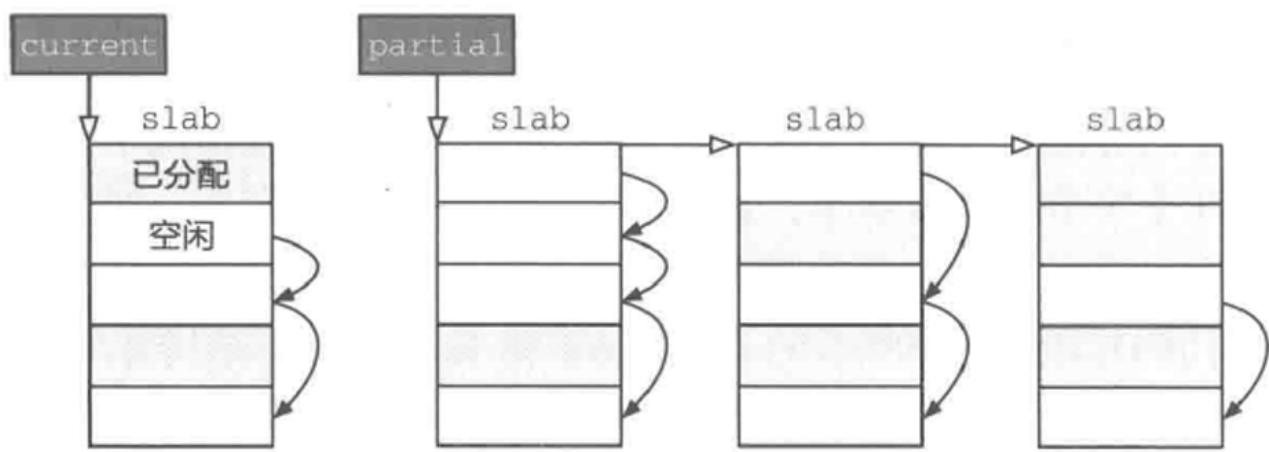
最后，若再次请求分配 **2 个连续物理页**，伙伴系统可直接提供合并后的 2 页块。从而完美避免了外部碎片的问题。

SLAB分配器的基本设计

SLAB分配器用于在操作系统中分配小内存。

本节介绍一种常用的SLAB分配器——SLUB。

SLUB分配器将从伙伴系统分配的大块内存进一步细分成小块内存进行管理。块大小通常为 2^n 字节 ($3 \leq n < 12$, 8B、16B、...、4KB)。支持自定义大小以优化内部碎片（例如根据实际需求设置96B或128B）。对于每一种块大小，SLUB分配器都会使用独立的内存资源池进行分配。下面我们介绍内存资源池的相关知识。



上图所示。SLUB 分配器向伙伴系统中请求一定大小的物理内存块（一个或多个连续的物理页），并将获得的物理内存块作为一个 slab。

- slab 会被划分成等长的小块内存，并且将其内部空闲的小块内存组织成空闲链表的形式。

一个内存资源池会维护两个指针，分别是 current 和 partial。

- current 指针仅指向一个 slab，所有的分配请求都将从该指针指向的 slab 中获得空闲内存块。partial 指针指向由所有拥有空闲块的 slab 组成的链表。
- 分配流程：1、根据请求大小，找到能满足需求且块大小最接近的内存资源池。2、从 current 指针指向的 slab 中取出第一个空闲块返回。3、**若当前 slab 无剩余空闲块（全部分配完）**：a. 从 partial 指针指向的链表中取出一个 slab。b. 将该 slab 设为新的 current 指针指向的 slab。**若 partial 链表为空：**a. 向伙伴系统申请新的物理内存块，生成新 slab。b. 将新 slab 设为 current 指针指向的 slab。
- 释放流程：1、将被释放的内存块插入对应 slab 的空闲链表。2、**若该 slab 原本已全部分配（释放前无空闲块）**：将该 slab 移动到 partial 指针指向的链表。**若该 slab 完全空闲（所有块均被释放）**：将整个 slab 返还给伙伴系统。3、何找到释放块所属的 slab？可以在 slab 头部加入元数据并且使得 slab 头部具有对其属性。

常用的空闲链表

- 隐式空闲链表将所有内存块（包括空闲和已分配的）串联在一个链表中。每个块的头部存储了块的大小和空闲状态信息，通过块大小可以定位到下一个块的位置。分配时需遍历整个链表找到第一个足够大的空闲块，若块过大则分割使用；释放时会检查相邻块是否空闲并进行合并。其优点是实现简单，无需额外空间存储指针，但分配效率较低，时间复杂度与总块数正相关，且容易产生外部碎片。
- 显式空闲链表仅将空闲块串联在链表中，每个空闲块通过前驱（prev）和后继（next）指针连接，指针直接存储在空闲块的数据区域。分配时只需遍历空闲块，找到合适块后分割使用；释放时插入链表并检查合并相邻块。由于仅需处理空闲块，分配效率高于隐式链表（时间复杂度与空闲块数正相关），尤其在内存使用率高时优势明显，但需依赖指针维护链表结构。
- 分离空闲链表在显式链表基础上进一步优化，通过维护多个显式链表，每个链表管理特定大小范围的内存块（8B、16B等）。分配时根据请求大小选择对应链表，若当前链表无合适块则向上查找更大的块链表，分割后的剩余部分插入对应链表；释放时合并相邻块后按大小归位。其优势在于分配速度快（搜索范围缩小）、内存利用率高（近似最优匹配），且支持并发操作。这是显式链表的进阶形式，适用于高性能场景，但实现复杂度更高，需维护多个链表及分链策略。

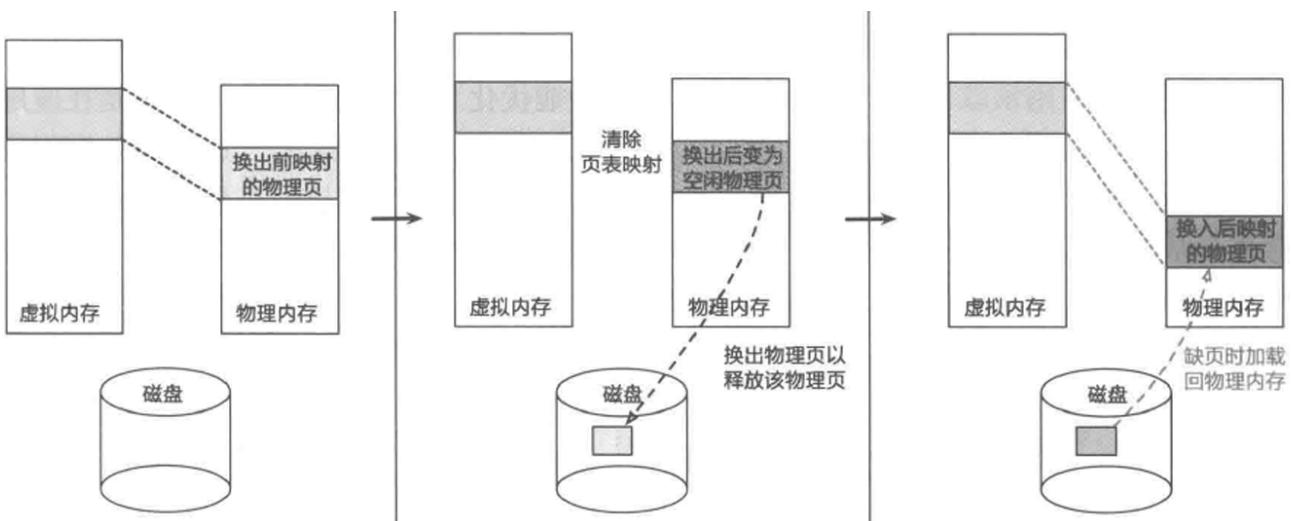
类型	管理对象	时间复杂度	碎片控制	适用场景
隐式空闲链表	所有内存块	O(总块数)	弱（需手动合并）	简单场景，内存需求不频繁
显式空闲链表	仅空闲块	O(空闲块数)	中等	内存使用率高，分配频繁
分离空闲链表	分大小管理空闲块	O(1) ~ O(分链数)	强（分链优化）	高性能、高并发系统

操作系统如何获得更多物理内存资源？



本节概要物理内存不够用时怎么办？——换页机制 何选择页面进行替换？——页替换策略
何利用虚拟内存抽象节约物理内存？——内存去重/内存压缩

换页机制



- 基本思想:** 当物理内存容量不足时，操作系统会将一些物理页的内容写入磁盘等大容量存储设备中，从而回收这些物理页以供其他进程使用。
- 换出 (Swap Out):** 当操作系统需要回收某个物理页时，会将该页的内容写入磁盘，并在页表中移除对应的虚拟页映射，同时记录该物理页在磁盘上的位置。
- 缺页异常:** 当进程访问一个已被换出的虚拟页时，由于页表中没有映射，CPU会触发缺页异常。
- 换入 (Swap In):** 操作系统的缺页异常处理函数会分配一个新的物理页，将磁盘上的数据重新加载到该物理页中，并在页表中建立虚拟页到物理页的映射。之后，进程可以从触发异常的指令处继续执行。

操作系统是用完所有物理页才进行换页的吗？ ——可能造成资源紧张时许多物理页分配操作都需要先进行换页，从而造成分配延时高的问题。**解决方法？** ——设置阈值。

- 当空闲物理页数量小于低水位线时，择机进行换页操作，目标是把空闲物理页数量恢复到高水位线；当空闲物理页数量低于最小水位线时，则立即进行换页操作，且批量换出。

换页机制使得操作系统能够获取更多物理内存资源，是否牺牲了什么作为代价呢？ ——磁盘操作会带来性能损失。

如何减少换页机制带来的性能损失？ ——预取机制。

- 基本思想:** 预取机制通过在应用进程发生缺页异常之前，提前将可能需要的内存页换入物理内存，从而避免因磁盘操作导致的性能下降。
- 实现方式:** 操作系统需要预测应用进程在执行过程中可能会访问哪些内存页。一种常见的预测方法是基于空间局部性特征，即假设进程在访问某个内存页后，可能会继续访问相邻的下一个内存页。
- 操作步骤:** 当操作系统需要换入某个内存页时，会同时将相邻的下一个内存页一并换入，以提前准备可能需要的资源。
- 如果预取机制预测准确，可以减少应用进程发生缺页异常的次数，从而提升系统性能；如果预测不准确，预取机制可能会导致不必要的换入操作，占用磁盘读取带宽和内存资源，反而拖累系统性能。

操作系统 何判断页表中没有映射是由于按需页面分配导致的还是换页导致的呢？

——操作系统通过在页表项中做标记来区分这两种情况：

- **按需页面分配**: 页表项内容为0，表示该虚拟页从未被分配过物理页。
- **换页机制**: 页表项的第0位（最低位）设置为0，表示该页已被换出，其余位记录页面在磁盘上的位置。

页替换策略

页替换策略依据硬件所提供的页面访问信息来猜测哪些页面应该被换出，从而最小化缺页异常的发生次数以提升性能。

- **MIN/OPT策略**: MIN策略（又称最优策略）选择未来最长时间内不会被访问的页面进行换出。虽然理论上是最优的，但由于无法预知未来的页面访问顺序，实际中难以实现，主要用于衡量其他替换算法的优劣。
- **FIFO策略**: FIFO（先入先出）策略选择最先进入内存的页面进行换出。它维护一个队列记录页面进入顺序，简单且开销低，但性能通常不佳，因为页面进入顺序与使用频率无关。
- **Second Chance策略**: 这是FIFO策略的改进版本，增加了访问标志位。如果页面被访问过，则将其标志位清零并移到队尾，避免立即换出。该策略在考虑页面访问信息后，通常优于FIFO策略。
- **LRU策略**: LRU（最近最少使用）策略选择最久未被访问的页面进行换出。它基于局部性原理，认为最近频繁访问的页面未来也可能被频繁访问。实现上需要维护一个按访问时间排序的链表，开销较大。
- **时钟算法策略**: 时钟算法将页面排成环形，使用一个指针检查页面的访问标志位。如果页面未被访问过，则换出；否则清零标志位并移动指针。相比Second Chance策略，时钟算法实现更高效。
- **随机替换策略**: 随机替换策略任意选择一个页面进行换出。它不需要维护页面访问信息，实现简单，但通常会导致更多的缺页异常，性能不如其他策略。

页表项中的访问位与页替换策略实现

```
// 在 physical_page 结构体中新增成员变量
struct physical_page {
    ...
    // 记录该物理页被映射到哪些页表项（称为反向映射）
    list pgtbl_entries;
};

struct physical_page pages[NUM_PHYSICAL_PAGE];

void add_mapping(u64 pgtbl, u64 va, u64 pa)
{
    ...
    // 记录物理页被填写到哪个页表项
    struct physical_page *page = &pages[pa/PAGE_SIZE];
    add_one_entry(page->pgtbl_entries, pgtbl_entry);
}

// 参数是需要换出的物理页数量
void scan_and_swap(int num_page_to_swap)
{
    int swap;

    // 获取需要遍历的物理页区间（例 物理页号为 0 ~ 1000 的区间）
    scan_range = get_scan_range();

    while (num_page_to_swap != 0) {
        for i in scan_range:
            swap = 1;
            list pgtbl_entries = pages[i]->pgtbl_entries;

            // 利用 for_each_pte 宏遍历某物理页反向映射中的每个页表项
            for_each_pte(pgtbl_entry, pgtbl_entries) {
                if (is_accessed(pgtbl_entry)) {
                    // 将页表项中的访问位清零
                    clear_access(pgtbl_entry);
                    swap = 0;
                }
            }

            if (swap) {
                // 把第 i 个物理页换出
                swap_out(i);
                --num_page_to_swap;
            }
    }
    tlb_flush();
}
```

通过 `scan_and_swap` 函数，系统会将未被访问的物理页交换出去，避免频繁访问未使用的物理页。为提高效率，只有访问过的虚拟页才会被映射，且当TLB缓存翻译后，MMU会更新页表项，避免重复访问未使用的页，从而减少内存访问冲突。

利用虚拟内存抽象节约物理内存资源

- **内存去重**: 基于写时拷贝机制，操作系统通过扫描内存中内容相同的物理页面（全零页），将多个虚拟页面映射到同一物理页，并释放冗余物理页以节省内存。该功能对用户透明，但可能因写入时触发缺页异常和内存拷贝导致性能下降。例如，Linux的KSM (Kernel Same-page Merging) 支持跨进程或进程内的页面合并，属于共享内存的一种实现。
- **内存压缩**: 在内存资源不足时，操作系统将“近期较少使用”的内存页数据压缩存储于内存中（而非换出到磁盘），以快速释放空间。访问时直接解压即可恢复，减少了磁盘I/O延迟。例如，Linux的zswap机制将待换出数据压缩后暂存于内存缓冲区，通过延迟或批量处理磁盘操作提升效率，同时压缩降低了读写数据量，兼顾了内存利用率和响应速度。

性能导向的内存分配扩展机制



本节概要为什么需要CPU缓存？CPU缓存的结构是怎样的？物理内存中的数据远大于CPU缓存的大小，采用何种策略进行物理页分配呢？多核和NUMA对于内存分配有什么影响？

相比于CPU执行的速度，内存访问速度是非常缓慢的：一条算术运算指令可能只需要一个或几个时钟周期即可完成，而一次内存访问则可能需要花费上百个时钟周期。如果每条内存读写指令都需要通过总线访问物理内存，那么CPU与物理内存之间的数据搬运可能成为显著的性能瓶颈。

为了降低访存的开销，现代CPU内部通常包含CPU缓存(CPU Cache)，用于存放一部分物理内存中的数据。访问CPU缓存比访问物理内存快很多，一般最快只需要几个时钟周期。当CPU需要向物理内存写入数据时，可以直接写在CPU缓存之中；当CPU需要从物理内存读取数据时，可以先在CPU缓存中查找，如果没找到再去物理内存中获取，并且把取回的数据放入缓存中，以便加速下次读取。由于程序在运行时访问物理内存数据通常具有局部性（包括时间局部性和空间局部性），因此缓存能够有效提升CPU访问物理内存数据的性能。

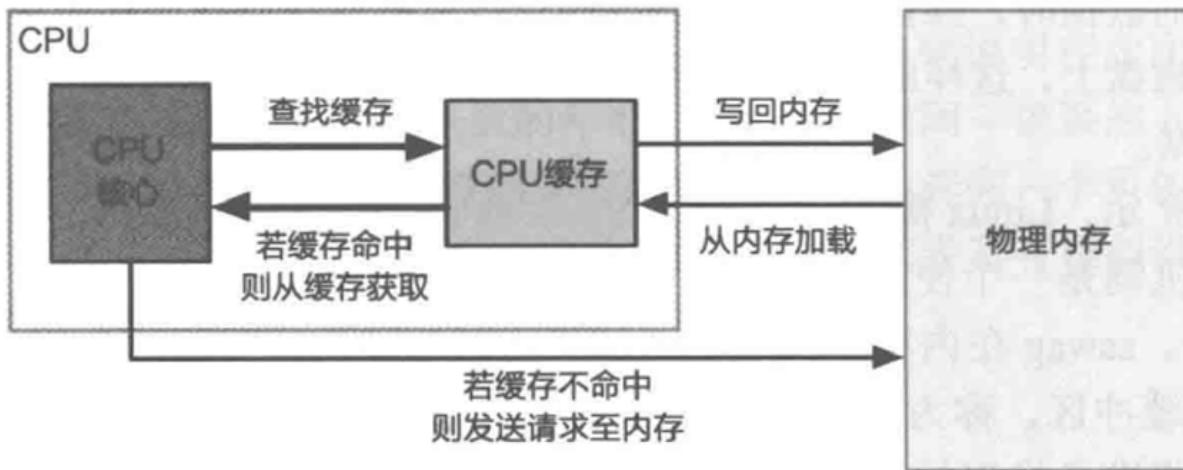


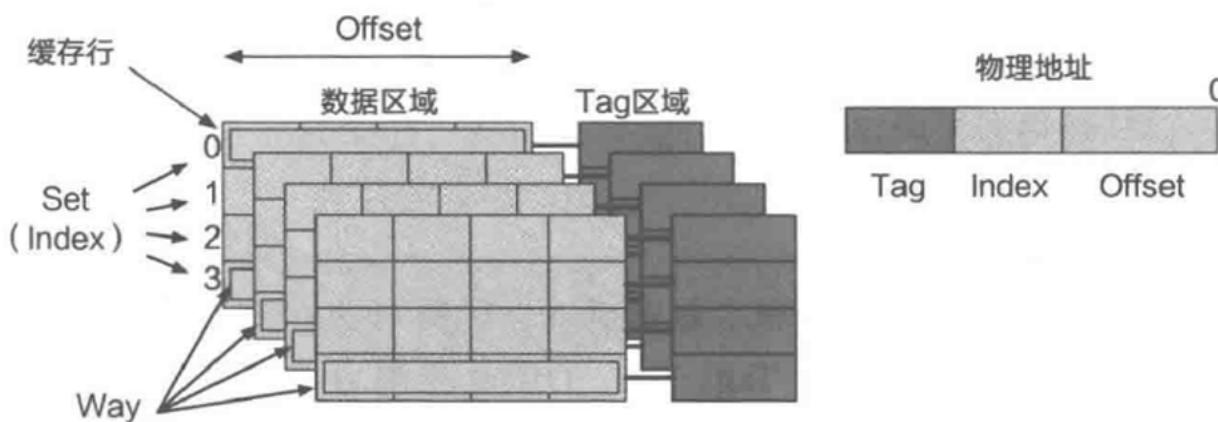
图 5.10 CPU 缓存与内存结构

缓存结构

1、CPU缓存中包含若干条**缓存行**和**每条缓存行相应**的状态信息。

- **缓存行**: CPU 缓存的基本单位，通常为 64 字节。CPU 以缓存行为粒度读取和写回物理内存中的数据。
- **状态信息**: 每条缓存行包含状态信息，包括：
 - **有效位 (Valid Bit)** : 表示该缓存行是否有效。
 - **标记地址 (Tag Address)** : 标识该缓存行对应的物理地址。
 - **其他信息**: 可能包括缓存行的其他状态信息。

2、典型的CPU缓存结构



- **物理地址划分**: 物理地址在逻辑上分为三段：

- **Tag**: 标记地址，用于标识缓存行对应的物理地址。
- **Set (Index)** : 组索引，用于确定缓存行所在的组。
- **Offset**: 偏移量，用于确定缓存行内的具体位置。

- **组 (Set) 和路 (Way) :**

- **组 (Set)** : 物理地址的 Set 段能表示的最大数目称为组。例 ， 果 Set 段的位数

是8，那么对应的CPU缓存的组数就是256 (2^8)。

- 路 (Way)：每组中支持的最大缓存行数目（最多的Tag数）称为路。例如，如果每组最多支持4个不同的Tag，那么该CPU缓存被称为4路组相联（4-Way Set Associative）。

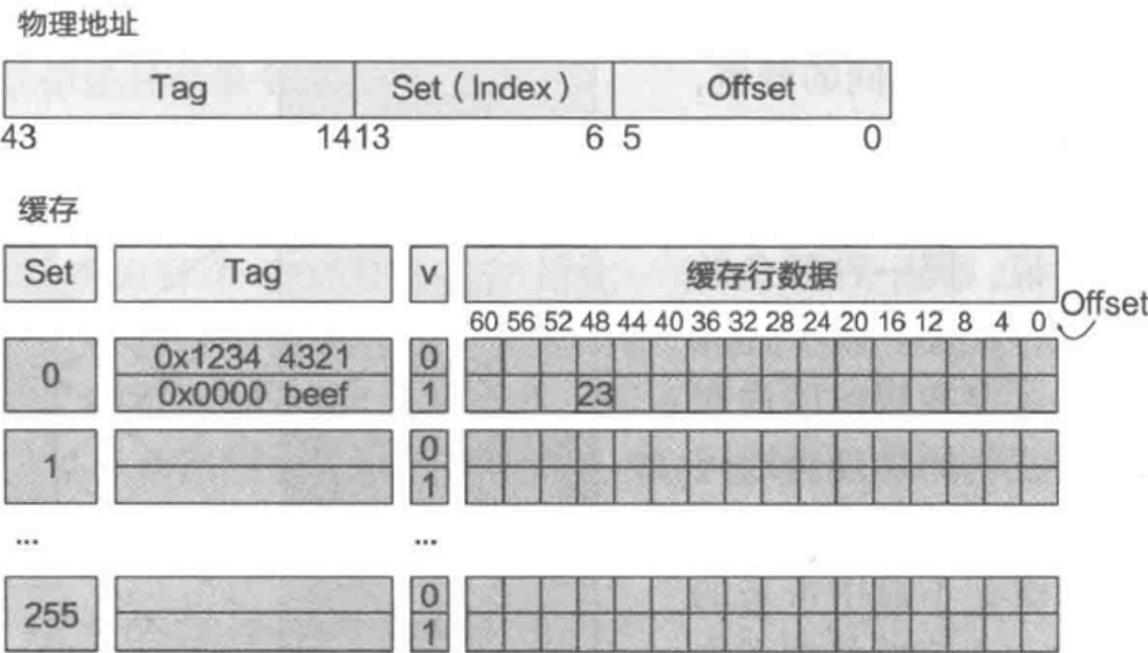
3、缓存寻址

以Cortex-A57 CPU (AArch64架构) 的L1数据缓存为例，介绍CPU缓存查找的一般过程。

- 缓存参数：

- 物理地址长度为44位。
- 缓存大小为32KB，缓存行大小为64字节。
- 256组，2路组相联缓存。

- 下面假设要读取以物理地址0x2FBBC030开始的4个字节的物理内存数据。



- Offset：每行缓存64字节，需要6位 ($2^6 = 64$)。

- Set：256组，需要8位 ($2^8 = 256$)。

- Tag：剩余的位数， $44 - 6 - 8 = 30$ 位。

所以，地址0x2FBBC030可以分解为：

- Tag: 0xBEEF
- Set: 0x0
- Offset: 0x30 (十进制为48)

从而 图所示，本例中取出的4字节字为23。

- 注意，如果在寻址过程中，虽然Set和Tag都匹配上了，但是Valid为0，那么该缓存行是无效的，需从物理内存搬运相应数据到CPU缓存中。

4、CPU缓存行何时写回物理内存？第一，存在专门的硬件指令负责写回某缓存行；第二，若某

组中的缓存行状态都为有效状态且被修改过，而CPU需要在该组中加载新的缓存行进行读写，此时CPU会首先选择该组中已有的某缓存行写回物理内存（空出一个缓存行）。

物理内存分配与CPU缓存

- CPU缓存比物理内存快得多，但容量有限。当物理内存中的数据被频繁访问时，如果能尽量存放在CPU缓存中，就能显著提高性能。然而，缓存容量有限，当缓存满了时，CPU会选择替换某些缓存行，这可能导致缓存冲突。
- **缓存着色技术：**将物理页标记为不同的颜色，确保分配给连续虚拟内存页的物理页不会引起缓存冲突。这样，当程序访问这些连续的虚拟内存页时，它们的数据都能在缓存中找到，避免了缓存不命中带来的性能损失。
- 举例说明，假设缓存可以容纳4个连续的物理页，操作系统会将第1到第4个物理页分别标记为4种不同的颜色，然后第5到第8个物理页再重复这4种颜色，以此类推。这样做的结果是，物理页的分配虽然变得复杂了，但如果能准确预测应用的内存访问模式，就能有效提升内存访问性能。FreeBSD和Solaris等操作系统已经采用了这种机制。

多核与内存分配

- 多核CPU中，每个核心可以同时运行不同的应用，这些应用在需要操作系统分配内存时，可能会导致多个核心同时调用内存分配接口。为了避免重复分配同一块内存，最初的方法是使用锁机制，确保只有一个核心可以进行内存分配，但这会导致其他核心等待，降低性能。
- 为了解决这个问题，操作系统开发人员提出了为每个CPU核心建立独立的内存分配器的方法。例如，如果有4个CPU核心，就初始化4个SLAB分配器，每个核心使用自己的分配器。这样，多个核心可以同时进行内存分配，不会互相干扰，从而充分发挥多核CPU的优势，提高系统的整体性能。

非一致内存访问（NUMA架构）

- **NUMA架构的背景：**多核和多处理器系统引入多个内存控制器，以解决单一内存控制器的性能瓶颈。
- **NUMA架构的组成：**每个处理器或处理器的一部分核心组成一个NUMA节点，节点内的核心可以快速访问本地内存，但访问远端内存时延较高。
- **NUMA架构的复杂性：**现代服务器可能有多个NUMA节点，节点之间的连接可能不是直接的，导致访问时延差异更加复杂。
- 在NUMA架构下，操作系统需要优化内存分配以减少远程访问的开销。现代操作系统通常采用两种策略：一是提供NUMA感知的内存分配接口，让应用显式指定内存分配位置；二是对于未显式使用这些接口的应用，操作系统会根据线程运行的节点，尽量将内存分配在本地节点，如果本地内存不足，再考虑邻近节点。

Linux提供了三种内存分配模式：绑定模式、优先模式和交错模式。绑定模式将内存分配绑定到

指定的NUMA节点，优先模式在分配失败时尝试从最近的节点分配内存，交错模式则以页为单位从多个节点交错分配内存。Linux还提供了libnuma库，封装了NUMA相关的系统调用，包括 `numa_alloc_onnode`、`numa_alloc_local` 和 `numa_alloc_interleaved` 接口，分别用于在指定节点、本地节点和交错模式下分配内存。此外，`numa_free` 接口用于释放通过上述接口分配的内存，防止内存泄漏。这些策略和工具对于理解NUMA架构下的内存管理非常有帮助，尤其是对于多核系统上的高性能计算应用。

Last change: 2025-02-14, commit: [9fa9752](#)

一些说明

本文章以Lab内容为基础进行相关拓展，主要涉及**物理内存分配器**，内核态与用户态在内存管理上的不同。本文不涉及伙伴系统，页表管理，缺页管理。

物理内存分配器

背景

内核中的物理内存由伙伴系统(buddy system)进行管理，它的分配粒度是以物理页帧(4KB)为单位的，但内核中有大量的数据结构只需要若干bytes的空间，倘若仍按页来分配，势必会造成大量的内存被浪费掉。slab分配器的出现就是为了解决内核中这些小块内存分配与管理的难题。

在内核的不断演进过程中，出现了三种物理内存分配器，slab，slob，slub。其中slab是最早的内存的分配器，由于有诸多的问题，后来被slob以及slub取代了。而slob在主要用于内存较小的嵌入式系统。Slub由于支持NUMA架构以及诸多的优点，逐渐的成为了当前内核中的主流内存分配器。我们已经在实验中了解了slab的内存分配功能，下面将介绍slab的另外两个功能以及slub。

SLAB 的第二使命：维护常用对象的缓存（简称对象缓存）

Slab 分配器的核心优势不仅在于高效地管理小内存块，还在于它能够**缓存常用对象**，使得对象在释放后能够被快速复用，而不需要每次重新初始化和分配。这个机制在内核中极为重要，因为许多数据结构需要频繁分配和释放，而初始化这些结构的开销可能与分配内存本身的开销相当，甚至更高。

在 Slab 分配器中，对象缓存（Object Caching）指的是

- 释放的对象不会立即归还给**伙伴系统（Buddy System）**，而是被**保留在 Slab 的内部缓存中**。
- 当同类型的对象再次需要分配时，**优先复用最近释放的对象**，而不是重新向伙伴系统申请内存并进行初始化。
- 由于释放的对象仍然保留在 Slab 缓存中，它们的物理地址通常不会改变，因此仍然可能**驻留在 CPU 的 L1/L2 缓存中**，从而提高访问速度。

这种机制在 Linux 内核中广泛应用于**需要频繁创建和销毁的内核数据结构**，：

- fs_struct（管理进程的文件系统信息）

- task_struct (进程描述符)
- inode (文件索引节点)

那么为什么需要对象缓存

(1) 减少初始化开销

某些数据结构的初始化开销**远远大于**分配它们的内存开销。例 ， fs_struct 结构用于存储进程的文件系统信息，当一个进程创建时，内核必须为其分配 fs_struct，并初始化其中的多个字段。这个初始化过程可能包括：

- 设置默认值
- 复制文件系统信息
- 分配多个子结构体

如果每次进程创建时都需要执行完整的初始化，系统开销将会非常大。因此，Slab 分配器允许**已经初始化的对象在释放后保留在缓存中**，下次分配时可以直接复用，大大减少初始化成本。

(2) 提高内存分配效率

传统的伙伴系统适用于管理**大块内存**，但对**小对象的频繁分配和释放**效率较低。Slab 分配器的对象缓存机制可以避免频繁调用伙伴系统，从而：

- **减少锁争用** (伙伴系统的操作通常需要全局锁 (在完成lab2的时候我们经常见到全局锁)，而 Slab 允许局部缓存)
- **降低碎片化** (伙伴系统可能因小对象频繁分配和释放导致碎片化)

(3) 提高 CPU 缓存命中率

当一个对象被释放后，它仍然可能留在 CPU 的缓存 (L1/L2 Cache) 中。如果系统很快又需要一个相同类型的对象，那么直接复用这个对象可以避免 CPU 重新加载内存，从而提高访问速度。例：

- 当一个 fs_struct 释放后，Slab 分配器不会立即释放其内存，而是保留在缓存中。
- 由于 fs_struct 仍然驻留在 CPU 缓存中，下次再创建进程时，内核可以直接复用该结构，减少访问 DRAM 的延迟。

Slab 分配器对象缓存的实现

(1) 创建 Slab 缓存

在 Linux 内核中，每种需要频繁分配的对象类型，都会有一个专门的 **Slab 缓存 (Slab Cache)**，用于存储该类型的对象。例 ， Linux 通过 kmem_cache_create() 创建 fs_struct 缓存：

```
fs_cachep = kmem_cache_create("fs_cache", sizeof(struct fs_struct), 0,  
SLAB_HWCACHE_ALIGN, NULL);
```

fs_cachep 是 fs_struct 类型的 Slab 缓存指针，该缓存会管理多个 fs_struct 结构，并在释放后保持缓存状态。

(2) 分配对象

当需要分配 fs_struct 时，内核会调用：

```
struct fs_struct *fs = kmem_cache_alloc(fs_cachep, GFP_KERNEL);
```

如果 fs_cachep 里有空闲对象，直接返回一个最近释放的对象，如果缓存已满，则分配一个新的 Slab 以此来避免使用伙伴系统，提高分配效率。

(3) 释放对象

当 fs_struct 不再需要时，调用：

```
kmem_cache_free(fs_cachep, fs);
```

fs 并不会被立即释放，而是放入 fs_cachep 的**空闲对象列表。**这样，当下次 kmem_cache_alloc() 需要一个 fs_struct 时，可以直接复用，避免重新分配和初始化。

注：这里“直接复用”的表述不太准确，两次所使用的结构体参数可能不同，而 slab 也有相应的解决方案，这里为了不过多赘述使用“直接复用”的说法。有关问题可以自行搜索。

SLAB 的第三使命：提高CPU硬件缓存的利用率

CPU 缓存与 SLAB 之间的关系

在现代计算机系统中，CPU 访问缓存的速度远远快于访问主存（RAM）。在 CPU 访问数据时，数据首先会被加载到**L1/L2/L3 缓存**中。CPU 读取缓存的速度要比访问主存快 10~100 倍，因此，如果 SLAB 分配器能够优化对象的存储方式，使其尽可能多地驻留在缓存中，就能显著提升系统性能。SLAB 分配器的设计目标之一就是**让分配的对象尽可能驻留在 CPU 缓存中，而不需要频繁访问主存。**

优化策略

SLAB 分配器通过以下几种方式优化 CPU 缓存的利用率：

- 分配对齐 (Cache Alignment)

在 SLAB 分配对象时，内核会尝试让对象按 CPU 缓存行 (Cache Line) 大小对齐，从而减少缓存行拆分 (Cache Line Splitting) 和 伪共享 (False Sharing)。

- 减少缓存污染 (Cache Pollution)

如果内核频繁分配和释放对象，而这些对象的地址在物理内存中分散，那么 CPU 缓存可能会被无用的数据填满，导致缓存命中率下降。这种情况称为缓存污染 (Cache Pollution)。

SLAB 如何减少缓存污染？

1. 优先复用最近释放的对象，而不是重新分配新内存。
2. 局部性优化 (Locality Optimization)，让同一类对象尽量存储在相邻的缓存行中，提高访问连续性。

- 提高缓存命中率 (Cache Hit Rate)

SLAB 分配器通过空间局部性 (Spatial Locality) 和 时间局部性 (Temporal Locality) 机制，提高 CPU 缓存的命中率。

空间局部性优化：SLAB 让相同类型的对象存储在连续的内存块中，使得 CPU 在访问一个对象时，能够预加载相邻的对象，提高缓存命中率。

例如：

```
struct task_struct *t1 = kmalloc(task_cache, GFP_KERNEL);
struct task_struct *t2 = kmalloc(task_cache, GFP_KERNEL);
```

如果 t1 和 t2 被分配到连续的物理地址，那么当 CPU 读取 t1 时，很可能会预加载 t2，提高缓存利用率。

时间局部性优化：SLAB 让最近使用的对象尽可能留在缓存中，如果短时间内需要再次使用相同的对象，就可以避免重新加载主存。

例如，当进程频繁创建/销毁 fs_struct，SLAB 会优先复用最近释放的 fs_struct，因为它仍然可能在 L1/L2 缓存中。

- NUMA 亲和性优化 (NUMA Affinity)

SLUB 简介

SLUB 分配器是 Linux 内核中的一种 优化版 SLAB 分配器，它的目标是 提高内存分配的效率，减少锁争用，并降低内存碎片化。相比 SLAB，SLUB 设计更简单、性能更好，因此是 Linux 内

核的默认分配器。下面讲述两者不同点。

管理结构

SLAB: 使用单独的 slab 结构来管理每个 slab 及其中的对象。每个 kmem_cache 对象有一个对应的 slab 列表，用来管理具体的内存分配。

```
struct kmem_cache {
    const char *name;
    size_t obj_size;
    unsigned int obj_per_slab;
    struct slab *slab_list; // 存储 slab 列表
};

struct slab {
    struct page *pages; // 对应的页结构
    struct kmem_cache *cache;
    void *freelist; // 空闲对象链表
};
```

SLUB: SLUB 去除了 slab 结构，直接使用 struct page 来管理 slab 中的对象，简化了内存管理结构。每个 CPU 都有自己的 kmem_cache_cpu，管理空闲对象列表。

```
struct kmem_cache {
    const char *name;
    size_t obj_size;
    struct page *page_list; // 使用 page 结构代替 slab
};

struct kmem_cache_cpu {
    void *freelist; // 每个 CPU 都有自己的空闲对象链表
};

struct page {
    struct kmem_cache *cache;
    void *objects; // 对象的内存区域
};
```

锁机制

SLAB: 由于多个 CPU 可能会同时访问同一个 slab，所以需要通过加锁来同步访问。每个 kmem_cache 都有一个全局的锁来管理 slab。

```
struct kmem_cache {
    spinlock_t lock; // 用于保护 kmem_cache 的访问
};
```

SLUB: SLUB 采用每个 CPU 独立管理空闲对象列表 (freelist) , 因此在大多数情况下, 不需要加锁, 从而避免了锁争用, 提高了并发性能。

```
struct kmem_cache_cpu {  
    void *freelist; // 每个 CPU 独立管理, 不需要加锁  
};
```

SLUB 基于每个 CPU 有独立的 freelist, 这避免了对共享资源的频繁锁操作, 因此在多核系统中, SLUB 具有更好的性能, 尤其是在高并发场景下。

内存碎片

SLAB: 由于每个 slab 内存块的大小是固定的, 可能会导致对象大小与 slab 大小不匹配, 产生内部碎片。此外, slab 管理结构复杂, 容易产生外部碎片。

SLUB: SLUB 通过动态调整 slab 的大小, 更好地利用内存, 减少碎片。此外, 由于 SLUB 直接通过 page 来管理内存, 内存的布局更加灵活, 减少了内存碎片的发生。

NUMA 亲和性

SLAB: SLAB 在 NUMA 系统上性能较差, 因为它在内存分配时不会考虑 CPU 的亲和性, 可能会导致远程 NUMA 节点的内存访问延迟。

SLUB: SLUB 提供了更好的 NUMA 亲和性, 它会尝试将内存分配限制在当前 CPU 所在的 NUMA 节点内, 从而减少跨节点的内存访问, 提高性能。

SLUB 代码:

```
struct kmem_cache_cpu {  
    void *freelist; // 每个 CPU 独立管理, 支持 NUMA 亲和性  
};  
  
struct page {  
    unsigned long node_id; // 页面所属的 NUMA 节点  
};
```

空闲对象的回收与分配

SLAB: SLAB 使用 slab 结构来回收和分配内存。每个 slab 对象包含多个内存对象, 空闲对象被存放在 freelist 中, 当需要新的对象时, 从 freelist 中分配。

SLAB 代码:

```

void *slab_alloc(struct kmem_cache *cache) {
    struct slab *slab;
    if (slab->freelist) {
        return slab->freelist; // 从 freelist 中分配
    }
    // 果 freelist 为空, 申请新的 slab
    return slab_get_new(cache);
}

```

SLUB: SLUB 在分配对象时, 首先检查当前 CPU 的 freelist, 果没有空闲对象, 则从 page 中分配新的对象。当对象被释放时, 它会被返回到 freelist。

SLUB 代码:

```

void *slub_alloc(struct kmem_cache *cache) {
    struct kmem_cache_cpu *cpu_cache = &per_cpu(cache->cpu_cache,
smp_processor_id());
    if (cpu_cache->freelist) {
        return cpu_cache->freelist; // 从当前 CPU 的 freelist 中分配
    }
    // 果 freelist 为空, 申请新的 page
    return slub_get_new(cache);
}

```

由于 SLUB 支持每个 CPU 独立管理空闲对象, 避免了共享数据结构的锁竞争, 在高并发和高负载的环境下, 分配和回收对象的效率更高。

参考论文:

- https://github.com/0voice/computer_expert_paper/blob/main/%E6%8E%A5%E8%BF%91%E5%8E%9F%E5%A7%8B%E7%9A%84LinuxOS/%E3%80%8ASlab%20allocators%20in%20the%20Linux%20KernelSLAB%2C%20SLOB%2C%20SLUB%E3%80%8B.pdf
- https://github.com/0voice/computer_expert_paper/blob/main/%E6%8E%A5%E8%BF%91%E5%8E%9F%E5%A7%8B%E7%9A%84LinuxOS/%E3%80%8AThe%20Slab%20AllocatorAn%20Object-Caching%20Kernel%20Memory%20Allocator%E3%80%8B.pdf

用户态 VS 内核态

在内存管理方面, **内核态**和**用户态**有很多本质的区别, 这些区别源于两者在设计目标、内存访问权限、以及管理方式等方面的差异。下面我将从几个主要角度对比并说明。

内存管理的目标与设计

内核态

内核的主要目的是提供操作系统核心功能，进程调度、硬件驱动、文件系统等。在内存管理上，内核态更关注 **高效的内存访问** 和 **内存保护**，确保内存的安全性和稳定性。内核通常要处理大量的系统资源管理，因此内存分配机制在内核态上需要具备较强的 **简单性** 和 **高效性**，且往往采用 **固定的粒度**（页或 slab 级别）来分配内存，以减少碎片问题。

内核内存管理的关键：

- **内核通过内存池、缓存优化时间局部性**

内核内存池和缓存的优化机制可以大大提升内存访问的效率。通过 **SLAB 分配器**，内核可以创建缓存池来存储常用的内存对象，以避免频繁的分配和回收。内存池还可以通过维护一些常用对象的缓存，提高内存的 **时间局部性**，从而减少内存访问的延迟。例如，SLAB 分配器通过将内存块组织成一个个对象的缓存池，可以有效地减少碎片，提高分配和回收的效率。

```
struct kmem_cache *create_cache(const char *name, size_t size)
{
    struct kmem_cache *cache;
    cache = kmem_cache_create(name, size, 0, SLAB_PANIC, NULL);
    return cache;
}
```

- **页粒度管理内存，关注内存的局部性**

在内核中，内存的管理通常是基于 **页**（Page）来进行的，操作系统通过 **分页机制** 来将内存划分为一个个固定大小的页（通常为4KB），从而进行内存的管理。分页机制不仅可以提高内存分配的效率，还能实现 **内存保护**，防止不同进程之间的内存干扰。

内核关注 **空间局部性**，即在同一时间，程序对内存的访问通常会集中在一些相邻的内存区域，因此内核会尽量优化内存的页分配策略。例如，通过 **伙伴系统**（Buddy System）或 **SLAB 分配器** 来管理和回收内存，减少内存碎片并提高内存分配的速度。

```
void *kmalloc(size_t size, gfp_t flags)
{
    struct page *page;
    size_t order;

    order = get_order(size); // 获取内存的页数
    page = alloc_pages(flags, order); // 分配指定页数的内存
    if (!page)
        return NULL;

    return page_address(page); // 返回实际的内存地址
}
```

在这个代码中，`kmalloc` 函数通过 `alloc_pages` 按页分配内存，并通过 `page_address` 获得实际的内存地址。

- 内存的分配与回收由内核控制，操作相对简单、直接

在内核中，内存的分配与回收是由内核来管理的。为了减少内存分配时的复杂度和开销，内核采用了简化的分配策略，比如 通过 `kmalloc` 进行内存分配，内存回收则使用 `kfree` 函数。在内核内存管理中，回收内存通常不需要依赖于垃圾回收机制，而是通过明确的函数调用进行。

内核空间的内存分配采用的策略是：通过 **伙伴系统**（Buddy System）进行物理内存块的管理，内核直接控制内存的分配与回收，因此操作简单、直接。

用户态

用户态的内存管理更注重 **灵活性** 和 **适应性**。用户进程的内存管理需要考虑到不同应用程序的内存访问模式、生命周期管理、缓存机制等。由于 **用户空间** 的内存不受内核直接管理，用户程序通常通过 **动态分配** 和 **缓存策略** 来优化内存的使用效率，最大化性能。

用户内存管理的关键：

- 用户内存管理更加灵活，使用堆进行内存分配

在用户空间，内存管理通常更加 **灵活**，最常见的方式就是通过 **堆**（Heap）来分配内存。堆内存的分配非常灵活，可以在程序运行时动态地申请和释放内存。与内核态的内存分配不同，用户空间通过标准库函数（`malloc`）来管理堆上的内存，并且可以在程序运行过程中对内存进行管理。`malloc` 提供了灵活的内存管理策略，例如 通过 **内存池** 和 **缓存机制** 来避免频繁的分配与释放操作，优化程序的性能。

相关代码（`malloc` 内存池）：

```
void *malloc(size_t size)
{
    void *ptr;
    ptr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE |
    MAP_ANONYMOUS, -1, 0); // 使用 mmap 进行内存分配
    return ptr;
}
```

这里，malloc 通过 mmap 系统调用分配内存，mmap 可以提供更灵活的内存映射方式，同时允许分配大块内存。

- 使用 malloc 等函数进行内存分配，通常会处理分配的生命周期和缓存等问题。

在用户空间，内存的生命周期管理通常由开发者或库函数来处理。malloc 和 free 等函数会自动管理内存的生命周期，开发者可以通过使用缓存机制来提升程序的性能(CSAPP 中的cache lab即考验此点)。

内存分配的效率与策略

内核态

由于内核空间的内存分配需要考虑到实时性和简洁性，内核一般采用 **固定大小的内存块** 来管理内存，例 使用 kmalloc，并采用一些内存分配算法（伙伴系统、SLAB 分配器）来降低分配开销。内核态内存管理的效率通常较高，但灵活性相对较低。

kmalloc 是内核空间的内存分配函数。其主要特点是提供快速分配小块内存，且内存对齐要求严格。在分配时会检查是否能够满足物理内存的连续性要求，若不满足会触发页面分配。通常情况下，kmalloc 会尝试高效地为内核分配连续的内存区域。

用户态

用户空间的内存管理更侧重于 **灵活性**，通过 malloc 这样的分配函数，用户可以按需分配内存。用户内存分配的性能往往与分配器的设计、使用模式等因素密切相关，malloc 通过内部的缓存池、合并空闲内存块等技术来提升分配效率。

malloc 是用户态的内存分配函数，背后通常由操作系统的动态链接库（**glibc**）来实现，采用的是 **堆管理** 和 **分配策略**，使用 **双向链表** 管理空闲内存块 (CSAPP 中的 malloc lab 即采用不同管理策略)。

内存释放与回收

内核态

内核通过 kfree 函数来释放 kmalloc 分配的内存。内核会更严格地控制内存的回收，避免内存泄漏，并且使用相对简单的回收机制来确保内存的及时回收。

用户态

用户态通过 free 来释放 malloc 分配的内存。内核空间和用户空间的回收机制不同，内核需要手动回收，而用户态一般依赖于 **垃圾回收** 或 **内存池管理** 来自动回收内存。

Last change: 2025-02-14, commit: [9fa9752](#)

Last change: 2025-02-14, commit: [23a4605](#)

源码解析

这部分内容是对进程——线程——异常部分的chcore源代码的详细解析，包括进程/线程管理、异常管理与系统调用三个部分。

Important

完成 Lab3 后，再阅读这部分内容。

Last change: 2025-02-14, commit: [9fa9752](#)

本部分内容主要讲解Chcore中进程/线程管理的部分。从能力组机制的作用，到其具体的实现，再到其具体落实到线程管理的过程。

能力组机制

能力组的概念

能力

我们知道，在Chcore中，系统资源的管理分配是通过将一切系统资源都视作Object（对象）来实现的。而能力，则是一种访问控制机制，可以理解为“带有权限的引用”。它包含两个基本要素：

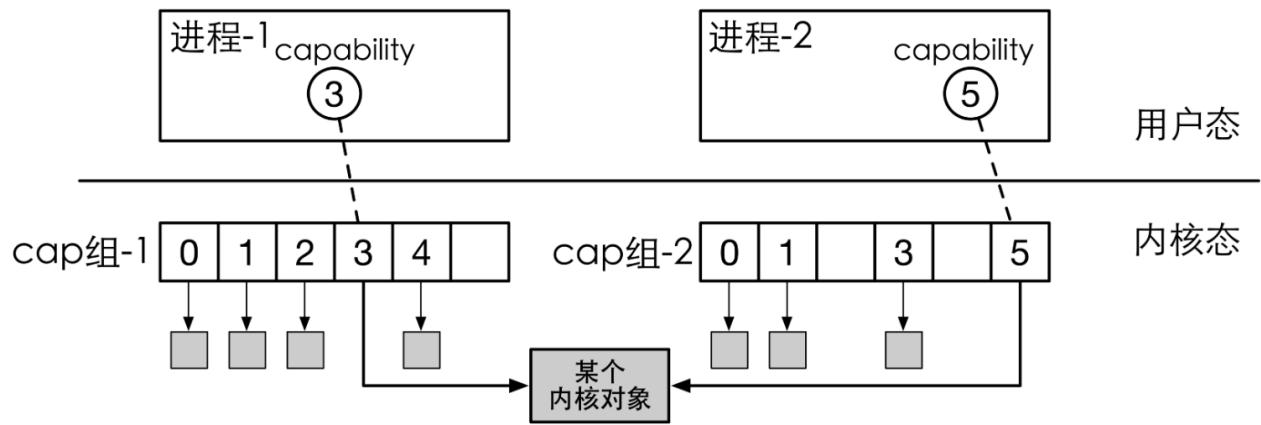
- 对象的引用（指向要访问的资源）
- 对该对象的权限（允许进行什么操作）

由此我们便可以理解文档中提到的“能力组本身只是一个包含指向object的指针的数组”

能力组

文档告诉我们：

1. 所有进程/线程都有一个独立的能力组，拥有一个全局唯一ID (**Badge**)
 2. 所有对象（包括进程或能力组本身）都属于一个或多个能力组当中，也就是说子进程与线程将属于父进程的能力组当中，在某个能力组的对象拥有一个能力组内的能力ID(**cap**)。
 3. 对象可以共享，即单个对象可以在多个能力组中共存，同时在不同cap_group中可以有不同的**cap**
 4. 对所有对象的取用和返还都使用引用计数进行追踪。当引用计数为0后，当内核垃圾回收器唤醒后，会自动回收。
 5. 能力组内的能力具有权限，表明该能力是否能被共享(CAP_RIGHT_COPY)以及是否能被删除(CAP_RIGHT_REVOCATION)
-



Linux中的应用

此事在Linux之中亦有记载.....

ref: <https://www.cnblogs.com/sparkdev/p/11417781.html>

wiki上的简单名词介绍 https://en.wikipedia.org/wiki/Capability-based_operating_system

tl;dr 能力组是为安全而生的进程“部分root”的机制，例 你可以在用户态运行/bin/ping

数据结构实现

说到这里，我们可以看看Chcore代码中对几个相关数据结构的实现：

```

struct object {
    u64 type;
    u64 size;
    struct list_head copies_head; // 指向同一对象的slots链表
    // ...其他字段
};

struct object_slot {
    int slot_id; // 槽位ID
    struct cap_group *cap_group; // 所属的能力组，反向引用
    struct object *object; // 指向实际对象
    struct list_head copies; // 指向同一对象的复制槽位链表
    cap_right_t rights; // 对对象的访问权限
}

struct slot_table {
    unsigned int slots_size; // 槽位表大小
    struct object_slot **slots; // 槽位数组
    unsigned long *full_slots_bmp; // 完全填满的槽位位图
    unsigned long *slots_bmp; // 槽位使用位图
    struct rwlock table_guard; // 读写锁保护
}

struct cap_group {
    struct slot_table slot_table; // 槽位表
    struct lock threads_lock; // 线程列表锁
    struct list_head thread_list; // 线程列表
    int thread_cnt; // 线程计数
    badge_t badge; // 全局唯一标识
    int pid; // 进程ID
    // ... 其他字段
}

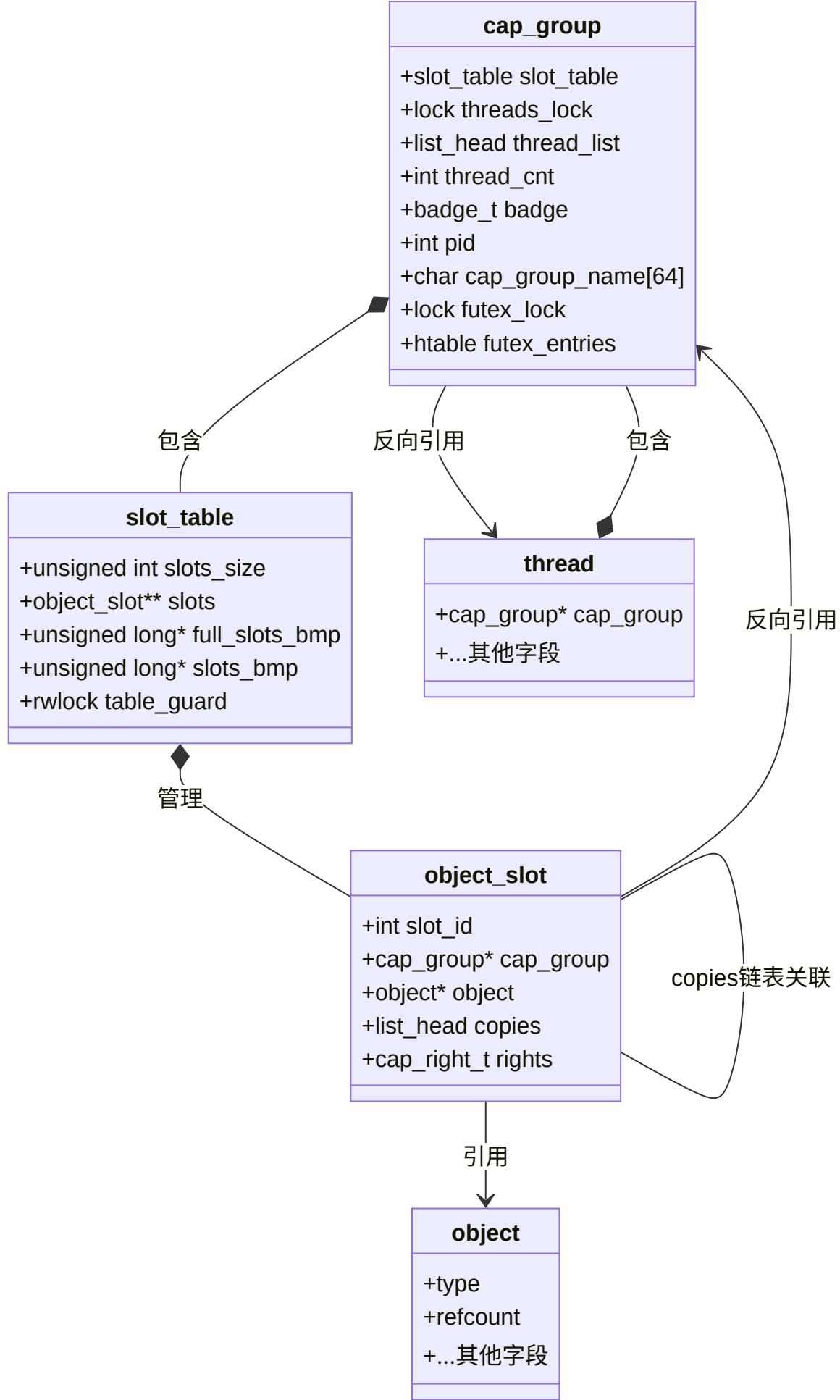
```

结合中文注释一步一步看下来就能理解文档的意思了：

- 能力组本身，即 `cap_group`，通过 `slot_table` 引用一系列的 `object_slot` 进而得到对对象的引用，也就是“指向object指针的数组”
- 能力本身，即 `object_slot`，指的即是包含了权限的内核对象的引用，文件描述符，网络连接等，通过上述指针关系，一步步从属于能力组的安排
- 那么object在多个能力组之间的共享又是何实现的？便在于 `object` 中对能力组的反向引用，从而得到一个具体的对象所属于的不同能力组
- 同样的，`cap_group` 里的 `thread_list` 同样是对线程的又一个反向引用，即实现了不同线程之间共享能力组的操作。举例：假设父进程p持有一系列内核对象的资源，例文件描述符等，那p的线程以及子进程皆应该有文件描述符等资源

