Lab 1 系统软件启动过程

练习1：理解通过make生成执行文件的过程

- 操作系统镜像文件 *ucore.img* 是如何一步一步生成的?

  1. 将kern下的文件编译链接生成bin/kernel
  2. 将boot下的文件编译生成bin/bootblock
  3. 使用dd命令创建ucore.img，并将bootblock和kernel输出到ucore.img

- 一个被系统认为是符合规范的硬盘主引导扇区的特征是什么?

  tools.sign.c中指定了主引导扇区大小为512字节，且最后两个字节为0x55AA。

练习2：使用qemu执行并调试lab1中的软件

练习3：分析bootloader进入保护模式的过程

0. 以Intel 80386为例，计算机加电后从0xFFFFFFF0（位于BIOS ROM（EPROM））执行一条跳转执行跳到BIOS程序，BIOS完成计算机硬件自检和初始化，选择一个启动设备将第一个扇区（主引导扇区）读取到内存一个特定地址0x7c00处，CPU再跳转到这个地址继续执行bootloader。
1. （boot/bootasm.S）通过修改A20地址线完成从实模式到保护模式的转换。首先等待8042控制器input buffer为空，然后将0xd1写入0x64端口表示要向8042的P2端口写数据，随后再次等待8042的input buffer为空，将0xdf写入0x60端口，即将A20位置1。
2. `lgdt` 指令加载GDT全局描述符表，将cr0寄存器的PE位置1。使用长跳转指令 `ljmp` 跳转到protcseg，进入32位模式。

练习4：分析bootloader加载ELF格式的OS的过程

boot/bootmain.c

从第二个扇区开始读取8个扇区的ELF header，随后依次读取每个代码段到内存指定位置，最后跳入entry处。

练习5：实现函数调用堆栈跟踪函数

完成 `print_stackframe` 函数（kern/debug/kdebug.c）

循环打印 *STACKFRAME_DEPTH* 个栈信息

练习6：完善中断初始化和处理

完善 `idt_init` 函数和 `trap_dispatch` 函数（kern/trap/trap.c）

扩展练习 Challenge 1

通过中断实现内核态/用户态的转换。

中断过程：

1. 如果当前CPU为ring 3，需要将内核栈的SS0和ESP0恢复到SS和ESP，并将原先用户栈的SSt、ESPt压栈。
2. 压栈EFLAGS、CS、EIP。
3. 如果是异常，将异常码error code压栈；如果是中断，关中断。
4. 跳入中断程序入口。

由于从内核态使用int指令时，不会保存ss和esp寄存器，但iret时会pop两个寄存器的值，故可以先将esp值减8，再调用int。在中断处理部分，设置中断帧。

扩展练习 Challenge 2

使用键盘实现用户模式/内核模式切换。

键盘中断 *IRQ_KBD* 处理中，调用 `switch_to_user` / `switch_to_kernl` 。

Lab 2 物理内存管理

练习1：实现first-fit连续物理内存分配算法

完善 `default_init` 、 `default_init_memmap` 、 `default_alloc_pages` 、 `default_free_pages` 函数（kern/mm/default_pmm.c）

first_fit算法维护一个按地址排序的空闲块链表，空闲块起始页Page中的property设置为该空闲块的页数。

练习2：实现寻找虚拟地址对应的页表项

完善 `get_pte` 函数（kern/mm/pmm.c）

找到虚拟地址对应的二级页表项PTE，若二级页表不存在则分配一个二级页表。

PDE/PTE的低12位为属性位，包括存在位、读写权限位、特权级等，高20位为页地址的高20位。

当发生页访问异常时，硬件将发生错误的线性地址保存在cr2寄存器，产生中断进行page_fault处理。

练习3：释放某虚地址所在的页并取消对应二级页表项的映射

完善 `page_remove_pte` 函数（kern/mm/pmm.c），当页引用计数减为0，调用 `free_page` 释放页，并将页表项置0。

kernel中的虚拟地址与物理地址线性映射，偏移为 `KERNBASE` 。如果希望虚拟地址与物理地址相等，可将 `KERNBASE` 置为0。

Lab 3 虚拟内存管理

练习1：给未被映射的地址映射上物理页

完成 `do_pgfault` 函数（kern/mm/vmm.c）

若对应页不存在，则调用 `pgdir_alloc_page` 分配一个物理页。

- 请描述页目录项（Page Directory Entry）和页表项（Page Table Entry）中组成部分对 ucore实现页替换算法的潜在用处。

  PDE/PTE的高20位为页地址的高20位，低12位为属性位：第0位为存在位，第1位表示是否允许读写，第2位表示该页访问需要的特权级，第3位表示是否使用write through缓存写策略，第4位表示是否不缓存该页，第5位表示该页是否被访问过，第6位为脏位，第7位为页属性表位等，其中替换算法可以利用的信息包括是否被访问、脏位，以及保留的未被系统使用的位用于记录替换算法需要的信息（如访问次数等）。

- 如果ucore的缺页服务例程在执行过程中访问内存，出现了页访问异常，请问硬件要做哪 些事情?

