lab10 Final Lab

本次实验大家将完成最后的整合，实现各种 system call 与最后的 shell。

需要完成的内容中，特殊标记的含义

（T）工具函数，如果使用别的实现可以不写这个

（O）不实现相关功能也可以启动shell，如果时间来不及，请优先保证你能启动shell

Console

> 本模块内的相关问题推荐联系陈立达助教。

存在一片内核缓冲区和记录已经被读取的(r)，已经写入的(w)，光标所在正在写的(e)。

```
struct {
    char buf[INPUT_BUF];
    usize r;  // Read index
    usize w;  // Write index
    usize e;  // Edit index
} input;
```

数组是环形数组。数组满时不能继续写入。遇到 `\n` 和 `ctrl+D` 时将未写入部分变为写入，即更新w到e处。

TODO 你需要完成 `kernel/console.c` 中的下列函数。

console_intr

处理串口中断。使用 `uart_put_char` 和 `uart_get_char` 来完成相关操作。

编辑（写入，backspace）缓冲区内容，并回显到console。

使用 `uart_get_char` 获取串口读入字符，完成下列情况：

1. `backspace`：删除前一个字符，`input.e--`。但是位于w前的已写入字符不能删除。（思考如何回显出删除的效果）。会用到 `uart_put_char('\b')`，效果是光标向左移动一位。
2. `Ctrl+U`：删除一行。
3. `Ctrl+D`：更新 `input.w` 到 `input.e`。表示 `EOF`。本身也要写入缓冲区和回显。
4. `Ctrl+C`：杀死当前程序。
5. 普通字符写入和回显。

回显会用到`uart_put_char`向console写入内容。

可以适当自定义`Ctrl+<字母>`。

console_write

`isize console_write(Inode *ip, char *buf, isize n)`

将`buf`中的内容在console显示。

要锁`inode ip`。返回n。

console_read

`isize console_read(Inode *ip, char *dst, isize n)`

读出console缓冲区的内容n个字符到`dst`。遇见`EOF`提前结束。返回读出的字符数。

Pipe

> 本模块内的相关问题推荐联系陈立达助教。

```
typedef struct pipe {
    SpinLock lock;
    Semaphore wlock,rlock;
    char data[PIPESIZE];
    u32 nread;  // number of bytes read
    u32 nwrite;  // number of bytes written
    int readopen;  // read fd is still open
    int writeopen;  // write fd is still open
} Pipe;
```

TODO 请完成`fs/pipe.c`中的下列函数。

pipeAlloc（O）

`int pipeAlloc(File** f0, File** f1)`

创建`pipe`并初始化，创建`pipe`两端的`File`放入`f0`,`f1`,分别对应读和写。

成功返回0，失败返回-1.

pipeClose（O）

`void pipeClose(Pipe* pi, int writable)`

关闭`pipe`的一端。如果检测到两端都关闭，则释放`pipe`空间。

pipeWrite（O）

`int pipeWrite(Pipe* pi, u64 addr, int n)`

向`pipe`写入n个byte，如果缓冲区满了则sleep。返回写入的byte数。

pipeRead（O）

`int pipeRead(Pipe* pi, u64 addr, int n)`

从`pipe`中读n个byte放入addr中，如果`pipe`为空并且writeopen不为0，则sleep，否则读完pipe，返回读的byte数。

sysfile.c：pipe2（O）

分配的`pipe`，并将`pipe`的`f0`,`f1`放入第一个参数指向的地址。

File Descriptor

> 本模块内的相关问题推荐联系杜雨轩助教。

TODO 请完成`fs/inode.c`、`fs/file.c`和`kernel/sysfile.c`中标注为TODO的内容，详细要求见代码注释，以下清单供参考。

```
inode.c:
namex

file.c:
init_ftable
init_oftable
filealloc
filedup
fileclose
filestat
fileread
filewrite

sysfile.c:
fd2file
fdalloc
sys_close
create
sys_chdir
```

fork+exec & file mapping

> 本模块内的相关问题推荐联系唐思源助教。

进行本部分任务前，建议先完成File Descriptor部分任务。

TODO

- `kernel/syscall.c: user_readable user_writeable`

  检查syscall中由用户程序传入的指针所指向的内存空间是否有效且可被用户程序读写

- `kernel/proc.c: fork TODO`
- `kernel/exec.c: execve TODO`
- `kernel/pt.c: copyout TODO` (T)
- `kernel/paging.c: copy_sections TODO` (T)
- `kernel/sysfile.c: mmap TODO` (O)
- `kernel/sysfile.c: munmap TODO` (O)
- `kernel/paging.c init_sections`: 不需要再单独初始化 heap 段了