概念关系图

由此我们可以画出这些概念在Chcore代码中的关系图，有点绕，但是确实 此



object虚函数表

最后，由于object实际上只是一切内存对象的基类，所以在实际析构的时候，还需要根据对象的类型选择对应的析构函数。因此在Chcore中，还可以看见用C语言写的“虚函数表”，从而用结构体实现了OOP的部分特性：

```
const obj_deinit_func obj_deinit_tbl[TYPE_NR] = {
    [0 ... TYPE_NR - 1] = NULL,
    [TYPE_CAP_GROUP] = cap_group_deinit,
    [TYPE_THREAD] = thread_deinit,
    [TYPE_CONNECTION] = connection_deinit,
    [TYPE_NOTIFICATION] = notification_deinit,
    [TYPE_IRQ] = irq_deinit,
    [TYPE_PMO] = pmo_deinit,
    [TYPE_VMSPACE] = vmspace_deinit,
#define CHCORE_OPENTRUSTEE
    [TYPE_CHANNEL] = channel_deinit,
    [TYPE_MSG_HDL] = msg_hdl_deinit,
#endif /* CHCORE_OPENTRUSTEE */
    [TYPE_PTRACE] = ptrace_deinit
};
```

针对这样的数据结构, capability.c之中小心地实现了 `cap_group` 的复制, 释放, 传递等函数, 之中由于基于引用计数的所有权的复杂性+并发+为了性能降低锁的粒度, 代码并不是非常可读, 感兴趣的读者可以自行深入研究

能力组创建

现在我们再来看Chcore是如何具体创建能力组的。在main函数中，系统将首先调用 `create_root_cap_group` 创建第一个 `cap_group` 进程，并在 `root_cap_group` 中创建第一个线程

此外，用户程序也可以通过 `sys_create_cap_group` 系统调用创建一个全新的 `cap_group`

`create_root_cap_group`

先看源码：

```

/* This is for creating the first (init) user process. */
struct cap_group *create_root_cap_group(char *name, size_t name_len)
{
    struct cap_group *cap_group;
    struct vmspace *vmspace;
    cap_t slot_id;

    cap_group = obj_alloc(TYPE_CAP_GROUP, sizeof(*cap_group));
    if (!cap_group) {
        kwarn("failed alloc cap_group in %s\n", __func__);
        return NULL;
    }
    cap_group_init(cap_group,
                   BASE_OBJECT_NUM,
                   /* Fixed badge */ ROOT_CAP_GROUP_BADGE);

    slot_id = cap_alloc(cap_group, cap_group);
    BUG_ON(slot_id != CAP_GROUP_OBJ_ID);

    vmspace = obj_alloc(TYPE_VMSPACE, sizeof(*vmspace));
    if (!vmspace) {
        kwarn("failed alloc vmspace in %s\n", __func__);
        return NULL;
    }

    /* fixed PCID 1 for root process, PCID 0 is not used. */
    vmspace_init(vmspace, ROOT_PROCESS_PCID);

    slot_id = cap_alloc(cap_group, vmspace);
    BUG_ON(slot_id != VMSPACE_OBJ_ID);

    /* Set the cap_group_name (process_name) for easing debugging */
    memset(cap_group->cap_group_name, 0, MAX_GROUP_NAME_LEN + 1);
    if (name_len > MAX_GROUP_NAME_LEN)
        name_len = MAX_GROUP_NAME_LEN;
    memcpy(cap_group->cap_group_name, name, name_len);

    root_cap_group = cap_group;
    return cap_group;
}

```

从上往下看，函数逻辑依次是：

- 创建 `cap_group` 对象，并分配 slot 槽位
- 创建 `vmspace` 对象，并分配 slot 槽位
- 为 `cap_group` 命名，这里包含名字长度检查等操作
- 设置根能力组，即 `root_cap_group` 为此能力组，结束函数

不难注意到一个有趣的事情：那就是这里的能力组本身同样是内核对象，因此指向自己的指针也被放在了 `slot_table` 里。而这则是由于root能力组的特殊性决定的——它没有父进程，是系统中的第一个能力组，因此需要建立初始的能力权限管理机制，即自身作为自身的权力来源，进而作为整个系统的权限起点

而普通的能力组创建，则需严格遵守能力组权限的要求，权限来自父进程

sys_root_cap_group

这里的 `sys_create_cap_group`，则作为一个syscall 的handler呈现

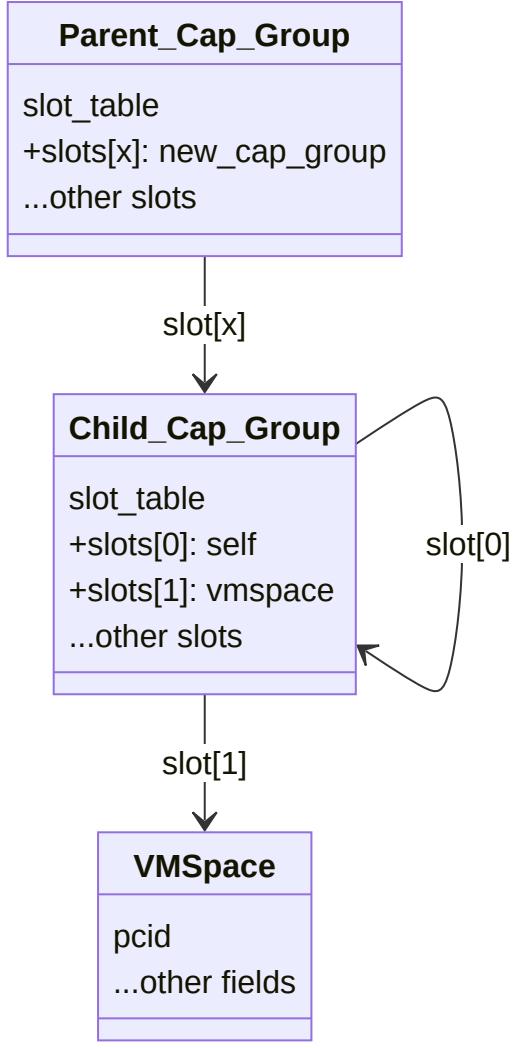
```
const void *syscall_table[NR_SYSCALL] = {  
    // ...  
    [CHCORE_SYS_create_cap_group] = sys_create_cap_group,
```

传入的 `*unsigned* *long* *cap_group_args_p` 就是用户态进程传递过来的在其vm下的，指向 `sys_create_cap_group` 这个 syscall 的参数的指针。在函数具体执行中会将其转换为需要的参数

明白了这个之后，它的逻辑就比较自然了

- 用户态的指针不可信，校验其地址空间是否伸进kernel，是否有创建权限
- copy 用户态的数据到内核（这里的拷贝是没有传递用户态的页表的，在现代OS之中，内核态页表包含用户态的页表项，所以 `copy_from_user` 只是一个简单的memcpy而已）
- 类似上面的逻辑，alloc新的cap_group对象，cap_init初始化，根据传入的参数给几个属性赋值
- 新cap_group应该带上默认的cap，即cap_group和vmspace

内存布局 下：



创建第一个进程/线程

在Chcore的main函数中，内核会通过调用 `create_root_thread` 函数来创建第一个进程与线程，这部分内容我们就来研究该函数，同时简单讲解一下对初始化线程上下文的函数

`init_thread_ctx`

在Lab文档里已经写了关于 `create_root_thread` 函数的流程图，所以这里着重研究源码的细节相关

`create_root_thread`

这部分源码很长

```
/*
 * The root_thread is actually a first user thread
 * which has no difference with other user threads
 */
void create_root_thread(void)
{
    struct cap_group *root_cap_group;
    cap_t thread_cap;
    struct thread *root_thread;
    char data[8];
    int ret;
    cap_t stack_pmo_cap;
    struct thread *thread;
    struct pmobject *stack_pmo;
    struct vmspace *init_vmspace;
    vaddr_t stack;
    vaddr_t kva;
    struct process_metadata meta;

    /*
     * Read from binary.
     * The msg and the binary of of the init process(procmgr) are linked
     * behind the kernel image via the incbin instruction.
     * The binary_procmgr_bin_start points to the first piece of info:
     * the entry point of the init process, followed by eight bytes of
     data
     * that stores the mem_size of the binary.
     */

    memcpy(data,
           ((void *)((unsigned long)&binary_procmgr_bin_start
                     + ROOT_ENTRY_OFF)),
           sizeof(data));
    meta.entry = (unsigned long)be64_to_cpu(*(u64 *)data);

    memcpy(data,
           ((void *)((unsigned long)&binary_procmgr_bin_start
                     + ROOT_FLAGS_OFF)),
           sizeof(data));
    meta.flags = (unsigned long)be64_to_cpu(*(u64 *)data);

    memcpy(data,
           ((void *)((unsigned long)&binary_procmgr_bin_start
                     + ROOT_PHEN_SIZE_OFF)),
           sizeof(data));
    meta.phentsize = (unsigned long)be64_to_cpu(*(u64 *)data);

    memcpy(data,
           ((void *)((unsigned long)&binary_procmgr_bin_start
                     + ROOT_PHNUM_OFF)),
           sizeof(data));
    meta.phnum = (unsigned long)be64_to_cpu(*(u64 *)data);

    memcpy(data,
           ((void *)((unsigned long)&binary_procmgr_bin_start
                     + ROOT_PHDR_ADDR_OFF)),
           sizeof(data));
    meta.phdr_addr = (unsigned long)be64_to_cpu(*(u64 *)data);
}
```

```
        sizeof(data));
meta.phdr_addr = (unsigned long)be64_to_cpu(*(u64 *)data);

root_cap_group = create_root_cap_group(ROOT_NAME, strlen(ROOT_NAME));

BUG_ON(!root_cap_group);

init_vmspace = obj_get(root_cap_group, VMSPACE_OBJ_ID, TYPE_VMSPACE);

BUG_ON(!init_vmspace);

/* Allocate and setup a user stack for the init thread */
stack_pmo_cap = create_pmo(ROOT_THREAD_STACK_SIZE,
                           PMO_ANONYM,
                           root_cap_group,
                           0,
                           &stack_pmo,
                           PMO_ALL_RIGHTS);
BUG_ON(stack_pmo_cap < 0);

ret = vmspace_map_range(init_vmspace,
                        ROOT_THREAD_STACK_BASE,
                        ROOT_THREAD_STACK_SIZE,
                        VMR_READ | VMR_WRITE,
                        stack_pmo);
BUG_ON(ret != 0);

/* Allocate the init thread */
thread = obj_alloc(TYPE_THREAD, sizeof(*thread));
BUG_ON(thread == NULL);

for (int i = 0; i < meta.phnum; i++) {
    unsigned int flags;
    unsigned long offset, vaddr, filesz, memsz;

    memcpy(data,
           (void *)((unsigned long)&binary_procmgr_bin_start
                     + ROOT_PHDR_OFF + i * ROOT_PHENT_SIZE
                     + PHDR_FLAGS_OFF),
           sizeof(data));
    flags = (unsigned int)le32_to_cpu(*(u32 *)data);

    memcpy(data,
           (void *)((unsigned long)&binary_procmgr_bin_start
                     + ROOT_PHDR_OFF + i * ROOT_PHENT_SIZE
                     + PHDR_OFFSET_OFF),
           sizeof(data));
    offset = (unsigned long)le64_to_cpu(*(u64 *)data);

    memcpy(data,
           (void *)((unsigned long)&binary_procmgr_bin_start
                     + ROOT_PHDR_OFF + i * ROOT_PHENT_SIZE
                     + PHDR_VADDR_OFF),
           sizeof(data));
    vaddr = (unsigned long)le64_to_cpu(*(u64 *)data);

    memcpy(data,
```

```
(void *)((unsigned long)&binary_procmgr_bin_start
+ ROOT_PHDR_OFF + i * ROOT_PHENT_SIZE
+ PHDR_FILESZ_OFF),
sizeof(data));
filesz = (unsigned long)le64_to_cpu(*(u64 *)data);

memcpy(data,
(void *)((unsigned long)&binary_procmgr_bin_start
+ ROOT_PHDR_OFF + i * ROOT_PHENT_SIZE
+ PHDR_MEMSZ_OFF),
sizeof(data));
memsz = (unsigned long)le64_to_cpu(*(u64 *)data);

struct pmo *segment_pmo;
size_t pmo_size = ROUND_UP(memsz, PAGE_SIZE);

vaddr_t segment_content_kvaddr =
((unsigned long)&binary_procmgr_bin_start) + offset;

/* According to the linker.ld in procmgr, no exact bss
exists
 * in elf.*/
BUG_ON(filesz != memsz);
// No additional memory for .bss, so we can directly reuse
// content in kernel image as their physical pages
ret = create_pmo(PAGE_SIZE,
                  PMO_DATA,
                  root_cap_group,
                  0,
                  &segment_pmo,
                  PMO_ALL_RIGHTS);
BUG_ON(ret < 0);
kfree((void *)phys_to_virt(segment_pmo->start));
#endif CHCORE_ARCH_X86_64
// See comments of embedded_bin_virt_to_phys
segment_pmo->start =
    embedded_bin_virt_to_phys(segment_content_kvaddr);
segment_pmo->size = pmo_size;
#else
segment_pmo->start = virt_to_phys(segment_content_kvaddr);
segment_pmo->size = pmo_size;
#endif
unsigned vmr_flags = 0;
if (flags & PHDR_FLAGS_R)
    vmr_flags |= VMR_READ;
if (flags & PHDR_FLAGS_W)
    vmr_flags |= VMR_WRITE;
if (flags & PHDR_FLAGS_X)
    vmr_flags |= VMR_EXEC;

ret = vmspace_map_range(init_vmspace,
                        vaddr,
                        segment_pmo->size,
                        vmr_flags,
                        segment_pmo);
BUG_ON(ret < 0);
}
```

```

obj_put(init_vmspace);

stack = ROOT_THREAD_STACK_BASE + ROOT_THREAD_STACK_SIZE;

/* Allocate a physical page for the main stack for prepare_env */
kva = (vaddr_t)get_pages(0);
BUG_ON(kva == 0);
commit_page_to_pmo(stack_pmo,
                     ROOT_THREAD_STACK_SIZE / PAGE_SIZE - 1,
                     virt_to_phys((void *)kva));

prepare_env((char *)kva, stack, ROOT_NAME, &meta);
stack -= ENV_SIZE_ON_STACK;

ret = thread_init(thread,
                  root_cap_group,
                  stack,
                  meta.entry,
                  ROOT_THREAD_PRIO,
                  TYPE_USER,
                  smp_get_cpu_id());
BUG_ON(ret != 0);

/* Add the thread into the thread_list of the cap_group */
lock(&root_cap_group->threads_lock);
list_add(&thread->node, &root_cap_group->thread_list);
root_cap_group->thread_cnt += 1;
unlock(&root_cap_group->threads_lock);

/* Allocate the cap for the init thread */
thread_cap = cap_alloc(root_cap_group, thread);
BUG_ON(thread_cap < 0);
thread->cap = thread_cap;

/* L1 icache & dcache have no coherence on aarch64 */
flush_idcache();

root_thread = obj_get(root_cap_group, thread_cap, TYPE_THREAD);
/* Enqueue: put init thread into the ready queue */
BUG_ON(sched_enqueue(root_thread));
obj_put(root_thread);
}

```

总体而言，可以分为 下几个的步骤：

读取元数据

从kernel镜像后链接的二进制文件中读取初始化进程（procmgr）的入口点、标志、程序头表项大小、程序头表项数量和程序头表地址。也就是函数开头的一系列 memcpy 函数操作

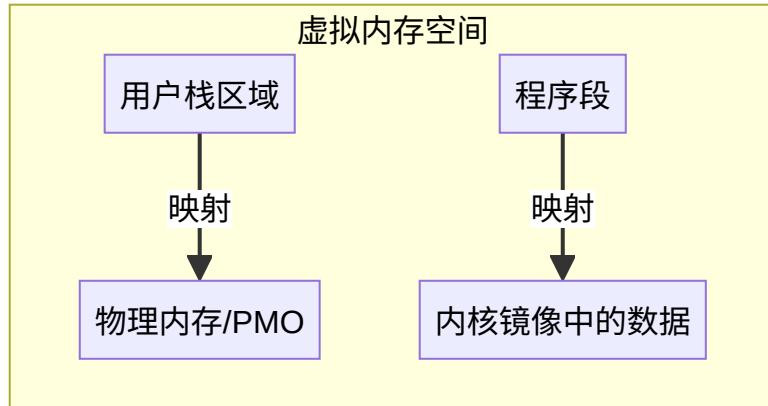
内存布局准备

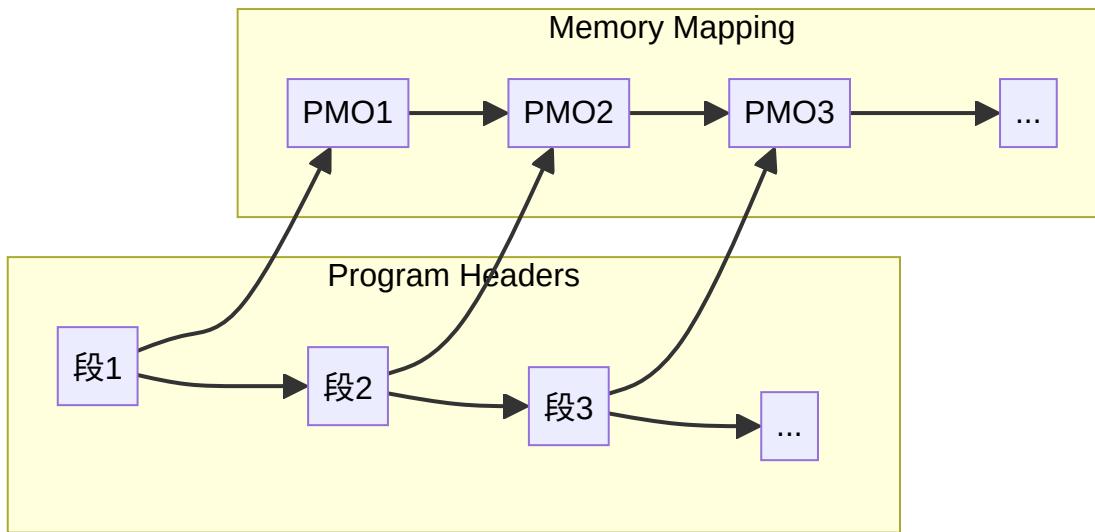
主要又分为几个小的步骤：

- **创建根能力组：**
 - 创建一个根能力组（root_cap_group），这是管理线程和进程的能力组
- **获取初始化虚拟地址空间：**
 - 从根能力组中获取初始化虚拟地址空间（init_vmspace）
- **为根线程分配用户栈：**
 - 分配一个物理内存对象（PMO）作为根线程的用户栈，并将其映射到初始化虚拟地址空间
- **分配根线程：**
 - 分配一个线程对象（thread）
- **映射程序头表项：**
 - 遍历程序头表项，为每个段分配PMO，并将其映射到初始化虚拟地址空间
 - 最后释放对初始化虚拟地址空间的引用

```
for (int i = 0; i < meta.phnum; i++) {  
    // 读取段信息  
    // 创建PMO  
    // 设置权限  
    // 映射到虚拟地址空间  
}  
obj_put(init_vmspace);
```

相关的示意图 下所示

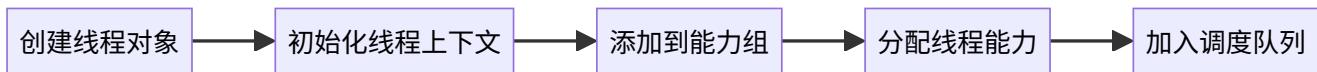




线程初始化

同理分为 下小的步骤：

- **准备环境：**
 - 为根线程准备环境，包括栈和程序入口点
- **初始化根线程：**
 - 使用根能力组、栈地址、入口点和优先级初始化根线程
 - 其中即包括初始化线程上下文的操作
- **将根线程添加到能力组的线程列表：**
 - 将根线程添加到根能力组的线程列表中，并增加线程计数
- **为根线程分配能力：**
 - 为根线程分配一个能力（`thread_cap`）
- **刷新缓存：**
 - 刷新指令缓存和数据缓存，以确保新线程的指令和数据是最新的
- **将根线程放入就绪队列：**
 - 将根线程放入调度器的就绪队列，准备执行



除了初始化线程之外，类比能力组的创建同样有关于创建系统线程的syscall即 `sys_create_thread`，其中涉及到的逻辑便比初始线程的创立简单不少，感兴趣的读者可以自行阅读。二者的区别 下图所示：

create_root_thread	sys_create_thread
+无参数检查	+严格参数检查
+固定优先级	+多种线程类型
+直接访问root_cap_group	+可配置优先级
+加载进程管理器二进制	+通过能力访问cap_group
+固定类型(TYPE_USER)	+不加载二进制

init_thread_ctx

初始化上下文的函数相对就要简单很多，根据规则配置相应的寄存器和一些字段即可

源码解析

接口：

```
void init_thread_ctx(  
    struct thread *thread, // 线程对象  
    vaddr_t stack, // 用户栈地址  
    vaddr_t func, // 入口函数地址  
    u32 prio, // 优先级  
    u32 type, // 线程类型  
    s32 aff // CPU亲和性  
)
```

源码：

```
void init_thread_ctx(struct thread *thread, vaddr_t stack, vaddr_t func,
                     u32 prio, u32 type, s32 aff)
{
    /* Fill the context of the thread */
    thread->thread_ctx->ec.reg[SP_EL0] = stack;
    thread->thread_ctx->ec.reg[ELR_EL1] = func;
    thread->thread_ctx->ec.reg[SPSR_EL1] = SPSR_EL1_EL0t;

    /* Set the state of the thread */
    thread->thread_ctx->state = TS_INIT;

    /* Set thread type */
    thread->thread_ctx->type = type;

    /* Set the cpuid and affinity */
    thread->thread_ctx->affinity = aff;

    /* Set the budget and priority of the thread */
    if (thread->thread_ctx->sc != NULL) {
        thread->thread_ctx->sc->prio = prio;
        thread->thread_ctx->sc->budget = DEFAULT_BUDGET;
    }

    thread->thread_ctx->kernel_stack_state = KS_FREE;
    /* Set exiting state */
    thread->thread_ctx->thread_exit_state = TE_RUNNING;
    thread->thread_ctx->is_suspended = false;
}
```

实际上就是设置寄存器、线程状态、调度相关设置以及状态标志

相关数据结构

线程上下文的数据结构 下所示，结构体成员的作用都写在注释里了，可以直接参考注释食用

~

```
/* 线程上下文：包含线程运行所需的所有上下文信息 */
struct thread_ctx {
    /* ===== 架构相关部分 ===== */
    /* 执行上下文：包含CPU寄存器状态
     * 在ARM64中包括：
     * - 通用寄存器 (X0-X30)
     * - 程序计数器 (PC)
     * - 栈指针 (SP)
     * - 处理器状态寄存器 (PSTATE)
     */
    arch_exec_ctx_t ec;

    /* 浮点运算单元 (FPU) 状态
     * 存储浮点寄存器和SIMD寄存器的内容
     * 仅在线程使用FPU时才会被保存和恢复
     */
    void *fpu_state;

    /* 线程局部存储 (TLS) 相关寄存器
     * 用于支持线程局部存储
     * 在ARM64中通常使用TPIDR_EL0寄存器
     */
    unsigned long tls_base_reg[TLS_REG_NUM];

    /* ===== 架构无关部分 ===== */
    /* FPU所有权标识
     * -1: 不拥有FPU
     * >=0: 表示拥有FPU的CPU ID
     * 用于延迟FPU上下文的保存和恢复
     */
    int is_fpu_owner;

    /* 调度上下文
     * 包含调度器需要的信息：
     * - 优先级
     * - 时间片
     * - 调度策略相关数据
     */
    sched_ctx_t *sc;

    /* 线程类型
     * 可能的值：
     * - TYPE_USER: 用户线程
     * - TYPE_KERNEL: 内核线程
     */
    unsigned int type;

    /* 线程状态 (不能被其他核心修改)
```

```
* 可能的值:  
* - TS_INIT: 初始化  
* - TS_READY: 就绪  
* - TS_RUNNING: 运行中  
* - TS_WAITING: 等待中  
*/  
unsigned int state;  
  
/* 线程挂起标志  
 * true: 线程被挂起  
 * false: 线程正常运行  
 */  
bool is_suspended;  
  
/* SMP亲和性  
 * -1: 可在任何CPU上运行  
 * >=0: 指定运行的CPU ID  
 */  
int affinity;  
  
/* 当前分配的CPU  
 * 记录线程当前或最后运行的CPU ID  
 */  
unsigned int cpuid;  
  
/* 内核栈状态  
 * volatile: 可能被中断处理修改  
 * 可能的值:  
 * - KS_FREE: 空闲  
 * - KS_LOCKED: 被锁定  
 */  
volatile unsigned int kernel_stack_state;  
  
/* 线程退出状态  
 * volatile: 可能被其他上下文修改  
 * 可能的值:  
 * - TE_RUNNING: 运行中  
 * - TE_EXITING: 正在退出  
 * - TE_EXITED: 已退出  
 */  
volatile unsigned int thread_exit_state;  
  
} /* 按照缓存行大小对齐, 避免伪共享 */  
__attribute__((aligned(CACHELINE_SZ)));
```

至此，线程管理相关的源码解析到此结束

Last change: 2025-02-14, commit: [9fa9752](#)

本部分内容讲解ChCore异常管理的部分

回顾：ARM异常ARM ABI

ARM 异常分类

这部分内容在缺页管理中首次提到

ARM将异常分为同步异常和异步异常两大类。同步异常是由指令执行直接引发的，例如 系统调用、页面错误或非法指令等，这类异常具有**确定性**，每次执行到特定指令时都会触发。而异步异常包括硬件中断（IRQ）、快速中断（FIQ）和错误（ERROR），它们与当前指令无关，通常由**外部事件或硬件故障引起**

- **sync**: 同步异常， 系统调用或页面错误。
- **irq**: 硬件中断请求（IRQ），由外部设备生成的中断。
- **fiq**: 快速中断请求（FIQ），用于更高优先级的中断处理。
- **error**: 处理其他类型的错误， 未定义指令或故障。

ARM C ABI

参数传递规则

- **寄存器传递**

前六个整型或指针参数（32/64位）依次通过寄存器x0-x5传递。例如：

- 第1个参数 → x0
- 第2个参数 → x1
- ...
- 第6个参数 → x5

- **栈传递**

若参数超过六个，剩余参数按声明顺序从右向左压入栈空间，由调用者分配和释放。

返回值传递规则

- **小型返回值（≤8字节）**

单个整型、指针或小结构体（≤64位）直接通过x0寄存器返回。

- **中型结构体（≤16字节）**

通过x0寄存器返回指向该结构体的内存指针，内存由调用者预分配（例 在栈上）。

- **大型结构体（>16字节）**

调用者需提前在栈中分配内存，并将该内存地址写入x8寄存器。被调用函数通过x8找到目标地址，直接将结构体内容写入此内存区域，同时x0也会返回该地址。

ChCore异常处理

现在我们再来看ChCore对异常处理的实际实现

Lab文档提到：

在 AArch64 中，存储于内存之中的异常处理程序代码被叫做异常向量（exception vector），而所有的异常向量被存储在一张异常向量表（exception vector table）中。

AArch64 中的每个异常级别都有其自己独立的异常向量表，其虚拟地址由该异常级别下的异常向量基址寄存器（VBAR_EL3，VBAR_EL2 和 VBAR_EL1）决定。每个异常向量表中包含 16 个条目，每个条目里存储着发生对应异常时所需执行的异常处理程序代码。

由于ChCore仅使用了EL0和EL1两个异常级别，故异常向量表也只有EL1这一张

我们还是先看看源码的实现，再逐步分析

源码

重复或者不重要的部分已省略

```
// .....
```

```
.macro exception_entry label
/* Each entry of the exception table should be 0x80 aligned */
.align 7
b \label
.endm
```

```
/* See more details about the bias in registers.h */
.macro exception_enter
sub sp, sp, #ARCH_EXEC_CONT_SIZE
stp x0, x1, [sp, #16 * 0]
stp x2, x3, [sp, #16 * 1]
stp x4, x5, [sp, #16 * 2]
stp x6, x7, [sp, #16 * 3]
stp x8, x9, [sp, #16 * 4]
stp x10, x11, [sp, #16 * 5]
stp x12, x13, [sp, #16 * 6]
stp x14, x15, [sp, #16 * 7]
// ...
.endm
```

```
.macro exception_exit
ldp x22, x23, [sp, #16 * 16]
ldp x30, x21, [sp, #16 * 15]
msr sp_el0, x21
msr elr_el1, x22
msr spsr_el1, x23
ldp x0, x1, [sp, #16 * 0]
ldp x2, x3, [sp, #16 * 1]
ldp x4, x5, [sp, #16 * 2]
ldp x6, x7, [sp, #16 * 3]
ldp x8, x9, [sp, #16 * 4]
// ...
add sp, sp, #ARCH_EXEC_CONT_SIZE
eret
.endm
```

```
.macro switch_to_cpu_stack
mrs x24, TPIDR_EL1
add x24, x24, #OFFSET_LOCAL_CPU_STACK
ldr x24, [x24]
mov sp, x24
.endm
```

```
.macro switch_to_thread_ctx
mrs x24, TPIDR_EL1
add x24, x24, #OFFSET_CURRENT_EXEC_CTX
ldr x24, [x24]
mov sp, x24
.endm
```

```
/* el1_vector should be set in VBAR_EL1. The last 11 bits of VBAR_EL1 are
reserved. */
.align 11
EXPORT(el1_vector)
exception_entry sync_el1t // Synchronous EL1t
```

```
exception_entry irq_el1t // IRQ EL1t
exception_entry fiq_el1t // FIQ EL1t
exception_entry error_el1t // Error EL1t

exception_entry sync_el1h // Synchronous EL1h
exception_entry irq_el1h // IRQ EL1h
exception_entry fiq_el1h // FIQ EL1h
exception_entry error_el1h // Error EL1h

exception_entry sync_el0_64 // Synchronous 64-bit EL0
exception_entry irq_el0_64 // IRQ 64-bit EL0
exception_entry fiq_el0_64 // FIQ 64-bit EL0
exception_entry error_el0_64 // Error 64-bit EL0

exception_entry sync_el0_32 // Synchronous 32-bit EL0
exception_entry irq_el0_32 // IRQ 32-bit EL0
exception_entry fiq_el0_32 // FIQ 32-bit EL0
exception_entry error_el0_32 // Error 32-bit EL0

/*
 * The selected stack pointer can be indicated by a suffix to the Exception
Level:
 * - t: SP_EL0 is used
 * - h: SP_ELx is used
 *
 * ChCore does not enable or handle irq_el1t, fiq_xxx, and error_xxx.
 * The SPSR_EL1 of idle threads is set to 0b0101, which means interrupt
 * are enabled during the their execution and SP_EL1 is selected (h).
 * Thus, irq_el1h is enabled and handled.
 *
 * Similarly, sync_el1t is also not enabled while we simply reuse the handler
for
 * sync_el0 to handle sync_el1h (e.g., page fault during copy_to_user and
fpu).
 */
irq_el1h:
    /* Simply reusing exception_enter/exit is OK. */
    exception_enter
#ifndef CHCORE_KERNEL_RT
    switch_to_cpu_stack
#endif
    bl handle_irq_el1
    /* should never reach here */
    b .

irq_el1t:
fiq_el1t:
fiq_el1h:
error_el1t:
error_el1h:
sync_el1t:
    bl unexpected_handler

sync_el1h:
    exception_enter
    mov x0, #SYNC_EL1h
```

```
mrs x1, esr_el1
mrs x2, elr_el1
bl handle_entry_c
str    x0, [sp, #16 * 16] /* store the return value as the ELR_EL1 */
exception_exit

sync_el0_64:
exception_enter
#ifndef CHCORE_KERNEL_RT
switch_to_cpu_stack
#endif
mrs x25, esr_el1
lsr x24, x25, #ESR_EL1_EC_SHIFT
cmp x24, #ESR_EL1_EC_SVC_64
b.eq el0_syscall
mov x0, SYNC_EL0_64
mrs x1, esr_el1
mrs x2, elr_el1
bl handle_entry_c
#ifdef CHCORE_KERNEL_RT
bl do_pending_resched
#else
switch_to_thread_ctx
#endif
exception_exit

el0_syscall:

/* hooking syscall: ease tracing or debugging */
#ifndef ENABLE_HOOKING_SYSCALL == ON
sub sp, sp, #16 * 8
stp x0, x1, [sp, #16 * 0]
stp x2, x3, [sp, #16 * 1]
stp x4, x5, [sp, #16 * 2]
stp x6, x7, [sp, #16 * 3]
stp x8, x9, [sp, #16 * 4]
stp x10, x11, [sp, #16 * 5]
stp x12, x13, [sp, #16 * 6]
stp x14, x15, [sp, #16 * 7]

mov x0, x8
bl hook_syscall

ldp x0, x1, [sp, #16 * 0]
ldp x2, x3, [sp, #16 * 1]
ldp x4, x5, [sp, #16 * 2]
ldp x6, x7, [sp, #16 * 3]
ldp x8, x9, [sp, #16 * 4]
ldp x10, x11, [sp, #16 * 5]
ldp x12, x13, [sp, #16 * 6]
ldp x14, x15, [sp, #16 * 7]
add sp, sp, #16 * 8
#endif

adr x27, syscall_table // syscall table in x27
uxtw x16, w8      // syscall number in x16
ldr x16, [x27, x16, lsl #3] // find the syscall entry
```

```
blr x16

/* Ret from syscall */
// bl disable_irq
#ifndef CHCORE_KERNEL_RT
    str x0, [sp]
    bl do_pending_resched
#else
    switch_to_thread_ctx
    str x0, [sp]
#endif
exception_exit

irq_el0_64:
    exception_enter
#ifndef CHCORE_KERNEL_RT
    switch_to_cpu_stack
#endif
    bl handle_irq
    /* should never reach here */
    b .

error_el0_64:
sync_el0_32:
irq_el0_32:
fiq_el0_32:
error_el0_32:
    bl unexpected_handler

fiq_el0_64:
    exception_enter
#ifndef CHCORE_KERNEL_RT
    switch_to_cpu_stack
#endif
    bl handle_fiq
    /* should never reach here */
    b .

// 实现线程切换功能，通过异常返回机制切换到目标线程
/* void eret_to_thread(u64 sp) */
BEGIN_FUNC(__eret_to_thread)
    mov sp, x0
    dmb ish /* smp_mb() */
#ifndef CHCORE_KERNEL_RT
    bl finish_switch
#endif
    exception_exit
END_FUNC(__eret_to_thread)
```

解析

异常向量表

向量表内容本身即和文档中图片所示结构一致，注意一下它是内存对齐的，通过宏定义实现：

```
.macro exception_entry label
/* Each entry of the exception table should be 0x80 aligned */
.align 7
b \label
.endm
```

这里的 .align 7 便把内存对齐到了 0x80 字节

关键宏定义

- exception_enter : 保存CPU上下文，包括通用寄存器和系统寄存器
- exception_exit : 恢复CPU上下文并返回
- switch_to_cpu_stack : 切换到CPU栈
- switch_to_thread_ctx : 切换到线程上下文

注意保存CPU上下文的这两个宏，它们采用的方式是直接把寄存器的值保存在了内核栈上。果你看过xv6的代码，就会发现这不同于xv6的_trampoline在内核页表上规定了一个特殊的位置 (trampoline page) 用于保存寄存器

两种设计的权衡（选读）：

trampoline page实现起来较易，但 果要支持多线程和抢占式调度，则相对麻烦

早期 Unix 选择 trampoline page 主要是因为当时的硬件限制，而现代 ARM OS 选择内核栈是因为现代硬件的进步和对系统安全性和性能的更高要求。两种方法各有优劣，选择哪种方法需要根据具体的硬件架构和操作系统设计目标进行权衡。

一些早期的操作系统，例 xv6-riscv，仍然使用 trampoline page 来处理用户态到内核态的转换。[2](#) Linux 内核中也使用了 trampoline 的概念，但其作用和实现方式与早期 Unix 中的 trampoline page 不同。Linux 中的 trampoline 主要用于处理内核地址空间布局随机化 (KASLR) 等安全特性。[1](#)

总而言之，选择 trampoline page 还是内核栈是一个 trade-off 的过程。Trampoline page 实现简单，节省内存，但安全性较低，难以支持多处理器和抢占式调度。内核栈实现复杂，占用内存较多，但安全性更高，更易于支持多处理器和抢占式调度。现代操作系统大多选择内核栈，是因为现代硬件的性能提升使得内核栈的开销可以接受，并且内核栈带来的安全性提升和功能扩展更加重要。

异常处理流程（C ABI兼容）

这部分又分为两大类：系统调用和普通异常处理

- 系统调用：

在AArch64中，系统调用由 `svc` 指令执行，这时触发 `sync_el0_64` 异常，进入向量表：

```
sync_el0_64:
    exception_enter                                // 保存上下文
#ifndef CHCORE_KERNEL_RT
    switch_to_cpu_stack                            // 非RT模式下切换到CPU栈
#endif
    mrs x25, esr_el1                               // 读取异常综合寄存器(ESR)
    lsr x24, x25, #ESR_EL1_EC_SHIFT             // 获取异常类别
    cmp x24, #ESR_EL1_EC_SVC_64                 // 判断是否为系统调用
    b.eq el0_syscall                             // 果是系统调用，跳转处理
```

随后进入系统调用的核心逻辑（这里的钩子是用来调试+监控的）

```
el0_syscall:
    // 果启用了系统调用钩子
#if ENABLE_HOOKING_SYSCALL == ON
    // 保存寄存器x0-x15
    sub sp, sp, #16 * 8
    stp x0, x1, [sp, #16 * 0]
    // ... 保存其他寄存器 ...

    mov x0, x8          // 系统调用号作为参数
    bl hook_syscall    // 调用钩子函数

    // 恢复寄存器
    ldp x0, x1, [sp, #16 * 0]
    // ... 恢复其他寄存器 ...
    add sp, sp, #16 * 8
#endif
    // 系统调用处理核心逻辑
    adr x27, syscall_table                         // 获取系统调用表地址
    uxtw x16, w8                                    // 系统调用号（保存在x8中）
    ldr x16, [x27, x16, lsl #3]                   // 查找系统调用处理函数
    blr x16                                         // 调用对应的处理函数
```

从系统调用表返回后，进行返回处理：

```
// 系统调用返回值处理
#ifndef CHCORE_KERNEL_RT
    str x0, [sp]                                // 保存返回值
    bl do_pending_resched                         // RT模式下检查是否需要重新调度
#else
    switch_to_thread_ctx                          // 切换回线程上下文
    str x0, [sp]                                // 保存返回值
#endif
exception_exit                                  // 恢复上下文并返回用户态
```

- 普通异常处理：以 sync_el1h 为例

```
sync_el1h:
exception_enter
    mov x0, #SYNC_EL1h
    mrs x1, esr_el1
    mrs x2, elr_el1
    bl handle_entry_c
    str    x0, [sp, #16 * 16] /* store the return value as the ELR_EL1 */
exception_exit
```

大体上即为保存上下文——调用函数——恢复上下文的流程

注意这里的C ABI的体现， x0, x1, x2 就是c函数的args, 返回值置于 x0

而对于其他异常/还没有实现的异常，则直接调用 unexpected_handler

```
error_el0_64:
sync_el0_32:
irq_el0_32:
fiq_el0_32:
error_el0_32:
    bl unexpected_handler
```

至此，异常管理部分源码解析结束

Last change: 2025-02-14, commit: [9fa9752](#)

系统调用是系统为用户程序提供的高特权操作接口。在本实验中，用户程序通过 `svc` 指令进入内核模式。在内核模式下，首先操作系统代码和硬件将保存用户程序的状态。操作系统根据系统调用号码执行相应的系统调用处理代码，完成系统调用的实际功能，并保存返回值。最后，操作系统和硬件将恢复用户程序的状态，将系统调用的返回值返回给用户程序，继续用户程序的执行

书接上回，在异常管理的部分已经讲了系统调用的整体流程。本部分内容将讲解其实现细节，并以 `printf` 函数为例探究一次系统调用的逻辑关系链

系统调用流程

我们在异常管理部分已经分析了系统调用的大体流程：

- 保存上下文，即当前线程的各个寄存器值，该工作由 `exception_enter` 完成。结合上回分解我们知道它们是直接被保存在内核栈上的
- 切换到内核栈，即 `switch_to_cpu_stack` 宏，此时由用户态进入内核态
- 根据系统调用表进行跳转，并执行相应的函数
- 处理返回值，恢复上下文，该工作由 `exception_exit` 完成
- 结束系统调用

内核栈切换

这里重点再分析一下之前没有讲到的内核栈切换，先看源码：

```
.macro switch_to_cpu_stack
    mrs      x24, TPIDR_EL1
    add x24, x24, #OFFSET_LOCAL_CPU_STACK
    ldr x24, [x24]
    mov sp, x24
.endm
```

注意到这个寄存器 `TPIDR_EL1`，Lab 文档告诉我们它可以读取到当前核的 `per_cpu_info`，我们作更深一步的了解：

`TPIDR_EL1` (Thread Process ID Register for EL1) 是 ARM 架构中一个特殊的寄存器，**用于存储当前执行线程或进程的上下文信息**。在操作系统内核中，这个寄存器经常被用来存储指向 `per_cpu_data` 结构的指针，该结构包含了特定于 CPU 的数据，比 CPU 的局部变量和栈指针

实质上，这是个“保留寄存器”，硬件上没有对其的直接操作，留给操作系统实现者自行使用。具体的初始化和设置在 `smp` 之中，`chcore` 将其设置为指向 `per_cpu_info` 结构体的指针，并且在之后不再变化

CPU信息结构体

现在让我们来看看这个结构体是个什么东东：

```
struct per_cpu_info {
    /* The execution context of current thread */
    u64 cur_exec_ctx;

    /* Per-CPU stack */
    char *cpu_stack;

    /* struct thread *fpu_owner */
    void *fpu_owner;
    u32 fpu_disable;

    char pad[pad_to_cache_line(sizeof(u64) +
        sizeof(char *) +
        sizeof(void *) +
        sizeof(u32))];
} __attribute__((packed, aligned(64)));
```

其中FPU指浮点运算单元，这个指针即表示当前使用FPU的线程，最后的pad以及结尾的编译器声明则旨在让结构体按照64字节大小对齐，从而避免多个CPU核心访问同一缓存行导致的性能问题

那么 TPIDR_EL1 又是在哪里被设置的呢？我们顺着看其init函数：

```
void init_per_cpu_info(u32 cpuid)
{
    struct per_cpu_info *info;

    if (cpuid == 0)
        ctr_el0 = read_ctr();

    info = &cpu_info[cpuid];

    info->cur_exec_ctx = 0;

    info->cpu_stack = (char *) (KSTACKx_ADDR(cpuid) + CPU_STACK_SIZE);

    info->fpu_owner = NULL;
    info->fpu_disable = 0;
    // 寄存器在此处被初始化
    asm volatile("msr tpidr_el1, %0:::r" (info));
}
```

这样一来，切换内核栈的那部分汇编代码就好理解了：系统直接按照结构体的大小读出CPU的栈指针，然后一把塞到 sp 寄存器里，即完成了栈的切换。那个 #OFFSET_XXX 宏的定义自然也能猜到是什么了，事实上，它就定义在 smp.h 中：

```

/*
 * The offset in the per_cpu struct, i.e., struct per_cpu_info.
 * The base addr of this struct is stored in TPIDR_EL1 register.
 *
 * IMPORTANT: modify the following offset values after
 * modifying struct per_cpu_info.
 */
#define OFFSET_CURRENT_EXEC_CTX 0
#define OFFSET_LOCAL_CPU_STACK 8
#define OFFSET_CURRENT_FPU_OWNER 16
#define OFFSET_FPU_DISABLE 24

```

用户态libc支持

接下来我们尝试分析printf这个用户态函数，文档已经给出了他在musl-libc之中的调用链，而跟踪这个调用链，我们就可以一窥API和ABI的边界

从printf到__stdio_write

由Lab文档知，printf经过一系列调用，会来到 __stdout_write 函数，并进一步去到 __stdio_write 函数

```

// user/system-services/chcore-libc/musl-libc/src/stdio/__stdout_write.c
size_t __stdout_write(FILE *f, const unsigned char *buf, size_t len)
{
    struct winsize wsz;
    f->write = __stdio_write;
    if (!(f->flags & F_SVB) && __syscall(SYS_ioctl, f->fd, TIOCGWINSZ, &wsz))
        f->lbf = -1;
    return __stdio_write(f, buf, len);
}

// user/system-services/chcore-libc/musl-libc/src/stdio/__stdio_write.c
size_t __stdio_write(FILE *f, const unsigned char *buf, size_t len)
{
    struct iovec iovs[2] = {
        { .iov_base = f->wbase, .iov_len = f->wpos-f->wbase },
        { .iov_base = (void *)buf, .iov_len = len }
    };
    struct iovec *iov = &iovs[0];
    size_t rem = iov[0].iov_len + iov[1].iov_len;
    int iovcnt = 2;
    ssize_t cnt;
    for (;;) {
        // HERE!!!
        cnt = syscall(SYS_writev, f->fd, iov, iovcnt);
        // ...循环剩余内容
    }
}

```

这里的 `SYS_writev` 是一个用户态中的宏，负责表示系统调用编号，从而和系统调用联系起来

用户态syscall宏展开

继续深究这里的 syscall 宏，其中暗藏大量玄机：

```
#ifndef __scc
#define __scc(X) ((long) (X))
typedef long syscall_arg_t;
#endif

hidden long __syscall_ret(unsigned long),
__syscall_cp(syscall_arg_t, syscall_arg_t, syscall_arg_t, syscall_arg_t,
            syscall_arg_t, syscall_arg_t, syscall_arg_t);

#define __syscall1(n,a) __syscall1(n,__scc(a))
#define __syscall2(n,a,b) __syscall2(n,__scc(a),__scc(b))
#define __syscall3(n,a,b,c) __syscall3(n,__scc(a),__scc(b),__scc(c))
#define __syscall4(n,a,b,c,d)
__syscall4(n,__scc(a),__scc(b),__scc(c),__scc(d))
#define __syscall5(n,a,b,c,d,e)
__syscall5(n,__scc(a),__scc(b),__scc(c),__scc(d),__scc(e))
#define __syscall6(n,a,b,c,d,e,f)
__syscall6(n,__scc(a),__scc(b),__scc(c),__scc(d),__scc(e),__scc(f))
#define __syscall7(n,a,b,c,d,e,f,g)
__syscall7(n,__scc(a),__scc(b),__scc(c),__scc(d),__scc(e),__scc(f),__scc(g))

#define __SYSCALL_NARGS_X(a,b,c,d,e,f,g,h,n,...) n
#define __SYSCALL_NARGS(...) __SYSCALL_NARGS_X(__VA_ARGS__,7,6,5,4,3,2,1,0,)
#define __SYSCALL_CONCAT_X(a,b) a##b
#define __SYSCALL_CONCAT(a,b) __SYSCALL_CONCAT_X(a,b)
#define __SYSCALL_DISP(b,...)
__SYSCALL_CONCAT(b,__SYSCALL_NARGS(__VA_ARGS__))(__VA_ARGS__)

#define __syscall(...) __SYSCALL_DISP(__syscall,__VA_ARGS__)
#define syscall(...) __syscall_ret(__syscall(__VA_ARGS__))
```

这里循环套圈了很多，我们做一个拆解：

```

// 原始调用
syscall(SYS_write, fd, buf, count);

// 1. 展开syscall宏
__syscall_ret(__syscall(SYS_write, fd, buf, count))

// 2. 展开__syscall宏
__syscall_ret(__SYSCALL_DISP(__syscall, SYS_write, fd, buf, count))

// 3. 确定参数数量 (3个) 并连接宏名
__syscall_ret(__syscall3(SYS_write, __scc(fd), __scc(buf), __scc(count)))

// 4. 类型转换
__syscall_ret(__syscall3(SYS_write,
    ((long)(fd)),
    ((long)(buf)),
    ((long)(count)))))

// 5. 最终调用实际的系统调用函数并处理返回值

```

遵循这个链继续到 `syscall_dispatcher.c` 文件，会发现它先经历了 `__syscall3` 后到 `__syscall6` 的调用，并进入 `chcore_writev`

这个函数只有三个参数，但是为什么会调用到有6个参数的syscall呢？这里既可能是为了灵活性的考量，也可能是不同架构下的write对应的syscall参数不同，选了比较大的那个（例 pwrite就需要5个参数）

继续追踪，来到 `chcore_write` 函数，这里调用了 `stdout` 这一个初始化的 `fd_ops` 的 `write` 函数

```

ssize_t chcore_write(int fd, void *buf, size_t count)
{
    if (fd < 0 || fd_dic[fd] == 0)
        return -EBADF;
    return fd_dic[fd]->fd_op->write(fd, buf, count);
}

```

找寻chcore_stdout_write真身

唉，发现找不下去了！这是因为这时候我们并不知道这里的 `write` 函数是什么！于是我们考虑反向搜寻，从 `chcore_stdout_write` 函数往前找：

```
// user/system-services/chcore-libc/libchcore/porting/overrides/src/chcore-
port/stdio.c
static ssize_t chcore_stdout_write(int fd, void *buf, size_t count)
{
    /* TODO: stdout should also follow termios flags */
    char buffer[STDOUT_BUFSIZE];
    size_t size = 0;

    for (char *p = buf; p < (char *)buf + count; p++) {
        if (size + 2 > STDOUT_BUFSIZE) {
            put(buffer, size);
            size = 0;
        }

        if (*p == '\n') {
            buffer[size++] = '\r';
        }
        buffer[size++] = *p;
    }

    if (size > 0) {
        put(buffer, size);
    }

    return count;
}
```

注意到这里使用了 `put` 函数，它则有了对 `syscall_table` 直接的调用，正式打破了用户态的边界：

```
static void put(char buffer[], unsigned size)
{
    chcore_syscall2(CHCORE_SYS_PUTSTR, (vaddr_t)buffer, size);
}
```

但是还有个问题：我们的 `chcore_stdout_write` 又是如何从 `printf` 调用到的呢？

继续反向追踪，我们可以发现 下的结构体定义：

```
// user/system-services/chcore-libc/libchcore/porting/overrides/src/chcore-
port/stdio.c
struct fd_ops stdout_ops = {
    .read = chcore_stdio_read,
    .write = chcore_stdout_write,
    .close = chcore_stdout_close,
    .poll = chcore_stdio_poll,
    .ioctl = chcore_stdio_ioctl,
    .fcntl = chcore_stdio_fcntl,
};
```

继续顺藤摸瓜，我们就能找到用这个结构体来初始化 `fd_dic` 的函数了：

```

// user/system-services/chcore-libc/libchcore/porting/overrides/src/chcore-
port/syscall_dispatcher.c
/*
 * This function is local to libc and it will
 * only be executed once during the libc init time.
 *
 * It will be executed in the dynamic loader (for dynamic-apps) or
 * just before calling user main (for static-apps).
 * Nevertheless, when loading a dynamic application, it will be invoked
twice.
 * This is why the variable `initialized` is required.
 */
__attribute__((constructor(101))) void __libc_chcore_init(void)
{
    static int initialized = 0;
    int fd0, fd1, fd2;
    struct termios *ts;
    char *pidstr;
    size_t i;
    elf_auxv_t *auxv;

    // .....

    /* STDOUT */
    fd1 = alloc_fd();
    assert(fd1 == STDOUT_FILENO);
    fd_dic[fd1]->type = FD_TYPE_STDOUT;
    fd_dic[fd1]->fd = fd1;
    fd_dic[fd1]->fd_op = &stdout_ops; // 这里! ! ! !

    // .....
}

```

这样一来，我们就打通了printf的整个调用函数链条，最终通过put函数向内核态调用syscall，从api过渡到abi

分析FILE*的write函数

上面是printf的整个流程，最终得到了 chcore_stdout_write 和内核交互。但是我们熟知的 fopen等FILE*的write又在哪里呢

FILE 是一个等效于 _IO_FILE 结构体的宏，而后者在 user/system-services/chcore-libc/musl-libc/src/internal/stdio_impl.h 中有着声明

```
struct _IO_FILE {
    unsigned flags;
    unsigned char *rpos, *rend;
    int (*close)(FILE *);
    unsigned char *wend, *wpos;
    unsigned char *mustbezero_1;
    unsigned char *wbase;
    size_t (*read)(FILE *, unsigned char *, size_t);
    size_t (*write)(FILE *, const unsigned char *, size_t);
    off_t (*seek)(FILE *, off_t, int);
    unsigned char *buf;
    size_t buf_size;
    FILE *prev, *next;
    int fd;
    int pipe_pid;
    long lockcount;
    int mode;
    volatile int lock;
    int lbf;
    void *cookie;
    off_t off;
    char *getln_buf;
    void *mustbezero_2;
    unsigned char *shend;
    off_t shlim, shcnt;
    FILE *prev_locked, *next_locked;
    struct __locale_struct *locale;
};
```

实际上，这个 `_IO_FILE` 是OS实现者自己完成的，与POSIX对接只需要有 `read`, `write`, `seek`, `close`四个方法，它的实现可以用这个函数来说明

```

FILE * __fdopen(int fd, const char *mode)
{
    FILE *f;
    struct winsize wsz;

    /* Check for valid initial mode character */
    if (!strchr("rwa", *mode)) {
        errno = EINVAL;
        return 0;
    }

    /* Allocate FILE+buffer or fail */
    if (!(f=malloc(sizeof *f + UNGET + BUFSIZ))) return 0;

    /* Zero-fill only the struct, not the buffer */
    memset(f, 0, sizeof *f);

    /* Impose mode restrictions */
    if (!strchr(mode, '+')) f->flags = (*mode == 'r') ? F_NOWR : F_NORD;

    /* Apply close-on-exec flag */
    if (strchr(mode, 'e')) __syscall(SYS_fcntl, fd, F_SETFD, FD_CLOEXEC);

    /* Set append mode on fd if opened for append */
    if (*mode == 'a') {
        int flags = __syscall(SYS_fcntl, fd, F_GETFL);
        if (!(flags & O_APPEND))
            __syscall(SYS_fcntl, fd, F_SETFL, flags | O_APPEND);
        f->flags |= F_APP;
    }

    f->fd = fd;
    f->buf = (unsigned char *)f + sizeof *f + UNGET;
    f->buf_size = BUFSIZ;

    /* Activate line buffered mode for terminals */
    f->lbf = EOF;
    if (!(f->flags & F_NOWR) && !__syscall(SYS_ioctl, fd, TIOCGWINSZ, &wsz))
        f->lbf = '\n';

    /* Initialize op ptrs. No problem if some are unneeded. */
    f->read = __stdio_read;
    f->write = __stdio_write;
    f->seek = __stdio_seek;
    f->close = __stdio_close;

    if (!libc.threaded) f->lock = -1;

    /* Add new FILE to open file list */
    return __ofl_add(f);
}

```

我们从write往回找可以看到，在调用 `__fdopen` 的时候，我们由一个fd，动态地生成了这个 `_IO_FILE` 结构体，并把他的方法用 `__stdio_xx` 赋值

在 `__stdio_xx` 内部是 libc 库实现的逻辑，但落到最后是调用 `SYS_readv`, `SYS_read` 的 syscall

```
size_t __stdio_read(FILE *f, unsigned char *buf, size_t len)
{
    struct iovec iov[2] = {
        { .iov_base = buf, .iov_len = len - !f->buf_size },
        { .iov_base = f->buf, .iov_len = f->buf_size }
    };
    ssize_t cnt;

    cnt = iov[0].iov_len ? syscall(SYS_readv, f->fd, iov, 2)
        : syscall(SYS_read, f->fd, iov[1].iov_base, iov[1].iov_len);
    if (cnt <= 0) {
        f->flags |= cnt ? F_ERR : F_EOF;
        return 0;
    }
    if (cnt <= iov[0].iov_len) return cnt;
    cnt -= iov[0].iov_len;
    f->rpos = f->buf;
    f->rend = f->buf + cnt;
    if (f->buf_size) buf[len-1] = *f->rpos++;
    return len;
}
```