  CPU会将产生异常的线性地址存储在CR2中，并把页访问异常类型的值保存在中断栈中。CPU在当前内核栈保存当前被打断的程序现场，即依次压入当前被打断程序使用的EFLAGS，CS，EIP，errorCode；由于页访问异常的中断号是0xE，CPU把异常中断号0xE对应的中断服务例程的地址（vectors.S中的vector14）加载到CS和EIP寄存器中，开始执行中断服务例程。

练习2：补充完成基于FIFO的页面替换算法

完成 `do_pgfault` 函数（kern/mm/vmm.c）， `map_swappable` 、 `swap_out_victim` 函数（kern/mm/swap_fifo.c）

维护一个先进先出的链表， `map_swappable` 、 `swap_out_victim` 分别将目标page插入或移除该链表。

`do_pgfault` 中，对于换出而导致的缺页，调用 `swap_in` 将page读回内存，使用 `page_insert` 建立虚拟地址的映射，再调用 `swap_map_swappable` 将page标记为swappable。

若要实现 *extended clock* 页替换算法，可以利用PTE中的访问位和脏位。

Lab 4 内核线程管理

内核线程：只运行在内核态，所有内核线程共用内核空间。

要让内核线程运行，首先需要创建内核线程对应的 `进程控制块` ，并通过链表连在一起，即进程控制块链表，便于进程管理。然后通过 `调度器` 来让不同内核线程在不同的时间段占用CPU执行。

`kern_init` 函数完成虚拟内存初始化后，调用 `proc_init` 函数，完成 `idleproc` 和 `initproc` 两个内核线程的创建或复制工作。

- `idleproc` ：不停地查询是否有其他内核线程可以执行，有则调度
- `initproc` ：显示"Hello World"

`proc_run` 函数，设置当前进程PCB为 *proc* ，设置esp0内核栈，加载cr3寄存器，并调用 `switch_to` 完成进程切换。

练习1：分配并初始化一个进程控制块

实现 `alloc_proc` 函数（kern/process/proc.c），分配并返回一个新的 `struct proc_struct` （PCB），完成最基本的初始化。

PCB中， `struct context` 用于保存运行上下文（除了eax之外的寄存器），用于进程切换； `struct trapframe` ，记录进程被中断前的状态，从中断返回时，能调整中断帧恢复寄存器的值。

练习2：为新创建的内核线程分配资源

实现 `do_fork` 函数（kern/process/proc.c），创建新的内核线程。 `do_fork` 创建当前内核线程的一个副本，执行上下文、代码、数据都一样，但存储位置不同。

`do_fork` 调用 `get_pid` 时使用了关中断保护线程不会被打断，以安全地获得唯一pid。

Lab 5 用户进程管理

第二个内核线程 `init_main` 通过 `kernel_thread` 创建 `user_main` 内核线程。 `user_main` 通过 `kernel_execve` 调用来把某一具体程序的执行内容放入内存。一旦执行了程序对应的进程，就会从内核态切换到用户态继续执行。

练习1：加载应用程序并执行

完成 `load_icode` 函数（kern/process/proc.c）

过程：

1. 第二个内核线程 `init_main` 通过 `kernel_thread` 创建 `user_main` 内核线程。
2. `user_main` 通过 `kernel_execve` 调用 `SYS_exec` 系统调用，最终调用 `do_execv` 函数完成用户进程的创建。将 ~~hello~~ exit应用程序执行代码。 `do_execv` 工作流程：
    1. 清空准备用户态内存空间（mm不为NULL时，此处mm为NULL）

2. 调用 `load_icode` 加载应用程序执行代码到当前进程的用户态虚拟空间。包括读ELF格式文件，申请内存空间，建立用户态虚存空间，拷贝应用程序各个段的内容到内核虚拟地址，创建用户栈，设置中断帧（实验编码部分）
3. 系统调用返回后，执行中断返回指令 `iret` 后，CPU转到用户态特权级，回到用户态内存空间，从用户进程的第一条语句（initcode.S中的_start处）开始执行。

