# 迭代式操作系统内核实验

# 简介

- 本操作系统内核实验遵循迭代式设计的基本思想
- 完整的实验分为三次迭代过程
- 循序渐进、由浅入深、加快反馈
- 每次迭代结束都是一个较为完整的操作系统内核

# 架构介绍

- 基于 AArch64
- 兼容鲲鹏、树莓派3、树莓派4

# 第一次迭代

- 目标
  - 构建一个具有现代操作系统内核基本结构的建议内核
  - 拥有运行一个用户程序且打印字符串的基本功能

# 第一次迭代实验内容

- 工具链
- 启动(Booting)
- 内存管理
- 异常(中断)处理
- 多核与锁
- 进程管理

# 第二次迭代

- 目标
  - 完善内核，使其更加符合现代操作系统内核的结构
  - 拥有现代操作系统内核的核心功能

# 第二次迭代实验内容

- I/O 框架
- 块设备驱动
- Libc
- 文件系统
- Shell

# 第三次迭代

- 第三次迭代提供了设计和研究操作系统内核的机会
- 根据场景和个人喜好设计、改造操作系统内核
- 内容上不做限制
- 提供一些可选主题以供参考

# 第三次迭代参考主题

- Buddy System
- 块设备驱动优化

# Lab0 工具链

- 实验目标
  - 本次实验主要是为后续实验搭建开发环境、熟悉相应工具。
- 开发环境
  - 类 Unix 操作系统(Ubuntu, Arch Linux 等)
  - GCC
  - GNU Make
  - 版本管理工具 Git
  - 硬件模拟器 QEMU
  - 调试工具 GDB

# 交叉编译工具链

- 编译是指将代码经过一系列步骤转换为可执行文件的过程
- 每个步骤都有对应的工具将前一步骤的输出作为输入进行相应处理，我们称这些工具为编译工具链
- 包括编译器（Compiler），汇编器（Assembler）和连接器（Linker）

# 交叉编译工具链

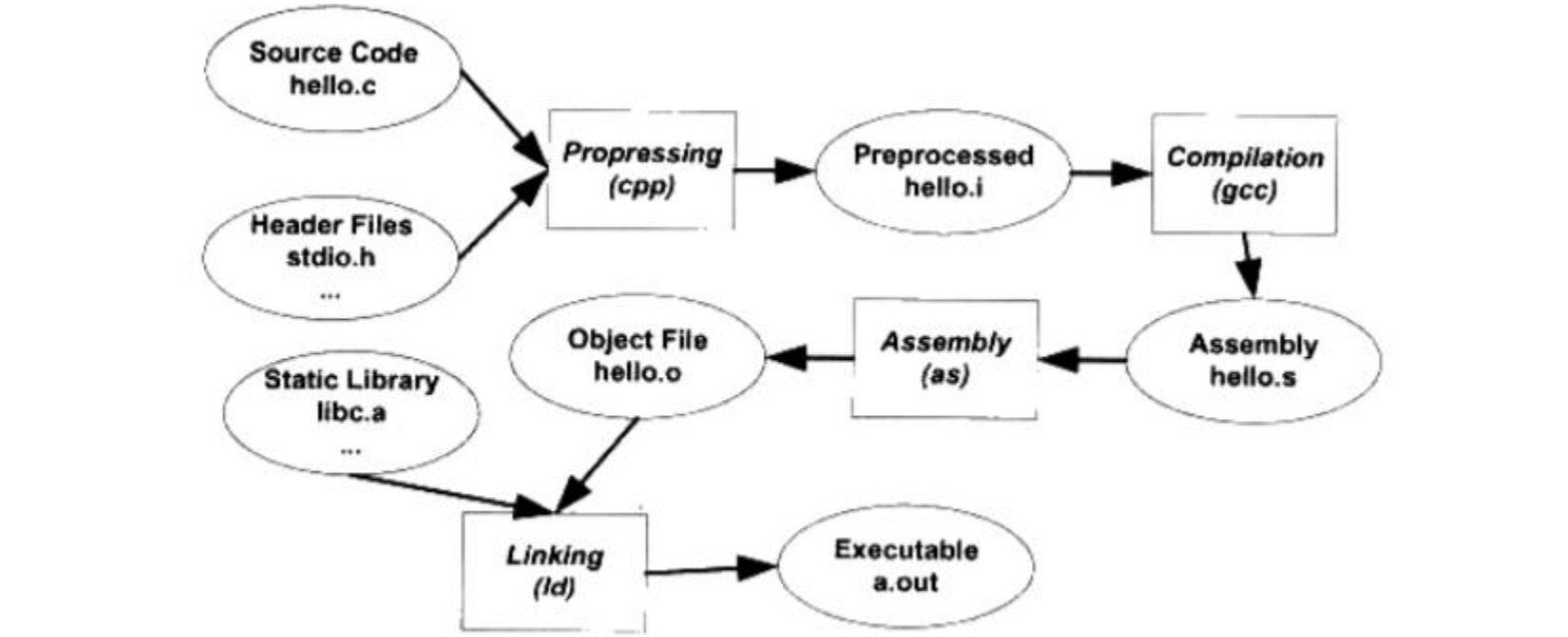



图 2-1　GCC 编译过程分解

# 交叉编译工具链

- 交叉编译是指在一种平台上编译出能在体系结构不同的另一种平台上运行的程序，交叉编译工具链即是该流程中所使用的工具。
- 对应到我们的实验则是指，我们在 X86 架构的 PC 平台(开发机)中编译出能在 ARM 架构 CPU (目标机)上运行的操作系统内核。

# 交叉编译工具链

- 我们推荐以 Ubuntu，Arch Linux 、WSL 等平台作为开发环境，（以下教程基于 Ubuntu 18.04-LTS 环境配置）
- 安装
  - sudo apt install gcc-aarch64-linux-gnu
- 测试
  - aarch64-linux-gnu-gcc –v
  - 显式 gcc 版本信息且 target 参数为 aarch64-linux-gnu 则安装成功

# QEMU

- 本实验内核支持鲲鹏、树莓派3、树莓派4等架构
- 但在开发初期使用真实物理硬件开发调试十分困难
- 通过硬件模拟器模拟真是环境来辅助调试
- 能有效提升开发效率与开发体验
- QEMU 是一个广泛使用的开源计算机模拟器和虚拟机
- 可以在一种架构（如 X86 PC）下运行另一种架构（如 ARM）下的操作系统和程序

# QEMU

- 安装
  - sudo apt install qemu qemu-system-arm qemu-efi-aarch64 qemu-utils
- 验证
  - qemu-system-aarch64 -M help

# GDB

- GDB 是非常常用的调试工具
- 系统自带的 gdb 并不支持跨平台的 debug，因此需要安装支持多体系结构的版本
- 安装
  - sudo apt install -y build-essential gdb gdb-multiarch

# Git

- 整个内核实验的开发和维护建议通过 Git 实现版本管理
- 安装
  - sudo apt install git

# GNU Make

- Make 会根据项目文件给定的依赖关系自动找出相应的文件进行编译

# GNU Make

```
target ...: prerequisites ...
    recipe
    ...
    ...
```

- Makefile 的规则如图所示
- target
  - 可以是一个 object file（目标文件），也可以是一个可执行文件，还可以是一个标签（label），例如 clean
- prerequisites
  - 生成该 target 所依赖的文件和/或 target
- recipe
  - 该 target 要执行的命令（任意的 shell 命令）
- 对 Make 感兴趣的同学，可自行参考 Makefile tutorial 进行学习。

# pwndbg(optional)

- gdb调试起来可能会不太美观、方便
- 推荐安装 pwndbg以提升调试体验
- 需注意配合交叉工具链的 pwndbg 需要自行配置

# Lab0 练习

- 请确保你当前处于项目根文件夹的 `lab0` 下，按顺序完成如下操作
- 通过 `git checkout -b dev` 创建、切换到 `dev` 分支
- 通过 `git branch -v` 可以查看所有本地分支，显示相应分支的信息，并指示当前所在分支
- 创建一个 `README.md` 文件，并写入你的名字和学号
- 通过 `git status` 可以查看当前分支下未保存的修改
- 通过 `git add` 和 `git commit` 来保存你的修改
- 通过 `git checkout lab0` 切换到 `lab0` 分支，通过 `git rebase dev` 将 `dev` 分支上的修改作用到 `lab0` 分支，并修复冲突
- 通过 `git log` 可以查看当前分支下的 log

# Lab0 练习

- Makefile Exercise
  - 请画出 `simple/Makefile` 中 make 命令的依赖关系。

# Lab1 启动

- 在本实验中，我们正式启动内核。
- 我们会学习 ARM v8 架构、树莓派 3 的启动流程、链接器脚本，然后为内核提供一个 C 的运行环境并跳转到 C 代码里面。

# ARM v8

- ARM 同 MIPS 一样是一种精简指令架构（RISC）
- 它相较复杂指令架构（CISC，如 X86）而言提供更简单的指令操作与访存模型（LOAD/STORE）
- 以对内存中某一个值加一为例
  - X86 的 `INC` 可直接将内存中的值加一
  - 而 ARM 必须通过 `LOAD` 将内存中的值读入寄存器
  - `ADD` 将寄存器加一
  - `STORE` 将寄存器写回内存
- 虽然 CISC 的指令功能更强大，但简单的指令操作可以缩短 RISC 的时钟周期以加快硬件执行。