最后给到用户的就是 `fopen` 了

```
FILE *fopen(const char *restrict filename, const char *restrict mode)
{
    FILE *f;
    int fd;
    int flags;

    /* Check for valid initial mode character */
    if (!strchr("rwa", *mode)) {
        errno = EINVAL;
        return 0;
    }

    /* Compute the flags to pass to open() */
    flags = __fmodeflags(mode);

    fd = sys_open(filename, flags, 0666);
    if (fd < 0) return 0;
    if (flags & O_CLOEXEC)
        __syscall(SYS_fcntl, fd, F_SETFD, FD_CLOEXEC);

    f = __fdopen(fd, mode);
    if (f) return f;

    __syscall(SYS_close, fd);
    return 0;
}
```

由此我们可以得出，内核里面始终只维护fd，而用户态的FILE*其实是libc做的一层包装，如果想要自定义kernel，只需要保证SYS_readv, SYS_writev, SYS_read, SYS_write这些宏存在，并处理对应参数的syscall就行

关于stdout

众所周知，stdout只是一个stdout文件的宏，而stdout文件就是FILE*类型的

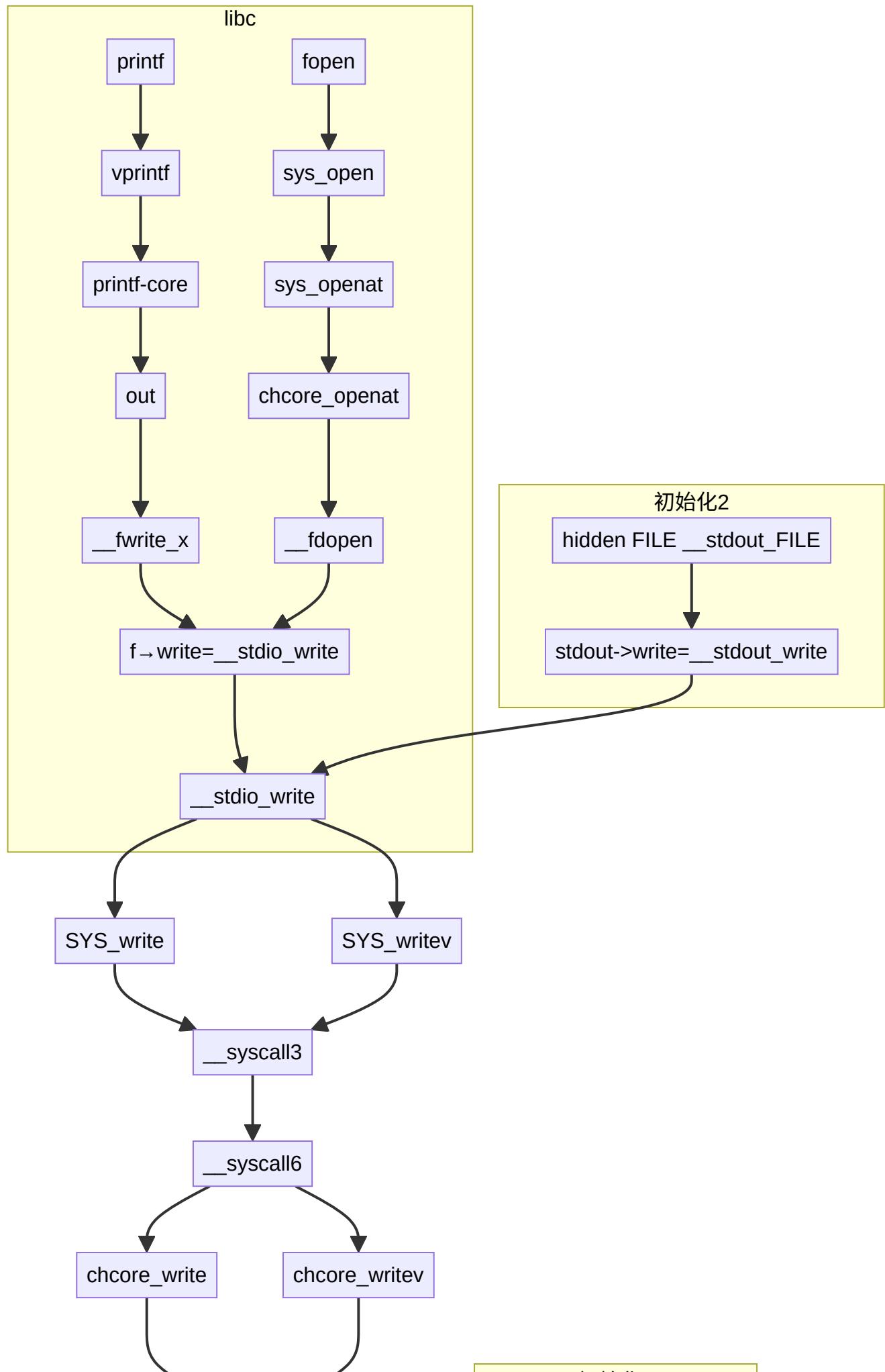
```
// user/system-services/chcore-libc/musl-libc/src/stdio/stdout.c
hidden FILE __stdout_FILE = {
    .buf = buf+UNGET,
    .buf_size = sizeof buf-UNGET,
    .fd = 1,
    .flags = F_PERM | F_NORD,
    .lbf = '\n',
    .write = __stdout_write,
    .seek = __stdio_seek,
    .close = __stdio_close,
    .lock = -1,
};
FILE *const stdout = &__stdout_FILE;

// user/system-services/chcore-libc/musl-libc/src/stdio/__stdout_write.c
#include "stdio_impl.h"
#include <sys/ioctl.h>

size_t __stdout_write(FILE *f, const unsigned char *buf, size_t len)
{
    struct winsize wsz;
    f->write = __stdio_write;
    if (!(f->flags & F_SVB) && __syscall(SYS_ioctl, f->fd, TIOCGWINSZ, &wsz))
        f->lbf = -1;
    return __stdio_write(f, buf, len);
}
```

调用逻辑图

最后，我们用一张逻辑链条图来结束今天的旅程：



用户态程序编写

万事具备，现在我们可以尝试自己动手编写我们的用户态程序了：

```
#include<stdio.h>

int main() {
    printf("Hello ChCore!\n");
    return 0;
}
```

按照文档指示，用已经生成好的工具链编译

不要忘了结果放到build/ramdisk文件夹里面，这样内核启动时将自动运行

```
./build/chcore-libc/bin/musl-gcc ./user/apps/my-apps/hello.c -o ./build/
ramdisk/hello-world.bin
```

然后 ./chbuild rambuild 重新生成内核镜像，再 ./build/simulate.sh 重新进入chcore
便可以看到我们编写的Hello-world!成功运行了

```
$ ./hello-world.bin  
Hello ChCore!
```

至此，系统调用部分的源码解析到此为止

Last change: 2025-02-14, commit: [9fa9752](#)



操作系统对于处理器资源的抽象是**进程和线程**。进程就是运行中的程序，他为程序提供了独享的处理器资源，从而简化了程序的编写。但是，一个进程只能运行在单个核心之上，在“多核”架构成为主流的当下，仅仅利用一个进程来处理任务显然无法充分利用计算资源。另外，由于进程包含的内容较多，且共享数据比较困难，基于多进程的方法会造成性能问题。因此，我们考虑在进程内部引入可并行的多个执行单元，并使其在不同核心上独立执行，即**线程**。

一、进程

1、宏内核视角

1、进程的内部表示——PCB

进程控制块（PCB）是一个结构体，包含着与进程相关的关键信息。

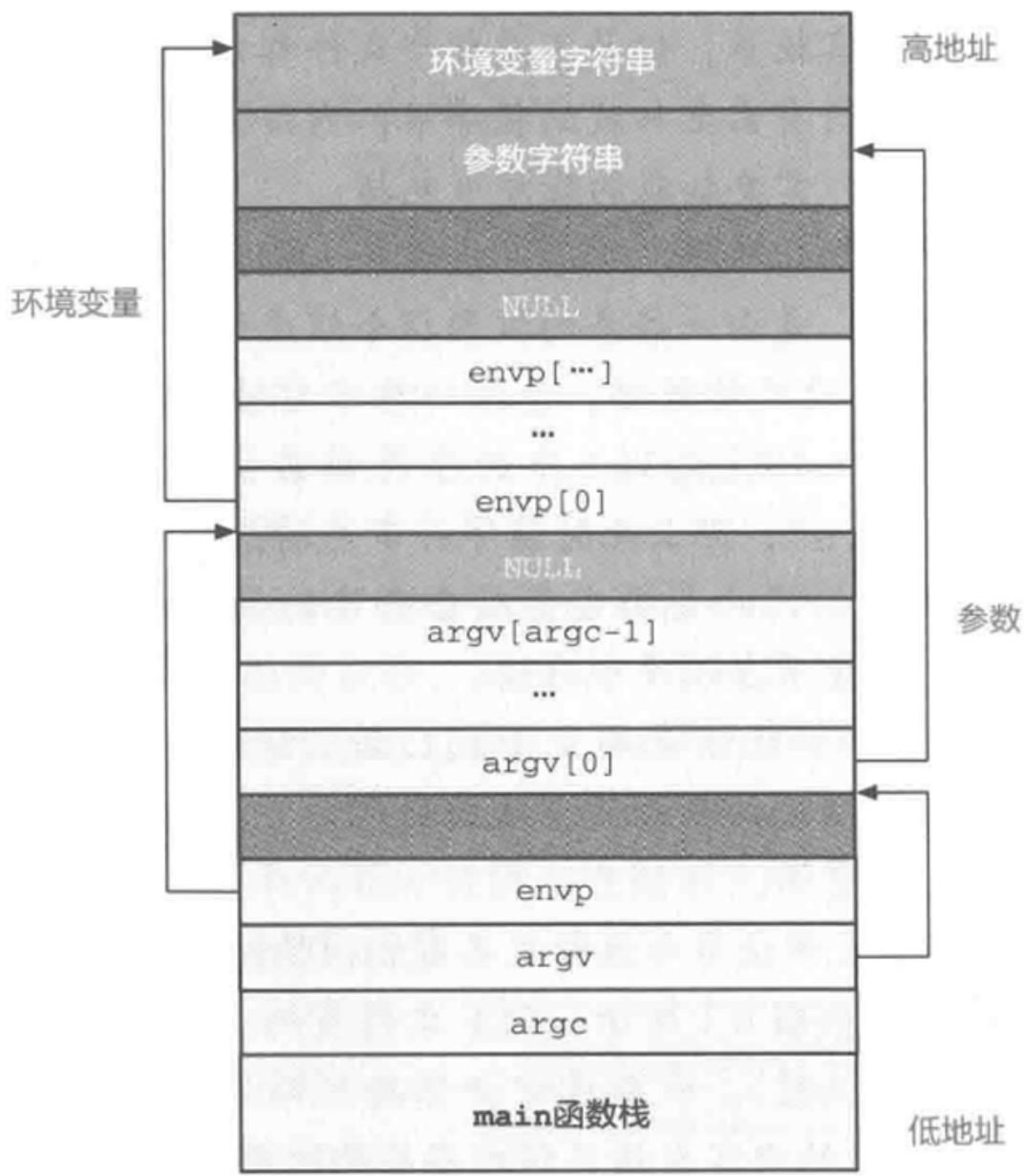
```
1 // PCB结构实现
2 struct process {
3     // 虚拟地址空间
4     struct vmspace *vmspace;
5     // 进程标识符
6     int pid;
7     // 子进程列表
8     pcb_list *children;
9     // 包含的线程列表
10    tcb_list *threads;
11    // 包含的线程总数
12    int thread_cnt;
13};
```

在引入线程之前，进程是操作系统进行资源分配和调度执行的单位，但引入线程之后，线程成为操作系统进行调度执行的单位，而进程主要负责资源管理。因此，与调度执行相关的信息（处理器上下文和执行状态）都从PCB中移入了PCB，而与资源管理相关的信息（比虚拟地址空间）依然保存在PCB中。另外，内核栈和退出状态也与执行相关，因此进程不再维护，改为由线程维护。

2、进程创建的实现

- 创建PCB。
- 虚拟内存初始化。
- 内核栈初始化。内核会预先分配物理页，作为进程的内核栈。

- 加载可执行文件到内存。其中，Linux中可执行文件格式为**可执行和可链接格式（ELF）**
- 初始化用户栈及运行环境。



- 处理器上下文初始化。大部分寄存器从未使用过，直接赋值为0，但是，由于特殊寄存器（PC,PSTATE,SP）保存了与硬件状态相关的信息，需要专门考虑。其中PC和PSTATE可以借由硬件机制，通过修改ELR_EL1来实现。SP则可以直接将用户栈的栈顶地址写入SP_EL0。

3、进程退出的实现

```
//进程退出的伪代码实现
void process_exit(int status)
{
    // 销毁上下文结构
    destroy_ctx(curr_proc->ctx);
    // 销毁虚拟地址空间
    destroy_vmspace(curr_proc->vmspace);
    // 保存退出状态
    curr_proc->exit_status = status;
    // 标记进程为退出状态
    curr_proc->is_exit = TRUE;

    // 告知内核选择下个需要执行的进程
    schedule();
}
```

注：我们假设内核维护着curr_proc变量，并总是指向当前正在运行的进程PCB。

schedule函数涉及内核的调度模块，即选择下一个需要执行的进程，我们在后文介绍其实现方法。

4、进程等待的实现

```

//进程等待的伪代码实现
void process_waitpid(int id, int *status)
{
    // 果没有子进程，直接返回
    if (!curr_proc->children)
        return;
    while (TRUE) {
        bool not_exist = TRUE;
        // 扫描子进程列表，寻找对应进程
        for (struct process *proc : curr_proc->children) {
            if (proc->pid == id) {
                // 标记已找到的对应进程，并检查其是否已经退出
                not_exist = FALSE;
                if (proc->is_exit) {
                    // 若发现该进程已经退出，记录其退出状态
                    *status = proc->exit_status;
                    // 销毁该进程的内核栈
                    destroy_kern_stack(proc->stack);
                    // 回收进程的 PCB 并返回
                    destroy_process(proc);
                    return;
                } else {
                    // 果没有退出，则调度下个进程执行
                    schedule();
                }
            }
        }
        // 果子进程列表中不存在该进程，则立即退出
        if (not_exist)
            return;
    }
}

```

核心逻辑

父进程等待指定子进程退出并回收其资源：

1. 若当前进程无子进程，直接返回
2. 循环检查子进程列表：
 - 遍历所有子进程，寻找 PID 匹配的目标进程
 - **若找到目标进程：**
 - 若目标已退出 → 记录退出状态、释放内核栈和 PCB → 结束等待
 - 若目标未退出 → 主动让出 CPU（触发调度），等待下次检查
 - **若未找到目标进程** → 立即结束等待

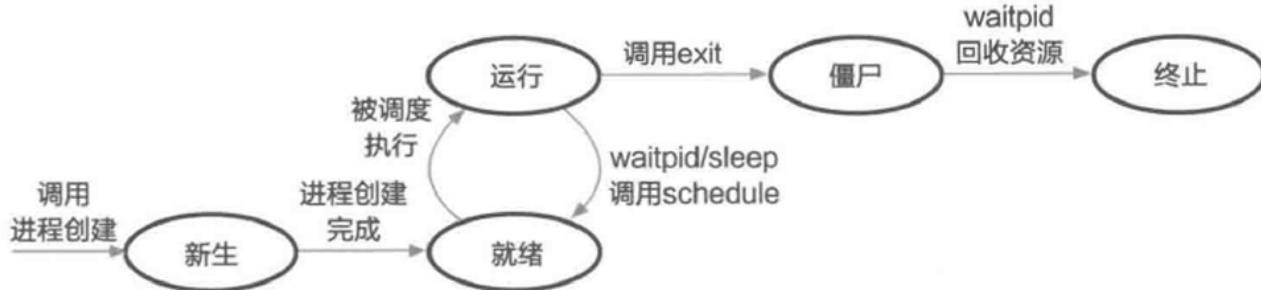
5、进程睡眠的实现

```
//进程睡眠的伪代码实现
void process_sleep(int seconds)
{
    struct *date start_time = get_time(); //获取当前时间作为睡眠起始时间
    while (TRUE) {
        struct *date cur_time = get_time();
        if (time_diff(cur_time, start_time) < seconds)
            schedule(); // 果时间未到，调度下一个进程执行
        else
            return; //时间已到，立即返回
    }
}
```

这个实现也很好理解，我们首先通过 `get_time()` 获取当前时间并存储在 `start_time` 中，作为基准点。然后进入一个无限循环 `while (TRUE)`，在循环中再次调用 `get_time()` 获取当前时间并存储在 `cur_time` 中。通过 `time_diff(cur_time, start_time)` 计算当前时间与开始时间的差值，果差值小于 `seconds`，则调用 `schedule()` 函数将当前进程挂起，让其他进程运行。果差值大于或等于 `seconds`，则退出循环，函数返回，表示进程的睡眠时间结束。

6、进程执行状态及其管理

首先，我们用一张图展示进程的五状态模型。



- **新生 (New) 状态**: `process_create` 被调用时，内核分配一个PCB，创建新进程，但初始化未完成，不能执行程序。
- **就绪 (Ready) 状态**: `process_create` 完成后，进程准备好执行，但可能需要等待调度器选择。
- **运行 (Running) 状态**: 调度器选择进程执行时，状态变为运行，之前运行的进程暂停，回到就绪状态。果进程调用 `schedule` 函数，也会回到就绪状态。
- **僵尸 (Zombie) 状态**: 子进程退出后，父进程调用 `process_waitpid` 获取退出状态，子进程的PCB未立即销毁，资源未完全回收。
- **终止 (Terminated) 状态**: 进程退出，所有资源被操作系统回收，进入最终状态。

此外，内核还可以引入**阻塞 (Blocked) **状态。

- **阻塞 (Blocked) 状态**: **需要在内核中等待，无法马上回到用户态执行的进程。**

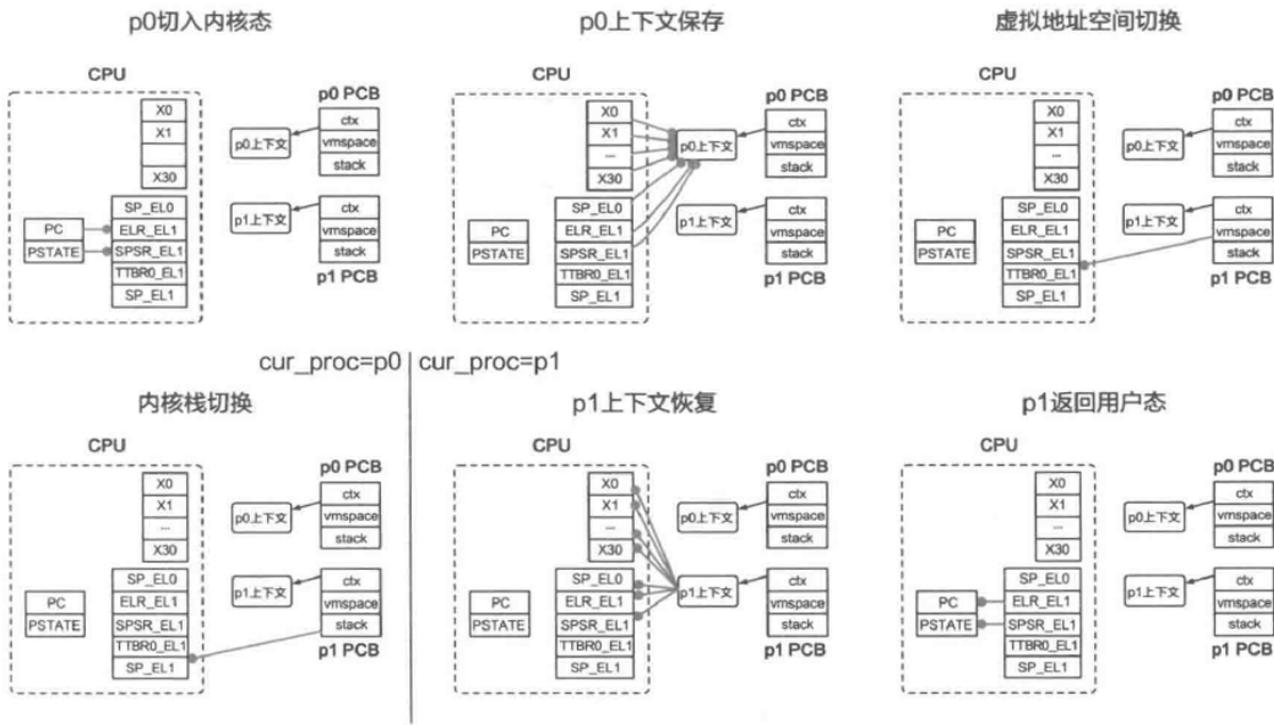
7、进程切换

进程切换的主要流程包括：原进程进入内核态→保存原进程的处理器上下文→切换进程上下文（切换虚拟地址空间和内核栈）→回复目标进程的处理器上下文→目标进程返回用户态。

进程的处理器数据结构包含以下寄存器中的值：

- 所有通用寄存器（X0 ~ X30）。
- 特殊寄存器中的用户栈寄存器 SP_ELO，需要手动保存以恢复栈顶地址。
- 系统寄存器中的 ELR_EL1 和 SPSR_EL1，用于保存程序计数器和处理器状态，确保在进程切换后正确恢复。

进程切换的触发方式分为主动和被动两种。主动切换是指进程主动放弃 CPU 资源，通过调用 process_exit、process_waitpid、process_sleep 等系统调用，最终调用 schedule 函数，使操作系统调度下一个进程执行。被动切换则是由操作系统强制触发，通常基于硬件中断实现，时钟中断，控制流转移到内核，进行切换。下面我们用一张图来展示进程切换的全过程。



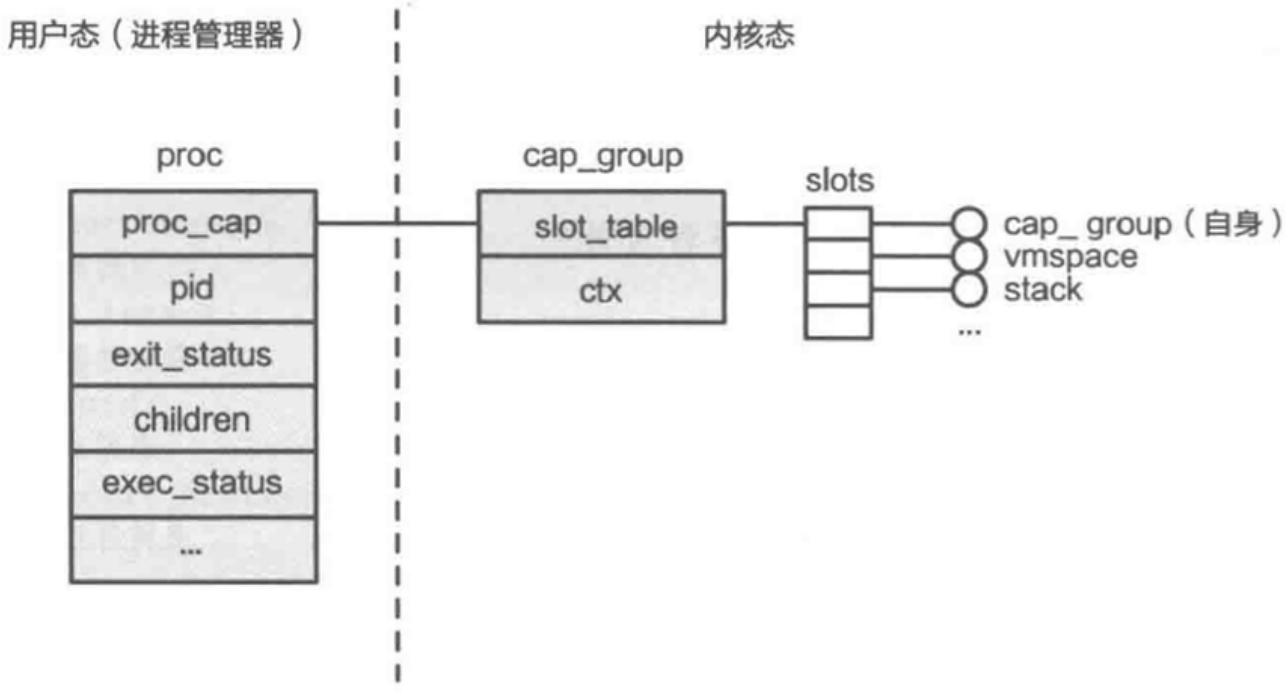
- 第一步**：p0从用户态进入内核态，通过系统调用、异常、中断等方法。硬件自动将PC和PSTATE寄存器的值分别保存到ELR_EL1和SPSR_EL1寄存器中。
- 第二步**：内核获取p0的处理器上下文结构，并将这些寄存器的值依次保存到处理器上下文中。
- 第三步**：内核获取p1的页表基地址，存储到TTBR0_EL1寄存器中，完成虚拟地址空间的切换。可能需要刷新TLB，防止后续执行时的地址翻译错误。
- 第四步**：内核将SP_EL1切换到进程p1私有的内核栈顶地址，完成内核栈的切换。内核不再访问与p0相关的数据，可以将curr_proc设置为p1，完成进程上下文的切换。
- 第五步**：内核从p1的PCB中获取其处理器上下文结构，并依次恢复到前述寄存器中。
- 第六步**：内核执行eret指令返回用户态，硬件自动将ELR_EL1和SPSR_EL1寄存器中的值恢复到PC和PSTATE中，p1恢复执行。

2、微内核视角——ChCore的进程管理

在宏内核中，与进程相关的数据结构（PCB）均放在内核中，管理接口也以系统调用的形式暴露给用户，核心功能全部在内核中完成。但在微内核中，包括进程管理在内的操作系统功能被拆分并移入用户态，因此与宏内核存在较大不同。

1、进程管理器与分离式PCB

ChCore操作系统采用了微内核设计，将功能分解并以模块形式移到用户态。进程管理功能也被移到用户态，形成了进程管理器（Process Manager）。用户进行进程管理操作时，实际上是调用了进程管理器，再由它与内核交互，实现进程管理。这种设计使进程控制块（PCB）从集中式变为分离式，PCB被分为内核态和用户态两部分。



- PCB内核态部分：cap_group ChCore 的PCB 在内核态的部分称为 cap_group，是能力组（Capability group）的简称。由于程序运行过程中需要不同类型的资源（如 CPU 和内存等），为了便于对资源进行管理，ChCore 对内核资源进行了抽象，每种资源对应一种类型的对象（Object），而访问某个具体对象所需的“凭证”就是能力（Capability）。由于 ChCore 将进程作为资源分配和管理的基本单位，因此进程自然也就成为拥有若干能力的**“能力组”——cap_group**。ChCore 的 cap_group 只包含两部分：存放对象的 slot_table 和处理器上下文 ctx。其中，slot_table 包含一个对象数组 slots，维护进程所持有的全部对象，包括它自身（进程本身也是对象）、虚拟地址空间、用户栈对应的物理内存等。而这些对象在数组中的偏移量就是它们对应的能力。另外，ChCore PCB 中的处理器上下文与宏内核 PCB 中保存的处理器上下文结构相同。
- PCB用户态部分：proc 保存了进程创建、退出、等待等功能所需的信息，还保存了与cap ——group对应的能力proc_cap，便于进程操作的实现。

2、ChCore的进程操作：以进程创建为例

```
//ChCore中进程创建(spawn) 的伪代码实现
int spawn(char *path, ...)
{
    // 根据指定的文件路径，加载需要执行的文件（同时创建相关物理内存对象PMO）
    struct user_elf *elf = readelf(path);

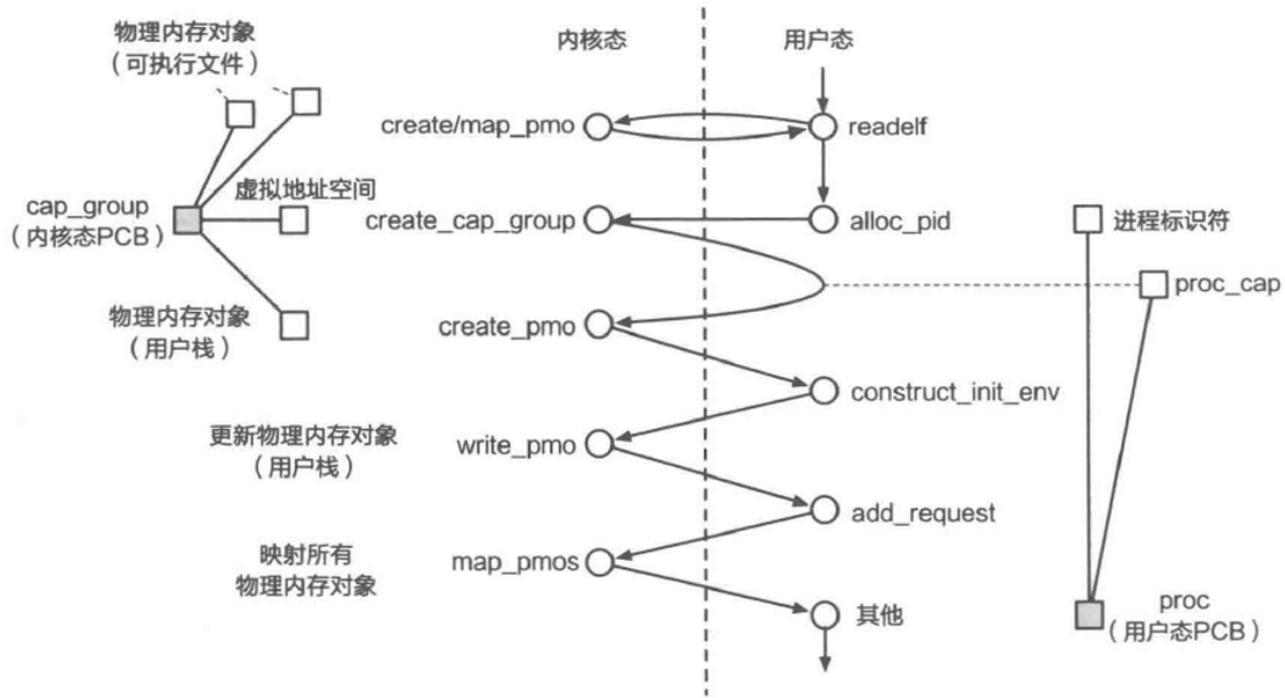
    // 进程管理器获取一个新的pid作为进程标识符
    int pid = alloc_pid();
    // 创建进程（包括创建PCB、虚拟地址空间、处理器上下文和内核栈）
    int new_process_cap = create_cap_group(pid, path, ...);

    // 为用户栈创建物理内存对象PMO
    int stack_cap = create_pmo(...);
    // 利用数组构造初始执行环境
    char init_env[ENV_SIZE];
    construct_init_env(init_env, elf, ...);
    // 更新执行环境到栈对应的PMO中
    write_pmo(stack_cap, init_env, ...);

    // 构建请求，用于映射栈对应的PMO
    struct pmo_map_request requests[MAX_REQ_SIZE];
    add_request(requests, stack_cap, STACK_VADDR, ...);
    // 构建请求，用于映射可执行文件中需要加载的段对应的PMO
    for (struct user_elf_seg seg: elf->loadable_segs)
        add_request(requests, seg.pmo, seg.p_vaddr, ...);

    // 完成上述PMO的实际映射
    map_pmoms(new_process_cap, requests, ...);
    // 完成其他部分的初始化（包括proc结构体）并返回
    ...
}
```

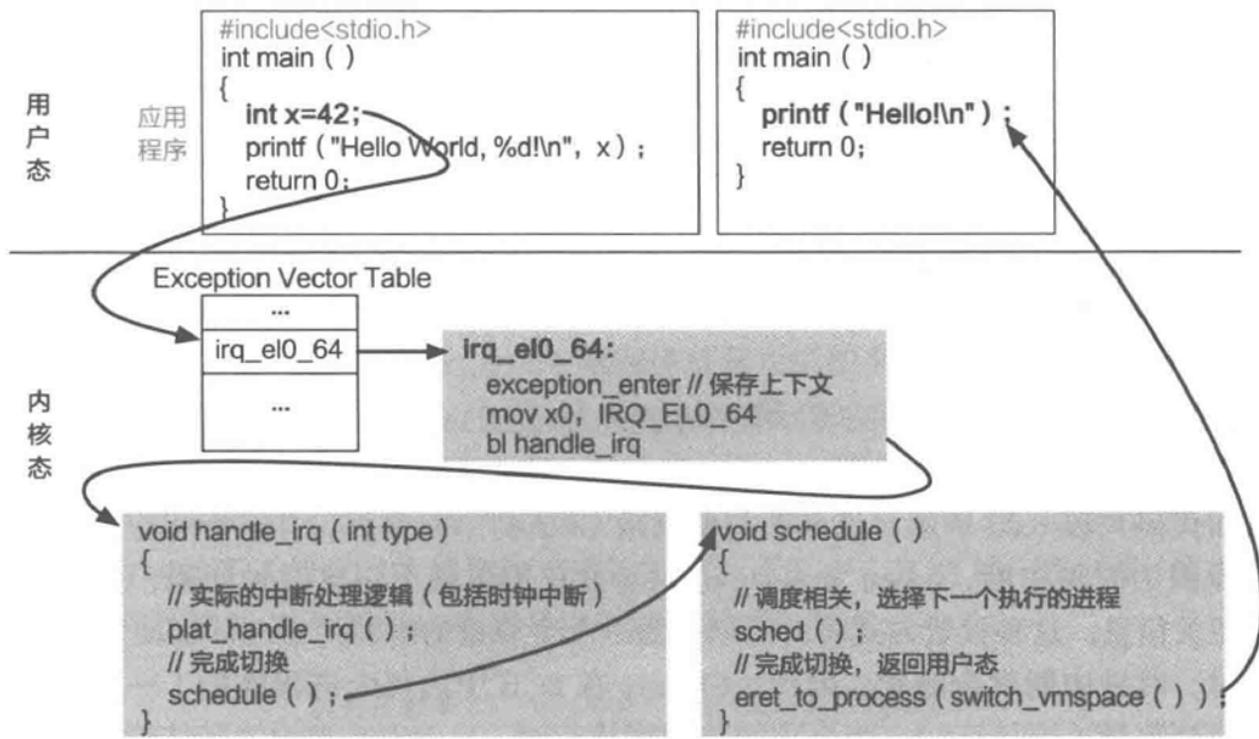
上述代码展示了ChCore中进程创建的伪代码实现，其步骤依次为：可执行文件加载、创建进程、分配用户栈、初始化用户栈、映射虚拟内存。需要注意，spawn并不是系统调用，而是处于用户态的系统服务提供的接口。下图展示了spawn函数的控制流在用户态和内核态之间切换的全过程。



从图中可以看出，当需要分配更多内核资源，或是需要对内核资源进行修改时，就要使用系统调用进入内核，其他功能都可以在用户态完成。

3、ChCore的进程切换实现

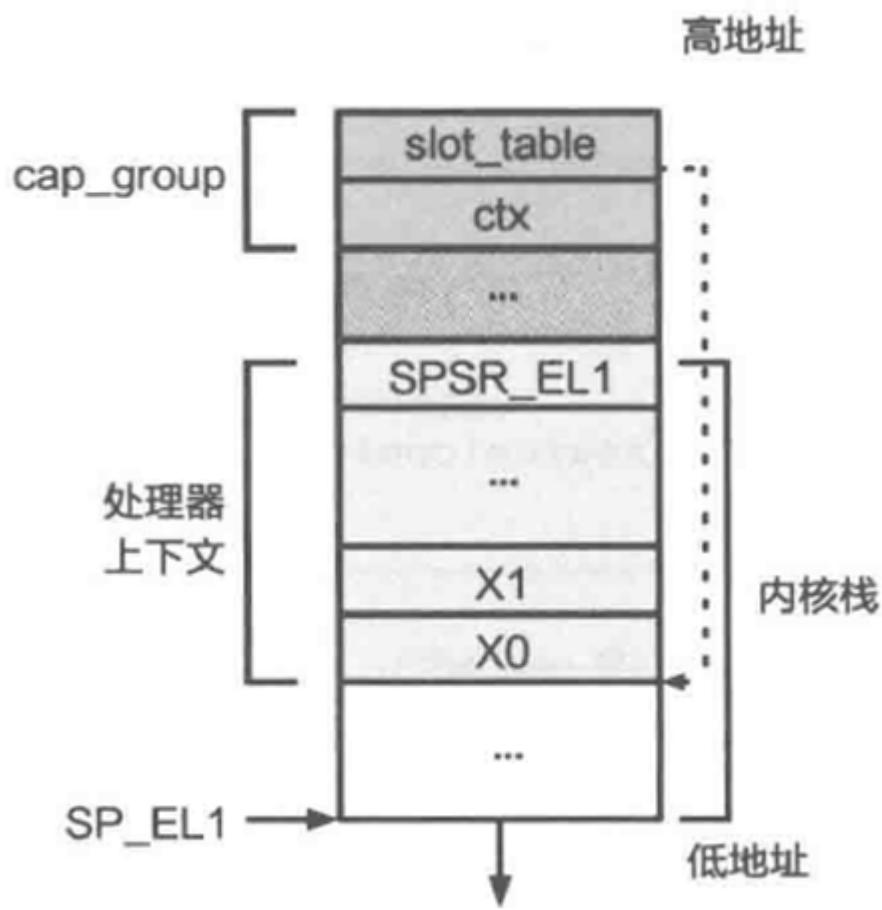
本节主要以时钟中断的处理为例详细介绍ChCore中进程被动切换的步骤。



- 中断发生时，无论当前进程在用户态执行什么代码，处理器都会保存状态并陷入内核态。

- 通过 `exception_enter` 函数保存处理器的上下文。
- 调用 `handle_irq` 函数执行时钟中断的具体逻辑。
- 调用 `schedule` 函数，其中 `sched` 选择下一个需要执行的进程。
- 通过 `eret_to_process` 切换到目标进程，并返回用户态执行。

下面我们逐个步骤详细分析。



ChCore中进程切换相关的数据结构

• 处理器上下文保存

上图所示，进程的处理器上下文位于其内核栈的底部，因此对其的保存只需将需要保存的寄存器值依次放入进程对应的内核栈中即可。代码实现 下。

```

1 .macro exception_enter
2     sub sp, sp, #ARCH_EXEC_CONT_SIZE
3     // 保存通用寄存器 (x0-x29)
4     stp x0, x1, [sp, #16 * 0]
5     stp x2, x3, [sp, #16 * 1]
6     stp x4, x5, [sp, #16 * 2]
7     ...
8     stp x28, x29, [sp, #16 * 14]
9     // 保存 x30 和上文提到的三个特殊寄存器: sp_el0, elr_el1, spsr_el1
10    mrs x21, sp_el0
11    mrs x22, elr_el1
12    mrs x23, spsr_el1
13    stp x30, x21, [sp, #16 * 15]
14    stp x22, x23, [sp, #16 * 16]
15 .endm

```

先使用 `stp` 指令将通用寄存器 `x0` 到 `x29` 的值存储到栈中，每个寄存器占用 8 个字节，偏移量以 16 为单位。再使用 `mrs` 指令将特殊寄存器 `sp_el0`、`elr_el1` 和 `spsr_el1` 的值加载到寄存器 `x21`、`x22` 和 `x23` 中，并将这些值也存储到栈中。

- **中断处理与进程调度** 操作系统通过 `exception_enter` 进入中断处理函数 `handle_irq`。

```

//ChCore时钟中断处理逻辑
void plat_handle_irq(void)
{
    u32 cpuid = 0;
    unsigned int irq_src, irq;
    // 获取当前CPU及其中断原因
    cpuid = smp_get_cpu_id();
    irq_src = get32(core_irq_source[cpuid]);
    irq = 1 << ctzl(irq_src);

    // 根据不同原因进行处理
    switch (irq) {
        case INT_SRC_TIMER3:
            // 更新下一次时钟中断间隔
            asm volatile ("msr cntv_tval_el0, %0" :: "r" (cntv_tval));
            // 更新剩余的时间片数量
            if (curr_proc->budget > 0)
                curr_proc->budget--;
            break;

        // 处理其他中断
        case ...:
    }
    return;
}

```

`handle_irq` 调用 `plat_handle_irq` 处理具体中断逻辑，获取当前 CPU 的 ID 和中断类型。若中断为时钟中断，`plat_handle_irq` 会更新下一次时钟间隔，并维护进程调度信息。系统将 CPU 时间划分为等长的时间片（time slice），每个进程的 PCB 中记录剩余时间片

(budget)。进程被调度时， budget 初始化为 DEFAULT_BUDGET；每次时钟中断触发，当前进程的 budget 减一。当 budget 减至 0 时，暂停当前进程，由调度器选择下一个进程执行。

```
int sched(void)
{
    // 果 curr_proc 不为空，且时间片还未用尽，则直接返回
    if (curr_proc && curr_proc->budget != 0)
        return 0;

    // 已经用尽，选择下个进程执行
    curr_proc = pick_next();
    // 将选中的进程执行状态变为运行
    curr_proc->exec_status = RUNNING;
    // 为下个进程配置时间片数量，然后返回
    curr_proc->budget = DEFAULT_BUDGET;
    return 0;
}
```

在这之后， handle_irq 调用 schedule 函数并进入 sched 函数中，执行调度相关逻辑。

sched 利用操作系统的调度器选取下一个需要执行的进程，作为切换的目标进程。上面展示了一种基于时间片数量的简单 sched 实现方法。首先，ChCore 需要判断 curr_proc 是否为空，这是因为在主动切换的情况下，前一个执行的进程可能已经调用了 process_exit，不可能再次被调度，所以在 process_exit 中会将 curr_proc 设为空。然后，ChCore 会读取其剩余时间片数量 budget，若时间片还有剩余则不需要切换，直接返回，否则执行调度策略，选择下一个进程，并将 curr_proc 指向该进程对应的 PCB。最后，ChCore 为下个要执行的进程配置时间片，然后返回。需要注意的是，虽然这里 curr_proc 已经被切换了，但此时内核还处于原进程的上下文中，直到下一个步骤（虚拟地址空间和内核栈切换），这种设计主要还是出于简化实现的考虑。

注意：对于陷入等待的进程，在其等待的条件满足之前，不应调度他们执行，否则只会浪费 CPU 资源。因此，ChCore 要求这些系统调用在进入 schedule 之前，将当前进程拥有的时间片设置为 0，表明放弃之后的时间片，请求内核调度下一个进程执行。

• 虚拟地址空间与内核栈切换

在调度逻辑完成后，内核会调用 switch_vmspace 函数，该函数有两个主要步骤，具体实现见下方代码片段。

第一步是切换虚拟地址空间。具体操作是从目标进程的 PCB 中获取页表基地址，将其转换为物理地址，然后设置到 TTBR0_EL1 寄存器中，从而切换到目标进程的虚拟地址空间。

第二步是返回目标进程的处理器上下文地址，这个地址将作为参数传递给 eret_to_process 函数。

```
//switch_vmspace函数的实现
void* switch_vmspace(void)
{
    // 切换虚拟地址空间：获取页表所在物理地址并设置
    set_ttbr0_el1(
        virt_to_phys(curr_proc->vmspace->pgtbl));
    // 返回处理器上下文所在地址
    return curr_proc->ctx;
}
```

接下来，`eret_to_process` 函数会将目标进程的处理器上下文地址从 `X0` 寄存器移动到 `SP_EL1` 栈寄存器，从而完成内核栈的切换。这一步是因为处理器上下文固定存储在内核栈的底部，所以切换到上下文地址实际上就是切换到了对应的内核栈。

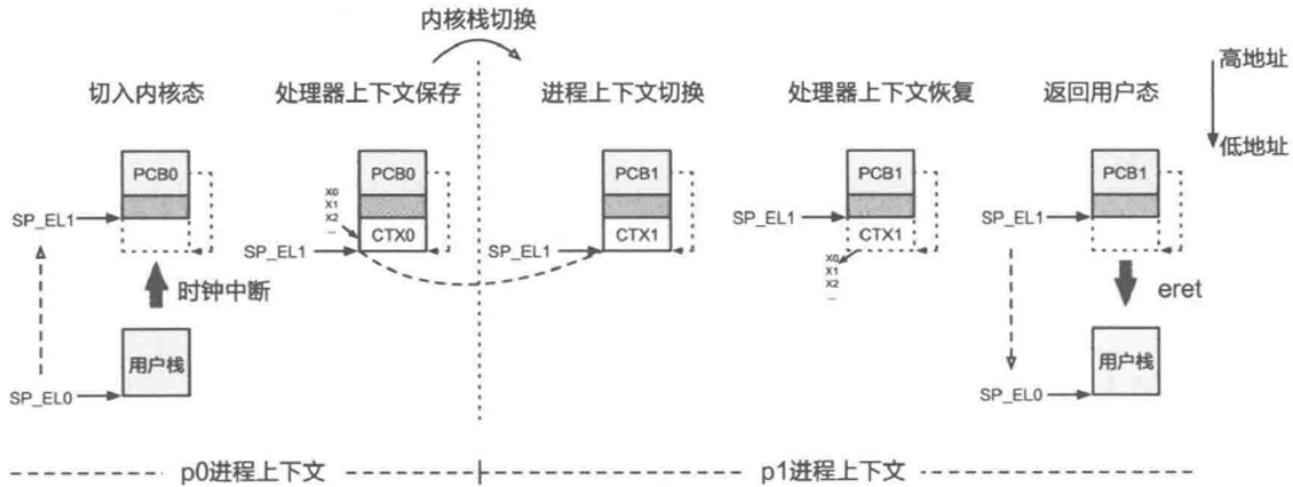
切换完成后，`SP_EL1` 栈寄存器的地址变成了目标进程的处理器上下文地址，这为恢复执行做好了准备。

```
//eret_to_process函数的实现
BEGIN_FUNC(eret_to_process)
// 函数原型: void eret_to_process(u64 sp)
// 内核栈切换
mov sp, x0
// 进程切换的剩余步骤
exception_exit
END_FUNC(eret_to_process)
```

- 处理器上下文恢复及返回用户态

```
1 .macro exception_exit
2     // 恢复 x30 和三个特殊寄存器
3     ldp x22, x23, [sp, #16 * 16]
4     ldp x30, x21, [sp, #16 * 15]
5     msr sp_el0, x21
6     msr elr_el1, x22
7     msr spsr_el1, x23
8     // 恢复通用寄存器 x0-x29
9     ldp x0, x1, [sp, #16 * 0]
10    ...
11    ldp x28, x29, [sp, #16 * 14]
12    add sp, sp, #ARCH_EXEC_CONT_SIZE
13    eret
14 .endm
```

`eret_to_process` 调用 `exception_exit` 完成进程切换的剩余步骤，`exception_exit` 的实现与 `exception_enter` 是对应的，它将内核栈中的值恢复到寄存器中，然后通过 `add` 指令将内核栈变为空，最后调用 `eret` 指令返回用户态执行。最后，我们用一张图来展示ChCore中从进程p0切换到进程p1过程中数据结构的变化。



内核栈切换完成后，ChCore只访问p1的状态，这表明内核栈切换是从p0到p1的上下文切换节点。返回用户态时，进程的内核栈总是空的，因为操作系统已经处理完中断或系统调用，无需保存栈帧。因此，进程进入内核态时，可以直接在内核栈底部保存处理器上下文，确保处理器上下文地址固定在内核栈底。

3、案例分析：Linux的进程创建

1、经典的进程创建方法：fork

```
// fork的伪代码实现
int fork(void)
{
    // 创建一个新的 PCB，用于管理新进程
    struct process *new_proc = alloc_process();
    // 虚拟内存初始化：初始化页表地址
    new_proc->vmspace->pgtbl = alloc_page();
    // 虚拟内存初始化：将当前进程（父进程）PCB 中的页表完整拷贝一份
    copy_vmspace(new_proc->vmspace, curr_proc->vmspace);
    // 内核栈初始化
    init_kern_stack(new_proc->stack);
    // 上下文初始化：将父进程 PCB 中的处理器上下文完整拷贝一份
    copy_context(new_proc->ctx, curr_proc->ctx);
    // 返回
}
```

操作系统通过 fork 和 exec 系统调用实现进程创建的职责分离：

- fork 的作用：**基于现有进程创建子进程的“骨架”，复制父进程的PCB（进程控制块），生成一个可运行的新进程，但不执行新程序。
- exec 的作用：**在 fork 的基础上，替换子进程的执行内容，载入新可执行文件，重新初

初始化PCB（虚拟地址空间、参数等），为其填充“血肉”。

两者的组合将进程的“创建框架”与“内容加载”分离，兼顾了进程管理的灵活性和效率。

但是fork的设计存在许多问题。

- **复杂性：** fork 的实现随着操作系统功能的增加变得越来越复杂。由于 fork 需要复制父进程的状态，每当操作系统增加新功能或扩展进程控制块（PCB）时，fork 的实现也需要相应修改，这增加了维护难度。POSIX 标准列出了调用 fork 时的 25 种特殊情况，这些情况需要开发者小心处理，增加了使用的不便性。
- **性能问题：** fork 的性能较差，因为它需要创建父进程的完整副本。随着进程状态的增加，fork 的性能会进一步下降。尽管写时拷贝技术可以减少内存拷贝，但对于内存需求较大的应用，建立页表中的内存映射仍然需要大量时间，导致 fork 的效率无法满足需求。
- **安全隐患：** fork 存在潜在的安全漏洞。由于 fork 创建的子进程与父进程的虚拟地址空间布局完全相同，这为攻击者提供了便利。攻击者一旦了解了父进程的虚拟地址空间布局，就可以直接攻击所有由 fork 创建的子进程，从而绕过随机性带来的安全防护。
- 除了上述问题，fork 还存在其他缺点，可扩展性差、与异构硬件不兼容、线程不安全等。因此，Linux 提出了多种替代方案，包括 vfork、spawn 和 clone 等，以解决 fork 的这些问题。

2、限定场景：vfork

```
//vfork的伪代码实现
int vfork(void)
{
    struct process *new_proc = alloc_process();
    new_proc->vm_space->pgtbl = curr_proc->vm_space->pgtbl;
    init_kern_stack(new_proc->stack);
    copy_context(new_proc->ctx, curr_proc->ctx);
    while (!exec_or_exit(new_proc))
        schedule();
}
```

vfork仍从父进程中创建子进程，但不会为子进程单独创建地址空间，子进程将与父进程共享同一地址空间。为保证正确性，vfork会使父进程在内核中等待，直到子进程调用exec创建自己独立的地址空间或者退出为止。

- 与fork相比，vfork省去了一次页表拷贝，因此其性能有明显提升。但是，vfork的使用场景相对受限，只适用于进程创建后立即使用exec的场景。

3、合二为一：posix_spawn

posix_spawn的实现方式：

- `posix_spawn` 调用 `vfork` 创建一个子进程。
- 子进程进入一个专门的准备阶段，根据调用者提供的参数进行配置。
- 子进程调用 `exec`，加载可执行文件并执行。

由于子进程调用了 `exec`，父进程也能从 `vfork` 中返回。`posix_spawn` 在创建进程后会立即调用 `exec`，因此非常适合使用 `vfork`。在较新版本的 Linux 中，`posix_spawn` 的性能明显优于 `fork` 和 `exec` 的组合，主要得益于 `vfork` 减少的页表拷贝。

```
//posix_spawn的伪代码实现
int posix_spawn(pid_t *pid, const char *path,
               const posix_spawnattr_t *attrp,
               char *const argv[],
               char *const envp[])
{
    //先执行vfork创建一个新进程
    int ret = vfork();
    if (ret == 0) {
        //子进程：在exec之前，根据参数对其进行配置
        prepare_exec(attrp, ...);
        exec(path, argv, envp);
    } else {
        //父进程：将子进程的pid设置到传入的参数中
        *pid = ret;
        return 0;
    }
}
```

但是，`posix_spawn` 不能完全替代 `fork`。虽然 `posix_spawn` 提供了 `exec` 之前的准备阶段来配置子进程，但它提供的参数表达能力有限，而 `fork+exec` 则有任意多种配置的可能。因此，`posix_spawn` 是一种比 `fork` 效率更高但灵活性较低的进程创建方式。

4、精密控制：`rfork/clone`

`clone` 的过程与 `fork` 比较相似，也是从已有进程中创建一份拷贝。但相比 `fork` 对父进程的所有结构一概进行复制，`clone` 允许应用传入参数 `flags`，指定应用不需要复制的部分。下面的代码片段中举了两个例子：应用可以设定 `CLONE_VM` 以避免复制内存，允许子进程与父进程使用相同的地址空间；同时可以设定 `CLONE_VFORK`，使父进程在内核中等待，直到子进程退出或调用 `exec`。也就是说，在设定了 `CLONE_VFORK` 和 `CLONE_VM` 之后，`clone` 的行为与 `vfork` 相似，如果不设定则和 `fork` 相似。除以上两个标志位以外，`flags` 还包含其他标志以“精密控制”进程的创建过程，从而使 `clone` 具备了各种功能，其应用场景因此比 `fork` 更加广泛（用于创建线程）。

```

//clone的伪代码实现
int clone(..., int flags, ...)
{
    // 创建一个新的 PCB，用于管理新进程
    struct process *new_proc = alloc_process();

    // 如果设置了 CLONE_VM，则直接使用父进程的页表，否则拷贝一份
    if (flags & CLONE_VM) {
        new_proc->vmspace->pgtbl = curr_proc->vmspace->pgtbl;
    } else {
        new_proc->vmspace->pgtbl = alloc_page();
        copy_vmspace(new_proc->vmspace, curr_proc->vmspace);
    }

    // 内核栈初始化
    init_kern_stack(new_proc->stack);

    // 上下文初始化：将父进程 PCB 中的上下文完整拷贝一份
    copy_context(new_proc->ctx, curr_proc->ctx);

    // 如果设置了 CLONE_VFORK，则使父进程在内核中等待
    if (flags & CLONE_VFORK) {
        while (!exec_or_exit(new_proc))
            schedule();
    }

    // 返回
}

```

二、线程

1、线程的实现：内核数据结构

由于每个线程都可以独立执行，因此操作系统需要为它们分别维护处理器上下文结构。前面介绍的 PCB 结构只能保存一个处理器上下文，因此很难满足同一进程中多个线程的需要。内核因而为线程设计了专门的数据结构——线程控制块（Thread Control Block, TCB）。除处理器上下文外，TCB 内还包含以下内容：

- **所属进程。** 为方便内核管理进程及其包含的线程，TCB 往往包含指向其所属进程 PCB 的指针。
- **内核栈。** 由于每个线程都是可独立执行的单元，操作系统为它们分别分配了内核栈。
- **线程退出状态。** 与进程类似，线程在退出时也可以使用整型变量表示其退出状态。
- **线程执行状态。** 由于线程是调度的基本单元，它也拥有与进程相似的执行状态，调度器通过查看线程的执行状态进行调度。
- ** **is_detached** 字段。**标识线程是否分离。线程退出时，果 **is_detached** 为

TRUE，则直接销毁 TCB；否则，需要等待其他线程调用 `join` 来回收资源。

下面给出一种简单的TCB结构实现。

```
// 一种简单的 TCB 结构实现
enum exec_status {NEW, READY, RUNNING, ZOMBIE, TERMINATED};

struct tcb {
    // 处理器上下文
    struct context *ctx;
    // 所属进程
    struct process *proc;
    // 内核栈
    void *stack;
    // 退出状态（用于与 exit 相关的实现）
    int exit_status;
    // 执行状态
    enum exec_status exec_status;
    // 分离相关
    bool is_detached;
};

};
```

2、线程的实现：管理接口

下面给出pthreads部分重要的线程管理接口：创建、退出、等待（合并）和分离。

```
// 线程创建
int pthread_create(pthread_t *restrict thread,
                  pthread_attr_t *restrict attr,
                  void *(*start_routine)(void *),
                  void *restrict arg);

// 线程退出
void pthread_exit(void *retval);

// 线程等待 / 合并
int pthread_join(pthread_t thread, void **	retval);

// 线程分离
int pthread_detach(pthread_t thread);
```