练习2：父进程复制自己的内存空间给子进程

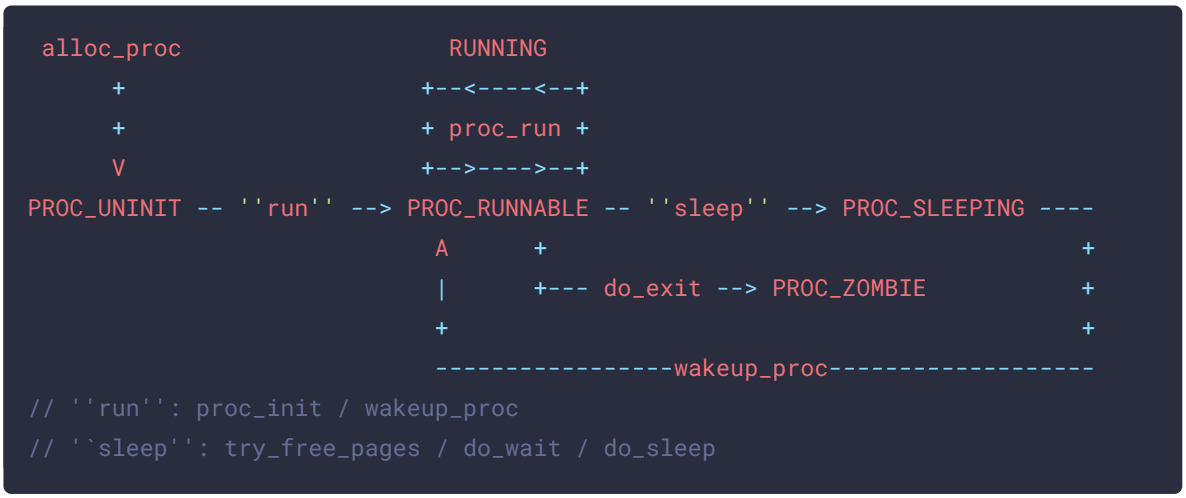
更新 `do_fork` （kern/process/proc.c），补充 `copy_range` 实现（kern/mm/pmm.c）

Copy-on-write机制： do_fork内存复制部分，不进行实际的内存复制，而是将虚拟页映射到同一个物理页面，并设置成不可写与共享。当子进程试图写该页，则page fault部分进行实际页复制。

练习3：阅读分析源代码，理解进程执行 fork/exec/wait/exit 的实现，以及系统调用的实现

- fork：创建子进程的PCB，复制内存空间，调用 `wakeup_proc` 将子进程标记成 *PROC_RUNNABLE* 。
- exec：见上文。
- wait：搜索处于 *PROC_ZOMBIE* 态的子进程，若有则对子进程进行最终的资源回收工作（内核堆栈、PCB）；否则进入 *PROC_SLEEPING* 态，睡眠原因为 *WT_CHILD* ，调用 `schedule` 函数切换。
- exit：回收当前进程的用户态虚拟空间，标记为 *PROC_ZOMBIE* 态。若父进程为等待子进程状态 *WT_CHILD* ，则 `wakeup_proc` 唤醒父进程，帮助自己完成最后的资源回收。若当前进程还有子进程，将子进程的父进程指针设置为内核线程 `initproc` ，如果某个子进程状态为 *PROC_ZOMBIE* ，则唤醒 `initproc` 。调用 `schedule` 。

进程状态转移：



系统调用

用户程序调用系统调用库函数，调用 `syscall` 函数，通过参数系统调用号区分具体系统调用。通过 *INT T_SYSCALL* 发起，CPU从用户态切换到内核态，保存现场到当前进程的中断帧，并跳转到 __vectors[T_SYSCALL]开始执行。完成服务后，操作系统按调用路径原路返回，恢复现场，最终通过 *IRET* 返回。

需要将trap.c中时钟中断的 `print_ticks` 注释，在make grade中会触发panic

Lab 6 调度器

进程切换过程：

1. 进程A执行时出现一个trap（如外设产生的中断），从进程A的用户态切换到内核态，并保存好进程A的trapframe。
2. 内核态处理中断时发现需要进行进程切换时，通过 `schedule` 函数选择下一个进程B，调用 `proc_run` 函数， `proc_run` 进一步调用 `switch_to` ，切换到进程B的内核态。
3. 通过 *iret* 指令，将执行权转交给进程B的用户空间。

练习1：使用Round Robin调度算法

调度框架 `struct sched_class`

```
struct sched_class {
    // 调度器的名字
    const char *name;
    // 初始化运行队列
    void (*init)(struct run_queue *rq);
    // 将进程p插入队列rq
    void (*enqueue)(struct run_queue *rq, struct proc_struct *proc);
    // 将进程p从队列rq中移除
    void (*dequeue)(struct run_queue *rq, struct proc_struct *proc);
    // 返回运行队列中下一个可执行的进程
    struct proc_struct *(*pick_next)(struct run_queue *rq);
    // timetick处理函数
    void (*proc_tick)(struct run_queue *rq, struct proc_struct *proc);
};
```