# 树莓派 3 与 BCM2837

- 除了 arm 架构外，我们还要了解树莓派 3 板子（基于 BCM2837）的硬件信息
- 本实验中我们只需知道如下两点：
  - 如何启动？即树莓派怎么加载内核的？
  - 物理内存中哪些是可用的（如用于给用户进程分配内存），哪些是和设备相关的 MMIO（Memory-mapped IO）？

# 启动流程

- 对于 qemu 模拟的树莓派 3，就是简单的把 kernel8.img 复制到 0x80000 的内存地址处，然后跳到 0x80000 开始执行
- 而真实的树莓派是从 GPU 的ROM启动的（类似 BIOS），然后它会读取 SD 卡的第一个分区（必须为 FAT32 分区），依次读取并执行其中的 bootcode.bin，start.elf 和内核镜像，然后将内核镜像直接加载到内存的某个位置
- 其中 bootcode.bin 和 start.elf 可理解为官方提供的 Boot Loader，并不是开源的
- 但我们可以进行一些配置（如内核镜像的名称及其加载到内存中的位置），具体请参考 Boot Folder

# 内存布局

| ARM 物理地址 | 描述 |
| --- | --- |
| [0x0, 3F000000) | Free memory |
| [0x3F000000, 0x40000000) | MMIO |
| [0x40000000, 0x40020000) | ARM timer, IRQs, mailboxes |

# Linker Scripts

- 我们用链接器脚本（kern/linker.ld）解决了如下两个问题：
  - 对于编译器而言，例如 `b main` （跳到名为 main 的函数）的指令应该编译成什么？显然，生成对应的机器码需要知道 main 的地址。而链接器脚本就是用来控制这些地址的。
  - 回顾一下树莓派的启动流程，加载内核时是把 kernel8.img 直接复制到物理地址 0x80000 处，然后从 0x80000 开始执行。于是，kernel8.img 的开头处就是我们内核的“入口”（kern/entry.S），换言之，我们需要将 entry.S 放在内核镜像文件的开头，并用 objcopy 去掉 ELF 头。

# Linker Scripts

- 下面仅介绍我们所用到的语法，具体请参考 Linker Scripts
  - `.` 被称为 location counter，代表了当前位置代码的实际上会运行在内存中的哪个地址。如 `. = 0xFFFF000000080000;` 代表我们告诉链接器内核的第一条指令运行在 0xFFFF000000080000。但事实上，我们一开始是运行在物理地址 0x80000（为什么仍能正常运行？），开启页表后，我们才会跳到高地址（虚拟地址），大部分代码会在那里完成
  - text 段中的第一行为 `KEEP(*(.text.boot))` 是指把 kern/entry.S 中我们自定义的 `.text.boot` 段放在 text 段的开头。`KEEP` 则是告诉链接器始终保留此段。
  - `PROVIDE(etext = .)` 声明了一个符号 `etext`（但并不会为其分配具体的内存空间，即在 C 代码中我们不能对其进行赋值），并将其地址设为当前的 location counter。这使得我们可以在汇编或 C 代码中得到内核各段（text/data/bss）的地址范围，我们在初始化 BSS 的时候需要用到这些符号。

# ELF 格式

- 运行 `make` 后，我们可以在 obj/kernel8.hdr 查看 ELF 头。在查看 SYMBOL TABLE 时，Linux 下可用 `sort <obj/kernel8.hdr` 查看对地址排序后的结果。
- 注意区分虚拟地址（VMA）和加载地址（LMA）：
  - VMA是执行时的地址，会影响到代码中地址的编译。 链接器脚本中的 `.` 就是用来影响 VMA 的。
  - LMA是加载的地址，这个地址只会出现在 ELF 头中，只是告诉 loader 需要把这个 elf 文件拷贝到内存中的哪个位置。由于树莓派加载的是去掉 ELF 头后的 obj/kernel8.img，我们不需要关心 LMA。

# 代码注释

- 调整 Exception Level
  - 我们只会用到 EL0 和 EL1（类似 x86 的 ring3 和 ring0）
  - 分别代表用户态和内核态
  - 在树莓派 3 硬件上，我们会从 EL2 启动
  - 但在 qemu 上可能会从 EL3 启动
  - 于是在 kern/entry.S 中，我们需要先判断一下当前的 EL，然后逐步降至 EL1。

# 代码注释

- 初始页表
  - 操作系统内核的代码通常位于高地址（为什么这么设计？），而鉴于 Armv8 的 MMU 机制，即可以根据地址的高 16 位来自动切换不同的页表，我们将内核放在虚拟地址 [0xFFFF'0000'0000'0000, 0xFFFF'FFFF'FFFF'FFFF]，剩余的虚拟地址 [0x0 ~ 0x0000'FFFF'FFFF'FFFF] 则留给用户代码。
  - 为了便于实现，我们在 kern/kpgdir.c 中直接硬编码了一个页表，将两块 1GB 的内存 [0, 1GB) 和 [KERNBASE, KERNBASE+1GB)，均映射到物理地址的 [0, 1GB)，其中 KERNBASE 为 0xFFFF‘0000’0000‘0000，如下

| **虚拟地址** | **物理地址** |
| --- | --- |
| [0x0, 0x4000'0000) | [0x0, 0x4000'0000) |
| [0xFFFF'0000'0000'0000, 0xFFFF'0000'4000'0000) | [0x0, 0x4000'0000) |

# 代码注释

- 初始页表
  - 为什么需要上表中的第一个映射（恒等映射）呢？
    - 第一个原因是开启 MMU 后我们的 PC 仍在低地址，若没有此映射，开启 MMU 后的第一条指令就会出错
    - 第二个原因是便于直接访问物理内存
  - 其实我们还把 [KERNBASE+1GB, KERNBASE+2GB) 映射到了 [1GB, 2GB)，这在之后配置时钟中断时会用到
  - 具体如何配置页表，我们会在后续 lab 中介绍。

# 代码注释

- 进入 C 代码
  - 开启页表后，我们把栈指针指向 _start，即 [0xFFFF'0000'0000'0000, 0xFFFF'0000'0008'0000) 为初始的内核栈
  - 然后就可以安全地跳到高地址处的 C 代码了（kern/main.c:main），main 函数不会返回。

# Lab1 习题

- 未初始化的全局变量和局部静态变量通常会被编译器安排在 BSS 段中，而为了减小可执行文件的大小，编译器不会把这段编入 ELF 文件中。我们需要手动对这其进行零初始化，请补全 kern/main.c 中的代码，你可能需要根据 kern/linker.ld 了解 BSS 段的起始和终止位置。
- 请补全 kern/main.c 中的代码，完成 console 的初始化并输出 "hello, world\n"，你可能需要阅读 inc/console.h, kern/console.c。

# Lab2 内存管理

- 内核主要负责四个任务:
  - 内存管理
  - 进程调度
  - 设备驱动
  - 系统调用
- 我们在本学期的教学操作系统内核实验中均会涉及到。
- 在本实验中，需要实现教学操作系统内核的内存管理系统。内存管理系统分为两部分:
  - 内核的物理内存分配器，用于分配和回收物理内存
  - 内核的页表，用于将虚拟地址映射到物理地址

# Git

- 在本次实验以及后续实验中，你需要根据文档指示自行实现教学操作系统的内核，我们仅会提供部分代码用于引导。
- 每次发布新的 lab，均会开启一个新的分支，以下（并非强制性的）获取新 lab 的流程在后续实验中会经常出现。
- 首先确保你当前所在分支为完成 lab1 实验的分支，假设为 `dev`。

# Git

- git pull # 通过 git pull 获取 github 仓库中的更新
- git checkout -b dev-lab2 # 从 dev 切换到新的分支 dev-lab2
- git rebase origin/lab2 # 将 lab2 中的更新作用于 dev-lab2
- work in lab2-dev branch...
- git add files you've modified
- git commit -m "balabala"
- git checkout dev # 在生成新的 commit 后，切换回 dev
- git rebase lab2-dev # 将 dev-lab2 中的修改作用于 dev，此时 dev 分支为完成 lab0-2 实验的分支

# 参考文档

- 因为内核需要与硬件大量交互，将参考 ARM 架构文档来理解相关硬件配置。我们会在实验文档中给出需要参考 ARM 架构文档的哪部分
- `ref: C5.2` 指示相关部分在 ARMv8 Reference Manual 的 C5.2
- `A53: 4.3.30` 指示相关部分在 ARM Cortex-A53 Manual 的 4.3.30
- `guide: 10.1` 指示相关部分在 ARMv8-A Programmer Guide 的 10.1

# 物理内存管理

- 物理内存是指 DRAM 中的储存单元，每个字节的物理内存都有一个地址，称为物理地址；
- 虚拟内存是程序（用户进程、内核）看到的存储空间。
- 在本实验中，你需要为内核实现一个物理内存分配器为后续用户进程的页表、栈等分配空间
- 内核会将物理内存中内核代码后（`PHYSTOP` 前）的物理内存用于分配，并且通过链表来管理空闲物理内存，分配即从链表中拿出一个物理页，回收将对应的物理页加入链表中。