1、线程创建

由于线程包含的内容本来就比较少，也不需要载入新的可执行文件，因此可直接提供 `thread_create` 接口，同时完成创建和执行功能。

从下面的伪代码片段可以看出，线程创建的过程与前面介绍的 `process_create` 比较相似，但进行了简化。首先，线程创建需要分配管理线程的数据结构——TCB，类似 `process_create` 中的 PCB 创建。之后，线程需要初始化内核栈和处理器上下文（主要是 PC 和 SP）并设置参数，这部分在 `process_create` 中也有体现。除此之外，线程还需要维护与所属进程之间的关系，便于操作系统的后续管理。不过，`thread_create` 不包括虚拟地址空间初始化、可执行文件载入、用户栈分配等过程，这也使得线程的创建要比进程简单得多。

```
int thread_create(u64 stack, u64 pc, void *arg)
{
    // 创建一个新的 TCB，用于管理新线程
    struct tcb *new_thread = alloc_thread();
    // 内核栈初始化
    init_kern_stack(new_thread->stack);

    // 创建线程的处理器上下文
    new_thread->ctx = create_thread_ctx();
    // 初始化线程的处理器上下文（主要包括用户栈和 pc）
    init_thread_ctx(new_thread, stack, pc);

    // 维护进程与线程之间的关系
    new_thread->proc = curr_proc;
    add_thread(curr_proc->threads, new_thread);
    // 设置参数
    set_arg(new_thread, arg);
    // 返回
}
```

2、线程退出

```
void thread_exit(int status) {
    // 获取线程的所属进程
    struct process *curr_proc = curr_thread->proc;

    // 存储返回值
    curr_thread->status = status;

    // 销毁上下文
    destroy_thread_context(curr_thread->ctx);

    // 从进程的列表中移除当前线程
    remove_thread(curr_proc->threads, curr_thread);
    curr_proc->thread_cnt--;

    // 果进程中不再包含任何线程，则销毁 TCB 和进程
    if (curr_proc->thread_cnt == 0) {
        destroy_thread(curr_thread);
        process_exit(curr_proc);
    }

    // 果是分离线程，则直接销毁其 TCB
    if (thread->is_detached)
        destroy_thread(curr_thread);

    // 告知内核选择下个需要执行的线程
    schedule();
}
```

上述代码片段，总结线程退出实现步骤 下：

- 获取当前线程所属的进程。
- 存储线程的返回值。
- 销毁线程的处理器上下文。
- 从进程的线程列表中移除当前线程，并减少线程计数。
- 果进程中没有其他线程，销毁当前线程的 TCB 和所属进程。
- 果是分离线程，直接销毁其 TCB。
- 调用调度函数选择下一个需要执行的线程。

3、线程等待和分离

线程管理中的 `thread_join` 接口类似于进程等待功能，用于线程间的合并操作。当一个线程调用 `thread_join` 并指定需要监控的线程时，调用线程会等待目标线程退出。一旦目标线程退出，`thread_join` 返回，并将目标线程的返回值传递给调用线程。果目标线程退出但没有其他线程调用 `thread_join`，则目标线程会进入僵尸状态。`thread_join` 的功能较为单一，仅监控指定线程的退出事件，而 `wait` 和 `waitpid` 可以监控任意子进程的多种事件。
`pthread_join` 与 `thread_join` 类似，但扩展了返回值的类型。

线程分离接口 `thread_detach` 允许线程自行回收资源，适用于无法调用 `thread_join` 的场景，Web 服务器。`thread_detach` 通过设置线程的 `is_detached` 标志为 `TRUE`，使得线程退出时可以直接回收其 TCB，无需其他线程调用 `join`。果线程处于分离状态，`join` 操作对其无效。

4、线程切换

线程切换的过程与进程切换大致相同，其工作流包括：

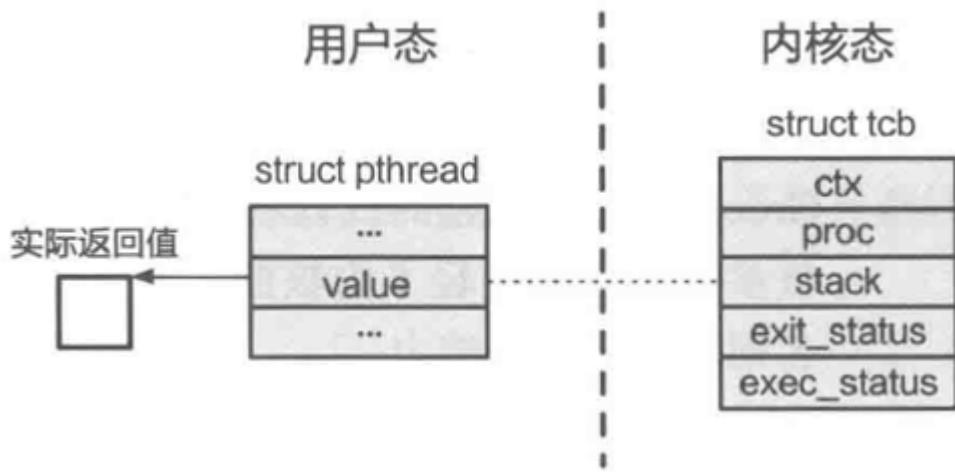
- **原线程进入内核态**：当线程需要进行切换时，首先会从用户态进入内核态。这通常是由系统调用或中断触发的。
- **保存原线程的处理器上下文**：内核会保存当前线程的处理器上下文，包括程序计数器、寄存器等信息。这些信息会被保存在当前线程的线程控制块（TCB）中。
- **切换线程上下文**：内核会选择一个新的线程来执行，并将当前线程的状态设置为就绪或等待状态。这个过程涉及到更新内核中的线程调度信息。
- **恢复目标线程的处理器上下文**：内核会从目标线程的TCB中恢复其处理器上下文，包括程序计数器、寄存器等信息。这使得目标线程能够从上次停止的地方继续执行。
- **目标线程返回用户态**：恢复上下文后，目标线程会从内核态返回用户态，继续执行其任务。

与进程切换相比，线程切换的主要区别在于：

- **虚拟地址空间**：进程切换时，果切换到一个不同的进程，通常需要切换虚拟地址空间。而线程切换时，果两个线程属于同一个进程，则不需要切换虚拟地址空间，因为它们共享相同的地址空间。
- **内核栈**：线程切换时，内核栈的切换相对简单，因为线程共享同一个进程的内核栈。而进程切换时，可能需要切换到完全不同的内核栈。

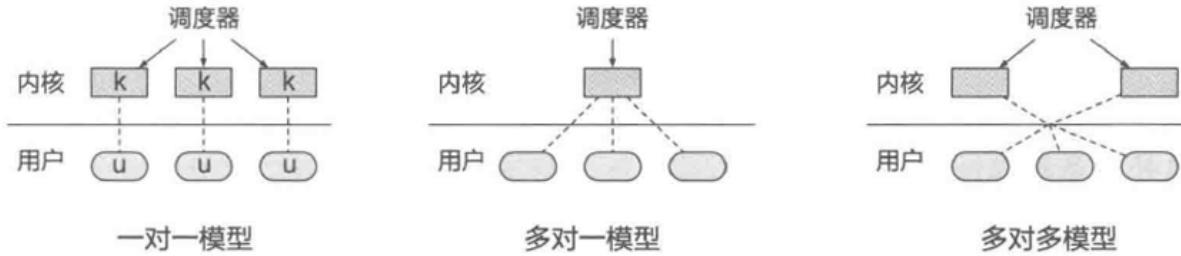
5、内核态线程与用户态线程

下图所示，`pthread` 与线程相关的数据结构实际上被拆分成两部分：内核结构（tcb）依然保存与线程相关的重要信息（所属进程和处理器上下文），这些信息不可被用户直接访问；其他信息则保存在用户态的数据结构（`pthread`）中，内核并不知晓这些数据与该线程的关系。



由于线程在内核态和用户态的执行实际上拥有较强的独立性，我们可以将其视为两类线程。**
内核态线程（Kernel Level Thread）是由内核创建并直接管理的线程，内核维护了与之对应的数据结构——TCB。而用户态线程（User Level Thread）是由用户态的线程库创建并管理的线程，其对应的数据结构保存在用户态，内核并不知晓该线程的存在，也不对其进行管理。由于内核只能管理内核态线程，其调度器只能对内核态线程进行调度，因此用户态线程如果想要执行，就需要“绑定”到相应的内核态线程才能执行。**

一般来说，多线程模型分为以下三类。



- **一对一模型：**每个用户态线程对应一个内核态线程，两者关系紧密，看起来像是同一个线程的两个部分。当内核态线程被调度执行时，其对应的用户态线程也会被执行。这种模型中，用户态线程和内核态线程一一对应，关系非常紧密。`pthread` 采用的就是这种模型。
- **多对一模型：**多个用户态线程映射到一个内核态线程。在这种模型下，进程只分配一个内核态线程，多线程环境由用户态线程库实现。这种模型减轻了内核的压力，因为线程的创建和管理都在用户态进行。
- **多对多模型 (M:N)：**允许用户自定义进程中包含的内核态线程数量，用户态线程映射到不同的内核态线程执行。这种模型既突破了多对一模型的可扩展性限制，又减少了资源开销。

三、纤程

以 `pthreads` 为代表的一对一线程模型通过绑定用户态线程与内核态线程，实现了内核功能的支持（线程同步），但其局限性随应用需求多样化逐渐显现：

- **调度优化不足**: 复杂应用可能包含大量异构线程（计算、网络、I/O等），应用自身对线程语义的理解可能优于内核调度器的通用策略。
- **创建开销大**: 线程创建需内核介入，耗时长，对短时任务（微秒级）的延迟影响显著。
- **切换性能低**: 线程切换涉及内核态操作，性能开销高，尤其影响高交互性任务（网络收发）。

因此，业界重新关注**用户态线程（Fiber）**，通过将调度权交还应用层，以降低开销并提升灵活性，适应现代高性能、高并发的需求。

1、POSIX的纤程支持：ucontext

POSIX用来支持纤程的是ucontext.h中的接口：

```
#include <ucontext.h>

int getcontext(ucontext_t *ucp);
int setcontext(const ucontext_t *ucp);
void makecontext(ucontext_t *ucp, void (*func)(), int argc, ...);
```

`getcontext` 保存当前纤程的上下文，`setcontext` 切换到另一个纤程的上下文，`makecontext` 设置纤程的上下文并从指定地址开始执行。这些上下文都在用户态保存，不涉及内核的TCB。

2、纤程切换

纤程切换是合作式的，通过 `yield` 接口主动让出CPU资源，而线程切换是抢占式的，依赖操作系统通过中断实现。

在用户态下，`setcontext` 函数用于纤程切换，性能更好，因为它只恢复通用寄存器，不涉及内核态和用户态的切换，且通过间接跳转指令完成PC切换，减少了开销。

```
ENTRY(__setcontext)
...
// 恢复被调用者保存的通用寄存器
ldp x18, x19, [x0, register_offset + 18 * SZREG]
ldp x20, x21, [x0, register_offset + 20 * SZREG]
ldp x22, x23, [x0, register_offset + 22 * SZREG]
ldp x24, x25, [x0, register_offset + 24 * SZREG]
...
// 恢复用户栈
ldr x2, [x0, sp_offset]
mov sp, x2
// 恢复浮点寄存器及参数
...
// 恢复PC并返回
ldr x16, [x0, pc_offset]
br x16
```

Last change: 2025-02-14, commit: [9fa9752](#)

说明

本文仍然基于lab内容进行拓展，包括权限和资源管理，进程调度，异常处理，系统调用。

Linux 中的权限和资源管理

在 ChCore 中，每个进程的资源和权限管理围绕 cap_group 进行，具体表现在：

1. cap_group 记录该进程拥有的 capability（能力），决定进程可以访问哪些资源（内存、设备、IPC）。
2. 每个 cap_group 只能访问自己拥有的 capability，这使得 ChCore 具备强隔离性。

接下来我们将讨论Linux中的资源和权限管理：

进程的权限和资源管理

在 Linux 内核中，每个进程都由 task_struct 结构体表示，它存储了进程的 **身份信息、权限信息、命名空间信息** 等。

代码

相关代码：**task_struct**（位于 `include/linux/sched.h`）

```
struct task_struct {  
    pid_t pid;           // 进程 ID  
    pid_t tgid;          // 线程组 ID (主线程 ID)  
  
    struct mm_struct *mm; // 进程的内存管理信息  
    struct files_struct *files; // 进程打开的文件表  
    struct cred *cred; // 进程的权限信息 (UID、GID、capabilities)  
    struct nsproxy *nsproxy; // 命名空间信息  
    struct signal_struct *signal; // 进程的信号处理信息  
    struct cgroup *cgroups; // 进程所属的 cgroup 组  
  
    struct list_head tasks; // 进程链表, 用于进程调度  
    struct sched_entity se; // 调度实体 (CFS 调度使用)  
    struct thread_struct thread; // 线程相关信息 (寄存器、堆栈等)  
    struct task_struct *parent; // 父进程指针  
    struct list_head children; // 子进程链表  
  
    struct mutex alloc_lock; // 进程资源分配锁  
};
```

- **cred**: 指向 struct cred, 存储进程的 **用户 ID (UID)**、**组 ID (GID)**、**capability 权限**等。
- **nsproxy**: 指向 struct nsproxy, 存储 **进程所属的命名空间** (pid namespace、mnt namespace 等)。
- **mm**: 存储进程的 **虚拟内存信息**, 类似 ChCore 中的 cap_group->vmspace。
- **files**: 存储进程的 **文件描述符表**。
- **cgroups** (控制组) 是 Linux 内核提供的一种机制, 用于**限制、隔离和管理**进程对系统资源 (CPU、内存、IO、网络等) 的使用。它常用于**容器、虚拟化、资源控制**等场景。

cred 机制

在 Linux 中, cred 结构体管理了 **进程的用户权限和 capabilities**, 类似于 ChCore 的 cap_group 机制。

相关代码: cred (位于 include/linux/cred.h)

```
struct cred {  
    kuid_t uid;      // 用户 ID  
    kgid_t gid;      // 组 ID  
    kuid_t euid;     // 有效用户 ID (effective UID)  
    kgid_t egid;     // 有效组 ID (effective GID)  
    kuid_t suid;     // 保存的用户 ID (saved UID)  
    kgid_t sgid;     // 保存的组 ID (saved GID)  
  
    struct group_info *group_info; // 进程所属的组信息  
  
    struct user_struct *user; // 用户的资源信息  
    struct key *session_keyring; // 进程的密钥信息  
  
    struct kernel_cap_struct cap_inheritable; // 可继承的 capability  
    struct kernel_cap_struct cap_permitted; // 允许的 capability  
    struct kernel_cap_struct cap_effective; // 当前生效的 capability  
};
```

namespace (进程的资源隔离)

Linux 采用 namespace 机制来实现 **进程间的资源隔离**，类似 ChCore 的 cap_group 机制，但粒度更细。

相关代码：nsproxy (位于 include/linux/nsproxy.h)

```
struct nsproxy {  
    struct uts_namespace *uts_ns; // 主机名/域名命名空间  
    struct ipc_namespace *ipc_ns; // 进程间通信 (IPC) 命名空间  
    struct mnt_namespace *mnt_ns; // 挂载点 (文件系统) 命名空间  
    struct pid_namespace *pid_ns; // 进程 ID 命名空间  
    struct net_namespace *net_ns; // 网络命名空间  
};
```

capability (进程的特权管理)

Linux 使用 capability 机制来控制进程的权限，类似 ChCore 的 capability 机制，但 **Linux 允许部分权限继承**。

相关代码：capability.h (位于 include/uapi/linux/capability.h)

```
#define CAP_CHOWN          0 // 修改文件所有权
#define CAP_DAC_OVERRIDE    1 // 绕过文件权限检查
#define CAP_NET_ADMIN        12 // 管理网络设备
#define CAP_SYS_ADMIN         21 // 执行系统管理操作
```

进程的 cred 结构体包含：

- **cap_permitted** → 进程 可以 使用的 capabilities
- **cap_effective** → 进程 正在 使用的 capabilities

线程的权限和资源管理

在 Linux 中，线程（Thread）实际上就是一个特殊的进程（Process）。Linux **并没有单独的线程结构**，而是通过 task_struct 来管理所有任务（进程和线程）。进程和线程的主要区别在于它们 **是否共享资源**。

Linux 通过 clone() 来创建线程或进程，参数决定它是新进程还是新线程。

相关代码：**kernel/fork.c**

```
long do_fork(unsigned long clone_flags, unsigned long stack_start,
             unsigned long stack_size, int __user *parent_tidptr,
             int __user *child_tidptr)
{
    struct task_struct *p;
    p = copy_process(clone_flags, stack_start, stack_size, parent_tidptr,
                     child_tidptr);
    return p ? task_pid_vnr(p) : -ENOMEM;
}
```

clone_flags → 决定是否共享资源：

- **CLONE_VM** → 共享地址空间（即线程）
- **CLONE_FILES** → 共享文件描述符表
- **CLONE_SIGHAND** → 共享信号处理

当 CLONE_VM 被设置时，线程 **不会** 拷贝进程的地址空间，而是与原进程共享，这样它们就能访问相同的数据，这与 ChCore 中的 cap_group 共享资源类似。

Linux 的进程调度

在 Linux 内核中，与 ChCore 线程上下文初始化及内核态切换到用户态的过程相对应的部分，

主要涉及 `fork`、`execve`、`schedule`、`switch_to`、`ret_to_user` 等函数。

1. 线程上下文初始化 (`init_thread_ctx` 对应)

在 Linux 中，新进程/线程的初始化类似于 ChCore 的 `init_thread_ctx()`，主要由 `copy_process()` 负责。

代码位置：

- `kernel/fork.c` -> `copy_process()`

核心步骤：

1. 分配 `task_struct`
2. 复制父进程的 `mm_struct` (用户地址空间)
3. 初始化寄存器状态
4. 加入调度队列

对应代码：

```
static struct task_struct *copy_process(struct clone_args *args) {  
    struct task_struct *p;  
  
    p = dup_task_struct(current); // 复制当前进程的 task_struct  
    p->mm = copy_mm(args, current); // 复制内存空间  
    p->files = copy_files(args, current); // 复制文件描述符  
  
    thread_copy(args, p); // 初始化线程上下文  
    return p;  
}
```

这个 `thread_copy()` 就相当于 `init_thread_ctx()`，它会初始化寄存器状态、设置栈指针等。

2. 调度 (`sched()` 对应)

ChCore 的 `sched()` 负责选择下一个要运行的线程，在 Linux 中，`schedule()` 负责这个工作。

代码位置：

- kernel/sched/core.c -> schedule()

```
void __sched schedule(void) {
    struct task_struct *next, *prev;

    prev = current;
    next = pick_next_task(prev); // 选择下一个要执行的进程

    context_switch(prev, next); // 切换到新的进程
}
```

3. 上下文切换 (switch_context() 对应)

ChCore 用 switch_context() 进行线程切换，Linux 采用 switch_to()。

代码位置：

- arch/arm64/kernel/process.c -> switch_to()

```
#define switch_to(prev, next, last) \
    do { \
        cpu_switch_to(prev, next); \
    } while (0)
```

其中 cpu_switch_to() 负责：

- 切换寄存器状态
 - 切换栈指针
 - 更新 task_struct
-

4. 内核态 → 用户态 (eret_to_thread() 对应)

在 ChCore，eret_to_thread() 负责从内核态切换到用户态，Linux 中类似的是 ret_to_user。

代码位置：

- arch/arm64/kernel/entry.S -> ret_to_user

```

ret_to_user:
    mov x0, sp          // 恢复用户态栈指针
    eret                // 从 EL1 返回到 EL0

```

这里 `eret` 指令的作用跟 ChCore 一样，是 ARM64 的指令，负责从 **EL1（内核态）** 切换到 **EL0（用户态）**。

总结：ChCore vs. Linux 对应关系

ChCore	Linux	作用
<code>init_thread_ctx()</code>	<code>thread_copy()</code>	线程上下文初始化
<code>sched()</code>	<code>schedule()</code>	选择下一个执行的线程
<code>switch_context()</code>	<code>switch_to()</code>	线程/进程上下文切换
<code>eret_to_thread()</code>	<code>ret_to_user</code>	从内核态切换到用户态

Linux 采用**宏内核架构**，所有的调度、资源管理都在 `task_struct` 中，而 ChCore 采用**微内核架构**，资源管理 (`cap_group`) 和调度 (`thread_t`) 分离。

Linux 的异常处理机制及其与 ChCore 的对比

在 AArch64 架构中，异常 (Exception) 是 CPU 在运行过程中，由于软件或硬件事件触发的特殊情况，可能需要操作系统内核介入处理。例：

- **同步异常 (Synchronous Exception)**：由于指令执行引起，非法指令、访问非法地址、系统调用 (`svc` 指令)。
- **异步异常 (Asynchronous Exception)**：与当前执行指令无关，IRQ (普通中断)、FIQ (快速中断)、SError (系统错误)。

Linux 内核必须实现完整的异常处理机制，而 ChCore 作为微内核，采用较精简的异常处理方式。下文对比两者在 **异常向量表**、**异常处理流程**、**系统调用**、**上下文切换** 等方面的不同。

1. 异常向量表 (Exception Vector Table)

AArch64 处理器在异常发生时，会跳转到 **异常向量表** 执行异常处理。Linux 和 ChCore 均需要初始化异常向量表。

- **Linux 的异常向量表** 位于 `arch/arm64/kernel/entry.S`，并由 `__exception_vectors` 统一管理不同异常。
- **ChCore 的异常向量表** 位于 `kernel/arch/aarch64/irq/irq_entry.S`，仅针对 EL1 进行初始化。

Linux 支持完整的 EL0-EL3 异常处理，而 ChCore 仅涉及 EL0 和 EL1。

2. 异常处理流程 (Exception Handling Flow)

Linux 异常处理流程

1. **CPU 触发异常** (同步异常或中断)。
2. 跳转到**异常向量表**，进入 `__exception_vectors` 处理。
3. 调用**异常处理函数**：
 - `el1_sync` 处理 EL1 级别同步异常。
 - `el1_irq` 处理 EL1 级别中断。
 - `el0_sync` 处理用户态 (EL0) 的异常，系统调用。
4. **解析异常原因** (`do_sync` 解析同步异常, `do_irq` 处理中断)。
5. 调用**具体处理逻辑**，`handle_syscall` 处理系统调用。
6. 恢复 CPU 状态，执行 `eret` 返回。

ChCore 异常处理流程

1. **进入异常向量表** (`vector_sync_el1h`)。
2. 调用 `handle_entry_c` 解析异常类型。
3. 针对 `svc` 指令调用 `handle_syscall` 处理系统调用。
4. 异常处理完毕后返回用户态。

ChCore 的异常处理逻辑比 Linux 更加简洁，主要是因为其微内核设计减少了内核的职责。

3. 系统调用 (Syscall)

Linux 系统调用流程

1. 用户态程序执行 `svc #0`。

2. 触发 **同步异常**，进入 `el0_sync` 处理。
3. 调用 `do_el0_svc` 解析 **系统调用号**（存储在 `x8` 寄存器）。
4. 调用相应的系统调用处理函数（`sys_write`）。
5. 处理完成后返回用户态。

```
el0_sync:
    save_context
    bl do_el0_svc
    restore_context
    eret
```

ChCore 系统调用流程

1. 用户态程序执行 `svc` 指令。
2. 进入 `vector_sync_el1h` 处理异常。
3. 调用 `handle_syscall` 解析系统调用号。
4. 调用相应的系统调用函数（`sys_write`）。
5. 处理完成后返回用户态。

```
vector_sync_el1h:
    save_context
    bl handle_syscall
    restore_context
    eret
```

对比项	Linux	ChCore
系统调用入口	<code>el0_sync</code>	<code>vector_sync_el1h</code>
处理函数	<code>do_el0_svc</code>	<code>handle_syscall</code>
系统调用解析	<code>sys_call_table</code>	<code>syscall_table</code>
异常返回	<code>ret_from_exception</code>	<code>eret_to_thread</code>

Linux 和 ChCore 的系统调用机制类似，但 Linux 由于支持更多特性（`seccomp`、安全沙盒），其实现更加复杂。

4. 上下文切换 (Context Switch)

上下文切换涉及进程/线程调度，需要保存和恢复 CPU 状态。

Linux 上下文切换

1. `context_switch` 负责保存当前进程的寄存器状态。
2. `switch_mm` 切换进程的地址空间（MMU 切换）。
3. `restore_context` 恢复新进程的 CPU 状态。
4. 继续执行新进程。

```
switch_to:
    save_context
    load_new_task
    restore_context
```

ChCore 上下文切换

1. `switch_context` 负责保存当前线程状态。
2. 切换到新的线程（`thread_ctx`）。
3. 恢复新线程状态，执行 `eret` 返回。

```
switch_context:
    save_thread_context
    load_new_thread
    eret
```

对比项	Linux	ChCore
上下文切换函数	<code>switch_to</code>	<code>switch_context</code>
进程调度	复杂（涉及 CFS、优先级等）	简单（仅切换线程）
地址空间切换	<code>switch_mm</code>	轻量化切换
恢复方式	<code>restore_context</code>	<code>eret</code> 返回

ChCore 的上下文切换较 Linux 更加轻量，因为它是微内核，仅切换 **线程上下文**，而 Linux 需要切换整个 **进程上下文**（包括 MMU、地址空间等）。

总结

特性	Linux	ChCore
异常向量表	<code>__exception_vectors</code>	<code>irq_entry.S</code>
异常处理流程	复杂，支持 EL0-EL3	仅处理 EL0、EL1

特性	Linux	ChCore
系统调用	do_el0_svc	handle_syscall
上下文切换	switch_to	switch_context
进程调度	完整进程管理	仅线程切换
异常返回	ret_from_exception	eret_to_thread

Linux 采用完整的异常处理机制，适用于通用 OS，而 ChCore 采用轻量化设计，更适合微内核架构。

Linux 与 ChCore 的系统调用机制对比

1. 系统调用概述

系统调用是操作系统提供给用户程序访问内核功能的接口，通常用于执行特权操作，文件管理、进程控制和内存操作。

- ChCore：用户程序通过 svc 指令进入内核模式。
- Linux：用户程序通常使用 syscall 指令（x86）或 svc 指令（AArch64）进入内核。

两者都需要保存当前执行状态，以便系统调用执行完毕后能正确返回用户态。

2. 进入内核模式的流程

ChCore

1. 用户态程序执行 svc 指令触发同步异常。
2. 进入异常向量表的 sync_el1h 处理代码。
3. exception_enter 负责保存当前线程的寄存器状态。
4. 切换到内核栈 switch_to_cpu_stack。
5. 根据系统调用编号调用相应的系统调用处理函数。
6. 执行完毕后，通过 exception_exit 恢复寄存器状态并返回用户态。

Linux

1. 用户态程序执行 syscall 指令（x86）或 svc 指令（AArch64）。
2. 进入异常向量表 entry.S，调用 do_syscall_64（x86）或 el0_svc（AArch64）。

3. `syscall_enter_from_user_mode` 负责保存用户态寄存器。
4. 通过 `sys_call_table` 查找并调用具体的系统调用函数。
5. `syscall_exit_to_user_mode` 负责恢复用户态寄存器并返回。

3. 上下文切换与寄存器管理

ChCore

- `exception_enter` 保存 **通用寄存器** 和 **程序状态**。
- `exception_exit` 恢复寄存器，并执行 `eret` 返回用户态。
- 需要手动管理栈切换，读取 `TPIDR_EL1` 获取 `cpu_stack`。

Linux

- `syscall_enter_from_user_mode` 负责 **用户态寄存器** 备份。
- `syscall_exit_to_user_mode` 负责 **恢复用户态寄存器**。
- 使用 `task_struct` 维护进程状态，避免手动栈切换。

4. 系统调用表与调用机制

ChCore

- `syscall_dispatcher` 解析系统调用号。
- 使用 `chcore_syscallx` 进行系统调用。
- 例 ， `sys_putstr` 用于 `printf` 实现字符输出。

Linux

- `sys_call_table` 存储系统调用函数指针。
- `sys_write` 用于 `printf` 输出，调用 `do_write` 处理文件描述符。
- 通过 `glibc` 提供 `write()`，封装 `syscall(SYS_write, ...)`。

5. 主要区别

机制	ChCore	Linux
进入方式	svc	syscall / svc
系统调用表	syscall_dispatcher	sys_call_table
进程管理	cap_group 机制	task_struct 结构体
寄存器保存	exception_enter	syscall_enter_from_user_mode
栈管理	需要手动切换	自动管理

7. 结论

- ChCore 由于是微内核架构，系统调用管理较为精简，需手动处理内核栈和寄存器切换。
- Linux 由于是宏内核，提供完整的 sys_call_table，更强大的进程管理，用户态库（glibc）隐藏了底层细节。

以上即为 Linux 与 ChCore 在系统调用机制上的核心对比。

Last change: 2025-02-14, commit: [9fa9752](#)

Last change: 2025-02-14, commit: [23a4605](#)

源码解析

这部分内容是对多核支持-多核调度-进程间通信部分的chcore源代码的详细解析，包括进程/线程管理、异常管理与系统调用三个部分。

Important

完成 Lab4 后，再阅读这部分内容。

Last change: 2025-02-14, commit: [9fa9752](#)

多核支持

本节内容负责解析ChCore关于多核支持方面的源码，包括多核原理，多核启动等部分

目录

- 知识回顾
 - CPU信息
 - 何按需启动多核
- 多核启动
 - 初始调度
 - 唤醒从核

知识回顾

支持多核，首先要有多核，这部分内容需要的知识其实在之前的源码解析中也提到过，我们先复习以前的内容

CPU信息

我们在讲解系统调用的时候曾经提到：

TPIDR_EL1 (Thread Process ID Register for EL1) 是ARM架构中一个特殊的寄存器，**用于存储当前执行线程或进程的上下文信息**。在操作系统内核中，这个寄存器经常被用来存储指向 per_cpu_data 结构的指针，该结构包含了特定于CPU的数据，比 CPU的局部变量和栈指针

那么多核，自然就会有多个这样的CPU信息块，具体数量又依硬件设备而定，以树莓派3为例：

```
#include <common/vars.h>

/* raspi3 config */
#define PLAT_CPU_NUM      4
#define PLAT_RASPI3
```

好的知道了，是4个核心。至于CPU_info，我们在讲解系统调用的内核栈切换时候也讲过：以结构体形式存在，通过指针+偏移量的形式访问结构体成员

```
#define OFFSET_CURRENT_EXEC_CTX 0
#define OFFSET_LOCAL_CPU_STACK 8
#define OFFSET_CURRENT_FPU_OWNER 16
#define OFFSET_FPU_DISABLE 24

struct per_cpu_info {
    /* The execution context of current thread */
    u64 cur_exec_ctx;

    /* Per-CPU stack */
    char *cpu_stack;

    /* struct thread *fpu_owner */
    void *fpu_owner;
    u32 fpu_disable;

    char pad[pad_to_cache_line(sizeof(u64) +
        sizeof(char *) +
        sizeof(void *) +
        sizeof(u32))];
} __attribute__((packed, aligned(64)));
```

如何按需启动多核

回忆在"机器启动"部分的内容：

主核恒为cpu 0, 在 `start.s` 之中我们比较当前cpu id和0，果是0核就跳进primary执行 `init_c`

而从核则是先循环等待bss段清零，再循环等待smp enable

primary:

```
/* Turn to el1 from other exception levels. */
bl arm64_elX_to_el1

/* Prepare stack pointer and jump to C. */
adr x0, boot_cpu_stack
add x0, x0, #INIT_STACK_SIZE
mov sp, x0

b init_c

/* Should never be here */
b .

/* Wait for bss clear */
wait_for_bss_clear:
// ...

wait_until_smp_enabled:
// ...
```

主核在 `init_c` 初始化uart之后用 `sev` 指令唤醒其他核（树莓派真机需求，在QEMU模拟器中是直接启动的），之后主核进入 `start_kernel`，初始化cpu内核栈、清空页表和TLB设置后进入 `main`

```
void init_c(void)
{
    /* Clear the bss area for the kernel image */
    clear_bss();

    /* Initialize UART before enabling MMU. */
    early_uart_init();
    uart_send_string("boot: init_c\r\n");

    wakeup_other_cores();

    /* Initialize Kernel Page Table. */
    uart_send_string("[BOOT] Install kernel page table\r\n");
    init_kernel_pt();

    /* Enable MMU. */
    el1_mmu_activate();
    uart_send_string("[BOOT] Enable el1 MMU\r\n");

    /* Call Kernel Main. */
    uart_send_string("[BOOT] Jump to kernel main\r\n");
    start_kernel(secondary_boot_flag);

    /* Never reach here */
}
```

在 `main` 中，则依次按照顺序：

- 初始化锁
- 初始化uart
- 初始化cpu info
- 初始化内存管理模块
- 初始化内核页表
- 初始化调度器
- 启动smp

此时其他核通过 `secondary_init` 初始化自己的cpu info和kernel stack之后让出cpu, 进入调度

之后由主核负责创建第一个用户态线程（即 `create_root_thread`），完毕后全部进入调度

```
/*
 * @boot_flag is boot flag addresses for smp;
 * @info is now only used as board_revision for rpi4.
 */
void main(paddr_t boot_flag, void *info)
{
    // ...
    /* Other cores are busy looping on the boot_flag, wake up those cores */
    enable_smp_cores(boot_flag);

    // ...

    smc_init();

    // ...

    /* Create initial thread here, which use the `init.bin` */
    create_root_thread();
    kinfo("[ChCore] create initial thread done\n");

    /* Leave the scheduler to do its job */
    sched();

    // ...
}
```

多核启动

初始调度

第一个问题来了：在创建第一个线程时，所有内核均已启动，而这时候并没有等待的别线程，那调度给谁呢？

答案是自己调度给自己，并且会有idle优化（空闲线程优化），这部分内容在Linux中亦有记

载：

(ref: <https://www.cnblogs.com/doitjust/p/13307378.html>)

我们以rr调度策略为例，来看看ChCore的实现（具体是哪种策略会在构建时决定，参考 main 函数的源代码）：

```
struct thread *rr_sched_choose_thread(void)
{
    unsigned int cpuid = smp_get_cpu_id();
    struct thread *thread = NULL;

    if (!list_empty(&(rr_ready_queue_meta[cpuid].queue_head))) {
        lock(&(rr_ready_queue_meta[cpuid].queue_lock));
        again:
            if (list_empty(&(rr_ready_queue_meta[cpuid].queue_head))) {
                unlock(&(rr_ready_queue_meta[cpuid].queue_lock));
                goto out;
            }
            /*
             * When the thread is just moved from another cpu and
             * the kernel stack is used by the original core, try
             * to find another thread.
             */
            if (!(thread = findRunnableThread(
                    &(rr_ready_queue_meta[cpuid].queue_head)))) {
                unlock(&(rr_ready_queue_meta[cpuid].queue_lock));
                goto out;
            }

            BUG_ON(__rr_sched_dequeue(thread));
            if (thread_is_exiting(thread) || thread_is_exited(thread)) {
                /* Thread need to exit. Set the state to TE_EXITED */
                thread_set_exited(thread);
                goto again;
            }
            unlock(&(rr_ready_queue_meta[cpuid].queue_lock));
            return thread;
    }
out:
    return &idle_threads[cpuid];
}
```

注意到在等待队列为空的时候，会来到标签 out，返回一个 idle_thread，即空闲线程

它的ctx会在初始化的时候被放在 idle_thread_routine 处，这个函数是体系结构相关的，旨在防止cpu空转降低功耗

```
/* Arch-dependent func declarations, which are defined in assembly files */
extern void idle_thread_routine(void);
```

进一步阅读汇编代码，这个函数在arm架构中是wfi指令，让cpu进入低功耗状态，在某些版本

中的实现是关闭几乎所有的时钟

```
BEGIN_FUNC(idle_thread_routine)
idle:    wfi
        b  idle
END_FUNC(idle_thread_routine)
```

唤醒从核

在"机器启动"栏目，我们只是简单的讲解了主核通过设置 `secondary_boot_flag` 来唤醒处于轮询状态的从核，这里我们细致分析这一过程：

首先看主核 `main` 函数的参数：

```
void main(paddr_t boot_flag, void *info)
```

这里的 `boot_flag` 即是之前在 `init_c` 中传入的 `secondary_boot_flag`

再来看看 `secondary_boot_flag` 自己是什么东西：

```
// kernel/arch/aarch64/boot/raspi3/init/init_c.c
/*
 * Initialize these variables in order to make them not in .bss section.
 * So, they will have concrete initial value even on real machine.
 *
 * Non-primary CPUs will spin until they see the secondary_boot_flag becomes
 * non-zero which is set in kernel (see enable_smp_cores).
 *
 * The secondary_boot_flag is initialized as {NOT_BSS, 0, 0, ...}.
 */
#define NOT_BSS (0xBEEFUL)
long secondary_boot_flag[PLAT_CPU_NUMBER] = {NOT_BSS}; // 0xBEEFUL
volatile u64 clear_bss_flag = NOT_BSS;
```

`secondary_boot_flag` 实际上是作为kernel .data 段的一个地址被加载的

毫无疑问，这时候内核页表都还没初始化，那它本身指的必然是物理地址

而它在什么时候发挥作用？是在 `main` 中：

```
/*
 * @boot_flag is boot flag addresses for smp;
 * @info is now only used as board_revision for rpi4.
 */
void main(paddr_t boot_flag, void *info)
{
// ...

/* Other cores are busy looping on the boot_flag, wake up those cores */
enable_smp_cores(boot_flag);

// ...

/* Create initial thread here, which use the `init.bin` */
create_root_thread();

/* Leave the scheduler to do its job */
sched();

// ...
}
```

从 `main` 函数的签名以及 `enable_smp_cores` 函数的实现也可以看出来，我们需要先进行一次转换得到虚拟地址，再进行后续的操作：

```
void enable_smp_cores(paddr_t boot_flag)
{
    int i = 0;
    long *secondary_boot_flag;

    /* 设置当前CPU（主核）的状态为运行状态 */
    cpu_status[smp_get_cpu_id()] = cpu_run;

    /* 将启动标志数组的物理地址转换为虚拟地址 */
    secondary_boot_flag = (long *)phys_to_virt(boot_flag);

    /* 遍历所有CPU核心 */
    for (i = 0; i < PLAT_CPU_NUM; i++) {
        /* 设置当前CPU的启动标志
         * 这个标志会被对应的CPU核心检测到，从而开始其启动流程
         */
        secondary_boot_flag[i] = 1;

        /* 刷新数据缓存区域
         * 确保启动标志的更新对所有CPU核心可见
         * 防止缓存一致性问题导致其他核心看不到更新
         */
        flush_dcache_area((u64) secondary_boot_flag,
                           (u64) sizeof(u64) * PLAT_CPU_NUM);

        /* 数据同步屏障
         * 确保在继续执行前，所有的内存操作都已完成
         * 这是多核系统中保证内存一致性的关键步骤
         */
        asm volatile ("dsb sy");

        /* 等待目标CPU改变其状态
         * 通过轮询检查cpu_status数组来确认CPU已经启动
         * cpu_hang表示CPU尚未启动完成
         * 当CPU完成初始化后，会将其状态改为非cpu_hang值
         */
        while (cpu_status[i] == cpu_hang)
            ;

        /* 打印CPU激活信息，用于调试和状态跟踪 */
        kinfo("CPU %d is active\n", i);
    }

    /* 所有CPU启动完成
     * 打印总结信息，标志着多核初始化的完成
     */
    kinfo("All %d CPUs are active\n", PLAT_CPU_NUM);

    /* 初始化处理器间中断（IPI）数据
     * 这是多核系统中进行核间通信的必要步骤
     * 必须在所有CPU都启动完成后才能初始化
     */
}
```

```
    init_ipi_data();  
}
```

为什么这时候又需要转换为虚拟地址？因为这个函数是在主核中被调用的，主核已经完成初始化页表的工作了，自然需要虚拟地址

我在主核改的flag，你从核又怎么看得见？通过刷新数据缓存，即 `flush_dcache_area` 函数，而这又和硬件设计联系在一起了

```
BEGIN_FUNC(flush_dcache_area)  
    dcache_by_line_op civac, sy, x0, x1, x2, x3  
    ret  
END_FUNC(flush_dcache_area)
```

至此，多核支持部分源码解析到此结束

Last change: 2025-02-14, commit: [9fa9752](#)

目录

- 多核调度
- 调度API
 - 调度器数据结构
 - 调度器API实现——以RR为例
 - 初始化
 - 入队操作
 - 出队操作
 - rr策略实现
- 协作式调度
- 抢占式调度
 - 物理时钟初始化
 - 物理时钟中断与抢占

多核调度

上一部分已经讲解了ChCore对多核支持的实现与多核的启动具体方式逻辑，本部分则来具体讲解ChCore对多核调度的实现——这是我们在ICS中也学过的内容，现在让我们把知识与ChCore的具体实现结合起来看看吧！

调度API

调度器数据结构

类比我们在 `printf` 的系统调用链中提到的 `fd_ops` 结构体，与调度相关的操作我们也有特定的数据结构 `sched_ops` 来表示：

```
// /kernel/include/sched.h

struct sched_ops {
    /* 调度器初始化函数
     * 在系统启动时调用，用于：
     * - 初始化调度器数据结构
     * - 设置初始调度参数
     * - 准备调度器运行环境
     * 返回值：成功返回0，失败返回错误码
     */
    int (*sched_init)(void);
}

/* 核心调度函数
 * 选择下一个要运行的线程，在以下情况下调用：
 * - 当前线程主动放弃CPU
 * - 时间片用完
 * - 系统调用后
 * 返回值：成功返回0，失败返回错误码
 */
int (*sched)(void);

/* 周期性调度函数
 * 在每个时钟中断时被调用，用于：
 * - 更新线程时间片
 * - 检查是否需要重新调度
 * - 维护调度统计信息
 * 返回值：如果需要重新调度返回1，否则返回0
 */
int (*sched_periodic)(void);

/* 将线程加入调度队列
 * @param thread: 要加入队列的线程
 * 使用场景：
 * - 线程创建时
 * - 线程从阻塞状态恢复
 * - 线程被唤醒时
 * 返回值：成功返回0，失败返回错误码
 */
int (*sched_enqueue)(struct thread *thread);

/* 将线程从调度队列中移除
 * @param thread: 要移除的线程
 * 使用场景：
 * - 线程退出时
 * - 线程进入阻塞状态
 * - 线程被挂起时
 * 返回值：成功返回0，失败返回错误码
 */
int (*sched_dequeue)(struct thread *thread);
```

```
/* 调试工具：显示调度器状态
 * 用于调试和监控：
 * - 打印当前调度队列状态
 * - 显示线程运行统计信息
 * - 输出调度器性能指标
 * 无返回值
 */
void (*sched_top)(void);
};
```

它本质上是一个函数指针的集合，里面囊括了我们实现一个调度器所需要的基本函数操作：

- `sched_init` : 初始化调度器
- `sched` : 进行一次调度。即将正在运行的线程放回就绪队列，然后在就绪队列中选择下一个需要执行的线程返回
- `sched_enqueue` : 将新线程添加到调度器的就绪队列中
- `sched_dequeue` : 从调度器的就绪队列中取出一个线程
- `sched_top` : 用于debug,打印当前所有核心上的运行线程以及等待线程的函数

那么不同的调度策略又是如何实现呢？只需要在编译ChCore的时候指定相应的config即可，这会在初始化该结构体的时候体现出来

调度器API实现——以RR为例

初始化

我们首先会在内核初始化的 `main` 函数中调用一个统一的 `sched_init` 函数：

```

void main(paddr_t boot_flag, void *info)
{
    // ...

    /* Init scheduler with specified policy */
#if defined(CHCORE_KERNEL_SCHED_PBFIFO)
    sched_init(&pbfifo);
#elif defined(CHCORE_KERNEL_RT) || defined(CHCORE_OPENTRUSTEE)
    sched_init(&pbrr);
#else
    sched_init(&rr);
#endif
    kinfo("[ChCore] sched init finished\n");

    // ...
}

```

这里传入的调度策略即为一个个 `sched_ops` 结构体的实例，我们将在这里的 `sched_init`（不是结构体里的）将全局变量 `cur_sched_ops` 设置为传入的引用

```

/* Provided Scheduling Policies */
extern struct sched_ops pbrr; /* Priority Based Round Robin */
extern struct sched_ops pbfifo; /* Priority Based FIFO */
extern struct sched_ops rr; /* Simple Round Robin */

/* Chosen Scheduling Policies */
extern struct sched_ops *cur_sched_ops;

// /kernel/sched/sched.c
int sched_init(struct sched_ops *sched_ops)
{
    BUG_ON(sched_ops == NULL);

    init_idle_threads();
    init_current_threads();
    init_resched_bitmaps();

    cur_sched_ops = sched_ops;
    cur_sched_ops->sched_init();

    return 0;
}

// /kernel/sched/policy_rr.c
struct sched_ops rr = { .sched_init = rr_sched_init,
                      .sched = rr_sched,
                      .sched_periodic = rr_sched,
                      .sched_enqueue = rr_sched_enqueue,
                      .sched_dequeue = rr_sched_dequeue,
                      .sched_top = rr_top};

```

所以真正初始化的地方是依据具体的策略而定的，我们这里以rr为例，来到 `rr_sched_init`：

```
int rr_sched_init(void)
{
    int i = 0;

    /* Initialize global variables */
    for (i = 0; i < PLAT_CPU_NUM; i++) {
        // 初始化队列链表头
        init_list_head(&(rr_ready_queue_meta[i].queue_head));
        // 初始化锁
        lock_init(&(rr_ready_queue_meta[i].queue_lock));
        // 初始化队列长度为0
        rr_ready_queue_meta[i].queue_len = 0;
    }

    return 0;
}
```

整体很简单，就是为每个核心初始化其相应的调度队列和锁，其中又涉及到 `queue_meta` 数据结构：

```
/* 就绪队列元数据结构 */
struct queue_meta {
    struct list_head queue_head;      // 就绪队列链表头
    unsigned int queue_len;           // 队列长度
    struct lock queue_lock;          // 队列锁
    char pad[pad_to_cache_line(sizeof(unsigned int)
                               + sizeof(struct list_head)
                               + sizeof(struct lock))]; // 缓存行对齐填充
};

/* 每个CPU的就绪队列数组 */
struct queue_meta rr_ready_queue_meta[PLAT_CPU_NUM];
```

队列依旧使用双向链表实现，初始化之后还需要把队列长度 `queue_len` 设置为0，结束初始化

入队操作

我们同样来看rr策略下对入队操作的实现：

```
/*
 * Sched_enqueue
 * Put `thread` at the end of ready queue of assigned `affinity` and `prio`.
 * If affinity = NO_AFF, assign the core to the current cpu.
 * If the thread is IDLE thread, do nothing!
 */
int rr_sched_enqueue(struct thread *thread)
{
    BUG_ON(!thread);
    BUG_ON(!thread->thread_ctx);
    if (thread->thread_ctx->type == TYPE_IDLE)
        return 0;

    int cpubind = 0;
    unsigned int cpuid = 0;
    int ret = 0;

    cpubind = get_cpubind(thread);
    cpuid = cpuid == NO_AFF ? rr_sched_choose_cpu() : cpubind;

    if (unlikely(thread->thread_ctx->sc->prio > MAX_PRIO))
        return -EINVAL;

    if (unlikely(cpuid >= PLAT_CPU_NUM)) {
        return -EINVAL;
    }

    lock(&(rr_ready_queue_meta[cpuid].queue_lock));
    ret = __rr_sched_enqueue(thread, cpuid);
    unlock(&(rr_ready_queue_meta[cpuid].queue_lock));
    return ret;
}

int __rr_sched_enqueue(struct thread *thread, int cpuid)
{
    if (thread->thread_ctx->type == TYPE_IDLE) {
        return 0;
    }

    /* Already in the ready queue */
    if (thread_is_ts_ready(thread)) {
        return -EINVAL;
    }

    thread->thread_ctx->cpuid = cpuid;
    thread_set_ts_ready(thread);
    obj_ref(thread);
    list_append(&(thread->ready_queue_node),
               &(rr_ready_queue_meta[cpuid].queue_head));
    rr_ready_queue_meta[cpuid].queue_len++;

    return 0;
}
```

发现这两个函数其实只是一个框架，包括的是各种corner case以及参数检查等。考虑到

ChCore的多核特性，选择入队的核心才是关键所在，不过在这里我们并没有采用很复杂的算法：

```
/* A simple load balance when enqueue threads */
unsigned int rr_sched_choose_cpu(void)
{
    unsigned int i, cpuid, min_rr_len, local_cpuid, queue_len;

    local_cpuid = smp_get_cpu_id();
    min_rr_len = rr_ready_queue_meta[local_cpuid].queue_len;

    if (min_rr_len <= LOADBALANCE_THRESHOLD) {
        return local_cpuid;
    }

    /* Find the cpu with the shortest ready queue */
    cpuid = local_cpuid;
    for (i = 0; i < PLAT_CPU_NUM; i++) {
        if (i == local_cpuid) {
            continue;
        }

        queue_len =
            rr_ready_queue_meta[i].queue_len + MIGRATE_THRESHOLD;
        if (queue_len < min_rr_len) {
            min_rr_len = queue_len;
            cpuid = i;
        }
    }

    return cpuid;
}
```

这里用的选择算法是简单负载均衡，也即：

- 若当前CPU队列长度低于负载阈值，则选择当前CPU，可提高缓存亲和性
- 否则选择最短队列的CPU，这里还要加上迁移开销，具体数量参考宏定义

```
/* The config can be tuned. */
#define LOADBALANCE_THRESHOLD 5
#define MIGRATE_THRESHOLD      5
```

加锁方面也是简单粗暴地直接加大锁，暂时没有什么优化

出队操作

出队不需要选择CPU，所以直接出就行：

```
/*
 * remove @thread from its current residual ready queue
 */
int rr_sched_dequeue(struct thread *thread)
{
    BUG_ON(!thread);
    BUG_ON(!thread->thread_ctx);
    /* IDLE thread will **not** be in any ready queue */
    BUG_ON(thread->thread_ctx->type == TYPE_IDLE);

    unsigned int cpuid = 0;
    int ret = 0;

    cpuid = thread->thread_ctx->cpuid;
    lock(&(rr_ready_queue_meta[cpuid].queue_lock));
    ret = __rr_sched_dequeue(thread);
    unlock(&(rr_ready_queue_meta[cpuid].queue_lock));
    return ret;
}

/* dequeue w/o lock */
int __rr_sched_dequeue(struct thread *thread)
{
    /* IDLE thread will **not** be in any ready queue */
    BUG_ON(thread->thread_ctx->type == TYPE_IDLE);

    if (!thread_is_ts_ready(thread)) {
        kwarn("%s: thread state is %d\n",
              __func__,
              thread->thread_ctx->state);
        return -EINVAL;
    }

    list_del(&(thread->ready_queue_node));
    rr_ready_queue_meta[thread->thread_ctx->cpuid].queue_len--;
    obj_put(thread);
    return 0;
}
```

注意各种corner case的判断，参数检查，以及更新对象计数等必须操作

rr策略实现

上面所述皆为对调度队列的基本操作，现在我们再来看看对Round Robin轮转策略具体实现：

```
/*
 * Schedule a thread to execute.
 * current_thread can be NULL, or the state is TS_RUNNING or
 * TS_WAITING/TS_BLOCKING. This function will suspend current running thread,
if
 * any, and schedule another thread from
 * `(rr_ready_queue_meta[cpuid].queue_head)` .
 * ***the following text might be outdated***
 * 1. Choose an appropriate thread through calling *chooseThread* (Simple
 * Priority-Based Policy)
 * 2. Update current running thread and left the caller to restore the
executing
 * context
 */
int rr_sched(void)
{
    /* WITH IRQ Disabled */
    struct thread *old = current_thread;
    struct thread *new = 0;

    if (old) {
        BUG_ON(!old->thread_ctx);

        /* old thread may pass its scheduling context to others. */
        if (old->thread_ctx->type != TYPE_SHADOW
            && old->thread_ctx->type != TYPE_REGISTER) {
            BUG_ON(!old->thread_ctx->sc);
        }

        /* Set TE_EXITING after check won't cause any trouble, the
         * thread will be recycle afterwards. Just a fast path. */
        /* Check whether the thread is going to exit */
        if (thread_is_exiting(old)) {
            /* Set the state to TE_EXITED */
            thread_set_exited(old);
        }

        /* check old state */
        if (!thread_is_exited(old)) {
            if (thread_is_ts_running(old)) {
                /* A thread without SC should not be
TS_RUNNING.

                */
                BUG_ON(!old->thread_ctx->sc);
                if (old->thread_ctx->sc->budget != 0
                    && !thread_is_suspend(old)) {
                    switch_to_thread(old);
                    return 0; /* no schedule needed */
                }
                rr_sched_refill_budget(old, DEFAULT_BUDGET);
                BUG_ON(rr_sched_enqueue(old) != 0);
            } else if (!thread_is_ts_blocking(old)
                && !thread_is_ts_waiting(old)) {
                kinfo("thread state: %d\n",
                      old->thread_ctx->state);
                BUG_ON(1);
            }
        }
    }
}
```

```

        }
    }

    BUG_ON(!(new = rr_sched_choose_thread()));
    switch_to_thread(new);

    return 0;
}

```

整体上是根据当前线程是否存在做的判断：

- 若当前线程不存在，则直接快进到choose一个新的
- 若当前线程存在，则继续做进一步检查与操作：
 - 调度上下文检查：除了影子线程和寄存器线程外都必须有调度上下文
 - 若当前线程状态是exited，则给它收尸（`thread_set_exited`）
 - 否则进入时间片（`sc->budget`）检查（运行状态的线程）：
 - 若时间片没用尽，线程也未被挂起，则继续运行
 - 若时间片已经用尽，则重新设置其时间片，并再次入队
 - 对非运行状态的线程作异常处理
- 最后的 `switch_to_thread` 是内核态的切换线程函数，它只负责 `current_thread` 等变量，并没有设置完整的上下文切换，所以需要搭配其他函数来完成(返回用户态的实例在 `sched_to_thread`，内部比较复杂，可能跨核调度，此时需要通过ipi（体系结构特定的处理器间中断）来等待目标cpu核处理好当前的中断等事件释放内核栈，再进行体系结构特定的上下文切换（寄存器的save/load）和用户态返回)

其中调度上下文是 下数据结构：

```

typedef struct sched_context {
    unsigned int budget; // 时间片预算
    unsigned int prio; // 线程优先级
} sched_ctx_t;

```

最后再来看看rr策略是如何选择新的可执行的线程吧，这个函数比较长，所以添加了相关的注释：

```
struct thread *rr_sched_choose_thread(void)
{
    unsigned int cpuid = smp_get_cpu_id();
    struct thread *thread = NULL;

    /* 检查当前CPU的就绪队列是否为空 */
    if (!list_empty(&(rr_ready_queue_meta[cpuid].queue_head))) {
        /* 获取队列锁，保护并发访问 */
        lock(&(rr_ready_queue_meta[cpuid].queue_lock));

again:   /* 重试标签：处理无效线程的情况 */
        /* 双重检查队列是否为空（可能在获取锁的过程中被清空） */
        if (list_empty(&(rr_ready_queue_meta[cpuid].queue_head))) {
            unlock(&(rr_ready_queue_meta[cpuid].queue_lock));
            goto out; // 队列为空，返回空闲线程
        }

        /* 尝试找到一个可运行的线程
         * 处理内核栈被其他核心使用的情况
         */
        if (!(thread = find_runnable_thread(
                &(rr_ready_queue_meta[cpuid].queue_head)))) {
            unlock(&(rr_ready_queue_meta[cpuid].queue_lock));
            goto out; // 没有可运行线程，返回空闲线程
        }

        /* 从就绪队列中移除选中的线程 */
        BUG_ON(__rr_sched_dequeue(thread));
    }

    /* 处理退出状态的线程 */
    if (thread_is_exiting(thread) || thread_is_exited(thread)) {
        thread_set_exited(thread);
        goto again; // 重新选择线程
    }

    unlock(&(rr_ready_queue_meta[cpuid].queue_lock));
    return thread; // 返回选中的线程
}

out:    /* 就绪队列为空，返回对应CPU的空闲线程 */
return &idle_threads[cpuid];
}

struct thread *find_runnable_thread(struct list_head *thread_list)
{
    struct thread *thread;

    /* 遍历就绪队列中的所有线程 */
    for_each_in_list (
        thread, struct thread, ready_queue_node, thread_list) {
        /* 检查线程是否满足运行条件：
         * 1. 未被挂起 (!thread_is_suspend)
         * 2. 内核栈可用 (KS_FREE)
         */
}
```

```

    * 或者是当前线程 (current_thread)
    */
if (!thread_is_suspend(thread)
    && (thread->thread_ctx->kernel_stack_state == KS_FREE
        || thread == current_thread)) {
    return thread;
}
return NULL; // 没有找到合适的线程
}