- `kernel/paging.c pgfault`：增加有关文件的处理逻辑（O）

fork

> 创建一个子进程拷贝（替身），一般和exec同时调用
>
> 从进程的结构体入手，依次判断其中的变量是否需要复制，是否需要修改
>
> 为了配合fork，你可能需要在原先的usercontext中加入所有寄存器的值。此外，你还需要保存tpidr0和q0，因为musl libc会使用它们。

elf

相关头文件：`musl/include/elf.h`

```
typedef struct {
unsigned char e_ident[EI_NIDENT];
    Elf64_Half    e_type;
    Elf64_Half    e_machine;
    Elf64_Word    e_version;
    Elf64_Addr    e_entry;
    Elf64_Off     e_phoff;
    Elf64_Off     e_shoff;
    Elf64_Word    e_flags;
    Elf64_Half    e_ehsize;
    Elf64_Half    e_phentsize;
       Elf64_Half    e_phnum;
       Elf64_Half    e_shentsize;
    Elf64_Half    e_shnum;
    Elf64_Half    e_shstrndx;
} Elf64_Ehdr;

typedef struct {
    Elf64_Word    p_type;
    Elf64_Word    p_flags;
    Elf64_Off p_offset;
    Elf64_Addr    p_vaddr;
    Elf64_Addr    p_paddr;
    Elf64_Xword   p_filesz;
    Elf64_Xword   p_memsz;
    Elf64_Xword   p_align;
} Elf64_Phdr;
```

> 需要用到以上两个结构体，各个参数的作用可以查看 https://en.wikipedia.org/wiki/Executable_and_Linkable_Format，按照 exec 中给出的流程使用即可
>
> 注意：elf文件中的 section header 和lab的 struct section并不一样，本次实验不用考虑 section header 的情况

exec

> 替换当前进程为filename所指的elf格式文件，并开始运行该文件（变身）
>
> 需要思考进程的哪些部分需要释放，哪些部分不需要

```
int execve(const char* filename, char* const argv[], char* const envp[])
//filename: 标识运行的可执行文件
//argv: 运行文件的参数（和 main 函数中的 argv 指的是一个东西）
//envp: 环境变量

//  execve 异常返回到可执行文件的入口，即elf.e_entry（可以认为就是以下格式的main函数），而在main函数看来依然是一般的函数调用
int main(int argc, char *argv[])
//入口函数，其中argc表示参数的数量，argv表示参数的指针数组，比如ls ..  其中 argc=2，argv[0]:"ls", argv[1]:".."

//example code
char * argv[ ]={"ls","..",(char *)0};
char * envp[ ]={"PATH=/bin",(char*)0};
if(fork()==0)
    execve("ls", argv, envp);
else

//user stack structure
/*
 * Step1: Load data from the file stored in `path`.
 * The first `sizeof(struct Elf64_Ehdr)` bytes is the ELF header part.
 * You should check the ELF magic number and get the `e_phoff` and `e_phnum` which is the starting byte of program header.
 *
 * Step2: Load program headers and the program itself
 * Program headers are stored like: struct Elf64_Phdr phdr[e_phnum];
 * e_phoff is the offset of the headers in file, namely, the address of phdr[0].
 * For each program header, if the type(p_type) is LOAD, you should load them:
 * A naive way is
 * (1) allocate memory, va region [vaddr, vaddr+filesz)
 * (2) copy [offset, offset + filesz) of file to va [vaddr, vaddr+filesz) of memory
 * Since we have applied dynamic virtual memory management, you can try to only set the file and offset (lazy allocation)
 * (hints: there are two loadable program headers in most exectuable file at this lab, the first header indicates the text section(flag=RX) and the second one is the data+bss section(flag=RW). You can verify that by check the header flags. The second header has [p_vaddr, p_vaddr+p_filesz) the data section and [p_vaddr+p_filesz, p_vaddr+p_memsz) the bss section which is required to set to 0, you may have to put data and bss in a single struct section. COW by using the zero page is encouraged)

 * Step3: Allocate and initialize user stack.
 * The va of the user stack is not required to be any fixed value. It can be randomized. (hints: you can directly allocate user stack at one time, or apply lazy allocation)
 * Push argument strings.
 * The initial stack may like
 *   +-------------+
 *   | envp[m] = 0 |
 *   +-------------+
```

```
 *   |    ....     |
 *   +-------------+
 *   |   envp[0]   |  ignore the envp if you do not want to implement
 *   +-------------+
 *   | argv[n] = 0 |  n == argc
 *   +-------------+
 *   |    ....     |
 *   +-------------+
 *   |   argv[0]   |
     +-------------+
     |    argv     | pointer to the argv[0]
 *   +-------------+
 *   |    argc     |
 *   +-------------+  <== sp
 * (hints: the argc and argv will be pop to x0 x1 registers in trap return)

 * ## Example
 * sp -= 8; *(size_t *)sp = argc; (hints: sp can be directly written if current pgdir
is the new one)
 * thisproc()->tf->sp = sp; (hints: Stack pointer must be aligned to 16B!)
 * The entry point addresses is stored in elf_header.entry
*/
```