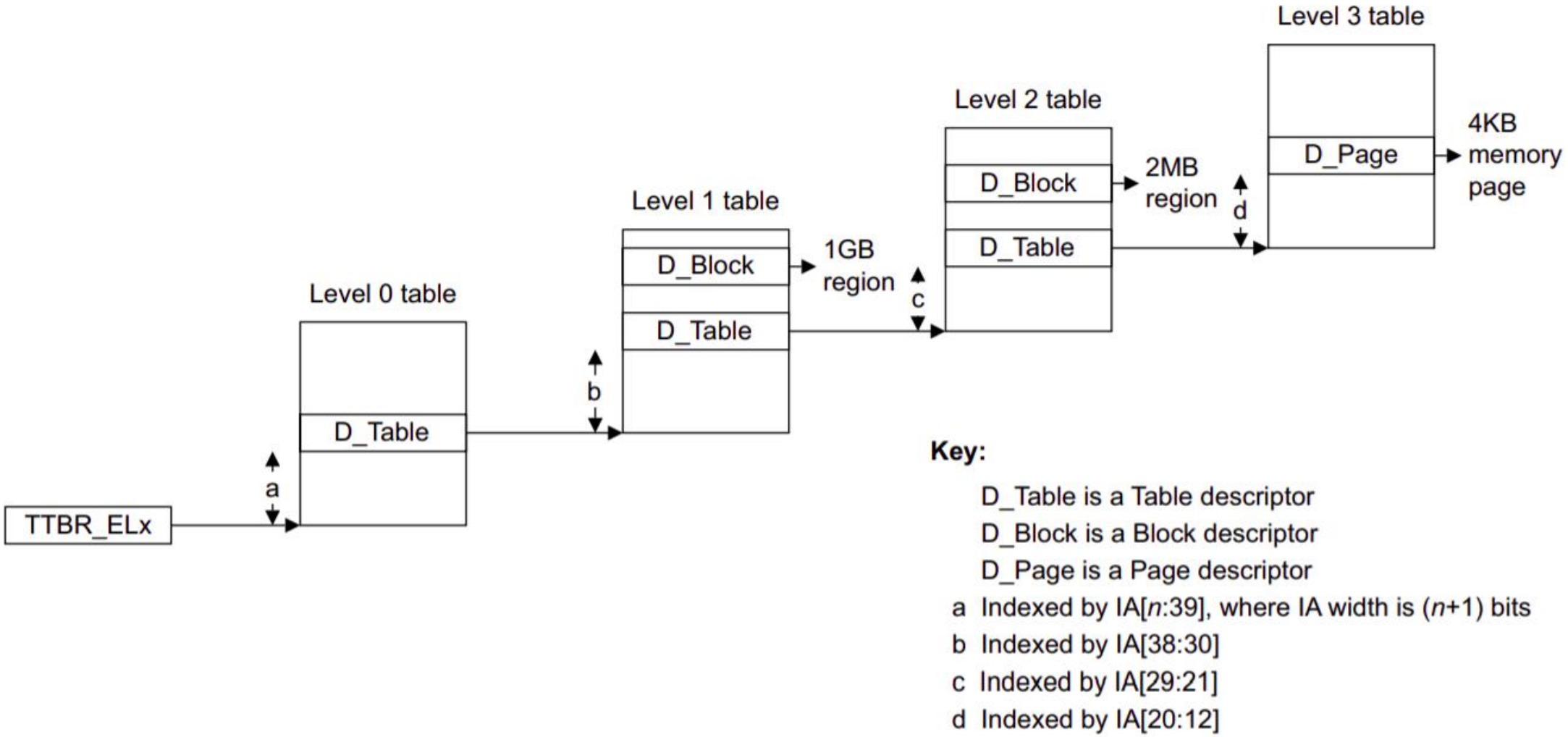
# 物理内存管理

- 物理内存分配器通过 `free_list` (kern/kalloc.c:20) 来管理所有空闲物理内存，`free_list` 即是链表头，链表中的每个节点是一个 `struct run` (kern/kalloc.c:15) 用来代表每个空闲页，那么 `struct run` 存在哪里呢？
- 进入内核代码后，内核会通过 `alloc_init` 将所有可用的物理内存回收。`end` 定义在链接器脚本 `kern/linker.ld` 中用于指示内核代码的终止，`free_range` 通过 `ROUNDUP` (kern/kalloc.c:48) 保证给定的空闲内存段首地址是按页对齐。`kfree` 将给定页的内容全置为 1 使得对空闲内存的访问无法获得原有内容方便调试

# 虚拟内存

- 虚拟内存与页表的管理（页的粒度、虚拟地址的大小等）与指令集架构密切相关
- 实验使用的 ARM 架构提供了三种页的粒度：64 KB(16 bits)，16 KB(14 bits)，4 KB(12 bits)
- 本学期的内核实验我们将采用 4 KB 的页大小
- 在 48-bit 索引的虚拟内存中虚拟地址通过页表转换为物理地址的过程如下图。

# 虚拟内存


Level 3 table
Level 2 table
Level 1 table
Level 0 table
D_Page
4KB
memory
page
D_Block
2MB
region
d
D_Table
D_Block
1GB
region
c
D_Table
b
D_Table
a
TTBR_ELx
Key:
D_Table is a Table descriptor
D_Block is a Block descriptor
D_Page is a Page descriptor
a Indexed by IA[n:39], where IA width is (n+1) bits
b Indexed by IA[38:30]
c Indexed by IA[29:21]
d Indexed by IA[20:12]

# 虚拟内存

- 当拿到 64-bit 的虚拟地址时
  - 如果前 16 位全 0，CPU 从 `ttbr0_el1` 读出 level-0 页表
  - 如果前 16 位全 1，CPU 从 `ttbr1_el1` 开始 level-0 页表
  - 否则，报错
- 拿到 level-0 页表后，va[47:39] 作为索引获取下一级页表信息（页表地址、访问权限）
- 页表项 entry 后 12 位用于权限控制
- 在 level-1、level-2 中 entry[1] 可用于指示当前项的属性（为 0 时 block、为 1 时 table）(ref:D4.3.1)
- 在 level-3 中 entry[1] 只能为 1，指示当前项中包含物理地址 (ref:D4.3.2)
- entry[0] 用于指示当前项是否有为空（可用于映射物理地址）(ref:D4.3.2)
- entry[7:6] 用于权限管理 (ref:D4.4.4)

# 虚拟内存

- `map_region` 函数用于在 `pgdir` 对应的页表上建立 `va` 到 `pa` 的映射
- 其中 `size` 为物理内存段的大小
- `perm` 指代该段的权限
- `vm_free` 用于回收页表 `pgdir` 占用的物理空间。

# Lab2 练习

- 物理内存分配器
  - 完成物理内存分配器的分配函数 `kalloc` 以及回收函数 `kfree`。
- 页表管理
  - 完成物理地址的映射函数 `map_region` 以及回收页表物理空间函数 `vm_free`。

# Lab3 中断与异常

- 本次实验我们会学习操作系统里中断相关的知识，将会涉及如下几个部分
  - 理解中断的流程
  - 了解中断的配置
  - 设计 trap frame，实现保存并恢复 trap frame （习题）
  - 理解中断处理函数

# 中断的流程

- 为了更好的了解中断，我们需要知道 CPU 在中断时究竟做了些什么。
- ARMv8 架构下中断时，CPU会先关闭所有中断，根据中断向量表进行跳转，保存一些中断信息，并进行栈的切换。更具体的:
  - 中断后自动关中断，这意味着所有的中断只会在用户态发生，即只有从 EL0 到 EL1 的中断
  - 根据不同的中断种类，跳到中断向量表中的不同位置
  - 保存中断前的 PSTATE 至 SPSR_EL1，保存中断前的 PC 到 ELR_EL1，保存中断原因信息到 ESR_EL1
  - 将栈指针 sp 从 SP_EL0 换为 SP_EL1，这也意味着 SPSEL_EL1 的值为 1

# 中断的配置

- 中断向量表
  - 在 `kern/main.c` 中我们用 `lvbar` 加载了ARMv8 的中断向量表 `kern/vectors.S`
  - 其作用是，当硬件收到中断信号时，会根据其中断类型（Synchronous/IRQ/FIQ/System Error）和当前 PSTATE（EL0/EL1，AArch32/AArch64），保存中断原因到 ESR_EL1，保存中断发生时的 PC 到 ELR_EL1，然后跳转到不同的地址上。描述这些地址的那张表称为中断向量表。
  - 中断向量表的地址可以通过 VBAR_EL1 配置（注意这是一个虚拟地址），表中的每一项是一种中断类型的入口，大小为 128 bytes，共有 16 项。
  - 上节中提到过我们只支持从 EL0 到 EL1 的中断，且我们的内核只支持 AArch64，于是在这里只需关注如下两项即可

# 中断的配置

- 中断向量表
  - 于是在 `kern/vectors.S` 中除了这两项，其他的都会执行 `kern/trap.c:irq_error` 来报错。

| Address | Exception type | Description | Usage |
| --- | --- | --- | --- |
| VBAR_EL1+0x400 | Synchronous | Interrupt from lower EL using AArch64 | System call |
| VBAR_EL1+0x480 | IRQ/vIRQ | Interrupt from lower EL using AArch64 | UART/timer/clock/sd |

# 中断的配置

- 中断路由
  - 中断产生后（多核情况下）硬件应该中断哪个 CPU 或哪些 CPU？
  - 这是我们需要告诉硬件的。
  - 树莓派比较特殊，所有的中断都会先发送到 GPU，再通过 GPU 来分配到具体哪些 CPU 上，详细请参考 BCM2836。