```

注意一共有两次判空，第一次判空后才拿的锁，以减少轻工作状态下的锁竞争

拿锁后遍历自己核心上的可执行队列，找一个当前状态为free的或者就是当前的thread，如果没找到则返回idle空闲线程

协作式调度

协作式调度，即需要用户态程序主动配合"让出CPU"，反映到系统调用上即为 sys_yield

```

/* SYSCALL functions */
void sys_yield(void)
{
    current_thread->thread_ctx->sc->budget = 0;
    sched();
    eret_to_thread(switch_context());
}

```

这里的 sched 即为rr策略下的调度函数，是用结构体模拟的重载：

```

static inline int sched(void)
{
    return cur_sched_ops->sched();
}

```

注意 sched 里的切换线程是不完整的，因此还需要切换上下文并 eret ，我们来看源码：

看函数开头前的注释也可以明白其用法

```
/*
 * This function is used after current_thread is set (a new thread needs to
be
 * scheduled).
 *
 * Switch context between current_thread and current_thread->prev_thread:
 * including: vmspace, fpu, tls, ...
 *
 * Return the context pointer which should be set to stack pointer register.
 */
vaddr_t switch_context(void)
{
    struct thread *target_thread;      // 目标线程
    struct thread_ctx *target_ctx;     // 目标线程上下文
    struct thread *prev_thread;        // 前一个线程

    /* 1. 基本检查 */
    target_thread = current_thread;
    if (!target_thread || !target_thread->thread_ctx) {
        kwarn("%s no thread_ctx", __func__);
        return 0;
    }

    target_ctx = target_thread->thread_ctx;
    prev_thread = target_thread->prev_thread;

    /* 2. 特殊情况: 切换到自己 */
    if (prev_thread == THREAD_ITSELF)
        return (vaddr_t)target_ctx;

    /* 3. FPU状态处理 */
#ifndef FPU_SAVING_MODE == EAGER_FPU_MODE
    /* 积极模式: 立即保存和恢复FPU状态 */
    save_fpu_state(prev_thread);
    restore_fpu_state(target_thread);
#else
    /* 懒惰模式: 仅在必要时处理FPU */
    if (target_thread->thread_ctx->type > TYPE_KERNEL)
        disable_fpu_usage();
#endif

    /* 4. 切换线程本地存储(TLS) */
    switch_tls_info(prev_thread, target_thread);

    /* 5. 虚拟内存空间切换 */
#ifndef CHCORE_KERNEL_TEST
    /* 正常情况下的虚拟内存处理 */
    BUG_ON(!target_thread->vmspace);
    /* 记录线程运行的CPU: 用于TLB维护 */
    record_history_cpu(target_thread->vmspace, smp_get_cpu_id());
    /* 果需要, 切换虚拟内存空间 */
    if ((!prev_thread) || (prev_thread->vmspace != target_thread-
>vmspace))
        switch_thread_vmspace_to(target_thread);

```

```

#else
    /* 测试模式下的特殊处理 */
    if (target_thread->thread_ctx->type != TYPE_TESTS) {
        BUG_ON(!target_thread->vmspace);
        record_history_cpu(target_thread->vmspace, smp_get_cpu_id());
        switch_thread_vmspace_to(target_thread);
    }
#endif

    /* 6. 架构相关的上下文切换 */
    arch_switch_context(target_thread);

    /* 7. 返回目标上下文地址 */
    return (vaddr_t)target_ctx;
}

void arch_switch_context(struct thread *target)
{
    struct per_cpu_info *info;

    info = get_per_cpu_info();

    info->cur_exec_ctx = (u64)target->thread_ctx; // 设置当前CPU信息
}

```

整体来看需要切换的东西还是很多的：

- vmspace：这个结构在thread里面(还记得这的核心数据机构是一个rbtree的root吗，回看内存管理)，但是需要切换页表（页表的地址也在vmspace之中维护）
- tls：(thread local storage，在arm架构的典型实现之中是TPIDR_EL0寄存器，它存着一个线程特定的标识符）
 - OS其实不知道这里面存的是什么东西，他只是把这个寄存器当成线程特定的标识符，并在自己的线程实现之中维护而已。至于语言层面的tls 何实现，那是编译器开发者或者库开发者的事情，例 存一个特定的空间的指针（例 小的在栈上，大的在堆上）
 - arm compiler的支持文档 <https://developer.arm.com/documentation/dui0205/g/arm-compiler-reference/compiler-specific-features/thread-local-storage>
 - 对tls的一些讨论 <https://forum.osdev.org/viewtopic.php?t=36597>
- fpu 相关
- 其他想要添加的机制，例 保存和清理TLB的一些数据 (history) 等
- 把切换的线程相关的寄存器设置到cpu上，见 arch_switch_context

抢占式调度

协作式调度需要用户态程序自己“体面地”让出CPU，那它要不体面怎么办？我们就来帮它体面，这便是抢占式调度，典型应用场景即 Lab文档所述：

ChCore启动的第一个用户态线程（执行 `user/system-services/system-servers/procmgr/procmgr.c` 的 `main` 函数）将创建一个“自旋线程”，该线程在获得CPU核心的控制权后便会执行无限循环，进而导致无论是该程序的主线程还是ChCore内核都无法重新获得CPU核心的控制权。就保护系统免受用户程序中的错误或恶意代码影响而言，这一情况显然并不理想，任何用户应用线程均可以像该“自旋线程”一样，通过进入无限循环来永久“霸占”整个CPU核心

而抢占式首先要支持的就是时钟中断。时钟中断的支持实际上和其他外设也没什么区别，抽象起来就是对寄存器的读和写，以及配置真正连接cpu引脚的发信号时间等硬件相关的操作

物理时钟初始化

回归主线，Lab文档也说了相关寄存器的信息：

我们需要处理的系统寄存器 下([Refer](#)):

- `CNTPCT_EL0`: 它的值代表了当前的 system count。
- `CNTFRQ_EL0`: 它的值代表了物理时钟运行的频率，即每秒钟 system count 会增加多少。
- `CNTP_CVAL_EL0`: 是一个64位寄存器，操作系统可以向该寄存器写入一个值，当 system count 达到或超过该值时，物理时钟会触发中断。
- `CNTP_TVAL_EL0`: 是一个32位寄存器，操作系统可以写入 TVAL，处理器会在内部读取当前的系统计数，加上写入的值，然后填充 CVAL。
- `CNTP_CTL_EL0`: 物理时钟的控制寄存器，第0位ENABLE控制时钟是否开启，1代表 enable，0代表 disable；第1位IMASK代表是否屏蔽时钟中断，0代表不屏蔽，1代表 屏蔽。

那么其初始化函数便好理解了：

```

void plat_timer_init(void)
{
    u64 count_down = 0;
    u64 timer_ctl = 0;
    u32 cpuid = smp_get_cpu_id();

    /* 1. 读取系统计数器和频率*/
    asm volatile ("mrs %0, cntpct_el0": "=r" (cntp_init));      // 获取当前计
数值
    kdebug("timer init cntpct_el0 = %lu\n", cntp_init);
    asm volatile ("mrs %0, cntfrq_el0": "=r" (cntp_freq));      // 获取计数器
频率
    kdebug("timer init cntfrq_el0 = %lu\n", cntp_freq);

    /* 2. 计算定时器值*/
    cntp_tval = (cntp_freq * TICK_MS / 1000);           // 计算多少个计数对应一个
时钟滴答
    tick_per_us = cntp_freq / 1000 / 1000;                // 计算每微秒的计数值
    kinfo("CPU freq %lu, set timer %lu\n", cntp_freq, cntp_tval);

    /* 3. 设置定时器计数值*/
    asm volatile ("msr cntp_tval_el0, %0": :"r" (cntp_tval)); // 设置计数
值
    asm volatile ("mrs %0, cntp_tval_el0": "=r" (count_down)); // 读回验证
    kdebug("timer init cntp_tval_el0 = %lu\n", count_down);

    /* 4. 启用定时器中断
     * CNTPNSIRQ: Physical Non-secure timer interrupt
     * CNTVIRQ: Virtual timer interrupt
     */
    put32(core_timer_irqcntl[cpuid],                      // 每个CPU核心独立的中断
控制
           INT_SRC_TIMER1 | INT_SRC_TIMER3);           // 使能物理定时器和虚拟定
时器中断

    /* 5. 配置控制寄存器
     * IMASK = 0: 不屏蔽中断
     * ENABLE = 1: 使能定时器
     * cntp_ctl_el0: Counter-timer Physical Timer Control register
     */
    timer_ctl = 0 << 1 | 1; /* IMASK = 0 ENABLE = 1 */
    asm volatile ("msr cntp_ctl_el0, %0": :"r" (timer_ctl)); // 设置控制
寄存器
    asm volatile ("mrs %0, cntp_ctl_el0": "=r" (timer_ctl)); // 读回验证
    kdebug("timer init cntp_ctl_el0 = %lu\n", timer_ctl);
}

```

物理时钟中断与抢占

何实现中断物理时钟后对CPU的"抢占"？核心便是在处理时钟中断时递减当前运行线程的时间片，并在当前运行线程的时间片耗尽时进行调度，选取新的线程运行

来到中断处理的代码：

```
/* Interrupt handler for interrupts happening when in EL0. */
void handle_irq(void)
{
    plat_handle_irq();
    sched_periodic(); // 在rr策略下即为调度函数，但其他策略下不一样，需要注意
    eret_to_thread(switch_context()); // 这个操作即为调度后返回用户态的标准
}
```

继续前进，直接来看处理物理时钟中断的部分：

```
void plat_handle_irq(void)
{
    // ...
    switch (irq) {
        case INT_SRC_TIMER1:
            /* CNTPNSIRQ (Physical Non-Secure timer IRQ) */
            handle_timer_irq();
        // ...
        return;
    }
}
```

```
void handle_timer_irq(void)
{
    u64 current_tick, tick_delta;
    struct time_state *local_time_state;
    struct list_head *local_sleep_list;
    struct lock *local_sleep_list_lock;
    struct sleep_state *iter = NULL, *tmp = NULL;
    struct thread *wakeup_thread;

    /* 获取当前CPU的睡眠队列信息 */
    current_tick = plat_get_current_tick();           // 获取当前时钟计数
    local_time_state = &time_states[smp_get_cpu_id()];
    local_sleep_list = &local_time_state->sleep_list;
    local_sleep_list_lock = &local_time_state->sleep_list_lock;

    /* 遍历并唤醒到期的线程 */
    lock(local_sleep_list_lock); // 加锁保护睡眠队列
    for_each_in_list_safe (iter, tmp, sleep_node, local_sleep_list) {
        if (iter->wakeup_tick > current_tick) {
            break; // 未到唤醒时间, 退出循环
        }
        wakeup_thread = container_of(iter, struct thread,
                                      sleep_state);
        lock(&wakeup_thread->sleep_state.queue_lock);
        list_del(&iter->sleep_node);
        BUG_ON(wakeup_thread->sleep_state.cb == sleep_timer_cb
               && !thread_is_ts_blocking(wakeup_thread));
        kdebug("wake up t:%p at:%ld\n", wakeup_thread, current_tick);
        BUG_ON(wakeup_thread->sleep_state.cb == NULL);
        wakeup_thread->sleep_state.cb(wakeup_thread);
        wakeup_thread->sleep_state.cb = NULL;
        unlock(&wakeup_thread->sleep_state.queue_lock);
    }

    /* 设置下一次定时器中断 */
    tick_delta = get_next_tick_delta(); // 计算下一次中断间隔
    unlock(local_sleep_list_lock);

    /* 更新下次过期时间并配置硬件定时器 */
    time_states[smp_get_cpu_id()].next_expire = current_tick +
    tick_delta;
    plat_handle_timer_irq(tick_delta);

    /* 递减当前线程的时间片 */
    if (current_thread) {
        BUG_ON(!current_thread->thread_ctx->sc);
        BUG_ON(current_thread->thread_ctx->sc->budget == 0);
        current_thread->thread_ctx->sc->budget--; // 减少时间片
    } else {
        kdebug("Timer: system not running!\n");
    }
}
```

为什么两次中断之间的间隔不是固定值而是需要通过计算得到？

- 时钟不只干定时硬中断的活
- 一些其他线程的等待、睡眠之类会导致下一次时钟irq的提前

支持了抢占式调度之后，我们的用户态程序就能打破初始进程procmgr的循环真正运行了

至此，多核调度部分的源码解析结束

Last change: 2025-02-14, commit: [9fa9752](#)

目录

- 进程间通信（IPC）
 - IPC服务端注册
 - 创建Client注册回调线程
 - 通过系统调用注册Server端
 - IPC客户端注册
 - 外层代码与整体逻辑
 - 客户端注册的syscall
 - 注册回调函数
- IPC Call
 - 向共享内存填充数据
 - ipc_call
- IPC Return
- 逻辑流程图

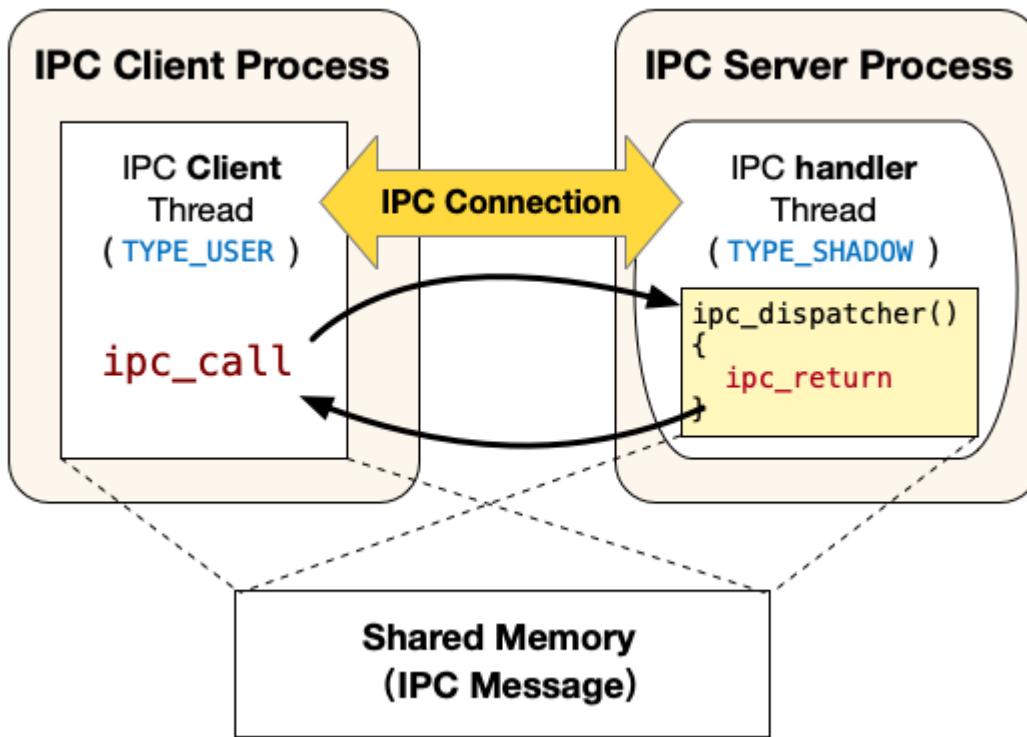
进程间通信（IPC）

本节内容讲解ChCore关于进程间通信（IPC）部分的源码

教材为我们讲解了IPC机制的历史演化：从简单IPC机制到共享内存/内核辅助，再到后来的LRPC，L4等进程间通信等模型。而ChCore的设计则是“择优取之”：

- **消息传递与通知：**基于LRPC的迁移线程技术+L4微内核的直接进程切换技术
- **数据传输：**基于用户态共享内存

整体上看，便是将IPC的两端进程分为Client和Server，注册IPC服务后通过能力组机制建立连接，之后的通信过程则通过syscall来进行，而具体实现则通过上述机制



下面以一次完整的IPC注册-调用的过程，来讲解IPC源码的设计与其机制的实现

IPC服务端注册

要使用IPC功能，首先需要进行服务器端的注册。对于扮演服务器端的进程来说，它需要调用 `ipc_register_server` 来声明自己为IPC的服务器端。先来看它的API接口：

```
int ipc_register_server(server_handler server_handler,
                       void *(*client_register_handler)(void *))
```

其接受两个参数，分别是两个函数，其作用为

- `client_register_handler`：为服务端提供的用于注册的回调函数，用于处理client注册。例 校验发出注册请求的client是否具有相应的capability等
- `server_handler`：服务器端的服务回调函数，在成功注册之后处理LRPC调用，也就是处理迁移线程模型之中，服务器端提供的"被迁移的"代码段

这里的 `server_handler` 定义 下：

```
typedef void (*server_handler)(
    void *shm_ptr, // 共享内存的起始地址
    unsigned int max_data_len, // 共享内存的总长度
    unsigned int send_cap_num, // 客户端发送的capability数量
    badge_t client_badge // 客户端的身份标识
);
```

然后再来看注册函数的代码：

```

int ipc_register_server(server_handler server_handler,
                       void *(*client_register_handler)(void *))
{
    return ipc_register_server_with_destructor(
        server_handler, client_register_handler, DEFAULT_DESTRUCTOR);
}

/*
 * Currently, a server thread can only invoke this interface once.
 * But, a server can use another thread to register a new service.
 */
int ipc_register_server_with_destructor(server_handler server_handler,
                                         void *(*client_register_handler)(void *),
                                         server_destructor server_destructor)
{
    cap_t register_cb_thread_cap;
    int ret;

    /*
     * Create a passive thread for handling IPC registration.
     * - run after a client wants to register
     * - be responsible for initializing the ipc connection
     */
#define ARG_SET_BY_KERNEL 0
    pthread_t handler_tid;
    register_cb_thread_cap =
        chcore_pthread_create_register_cb(&handler_tid,
                                         NULL,
                                         client_register_handler,
                                         (void *)ARG_SET_BY_KERNEL);
    BUG_ON(register_cb_thread_cap < 0);
    /*
     * Kernel will pass server_handler as the argument for the
     * register_cb_thread.
     */
    ret = usys_register_server((unsigned long)server_handler,
                               (unsigned long)register_cb_thread_cap,
                               (unsigned long)server_destructor);
    if (ret != 0) {
        printf("%s failed (retval is %d)\n", __func__, ret);
    }
    return ret;
}

```

此处做了一个带destructor的包装，不过这里传入的默认的destructor是个NULL，暂时先不用管

这里实际上做了两件事，调用了两个函数：

- 创建Client注册回调线程：通过 `chcore_pthread_create_register_cb` 函数实现，返回值为创造的线程的能力

- 该线程是被动线程，负责处理Client端端IPC注册请求
- 线程类型为 `TYPE_REGISTER`，平时不会被调度执行
- 只有当IPC客户端需要注册时该线程才运行，负责初始化IPC连接
- 通过系统调用注册Server端：即 `usys_register_server` 函数，它实际上就是系统调用

下面分别展开讲解

创建Client注册回调线程

核心便是 `chcore_pthread_create_register_cb` 函数，源代码200多行这里就不放了，主要做的还是创建线程那一套：

- 分配线程栈空间
- 设置线程TLS
- 设置线程的回调特性
- 将线程添加到线程链表

关键便在于第三点，我们将回调函数作为参数传入了 `chcore_pthread_create_register_cb`，而它直接被设置为新线程的entry：

```
struct thread_args _args;
_args.cap_group_cap = 0; // SELF_CAP_GROUP
_args.stack = (unsigned long)stack;
// 关键：注册回调线程直接使用entry作为入口点
_args.pc = (type != TYPE_USER ? (unsigned long)entry :
            (unsigned long)(c11 ? start_c11 : start));
_args.arg = (type != TYPE_USER ? (unsigned long)arg : (unsigned long)args);
_args.prio = attr._a_sched? attr._a_prio: CHILD_THREAD_PRIO;
_args.tls = (unsigned long)TP_ADJ(new);
_args.type = type;
_args.clear_child_tid = (int *)&__thread_list_lock;

ret = new->tid = usys_create_thread((unsigned long)&_args);
```

忘了这是什么东西？回忆一下结构体组成即可：

```
struct thread_args {
    /* specify the cap_group in which the new thread will be created */
    cap_t cap_group_cap;
    vaddr_t stack;
    vaddr_t pc;
    unsigned long arg;
    vaddr_t tls;
    unsigned int prio;
    unsigned int type;
    int *clear_child_tid;
};
```

随后便用传入的参数通过系统调用创建线程了

通过系统调用注册Server端

直接看 `usys_register_server` :

```
int usys_register_server(unsigned long callback,
                        cap_t register_thread_cap,
                        unsigned long destructor)
{
    return chcore_syscall3(CHCORE_SYS_register_server,
                           callback,
                           register_thread_cap,
                           destructor);
}
```

顺藤摸瓜找到真正被调用的syscall `sys_register_server` :

```
int sys_register_server(unsigned long ipc_routine, cap_t register_thread_cap,
                       unsigned long destructor)
{
    return register_server(
        current_thread, ipc_routine, register_thread_cap,
        destructor);
}

static int register_server(struct thread *server, unsigned long ipc_routine,
                          cap_t register_thread_cap, unsigned long
destructor)
```

这里已经嵌套多层了，注意各个参数：

- `server` : 代表服务器线程，在这里即为 `current_thread`，指的是服务器进程端的主线程（可以参考Lab文档）
- `ipc_routine` : 服务回调函数，即为前面的 `server_handler`
- `register_thread_cap` : 注册回调线程的能力
- `destructor` : 析构函数，这里还是null

接下来就看看这个函数做了什么：

```
/**  
 * @param server 要注册为IPC服务器的线程  
 * @param ipc_routine IPC服务例程的入口点地址  
 * @param register_thread_cap 注册回调线程的capability  
 * @param destructor 连接关闭时的清理函数地址  
 * @return int 0表示成功，负值表示错误码  
 */  
static int register_server(struct thread *server, unsigned long ipc_routine,  
                           cap_t register_thread_cap, unsigned long  
                           destructor)  
{  
    struct ipc_server_config *config;  
    struct thread *register_cb_thread;  
    struct ipc_server_register_cb_config *register_cb_config;  
  
    /* 确保服务器线程存在 */  
    BUG_ON(server == NULL);  
  
    /* 检查服务器线程是否已经注册过IPC服务 */  
    if (server->general_ipc_config != NULL) {  
        kdebug("A server thread can only invoke **register_server**  
once!\n");  
        return -EINVAL;  
    }  
  
    /*  
     * 获取并验证注册回调线程  
     * 该线程用于处理客户端的注册请求，必须是TYPE_REGISTER类型  
     */  
    register_cb_thread =  
        obj_get(current_cap_group, register_thread_cap, TYPE_THREAD);  
    if (!register_cb_thread) {  
        kdebug("A register_cb_thread is required.\n");  
        return -ECAPABILITY;  
    }  
  
    /* 验证回调线程类型 */  
    if (register_cb_thread->thread_ctx->type != TYPE_REGISTER) {  
        kdebug("The register_cb_thread should be TYPE_REGISTER!\n");  
        obj_put(register_cb_thread);  
        return -EINVAL;  
    }  
  
    /* 为服务器分配配置结构 */  
    config = kmalloc(sizeof(*config));  
    if (!config) {  
        obj_put(register_cb_thread);  
        return -ENOMEM;  
    }  
  
    /*  
     * 设置IPC服务例程入口点  
     * 注：不检查地址合法性，如果地址非法服务器会自行终止  
     */
```

```
config->declared_ipc_routine_entry = ipc_routine;

/* 记录注册回调线程 */
config->register_cb_thread = register_cb_thread;

/* 为注册回调线程分配配置结构 */
register_cb_config = kmalloc(sizeof(*register_cb_config));
if (!register_cb_config) {
    kfree(config);
    obj_put(register_cb_thread);
    return -ENOMEM;
}
register_cb_thread->general_ipc_config = register_cb_config;

/*
 * 初始化注册锁
 * 该锁用于确保同一时间只有一个客户端可以进行注册
 * 即注册过程是串行的，避免并发注册可能带来的问题
 */
lock_init(&register_cb_config->register_lock);

/*
 * 记录注册回调线程的关键信息
 * 包括：入口点地址(PC)、初始栈指针(SP)和析构函数
 */
register_cb_config->register_cb_entry =
    arch_get_thread_next_ip(register_cb_thread);
register_cb_config->register_cb_stack =
    arch_get_thread_stack(register_cb_thread);
register_cb_config->destructor = destructor;
obj_put(register_cb_thread);

#if defined(CHCORE_ARCH_AARCH64)
/*
 * ARM架构需要内存屏障
 * 确保在设置server->general_ipc_config之前
 * 所有配置相关的数据（register_lock）都已经正确初始化并可见
 */
smp_mb();
#else
/* x86等使用TSO内存模型的架构不需要内存屏障 */
#endif

/*
 * 最后一步：设置服务器的IPC配置
 * 这个字段同时也标志着该服务器线程已声明IPC服务
 * 只有设置了这个字段，服务才算真正就绪
 */
server->general_ipc_config = config;

return 0;
}
```

注册的过程就是配置 `config`，这里一共配置了两个线程的 `config`，分别是主线程和回调线程。看看结构体的成员：

```
/**  
 * 当服务器线程调用"register_server"时会创建此配置。  
 * 该线程在服务器进程中声明IPC服务，并对客户端可见。  
 * 客户端可以通过"register_client"与此服务器线程建立连接。  
 */  
struct ipc_server_config {  
    /**  
     * 这个线程专门用于处理新客户端的注册请求  
     * 一个服务器只能有一个注册回调线程  
     * 该线程会被阻塞直到有新的客户端请求注册  
     */  
    struct thread *register_cb_thread;  
  
    /**  
     * 记录服务器提供的IPC处理函数的地址  
     * 当客户端发起IPC请求时，处理线程会跳转到此地址执行  
     * 这个地址在服务器注册时由服务器线程提供  
     */  
    unsigned long declared_ipc_routine_entry;  
};  
  
/**  
 * 存储处理客户端注册请求的线程的相关信息  
 * 这个线程专门用于处理IPC注册过程  
 */  
struct ipc_server_register_cb_config {  
    /**  
     * 用于序列化客户端的注册请求  
     * 确保同一时间只处理一个客户端的注册  
     * 防止并发注册可能导致的竞态条件  
     */  
    struct lock register_lock;  
  
    /**  
     * 记录处理注册请求的函数地址  
     * 当有新的注册请求时，线程从这个地址开始执行  
     */  
    vaddr_t register_cb_entry;  
  
    /* 记录线程的栈顶地址 */  
    vaddr_t register_cb_stack;  
  
    /**  
     * 当连接被关闭时调用的清理函数  
     * 用于释放连接相关的资源  
     */  
    vaddr_t destructor;  
  
    /**  
     * 以下三个字段用于临时存储正在建立的连接相关的capability  
     * 这些字段只在连接建立过程中有效  
     */
```

```
/* 客户端中的连接capability */
cap_t conn_cap_in_client;
/* 服务器中的连接capability (当前未使用, 预留给未来扩展) */
cap_t conn_cap_in_server;
/* 服务器中的共享内存capability */
cap_t shm_cap_in_server;
};
```

对服务器主线程来说，注册的过程为其config配置了注册回调线程和服务线程的入口函数，即前面的 server_handler；对注册回调线程来说，它需要保存自己的PC和SP等信息，为后面的线程迁移做准备。最后处理TLB，并设置server主线程的相应字段，结束注册过程

这时候Server端注册已经完成，下一步是注册客户端并建立IPC连接

IPC客户端注册

外层代码与整体逻辑

上述注册服务端的操作皆在服务器进程上完成，而之后则进行IPC的客户端注册与连接，是在IPC客户端线程上执行函数完成的，我们先看源码：

```
/**  
 * @brief 注册IPC客户端  
 *  
 * 该函数负责将当前线程注册为指定服务器的IPC客户端。  
 * 注册过程包括：创建共享内存、建立连接、初始化IPC结构。  
 * 一个客户端线程可以多次注册，与不同的服务器建立连接。  
 *  
 * @param server_thread_cap 目标服务器线程的capability  
 * @return ipc_struct_t* 成功返回IPC结构体指针，失败返回NULL  
 * @note 返回的结构体是从堆上分配的，使用完需要调用者释放  
 */  
  
ipc_struct_t *ipc_register_client(cap_t server_thread_cap)  
{  
    /* 连接的capability */  
    cap_t conn_cap;  
    /* 客户端IPC结构体 */  
    ipc_struct_t *client_ipc_struct;  
    /* 共享内存配置 */  
    struct client_shm_config shm_config;  
    /* 共享内存的能力 */  
    cap_t shm_cap;  
  
    /* 为IPC结构体分配内存 */  
    client_ipc_struct = malloc(sizeof(ipc_struct_t));  
    if (client_ipc_struct == NULL) {  
        return NULL;  
    }  
  
    /*  
     * 在向服务器注册之前，客户端需要先分配共享内存  
     * 这块内存后续会与服务器共享  
     *  
     * 注意：这里使用PMO_DATA而不是PMO_SHM的原因是：  
     * - 当前IPC共享内存(IPC_PER_SHM_SIZE)只有一页大小  
     * - 在这种情况下PMO_DATA更加高效  
     *  
     * 果将来共享内存需要更大，可以切换回PMO_SHM  
     * 两种类型都经过测试，都能正常工作  
     */  
    shm_cap = usys_create_pmo(IPC_PER_SHM_SIZE, PMO_DATA);  
    if (shm_cap < 0) {  
        printf("usys_create_pmo ret %d\n", shm_cap);  
        goto out_free_client_ipc_struct;  
    }  
  
    /* 配置共享内存参数 */  
    shm_config.shm_cap = shm_cap;  
    shm_config.shm_addr = chcore_alloc_vaddr(IPC_PER_SHM_SIZE);  
  
    /* 循环尝试注册，直到成功或出错 */  
    while (1) {  
        /* 向服务器发起注册请求 */
```

```

conn_cap = usys_register_client(server_thread_cap,
                                (unsigned long)&shm_config);

    if (conn_cap == -EIPC_RETRY) {
        /*
         * 服务器IPC可能还未就绪
         * 让出CPU并重试
         */
        usys_yield();
    } else if (conn_cap < 0) {
        /* 注册失败，打印错误信息 */
        printf("client: %s failed (return %d),
server_thread_cap is %d\n",
               __func__,
               conn_cap,
               server_thread_cap);
        goto out_free_vaddr;
    } else {
        /* 注册成功，跳出循环 */
        break;
    }
}

/* 初始化IPC结构体 */
client_ipc_struct->lock = 0; // 初始化锁
client_ipc_struct->shared_buf = shm_config.shm_addr;// 设置共享内存地址
client_ipc_struct->shared_buf_len = IPC_PER_SHM_SIZE;// 设置共享内存大小
client_ipc_struct->conn_cap = conn_cap; // 保存连接capability

return client_ipc_struct;

/* 错误处理路径 */
out_free_vaddr:
    /* 撤销共享内存capability并释放虚拟地址 */
    usys_revoke_cap(shm_cap, false);
    chcore_free_vaddr(shm_config.shm_addr, IPC_PER_SHM_SIZE);

out_free_client_ipc_struct:
    /* 释放IPC结构体 */
    free(client_ipc_struct);

return NULL;
}

```

这部分要干的活就多了：

- 通过syscall申请一块共享内存
- 执行另一个syscall，完成客户端共享内存的映射，这里还会进入之前创建好的注册回调线程，创建服务线程，处理服务端的相应信息
- 最后设置IPC结构体的字段并返回之

和之前服务器端的注册函数一样，这里的 `usys_create_pmo` 和 `usys_register_client` 也是系统调用的一层包装，我们需要重点关注的是后者，它将进一步涉及到注册回调线程的相关操作

客户端注册的syscall

源码 下所示，可以参考注释理解函数内容

```
/**  
 * @brief 注册IPC客户端的系统调用  
 *  
 * 该函数处理客户端的IPC注册请求，建立客户端与服务器之间的IPC连接。  
 * 整个注册过程包括：共享内存设置、连接创建、线程迁移等步骤。  
 *  
 * @param server_cap 服务器线程的capability  
 * @param shm_config_ptr 指向用户空间共享内存配置的指针  
 * @return 成功返回连接的capability，失败返回负数错误码  
 */  
cap_t sys_register_client(cap_t server_cap, unsigned long shm_config_ptr)  
{  
    struct thread *client;  
    struct thread *server;  
  
    /*  
     * 共享内存配置结构体  
     * 注：实际无需初始化，但fbinfer静态分析工具会因无法识别copy_from_user而报警  
     */  
    struct client_shm_config shm_config = {0};  
    int r;  
    struct client_connection_result res;  
  
    /* 获取当前线程作为客户端 */  
    client = current_thread;  
  
    /* 通过capability获取服务器线程对象 */  
    server = obj_get(current_cap_group, server_cap, TYPE_THREAD);  
    if (!server) {  
        r = -ECAPABILITY;  
        goto out_fail;  
    }  
  
    /* 获取服务器的IPC配置 */  
    server_config =  
        (struct ipc_server_config *) (server->general_ipc_config);  
    if (!server_config) {  
        /* 服务器未完成IPC初始化 */  
        r = -EIPCRETRY;  
        goto out_fail;  
    }  
  
    /*  
     * 首先定位注册回调线程  
     * 之后会直接将控制流转移给该线程以完成注册  
     *  
     * 完整的注册流程：  
     * 客户端线程 -> 服务器注册回调线程 -> 客户端线程  
     */  
    register_cb_thread = server_config->register_cb_thread;  
    register_cb_config =  
        (struct ipc_server_register_cb_config  
         *) (register_cb_thread->general_ipc_config);
```

```
/*
 * 获取注册锁以避免并发注册
 *
 * 使用try_lock而不是lock的原因:
 * ChCore不支持互斥锁, unlock操作由另一个线程完成
 * 果使用lock可能导致死锁
 */
if (try_lock(&register_cb_config->register_lock) != 0) {
    r = -EIPCRETRY;
    goto out_fail;
}

/* 在访问用户空间数据前验证地址合法性 */
if (check_user_addr_range(shm_config_ptr, sizeof(shm_config) != 0)) {
    r = -EINVAL;
    goto out_fail_unlock;
}

/* 从用户空间复制共享内存配置 */
r = copy_from_user((void *)&shm_config,
                    (void *)shm_config_ptr,
                    sizeof(shm_config));
if (r) {
    r = -EINVAL;
    goto out_fail_unlock;
}

/* 在当前cap组中映射共享内存PMO */
r = map_pmo_in_current_cap_group(
    shm_config.shm_cap, shm_config.shm_addr, VMR_READ |
VMR_WRITE);

if (r != 0) {
    goto out_fail_unlock;
}

/* 创建IPC连接对象 */
r = create_connection(
    client, server, shm_config.shm_cap, shm_config.shm_addr,
&res);

if (r != 0) {
    goto out_fail_unlock;
}

/* 记录客户端进程的连接capability */
register_cb_config->conn_cap_in_client = res.client_conn_cap;
register_cb_config->conn_cap_in_server = res.server_conn_cap;
/* 记录当前连接的服务器端共享内存capability */
register_cb_config->shm_cap_in_server = res.server_shm_cap;

/* 将当前线程标记为阻塞状态 */
thread_set_ts_blocking(current_thread);
```

```

/* 设置目标线程的栈指针、程序计数器和参数 */
arch_set_thread_stack(register_cb_thread,
                      register_cb_config->register_cb_stack);
arch_set_thread_next_ip(register_cb_thread,
                        register_cb_config->register_cb_entry);
arch_set_thread_arg0(register_cb_thread,
                     server_config->declared_ipc_routine_entry);
obj_put(server);

/* 传递调度上下文 */
register_cb_thread->thread_ctx->sc = current_thread->thread_ctx->sc;

/* 成功后：切换到服务器的回调线程 */
sched_to_thread(register_cb_thread);

/* 永远不会返回到这里 */
BUG_ON(1);

/* 错误处理路径 */
out_fail_unlock:
    /* 解锁注册锁 */
    unlock(&register_cb_config->register_lock);
out_fail: /* 可能返回EAGAIN */
    /* 清理服务器线程对象引用 */
    if (server)
        obj_put(server);
    return r;
}

```

这个注册函数的大体流程即为

- 在客户端映射共享内存
 - 从当前thread的 cap_group 里面找到传入的 server_cap 对应的slot，进而得到 server 线程对象
 - 从 server 获取它的 ipc_config
 - 拿锁，避免并发问题
 - 检查 client 声明的共享内存地址，并将之拷贝到内核态，再建立共享内存上的映射
- 创建 ipc_connection 对象，并把 cap 给到 server 和 client
- 切换到注册回调线程
 - 设置好调用参数，栈寄存器，异常处理寄存器
 - 这一部分用到了注册 server 时配置好的 config
 - 调用 sched 切换控制权给 server 的注册回调线程，并进入设置好的注册回调函数

关于创建 ipc_connection 的过程，可以参考 create_connection 函数，配置好的结构体下：

```
struct ipc_connection *conn = {
    /*
     * 当前使用此连接的客户端线程
     * 在create_connection时设置为发起注册的客户端线程
     */
    .current_client_thread = client_thread,

    /*
     * 服务器处理线程
     * 在create_connection时为NULL
     * 将在sys_ipc_register_cb_return中由服务器设置
     * 这里还没设置
     */
    .server_handler_thread = NULL,

    /*
     * 客户端标识信息
     * badge: 用于在IPC过程中标识不同的客户端进程
     * pid: 客户端进程ID, 用于资源管理
     */
    .client_badge = current_cap_group->badge,
    .client_pid = current_cap_group->pid,

    /*
     * 共享内存配置
     * 包含了客户端和服务器端的地址和capability信息
     */
    .shm = {
        .client_shm_uaddr = shm_addr_client,           // 客户端映射地址
        .server_shm_uaddr = 0,                          // 服务器映射地址 (尚未
设置)
        .shm_size = shm_size,                          // 共享内存大小
        .shm_cap_in_client = shm_cap_client,          // 客户端的共享内存
capability
        .shm_cap_in_server = shm_cap_server           // 服务器的共享内存
capability
    },
    /*
     * 资源回收相关字段
     */
    .ownership = LOCK_INIT_VAL,                    // 初始化的锁
    .conn_cap_in_client = conn_cap,                // 客户端的连接
capability
    .conn_cap_in_server = server_conn_cap,         // 服务器的连接
capability
    .state = CONN_INCOME_STOPPED,                  // 初始状态: 停止接收

    /*
     * capability传输缓冲区
     * 用于在IPC过程中传输capability
     */
}
```

```
* 初始状态下都是空的
*/
.server_cap_buf = {
    [0 ... MAX_CAP_TRANSFER-1] = {
        .valid = false,
        .cap = 0,
        .mask = 0,
        .rest = 0
    }
},
.client_cap_buf = {
    [0 ... MAX_CAP_TRANSFER-1] = {
        .valid = false,
        .cap = 0,
        .mask = 0,
        .rest = 0
    }
}
};

};
```

然后 `sys_register_client` 的历史使命就结束了，它把尚未完成的任务交给了注册回调线程

注册回调函数

注册回调线程运行的入口函数为主线程调用 `ipc_register_server` 提供的 `client_register_handler` 参数，一般会使用默认的 `DEFAULT_CLIENT_REGISTER_HANDLER` 宏定义的入口函数，即定义在 `user/chcore-libc/musl-libc/src/chcore-port/ipc.c` 中的 `register_cb`

根据Lab文档的指引我们来到 `register_cb` 的地盘，还记得这个 `client_register_handler` 是干嘛的不？它在服务端主线程创建注册回调线程的时候被设置为了注册回调线程的入口。那么在注册客户端函数将线程切换过来的时候（注意这里用的切换函数是 `sched_to_thread`），便会执行 `register_cb` 的代码

那么它又会肩负怎样的IPC使命呢？且看代码（注意现在已经是server端进程下的注册回调线程了）：

```
/*
 * @param ipc_handler 服务器的IPC处理函数，就是我们之前传进来的server_handler
 * 这里我们需要用它来注册server进程下的服务线程，用于之后具体处理client发来的IPC请求
 * @return void* 总是返回NULL
 */
void *register_cb(void *ipc_handler)
{
    /* 服务器处理线程的capability */
    cap_t server_thread_cap = 0;
    /* 服务器端共享内存地址 */
    unsigned long shm_addr;

    /* 为服务器分配共享内存的虚拟地址空间 */
    shm_addr = chcore_alloc_vaddr(IPC_PER_SHM_SIZE);

    /*
     * 创建服务器处理线程（影子线程）
     * 这个线程将负责处理来自该客户端的所有IPC请求
     */
    pthread_t handler_tid;
    server_thread_cap = chcore_pthread_create_shadow(
        &handler_tid,      // 存储线程ID
        NULL,              // 线程属性（使用默认值）
        ipc_handler,       // IPC处理函数
        (void *)NO_ARG    // 无参数
    );
    BUG_ON(server_thread_cap < 0);

    /* 调用系统调用完成注册过程 */
#ifndef CHCORE_ARCH_X86_64
    ipc_register_cb_return(
        server_thread_cap,
        (unsigned long)ipc_shadow_thread_exit_routine,
        shm_addr
    );
#else
    ipc_register_cb_return(
        server_thread_cap,
        (unsigned long)ipc_shadow_thread_exit_routine_naked,
        shm_addr
    );
#endif

    return NULL;
}
```

创建线程的部分和之前 出一辙，只是注意这里的服务线程同样是没有调度上下文的影子线程
忘记什么是服务线程了？回看Lab文档：

ChCore的IPC接口不是传统的send/recv接口。其更像客户端/服务器模型，其中IPC请求

接收者是服务器，而IPC请求发送者是客户端。服务器进程中包含三类线程：

- 主线程：该线程与普通的线程一样，类型为 `TYPE_USER`。该线程会调用 `ipc_register_server` 将自己声明为一个IPC的服务器进程，调用的时候会提供两个参数：服务连接请求的函数 `client_register_handler` 和服务真正IPC请求的函数 `server_handler`（即图中的 `ipc_dispatcher`），调用该函数会创建一个注册回调线程；
- 注册回调线程：该线程的入口函数为上文提到的 `client_register_handler`，类型为 `TYPE_REGISTER`。正常情况下该线程不会被调度执行，仅当有Client发起建立IPC连接的请求时，该线程运行并执行 `client_register_handler`，为请求建立连接的Client 创建一个服务线程（即图中的IPC handler thread）并在服务器进程的虚拟地址空间中分配一个可以用来映射共享内存的虚拟地址。
- 服务线程：当Client发起建立IPC连接请求时由注册回调线程创建，入口函数为上文提到的 `server_handler`，类型为 `TYPE_SHADOW`。正常下该线程不会被调度执行，仅当有Client端线程使用 `ipc_call` 发起IPC请求时，该线程运行并执行 `server_handler`（即图中的 `ipc_dispatcher`），执行结束之后会调用 `ipc_return` 回到Client端发起IPC请求的线程。

还记得 `sys_register_client` 尚未完成的历史使命吗？现在共享内存还只是在client端做好了映射，server端目前仅有一个申请好的虚拟地址；同时 `struct ipc_connection` 也还有部分关于server的字段没有填写，于是这里的 `ipc_register_cb_return` 将接过IPC的接力棒：

```
/** 
 * 向内核发起注册完成请求，该函数是用户态库函数，将注册完成请求传递给内核。
 * 在注册回调线程创建完服务器处理线程后调用。
 *
 * @param server_thread_cap 服务器处理线程的capability
 * @param server_thread_exit_routine 处理线程的退出例程
 * @param server_shm_addr 服务器端共享内存地址
 */
void ipc_register_cb_return(
    cap_t server_thread_cap,
    unsigned long server_thread_exit_routine,
    unsigned long server_shm_addr)
{
    /* 直接调用系统调用完成注册 */
    usys_ipc_register_cb_return(
        server_thread_cap,
        server_thread_exit_routine,
        server_shm_addr
    );
}
```

虚晃一枪，原来是syscall套壳，我们继续前进：

```
/**  
 * 该函数在内核中完成IPC注册的最后阶段，包括：  
 * 1. 设置服务器端共享内存  
 * 2. 初始化服务器处理线程  
 * 3. 完成连接建立  
 * 4. 返回客户端  
 *  
 * @param server_handler_thread_cap 服务器处理线程的capability  
 * @param server_thread_exit_routine 处理线程的退出例程地址  
 * @param server_shm_addr 服务器端共享内存地址  
 * @return 成功返回0，失败返回负数错误码  
 */  
int sys_ipc_register_cb_return(  
    cap_t server_handler_thread_cap,  
    unsigned long server_thread_exit_routine,  
    unsigned long server_shm_addr)  
{  
    struct ipc_server_register_cb_config *config;  
    struct ipc_connection *conn;  
    struct thread *client_thread;  
    struct thread *ipc_server_handler_thread;  
    struct ipc_server_handler_config *handler_config;  
    int r = -ECAPABILITY;  
  
    /* 获取当前线程（注册回调线程）的IPC配置 */  
    config = (struct ipc_server_register_cb_config *)  
        current_thread->general_ipc_config;  
    if (!config)  
        goto out_fail;  
  
    /* 获取IPC连接对象 */  
    conn = obj_get(  
        current_cap_group,  
        config->conn_cap_in_server,  
        TYPE_CONNECTION  
    );  
    if (!conn)  
        goto out_fail;  
  
    /* 获取服务器处理线程（即服务线程）对象 */  
    ipc_server_handler_thread = (struct thread *)obj_get(  
        current_cap_group,  
        server_handler_thread_cap,  
        TYPE_THREAD  
    );  
    if (!ipc_server_handler_thread)  
        goto out_fail_put_conn;  
  
    /* 在服务器地址空间映射共享内存 */  
    r = map_pmo_in_current_cap_group(  
        config->shm_cap_in_server,  
        server_shm_addr,  
        VMR_READ | VMR_WRITE  
    );
```

```
if (r != 0)
    goto out_fail_put_thread;

/* 获取发起注册的客户端线程 */
client_thread = conn->current_client_thread;
/* 设置客户端的返回值 (连接capability) */
arch_set_thread_return(client_thread, config->conn_cap_in_client);

/*
 * 初始化服务线程的IPC配置
 * 果配置已存在 (线程重用) , 则跳过初始化
 */
if (!ipc_server_handler_thread->general_ipc_config) {
    handler_config = (struct ipc_server_handler_config *)kmalloc(
        sizeof(*handler_config));
    if (!handler_config) {
        r = -ENOMEM;
        goto out_fail_put_thread;
    }
    ipc_server_handler_thread->general_ipc_config =
handler_config;
    lock_init(&handler_config->ipc_lock);

    /* 记录处理线程的执行上下文, 包括PC和SP等 */
    handler_config->ipc_routine_entry =
        arch_get_thread_next_ip(ipc_server_handler_thread);
    handler_config->ipc_routine_stack =
        arch_get_thread_stack(ipc_server_handler_thread);
    handler_config->ipc_exit_routine_entry =
        server_thread_exit_routine;
    handler_config->destructor = config->destructor;
}
obj_put(ipc_server_handler_thread);

/* 更新连接对象的服务器信息 */
conn->shm.server_shm_uaddr = server_shm_addr;
conn->server_handler_thread = ipc_server_handler_thread;
conn->state = CONN_VALID; // 标记连接为有效
conn->current_client_thread = NULL;
conn->conn_cap_in_client = config->conn_cap_in_client;
conn->conn_cap_in_server = config->conn_cap_in_server;
obj_put(conn);

thread_set_ts_waiting(current_thread);
unlock(&config->register_lock);
current_thread->thread_ctx->sc = NULL;
sched_to_thread(client_thread);

/* 错误处理 */
out_fail_put_thread:
    obj_put(ipc_server_handler_thread);
out_fail_put_conn:
    obj_put(conn);
out_fail:
    return r;
```

}

最后返回一个连接信息的 `ipc_struct`，即IPC控制块（后文的ICB），并将线程切换回client线程，结束注册过程

总体来说，干了三件事：

- 完成共享内存的创建以及在两边进程的映射
- 创建存储信息的相关结构体并完善其内容
- 通过注册回调线程创建了第一个待命的服务线程

至此IPC的准备工作结束，下面开始正式的 `ipc_call`

IPC Call

向共享内存填充数据

首先，client会调用 `ipc_create_msg` 和 `ipc_set_msg_data` 向共享内存（前面注册时候和server“沟通完毕”）填充数据，之后将前面注册过程的 `ipc_struct` 和 `ipc_create_msg` 得到的msg作为参数，调用 `ipc_call`

向共享内存填写数据的代码 下：

```
// 下面的函数是在创建需要发送的消息
ipc_msg_t *ipc_create_msg(ipc_struct_t *icb, unsigned int data_len)
{
    /* 调用完整版本，cap数量为0 */
    return ipc_create_msg_with_cap(icb, data_len, 0);
}

/**
 * @param icb IPC控制块
 * @param data_len 消息数据长度
 * @param send_cap_num 要发送的capability数量
 * @return ipc_msg_t* 创建的IPC消息，失败则退出程序
 */
ipc_msg_t *ipc_create_msg_with_cap(ipc_struct_t *icb, unsigned int data_len,
                                    unsigned int send_cap_num)
{
    /* 编译时检查：确保消息结构体不会超过缓冲区大小 */
    BUILD_BUG_ON(sizeof(ipc_msg_t) > SERVER_IPC_MSG_BUF_SIZE);
    ipc_msg_t *ipc_msg;
    unsigned long buf_len;

    if (unlikely(icb->conn_cap == 0)) {
        if (connect_system_server(icb) != 0) {
            printf("connect ipc server failed!\n");
            exit(-1);
        }
    }

    /* 获取IPC锁，保护消息设置过程 */
    chcore_spin_lock(&(icb->lock));
    buf_len = icb->shared_buf_len;

    if (data_len > buf_len) {
        printf("%s failed: too long msg (the usable shm size is
0x%lx)\n",
               __func__,
               buf_len);
        goto out_unlock;
    }

    /* 分配IPC消息结构体 */
    ipc_msg = (ipc_msg_t *)malloc(sizeof(ipc_msg_t));
    if (!ipc_msg) {
        goto out_unlock;
    }

    /* 初始化IPC消息 */
    ipc_msg->data_ptr = SHM_PTR_TO_CUSTOM_DATA_PTR(icb->shared_buf); // 设置数据指针
    ipc_msg->max_data_len = buf_len; // 设置最大长度
    ipc_msg->send_cap_num = send_cap_num; // 设置capability数量
    ipc_msg->response_hdr = (struct ipc_response_hdr *)icb->shared_buf;//
```

设置响应头

```
    ipc_msg->icb = icb; //
```

关联IPC控制块

```
    ipc_msg->thread_type = THREAD_CLIENT; //
```

标记为客户端消息

```
return ipc_msg;
```

out_unlock:

```
    /* 错误处理：释放锁并退出 */
    chcore_spin_unlock(&(icb->lock));
    printf("ipc create msg failed!\n");
    exit(-1);
}
```

// 将消息填写到共享内存里

/**

* 该函数将数据写入到IPC消息的共享内存中。

* 消息结构示意图：

*

* 共享内存布局：

```
* +-----+
* |       响应头      | <-- ipc_msg->response_hdr
* +-----+
* |       数据区      | <-- ipc_msg->data_ptr
* |   (可写入的区域)  |
* +-----+
```

*

* @param ipc_msg IPC消息对象（由ipc_create_msg创建）

* @param data 要写入的数据

* @param offset 写入位置的偏移量

* @param len 要写入的数据长度

* @return 成功返回0，失败返回-1

*/

```
int ipc_set_msg_data(ipc_msg_t *ipc_msg, void *data, unsigned int offset,
                     unsigned int len)
```

{

/*

* 检查是否发生溢出：

* 1. offset + len < offset: 加法溢出

* 2. offset + len > max_data_len: 超出缓冲区范围

*/

```
if ((offset + len < offset) || (offset + len > ipc_msg-
>max_data_len)) {
```

```
    printf("%s failed due to overflow.\n", __func__);
    return -1;
}
```

/*

* 将数据复制到消息缓冲区

* ipc_get_msg_data(ipc_msg): 获取数据区起始地址

* + offset: 定位到写入位置

```
*/  
memcpy(ipc_get_msg_data(ipc_msg) + offset, data, len);  
return 0;  
}
```

ipc_call

整个 ipc_call 也是一层系统调用的包装：

```
long ipc_call(ipc_struct_t *icb, ipc_msg_t *ipc_msg)  
{  
    long ret;  
  
    /* 按需创建IPC连接 */  
    if (unlikely(icb->conn_cap == 0)) {  
        if ((ret = connect_system_server(icb)) != 0)  
            return ret;  
    }  
  
    /*  
     * 循环尝试发起系统调用  
     * 当返回-EIPCRETRY时重试（表示暂时无法获取锁）  
     */  
    do {  
        ret = usys_ipc_call(  
            icb->conn_cap, // 连接capability  
            ipc_get_msg_send_cap_num(ipc_msg) // 要传输的  
capability数量  
        );  
    } while (ret == -EIPCRETRY);  
  
    return ret;  
}
```

继续前进找到系统调用的实现：

```
/**  
 * @param conn_cap 连接capability  
 * @param cap_num 要传输的capability数量  
 * @return 调用结果  
 */  
unsigned long sys_ipc_call(cap_t conn_cap, unsigned int cap_num)  
{  
    struct ipc_connection *conn;  
    int r = 0;  
  
    /* 检查capability数量 */  
    if (cap_num > MAX_CAP_TRANSFER) {  
        return -EINVAL;  
    }  
  
    /* 获取连接对象 */  
    conn = obj_get(current_cap_group, conn_cap, TYPE_CONNECTION);  
    if (unlikely(!conn)) {  
        return -ECAPBILITY;  
    }  
  
    /* 尝试获取连接所有权锁 */  
    if (try_lock(&conn->ownership) == 0) {  
        if (conn->state != CONN_VALID) {  
            unlock(&conn->ownership);  
            obj_put(conn);  
            return -EINVAL;  
        }  
    } else {  
        obj_put(conn);  
        r = check_if_exiting();  
        return r;  
    }  
    if ((r = lock_ipc_handler_thread(conn)) != 0)  
        goto out_obj_put;  
  
    /* 清空服务器cap缓冲区 */  
    for (int i = 0; i < MAX_CAP_TRANSFER; i++) {  
        conn->server_cap_buf[i].valid = false;  
    }  
  
    /*  
     * 迁移到服务器线程  
     * 注意：共享内存存在客户端和服务器可能映射到不同地址  
     */  
    ipc_thread_migrate_to_server(  
        conn, // 连接对象  
        conn->shm.server_shm_uaddr, // 服务器端共享内存地址  
        conn->shm.shm_size, // 共享内存大小  
        cap_num // capability数量  
    );  
  
    BUG("should not reach here\n");
```

```
out_obj_put:  
    unlock(&conn->ownership);  
    obj_put(conn);  
    return r;  
}
```