Round Robin调度算法：

- `RR_enqueue` ：将进程的PCB指针放入到rq队尾，如果时间片为0，重置为 *max_time_slice* 。
- `RR_dequeue` ：将PCB指针的队列元素删除，并将就绪进程个数 *proc_num* 减一。
- `RR_pick_next` ：选取rq队头元素。
- `RR_proc_tick` ：将PCB的时间片减一，若降到0，则设置 *need_resched* 为1。

多级反馈队列调度算法：

多个不同的队列，具有不同的优先级，优先级越高的队列时间片越短。

enqueue：若进程的时间片用完，则加入到下一个优先级的队列中，否则加入到当前优先级队列。

pick_next：优先从高优先级队列中选择进程。

练习2：实现Stride Scheduling调度算法

- init：初始化调度器类的信息，初始化当前的运行队列
- enqueue：初始化刚进入运行队列的进程的 *stride* 属性，将进程加入运行队列
- dequeue：从运行队列删除相应元素
- pick_next：返回 *stride* 最小的进程，更新 *stride* ，$\text{pass} = \frac{\text{BIG\_STRIDE}}{\text{P->priority}}$，$\text{p->stride} \mathrel{+}= \text{pass}$
- proc_tick：检测当前进程是否已经用完时间片，如果用完则正确设置相关标记引起进程切换

ubuntu 20.04 LTS、qemu 4.2版本，参考答案也无法成功运行，在check_pagefault()中无法触发page fault，导致后续free page触发assert

Lab 7 同步互斥

练习1：理解内核级信号量的实现和基于内核级信号量的哲学家就餐问题

内核级信号量相关实现位于kern/sync/sem.[ch]。

- 信号量包括一个整型变量和一个等待队列。信号量操作通过开关中断保护原子性。
- `sem_init`：初始化信号量。
- `down`：P操作。若当前信号量大于0，则将其减1后返回；否则调用 `wait_current_set` 将当前进程置为 *PROC_SLEEPING* 并加入到等待队列，再调用 `schedule` 。当被V操作唤醒时，调用 `wait_current_del` 把自身关联的wait从等待队列中删除。
- `up`：V操作。若等待队列为空，则将信号量值加1后返回；否则调用 `wakeup_wait` 唤醒等待队列第一个线程。
- `try_down`：失败不进入等待队列的P操作。

基于内核信号量的哲学家就餐问题：创建5个哲学家对应的5个信号量，5个内核线程对应5个哲学家，每个内核线程完成思考、吃饭、睡觉操作。

练习2：完成内核级条件变量和基于内核级条件变量的哲学家就餐问题

实现内核级条件变量相关函数（kern/sync/monitor.c）

- wait_cv（`cond_wait`）：某进程等待某个条件不为真，执行该函数，需要睡眠。将睡眠进程个数 *cv.count* 加一。

    - 若 *monitor.next_count* 大于0，表示有进程因为执行了 `cond_signal` 而睡眠在 *monitor.next* 信号量上，因此调用 `up` 唤醒 *next* 的等待队列中的一个进程。
    - 否则需要调用 `up` 唤醒互斥条件 *monitor.mutex* 限制而无法进入管程的进程。

  然后调用 `down` 将自己睡在 *cv.sem* 上。被唤醒后，将 *cv.count* 减一。

- signal_cv（`cond_signal`）：若 *cv.count* 不大于0，则说明没有需要唤醒的进程，直接返回。若 *cv.count* 大于0，则表示有执行 `cond_wait` 而睡眠的进程，因此调用 `up` 唤醒睡在 *cv.sem* 上的进程，并将 *monitor.next_count* 加一，调用 `down` 将自己睡在 *monitor.next* 上，当被唤醒后，将 *monitor.next_count* 减一。

Lab 8 文件系统

0. 需要修改包括PCB初始化（proc.c）

练习1：完成读文件操作的实现

补充 `sys_io_nolock` 函数（kern/fs/sfs/sfs_inode.c）：逐块读取数据，对于头尾与块不对齐的情况，调用 `sfs_buf_op` ，对完整的块调用 `sfs_block_op` 完成io。

UNIX的PIPE机制：可建立临时文件，作为进程之间的PIPE载体。临时文件可仅存在于内存中，通过VFS实现。

练习2：完成基于文件系统的执行程序机制的实现

改写 `load_icode` 函数（proc.c）

1. 为当前进程创建mm
2. 创建PDT
3. 复制TEXT/DATA/BSS到进程内存空间
4. 调用 `mm_map` 创建用户栈
5. 设置当前进程的mm、cr3、pgidr
6. 在用户栈设置uargc和uargv
7. 设置中断帧
8. 如果失败，进行清理工作

软硬链接：

- 硬链接：文件指向相同的inode，inode中的nlinks / ref_count相应增加。
- 软链接：软链接为链接类型的文件，有单独的inode，文件的数据块指向被链接文件的inode。