# 中断的配置

- 中断路由
  - 我们只会用到如下几种中断，为了便于实现，我们将所有全局的中断都路由到了第一个核（CPU0）：
    - UART 输入：在 qemu 中就是键盘输入，路由至 CPU0
    - 全局时钟（clock）：这是个时钟是固定频率的，在现实情况下用于记录系统时间或进行系统监控等，路由至 CPU0
    - 局部时钟（timer）：每个 CPU 都有的一种时钟中断，共四个，频率随 CPU 频率的变化而变化，用于记录进程的时间片，四个中断分别路由至四个 CPU
    - SD卡设备：之后文件系统部分会加入，路由至 CPU0
  - 上述已实现的中断的初始化和处理函数分别位于 `kern/uart.c`、`kern/clock.c`、`kern/timer.c`。

# Trap frame

- Trap frame 结构体的作用在于保存中断前的所有寄存器加上一些必要的中断信息。
- 在我们的实验中，需要关注
  - 30个通用寄存器的值（x1~x30）
  - 系统寄存器 ELR_EL1，SPSR_EL1，SP_EL0。

# 中断的处理

- 所有的合法中断会被 `kern/vectors.S` 发往 `kern/trapasm.S`，进而调用 `kern/trap.c:trap` 函数，这就是我们的中断处理函数
- 其中根据不同的中断来源而调用相应的处理函数。

# 中断的处理

- 测试中断
  - 如果想测试一下中断是否正确实现的话，我们需要手动在内核开启中断（后续请务必记得关闭，除非你想支持内核态中断），看看我们的 timer/clock/uart 是否能够正常中断，需要做如下修改：
    - 在 `kern/vectors.S` 中将 `verror(5)` 替换成 `ventry`，因为这一项代表在使用 SP_EL1 栈指针的 EL1 的 IRQ 中断。注意到 `kern/entry.S` 中 `msr spsel, #1` 指令已经将栈指针切换成了 SP_EL1，这就是我们的内核栈，之后用户进程使用的是 SP_EL0。
    - 在 `kern/main.c:main` 函数最后调用 `inc/arm.h` 中的 `sti` 函数开启中断。
  - 正常的话，qemu 上会有 timer/clock 的输出，输入字母的话也会显示在上面。

# Lab3 练习

- 请简要描述一下在你实现的操作系统中，中断时 CPU 进行了哪些操作。
- 请在 `inc/trap.h` 中设计你自己的 trap frame，并简要说明为什么这么设计。
- 请补全 `kern/trapasm.S` 中的代码，完成 trap frame 的构建、恢复。

# Lab4 多核与锁

- 回顾一下 lab3 中的一些问题：
  - 栈指针 16 Byte 对齐问题，只用 stp/ldp
  - c 和汇编的交互 trapframe 第一个成员是较低地址的
  - trapframe 是否可以只存 caller-saved 寄存器？
  - sp、SP_SEL、SP_EL1 的关系？
  - SP_EL1 不能直接访问，得先把 SP_SEL 置 1，然后访问 sp 的值
  - call trap(struct trapframe *tf) 中 tf 是 sp，trapframe 里面保存了一个用户栈指针 SP_EL0

# Lab4 多核与锁

- 本次实验我们会学习操作系统中多核和锁机制的相关知识，将会涉及如下几个部分：
- 了解 SMP 架构的定义
- 了解树莓派 3 上多核的初始状态以及如何开启（习题）
- 对之前实现的模块进行加锁以适配多核（习题）

# SMP 架构

- 树莓派以及大部分笔记本都是 SMP（symmetric multiprocessing） 架构
- 即所有的 CPU 拥有各自的寄存器（包括通用寄存器和系统寄存器），但共享相同的内存和 MMIO。
- 尽管所有 CPU 的功能都是一样的，我们通常会把第一个启动的、运行 BIOS 初始化代码的 CPU 称为 BSP（ bootstrap processor），其他的 CPU 称为 APs（application processors）
- 但树莓派比较奇葩，是从 GPU 启动的，即 GPU 完成了类似 BIOS 的硬件初始化过程，然后才进行 CPU 的执行
- 由于操作系统中有时候需要知道当前运行在哪个 CPU 上，ARMv8 用 MPIDR_EL1 寄存器来提供每个 CPU 自己的 ID。获取当前 CPU 的 ID 的方法见 `inc/arm.h:cpuid`。

# APs 的初始状态

- qemu 启动时，会把 armstub8.S 编译后的二进制直接拷贝到物理地址为 0 的地方
- 然后所有 CPU 均以 PC 为 0 开始（但有些 CPU 会被 armstub8.S 停住，只有一个 CPU 会跳到 0x80000，具体见下文）
- 而在真实的树莓派 3 上，armstub8.S 编译后的可执行代码已经被集成在了 start.elf 或 bootcode.bin 中，上板子时，我们需要把这两个文件拷贝到 SD 卡的第一个分区（必须是 FAT32），这样启动后 GPU 就会把 armstub8.S 的二进制代码从中抽取出来并直接拷贝到物理地址为 0 的地方，然后所有 CPU 均以 PC 为 0 开始

# APs 的初始状态

- armstub8.S 的伪代码如右所示
- 简言之，cpuid 为零 的 CPU（相当于 BSP）将直接跳转到 0x80000 的地方，而 cpuid 非零的 CPU（相当于 APs）将进行死循环直到各自的入口值非零，每个 CPU 的入口值在内存中的地址为 `0xd8 + cpuid()`。

```
extern int cpuid();

void boot_kernel(void (*entry)())
{
    entry();
}

void secondary_spin()
{
            void *entry;
    while (entry = *(void **)(0xd8 + (cpuid() << 3)) == 0)
        asm volatile("wfe");
    boot_kernel(entry);
}

void main()
{
    if (cpuid() == 0) boot_kernel(0x80000);
    else secondary_spin();
}
```

# 开启 APs

- 至于如何开启 APs，我们的 BSP 在 `kern/entry.S:_start` 中修改了 APs 的入口值并用 `sev` 指令唤醒它们
- 为了提高代码的复用性，入口值都被修改成了 `kern/start.S:mp_start`，
- 相当于所有 CPU 都会从这里开始并行执行类似 [Lab1 启动](lab1.md) 中的过程，只不过每个 CPU 都有不同的栈指针。
- 从本次实验开始，开启 APs 之后的所有代码都会被所有 CPU 并行执行，修改或读取全局变量时记得加锁。

# 同步和锁

- 在 SMP 架构下，我们通常需要对内存中某个位置上的互斥访问。
- ARMv8 为我们提供了一些简单的原子指令如读互斥 LDXR 和写互斥 STXR 来完成此类操作，但这些指令只能被用于被设定为 inner or outer sharable, inner and outer write-back, read and write allocate, non-transient 的 Normal Memory 中。
- 而我们确实在 `inc/mmu.h:PTE_NORMAL` 和 `inc/mmu.h:TCR_VALUE` 中进行了相应的配置，于是这类原子指令可以正常使用。
- 我们在 `kern/spinlock.c` 中用 GCC 提供的 atomic 宏实现了一种最简单的锁——自旋锁，相应的汇编代码可在导出的 `obj/kernel8.asm` 中找到。

# 原子操作

- 下面列出了几种常见的原子操作的伪代码
- Test-and-set (atomic exchange)

```
int test_and_set(int *p, int new)
{
    int old = *p;
    *p = new;
    return old;
}
```

- Fetch-and-add (FAA)

```
int fetch_and_add(int *p, int inc)
{
    int old = *p;
    *p += inc;
    return old;
}
```

# 原子操作

- Compare-and-swap (CAS)

```
int compare_and_swap(int *p, int cmp, int new)
{
    int old = *p;
    if (old == cmp)
        *p = new;
    return old;
}
```

# 自旋锁

```
struct spinlock {
    int locked;
};
void spin_lock(struct spinlock *lk)
{
    while (test_and_set(lk->locked, 1)) ;
}
void spin_unlock(struct spinlock *lk)
{
    test_and_set(lk->locked, 0);
}
```

# Ticket lock

```
struct ticketlock {
    int ticket, turn;
};
void ticket_lock(struct ticketlock *lk)
{
    int t = fetch_and_add(lk->ticket, 1);
    while (lk->turn != t) ;
}
void ticket_unlock(struct ticketlock *lk)
{
    lk-turn++;
}
```

# Ticket lock

- Ticket lock 中等待的 CPU 都在不停地访问同一个变量 `lk->turn`,
- 一旦锁被释放并修改 `lk->turn`，则需要修改所有等待中的 CPU 的 cache
- 当 CPU 数较多时，性能较差

# MCS 锁

- 这是一种基于队列的锁，对多核的缓存比较友好，并用 test-and-set 和 compare-and-swap 提高了效率

```
struct mcslock {
    struct mcslock *next;
    int locked;
};
void mcs_lock(struct mcslock *lk, struct mcslock *i)
{
    i->next = i->locked = 0;
    struct mcslock *pre = test_and_set(&lk->next, i);
    if (pre) {
        i->locked = 1;
        pre->next = i;
    }
    while (i->locked) ;
}
void mcs_unlock(struct mcslock *lk, struct mcslock *i)
{
    if (i->next == 0)
        if (compare_and_swap(&lk->next, i, 0) == i)
            return;
    while (i->next == 0) ;
    i->next->locked = 0;
}
```

# 读写锁

- 下面用两个自旋锁（或支持睡眠的 mutex）来实现读优先的读写锁

```
struct rwlock {
    struct spinlock r, w; /* Or mutex. */
    int cnt; /* Number of readers. */
};
void read_lock(struct rwlock *lk) {
    acquire(&lk->r);
    if (lk->cnt++ == 0)
        acquire(&lk->w);
    release(&lk->r);
}
void read_unlock(struct rwlock *lk) {
    acquire(&lk->r);
    if (--lk->cnt == 0)
        release(&lk-w);
    release(&lk->r);
}
void write_lock(struct rwlock *lk) {
    acquire(&lk->w);
}
void write_unlock(struct rwlock *lk) {
    release(&lk->w);
}
```