整体上就是由cap找到具体的conn对象，随后以此发起线程迁移（LRPC中的技术），可参考Lab文档

该系统调用将设置服务器端的服务线程的栈地址、入口地址、各个参数，然后迁移到该服务器端服务线程继续运行。由于当前的客户端线程需要等待服务器端的服务线程处理完毕，因此需要更新其状态为TS_WAITING，且不要加入等待队列

```
static void ipc_thread_migrate_to_server(
    struct ipc_connection *conn,           // IPC连接对象
    unsigned long shm_addr,                // 服务器端共享内存地址
    size_t shm_size,                      // 共享内存大小
    unsigned int cap_num)                 // 要传输的capability数量
{
    /* 1. 获取目标线程和配置 */
    struct thread *target = conn->server_handler_thread;
    struct ipc_server_handler_config *handler_config =
        (struct ipc_server_handler_config *)target-
>general_ipc_config;

    /*
     * 2. 记录活动连接
     * 因为一个服务器处理线程可能服务多个连接
     * 需要记录当前正在处理哪个连接
     */
    handler_config->active_conn = conn;

    /*
     * 3. 记录活动客户端线程
     * 因为多个客户端线程可能共享同一个连接
     * 需要记录当前服务的是哪个客户端线程
     */
    conn->current_client_thread = current_thread;

    /* 4. 设置客户端线程状态为阻塞 */
    thread_set_ts_blocking(current_thread);

    /*
     * 5. 传递调度上下文
     * 服务器线程将使用客户端线程的调度配额
     */
    target->thread_ctx->sc = current_thread->thread_ctx->sc;

    /*
     * 6. 设置目标线程的执行上下文
     * - 设置栈指针
     * - 设置程序计数器（入口点）
     */
    arch_set_thread_stack(target, handler_config->ipc_routine_stack);
    arch_set_thread_next_ip(target, handler_config->ipc_routine_entry);

    /*
     * 7. 设置IPC参数
     * arg0: 共享内存地址
     * arg1: 共享内存大小
     * arg2: capability数量
     * arg3: 客户端标识 (badge)
     */
    arch_set_thread_arg0(target, shm_addr);
    arch_set_thread_arg1(target, shm_size);
```

```
        arch_set_thread_arg2(target, cap_num);
#define CHCORE_OPENTRUSTEE
    /* OpenTrustee环境下的特殊处理 */
    arch_set_thread_arg3(
        target, pid_to_taskid(current_thread->cap, conn-
>client_badge));
#else
    arch_set_thread_arg3(target, conn->client_badge);
#endif

/* 8. 设置架构特定的IPC状态 */
set_thread_arch_spec_state_ipc(target);

/* 9. 切换到目标线程 */
sched_to_thread(target);

/* 到这里说明完蛋了 */
BUG_ON(1);
}
```

总结一下干了四件事：

- 更新状态为 TS_WAITING
- 设置conn为active（避免并发问题）
- 设置A-stack，寄存器
- 调用 sched_to_thread 切换控制流

切换到服务线程之后便会执行我们的 server_handler，即LRPC的线程迁移技术中提到的"把代码拉过来执行"中被执行的部分，随后就是server端的处理操作了

IPC Return

最后，IPC的服务端在操作完成后会使用 ipc_return 返回，毕竟这时候还是在server中

- ipc_return 会发起 sys_ipc_return 系统调用，该系统调用会迁移到IPC客户端线程继续运行，IPC客户端线程从 ipc_call 中返回
- 也就是正常 return 需要替换成 ipc_return
- 在 server_handler 之中，根据req的信息完成dispatch的工作

```
// ipc server示例: 文件系统handler
DEFINE_SERVER_HANDLER(fsm_dispatch)
{
    int ret = 0;
    struct fsm_request *fsm_req;
    bool ret_with_cap = false;

    if (ipc_msg == NULL) {
        ipc_return(ipc_msg, -EINVAL);
    }
    // ...
}

/* Server uses **ipc_return** to finish an IPC request */
void ipc_return(ipc_msg_t *ipc_msg, long ret)
{
    if (ipc_msg != NULL) {
        ipc_set_msg_return_cap_num(ipc_msg, 0);
    }
    usys_ipc_return((unsigned long)ret, 0);
}
```

层层抽丝剥茧，来到我们最后的 `sys_ipc_return`，这个syscall的源码很长，但是主要是在处理edge case。概括的说它主要干了：

- 果server线程退出，需要回收资源，并设置错误码
- 果client线程退出，需要区分是普通线程还是影子线程（例 client线程也是ipc调用的server, 即链式ipc调用），普通线程应该立刻回收，触发调度（做正常return的工作），影子线程的话就顺着调用链条，只做控制转移，让上层回收
- 能力等其他资源的清理
- 最后简单 `thread_migrate_to_client(client, ret);`；交换控制权，切换线程，完成整个调用，将ret值传回

带注释的源码 下，感兴趣者可作学习参考：

```
/**  
 * @brief IPC返回系统调用，处理服务器完成IPC请求后返回客户端的过程  
 * @param ret 返回给客户端的值  
 * @param cap_num 要传输的capability数量  
 * @return 成功返回0，失败返回负数错误码  
 */  
int sys_ipc_return(unsigned long ret, unsigned int cap_num)  
{  
    struct ipc_server_handler_config *handler_config;  
    struct ipc_connection *conn;  
    struct thread *client;  
  
    /* 获取当前活动的IPC连接 */  
    handler_config = (struct ipc_server_handler_config *)  
        current_thread->general_ipc_config;  
    conn = handler_config->active_conn;  
  
    if (!conn)  
        return -EINVAL;  
  
    /*  
     * 获取发起此IPC请求的客户端线程  
     *  
     * 注意：无需将conn->current_client_thread设置为NULL  
     * i.e., conn->current_client_thread = NULL.  
     */  
    client = conn->current_client_thread;  
  
    /* 步骤1：检查服务器处理线程(current_thread)是否正在退出  
     *      -> 是：设置server_handler_thread为NULL，然后继续步骤2  
     *      -> 否：直接继续步骤2  
     */  
    if (thread_is_exiting(current_thread)) {  
        kdebug("%s:%d Step-1\n", __func__, __LINE__);  
  
        /* 停止接收新的IPC请求 */  
        conn->state = CONN_INCOME_STOPPED;  
  
        /* 标记服务器线程为已退出状态 */  
        thread_set_exited(current_thread);  
  
        /* 返回错误码给客户端 */  
        ret = -ESRCH;  
    }  
  
    /* 步骤2：检查客户端线程是否正在退出  
     *      -> 是：设置current_client_thread为NULL  
     *            然后检查客户端是否为影子线程  
     *                  -> 否：设置客户端为已退出状态并重新调度  
     *                  -> 是：返回给客户端（它将在下一次ipc_return时自行回收）  
     *      -> 否：正常返回  
     */
```

```
if (thread_is_exiting(client)) {
    kdebug("%s:%d Step-2\n", __func__, __LINE__);

    /*
     * 当前假设连接属于客户端进程
     * 因此，果客户端正在退出，则连接也将无效
     * Currently, a connection is assumed to belong to the client
process.
    */
    conn->state = CONN_INCOME_STOPPED;

    /* 果客户端不是影子线程，则直接标记为已退出并重新调度
     * 否则，客户端是链式IPC(A:B:C)中的B，current_thread是C
     * 因此，C返回给B，之后B会返回给A
     */
    if (client->thread_ctx->type != TYPE_SHADOW) {
        kdebug("%s:%d Step-2.0\n", __func__, __LINE__);
        handler_config->active_conn = NULL;

        /* 设置服务器线程状态为等待 */
        thread_set_ts_waiting(current_thread);

        /* 清除调度上下文 */
        current_thread->thread_ctx->sc = NULL;

        /* 释放锁和引用 */
        unlock(&handler_config->ipc_lock);
        unlock(&conn->ownership);
        obj_put(conn);

        /* 标记客户端为已退出并重新调度 */
        thread_set_exited(client);
        sched();
        eret_to_thread(switch_context());
        /* 控制流不会到达这里 */
    }
}

/* 处理capability传输 */
if (cap_num != 0) {
    /* 重置客户端capability缓冲区 */
    for (int i = 0; i < MAX_CAP_TRANSFER; i++) {
        conn->client_cap_buf[i].valid = false;
    }
    /* 从服务器向客户端发送capability */
    int r = ipc_send_cap(current_cap_group,
                          conn->current_client_thread->cap_group,
                          conn->server_cap_buf,
                          conn->client_cap_buf,
                          0,
                          cap_num);
    if (r < 0)
        return r;
}
```

```
/* IPC即将完成，清除活动连接 */
handler_config->active_conn = NULL;

/*
 * 准备返回控制流（调度上下文）
 * 将当前线程状态重新设置为等待
 */
thread_set_ts_waiting(current_thread);

/*
 * 影子线程不应再使用客户端的调度上下文
 * 注意：必须在解锁之前将服务器线程的sc设置为NULL
 * 否则，后续的客户端线程可能会在此操作之前转移其sc
 */
current_thread->thread_ctx->sc = NULL;

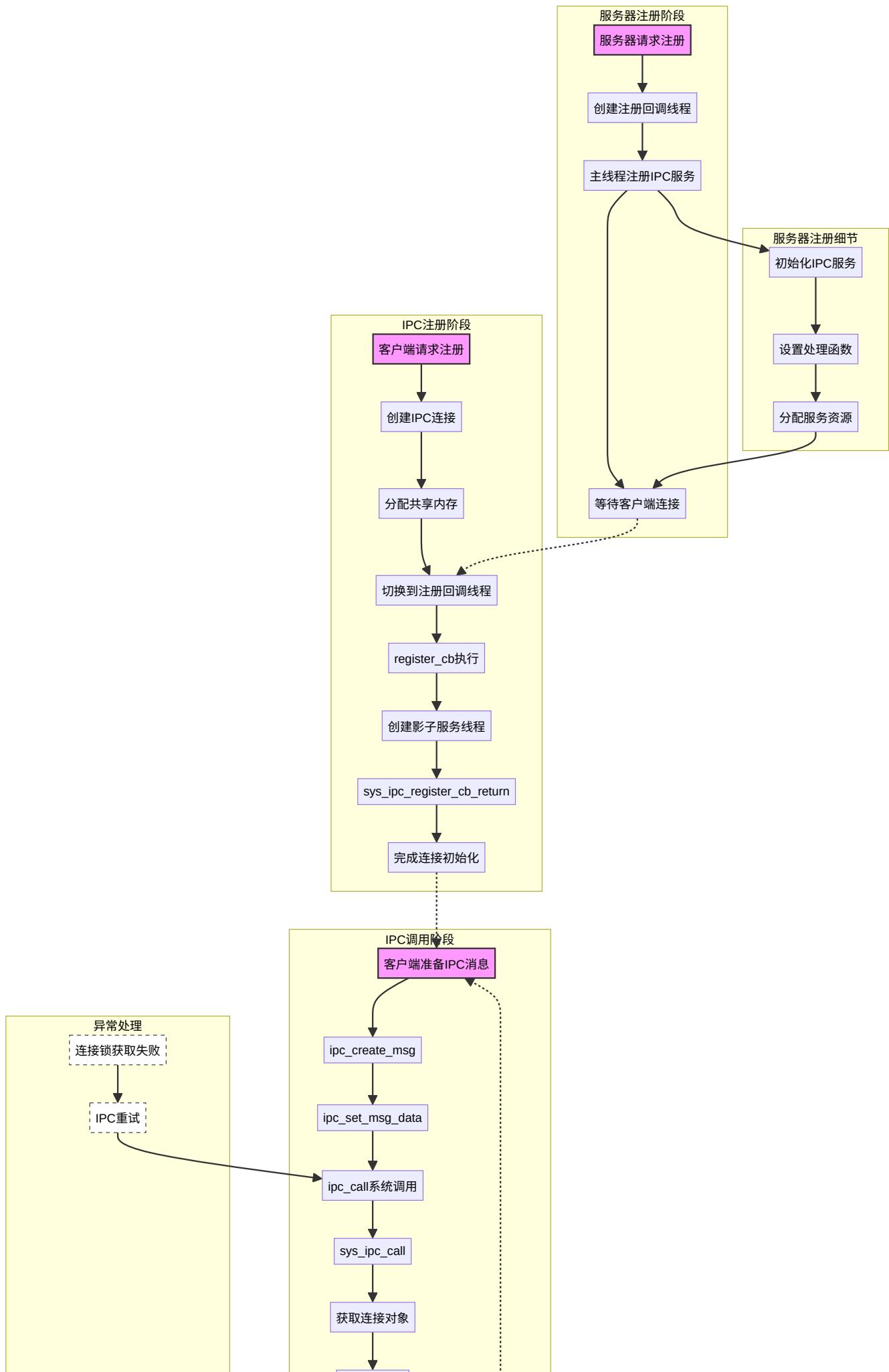
/*
 * 释放IPC锁，表示服务器处理线程可以处理其他请求*/
unlock(&handler_config->ipc_lock);
unlock(&conn->ownership);
obj_put(conn);

/* 返回到客户端 */
thread_migrate_to_client(client, ret);

BUG("should not reach here\n");
__builtin_unreachable();
}
```

这样便成功返回了Client进程，继续做它该做的事情

逻辑流程图



至此，IPC部分的源码解析到此结束，希望能对你的学习有所帮助

Last change: 2025-02-14, commit: [9fa9752](#)

目录

- 1、多核处理器的调度机制

- 运行队列
- 负载均衡与负载追踪
- 处理器亲和性
 - 基本概念
 - 核心机制
 - 系统调用接口
 - 代码示例

- 2、案例分析：Linux调度器

- O(N)调度器
 - 工作原理
 - 效率问题
- O(1)调度器
 - 工作原理
 - 问题
- 完全公平调度器(CFS)
 - CFS的动态时间片
 - CFS使用红黑树作为运行队列
 - CFS阻塞任务唤醒
- Linux的细粒度负载追踪(PELT)
- Linux的NUMA感知调度

- 3、进程间通信

- 进程间通信基础知识
 - 基本概念与设计选择
 - 主要IPC机制
 - 进阶
- 文件接口IPC：管道
 - 管道的核心概念与工作机制
 - Linux中管道进程间通信的实现
 - 命名管道和匿名管道
- 内存接口IPC：共享内存
 - 基础知识
 - 生产者-消费者问题实现
- 消息接口IPC：消息队列
- L4微内核的IPC优化
 - L4消息传递
 - L4控制流转移

- L4通信连接
- L4通信控制
- LRPC的迁移线程模型
 - 核心知识点
 - 与L4直接进程切换的对比
 - LRPC数据传递机制
 - 优势与适用场景
- ChCore进程间通信机制
 - ChCore的IPC用户态实现
 - 内核态系统调用接口
 - 关键机制解析

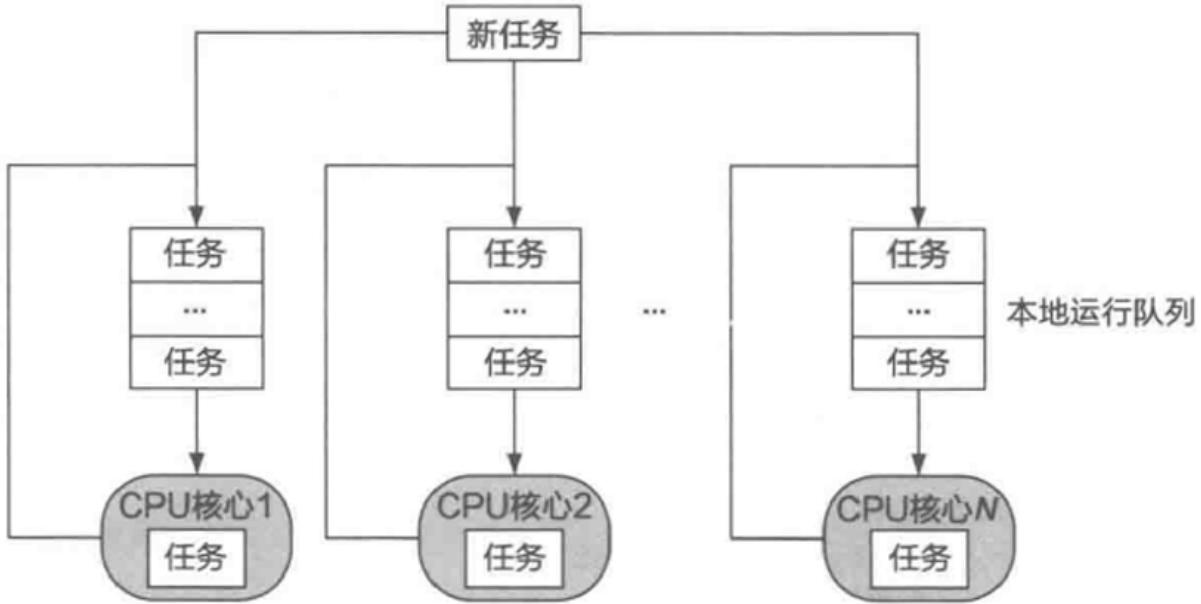
1、多核处理器的调度机制

在之前ICS课程的学习中，我们讨论了单核场景下的处理器调度策略，在该场景下，调度器只需要选择合适的任务让唯一的CPU核心执行即可。然而，在多核场景下，调度器不仅需要选取合适的任务，还需要分配合适的CPU核心用于执行任务。为了支持多核处理器调度，调度器机制也要进行对应的扩展。

1、运行队列

如果让所有CPU核心共享一个全局的运行队列，一方面，多个CPU核心对全局运行队列的访问会产生锁竞争，随着CPU核心数量的上升，任务调度的开销会越来越大；另一方面，系统中的任务会在不同的CPU核心间来回切换，导致无法有效利用每个核心上的高速缓存，并对任务的执行效率产生影响。

为避免上述问题，现代操作系统常用的方式是让每个CPU核心维护一个本地的运行队列。图所示。



当新任务到达系统时，会被分配到某一个 CPU 核心的本地队列，参与该 CPU 核心的单核调度。由于 CPU 核心在调度任务时只需要访问本地数据，且通常情况下任务不会被频繁跨核迁移，因此能够保证高效的任务调度与执行。

调度器通过将任务放到 CPU 核心的本地运行队列，避免了任务在多核间切换的性能开销，因而在多核调度时有良好的性能。然而，如果任务在它的生命周期中只在一个 CPU 核心上运行，则可能导致多核间的负载不均衡，例如某个 CPU 核心的利用率为 100%，而剩余 CPU 核心的利用率为 0%。为了解决这一问题，多核处理器调度必须考虑**负载均衡**，通过追踪每个 CPU 核心当前的负载情况，将高负载 CPU 核心管理的任务迁移到低负载 CPU 核心上，尽可能地保证每个核心的负载大致相同。

2、负载均衡与负载追踪

1. 负载均衡

系统负载的一种定义是简化为每个 CPU 核心本地队列的长度，从而负载均衡的目标是尽可能使所有 CPU 核心的队列长度保持均匀。但不同负载定义可能衍生不同的均衡策略。

2. 负载均衡机制

- **工作窃取 (Work Stealing)**：当某个 CPU 核心的本地队列任务执行完毕时，可从其他核心的队列中“窃取”待执行任务，避免核心空闲，从而实现全局负载均衡。

3、处理器亲和性

1. 基本概念

处理器亲和性允许开发者指定任务（以线程为单位）只能在特定的 CPU 核心上执行。其目的是避免操作系统因负载均衡策略将任务迁移到其他核心，从而满足特定程序的性能需求或资源独占要求。

2. 核心机制

- 通过 `cpu_set_t` 掩码表示允许运行的 CPU 核心集合，掩码中的每一位对应一个核心。
- 操作系统提供以下宏操作 `cpu_set_t`：
 - `CPU_ZERO(set)`：初始化集合为空。
 - `CPU_SET(cpu, set)`：将指定核心加入集合。
 - `CPU_CLR(cpu, set)`：从集合中移除指定核心。
 - `CPU_ISSET(cpu, set)`：检查核心是否在集合中。
 - `CPU_COUNT(set)`：返回集合中的核心数量。

3. 系统调用接口

- `sched_setaffinity(pid, size, mask)`：设置线程的 CPU 亲和性（`pid=0` 表示当前线程）。
- `sched_getaffinity(pid, size, mask)`：获取线程的当前亲和性配置。

4. 代码示例

```
#include <sched.h>
#include <stdio.h>

int main() {
    cpu_set_t mask;
    CPU_ZERO(&mask);           // 初始化空集合
    CPU_SET(0, &mask);         // 添加 CPU 0
    CPU_SET(2, &mask);         // 添加 CPU 2
    sched_setaffinity(0, sizeof(mask), &mask); // 设置当前线程亲和性
    // 后续任务仅会在 CPU 0 和 2 上执行
    ...
    return 0;
}
```

- 操作系统调度时，会检查目标核心是否在亲和性集合内，若不在则禁止迁移。
- 亲和性以线程为粒度控制，需通过线程 ID（`pid`）操作。
- 适用于需要绑定核心的场景（高性能计算、避免缓存失效、资源隔离等）。

2、案例分析：Linux调度器

Linux 调度器需平衡以下核心需求：

1. 公平性：

- 关注任务的等待时间，防止某些任务长时间未被执行。

2. 资源利用率优化：

- I/O 密集型任务：优先调度以提升整体资源利用率（避免 I/O 设备空闲）。
- 计算密集型任务：允许较长执行时间，但需避免独占 CPU。

3. 任务类型优先级：

- 实时任务：必须优先于交互式和批处理任务执行。
- 交互式任务：需快速响应（用户界面操作）。
- 批处理任务：后台任务，允许延迟执行。

4、此外，还需要考虑多核调度相关的因素。

1、O (N) 调度器

工作原理

- **动态优先级计算：** $O(N)$ 调度器在调度决策时，需要遍历运行队列中的所有任务，并重新计算它们的动态优先级，然后选取动态优先级最高的任务执行。
- **时间片分配：** 为了保证公平性，Linux 为非实时任务设置了时间片，避免任务饥饿。早期 Linux 倾向于为任务设置尽可能长的时间片，但过长的时间片会导致任务响应时间过长。因此，调度器将时间分为多个调度时间段（Epoch），每经过一个时间段，调度器会重新分配任务的时间片，避免所有任务全部执行一次的总时间片过长。

效率问题

- **调度开销大：** 随着任务数量的增加， $O(N)$ 调度器的调度开销显著增大，导致调度决策时间过长，浪费 CPU 资源。
- **时间片更新开销：** 在所有任务执行完一个时间片后， $O(N)$ 调度器需要更新它们的时间片，这也会造成额外的调度开销。

2、O (1) 调度器

工作原理

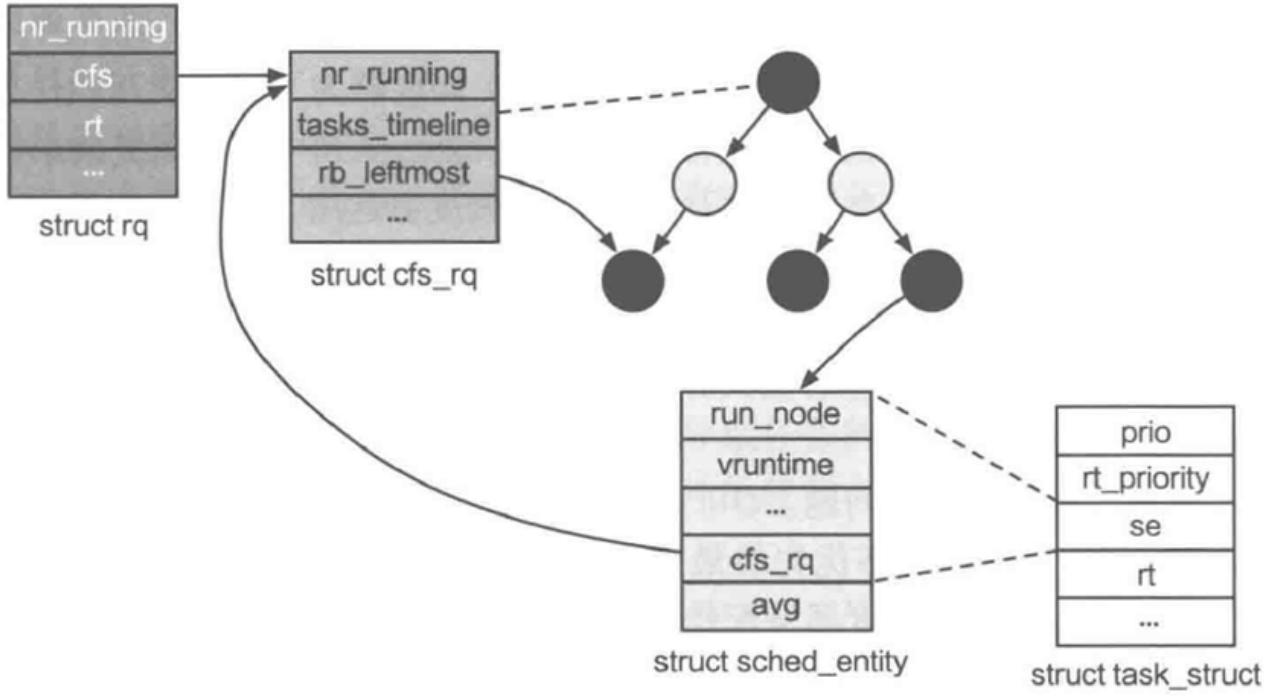
- 通过限制任务优先级范围（实时任务[0, 100]，非实时任务[100, 140]）和使用多级队列结构来实现高效调度。具体来说，运行队列分为**激活队列**和**过期队列**，分别管理有剩余时间片和时间片耗尽的任务。当激活队列为空时，调度器会将过期队列中的任务重新加入激活队列，以确保任务能够及时调度。

问题

- 尽管O(1)调度器在调度开销上表现出色，但它也存在一些问题。首先，交互式任务的判定算法较为复杂，可能导致在特定场景下无法及时响应用户操作。其次，时间片分配机制存在问题，高优先级任务的时间片过长，与实际需求不符。此外，随着任务数量的增加，调度时延也会增加，影响系统的响应时间。

3、完全公平调度器（Completely Fair Scheduler, CFS）

CFS 是 Linux 中默认的调度器，基于公平共享调度策略，确保每个任务根据其分配的份额公平地共享 CPU 时间。CFS 简化了调度器的复杂性，只关注非实时任务的公平共享，避免了复杂的调度算法和调参。它通过动态设置任务的时间片来确保调度延迟不会过高，从而保持系统的响应性。



CFS运行队列示意图

CFS 的运行队列结构中，每个 CPU 核心分配一个运行队列结构（rq），其中 cfs 指针指向 CFS 专用的运行队列实现（cfs_rq）。每个任务由一个任务结构（task_struct）表示，包含一个调

度实体 (`sched_entity`) 数据结构，维护调度任务所需的信息。

CFS 使用的调度策略类似于步幅调度，调度实体中维护了任务的虚拟运行时间 (`vruntime`)，选择虚拟运行时间最短的任务进行调度。CFS 静态设置了非实时任务的静态优先级与任务权重 (`Weight`) 的对应关系，静态优先级越高，任务的权重就越高，可以分配到更多的 CPU 时间。

1、CFS的动态时间片

- **调度周期 (`sched_period`)**：CFS 通过动态调整调度周期解决静态时间片分配问题。默认周期为 6ms，确保每个可运行任务在一个周期内至少执行一次，避免调度延迟过长。
- **时间片权衡：**
 - 周期过长 → 公平性体现延迟，调度开销低；
 - 周期过短 → 调度开销增加，任务切换频繁。
- **最长时间片限制：**当任务数量过多时，调度周期调整为 $\text{任务数} \times 0.75\text{ms}$ ，确保每个任务分得的时间片不低于最小平均值 (0.75ms)，避免因时间片过小导致调度开销激增。
- **权重分配：**根据任务优先级 (权重) 动态调整每个任务在调度周期内的运行时间占比。

2、CFS使用红黑树作为运行队列

CFS 使用红黑树的优势

- **高效维护运行队列：**
 - **插入/删除复杂度：**红黑树为平衡二叉查找树，插入节点复杂度为 $O(\log N)$ ，远优于普通队列的 $O(N)$ 。
 - **快速选取任务：**通过 `rb_leftmost` 指针直接获取虚拟时间最短的任务，调度决策复杂度为 $O(1)$ 。
- **结构设计：**
 - 红黑树仅维护可运行任务 (`cfs_rq`)，减少无效维护开销；
 - 节点关联任务数据结构 (`task_struct`)，支持快速调度操作。

3、CFS阻塞任务唤醒

当任务因阻塞或睡眠未运行时，其虚拟时间不会增加；一旦任务再次进入运行状态，调度器会重新计算该任务的优先级，以避免任务长期占用 CPU 时间。调度器会设置该任务的虚拟时间为该任务当前虚拟时间与运行队列中任务的最小虚拟时间中的较大值，从而确保任务能公平地获取 CPU 时间。

4、Linux的细粒度负载追踪——调度实体粒度负载追踪（PELT）

1. PELT 机制核心

- 细粒度追踪：以调度实体（单个任务）为粒度，记录任务历史执行状态。
- 负载计算：
 - 每 1024 微秒为一个周期，统计任务处于可运行状态的时间（ x 微秒）。
 - 当前周期负载： $L = scale_cpu_capacity \times (x/1024)$
($scale_cpu_capacity$ 为 CPU 处理能力归一化参数)。
- 累计负载计算：
 - 引入衰减系数 y ，通过公式 $L' = L_{\text{旧}} \times y + L_{\text{新}}$ 动态更新累计负载，降低历史数据的权重，确保近期的负载贡献更大。

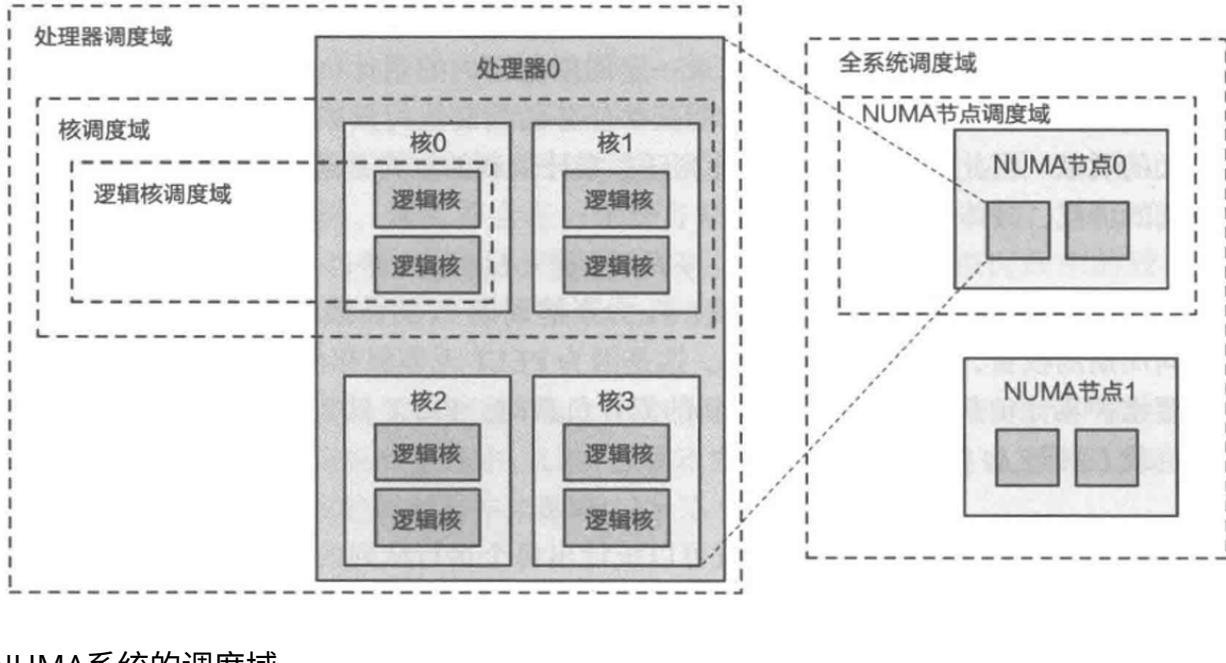
2. 优势

- 开销低（仅需维护累计值，无需存储所有历史数据）。
- 提升负载均衡决策的精确性，帮助调度器选择合适的任务进行迁移。

5、Linux 的 NUMA 感知调度

1. 核心矛盾

- 负载均衡需求与任务本地性（减少跨 NUMA 节点迁移的高开销）之间的权衡。



NUMA系统的调度域

2. 调度域 (Scheduling Domain)

- **分层结构：**根据硬件拓扑将 CPU 划分为多层次调度域（树状结构）：
 - 底层：逻辑核调度域（共享 L1/L2 缓存，迁移开销最低）。

- 向上依次为核调度域、处理器调度域、NUMA 节点调度域、全系统调度域。
- **负载均衡策略：**
 - 越底层域（逻辑核）负载均衡越频繁（迁移开销低）。
 - 高层域（NUMA 节点）负载均衡极少执行（避免高开销跨节点迁移）。

3. 优势

- 在保证一定程度负载均衡的同时，优先利用本地性（缓存亲和性），显著降低任务迁移的开销。

2、进程间通信

1、进程间通信基础知识

1. 基本概念与设计选择

1. 通信方向性：

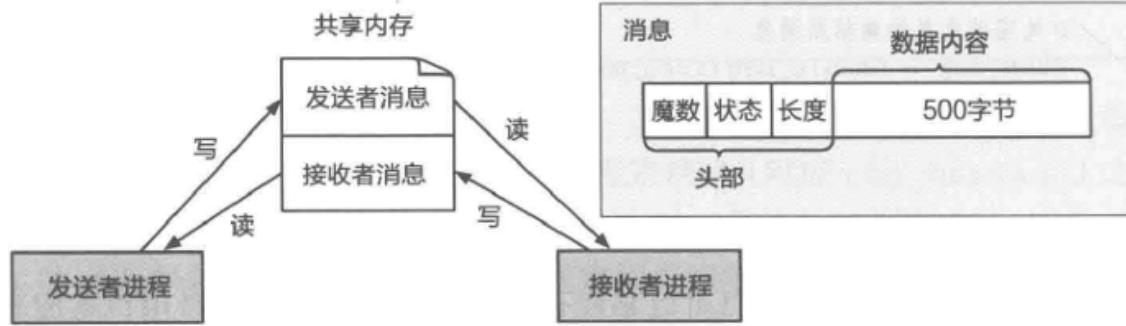
- **单向**：数据仅能单向传输（管道、信号）。
- **双向**：支持双向交互（RPC）。
- **混合模式**：根据配置支持单向或双向（消息队列）。

2. 同步与异步：

- **同步**：发送者阻塞等待操作完成（RPC），适用于简单逻辑。
- **异步**：非阻塞操作，通过回调或轮询获取结果，适用于高并发场景（Android Binder）。

3. 数据传递方式：

- **共享内存**：用户态实现，零拷贝高效，但需内核建立共享区域。



- **操作系统辅助**：需内核介入（管道、消息队列），通过两次内存拷贝完成传输，

但安全性更高。

4. 通知机制：

- **轮询**：简单IPC中采用，但浪费CPU资源。
- **内核唤醒**：通过进程状态切换（阻塞/唤醒）实现高效通知（管道、消息队列）。

2、主要IPC机制

机制	数据抽象	方向	内核介入	特点
管道	字节流	单向	内核态	基于文件接口，匿名管道用于父子进程，命名管道支持多进程。
共享内存	内存区间	双向	用户态	高效但需同步机制（信号量），适用于大数据传输。
消息队列	消息	单向/双向	内核态	支持多进程，按类型组织消息，可缓冲。
信号	事件编号	单向	内核态	轻量级，用于进程控制（终止、挂起）。
信号量	计数器	双向	内核态	用于同步共享资源访问，通过PV操作管理临界区。
Socket	网络流	双向	内核态	支持跨网络通信，基于IP/端口或文件路径寻址。

3、进阶

1. 超时机制：

- **目的**：防止恶意阻塞或长时间等待（DoS攻击）。
- **实现**：允许设置阻塞、立即返回或自定义超时时间（Linux的select/poll）。
- **挑战**：合理超时时间难以确定，需权衡功能与安全性。

2. 通信连接管理：

- **直接通信**：显式标识对方进程（Send(P, msg)），适用于固定进程对。
- **间接通信**：通过中间信箱（管道、消息队列），支持多对多通信。

3. 权限检查：

- **System V IPC**：基于文件权限模型（IPC_PERM结构），检查用户/组权限。
- **微内核**：通过Capability机制控制内核对象访问权限。

4. 命名服务：

- **作用**：全局协调服务注册与发现（文件系统服务注册）。

- **实现：**通常为用户态服务（ROS、Android Binder），支持动态连接分发。

2、文件接口IPC：管道

1、管道（Pipe）的核心概念与工作机制

1. **基本定义** 管道是一种单向的进程间通信（IPC）机制，由内核管理，允许两个进程通过一个通道进行单向数据传输（一端写入数据，另一端读取数据）。

2. **典型应用场景例** 在 `ps aux | grep target` 命令中：

- **| 符号作用：**通过管道将 `ps` 命令的标准输出（`stdout`）连接到 `grep` 的标准输入（`stdin`）。
- **实现原理：**Shell 调用操作系统的 `pipe` 系统调用创建管道，分配读写两端文件描述符，并分别配置到两个进程的输入输出端。

3. **关键特性**

- **单向性：**数据仅从管道写端流向读端，无法反向。
- **字节流传输：**数据以无结构的字节流形式传输，需应用自行解析（分割消息）。
- **内核缓冲区：**内核为管道分配缓冲区暂存数据，平衡两端读写速度差异。
- **两进程协作限制：**每个管道严格绑定两个进程（一个写进程，一个读进程）。

4. **实现细节**

- 通过 Linux 的 `pipe()` 系统调用创建管道，返回读、写端文件描述符。
- Shell 通过重定向命令的输入输出文件描述符（`stdout` → 写端，`stdin` → 读端）实现进程串联。

2、Linux中管道进程间通信的实现

1、管道的创建

管道的创建是由 `pipe` 系统调用完成的，这个系统调用会返回两个文件描述符，对应管道的两端。

```

SYSCALL_DEFINE2(pipe2, int __user *, fildes, int, flags)
{
    struct file *files[2];
    int fd[2];
    int error;

    error = __do_pipe_flags(fd, files, flags);
    if (!error) {
        if (unlikely(copy_to_user(fildes, fd, sizeof(fd)))) {
            fput(files[0]);
            fput(files[1]);
            put_unused_fd(fd[0]);
            put_unused_fd(fd[1]);
            error = -EFAULT;
        } else {
            fd_install(fd[0], files[0]);
            fd_install(fd[1], files[1]);
        }
    }
    return error;
}

```

Linux 内核通过 `__do_pipe_flags` 创建管道的缓冲区，这个缓冲区是通过特殊的文件系统实现的。创建时，会生成两个文件描述符，一个是 `O_RDONLY`，另一个是 `O_WRONLY`，这样就确保了一个是写端，另一个是读端。文件描述符创建成功后，内核会通过 `copy_to_user` 和 `fd_install` 函数将它们返回给用户态程序。用户态程序拿到这两个文件描述符后，就可以通过文件接口来使用它们，实现进程间的通信。那么这个管道真实的存储空间在哪里呢？Linux 中通过 `pipe_inode_info` 这个结构体来管理管道在内核中的信息，代码片段所示。在这个结构体中，内核会维护 `bufs` 的管道缓冲区，由其来保存通信的数据。

```

struct pipe_inode_info {
    struct mutex mutex;           // 保护管道
    wait_queue_head_t rd_wait, wr_wait; // 读者和写者的等待队列
    unsigned int head;           // 缓冲区头
    unsigned int tail;           // 缓冲区尾
    unsigned int readers;         // 并发读者数
    unsigned int writers;         // 并发写者数
    struct pipe_buffer *bufs;     // 管道缓冲区
};

```

在 Linux 中，管道的读写操作和普通的文件读写操作一样。用户程序通过 `read` 和 `write` 系统调用来读写管道内的数据。在 Linux 系统的设计中，这两个系统调用会最终调用到管道实现中注册的文件操作 `pipe_read` 和 `pipe_write`。管道读和管道写的实现十分类似，我们以管道读为例来进行介绍。

```
static ssize_t
pipe_read(struct pipe_inode_info *pipe, struct user_buffer *to)
{
    // 获取用户态的缓冲区大小
    size_t total_len = buffer_count(to);
    ssize_t ret;

    // 果缓冲区大小为 0，直接返回
    if (unlikely(total_len == 0))
        return 0;

    ret = 0;
    // 锁住管道，对应 pipe_inode_info 中的 mutex
    pipe_lock(pipe);

    for (;;) {
        unsigned int head = pipe->head;
        unsigned int tail = pipe->tail;
        unsigned int mask = pipe->ring_size - 1;

        if (!pipe_empty(head, tail)) {
            struct pipe_buffer *buf = &pipe->bufs[tail & mask];
            size_t chars = buf->len;

            if (chars > total_len) {
                chars = total_len;
            }

            copy_page_to_user_buffer(buf->page, buf->offset, chars, to);
            ret += chars;
            buf->offset += chars;
            buf->len -= chars;

            if (!buf->len) {
                release_pipe_buf(pipe, buf);
                tail++;
                pipe->tail = tail;
            }
            total_len -= chars;
            if (!total_len)
                break;
        } else {
            // 没有数据，阻塞等待（或直接返回错误信息）
            ...
        }
    }

    pipe_unlock(pipe); // 释放管道锁
    // ...
}
```

上面这段代码展示了 pipe_read 函数的实现。它首先获取用户态的缓冲区大小，然后检查缓冲区是否为零。果为零，则直接返回。接下来，它锁住管道，进入一个循环，检查管道是否为空。果不为空，它会从管道缓冲区中读取数据，并将其复制到用户态缓冲区。果缓冲区中

的数据被完全读取，它会释放该缓冲区并更新管道的尾指针。如果用户态缓冲区已满，它会退出循环。如果管道为空，它会阻塞等待或返回错误信息。最后，它会释放管道锁。

3、命名管道和匿名管道

在经典的 UNIX 系统中，管道分为命名管道和匿名管道，主要区别在于创建方式。

- 匿名管道通过 `pipe` 系统调用创建，进程会获得两个文件描述符，分别用于读写操作。由于没有全局名称，只能通过这两个文件描述符来使用。通常与 `fork` 结合使用，父进程创建管道后，通过 `fork` 创建子进程，子进程继承文件描述符，完成 IPC 权限的分配。父子进程可以通过管道通信，但需要主动关闭多余的端口，否则可能导致通信错误。这种方式适合父子进程等有创建关系的进程，但对关系较远的进程不太适用。
- 命名管道通过 `mkfifo` 命令创建，指定一个全局的文件名，`/tmp/namedpipe`，这个文件名代表一个具体的管道。只要两个进程通过相同的管道名创建，并且有访问权限，就可以在任意两个进程之间建立通信连接，解决了匿名管道在非父子进程间通信的局限性。

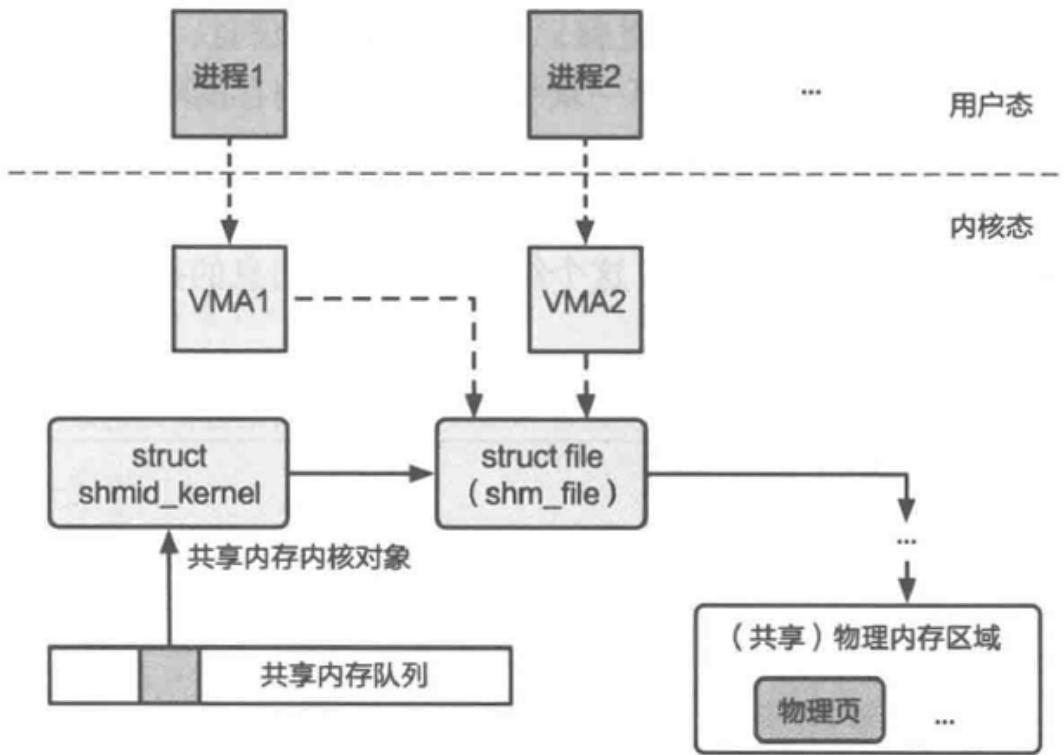
3、内存接口IPC：共享内存

1、基础知识

1. 共享内存的基本原理

- 共享内存允许多个进程将同一物理内存页映射到各自的虚拟地址空间，从而实现高效通信。
- 内核通过全局共享内存队列（`shmid_kernel` 结构体）管理共享内存区域，每个区域与唯一的 IPC key 绑定，权限检查通过后进程可映射（`shmat`）或取消映射（`shmdt`）。

！



2. 内核支持与实现细节

- 共享内存通过文件系统抽象管理物理内存页，支持动态分配（Demand Paging）和交换（Swap）。
- 每个进程映射共享内存时会创建独立的 VMA（Virtual Memory Area），指向同一物理内存，但虚拟地址可不同。
- 取消映射仅影响当前进程，其他进程仍可继续使用共享内存。

3. 性能优势

- 数据直接通过共享内存传输，无需内核参与后续通信，避免了消息队列、管道等机制中的多次数据拷贝和上下文切换。

2、生产者-消费者问题实现

1. 共享数据结构

共享内存中定义了一个环形缓冲区 `buffer`，用于存储生产者生成的消息（`item` 结构体），并通过以下全局变量协调生产者和消费者的操作：

```

#define BUFFER_SIZE 10
typedef struct {
    struct msg_header header;
    char data[0];
} item;

item buffer[BUFFER_SIZE];           // 环形共享缓冲区
volatile int buffer_write_cnt = 0;   // 生产者写入位置索引
volatile int buffer_read_cnt = 0;    // 消费者读取位置索引
volatile int empty_slot = BUFFER_SIZE; // 空闲槽位数
volatile int filled_slot = 0;        // 已填充槽位数

```

- **环形缓冲区**: 通过 `% BUFFER_SIZE` 取模运算实现循环写入/读取。

- **计数器作用**:

- `empty_slot` 和 `filled_slot`: 标识缓冲区的空闲和已占用状态。
 - `buffer_write_cnt` 和 `buffer_read_cnt`: 记录生产者和消费者的操作位置。
-

2. 生产者逻辑

生产者通过 `send` 函数向缓冲区写入消息:

```

int send(item msg) {
    while (empty_slot == 0);           // 忙等待: 直到有空闲槽位
    empty_slot--;                     // 占用一个空闲槽位
    buffer[buffer_write_cnt] = msg;   // 写入消息
    buffer_write_cnt = (buffer_write_cnt + 1) % BUFFER_SIZE; // 环形索引更新
    filled_slot++;                  // 增加已填充槽位
    return 0;
}

```

- **忙等待**: 通过 `while (empty_slot == 0)` 循环检查空闲槽位，若缓冲区满则持续等待。
 - **写入与更新**: 写入后更新 `buffer_write_cnt` (取模实现环形)，并调整 `empty_slot` 和 `filled_slot`。
-

3. 消费者逻辑

消费者通过 `recv` 函数从缓冲区读取消息:

```

item recv(void) {
    while (filled_slot == 0);         // 忙等待: 直到有未处理消息
    filled_slot--;                  // 减少已填充槽位
    item msg = buffer[buffer_read_cnt]; // 读取消息
    buffer_read_cnt = (buffer_read_cnt + 1) % BUFFER_SIZE; // 环形索引更新
    empty_slot++;                  // 释放一个空闲槽位
    return msg;
}

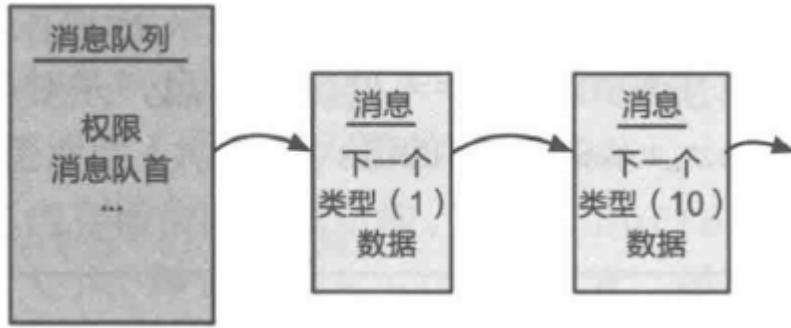
```

4、消息接口IPC：消息队列

1. 消息队列的作用与特点

- 消息队列是内核提供的通信机制，以消息为数据抽象单位，支持多发送者与接收者。
- 相比管道等字节流通信，消息队列更灵活，允许消息按类型分类处理（类型由用户程序定义语义）。
- 适用于需要异步通信或选择性处理消息的场景。

2. 消息队列的结构



- 内核中通过链表实现队列结构，每个消息包含：
 - **类型字段**（用户自定义，内核仅存储和匹配）。
 - **数据字段**（类似管道中的字节流）。
 - 指向下一个消息的指针（链表连接）。
- 消息队列对象包含权限信息、消息头指针等元数据。

3. 基本操作 (System V 消息队列)

- `msgget`：创建或获取消息队列的标识符（类似信箱的地址）。
- `msgsnd`：向队列发送消息（需指定类型和数据）。
- `msgrcv`：从队列接收消息（可指定类型进行过滤）。
- `msgctl`：管理队列（修改权限、删除队列）。

4. 通信特性

- **默认阻塞行为**：发送时队列满或接收时队列空会阻塞进程。
- **非阻塞选项**：通过 `NOWAIT` 标志可避免阻塞，直接返回错误。
- **多进程支持**：任意数量的进程可连接到同一队列（需权限）。

5、L4微内核的IPC优化

在L4微内核中，内核只保留了基本的功能，包括地址空间、线程、进程间通信等，并且不考虑

兼容性等要求，而是选择针对特定硬件做极致的性能优化。这样做的好处是内核的代码量非常少，可以为少量的功能提供尽可能完善的支持。

1、L4消息传递

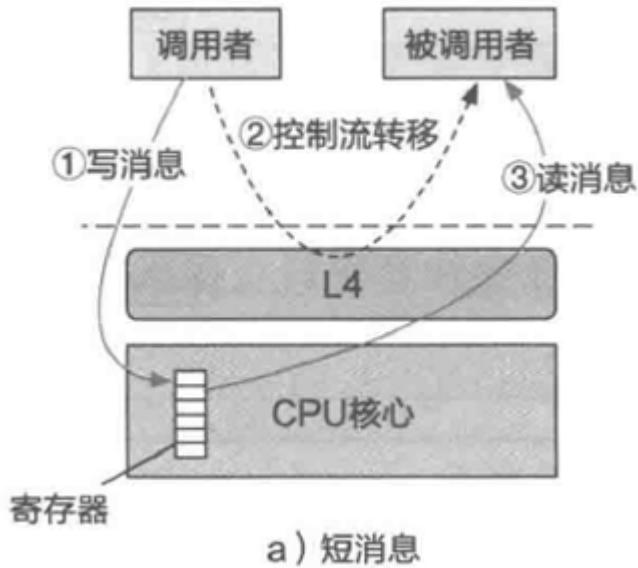
1. L4 的设计目标

- **最小化内核接口：**通过单一通信接口（消息传递）支持丰富语义，函数调用、数据缓冲区、字符串及 Capability 的传递。
- **优化通信开销：**通过减少数据拷贝和上下文切换提升性能。

1. 短消息传输机制

◦ 寄存器直接传递：

- 消息较短时，直接通过**硬件寄存器**实现零拷贝传输。
- **局限性：**数据量受硬件架构限制（x86-32的寄存器数量有限）。



◦ 虚拟消息寄存器 (VMR) :

- **Pistachio 的优化：**将物理寄存器与内存虚拟寄存器解耦，允许自定义虚拟寄存器集合（64字节）。
- **实现方式：**部分虚拟寄存器映射到物理寄存器，其余存于固定地址的内存空间，用户态封装接口隐藏差异。
- **优点：**扩展数据容量，降低移植复杂性，被后续系统（sel4、Fiasco.OC）沿用。

1. 长消息传输机制

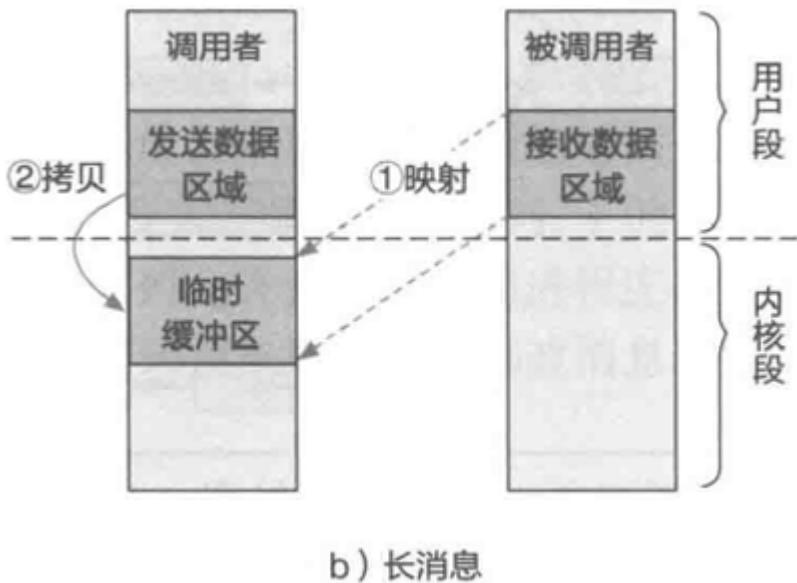
◦ 内核辅助传输：

- 默认需两次拷贝（发送者→内核缓冲区→接收者），但通过优化减少拷贝次

数。

- 优化手段：

- 临时映射区域：内核在进程地址空间预留临时缓冲区，通过虚拟地址映射实现单次拷贝。



- 多缓冲区支持：单次通信可传输多个不连续缓冲区，减少上下文切换开销。

- 兼容性与问题：

- 兼容POSIX等依赖缓冲区的接口，但可能触发缺页异常（需用户态页处理程序介入，增加内核复杂性）。

- 后续改进：

- 采用共享内存替代长消息机制，由进程自行管理数据传输，简化内核逻辑。

2、L4控制流转移

1. 惰性调度 (Lazy Scheduling)

- 背景问题：

同步IPC操作中，线程状态频繁切换（就绪↔阻塞）导致调度队列操作过多（频繁移入/移出），引发缓存/TLB不命中等开销。

- 核心思想：

- 延迟更新调度队列：阻塞线程仍保留在就绪队列，仅更新其TCB（线程控制块）状

态为 `IPC_blocked`，由调度器动态选择跳过阻塞线程。

- **避免队列操作：**减少了线程状态变更时的队列维护开销，通过TCB状态标记动态过滤可运行线程（见伪代码对比）。

- **优势与缺陷：**

减少调度开销：避免线程频繁移入/移出队列的操作。

潜在问题：

- 就绪队列因阻塞线程积累可能增大调度时延。
 - 依赖假设“阻塞状态短暂”成立，不适用于实时性要求高的场景。
-

2. 直接进程切换 (Direct Process Switching)

- **背景问题：**

微内核早期IPC控制流转移依赖调度程序，调度不确定性导致**IPC时延不可控**。

- **核心思想：**

- **取消调度介入：**从调用者到被调用者的控制流切换直接完成，不触发调度程序。
- **同步路径优化：**调用者发送消息后立即切换到接收者，无其他进程干扰（返回过程对称处理）。

- **优势与缺陷：**

降低时延：避免调度路径的中断和上下文切换开销。

提升缓存效率：接收者可直接操作调用者的地址空间数据，减少缓存污染。

缩短内核态时间：减少内核干预，提高系统整体性能。

破坏实时场景下任务的优先级：如果只要调用者发起通信，被调用者就一定响应，那么在一些实时的场景下将无法保证对不同优先级任务的区别处理。

3、L4通信连接

- **早期直接通信：**

- **目标：**以线程为通信目标，避免中间抽象（端口）导致的性能开销（缓存和TLB污染）。
- **问题：**
 - **全局唯一线程 ID：**可能引发隐蔽信道（Covert Channel）风险，导致信息泄露。