在exec时，一般可执行文件中可以加载的只有两段：RX的部分+RW的部分（其它部分会跳过）（因此只设置两种状态，一种是RX，另一种是RW）

- RX的部分： 代码部分，可以设置一个段为 SWAP+FILE+RO，此时需要“打开”对应的可执行文件，这样才能对其进行引用
- RW的部分：数据部分，包括了data+bss段，因此没办法分开设置成两个section（WHY？），因此也不能做成file-backed的段（bss段不是file-backed的），可以直接写入物理地址，并设置对应的一个段为 RW

copyout（T）

> 复制内容到给定的页表上，在exec中，为了将一个用户地址空间的内容复制到另一个用户地址空间上，可能需要调用这样的函数，

```
/*
 * Copy len bytes from p to user address va in page table pgdir.
 * Allocate a file descriptor for the given file.
 * Allocate physical pages if required.
 * Takes over file reference from caller on success.
 * Useful when pgdir is not the current page table.
 * return 0 if success else -1
 */
int copyout(struct pgdir* pd, void* va, void *p, usize len)
```

mmap munmap（O）

相关定义`musl/include/sys/mman.h`

> 参考讲解：彻底理解mmap()_Holy_666的博客-CSDN博客_mmap
>
> 参考代码：XV6学习（15）Lab mmap: Mmap - 星見遥 - 博客园

```
void *mmap (void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```

可以利用我们之前实现的 section 数据结构

Q&As

- 之前page fault实验中，swapin和swapout的并发相关测试太弱，有没有专门的测试？考虑到时间、复杂程度以及最终的效果，本次实验最后不会测试并发相关的内容，相当于简化了（~~其实也测不出来~~，因为相对于内存而言，磁盘太小了），也就是说，此前的page fault实验的swap部分仅要求根据当时的文档学习了解即可
- 那是不是说swap flag就没用了？并不是，现在swap flag依然可以标识该段是否在磁盘上，只不过这次只有file-backed section的形式，没有了以anonymous section的形式存在于磁盘上的部分
- 进程结构体中的 stksz，sz，base 等等的作用是什么？取决于是否实现对应的动态内存分配（比如将stack作为一个段section，那么就可以不管stksz这个变量，将由section中的begin和end变量标识），没有的话目前可以不管（或者用来debug）

Shell

> 本模块内的相关问题推荐联系金润杰助教。

TODO

完成两个用户态程序

- `user/cat/main.c` （O）
- `user/mkdir/main.c` （O）

完成`kernel/core.c: kernel_entry`，需要返回到用户态执行`user/init.S`。

测试

编译用户态程序需要先使用`make libc -j`编译musl libc，编译一次后后面即可直接使用。

当你完成全部任务后，如果一切顺利，将进入一个shell，你可以自己运行里面的程序。我们也编写了一个usertests程序可供运行（你可以自己改一改来加强测试）。

请注意助教最终不一定会只使用公布的测试程序来测试你的代码。

Final Lab的提交时间为1月20日，如有特殊情况请提前联系助教，一般不接受迟交。