# 信号量

- 信号量（Semaphore）适用于控制一个仅支持有限个用户的共享资源，可以保证同一时刻最多有 cnt 个该资源被占用（cnt 的定义如下）。当 cnt 等于 1 时就是互斥锁（Mutex）。

```
struct semaphore {
    /* If cnt < 0, -cnt indicates the number of waiters. */
    int cnt;
    struct spinlock lk;
};
void sem_init(struct semaphore *s, int cnt)
{
    s->cnt = cnt;
}
void sem_wait(struct semaphore *s)
{
    acquire(&s->lk);
    if (--s->cnt < 0)
        sleep(s, &s->lk); /* Sleep on s. */
    release(&s->lk);
}
void sem_signal(struct semaphore *s)
{
    acquire(&s->lk);
    if (s->cnt++ < 0)
        wakeup1(s); /* Wake up one process sleeping on s. */
    release(&s->lk);
}
```

# Lab4 习题

- 为了确保你完全掌握了多核的启动流程，请简要描述一下 `kern/entry.S` 中各个 CPU 的初始状态如何、经历了哪些变化？至少包括对 PC、栈指针、页表的描述。
- 请阅读 `kern/spinlock.c` 并思考一下，如果我们在内核中没有关中断的话，`kern/spinlock.c` 是否有问题？如果有的话，应该如何修改呢？
- 注意到所有 CPU 都会并行进入 `kern/main.c:main`，而其中有些初始化函数是只能被调用一次的，请简单描述一下你的判断和理由，并在 `kern/main.c` 中加锁来保证这一点。

# Lab5 进程管理

- 在本次实验中，需要实现教学操作系统内核的进程管理部分，分为：
  - 内核的进程管理模块，负责进程的创建、调度（执行）
  - 内核的（简单）系统调用模块，负责响应用户进程请求的系统调用
- 最终实现将 `user/initcode.S` 作为第一个用户进程予以调度，并响应其执行过程中的系统调用。
- 建议在实验中先切换为单处理器模式，确保单处理器下所有功能均正常，再切换为多处理器模式，以将并发问题与其他功能性问题做到分离。

# 进程管理

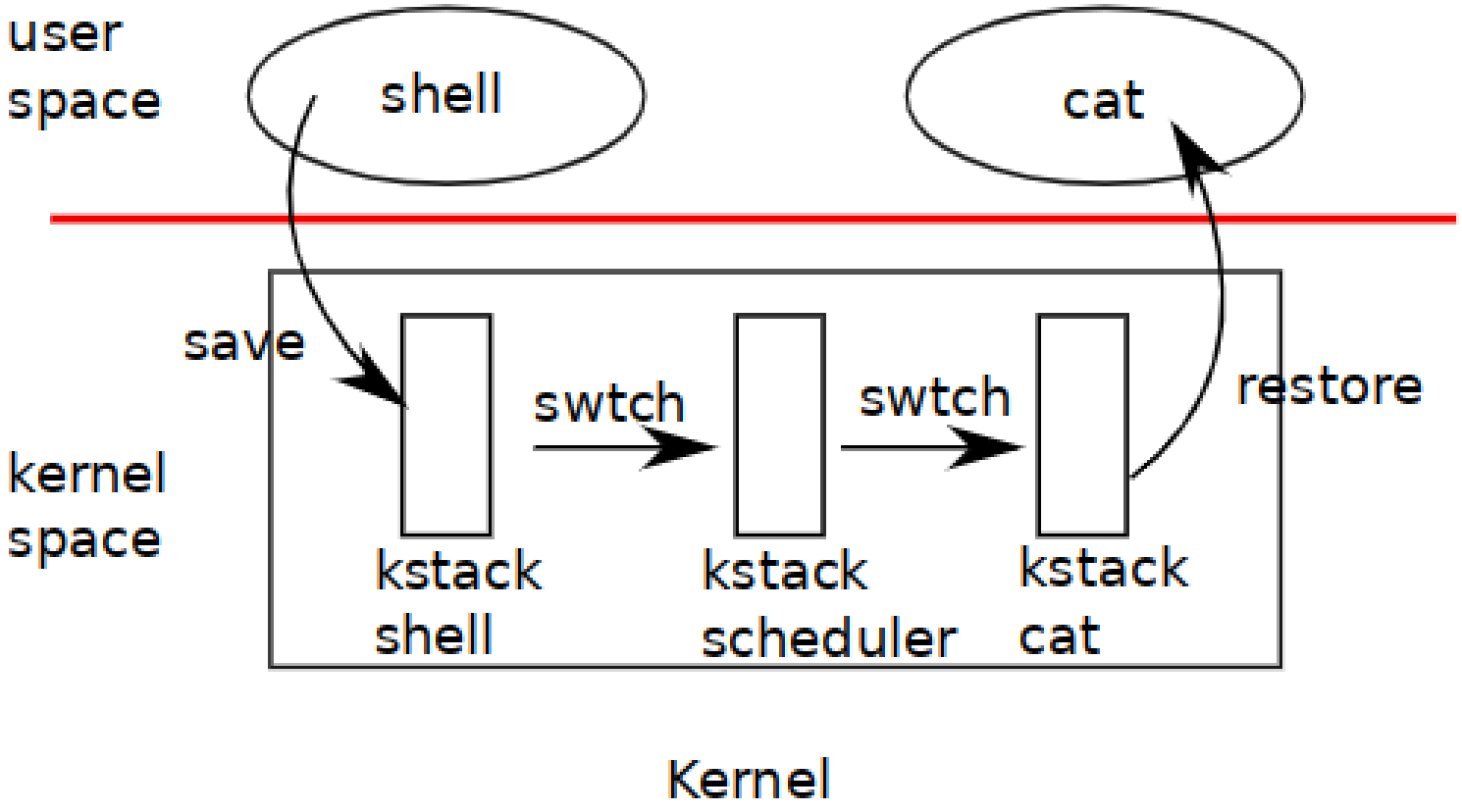
- 我们在 `inc/proc.h` 定义了进程的概念
- 本实验中仅需关注如下内容
  - 每个进程都拥有自己的内核栈，页表（与对应的地址空间）以及上下文信息。

```
struct proc {
    uint64_t sz;             /* Size of process memory (bytes)   */
    uint64_t *pgdir;         /* Page table */
    char *kstack;            /* Bottom of kernel stack for this process */
    enum procstate state;    /* Process state*/
    int pid;                 /* Process ID   */
    struct proc *parent;     /* Parent process  */
    struct trapframe *tf;    /* Trapframe for current syscall  */
    struct context *context; /* swtch() here to run process */
};
```

# Context switch

- context switch（即上下文切换）是操作系统多道程序设计（multitasking/multiprogramming）的重要组成部分
- 它实现了从一个进程切换到另一个进程
- 此时需要切换能准确描述当前进程在 CPU 上运行情况的上下文信息包括通用寄存器堆、运行时栈（即每个进程的内核栈）以及 PC（lr/x30 寄存器）。

# Context switch



# Context switch

- 当用户进程放弃 CPU 时
- 该进程的内核线程（kernel thread）会调用 `kern/swtch.S` 中的 `swtch` 来存储当前进程的上下文信息
- 并切换到内核调度器的上下文
- 每个进程的上下文信息均由 `struct proc` 的 `struct context*` 描述。
- 请完成 `inc/proc.h` 中 `struct context` 的定义以及 `kern/swtch.S` 中 context switch 的实现。

# 创建与调度进程

- 请根据 `kern/proc.c` 中相应代码的注释完成内核进程管理模块以支持调度第一个用户进程 `user/initcode.S`。
- 因为当前内核中还没有文件系统，我们通过修改 Makefile 文件将 `initcode.S` 链接进了 kernel 中。
- 大致流程为：
  - 首先将 `initcode.S` 编译、转换为 `obj/user` 下的二进制文件
  - 再与 `kern` 目录下编译出的内核代码共同链接成内核的可执行文件
  - `make qemu` 后可在 `obj/kernel8.hdr` 的 symbol table 中发现与之相关的三个 symbol：
    - _binary_obj_user_initcode_size
    - _binary_obj_user_initcode_start
    - _binary_obj_user_initcode_end
  - 来告知内核 `initcode` 的位置信息。

# 创建与调度进程

- 可参照如下顺序实现 `kern/proc` 下的函数
  - proc_init 完成 ptable 锁的初始化
  - user_init 初始化第一个用户进程 `user/initcode.S
    - proc_alloc 创建一个进程
    - pgdir_init 为进程创建一张页表
    - uvm_init 将进程的页表初始化
  - scheduler 内核开始调度
  - uvm_switch 为用户进程切换页表（ttbr0_el1）
  - swtch 从内核的调度器切换到用户进程
  - forkret 过渡，为以后的一些初始化留位置

# 系统调用

- 目前内核已经支持基本的异常处理，在本实验中还需要进一步完善内核的系统调用模块
- 用户进程通过系统调用来向操作系统内核请求服务以完成需要更高权限运行的任务
- 在 armv8 中，用户进程通过 `svc` 指令来请求系统调用，内核会根据 `ESR_EL1` 中的 `EC[31:26]` 以分派给相应的 handler 进行处理
- 用户进程在请求系统调用时，应告知内核相应的 system call number 以及系统调用所需的参数信息
- system call number 的宏定义可见 `inc/syscallno.h`。