▪ **信息隐蔽性差**: 服务需暴露内部线程结构（线程数量和 ID），难以实现负载均衡。

- **转向间接通信**:

- **原因**: 现代硬件支持大页码机制，缓解了 TLB 污染问题。

- **实现**: 采用类似 Mach 的"信箱"或"端口"模型（sel4 和 Fiasco.OC），提升灵活性和安全性。

4、L4通信控制（权限检查）

1. 直接通信的权限问题

- **拒绝服务攻击**: 恶意进程通过大量消息"轰炸"接收者，消耗其资源。

- **早期解决方案：氏族与酋长机制**

- **原理**: 进程按"氏族"层次组织，每个氏族由"酋长"控制消息流向。

- **限制**:

- **性能开销**: 消息需多次重定向，增加 IPC 调用次数。

- **安全漏洞**: 酋长成为攻击目标，可能被利用以阻断通信。

2. 现代权限控制：Capability 机制

- **核心思想**:

- **Capability** 是内核对象的索引和权限凭证（IPC 连接、内存等）。

- 通信需基于有效的 Capability，内核检查权限后建立连接。

- **优势**:

- **安全性**: 阻止未授权通信和拒绝服务攻击。

- **灵活性**: 支持细粒度权限管理（仅允许特定进程通信）。

- **应用**: 现代微内核（sel4）普遍采用 Capability 管理内核对象。

6、LRPC的迁移线程模型

1、核心知识点

1. 迁移线程模型的核心思想

- **优化目标**: 减少传统 IPC 的控制流切换和数据传输开销。

- **关键设计**: 将服务端代码"拉"到客户端线程中执行，避免线程切换和数据拷贝。

- **类似函数调用**: 跨进程调用转化为本地代码执行，仅切换必要状态（页表、栈指针）。

- **内核不参与调度**: 不阻塞调用者线程，无完整上下文切换，避免调度器和优先级切换的开销。

2、与 L4 直接进程切换的对比

特性	L4 直接进程切换	LRPC 迁移线程模型
线程切换	完整上下文切换（寄存器、栈等）	无线程切换，仅切换地址空间等关键状态
性能开销	较高（涉及内核调度和状态保存）	极低（类似函数调用，无调度开销）
多核优化	需跨核通信（IPI 通知）	单核执行，避免跨核通信开销
数据传递	通过寄存器或缓冲区拷贝	共享参数栈，零拷贝传输

3、LRPC 数据传递机制

1. 参数栈与寄存器共享

- **参数栈 (Argument Stack) :**
 - 内核为每个 LRPC 连接预分配，同时映射到客户端和服务端地址空间。
 - 参数通过栈传递，无需内核拷贝（类似函数调用参数准备）。
- **寄存器：**优先使用通用寄存器传递参数，不足时补充参数栈。

2. 通信连接建立

- **服务描述符：**服务端注册处理函数等信息到内核。
- **内核资源分配：**
 - **参数栈：**用于跨进程参数传递。
 - **连接记录 (Linkage Record) :**保存调用者返回地址、栈指针等信息，支持嵌套调用。
 - **绑定对象 (Binding Object) :**客户端通过此对象发起通信，需内核验证权限。

```
//**通信过程源码解析**
int ipc_call (A_stack, ...) {
    verify_binding(A_stack);                                // 验证参数栈合法性
    service_descriptor = get_desc_from_A(A_stack);
    save_ctx_to_linkage_record();                          // 保存调用者上下文到连接记录
    switch_PT();                                         // 切换页表（地址空间）
    switch_sp();                                         // 切换栈指针到服务端运行栈
    ctx_restore_with_args(ret);                           // 恢复上下文并执行服务端代码
}
```

关键步骤：

1. **验证参数栈：**确保通信连接的合法性。

2. **切换状态**: 仅修改页表、栈指针等必要状态，避免完整上下文切换。
 3. **执行服务端代码**: 直接在客户端线程中运行服务端处理函数。
-

4、优势与适用场景

- **优势**:
 - **极低延迟**: 减少内核调度和数据拷贝，性能接近本地函数调用。
 - **高并发支持**: 避免全局共享数据结构，减少锁竞争。
- **适用场景**:
 - 高频同步调用（微服务、分布式系统）。
 - 对延迟敏感的实时系统。

7、ChCore进程间通信机制

在ChCore的通信机制中，消息的传递和通知基于LRPC中的迁移线程技术和L4系列微内核中的直接进程切换技术，而数据的传输则是基于灵活的用户态共享内存。

1、ChCore的IPC用户态实现

ChCore 的 IPC 用户态实现，分为**服务端**和**客户端**两部分：

1. 服务端代码

- **注册服务**:

```
ipc_register_server(ipc_dispatcher);
```

- 服务端调用 `ipc_register_server`，向内核注册服务，核心参数是逻辑处理函数 `ipc_dispatcher`。

- **请求处理**:

```
void ipc_dispatcher(ipc_msg_t *ipc_msg) {  
    char* data = ipc_get_msg_data(ipc_msg); // 获取消息数据  
    ... // 执行业务逻辑  
    ipc_return(ret); // 返回结果给客户端  
}
```

- `ipc_get_msg_data`：从 IPC 消息中提取数据指针（可能是共享内存地址）。
- `ipc_return`：通过系统调用 `sys_ipc_return` 将结果返回客户端。

1. 客户端代码

- **注册客户端连接：**

```
ipc_register_client(server_process_cap, &client_ipc_struct);
```

- `server_process_cap`：通过 Capability 子系统获取服务端的权限标识符。
- `client_ipc_struct`：客户端通信结构体，保存连接信息。

- **发起 IPC 调用：**

```
u64 ret = ipc_call(&client_ipc_struct, ipc_msg);
```

- `ipc_msg`：消息结构体，可传递**数据**和**Capability**（ 共享内存的权限）。

2、内核态系统调用接口

ChCore 的 IPC 核心逻辑通过两个系统调用实现：

1. `sys_ipc_call` （客户端→服务端）

```
u64 sys_ipc_call(u32 conn_cap, ipc_msg_t *ipc_msg) {
    struct ipc_connection *conn = get_connection(conn_cap); // 获取连接对象
    ...
    thread_migrate_to_server(conn, arg); // 控制流迁移
    // 到服务端
    BUG("This function should never return\n"); // 迁移后不再
    // 返回
}
```

- **关键操作：**

- `get_connection`：通过 Capability 验证连接的合法性。
- `thread_migrate_to_server`：将**客户端线程的控制流直接迁移到服务端**，类似 LRPC 的迁移线程模型。
 - **无完整上下文切换**：仅切换页表、栈指针等必要状态，保留寄存器内容。
 - **内核不调度**：避免调度器介入，减少性能开销。

1. sys_ipc_return (服务端→客户端)

```
void sys_ipc_return(u64 ret) {
    struct ipc_connection *conn = get_current_connection();
    ...
    thread_migrate_to_client(conn, ret); // 控制流迁移回客户端
    BUG("This function should never return\n");
}
```

- **关键操作:**

- `thread_migrate_to_client`: 将控制流从服务端迁移回客户端，并携带返回值 `ret`。

3、关键机制解析

1. Capability 子系统

- **作用:** 管理内核对象（IPC 连接、内存区域）的权限。
- **实现:**
 - 客户端通过 `ipc_register_client` 获取服务端的 Capability，建立合法连接。
 - 内核通过验证 Capability 确保通信的安全性（防止未授权访问）。

2. 数据传递与共享内存

- **ipc_msg_t 结构:**

```
typedef struct {
    void *data;          // 数据缓冲区指针
    capability_t cap;   // 可传递的 Capability ( 共享内存权限)
} ipc_msg_t;
```

- **共享内存建立:**

- 服务端和客户端通过传递内存 Capability，将同一物理内存映射到双方地址空间，实现零拷贝数据传输。

3. 控制流迁移 (LRPC 与 L4 的对比)

特性	ChCore	LRPC	L4 直接切换
上下文切换	仅切换页表、栈指针等关键状态	类似，无线程切换	完整线程上下文切换
调度器介入	否	否	是

特性	ChCore	LRPC	L4 直接切换
性能优化	高 (类似函数调用)	高	中
多核支持	单核执行，避免跨核通信	单核执行	需跨核通信 (IPI)

Last change: 2025-02-14, commit: [9fa9752](#)

Lab 4 的实验笔记

目录

- 说明
- 多核调度
 - Linux调度的底层依赖
 - task_struct (任务结构体)
 - runqueue (运行队列)
 - sched_entity (调度实体)
 - 调度器核心操作 (核心调度函数)
 - 时间片与抢占
 - 总结
 - Linux的调度策略及其实现
 - CFS (完全公平调度器)
 - 实时调度策略 (Real-time scheduling)
 - SCHED_IDLE
 - SCHED_BATCH
- 进程间通信 (IPC)
 - 1. 管道 (Pipe)
 - 2. 命名管道 (FIFO)
 - 3. 消息队列 (Message Queue)
 - 4. 共享内存 (Shared Memory)
 - 5. 信号量 (Semaphore)
 - 6. 套接字 (Socket)
 - 7. 信号 (Signal)
- 相关阅读
 - Linux的调度
 - 论文
 - 博客
 - Linux的IPC 机制
 - 论文
 - 博客

说明

本文包括 Linux 调度策略的介绍，Linux IPC机制的介绍，并为读者提供相关阅读材料。读者阅读之后可以思考Linux 与 Chcore 在调度以及IPC 方面的不同之处以及why。

多核调度

Linux调度的底层依赖

Linux调度器是一个基于抢占和时间片轮转机制的复杂调度系统。它的底层依赖主要集中在以下几个关键的数据结构和机制中。

task_struct (任务结构体)

task_struct是Linux内核中表示一个进程或线程的核心数据结构，它包含了进程的所有信息，调度器使用它来决定调度哪些进程或线程执行，跟踪进程的状态，管理进程的优先级、时间片等。例 task_struct不仅包含进程的执行状态（TASK_RUNNING、TASK_SLEEPING），还包含任务优先级、调度实体等信息，调度器利用这些信息来决定哪个任务在何时运行。

```
struct task_struct {  
    ...  
    unsigned int state;           // 任务状态（在调度中判断是否可运行）  
    int priority;                // 任务的优先级（影响调度顺序）  
    int time_slice;              // 分配的时间片长度（用于控制每个任务的执行时间）  
    struct list_head run_list;   // 在运行队列中的链表（调度器用来排序）  
    struct sched_entity se;      // 调度实体，包含虚拟运行时间等调度信息  
    ...  
};
```

相关注释：

state: 进程的当前状态，调度器通过state来判断任务是否可以被调度（例 TASK_RUNNING 表示进程可以调度）。

priority: 进程的优先级，调度器根据优先级来决定任务的调度顺序。

run_list: 链接任务在各个调度队列中的位置，调度器通过它来遍历队列并选择任务执行。

sched_entity: sched_entity包含任务的调度实体，决定任务 何进入调度队列以及 何被调度。

runqueue (运行队列)

runqueue是每个CPU核心的运行队列，用来管理该核心上所有准备执行的任务。Linux调度器通过runqueue来组织和调度任务。

```

struct rq {
    struct list_head queue;           // 就绪队列，保存当前CPU核心上的所有可调度任务
    int nr_running;                  // 当前运行队列中可运行的任务数
    struct task_struct *curr;        // 当前正在运行的任务
    ...
};


```

相关注释：

queue: queue是一个双向链表，用来按优先级或其他策略存储任务。调度器从队列中挑选最合适的任务执行。

nr_running: 记录队列中正在运行的任务数量，调度器可以使用它来确定是否有足够的任务在队列中等待调度。

curr: 指向当前正在运行的任务，调度器需要使用它来了解当前CPU核心正在运行哪个任务。

sched_entity (调度实体)

sched_entity是调度器内部的核心数据结构之一，代表一个任务在调度过程中的实体，它包含了该任务的运行时间、调度权重、时间片等信息。

```

struct sched_entity {
    unsigned int exec_start;          // 任务开始执行的时间
    unsigned int sum_exec_runtime;     // 任务已执行的总时间
    unsigned int period;              // 任务的时间周期
    unsigned int weight;             // 任务的权重（优先级等）
    ...
};


```

相关注释：

exec_start: 任务开始执行的时间戳，用于计算任务的执行时间。

sum_exec_runtime: 任务已执行的累计时间，调度器通过这个信息来判断任务是否需要调度。

weight: 任务的权重，决定任务在调度中的优先级，调度器根据weight来决定任务执行的频率和顺序。

调度器核心操作 (核心调度函数)

调度器的核心函数用于管理任务的调度流程。这些函数通过操作上述数据结构来实现任务的入队、出队、选择执行任务等功能。

schedule(): 这是调度器的核心函数，负责选择下一个任务并执行任务切换。在此函数中，调度器根据当前CPU核心的runqueue选择下一个要执行的任务，并通过switch_to()函数切换到该任务。

```
void schedule(void) {
    struct task_struct *next_task = pick_next_task(); // 选择下一个要执行的任务
    switch_to(next_task); // 切换到选中的任务
}
```

相关注释：

- task_struct: task_struct是Linux内核中表示一个进程或线程的核心数据结构，包含进程的状态、上下文信息、优先级、时间片等。pick_next_task()函数返回的next_task即为下一个要运行的任务，switch_to()函数会利用这个task_struct来进行上下文切换。
- pick_next_task(): 该函数从就绪队列中选择下一个任务，因此schedule()函数依赖于pick_next_task()来进行任务选择。

pick_next_task(): 用于从runqueue中挑选出一个任务，在时间片到期或其他事件发生时进行任务切换。它是任务调度过程的第一步，决定了当前CPU要执行哪个任务。

```
struct task_struct *pick_next_task(struct rq *rq) {
    struct task_struct *next = list_first_entry(&rq->queue, struct task_struct, run_list);
    return next; // 返回队列中第一个任务
}
```

相关注释：

- rq (**runqueue**) : runqueue是每个CPU核心的就绪队列，存储所有可以运行的任务。它是一个结构体，包含一个双向链表queue来存储任务。在pick_next_task()函数中，rq->queue表示当前CPU上所有可调度任务的队列。
- list_first_entry(&rq->queue, struct task_struct, run_list): 该宏从runqueue中的任务链表(queue)获取第一个任务。run_list是task_struct中的链表节点，它指向任务在就绪队列中的位置。

enqueue_task(): 将任务加入到队列中，使任务能够被调度器选中。任务可能是在创建时添加到队列中，或者在任务从阻塞状态恢复时重新加入队列。

```
void enqueue_task(struct rq *rq, struct task_struct *p) {
    list_add_tail(&p->run_list, &rq->queue); // 将任务加入到队列的尾部
}
```

相关注释：

- rq (**runqueue**) : runqueue表示当前CPU的就绪队列，rq->queue是一个双向链表，存储所有当前CPU上可运行的任务。
- list_add_tail(&p->run_list, &rq->queue): 这是Linux内核提供的一个宏，用于将一个任

务 (p) 添加到队列 (rq->queue) 的尾部。run_list是task_struct中的链表节点，它指向任务在就绪队列中的位置。

- task_struct: 每个任务都包含run_list成员，用于在队列中挂载任务。enqueue_task()函数将任务节点 (run_list) 添加到队列的尾部，表示任务已经准备好执行，等待调度。

dequeue_task(): 将一个任务 (p) 从当前CPU的就绪队列中移除，通常在任务完成或阻塞时进行。

```
void dequeue_task(struct rq *rq, struct task_struct *p) {
    list_del(&p->run_list); // 将任务从队列中移除
}
```

相关注释：

- rq (**runqueue**) : rq表示当前CPU的就绪队列，rq->queue是一个双向链表，存储当前CPU上的所有可调度任务。
- list_del(&p->run_list): 这是Linux内核提供的宏，用于从链表中删除一个任务节点。run_list是task_struct中的链表成员，表示任务在就绪队列中的位置。dequeue_task()函数通过这个宏将任务从队列中移除，表示该任务已经不再可调度（执行完毕或被阻塞）。
- task_struct: 每个任务都包含run_list成员，用来在调度队列中定位任务。dequeue_task()会通过list_del()将task_struct中的run_list节点从就绪队列中移除。

时间片与抢占

时间片和抢占是调度器的核心概念，用来确保公平调度和响应实时任务。每个任务被分配一定的时间片，时间片耗尽后任务会被重新调度。

时间片：每个任务都会分配一个时间片，调度器使用它来控制任务的执行时长。

抢占：当一个任务的时间片耗尽时，调度器会抢占任务并调度下一个任务。

scheduler_tick(): 此函数每次时钟中断触发时检查当前任务的时间片，若时间片耗尽则调用schedule()执行任务切换。

```
void scheduler_tick(void) {
    if (--current->time_slice == 0) {
        schedule(); // 时间片耗尽，执行调度
    }
}
```

总结

在Linux调度系统中，底层数据结构（task_struct、runqueue、sched_entity）起到了组织

和管理任务的作用，它们帮助调度器高效地进行任务的选择和切换。调度核心函数（`schedule()`、`pick_next_task()`）操作这些数据结构来实现不同的调度策略，保证操作系统可以公平且高效地管理多任务环境。

Linux的调度策略及其实现

CFS（完全公平调度器）

CFS（Completely Fair Scheduler）是Linux的默认调度策略，旨在提供每个进程公平的CPU时间。CFS采用红黑树来管理就绪队列，每个任务按其“虚拟运行时间”（vruntime）排序。虚拟运行时间是根据任务的优先级和实际运行时间计算出来的，目的是让每个任务获得一个公平的时间片。

关键特性：

- **虚拟运行时间：**每个任务有一个vruntime值，它衡量了任务消耗CPU的时间。CFS会优先选择虚拟运行时间最小的任务。
- **红黑树：**CFS使用红黑树数据结构来管理就绪队列，能够高效地选择和调整下一个执行的任务。
- **公平性：**任务的vruntime会随着运行而增加，vruntime越小的任务越早被调度。

原理：

在CFS中，vruntime的计算公式大致是将任务的实际运行时间和任务的优先级相结合来动态调整任务的调度顺序，具体来说，vruntime随着任务的实际运行时间增加而递增，但任务的优先级会影响该增量的大小，优先级较高的任务（vruntime较小）会被调度得更频繁，而优先级较低的任务则会相对延迟调度，这种方式保证了每个任务按照公平的原则来获得CPU时间，避免了长时间占用CPU的情况，进而提高了系统的响应性和公平性。

CFS使用红黑树来管理就绪队列。任务按照vruntime值存储在红黑树中，红黑树的特点是可以在对数时间内进行插入、删除和查找操作，这对于调度器非常重要，因为调度需要快速找到vruntime最小的任务。

相关代码：

```
// CFS调度器：选择下一个要执行的任务
// 通过从红黑树中选取vruntime最小的任务，来决定下一个执行的任务
struct task_struct *pick_next_task_cfs(struct rq *rq) {
    // 获取红黑树中的第一个节点（即 vruntime 最小的任务）
    struct rb_node *n = rb_first(&rq->cfs_tasks);
    if (!n) return NULL; // 果红黑树为空，返回 NULL
    struct task_struct *next = rb_entry(n, struct task_struct, rb_node);
    return next; // 返回最小 vruntime 的任务
}

// CFS调度器：将任务加入 CFS 队列（红黑树）
// 新任务根据其 vruntime 值被插入到红黑树中的合适位置
void enqueue_task_cfs(struct rq *rq, struct task_struct *p) {
    // 插入任务时使用任务的 vruntime 作为键，红黑树自动按此排序
    struct rb_node **new = &(rq->cfs_tasks.rb_node), *parent = NULL;

    // 寻找插入位置，保持红黑树的顺序
    while (*new) {
        struct task_struct *this = rb_entry(*new, struct task_struct,
rb_node);
        parent = *new;
        // 根据任务的 vruntime 值来决定是插入左子树还是右子树
        if (p->vruntime < this->vruntime)
            new = &(*new)->rb_left;
        else
            new = &(*new)->rb_right;
    }
    // 将新任务的红黑树节点链接到父节点
    rb_link_node(&p->rb_node, parent, new);
    // 将新节点插入红黑树
    rb_insert_color(&p->rb_node, &rq->cfs_tasks);
}

// CFS调度器：将任务从 CFS 队列（红黑树）移除
// 任务不再就绪时，移除其在红黑树中的节点
void dequeue_task_cfs(struct rq *rq, struct task_struct *p) {
    // 从红黑树中删除任务的节点
    rb_erase(&p->rb_node, &rq->cfs_tasks);
}

// CFS调度器：更新任务的虚拟运行时间
// 任务的 vruntime 会根据实际运行时间增加，目的是让运行时间较长的任务，vruntime 增加得更快
void update_task_runtimes(struct task_struct *p) {
    // 更新任务的 vruntime，通常情况下 vruntime 会根据任务的实际运行时间增加
    // 假设每次调度时的时间增量为10
    p->vruntime += 10; // 这里假设每次调度时的时间增量为10
}
```

实时调度策略 (Real-time scheduling)

实时调度策略用于保证某些任务能够按时完成。Linux提供了两个主要的实时调度策略：SCHED_FIFO和SCHED_RR。

- **SCHED_FIFO**: 先进先出调度策略，任务按它们到达的顺序执行，不会被其他任务抢占，直到它们完成或者主动放弃CPU。
- **SCHED_RR**: 轮转调度策略，SCHED_FIFO的变种，任务在获得CPU后运行一个时间片，然后强制切换到下一个任务。

关键特性：

- **SCHED_FIFO**: 实时任务在运行时不会被抢占，直到其时间片用完或任务完成。
- **SCHED_RR**: 实时任务会轮流执行，每个任务有一个固定的时间片。

相关代码：

SCHED_FIFO: 先进先出调度策略:

```
// SCHED_FIFO调度策略: 选择下一个要执行的任务
// 任务按它们到达的顺序执行, 不会被其他任务抢占
struct task_struct *pick_next_task_fifo(struct rq *rq) {
    struct task_struct *next = NULL;

    // 检查队列是否为空
    if (!list_empty(&rq->fifo_tasks)) {
        // 选择队列中的第一个任务, 即最先到达的任务
        next = list_first_entry(&rq->fifo_tasks, struct task_struct,
                               run_list);
    }
    return next;
}

// SCHED_FIFO调度策略: 将任务加入到 FIFO 队列
// 新任务按到达的顺序加入队列, 队列是先进先出的
void enqueue_task_fifo(struct rq *rq, struct task_struct *p) {
    // 将任务加入到队列的尾部, 保持队列的顺序
    list_add_tail(&p->run_list, &rq->fifo_tasks);
}

// SCHED_FIFO调度策略: 将任务从 FIFO 队列中移除
// 当任务执行完成或者被主动放弃时, 将其从队列中移除
void dequeue_task_fifo(struct rq *rq, struct task_struct *p) {
    // 将任务从队列中移除
    list_del(&p->run_list);
}
```

SCHED_RR: 轮转调度策略

```
// SCHED_RR调度策略：选择下一个要执行的任务
// 轮转调度策略，任务在获得CPU后运行一个时间片，然后强制切换到下一个任务
struct task_struct *pick_next_task_rr(struct rq *rq) {
    struct task_struct *next = NULL;

    // 检查队列是否为空
    if (!list_empty(&rq->rr_tasks)) {
        // 选择队列中的第一个任务（轮转调度，首先执行队列最前面的任务）
        next = list_first_entry(&rq->rr_tasks, struct task_struct, run_list);
    }
    return next;
}

// SCHED_RR调度策略：将任务加入到 RR 队列
// 新任务加入队列时，按轮转顺序进行调度
void enqueue_task_rr(struct rq *rq, struct task_struct *p) {
    // 将任务加入到队列的尾部，保证轮转调度的顺序
    list_add_tail(&p->run_list, &rq->rr_tasks);
}

// SCHED_RR调度策略：将任务从 RR 队列中移除
// 任务完成或被主动放弃时，从队列中移除
void dequeue_task_rr(struct rq *rq, struct task_struct *p) {
    // 将任务从队列中移除
    list_del(&p->run_list);
}

// SCHED_RR调度策略：更新任务的时间片
// 每个任务在运行时间片后会被强制切换，重新加入队列等待下次运行
void update_task_rr(struct task_struct *p) {
    // 假设时间片为 10ms，任务运行结束后，时间片重置
    p->time_slice = 10; // 为任务分配一个新的时间片
}
```

SCHED_IDLE

SCHED_IDLE策略是用于CPU空闲时的调度。它的主要作用是当系统空闲时，调度一个低优先级的任务，防止系统空闲过长时间，浪费CPU资源。

关键特性：

- **CPU空闲时运行：**仅在没有其他任务需要执行时，系统才会调度此类任务。
- **最低优先级：**SCHED_IDLE任务的优先级非常低，因此只有在没有其他任务时才会被调度。

相关代码：

```
// SCHED_IDLE调度策略：选择下一个要执行的任务
// 空闲调度策略主要用于处理 CPU 空闲的情况。当系统没有其他高优先级任务时，选择
SCHED_IDLE 任务
struct task_struct *pick_next_task_idle(struct rq *rq) {
    struct task_struct *next = NULL;

    // 果没有其他任务等待执行，选择空闲任务
    if (list_empty(&rq->idle_tasks)) {
        next = NULL; // 果没有任务，则返回空
    } else {
        // 果有空闲任务，选择队列中的第一个空闲任务
        next = list_first_entry(&rq->idle_tasks, struct task_struct,
run_list);
    }

    return next;
}

// SCHED_IDLE调度策略：将任务加入到空闲队列
// 果系统进入空闲状态时，将任务加入到空闲队列中
void enqueue_task_idle(struct rq *rq, struct task_struct *p) {
    // 将任务加入到空闲队列中
    list_add_tail(&p->run_list, &rq->idle_tasks);
}

// SCHED_IDLE调度策略：将任务从空闲队列中移除
// 当任务不再是空闲状态时，将其从队列中移除
void dequeue_task_idle(struct rq *rq, struct task_struct *p) {
    // 将任务从空闲队列中移除
    list_del(&p->run_list);
}

// SCHED_IDLE调度策略：更新任务的运行时间
// 空闲任务的运行时间并不会增加过多，主要目的是避免长时间占用 CPU
void update_task_idle(struct task_struct *p) {
    // 空闲任务的虚拟运行时间不会增加太多，以避免占用 CPU
    p->vruntime += 1; // 空闲任务的虚拟运行时间增加非常少
}
```

SCHED_BATCH

SCHED_BATCH策略用于批量处理任务，通常用于长时间运行的计算密集型任务。与CFS不同，SCHED_BATCH没有精细的时间片管理，任务可以连续运行较长时间。

关键特性：

- **长时间运行：** SCHED_BATCH任务可以长时间占用CPU，不会频繁地进行时间片轮转。
- **不需要高响应性：** 适用于长时间运行的计算任务，对响应时间的要求不高

相关代码：

```
// SCHED_BATCH调度策略：选择下一个要执行的任务
// 批处理任务调度会选择就绪队列中下一个待执行的任务
struct task_struct *pick_next_task_batch(struct rq *rq) {
    struct task_struct *next = NULL;

    // 选择就绪队列中第一个任务作为下一个要执行的任务
    if (!list_empty(&rq->batch_tasks)) {
        next = list_first_entry(&rq->batch_tasks, struct task_struct,
run_list);
    }

    return next;
}

// SCHED_BATCH调度策略：将任务加入到批处理队列
// 当一个任务被标记为批处理任务时，加入批处理队列
void enqueue_task_batch(struct rq *rq, struct task_struct *p) {
    // 将任务加入到批处理任务队列
    list_add_tail(&p->run_list, &rq->batch_tasks);
}

// SCHED_BATCH调度策略：将任务从批处理队列中移除
// 当批处理任务不再需要执行时，移除它
void dequeue_task_batch(struct rq *rq, struct task_struct *p) {
    // 将任务从批处理任务队列中移除
    list_del(&p->run_list);
}

// SCHED_BATCH调度策略：更新任务的虚拟运行时间
// 批处理任务的虚拟运行时间增量比其他任务更慢，以减少其对 CPU 资源的争夺
void update_task_batch(struct task_struct *p) {
    // 批处理任务的 vruntime 增加得相对较慢，减少对 CPU 的占用
    p->vruntime += 10; // 增加较小的虚拟运行时间增量
}
```

注：SCHED_BATCH 调度策略与 SCHED_IDLE 相似，都是为了确保 CPU 资源的合理利用，不过批处理任务通常适用于运行较长时间且对响应时间要求不高的任务。

进程间通信（IPC）

在 Linux 系统中，常见的 IPC（进程间通信）机制包括：

1. 管道（Pipe）
2. 命名管道（FIFO）
3. 消息队列（Message Queue）
4. 共享内存（Shared Memory）
5. 信号量（Semaphore）

6. 套接字 (Socket)

7. 信号 (Signal)

接下来，我将介绍每种 IPC 机制的原理、设计、优化方向、应用场景，以及底层实现和依赖的数据结构。

1. 管道 (Pipe)

原理和设计，优化方向，应用场景

- **原理和设计：**管道是一种半双工的通信方式，数据只能单向流动。通常用于父子进程之间的通信。管道是基于内核的缓冲区实现的。
- **优化方向：**可以通过增加缓冲区大小来提高数据传输效率，或者使用多个管道实现双向通信。
- **应用场景：**常用于 shell 命令中的管道操作，`ls | grep "txt"`。

底层实现和底层依赖

- **底层实现：**管道是通过 `pipe()` 系统调用创建的，内核会维护一个环形缓冲区。
- **底层依赖：**依赖于文件描述符和内核缓冲区。

相关代码：

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main() {
    int fd[2];
    pid_t pid;
    char buf[128];

    // 创建管道
    if (pipe(fd) == -1) {
        perror("pipe");
        return 1;
    }

    // 创建子进程
    pid = fork();
    if (pid < 0) {
        perror("fork");
        return 1;
    }

    if (pid == 0) { // 子进程
        close(fd[1]); // 关闭写端
        read(fd[0], buf, sizeof(buf)); // 从管道读取数据
        printf("Child received: %s\n", buf);
        close(fd[0]);
    } else { // 父进程
        close(fd[0]); // 关闭读端
        const char *msg = "Hello from parent!";
        write(fd[1], msg, strlen(msg) + 1); // 向管道写入数据
        close(fd[1]);
    }

    return 0;
}
```

2. 命名管道 (FIFO)

原理和设计，优化方向，应用场景

- **原理和设计：**命名管道是一种特殊的文件，允许无亲缘关系的进程通过文件系统路径进行通信。
- **优化方向：**可以通过调整缓冲区大小或使用非阻塞模式来提高性能。
- **应用场景：**适用于需要持久化通信的场景，日志收集。

底层实现和底层依赖

- **底层实现：**通过 `mkfifo()` 系统调用创建，内核维护一个 FIFO 队列。
- **底层依赖：**依赖于文件系统和内核缓冲区。

相关代码：

```
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int main() {
    const char *fifo_path = "/tmp/my_fifo";

    // 创建命名管道
    mkfifo(fifo_path, 0666);

    int fd = open(fifo_path, O_WRONLY); // 打开写端
    const char *msg = "Hello from writer!";
    write(fd, msg, strlen(msg) + 1); // 写入数据
    close(fd);

    return 0;
}
```

3. 消息队列 (Message Queue)

原理和设计，优化方向，应用场景

- **原理和设计：**消息队列是一个消息的链表，允许进程通过消息类型进行通信。
- **优化方向：**可以通过调整消息大小和队列长度来优化性能。
- **应用场景：**适用于需要按优先级处理消息的场景，任务调度。

底层实现和底层依赖

- **底层实现：**通过 `msgget()`、`msgsnd()` 和 `msgrcv()` 系统调用实现。
- **底层依赖：**依赖于内核维护的消息队列数据结构。

相关代码：

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>

struct msg_buffer {
    long msg_type;
    char msg_text[128];
};

int main() {
    key_t key = ftok("msg_queue", 65);
    int msgid = msgget(key, 0666 | IPC_CREAT);

    struct msg_buffer message;
    message.msg_type = 1;
    strcpy(message.msg_text, "Hello from sender!");

    // 发送消息
    msgsnd(msgid, &message, sizeof(message), 0);

    // 接收消息
    msgrcv(msgid, &message, sizeof(message), 1, 0);
    printf("Received: %s\n", message.msg_text);

    // 删除消息队列
    msgctl(msgid, IPC_RMID, NULL);

    return 0;
}
```

4. 共享内存 (Shared Memory)

原理和设计，优化方向，应用场景

- **原理和设计：**共享内存允许多个进程共享同一块内存区域，是最高效的 IPC 方式。
- **优化方向：**可以通过调整共享内存大小和使用同步机制（信号量）来优化。
- **应用场景：**适用于需要高效传输大量数据的场景，数据库系统。

底层实现和底层依赖

- **底层实现：**通过 `shmget()`、`shmat()` 和 `shmdt()` 系统调用实现。
- **底层依赖：**依赖于内核维护的共享内存段。

相关代码：

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

int main() {
    key_t key = ftok("shm_file", 65);
    int shmid = shmget(key, 1024, 0666 | IPC_CREAT);

    // 附加共享内存
    char *str = (char *)shmat(shmid, (void *)0, 0);

    // 写入数据
    strcpy(str, "Hello from writer!");

    // 读取数据
    printf("Data read from memory: %s\n", str);

    // 分离共享内存
    shmdt(str);

    // 删除共享内存
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

5. 信号量 (Semaphore)

原理和设计，优化方向，应用场景

- **原理和设计：**信号量用于进程间的同步，通常用于控制对共享资源的访问。
- **优化方向：**可以通过调整信号量的初始值和操作方式（非阻塞）来优化。
- **应用场景：**适用于需要同步的场景，生产者-消费者问题。

底层实现和底层依赖

- **底层实现：**通过 `semget()`、`semop()` 和 `semctl()` 系统调用实现。
- **底层依赖：**依赖于内核维护的信号量集合。

相关代码：

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <unistd.h>

union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
};

int main() {
    key_t key = ftok("sem_file", 65);
    int semid = semget(key, 1, 0666 | IPC_CREAT);

    union semun arg;
    arg.val = 1; // 初始化信号量值为 1
    semctl(semid, 0, SETVAL, arg);

    struct sembuf sb = {0, -1, 0}; // P 操作
    semop(semid, &sb, 1);

    printf("Critical section\n");

    sb.sem_op = 1; // V 操作
    semop(semid, &sb, 1);

    // 删除信号量
    semctl(semid, 0, IPC_RMID, arg);

    return 0;
}
```

6. 套接字 (Socket)

原理和设计，优化方向，应用场景

- **原理和设计：**套接字是一种网络通信机制，支持不同主机间的进程通信。
- **优化方向：**可以通过调整缓冲区大小和使用非阻塞模式来优化。
- **应用场景：**适用于网络通信，Web 服务器和客户端。

底层实现和底层依赖

- **底层实现：**通过 `socket()`、`bind()`、`listen()` 和 `accept()` 等系统调用实现。
- **底层依赖：**依赖于网络协议栈和内核缓冲区。

相关代码：

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    char buffer[1024] = {0};
    const char *hello = "Hello from server";

    // 创建套接字
    server_fd = socket(AF_INET, SOCK_STREAM, 0);

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(8080);

    // 绑定套接字
    bind(server_fd, (struct sockaddr *)&address, sizeof(address));

    // 监听
    listen(server_fd, 3);

    // 接受连接
    new_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t *)&addrlen);

    // 发送数据
    send(new_socket, hello, strlen(hello), 0);
    printf("Hello message sent\n");

    // 关闭套接字
    close(new_socket);
    close(server_fd);

    return 0;
}
```

7. 信号 (Signal)

原理和设计，优化方向，应用场景

- **原理和设计：**信号是一种异步通信机制，用于通知进程发生了某个事件。

- **优化方向：**可以通过合理设计信号处理函数来避免竞态条件。
- **应用场景：**适用于进程控制，终止进程或处理异常。

底层实现和底层依赖

- **底层实现：**通过 `signal()` 或 `sigaction()` 系统调用实现。
- **底层依赖：**依赖于内核的信号处理机制。

相关代码：

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void signal_handler(int signum) {
    printf("Received signal %d\n", signum);
}

int main() {
    // 注册信号处理函数
    signal(SIGINT, signal_handler);

    while (1) {
        printf("Running...\n");
        sleep(1);
    }

    return 0;
}
```

相关阅读

Linux的调度

论文

"The Completely Fair Scheduler" - Ingo Molnar

<https://dl.acm.org/doi/fullHtml/10.5555/1594371.1594375>

博客

<https://www.geeksforgeeks.org/cpu-scheduling-in-operating-systems/>

<https://tldp.org/LDP/tlk/kernel/processes.html>

Linux的IPC 机制

论文

"Communications in Operating Systems" - William Stallings

<https://dl.acm.org/doi/pdf/10.1145/359340.359342>

博客

<https://www.geeksforgeeks.org/inter-process-communication-ipc/>

<https://tldp.org/LDP/lpg/node7.html>

Last change: 2025-02-14, commit: [9fa9752](#)

Last change: 2025-02-14, commit: [23a4605](#)

附录

Last change: 2024-08-17, commit: [5b84259](#)

工具链教程

本教程会对实验中所需使用的一些命令行工具进行介绍，包括为什么要使用这些工具、宏观功能介绍以及部分工具的详细介绍，对于感兴趣的同学，我们也列出了一些更详细的资料供进一步学习。本教程中介绍的工具仅代表我们推荐的，在拆弹和后续ChCore实验中的常用命令行工具，但我们不限定同学们在完成实验时所使用的具体工具或方式，同学们可以使用任何其他自己熟悉的工具来完成实验。本教程中所介绍的工具和命令，均假设在Linux shell环境（bash等）下执行。如果同学们希望使用其他操作系统或平台，请自行查阅相关工具的安装方法以及可能的命令语法不同之处。

Last change: 2024-08-17, commit: [5b84259](#)

TL;DR Cheatsheet

Info

<参数> 代表需要被替换的参数，请将其替换为你需要的实际值（包括左右两侧尖括号）。

- [tmux](#)
- [gdb](#)
- [objdump操作](#)

tmux

- 创建新会话： tmux new -s <会话名>
- 进入会话： tmux attach -t <会话名>
- 临时退出会话（会话中程序保持在后台继续运行）： Ctrl-b d
- 关闭会话及其中所有程序： tmux kill-session -t <会话名>
- 水平分屏： Ctrl-b "
- 垂直分屏： Ctrl-b %

gdb

- 不输入任何命令，直接按下回车键：重复上一条命令
- break *address : 在地址 address 处打断点
- break <函数名> : 在函数入口处打断点
- continue : 触发断点后恢复执行
- info breakpoints : 列出所有断点
- delete : 删除编号为 NUM 的断点
- stepi : 触发断点后单步执行一条指令
- print : 打印表达式的求值结果，可以使用部分C语法，例 print*(int*) 0x1234 可将地址 0x1234 开始存储的4个字节按32位有符号整数解释输出
- print/x : 以16进制打印 的求值结果
- lay asm : 使用TUI汇编视图
- lay src : 使用TUI源代码视图（要求被调试可执行文件包含调试信息，且本地具有相应源代码文件）
- tui disable : 退出TUI

objdump操作

- objdump -dS <可执行文件> > <输出文件> : 反汇编可执行文件中的可执行section（不含数据）, 并保存到输出文件中。在可能的情况下（有调试信息和源文件），还会输出汇
- 编指令所对应的源代码。
- objdump -dsS <可执行文件> > <输出文件> : 将可执行文件中的所有sections的内容全部导出，但仍然只反汇编可执行sections，且在可能情况下输出源代码

Last change: 2024-08-17, commit: [5b84259](#)

TMUX

- 简介
- tmux vs 多个终端模拟器窗口
- Session/Window/Pane
 - Pane
 - Window
 - Session
 - 常用快捷键与子命令
- 拓展阅读

简介

今大部分同学开始使用电脑接触的就是Windows与GUI（图形用户界面）。在GUI下，我们可以通过桌面和窗口的方式，管理显示器上的二维空间（桌面空间）。许多情况下，桌面空间对于单个应用程序来说，有些太大了，只使用一个应用程序不能有效地利用显示空间；又或者，我们想要同时处理多个任务，比 在写文档或者论文时，希望能非常方便地看到自己的参考资料.....部分同学可能已经掌握 何使用分屏以及虚拟桌面来解决这些问题。的确，不论 Windows10/11还是macOS抑或是Linux下的桌面环境，甚至众多基于Android的操作系统，分屏与虚拟桌面已经成为构建高效的GUI工作环境的基本功能。分屏解决的是 何有效利用桌面空间的问题。通过分屏功能的辅助，我们可以快速将多个应用程序的窗口紧密排列在桌面空间上，且可以更方便地调整这些窗口所占据的区域，避免自己手动排列带来的混乱和不便，让我们可以同时使用多个相关的应用程序，最大化利用桌面空间与提高效率。

虚拟桌面则解决的是 何有效隔离多种使用场景的问题，它是分屏功能的进一步衍生。我们经常需要同时处理几种不同性质的任务，例 ，一边在编写文档或论文，一边还可能需要看QQ、微信和回复消息。 果只有一个桌面空间，要么不时切出和最小化QQ微信的窗口，要么把它们也使用分屏与文档窗口排列在一起，但这两种使用方式，或多或少都会影响需要专注的文档编写任务。此时，我们可以使用虚拟桌面。虚拟桌面是在物理桌面空间上“虚拟”出多个互不相干的桌面空间，每个桌面空间内都可以有自己的窗口布局。虽然同时只能使用一个虚拟桌面，但我们在多个虚拟桌面间快速切换。使用虚拟桌面后，我们可以将比较相关的一类程序的窗口放在同一个虚拟桌面中，其余不相干的程序则放在其他虚拟桌面中， 此，可以有效减少其他程序对于当前工作任务的干扰，同时又能在多种不同工作环境中快速切换。

分屏与虚拟桌面有效提高了GUI下的窗口管理效率。但是，窗口和桌面的概念，并非只能局限于GUI中。利用除了字母数字外的各种字符和颜色，我们同样可以在命令行用户界面（CLI）下“绘制”窗口，相较于通过命令行参数，窗口这种交互方式对用户更友好，更直观。同样地，在CLI下的窗口中，分屏和虚拟桌面需要解决的这些效率问题同样是存在的，也一样有着解决这些问题的需求。tmux (terminal multiplexer) 项目则是目前在CLI环境下这些问题的主要解决方案。顾名思义，它是一个“终端多路复用器”， 果说分屏和虚拟桌面是有效利用GUI中的桌面空间，tmux则主要是有效利用终端中的空间。这里的终端，可以是GUI下的终端模拟器，比

Windows Terminal, iTerm2等，也可以是运行在命令行模式下的Linux的显示器空间等等。

tmux vs 多个终端模拟器窗口

- 同一窗口内部布局自由构建（部分终端模拟器也可实现）
- 统一管理多个窗口、便捷切换（开多个终端容易混淆，不利于随意切换）
- tmux可以在不使用时将进程保持在后台继续运行（detach，而终端模拟器一旦关闭就会杀死其中所有进程）
- tmux还是一个服务器，可以通过网络连接，如果有需要，可以允许其他人通过网络连接到你的
- tmux界面中，实现网络协作（终端模拟器不支持）
- tmux支持高度自定义的配置，且有丰富的插件生态

Session/Window/Pane

Note

在理解了GUI下为什么需要有分屏和虚拟桌面后，类比GUI下的概念，可以很容易地理解tmux中的相关概念。

Pane

pane相当于GUI下的一个窗口。只不过相较于GUI下窗口可以自行自由移动，也可以使用分屏辅助排列，CLI下窗口还是基于字符的，所以tmux下的pane只能实现类似于GUI分屏的紧密排列，不能自由移动，也不能实现pane之间的重叠。

Window

window则相当于GUI下的虚拟桌面。一个window是一组pane的集合，不同的window拥有独立的pane以及pane的布局，且可以在多个window间通过快捷键快速切换。

Session

session是tmux特有的概念，它是一组window的集合，代表一个完整的工作环境。一般来说，不论我们通过显示器、Linux桌面环境下的终端模拟器、Windows或macOS上的终端模拟器+ssh.....等方式访问Linux命令行，首先都是进入一个shell中，而并不能直接进入tmux。因此，在tmux中，相较于“打开”和“关闭”tmux，我们更常说“attach”、“detach”到session。

attach指的是从当前运行的shell进入一个tmux session的过程，而detach则是从tmux session离开，回到单个shell的过程。相较于“打开”和“关闭”，session+attach/detach有下列好处：

- 在同一个终端中，也可以方便切换不同工作环境（这是最基础的功能）
- 避免同一个session中有过多不相干的window，降低切换效率
- detach不是关闭session，detach后，session中运行的所有程序会在后台继续运行，且tmux会负责收集它们产生的输出，我们随时可以重新attach到某一个session，就可以看到其中程序最新的运行情况和历史输出，这对于某些需要长时间后台运行的任务是非常方便的。
- 可以有多个终端同时attach到一个session，搭配tmux的服务器功能，可以实现向他人共享你的工作界面和环境，以及协作工作。

```
The following i386/x86-64 specific disassembler options are supported for use with the -M switch (multiple options should be separated by commas):
x86-64      Disassemble in 64bit mode
i386        Disassemble in 32bit mode
i8086       Disassemble in 16bit mode
att         Display instruction in AT&T syntax
intel        Display instruction in Intel syntax
att-mnemonic          Display instruction in AT&T mnemonic
intel-mnemonic        Display instruction in Intel mnemonic
addr64       Assume 64bit address size
addr32       Assume 32bit address size
addr16       Assume 16bit address size
data32       Assume 32bit data size
data16       Assume 16bit data size
suffix      Always display instruction suffix in AT&T syntax
ax
amd64       Display instruction in AMD64 ISA
intel64     Display instruction in Intel64 ISA

Options supported for -P/--private switch:
For PE files:
  header      Display the file header
  sections    Display the section headers
Report bugs to <https://bugs.archlinux.org/>.
→ bomb-lab git:(master)
[bomb-lab]1 zsh* 2:zsh- window
Continuum status: 5
```

常用快捷键与子命令

tmux支持丰富的快捷键，同时，也可以通过tmux命令的一组子命令来与tmux进行交互。事实上，如果研究tmux的配置文件，可以看到其中重要的一项内容就是将快捷键绑定到相应的子命令上。

- 前缀(prefix table)快捷键与根(root table)快捷键
 - 许多CLI应用程序都会定义自己的快捷键。作为更底层的程序，tmux需要尽可能避免自己的快
 - 捷键和上层应用程序的快捷键冲突。前缀键就是为了解决这个问题引入的。
 - 前缀快捷键：需要先按下前缀键（默认是 Ctrl+b），然后再按下相应的快捷键。
 - Root快捷键：直接按下对应的快捷键，不需要前缀键。这些通常是一些不与通常使用的终端快捷键冲突的键。
- 命令行模式

- <prefix> : , 可以进入tmux的内部命令行。在该命令行中可以直接输入tmux子命令 (而不是 tmux <子命令>) , 可以用于使用未绑定快捷键的命令。

- pane 管理

- 切换pane: <prefix> 方向键, tmux命令版本是 select-pane -[UDLR] , U代表上, D代表下, L代表左, R代表右。同时只能有一个pane接收输入, 利用这个命令可以切换当前接收输入的pane。分割当前pane, 创建两个新的pane: <prefix>" (上下分割), <prefix>% (左右分割), tmux命令版本是 split-window (默认是上下分割), split-window -h (左右分割)。在初始情况下, 每个window只有一个占满整个window的pane, 使用这个命令可以分割当前接收输入的pane, 创建两个pane, 类似于GUI下的分屏功能。
- 关闭pane: <prefix> x , 然后按 y 确认。tmux命令版本是 kill-pane 。

- window管理

- 创建Window: <prefix>c , tmux命令版本是 new-window 。
- 切换Window: <prefix>n (下一个Window), <prefix> p (上一个Window), <prefix> l (最后一个活跃Window), tmux命令版本是 next-window , previous-window , last-window 。
- 切换到指定Window: <prefix> 窗口序号 , tmux命令版本是 select-window -t : 窗口序号。

- session管理 (部分命令没有快捷键, 因为不是在session内部进行操作)

- 创建新session: tmux new-session -s mysession , mysession 是新session的名字。
 - 列出所有session: tmux list-sessions 。
 - attach到一个session: tmux attach -t mysession , mysession 是目标session的名字。
 - detach当前session: <prefix>d , tmux命令版本是 detach 。
 - 终止某个session: tmux kill-session -t mysession , mysession 是你想要终止的 session的名字。
- 在某些GUI终端模拟器中, 还可以通过鼠标与tmux交互, 例 可以通过鼠标拖拽pane的边界来调整各个pane的大小, 点击window的名字来切换到指定window等。

拓展阅读

- <http://man.openbsd.org/OpenBSD-current/man1/tmux.1>
- <https://github.com/rothgar/awesome-tmux>
- <https://github.com/gpakosz/.tmux>
- 如果你希望通过tmux提高自己的工作效率, 我们强烈建议你编写适合自己使用的

tmux配置文件。

Last change: 2024-08-17, commit: [5b84259](#)

GDB

gdb是目前最常用的动态调试工具之一。所谓动态调试，指的是在程序运行的过程中对程序进行观测或施加干预的过程，一种常见的动态调试方法是断点，通过插入断点使得程序在特定点位暂停运行，让动态调试器可以对程序的状态进行进一步的观测。与动态调试相对的是静态分析。静态分析不需要实际运行程序，甚至不需要编译程序，而是在程序源代码或指令的层级进行一系列分析甚至“枚举”，预测程序的可能执行情况，寻找潜在的问题。某种程度上来说，对于存在问题的程序，程序员直接阅读代码或汇编指令并分析问题，也可以被视为一种静态分析。

相较于静态分析，动态调试可以真实地反映程序运行的实际情况，包括各种数据的实际值、程序的实际执行路径等。在某些场景下，使用动态调试寻找程序的问题或理解程序的行为，比直接阅读程序源码要简单许多，例 使用动态调试器单步运行程序，在每一步运行的前后观察程序的相关状态，可以非常直观地找到导致问题的程序指令或代码。当然，静态分析也有着广泛的应用，许多静态分析工具在无需编译或运行程序的情况下便可分析程序的潜在问题，这可以有效节约程序运行的时间，同时静态分析可以尽可能地枚举程序的可能执行路径，有助于发现实际运行程序时不会出现或非常罕见的问题。

源码级调试 vs 汇编级调试

动态调试是在程序的运行过程中施加干预和观测状态，问题在于， 何干预程序的运行？又该观测程序的什么状态？以打一个断点为例，应该在什么地方打断点，程序触发断点后，又该检查程序的哪些状态？动态调试器本身只是为程序员提供了完成上述工作的能力，但 何运用这些能力，仍然需要程序员本身对于程序的了解。站在程序员的角度，无疑希望能直接在程序执行到某一行源代码时触发断点，触发断点后，又可以直接检查程序中某个变量的值甚至复杂对象的内容。许多同学此前可能已经接触过各种IDE自带的调试功能，它们大多都允许程序员在源代码中设置断点，并且可以在触发断点时直接看到各个变量和对象的内容。这种程序员直接站在源代码的层级，使用源代码级的概念（代码行、变量、对象等）进行调试的过程称为源码级调试。

然而，从动态调试器的功能而言，要支持源码级调试并非仅有动态调试器即可做到。这是因为，CPU本身只能执行二进制形式的机器指令，不论是编译执行或是解释执行的高级编程语言，最终在程序运行时，动态调试器能观察和控制的只是最终的机器指令、寄存器和内存地址。例 ，仅使用动态调试器本身，我们只能指定在某一条指令暂停执行，也不能直接检查变量或者对象的内容，因为在机器指令的层面并没有变量和对象的概念，只有寄存器和内存。这种只使用机器执行过程中直接可见的概念（指令、寄存器、内存）进行调试的过程称为汇编级调试/机器级调试。造成上述问题的原因是，在从源代码到可执行程序的编译或解释过程中，许多信息都丢失了，因为这些信息对于程序的最终执行并无任何帮助：从程序执行的角度来说，CPU不需要理解某条指令对应源代码中的哪个文件的哪一行，也不需要理解某个寄存器在某一时刻存储的是哪个变量的值。但是，这些信息的丢失，就给调试带来了较大的困难，因为高级语言翻译成汇编指令的方式非常多样，并且存在各种复杂的细节，从而使得汇编级调试并不直观，往往需要远多于源码级调试的时间精力才有可能定位和理解程序存在的问题。

如果想要进行源码级调试，就需要在程序可执行文件中加入一系列的额外信息，来弥补编译/解释过程中损失的信息，让动态调试器可以把指令地址、寄存器、内存地址“还原”为源代码级的概念。源代码行、变量、对象等等。但是，嵌入这些信息会使得程序可执行文件的体积增大，所以如果不是在编译时使用特定的选项，程序往往是没有这些额外信息的。以Linux下最常见的可执行文件格式ELF为例，若要支持源码级调试，需要ELF文件存在符号表和专门的调试信息。其中符号表可以用于将一些内存地址还原回函数或全局变量等，除了调试之外，还有许多其他用途，而调试信息则是专门为了将机器级概念还原到源代码级存在的。如果一个ELF文件只有符号表，没有调试信息，那么绝大多数源码级调试功能也都是不可用的，但是也可以支持一定程度的源码级调试，例如在函数的入口打断点，检查全局变量的值等等。对于解释型语言，情况还要更复杂一些。这是因为，gdb等动态调试器，都是把程序视为一个“黑盒”，它们并不理解一个程序是在完成自身的工作，还是在作为解释器，为一种更高级的语言（Python）提供支持。以Python为例，即使Python解释器程序本身有包含调试信息，从gdb的角度来看，也只能看到解释器本身的工作情况，例如它是如何解析Python字节码的，这个过程中它调用了自身的哪些函数，修改了自己内部的哪些变量等等。但是从Python程序员的角度来说，往往假设Python解释器本身是正确的，问题在于自己编写的Python代码，比起理解解释器内部的执行情况，更关注的是Python语言层级的概念。但gdb等通用动态调试器是无法在Python语言层级进行调试的。对于使用解释执行的语言，需要解释器本身支持调试功能，往往还需要使用专门的调试器。

在本次实验中，我们提供的炸弹程序是使用非常短的C代码编译而成的，且没有使用过高的优化等级，汇编指令与原始C代码是高度对应的；程序保留了符号表，但移除了调试信息。这是因为在操作系统中不可避免地存在无法使用高级语言，必须使用汇编语言编写的部分。因此我们希望通过一个复杂度有限的汇编程序，提高同学们对于汇编语言以及C语言编译到汇编语言过程的理解，同时增强同学们对gdb的熟悉程度和调试能力，为后续的实验打下基础。

设置调试目标

Last change: 2024-08-17, commit: [5b84259](#)

源码级调试 vs 汇编级调试

动态调试是在程序的运行过程中施加干预和观测状态，问题在于，何干预程序的运行？又该观测程序的什么状态？以打一个断点为例，应该在什么地方打断点，程序触发断点后，又该检查程序的哪些状态？动态调试器本身只是为程序员提供了完成上述工作的能力，但何运用这些能力，仍然需要程序员本身对于程序的了解。站在程序员的角度，无疑希望能直接在程序执行到某一行源代码时触发断点，触发断点后，又可以直接检查程序中某个变量的值甚至复杂对象的内容。许多同学此前可能已经接触过各种IDE自带的调试功能，它们大多都允许程序员在源代码中设置断点，并且可以在触发断点时直接看到各个变量和对象的内容。这种程序员直接站在源代码的层级，使用源代码级的概念（代码行、变量、对象等）进行调试的过程称为源码级调试。

然而，从动态调试器的功能而言，要支持源码级调试并非仅有动态调试器即可做到。这是因为，CPU本身只能执行二进制形式的机器指令，不论是编译执行或是解释执行的高级编程语言，最终在程序运行时，动态调试器能观察和控制的只是最终的机器指令、寄存器和内存地址。例如，仅使用动态调试器本身，我们只能指定在某一条指令暂停执行，也不能直接检查变量或者对象的内容，因为在机器指令的层面并没有变量和对象的概念，只有寄存器和内存。这种只使用机器执行过程中直接可见的概念（指令、寄存器、内存）进行调试的过程称为汇编级调试/机器级调试。造成上述问题的原因是，在从源代码到可执行程序的编译或解释过程中，许多信息都丢失了，因为这些信息对于程序的最终执行并无任何帮助：从程序执行的角度来说，CPU不需要理解某条指令对应源代码中的哪个文件的哪一行，也不需要理解某个寄存器在某一时刻存储的是哪个变量的值。但是，这些信息的丢失，就给调试带来了较大的困难，因为高级语言翻译成汇编指令的方式非常多样，并且存在各种复杂的细节，从而使得汇编级调试并不直观，往往需要远多于源码级调试的时间精力才有可能定位和理解程序存在的问题。

果想要进行源码级调试，就需要在程序可执行文件中加入一系列的额外信息，来弥补编译/解释过程中损失的信息，让动态调试器可以把指令地址、寄存器、内存地址“还原”为源代码级的概念——源代码行、变量、对象等等。但是，嵌入这些信息会使得程序可执行文件的体积增大，所以如果不是在编译时使用特定的选项，程序往往是没有这些额外信息的。以Linux下最常见的可执行文件格式ELF为例，若要支持源码级调试，需要ELF文件存在符号表和专门的调试信息。其中符号表可以用于将一些内存地址还原回函数或全局变量等，除了调试之外，还有许多其他用途，而调试信息则是专门为了将机器级概念还原到源代码级存在的。果一个ELF文件只有符号表，没有调试信息，那么绝大多数源码级调试功能也都是不可用的，但是也可以支持一定程度的源码级调试，例如在函数的入口打断点，检查全局变量的值等等。对于解释型语言，情况还要更复杂一些。这是因为，gdb等动态调试器，都是把程序视为一个“黑盒”，它们并不理解一个程序是在完成自身的工作，还是在作为解释器，为一种更高级的语言（Python）提供支持。以Python为例，即使Python解释器程序本身有包含调试信息，从gdb的角度来看，也只能看到解释器本身的工作情况，例如它是何解析Python字节码的，这个过程中它调用了自身的哪些函数，修改了自己内部的哪些变量等等。但是从Python程序员的角度来说，往往假设Python解释器本身是正确的，问题在于自己编写的Python代码，比起理解解释器内部的执行情况，更关注的是Python语言层级的概念。但gdb等通用动态调试器是无法在Python语言层级进行调试的。对于使用解释执行的语言，需要解释器本身支持调试功能，往往还需要使用专门的调试器。

在本次实验中，我们提供的炸弹程序是使用非常短的C代码编译而成的，且没有使用过高的优化等级，汇编指令与原始C代码是高度对应的；程序保留了符号表，但移除了调试信息。这是因为在操作系统中不可避免地存在无法使用高级语言，必须使用汇编语言编写的部分。因此我们希望通过一个复杂度有限的汇编程序，提高同学们对于汇编语言以及C语言编译到汇编语言过程的理解，同时增强同学们对gdb的熟悉程度和调试能力，为后续的实验打下基础。

Last change: 2024-08-17, commit: [5b84259](#)

使用简介

设置目标

gdb支持调试本地运行的进程，也支持通过网络等方式远程调试其他机器上运行的进程。在Linux机器上调试本地进程时，gdb依赖ptrace这个syscall，它从操作系统层面为gdb控制其他进程的运行提供了基础支持。一般而言，出于安全性考虑，各个Linux发行版都对ptrace syscall的调用进行了程度不等的权限控制。例如只允许通过ptrace调试子进程等。

启动程序为子进程

Tip