# 系统调用

- 可参照如下顺序实现
  - 从 `alltraps` 跳转到 `kern/trap.c` 中的 `trap` 函数， 当因为系统调用而陷入时进入 `syscall` 中
  - 在 `kern/syscall.c` 中的 `syscall` 中根据 system call number 跳转到相应的 handler 中。
  - `sys_exec` 已经在 `kern/sysfile.c` 中实现，`sys_exit` 定义在了 `kern/sysproc.c` 中。
  - 为 kernel thread 添加一个 sys_yield 操作，当用户进程当前时间片耗尽时，kernel thread 可以通过 yield 让用户进程放弃占用 CPU

# 系统调用

- 在成功实现所有模块后，`make qemu` 在单处理器模式下应显示

```
main: [CPU0] is init kernel
main: Allocator: Init success.
irq_init: - irq init
main: [CPU0] Init success.
sys_exec: executing /init with parameters: /init
sys_exit: in exit
```

# 系统调用

- 在多处理器模式下应显示

```
qemu-system-aarch64 -M raspi3 -nographic -serial null -
serial mon:stdio -kernel obj/kernel8.img
main: [CPU1] is init kernel
main: Allocator: Init success.
irq_init: - irq init
main: [CPU1] Init success.
main: [CPU2] Init success.
main: [CPU3] Init success.
main: [CPU0] Init success.
sys_exec: executing /init with parameters: /init
sys_exit: in exit
```

# Lab5 习题

- 在 proc（即 PCB）中仅存储了进程的 trapframe 与 context 指针，请说明 trapframe 与 context 的实例存在何处，如果在 proc 中存储 `struct context context` 与 `struct trapframe tf` 应如何实现 trap 与 context switch?
- 请完成 `inc/proc.h` 中 `struct context` 的定义以及 `kern/swtch.S` 中 context switch 的实现。
- 在 `kern/proc.c` 中将 `swtch` 声明为 `void swtch(struct context **, struct context *)`，如果声明为 `void swtch(struct context *, struct context *)` 应如何实现？`context` 中仅需要存储 callee-saved registers，请结合 PCS 说明为什么？与 trapframe 对比，请说明为什么 trapframe 需要存储这么多信息？trapframe 似乎 已经包含了 context 中的内容，为什么上下文切换时还需要先 trap 再 switch?

# Lab6 驱动和 libc

- 本次实验中，我们会
  - 实现一个简易的 sd 卡驱动
  - 了解 MBR 分区格式
  - 学习 libc 与操作系统之间的接口

# I/O 框架

- 本节抽象地介绍了简易的基于队列的异步 I/O 框架

# 异步读写

- 对于设备驱动，往往需要实现对其的读写操作
- 而由于 I/O 速度较慢，为了避免阻塞当前 CPU，我们可以采用异步的方式来实现，如下
  - 读：向设备发送读请求和目标地址，然后等待中断直到设备完成读取，再读取设备返回的数据
  - 写：向设备发送写请求、目标地址以及待写入的数据，等待中断直到设备完成写入
- 当然其中还有很多细节需要考虑，如中断后清除相应的 flag 来避免重复的中断以及若设备发生错误中断则重新执行当前操作等。

# 请求队列

- 对于多个进程均想使用 I/O 的情况，则需要一个请求队列。
- 进程作为生产者，每次请求都将一个 buffer 入队，然后阻塞（睡眠）直到驱动来唤醒它
- 而 I/O 驱动作为消费者，每次取出并解析队首的 buffer，向设备发出具体的读写操作，完成后唤醒对应的进程。

# 块设备驱动

- 在块设备中，我们会用到如下几个简单的术语：
  - 块大小：每个块的大小，通常为 512 B
  - 逻辑区块地址（Logical block addressing, LBA）是从 0 开始的块下标，即第几块
- 对于内核的其他模块而言，异步的块设备驱动通常包括三个函数，
  - 初始化函数如 `sd_init`
  - 设备读写请求函数如 `sdrw`
  - 设备中断处理函数如 `sd_intr`。
- 设备的初始化比较硬件，暂且略过，让我们从读写函数开始讲起。

# 读写请求

- 这是给内核其他模块（如文件系统）使用设备的接口
- 其接受一个 buffer，然后进行类似上述异步读写中的步骤
- 在等待过程中我们会用 sleep/wakeup 来实现

# Sleep

- sleep 和 wakeup 是互斥锁的实现方式。
- 复习一下我们学过的条件变量，这是操作系统给用户程序提供的同步原语
- 在我们的 libc 中，其依赖的系统调用是 futex
- 见 `libc/src/thread/pthread_cond_*.c`，不过我们的内核还未实现这个系统调用
- 而这通常会在内核中类似 sleep、wakeup 的过程来完成进程的睡眠等待和唤醒。

# Sleep

- 这两个操作类似于条件变量的 wait和 signal 操作
- 其中 sleep 的函数声明是 `void sleep(void *chan, struct spinlock *lk)`
- 第一个是指需要等待的资源的地址
- 为什么这么设计?
  - 因为只要是内存中的资源，我们总能用它的地址来作为其唯一标识符
- 第二个是一个锁
- 用来保证从调用 sleep 开始到进程睡眠的过程是原子的
- wakeup 函数同理

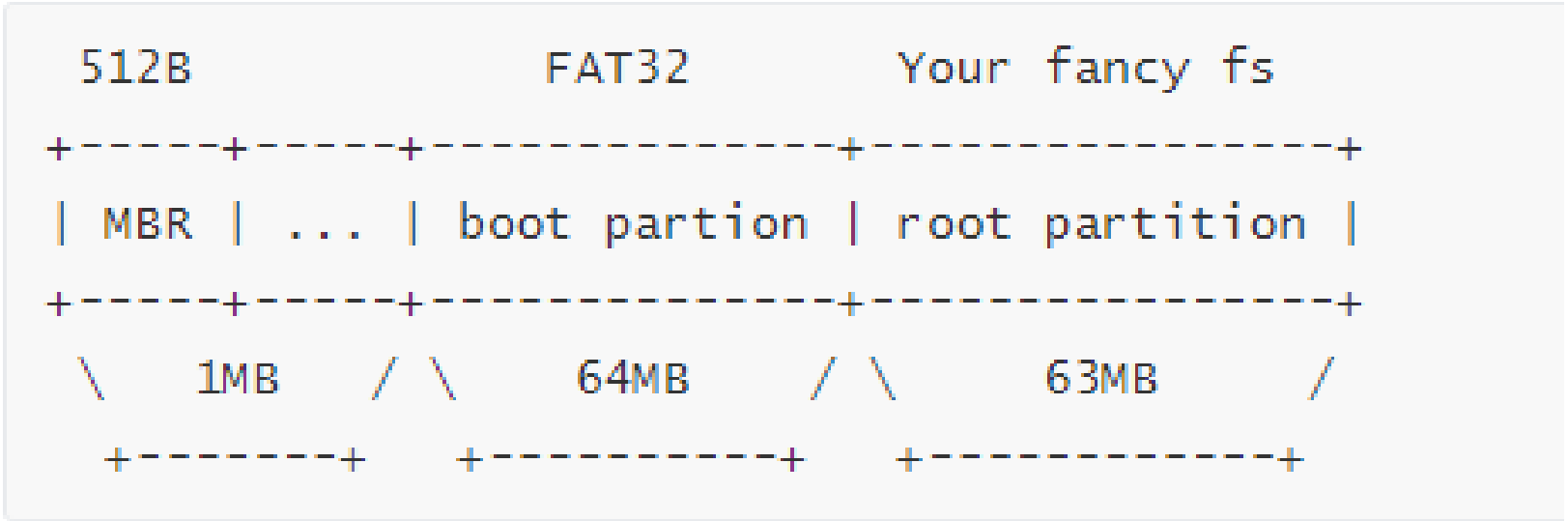
# 设备中断

- 设备中断的到来意味着 buffer 队首的请求执行完毕
- 于是我们会根据队首是请求类型进行不同的操作
  - 读请求，合法的中断说明数据已从设备中读出，我们将其从 MMIO 中读取到 buffer 的 data 字段上即可
  - 写请求，合法的中断说明数据已经写入了设备
- 然后清中断 flag，并继续进行下一个 buffer 的读写请求

# 制作启动盘

- 现代操作系统通常是作为一个硬盘镜像来发布的，而我们的也不例外
- 但在制作镜像时，需要注意遵守树莓派的规则
  - 即第一个分区为启动分区，文件系统必须为 FAT32，剩下的分区可由我们自由分配
- 为了简便，我们采用主引导记录（Master boot record, MBR来进行分区
- 第一个分区和第二个分区均约为 64 MB
- 第二分区是根目录所在的文件系统（我们会在后续 lab 完成这一部分）
- 简言之，SD 卡上的布局如下，具体见 `mksd.mk`

# 制作启动盘


512B
FAT32
Your fancy fs
MBR
...
boot partion
root partition
1MB
64MB
63MB

# MBR

- MBR 位于设备的前512B，有多种格式，不过大同小异
- 一种常见的格式如表

| Address | Description | Size (bytes) |
|---|---|---|
| 0x0 | Bootstrap code area and disk information | 446 |
| 0x1BE | Partition entry 1 | 16 |
| 0x1CE | Partition entry 2 | 16 |
| 0x1DE | Partition entry 3 | 16 |
| 0x1EE | Partition entry 4 | 16 |
| 0x1FE | 0x55 | 1 |
| 0x1FF | 0xAA | 1 |

# MBR

- 但这里我们只需要获得第二个分区的信息，即上表中的 Partition entry 2
- 这 16B 中有该分区的具体信息，包括它的起始 LBA 和分区大小（共含多少块）如表

| Offset (bytes) | Field length (bytes) | Description |
| --- | --- | --- |
| … | … | … |
| 0x8 | 4 | LBA of first absolute sector in the partition |
| 0xC | 4 | Number of sectors in partition |

# 启动分区

- 这是树莓派相关
- 我们用 mtools 把 `boot/` 目录下官方提供的固件和配置文件 `config.txt` 以及内核镜像 `obj/kernel8.img` 复制到了该分区里

# 根目录分区

- 这是我们的文件系统所在
- 下一个 lab 需要我们手动完成文件系统设计和构建
- 见 `user/src/mkfs`，它是一个命令行工具
  - `Makefile` 会调用 `user/Makefile`，后者会遍历 `user/src/XXX` 并用交叉编译器分别编译其目录下的所有文件为可执行文件 `user/bin/XXX`
  - `mksd.mk` 会将 `user/src/mksd` 用当前编译器编译到 `obj/mksd`，然后执行 `obj/mksd obj/fs.img user/bin/A user/bin/B ...`，即用 `user/bin/A`、`user/bin/B` 等文件来构建我们的第二分区镜像 `obj/fs.img`
  - 目前 `user/src/mksd` 的实现比较 naive，就是所有文件一个接一个地着写入 fs.img，后续实验中需要修改以符合自己设计的文件系统格式

# libc

- 由于重新实现 C 库是一项复杂的工作
- 于是我们迁移了一个完整的 C 库 musl作为一个 git submodule 以供使用
- 如此一来，我们只需要实现对应的系统调用，就能实现一个真正可用的操作系统了！

# 程序入口

- 在内核中 exec 启动一个用户态程序时，应该初始化 PC 到什么值呢？
- 也就是用户态执行的第一条指定在哪？
- 这个信息会被编译器编译到 ELF 头中，GCC 中默认是 _start 符号，可通过 linker script 中的 ENTRY 字段来自定义
- 内核的加载器（loader）通过解析这个 ELF 头即可得知

# crt

- C 运行时（C Runtime）是一段运行于 main 函数前后的代码片段
- gcc 会默认链接到可执行文件中（如 `libc/lib/crt1.o`）
- 这是和编译器相关的，其他语言也可能有自己的运行时，如 C++ 需要运行时的支持来完成异常处理、内存管理（全局的构造、析构函数）
- 关于 main 之前的运行时，我们需要依次关注如下几个文件
  - libc/arch/aarch64/crt_arch.h
  - libc/crt/crt1.c
  - libc/src/env/__libc_start_main.c
- 其中包含了一些初始化的代码，如 BSS 段清零、初始化 TLS
- 我们可以先修改 crt1.c 进行测试，因为完整的启动流程需要实现多个系统调用

# 系统调用

- 内核和 C 库通过系统调用交互，musl 中系统调用的实现在 `libc/arch/aarch64/syscall_arch.h`
- 对于内核来说，需要知道如下信息
  - syscall number：在 `libc/obj/include/bits/syscall.h` 中
  - syscall 函数声明：在 `libc/include/unistd.h`，调度相关的会在其他地方
- 我们以如何接入 printf 为例，我们需要知道它对应的系统调用是什么，在一切皆文件的 Unix 系统中，这其实就是向 stdout 文件写一些字符串
- 注意到 libc 帮我们完成了很多工作，如字符串 %d 格式化
- 至于这个文件是什么、具体的系统调用是什么，请自行在 `libc/src/stdio` 目录下全局搜索一下 stdout 后进行探索。

# Lab6 练习

- 请补全 `inc/buf.h` 以便于 SD 卡驱动中请求队列的实现，即每个请求都是一个 buf，所有请求排成一队。队列的实现可以手写，但建议将其单独抽成一个头文件，方便复用，具体请参考 list.h
- 请完成 `kern/proc.c` 中的 `sleep` 和 `wakeup` 函数，并简要描述并分析你的设计
- 请完成 `kern/sd.c` 中的 `sd_init`，`sd_intr`，`sdrw`，然后分别在合适的地方调用 `sd_init` 和 `sd_test` 完成 SD 卡初始化并通过测试
- 请在 `sd_init` 中解析 MBR 获得第二分区起始块的 LBA 和分区大小以便后续使用

# Lab7 文件系统和 Shell

- 在这一部分的实验中，你需要把之前写的所有代码都串联起来
- 并且添加大量新代码
- 以支持一个文件系统和一个简单的 shell

# 文件系统

- 在 Lab 6 中，我们已经实现了一个 SD 卡的驱动，并且可以实现以 block 为单位的原子读写
- 我们接下来就在这个驱动的基础上实现文件系统
- 简单来说，底层驱动为文件系统提供了一个线性寻址、单位为 section（512B）的可持久化存储空间
- 然而这个访问方式过于原始，没有提供足够的抽象给程序员，因此出现了以文件为单位的抽象

# 文件系统

- 这个抽象在 UNIX 操作系统中得到了发扬光大，出现了万物皆文件的统一界面
- 并通过系统调用，提供给了对程序员十分友好的编程范式
- 而操作系统内的文件系统的意义就在与弥合从底层驱动到这一层系统调用界面之间的空隙

# 设计

- 文件系统是一个复杂的系统
- 独立设计并实现难度过高
- 建议参考 xv6 中文件系统的设计

# 设计

- 文件系统大致可以分为七层，从下到上依次为
  - 硬盘驱动，在 Lab6 中已实现
  - Buffer cache，对存储介质上的单位空间做 buffer，并维护在内存中
  - 磁盘日志，用于维护系统调用的崩溃一致性
  - Inode层
  - 文件层
  - 系统调用层
  - 文件系统层

# Buffer Cache

- Buffer Cache 有两个任务：
  - 同步对磁盘块的访问，以确保内存中只有一个块的副本，并且一次只有一个内核线程使用该副本。
  - 缓存流行的块，这样它们就不需要从慢速磁盘中重新读取。

# 磁盘日志

- 如果系统崩溃并重新启动，文件系统代码将在任何进程运行之前从崩溃中恢复。
- 如果日志被标记为包含完整的操作，则恢复代码会将写入复制到它们在磁盘文件系统中所属的位置
- 如果日志未标记为包含完整操作，则恢复代码将忽略该日志
- 恢复代码通过擦除日志完成。
- 日志存放在从2号块开始的几个块上
- 它由一个 header block 和一系列更新的logged blocks组成
- Header block 包含一个扇区号数组，每个记录块一个
- Logged blocks 还包含记录块的计数

# Inode 层

- 术语 inode 有两种含义
  - 可能是指包含文件大小和数据块编号列表的磁盘数据结构
  - 或者，“inode”可能指的是内存中的 inode，它包含磁盘上 inode 的副本以及内核中所需的额外信息。
- 一个 Inode 对应一个包含大量磁盘块的文件

# 文件层

- 文件层抽象了文件、文件夹以及路径等定义
- 前面提到在 Unix 中大多数资源都可以表示成文件
- 而文件描述符便是操作系统访问、操作文件的接口
- 每个进程都有自己的文件描述符表，用于管理文件资源
- 文件描述符和文件描述符表使得同一个文件能够被多个进程打开
- 并且能够对可读、可写字段进行追踪

# 系统调用层

- 这一层实现了一些和文件系统相关的系统调用，例如：
  - Open
  - Read
  - Write
  - Link
  - Unlink
  - Stat

# Shell

- Shell 是用户与类 UNIX 操作系统的交互界面
- 因此如果想要真正作为用户体验自己实现的操作系统，你必须为其实现一个 shell。

# 启动 Shell

- Shell 的本质是一个用户态程序
- Shell 是由 init fork 出来的
- 其 stdout、stderr 也是由 init 进程通过 `mknod` + `dup` 来手动配置的
- 于是之后 fork 出来的各个进程都会继承这些文件描述符
- 其启动流程如下：
  - `user_init()` 启动 `initcode.S`
  - `initcode.S` 执行 exec 系统调用将自身替换为 `user/src/init` 中的 init 进程
  - init 进程不断地 fork 启动 `user/src/sh` 中的 shell 进程

# Lab7 练习

- 请实现文件系统，并通过修改 Makefile 添加新的编译项（如 `make testfs`），并使用 `-DTEST_FILE_SYSTEM` 的条件编译指令在代码中实现无侵入的测试
- 请修改 `syscall.c` 以及 `trapasm.S` 来接上 musl，或者修改 Makefile，从而允许用户态程序通过调用系统调用来操作文件系统。
- 我们已经实现了简易 shell，但需要实现 brk 系统调用来使用 malloc，你也可以自行实现一个简单的 shell。请在 `user/src/cat` 中实现 cat 命令并在你的 shell 中执行。

# 第三次迭代

- 本实验的前两次迭代是整个内核实验中较为基础的内容，需要同学们按照框架内容来搭建操作系统内核
- 而第三次迭代则为内核实验保留了扩展性和更多的可能
- 第三次迭代是自选主题的扩展实验，是整个实验内容中的可选项
- 虽然第三次迭代并不限制具体主题，但是为了提供一些参考和方向，本章节将给出一些演进方向和指导。