```
gdb <program>
```

其中，<program>为需要被调试的程序名，可以是一个完整的可执行文件路径，也可以只提供程序自身的可执行文件的名称。对于后一种情况，gdb会搜索\$PATH环境变量来找到可执行文件的实际路径

执行该命令后，会进入gdb命令行。此时，gdb并不会立即开始执行被调试的程序。但此时gdb已经载入了可执行文件中的符号表和调试信息（有），可以在实际执行程序前

就设置一些断点等，以便调试程序运行早期的代码或不接收输入的程序等等。确认完成准备工作后，在gdb命令行中执行 run 命令，作为gdb的子进程运行被调试程序。

Attach 进入子进程

Tip

```
sudo gdb -p <pid>
```

Important

一般而言，在常见Linux发行版上，直接attach到运行中进程可能需要root权限。

远程调试

gdb支持通过网络、串口等调试其他机器上运行的程序。这里我们以网络远程调试为例说明远程

调试的基本原理。 果想使用网络远程调试，需要在实际运行被调试程序的机器上启动一个 gdbserver，或任何实现了gdbserver协议的代理程序（以下统称为gdbserver）。gdb将会连接到这个gdbserver，并通过网络向gdbserver发送命令，以及通过gdbserver，读取运行中程序的信息等。在这个架构下，实际控制程序运行的是gdbserver，而不是gdb，但gdbserver又受到gdb的控制。远程调试的主要优点是提供了一种通用的将调试和程序的运行解耦的方法，而不是必须将被调试程序作为gdb的子进程运行。例 ，远程调试可以允许程序在其他机器上运行，这对于一些必须在内网中运行或是依赖特殊硬件等的程序很有用；此外，远程调试基于gdbserver协议，任何程序只要实现了该协议，都可以“表现为”一个gdbserver，此，程序可以主动将自己的一些内部信息暴露给gdb。理论上而言，这可以用于实现Python、Go等由虚拟机运行的程序的调试（虽然实际上这些语言的调试不是这么实现的），也可以用于调试由qemu运行的程序（详见下文）。使用 target remote <ip|domain name>: <port>，即可将gdb的调试目标设为远程的gdbserver。

果通过gdb进行远程调试，在运行gdb时是否还需要指定 <program>？事实上，答案并非是完全不需要。 果只想进行纯粹的汇编级调试，的确可以不需要指定 <program>，但 果想要进行源码级调试，则仍需要指定 <program>。在这种情况下，<program> 的主要作用是让gdb读取其中的符号表和调试信息。gdb不会直接从gdbserver中读取这些信息，因为gdbserver不一定能提供这些信息，gdb只会和gdbserver交换汇编级的信息，它依赖本地<program>文件中的符号表和调试信息将源码级信息翻译回汇编级信息。此外，还需要注意，在触发一个断点后， 果希望看到当前所执行到的程序源代码，则还需要本地保存了程序的源代码。这是因为，调试信息（以ELF所使用的DWARF格式为例）只保存了某条指令所对应的源代码的文件路径与行数，并没有直接保存源代码，gdb所做的工作是根据调试信息中的路径和行数，读取本地的代码文件并显示相应源代码。 果本地相应路径没有源代码文件或内容有误，那么gdb将无法正确显示源代码。

果运行gdb时没有指定 <program>，还可以通过 add-symbol-file 命令指定 <program>。 果程序不是在本地编译的，那么源代码的绝对路径可能与本地保存源代码的路径不同，可以用 set-substitute-path 命令进行替换。

断点控制

- break <expr> : 在 <expr> 处创建一个断点。<expr> 是一个最终可以求值为某条指令的地址的表达式。例 ， 果想直接在某个地址处设置断点，可以写成 * <address>， 果被调试程序有符号表，可以直接使用函数名。 果被调试程序有调试信息，可以指定在某个源文件的某一行设置断点，甚至可以使用更复杂的C表达式，更详细的信息可以参考手册。但需要注意，不论使用何种表达式，最终其实都是求值到一条指令的地址，gdb只是利用调试信息并帮助程序员简化了这一步骤。
- info breakpoints : 列出当前的所有断点
- disable/enable <NUM> : 禁用/启用编号为 <NUM> 的断点，断点编号可以通过前述 info 命令查看。
- delete <NUM> : 删除编号为 <NUM> 的断点。

执行控制

- 不输入任何命令，直接按下回车键：重复上一条命令
- 在触发断点后，可以通过下列命令控制 何恢复程序的执行
- continue : 恢复程序执行，直到触发下一个断点
- continue <NUM> : 恢复程序执行，且忽略此断点 <NUM> 次
- kill : 终止程序执行
- quit : 退出gdb
- stepi : 执行下一条机器指令，随后继续暂停执行（单步调试）
- stepi <NUM> : 执行接下来 <NUM> 条指令
- step : 执行下一条语句，这属于源代码级调试，需要调试信息
- nexti : 执行下一条指令，且不跟踪（step through）函数调用。与 stepi 不同， 果当前指
- 令是一条 bl 等指令，那么 stepi 会在被调用函数的第一条指令暂停（step in），而 nexti 会在 bl 指令之后的那条指令，即被调用函数返回后的那条指令上暂停。
- nexti <NUM> , next : 与 stepi , step 类似，只是不跟踪函数调用。
- finish : 恢复执行直到当前被调用的函数返回

显示信息

- backtrace : 显示栈跟踪信息，可以看到当前函数是 何一步步被调用到的。但需要注意，由于编译器的优化等因素， 果没有调试信息， 栈跟踪信息可能是不准确的，甚至无法提供栈跟踪信息。
- print <mods> <expr> : 打印表达式 <expr> 执行的值。其中 <expr> 可以是寄存器， pc,sp , x0，也可以是C表达式等。如果没有调试信息，则所使用的C表达式不能涉及到程序中变 x0+\$x1 。 <mods> 是可选的修饰符，可以用来控制打印的格式，详细信息可以参考手册。表达式求值时，寄存器、变量、内存中的值都基于触发断点时程序的状态。
- x/NFU <address> : 打印内存中指定长度的内容。与 print 命令不同，该命令只能打印内存中的内容，且不需要 print一样将内存地址转换为指针并求值。这是因为此命令只是将内存视为字节数组，打印出其中给定数目的字节的内容，而不对字节内容做任何解释。相反，print 在打印指针求值表达式时，会根据指针的类型信息和 <mods> 来计算需要打印多少字节，且会对多字节的内容进行解释，例 将它们显示为一个完整的整数，而非整数在内存中的各个字节。此命令的修饰符中， N 表示需要打印的单元数目， U 的取值可以为 b , h , w , g ，分别表示一个单元大小为1, 2, 4, 8字节。 F 表示每个单元的打印格式， i 表示作为指令反汇编， x 表示十六进制， d 表示十进制等，详细信息可以查看手册。
- display <mods> <expr> : 在每次程序暂停执行时，自动根据 <mods> 打印 <expr> 的值，适用于一些频繁更改的表达式。利用这条命令可以在单步调试时自动查看所需关注的

值，而无需反复输入print命令。

- info display : 列出所有的自动打印。
- delete display <NUM> : 删除编号为 <NUM> 的自动打印

TUI

Note

TUI(text UI)是gdb内置的高级命令行界面功能。TUI在gdb内部引入了类似窗口的概念，允许用户在使用gdb命令的同时查看被调试程序的汇编指令、源代码等，也允许用户设置自定义的UI布局，提高工作效率。

- lay asm : 进入TUI默认汇编布局，此布局下会同时开启一个gdb命令窗口以及一个汇编指令窗口，用户可以直观看到当前正在执行的汇编指令。
- lay src : 进入TUI默认源代码布局，此布局下会同时开启gdb命令窗口和源代码窗口。需要具备调试信息和源代码文件。
- tui disable : 退出TUI。

扩展阅读

- [ptrace syscall](#)
- [gdb手册](#)（想深入了解gdb或参考命令的详细用法，非常建议阅读，此文档只是给出了一个非常简要的介绍）：TUI: <https://sourceware.org/gdb/onlinedocs/gdb/TUI.html>
- [TUI](#)

Last change: 2024-08-17, commit: [5b84259](#)

Objdump

- 简介
- GNU实现 vs LLVM实现
- 可执行文件格式
- objdump使用
- 扩展阅读

简介

objdump是GNU binutils中的标准工具之一。它的主要作用是显示二进制程序的相关信息，包括可执行文件、静态库、动态库等各种二进制文件。因此，它在逆向工程和操作系统开发等底层领域中非常常用，因为在这些领域中，我们所接触到的程序很可能不提供源代码，只有二进制可执行文件；抑或是所面临的问题在高级语言的抽象中是不存在的，只有深入到机器指令的层次才能定位和解决。在这些领域中，objdump最常见的应用是反汇编，即将可执行文件中的二进制指令，根据目标架构的指令编码规则，还原成文本形式的汇编指令，以供用户从汇编层面分析和理解程序。例如，在拆弹实验中，我们只提供了炸弹程序的极少一部分源代码，同学们需要通过objdump等工具进行逆向工程，从汇编层级理解炸弹程序的行为，进而完成实验。

GNU实现 vs LLVM实现

objdump最初作为GNU binutils的一部分发布的。由于它直接处理二进制指令，因此它的功能与CPU架构等因素强相关。在常见Linux发行版中，通过包管理安装的objdump均为GNU binutils提供的实现，且它一般被编译为用于处理当前CPU架构的二进制指令。例如，在x86_64 CPU上运行的Linux中安装objdump，它一般只能处理编译到x86_64架构的二进制文件。对于GNU binutils提供的实现，如果需要处理非当前CPU架构的二进制可执行文件，则需要额外安装为其他架构编译的objdump，例如，如果要在x86_64 CPU上处理aarch64架构的二进制文件，一般需要使用aarch64-linux-gnu-objdump。LLVM是另一个被广泛应用的模块化编译器/工具链项目集合。LLVM项目不仅提供了另一个非常常见的C/C++编译器clang，也提供了一组自己的工具链实现，对应GNU binutils中的相关工具。例如，LLVM也提供了自己的objdump实现，在许多Linux发行版上，你需要安装和使用llvm-objdump命令。LLVM所提供的工具链实现与GNU binutils中的相应工具是基本兼容的，这意味着它们的命令行参数/语法等是相同的，但LLVM工具链也提供了其他扩展功能。从用户的角度来说，LLVM工具链与GNU binutils的一个主要区别是LLVM工具链不需要为每个能处理的架构编译一个单独的版本，以llvm-objdump为例，不论处理x86_64还是aarch64架构中的二进制文件，均可以使用llvm-objdump命令（需要在编译llvm-objdump时开启相应的支持，从发行版包管理中安装的版本通常有包含）。

可执行文件格式

为了更好地加载和执行程序，操作系统在一定程度上需要了解程序的内部结构和一系列元数据。因此，一个二进制可执行文件并不仅仅是将所有指令和数据按顺序写到文件中即可，通常它需要以一种特定的可执行文件格式存储，使用的格式是由它将要运行在的操作系统决定的。例如，对于Unix/Linux操作系统，ELF格式是目前最主流的可执行文件格式。而Windows下主要的可执行文件格式是PE/COFF。除了反汇编之外，objdump还可以显示与可执行文件格式相关的众多信息，但需要用户对可执行文件格式自身的理解。详细分析可执行文件格式超出了本教程的范围，感兴趣的同学建议查阅与ELF相关的资料。在后续ChCore实验中，也会涉及部分关于ELF的知识。

objdump使用

本节只介绍拆弹实验中可能用到的基础用法。详细用法感兴趣的同学可以参考扩展阅读部分。

- `objdump -dS a.out a.s` : 反汇编 a.out 中的可执行section（不含数据 sections），并保存到输出文件 a.s 中。在可能的情况下（有调试信息和源文件），还会输出汇编指令所对应的源代码。
- `objdump -dsS a.out a.s` : 将 a.out sections 的内容全部导出，但仍然只反汇编可执行 sections，且在可能情况下输出源代码。

扩展阅读

- [objdump手册](#)
- https://en.wikipedia.org/wiki/Executable_and_Linkable_Format
- <https://maskray.me/blog/>
- https://fourstring.github.io/ya-elf-guide/guide_zh

Last change: 2024-08-17, commit: [5b84259](#)

Make

- 简介
- 扩展阅读

简介

make是Linux操作系统上常见的较为基础的构建系统（build system）。构建系统的主要工作，是管理一个较大规模或较为复杂的项目的各个“部件”，以及 何将这些“部件”“组装”成为最终的项目产物。以C语言项目的编译过程为例。一个最简单的C语言项目，只需要有一个源文件即可，这种简单的项目，确实对构建系统没有很强的需求，只需要一条很简短的gcc命令即可完成编译。但是，随着项目规模的增加，项目的源文件也会越来越多，诚然，我们可以把所有源文件的文件名都提交给gcc命令进行编译，但是这样会导致每次都要输入一个很长的命令。或许有同学会说，命令历史记录或者写一个简单的脚本都可以简化这个操作。但是构建系统能做的不仅如此。果每次都把所有源文件提交给gcc让它进行编译，那么gcc每次都会执行一次完整编译，即重新将每一个源文件编译成对象文件，再将他们进行链接。然而，事实上，在一个规模较大的项目中，我们在更新这个项目时，通常不会修改项目的所有源文件，常常只有一小部分或者一部分源文件被修改了，那么那些没有被修改的源文件就完全没有必要再重新编译，只需要将已有的中间产物（对象文件）与修改过的文件重新编译后的对象文件进行链接即可，也即所谓的部分编译。换句话说，果每次都使用前述的一条完整gcc命令，那么相当一部分编译时间就被浪费了，因为gcc自身并不理解哪些文件需要重新编译，哪些文件不需要重新编译，而且除非添加特定参数，gcc也不会保留中间产物或利用现有的中间产物。相比于在一条命令中把所有源文件都提交给gcc，更好的做法是把整个构建过程拆散，先逐个将源文件编译为对象文件，并且保存这些对象文件作为中间产物，随后再将这些中间产物链接到一起，产生完整的程序。当然，这么做相较于一条gcc命令会复杂很多，但对于大型项目可以显著减少编译时间。由于整个流程从一条命令变成了需要以特定顺序执行的许多条命令，因此出于可维护性的考虑，应当把整个流程以文件形式保存下来，这就是构建系统中常见的规则文件。

前所述，一个比较复杂的项目，其编译流程涉及多条（甚至是大量）需要按照特定顺序执行的命令。何确定这些命令的顺序？何在项目的结构发生变更后重新确定这些命令执行的顺序？这些问题，是单纯的shell脚本所不能解决的，因此构建系统的规则文件通常并不是简单的shell脚本。一般而言，构建系统的规则文件都包含几个基本元素（不同系统中的术语可能不同），以make为例：目标（target）、配方（recipe）、前置条件（prerequisites）。目标指的是构建系统需要负责生成的一个文件或完成的一项任务，这个文件可以是某个中间产物文件，也可以是最终的产物或任何其他文件；此外，它也可以是一项抽象的任务，例 本实验中用到的使用qemu运行炸弹程序，也被作为make的一个目标。前置条件指的是在完成这个目标前需要完成的其他目标，一个目标只有在前置目标都已完成时才能被执行。最简单的前置目标是构建这个目标所需的源文件，例 用于生成对象文件app.o的目标，其前置目标可以是其源文件app.c。构建系统可以根据文件系统中app.o和app.c两个文件的修改时间，来判断是否需要执行这个目标，果app.c在app.o产生之后又被修改过，则认为app.o需要重新生成，否则就可以直接使用现有的app.o。配方则是定义 何完成某一个特定的目标，通常包含一系列shell命

令等。从上面的描述中不难看到，规则文件的作用实际上是定义了一个依赖图，图中每个目标是一个顶点，如果目标A依赖目标B，则目标B有一条边指向目标A。从原理上来说，构建系统会根据规则文件构建出依赖图，随后依照图的拓扑序执行每个顶点所定义的配方，最终就能生成完整的产物。还需要注意，构建系统是独立于编译器之外的，构建系统能做的，是给程序员提供自动生成和执行依赖图的能力，但依赖图的结构本身仍然是规则文件的编写者定义的，例如，项目的构建涉及到多少目标，每个目标的配方使用什么命令，每个目标的前置条件是什么，这些问题是不可能由构建系统自身来解决的，而是要由规则文件的编写者“告知”构建系统。

总而言之，使用构建系统而不是直接调用编译器，对于复杂C/C++项目有下列好处：

- 增强构建流程可维护性
- 简化执行部分任务（类似于脚本），例如清理构建中间产物等
- 通过部分编译/增量编译，可以显著减少大规模项目的编译时间
- 可以并发执行所有前置已被满足的目标（并发编译），有效利用多核CPU，进一步缩短编译时间

扩展阅读

Tip

make是一个比较底层的构建系统，在使用上仍有诸多繁琐之处，于是又出现了一些用于替代make或在make更上层，更为简化的构建系统，前者的代表项目之一是ninja，后者的代表项目之一是CMake，感兴趣的同学可以进一步了解。

- <https://faculty.cs.niu.edu/~mcmahon/CS241/Notes/makefiles.html>
- https://www.gnu.org/software/make/manual/html_node/index.html

Last change: 2024-08-17, commit: [5b84259](#)

QEMU

qemu是目前广泛应用的开源模拟器和虚拟机项目。它可以在一种架构的CPU（x86）上，模拟其他多种架构的CPU（aarch64等），这使得可以通过qemu在x86 CPU上运行为其他CPU架构编译的程序。例如，本实验的目的是让同学们熟悉aarch64汇编，因此我们提供的炸弹程序是为aarch64 CPU编译的，如果没有qemu，则程序不能在x86 CPU上执行。此外，qemu也可以模拟一个完整的机器，包括CPU、内存、磁盘以及多种其他外部硬件设备，此时它的功能基本等价于虚拟机，在这种场景下，它还可以与KVM技术配合，使用硬件虚拟化提高虚拟机的运行性能，但此时它就不能再运行为其他架构编译的操作系统了。

Last change: 2024-08-17, commit: [5b84259](#)

进程级模拟 vs 系统级模拟

qemu的模拟粒度可以分为进程级和系统级。在进程级模拟下，qemu只负责运行一个为其他架构编译的普通程序，这个程序与当前系统中运行的程序的唯一区别是它所使用的指令集不同，qemu会负责将它所使用的指令集翻译为当前机器可以执行的指令。除此之外，该程序相当于当前系统中的一个普通进程，它仍然通过当前系统的内核来访问操作系统提供的功能。默认情况下，qemu命令执行进程级模拟。例如，我们提供的炸弹程序就只需要使用进程级模拟，它本质上就是一个普通的用户态程序，只是编译到了aarch64指令集而非x86指令集。除此之外，它的输入/输出等功能，仍然是调用当前操作系统提供的syscall。

qemu-system 命令可以用于系统级模拟。在系统级模拟下，QEMU会模拟一整套硬件，包括CPU、内存、磁盘以及多种可选硬件设备，此时QEMU的功能类似于虚拟机。在系统级模拟下，QEMU不能直接运行普通的用户态程序，而是需要运行完整的操作系统，由操作系统来管理QEMU模拟出的硬件资源。系统级模拟与是否使用KVM等硬件虚拟化进行加速是正交的。如果不使用KVM，QEMU仍然通过动态指令集翻译来运行被模拟的操作系统，此时它可以运行为其他架构编译的操作系统。否则，使用硬件虚拟化可以增强性能，但QEMU不再可以运行为其他架构编译的操作系统。

Last change: 2024-08-17, commit: [5b84259](#)

QEMU + GDB

GDBServer

qemu实现了gdbserver协议，这使得gdb可以连接到qemu，并且给qemu发送指令，以及从qemu中读取信息。通过gdbserver，qemu可以把自己内部的信息暴露出来，或者说，可以让gdb理解它需要调试的目标是qemu中被模拟的程序（或操作系统），而不是qemu自身。例如，当gdb发送在地址0x400000设置一个断点的指令时，qemu可以在被模拟程序执行到0x400000时暂停被模拟程序的执行，并告知gdb客户端，而不是在qemu自身执行到0x400000时暂停。如果gdb需要读取内存中的值，qemu可以返回被调试程序的内存数据，而不是自身的内存数据。还需要强调，由于gdb和gdbserver之间交换的信息仅停留在汇编级，而且gdb完全使用本地的可执行程序文件中的符号表和调试信息执行源码级信息到汇编级信息之间的转换，因此在使用qemu系统级模拟时，我们可能会发现一些意料之外的行为。这是因为，在系统级模拟时，QEMU自身相当于站在CPU的抽象层面，而不是操作系统内核的抽象层面，这使得QEMU自身并不能理解它内部正在运行的操作系统的相关信息。例如，如果在qemu中运行一个Linux操作系统，其中的每一个进程的虚拟地址空间都有许多重叠，那么如果此时在0x400000打一个断点，qemu并不能识别出这个断点是针对哪个进程的，它所能做的只是在CPU“执行”到地址为0x400000的指令时暂停执行，不论当前执行的到底是哪个进程。又由于gdb客户端是完全根据用户所指定的可执行文件进行源码级到汇编级的翻译的，所以可能会出现这样一种情况：

- gdb客户端从appA中读取符号表，其中，函数funcA的地址为0x412345，它有2个参数
- 设置断点：break funcA，但实际上，gdb会根据符号表把funcA翻译回0x412345，qemu的gdbserver只能接收到在0x412345打断点的指令。
- 在qemu模拟的Linux操作系统中，进程appB恰好在执行，且它的函数funcB（只有1个参数）的地址恰好也为0x412345，qemu并不能理解这一点，它同样会暂停进程appB的执行。
- 根据appA的调试信息，gdb客户端知道funcA有2个参数，所以它会要求gdbserver读取寄存器x0和x1的值，用于显示funcA的参数。但由于此时暂停执行的实际上是只有1个参数的funcB，所以x1中保存的是一个随机值，x0中保存的值很可能也不符合funcA的第一个参数的语义。于是我们会看到虽然触发了断点，但函数参数却像是乱码。
- 同学们在后续chcore用户态相关的实验中，会对这个问题有更进一步的体会和理解。

扩展阅读

- <https://www.qemu.org/docs/master/>

Last change: 2024-08-17, commit: [5b84259](#)

ELF 文件格式

在Lab1中ChCore 的构建系统将会构建出 `build/kernel.img` 文件，该文件是一个 ELF 格式的“可执行目标文件”，和我们平常在 Linux 系统中见到的可执行文件 出一辙。ELF 可执行文件以 ELF 头部 (ELF header) 开始，后跟几个程序段 (program segment)，每个程序段都是一个连续的二进制块，其中又包含不同的分段 (section)，加载器 (loader) 将它们加载到指定地址的内存中并赋予指定的可读 (R)、可写 (W)、可执行 (E) 权限，并从入口地址 (entry point) 开始执行。

可以通过 `aarch64-linux-gnu-readelf` 命令查看 `build/kernel.img` 文件的 ELF 元信息（比如通过 `-h` 参数查看 ELF 头部、`-l` 参数查看程序头部、`-s` 参数查看分段头部等）：

```
$ aarch64-linux-gnu-readelf -h build/kernel.img
ELF Header:
  Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
        : ELF64
  Class: ELF64
  Data: 2's complement, little endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  ABI Version: 0
  Type: EXEC (Executable file)
  Machine: AArch64
  Version: 0x1
  Entry point address: 0x80000
  Start of program headers: 64 (bytes into file)
  Start of section headers: 271736 (bytes into file)
  Flags: 0x0
  Size of this header: 64 (bytes)
  Size of program headers: 56 (bytes)
  Number of program headers: 4
  Size of section headers: 64 (bytes)
  Number of section headers: 15
  Section header string table index: 14
```

更多关于 ELF 格式的细节请参考 [ELF - OSDev Wiki](#)。

Last change: 2024-08-17, commit: [fbdd223](#)

Linker Script

在Chcore 构建的最后一步，即链接产生 `build/kernel.img` 时，ChCore 构建系统中指定了使用从 `kernel/arch/aarch64/boot/linker.tpl.ld` 模板产生的 linker script 来精细控制 ELF 加载后程序各分段在内存中布局。

具体地，先将所有的调试信息段使用 `DWARF_DEBUG` 这个宏展开都放到虚拟地址 0 开始的地方，同时将 `${init_objects}` 中所有有效数据段（即除去调试信息段的数据）（即 `kernel/arch/aarch64/boot/raspi3` 中的代码编成的目标文件）放在了 ELF 内存的 `TEXT_OFFSET`（即 `0x80000`）位置，`.text`（代码段）、`.data`（数据段）、`.rodata`（只读数据段）和 `.bss`（BSS 段）依次紧随其后。

这里对这些分段所存放的内容做一些解释：

- `init`：内核启动阶段代码和数据，因为此时还没有开启 MMU，内核运行在低地址，所以需要特殊处理
- `.text`：内核代码，由一条条的机器指令组成
- `.data`：已初始化的全局变量和静态变量
- `.rodata`：只读数据，包括字符串字面量等
- `.bss`：未初始化的全局变量和静态变量，由于没有初始值，因此在 ELF 中不需要真的为该分段分配空间，而是只需要记录目标内存地址和大小，在加载时需要初始化为 0

除了指定各分段的顺序和对齐，linker script 中还指定了它们运行时“认为自己所在的内存地址”和加载时“实际存放在的内存地址”。例 前面已经说到 `init` 段被放在了 `TEXT_OFFSET` 即 `0x80000` 处，由于启动时内核运行在低地址，此时它“认为自己所在的地址”也应该是 `0x80000`，而后面的 `.text` 等段则被放在紧接着 `init` 段之后，但它们在运行时“认为自己在” `KERNEL_VADDR + init_end` 也就是高地址。

更多关于 linker script 的细节请参考 [Linker Scripts](#) 的官方手册。

Last change: 2024-09-06, commit: [668593f](#)

调试指北

Important

调试占据着你的大部分做lab的时间，本节主要讲解实验当中你可能需要使用到的调试手段，以及对应的工具使用方法。

- [VSCode](#)
 - [环境搭建](#)
 - [使用说明](#)
 - [Debug Console](#)
 - [Disassembler](#)
 - [Hex Editor](#)
 - [GDB Stub](#)
- [代码跳转 compile_commands.json](#)
- [防御式编程 Assertions](#)

VSCode

Gdb

你仍然可以使用gdb，我们在Makefile里保留了gdb的启动指令，运行 `make gdb` 也可以直接使用。

在Lab0中我们学习并熟悉了GDB的使用方法，但是从Lab1开始，Lab中的可调试文件就陡然增加了，使用GDB最麻烦的地方就是定位源文件进行单步调试，同时在调试的同时我们也无法迅速的对文件进行修改来做出调整，为此我们新增加了vscode的gdb设置，使用它你就可以一步进行对于内核镜像的调试工作。

环境搭建

Devcontainer

果你使用我们提供的vscode devcontainer配置，你可以直接跳过本节。

你需要安装Microsoft所提供的官方[C/C++插件](#).

使用说明

首先你需要在make之后所生成的.config中找到 CHCORE_KENREL_DEBUG 并将其值修改 on。然后运行 make clean && make 即可生成带调试符号的内核镜像。你可以使用 aarch64-linux-gnu-objdump 查看这个镜像的所有调试信息。

```
stu@Chcore:/workspaces/OS-Course-Lab/Lab1/build$ aarch64-linux-gnu-objdump --dwarf=info kernel.img
kernel.img:      file format elf64-littleaarch64
```

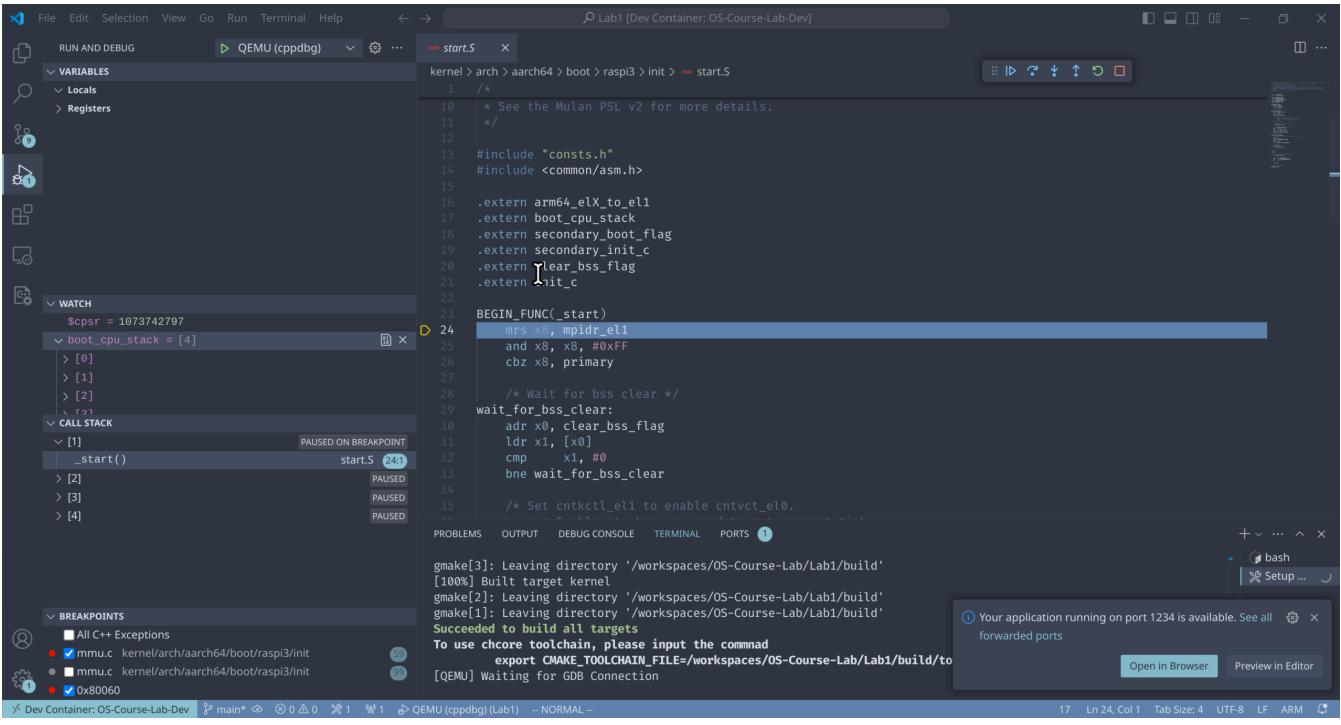
Contents of the .debug_info section:

```
Compilation Unit @ offset 0x0:
  Length:          0x2a (32-bit)
  Version:         2
  Abbrev Offset:  0x0
  Pointer Size:   8
<0><b>: Abbrev Number: 1 (DW_TAG_compile_unit)
  <c>  DW_AT_stmt_list    : 0x0
  <10> DW_AT_low_pc       : 0x800000
  <18> DW_AT_high_pc      : 0x80080
  <20> DW_AT_name         : (indirect string, offset: 0x0): /workspaces/
OS-Course-Lab/Lab1/kernel/arch/aarch64/boot/raspi3/init/start.S
  <24> DW_AT_comp_dir     : (indirect string, offset: 0x4c): /workspaces/
OS-Course-Lab/Lab1/build/kernel
  <28> DW_AT_producer     : (indirect string, offset: 0x78): GNU AS 2.34
  <2c> DW_AT_language      : 32769      (MIPS assembler)
...
...
```

Dwarf

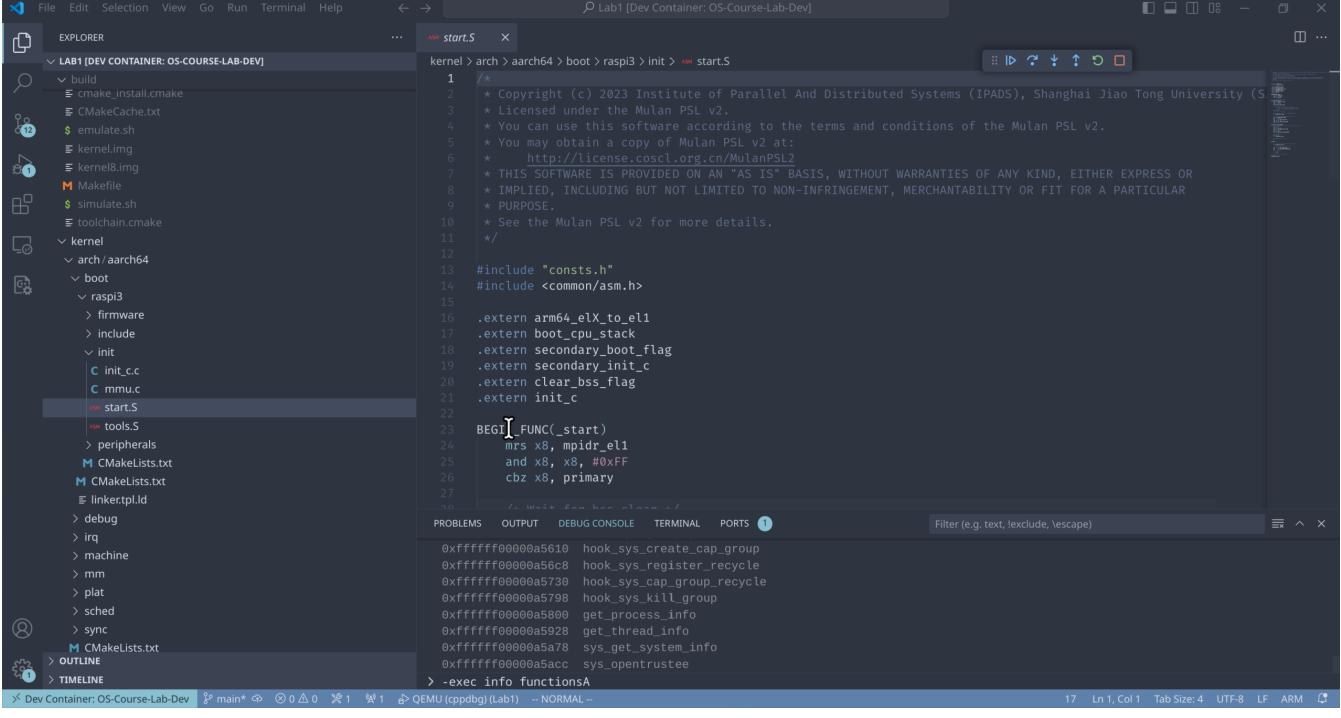
ELF文件所使用的调试信息格式为DWARF格式，果感兴趣你可以它的官网¹来详细了解。

构建完成后，在左侧侧边栏找到调试按钮，点击后在上方找到 QEMU(cppdbg) 的调试设置点击调试，既可以运行我们预先准备好的vscode gdb设置。此时你的调试器会暂停自动停在 start.s 中，此时你就可以像c++课上进行debug的方式一样自由的打断点以及观察点并进行调试了。由于 qemu-system-aarch64 模拟了所有的机器核心，在左侧显示了四个线程的调用堆栈。当你关闭调试后，qemu会自动退出，与调试普通的程序是一致的。



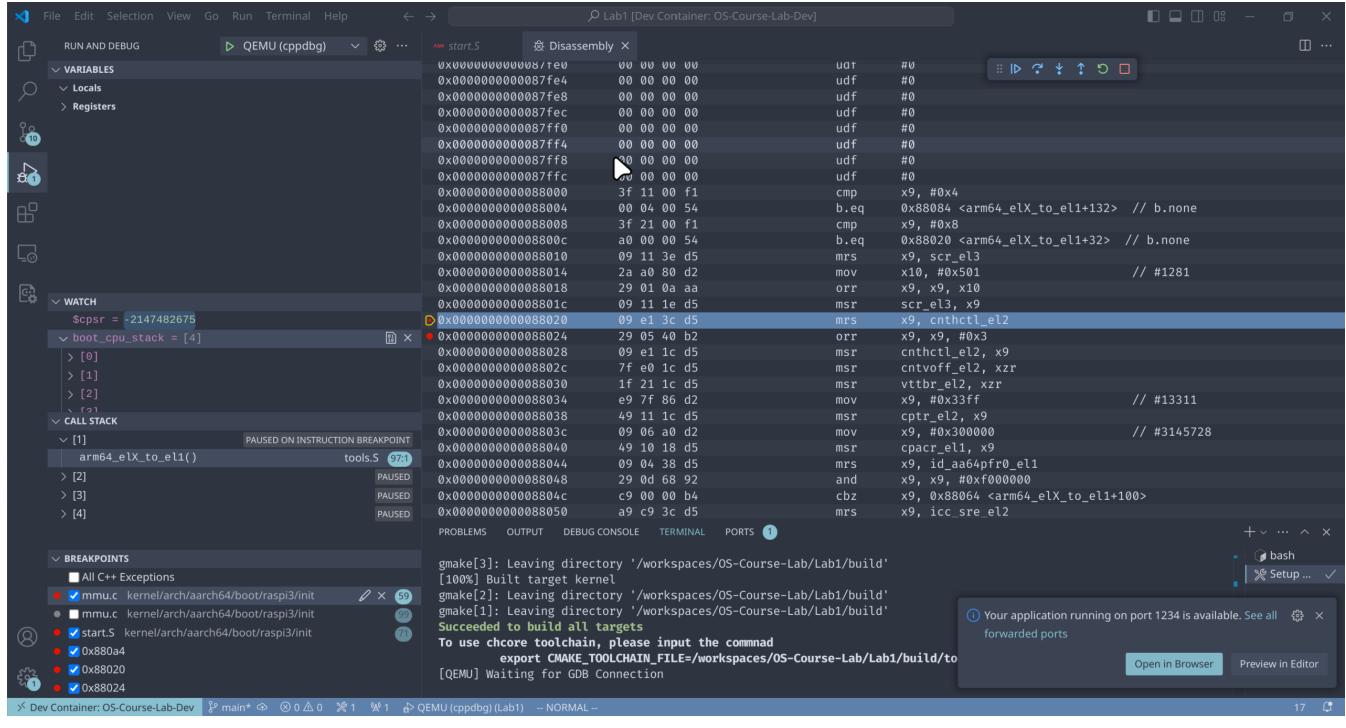
Debug Console

那么果我们有的时候还是想要敲一下键盘输一些gui没有的命令，比让gdb罗列一下这个镜像里有多少函数，此时 **debug adapter protocol** 协议会帮助我们向gdb传输命令，在代码编辑栏下方的 **DEBUG CONSOLE** 展示的就是我们cpptools所连接的 **gdb-mi** 的所返回的输出，此时使用 **-exec <command>** 即可以同往常我们使用gdb一样获取我们所需要的信息，**-exec info functions** 即可以提取我们需要该镜像中的所有符号。



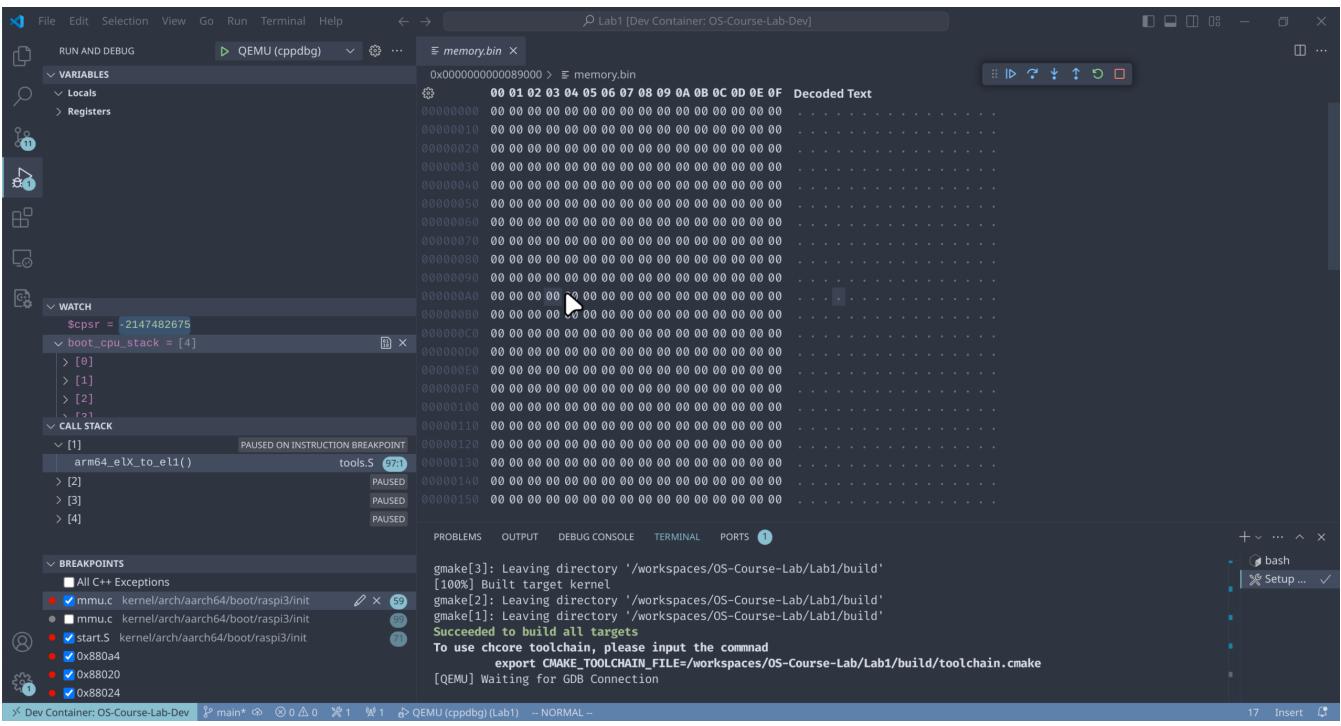
Disassembler

由于我们对一些重要的代码只提供了预先编译的可链接文件，所以这些文件一般都没有调试符号，但是你仍然可以对其进行的反编译代码进行查看。例如 `arm64_elX_to_el1` 函数你可以使用 `-exec b arm64_elX_to_el1` 进行断点添加，当你点击继续执行后此时 `gdb-mi` 会抛出异常并停下执行。此时打开vscode的命令面板输入 `disassembly` 打开反编译器面板即找到断点出的代码信息，在这个面板里你也可以使用单步调试对汇编进行调试。



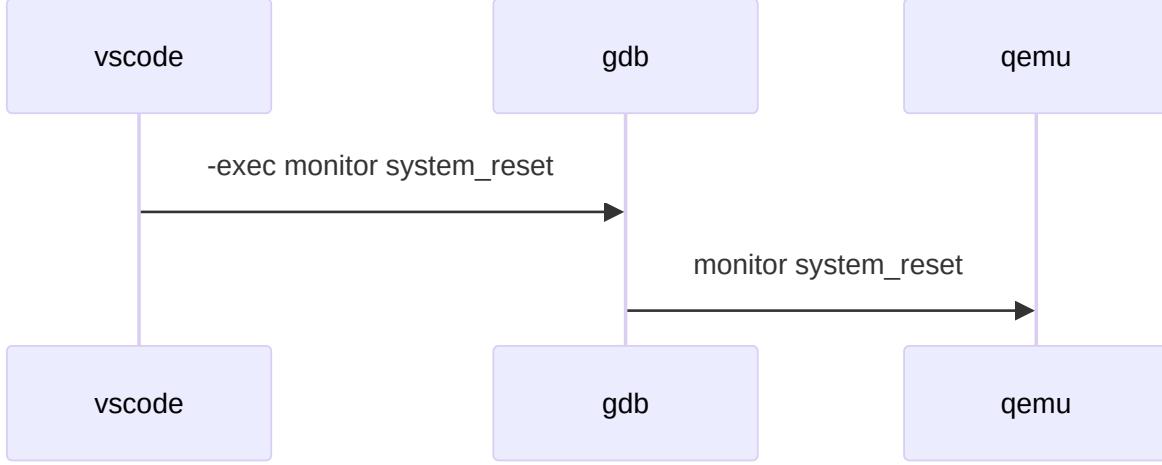
Hex Editor

有的时候你想对一块内存区域使用16进制编码进行查看，但是苦于gdb羸弱的表达方法效率很差，此时你可以使用Microsoft官方的 hex editor 来直接进行查看，你可以在左方的 watchpoint 中输入内存地址或者变量地址，当你停下后在watchpoint的右侧会出现一个16进制的按钮点开既可以对你监视点的内存区域的内容进行16进制的编辑与阅读。这个工具当你在苦于调试一段内存读写代码的时候很有用处。



GDB Stub

你可能不知道由于Qemu 实现了GDB Stub的同时也给gdb stub添加了与qemu相同的指令 system, memory等等指令，所以当你暂停时你同样可以通过 -exec monitor <command> 来对 qemu-monitor进行操作，例 运行 -exec monitor system_reset 此时你的qemu会从头开始重新执行，主要的关系 下



代码跳转 compile_commands.json

有的时候，你可能会奇怪为什么代码提示出错了，与脚本语言不同，C/C++的lsp代码提示也是依赖于编译选项的，例 你为编译器指定的包含路径以及宏定义（即编译时的上下文）。果我们不给LSP服务器提供我们浏览文件是怎么编译的，那么LSP就会按照默认的设置搜索默认的包含路径以及符号表，果恰好你的项目里有多个同名头文件，亦或者是在不同的文件里面定

义了相同的符号，那么LSP就会报错，因为它不知道到底哪个文件以及哪个符号是参与了最终构建。此时你就需要给LSP服务器提供一个提示性的信息用于告知该文件编译时的上下文。通常来说这个编译时的上下文构建系统是知道的，而llvm他们定义了一个标准的上下文文件格式即 `compile_commands.json`，你可以在 `build/kernel/compile_commands.json` 里看到它，这个文件在 CMake 生成 build 目录的时候会根据需要编译的文件自动生成。其记录了构建这个elf文件的过程中所使用的编译命令，此时LSP服务器即可以解析这个文件获取编译时刻的上下文信息，完成后就能得出准确的提示信息了。

```
[  
{  
    "directory": "/workspaces/OS-Course-Lab/Lab1/build/kernel",  
    "command": "/usr/bin/aarch64-linux-gnu-gcc -DCHCORE -DCHCORE_ARCH=\\\"\\\"aarch64\\\"\\\" -DCHCORE_ARCH_AARCH64 -DCHCORE_ASLR -DCHCORE_CROSS_COMPILE=\\\"\\\"aarch64-linux-gnu-\\\"\\\" -DCHCORE_KERNEL_DEBUG -  
DCHCORE_KERNEL_ENABLE_QEMU_VIRTIO_NET -DCHCORE_PLAT=\\\"\\\"raspi3\\\"\\\" -  
DCHCORE_PLAT_RASPI3 -DCHCORE_SUBPLAT=\\\"\\\"\\\"\\\" -DLOG_LEVEL=2 -I/workspaces/OS-  
Course-Lab/Lab1/kernel/include -I/workspaces/OS-Course-Lab/Lab1/kernel/user-  
include -I/workspaces/OS-Course-Lab/Lab1/kernel/include/arch/aarch64 -I/  
workspaces/OS-Course-Lab/Lab1/kernel/include/arch/aarch64/plat/raspi3 -I/  
workspaces/OS-Course-Lab/Lab1/kernel/arch/aarch64/boot/raspi3/include -g -  
Og -g -Wall -Werror -Wno-unused-variable -Wno-unused-function -nostdinc -  
ffreestanding -march=armv8-a+nofp -fno-pic -fno-pie -mcmodel=large -o  
CMakeFiles/kernel.img.dir/arch/aarch64/boot/raspi3/init/mmu.c.obj -c /  
workspaces/OS-Course-Lab/Lab1/kernel/arch/aarch64/boot/raspi3/init/mmu.c",  
    "file": "/workspaces/OS-Course-Lab/Lab1/kernel/arch/aarch64/boot/raspi3/  
init/mmu.c"  
},  
{  
    "directory": "/workspaces/OS-Course-Lab/Lab1/build/kernel",  
    "command": "/usr/bin/aarch64-linux-gnu-gcc -DCHCORE -DCHCORE_ARCH=\\\"\\\"aarch64\\\"\\\" -DCHCORE_ARCH_AARCH64 -DCHCORE_ASLR -DCHCORE_CROSS_COMPILE=\\\"\\\"aarch64-linux-gnu-\\\"\\\" -DCHCORE_KERNEL_DEBUG -  
DCHCORE_KERNEL_ENABLE_QEMU_VIRTIO_NET -DCHCORE_PLAT=\\\"\\\"raspi3\\\"\\\" -  
DCHCORE_PLAT_RASPI3 -DCHCORE_SUBPLAT=\\\"\\\"\\\"\\\" -DLOG_LEVEL=2 -I/workspaces/OS-  
Course-Lab/Lab1/kernel/include -I/workspaces/OS-Course-Lab/Lab1/kernel/user-  
include -I/workspaces/OS-Course-Lab/Lab1/kernel/include/arch/aarch64 -I/  
workspaces/OS-Course-Lab/Lab1/kernel/include/arch/aarch64/plat/raspi3 -I/  
workspaces/OS-Course-Lab/Lab1/kernel/arch/aarch64/boot/raspi3/include -g -  
Og -g -Wall -Werror -Wno-unused-variable -Wno-unused-function -nostdinc -  
ffreestanding -march=armv8-a+nofp -fno-pic -fno-pie -mcmodel=large -o  
CMakeFiles/kernel.img.dir/arch/aarch64/boot/raspi3/init/init_c.c.obj -c /  
workspaces/OS-Course-Lab/Lab1/kernel/arch/aarch64/boot/raspi3/init/init_c.c",  
    "file": "/workspaces/OS-Course-Lab/Lab1/kernel/arch/aarch64/boot/raspi3/  
init/init_c.c"  
},  
{  
    "directory": "/workspaces/OS-Course-Lab/Lab1/build/kernel",  
    "command": "/usr/bin/aarch64-linux-gnu-gcc -DCHCORE -DCHCORE_ARCH=\\\"\\\"aarch64\\\"\\\" -DCHCORE_ARCH_AARCH64 -DCHCORE_ASLR -DCHCORE_CROSS_COMPILE=\\\"\\\"aarch64-linux-gnu-\\\"\\\" -DCHCORE_KERNEL_DEBUG -  
DCHCORE_KERNEL_ENABLE_QEMU_VIRTIO_NET -DCHCORE_PLAT=\\\"\\\"raspi3\\\"\\\" -  
DCHCORE_PLAT_RASPI3 -DCHCORE_SUBPLAT=\\\"\\\"\\\"\\\" -DLOG_LEVEL=2 -I/workspaces/OS-  
Course-Lab/Lab1/kernel/include -I/workspaces/OS-Course-Lab/Lab1/kernel/user-  
include -I/workspaces/OS-Course-Lab/Lab1/kernel/include/arch/aarch64 -I/  
workspaces/OS-Course-Lab/Lab1/kernel/include/arch/aarch64/plat/raspi3 -I/  
workspaces/OS-Course-Lab/Lab1/kernel/arch/aarch64/boot/raspi3/include -g -  
Og -g -Wall -Werror -Wno-unused-variable -Wno-unused-function -nostdinc -  
ffreestanding -march=armv8-a+nofp -fno-pic -fno-pie -mcmodel=large -o  
CMakeFiles/kernel.img.dir/arch/aarch64/boot/raspi3/peripherals/uart.c.obj -  
c /workspaces/OS-Course-Lab/Lab1/kernel/arch/aarch64/boot/raspi3/peripherals/  
uart.c",  
    "file": "/workspaces/OS-Course-Lab/Lab1/kernel/arch/aarch64/boot/raspi3/  
peripherals/uart.c"  
}
```

]

我们在每个lab的 .vscode 下都为 clangd 这个LSP服务器提供了 compile_commands.json 的位置配置，同时也不使用 vscode 的同学们提供了 .clangd 的配置，用于引导clangd正确找到 compile_commands.json 的位置，在为LSP获取到正确的编译上下文信息后，他就可以为我们提供正常的重构，跳转，以及代码查看的功能了。

Compile Database

针对LLVM Compile Database的更多信息，请查阅llvm compile database的官方文档²

防御式编程 Assertions

调试器针对比较小范围的代码的作用是很大的，但大多数时候bug的发生点和异常发生点并不在同一个地方，例 在Lab1中你会配置页表，当你页表配置错误时，其实异常也是发生在配完页表实际执行到页表所映射的虚拟内存区域时才会出错，或者当你数据结构课编写AVL树时，你会发现你访问了不该是空指针的指针，这种异常的发生和bug发生点不一致的情况，通常是由于我们违反了 不定式(invariant) 所造成的。违反不定式并不会立即造成程序崩溃，我们需要做的是让异常在程序违反不定式时立即发生，此时我们就需要使用assert了。例 在AVL树中进行左旋以及右旋的时候，你可以通常使用 assert(node->left), assert(node) 等手段对其运行时行为进行检查。在ChCore中，我们准备了 BUG_ON 宏用于检查运行时的一些 expr 是否正常，除了 kinfo 以外，你可以使用这个宏来强行拦截异常，便于缩小你的debug的范围。

Last change: 2024-09-07, commit: [1f766de](#)

-
1. DWARF: <https://dwarfstd.org/dwarf5std.html> ↩
 2. LLVM Compile Database: <https://clang.llvm.org/docs/JSONCompilationDatabase.html> ↩