# Buddy System

- 回顾 Lab 2 中的内存管理，我们使用了非常简单的链表式内存分配器
- 该内存分配器的优点非常明显，易于理解和实现，非常适合作为前期的实验内容
- 然而，链表式的内存分配器除此之外，可以说是没有其他优点了
- 使用该内存分配器之后，当系统经过长时间的运行，不停地分配内存、回收内存之后，整个内存将会变成大大小小的若干块
- 严重时会出现有足够的空余内存总量，但不能分配出足够大的连续内存片段。这种情况我们将其称之为内存碎片化

# Buddy System

- 内存碎片化是一个十分有历史的问题
- 远在 Linux 2.X 版本时，就有非常多的开发者致力于缓解内存碎片化的问题
- 直到 Linux 3.10 版本，Linux 内核引入了 Buddy System(伙伴系统内存分配器)，用于预防和缓解内存碎片化。

# 内存碎片指数

- 在这次实验之前，我们需要完成一个小实验，用来直观地感受和验证内存碎片化的过程和结果
- 在这一部分内容中，你需要完成以下内容:
  - 完成内存碎片相关指数的监视，至少包括:
    - 总内存大小
    - 总空闲内存大小
    - 连续内存的块数
    - 最大的连续空闲内存大小
  - 通过上述参数，我们能够对内存碎片情况有一个大概的感受
  - 通常来讲，在空闲内存大小一定的情况下，连续内存块数越多，碎片化越严重
  - 同理，在连续内存的块数相同的情况下，总空间内存越小，碎片化越严重
  - 而最大的连续空闲内存大小则决定了能分配的最大的内存块大小。

# 内存碎片指数

- 实现一段用户态代码，用于使得操作系统不停的申请、释放内存，用来模拟操作系统长时间运行
- 在执行一段时间后，通过查看在上一个步骤中添加的指数，来感受内存碎片化。

# Buddy System

- Buddy System 的本质是通过一种特殊的分配方式来减少外内存碎片的存在，以此来达到缓解碎片化的目的
- 这种特殊的分配方式便是将内存以2的幂进行划分，通过一组链表来维护不同大小的内存块，并保证每个链表中的内存块大小相同
- Buddy System 有以下特点：
  - 分配和释放的代价很低
  - 没有外碎片
  - 但无法避免内碎片，不过内碎片可以在应用层以内存池的方式来减少

# Buddy System 分配算法

- 寻找到最小的大于等于所需大小的内存块，假设大小为 2^n
  - 存在 -> 分配结束
  - 不存在
    - 若当前是最大的内存块，则返回失败
    - 否则寻找大小为 $2^{n+1}$的内存块
    - 取其中一半用于分配，另一半存放在链表中

# Buddy System 释放算法

- 释放该内存块
- 查看相邻的内存块，若同样被释放，则合并他们，重复该步骤直到不满足条件，或者达到最大内存块大小
- 需要注意的是，在 Buddy System 算法中，查看相邻内存块是一个 O(1) 的操作
- 这里使用了一个小技巧，通过翻转编号的最后一个二进制位来快速定位“相邻块”
- 他们也就是伙伴系统中的伙伴所代表的含义

# Buddy System 总结

- 不难发现，不管是分配还是释放，时间复杂度都是 O(logN)
- 基于此，需要继续完成本次实验的后半部分：
  - 实现 Buddy System 来替换链式内存分配器。
  - 采用与之前相同的代码和测试，对内存碎片相关参数进行对比，以验证 Buddy System 的效果

# 文件系统数据校验与测试

- 在 Lab7 中，我们只要求实现文件系统，但是并没有完整的文件系统正确性测试
- 在这里，我们设计了一个文件系统测试框架，用于验证文件系统的正确性。

# 测试设计

- 这里采用了较为简单的测试方法:
  - 首先将根目录下所有的文件列出来，同时列出文件的大小
  - 读取并打印 /TEST 文件中的内容，之后进行随机重复测试:
    - 创建目录 /rand_test
    - 创建文件 /rand_test/test_file
    - 在文件内写入随机内容
    - 最后打开 /rand_test/test_file 检查文件内容是否正确

# 文件系统数据校验

- 通常，为了验证文件内容与预期相同，我们会通过使用 MD5 校验的方式来检查。
- MD5 的本质是一种哈希算法，虽然 MD5 有着一定的缺陷，但目前仍然在一定程度上被使用。
- 这里我们在文件系统中集成一个 MD5 校验算法来对文件内容进行校验
- 并将其加入到上述阐述的测试框架中。

# 进程管理优化

- 回顾 Lab 5，我们的简易内核中实现了基于轮询的进程调度器
- 轮询调度器原理简单、实现容易、易于理解
- 但是，轮询调度器同样无法应对复杂、多变的需求
- 特别是对于个人通用计算机操作系统的需求无法满足
- 因此，如果希望追求应用，或者吞吐率等参数，优化进程管理是非常直观的方式

# 多级反馈队列调度算法

- 更换进程调度算法是非常直接的一种想法
- 相比于轮询而言，多级反馈队列调度算法是一种更加优秀的调度算法
- 它能够满足不同类型进程的需求，在一定程度上保证高优先级进程的响应速度，同时又能使的短作业进程被较快完成，同时也不会使得长作业进程被一直阻塞或者被不停打断
- 多级反馈队列调度算法是从先来先服务(FCFS)队列优化而来的
- 它通过对不同的进程分类，将其放入不同的队列中进行调度，同时控制不同队列的时间片的长度来达到目的。

# 多级反馈队列调度算法

- 按照 FCFS，设立 n 个队列，这 n 个队列有不同的优先级，假设第一个队列优先级最高
- 为每个队列设置时间片长度，满足优先级越高的队列时间片长度越短
- 当进程首次进入调度器时，将其放入最高优先级的队列中
- 当需要从调度器中选择进程时，按照优先级依次选择队列
- 同一队列中的进程按照时间片轮转的调度策略
- 若某一进程花费完了分配给它的时间片，并且仍然还未执行结束，则将其放入低一优先级的队列中，同理类推

# 练习

- 实现多级反馈队列调度算法替换轮询调度器
- 为内核实现多种调度算法共存的接口，使得能够通过编译参数来选择不同的调度算法
- 在哪种场景下多级反馈队列调度算法比轮询调度器更加优秀？尝试编写用户态程序复现具体场景，并通过两种调度算法的参数对比来验证你的猜想

# 内存顺序杂谈

- 回顾 lab 4 中的一些问题：
  - 虽然开了多核，但内存只有一个，编译后内核还是只有一份的，就放在 0x80000
  - 关于 BSS 段，通常编译后 BSS 段是不会在可执行文件中的，而是只在 ELF 头中标一个范围，其初始化是由加载此程序的 loader 来完成的
  - 习题三，main.c 需要保证 memset 只执行一次，这需要锁，于是这个锁（全局变量）需要显示初始化一下，这样就不会被编译到 BSS 段中了（因为 BSS 段中的值是不确定的），如 `struct spinlock my_lock = {0};`
  - 寄存器是每个 CPU 都有一套的，于是对于系统寄存器的操作是每个 CPU 都要执行一次的
  - 我们目前还在 EL1（内核态），并未进入 EL0

# 内存顺序杂谈

- 现代计算机体系对于内存访问进行了玩命优化，于是有了如下两个层面的优化
  - 编译器优化，可能会导致编译产生的汇编代码的乱序
  - 处理器优化，包括指令乱序执行、cache 一致性

# 内存顺序杂谈

- 可以看出这两者优化都是基于乱序
- 但完全乱序显然不行
- 于是为了便于编程并保证正确性
- 大部分编译器和处理器会遵循一定的原则，即「不能修改单线程的行为」(Thou shalt not modify the behavior of a single-threaded program).
- 但在多线程的程序中会有一些问题，例如下述程序执行后 r1、r2 可能同时为 0。

# 内存顺序杂谈

X = 0, Y = 0;

Thread 1:
X = 1;
r1 = Y;

Thread 2:
Y = 1;
r2 = X;

# 内存顺序杂谈

- 于是我们需要显式的告诉编译器（C++ 内存模型，最终也是通过处理器提供的指令来实现）或处理器（指令屏障和内存屏障）一些额外的信息
- 因为编译器和处理器并不知道哪些数据是在线程间共享的，这是需要我们手动控制的

# 内存顺序杂谈

- 在大多数弱内存模型（简称WMO，Weak Memory Ordering）的架构上，可通过汇编指令来告诉处理器，我们期望的内存访问读取的顺序。ARMv8 上用屏障来实现，有如下三种屏障相关的指令：
  - Instruction Synchronization Barrier (ISB)：清空处理器中的指令流水，可确保在 ISB 指令完成后，才从 cache 或内存中读取位于 ISB 指令后的其他所有指令，通常作为修改系统寄存器（如 MMU）时的屏障
  - Data Memory Barrier (DMB)：在当前 shareability domain 内，该指令之前的所有内存指令一定先于其后的内存指令，但对非内存指令没有要求
  - Data Synchronization Barrier (DSB)：在当前 shareability domain 内，该指令之前的所有内存指令一定先于其后的内存指令，其后的所有指令均需要等到 DSB 指令完成后才能执行

# 内存顺序杂谈

- ARMv8 还提供两种内存操作中常用的单向屏障（One-way barriers）如下
- 单向屏障的性能优于之前的 DMB，且天然适合锁的实现。