lab3

第一部分：实现用户进程

练习1：在kernel/process/thread.c、kernel/sched/context.c和kernel/sched/sched.c中，请补全以下函数，以实现对第一个用户进程和线程的创建和执行：

- load_binary：解析 ELF 文件，并将其内容加载到新线程的用户内存空间中。
- init_thread_ctx：初始化线程的上下文，以便启动当前线程。
- switch_context：切换到当前线程的上下文。

  见代码

练习2：请简要描述process_create_root这一函数的逻辑。注意：描述中需包含thread_create_main函数的详细说明，建议绘制函数调用图以描述相关逻辑

process 先调用 ramdisk_read_file 读 ramdisk 的文件，然后调用 process_create 创建进程，再调用 thread_create_main 创建线程，然后用 obj_get 取得线程并用 obj_put 把线程加入 ready queue 中

thread_create_main 是用来创建线程的，它先调用 obj_get 和 obj_put 取得虚拟空间，随后为线程分配并设置一个用户栈，然后使用 obj_alloc 初始化线程、填充线程所需的参数，最后刷新 L1 的 icache 和 dcache 并返回创建的线程

process_create_root 的函数调用图：

- ramdisk_read_file
- process_create
- thread_create_main
  - obj_get & obj_put
  - obj_alloc & pmo_init & cap_alloc & vmspace_map_range
  - obj_alloc & load_binary & prepare_env & thread_init & cap_alloc
  - flush_idcache
- obj_get
- obj_put

第二部分：异常处理

练习3：完善异常处理，需要阅读与修改kernel/exception/下的exception_table.s、exception.S和exception.c。需要修改的内容包括：

- 修改exception_table.S的内容，可借助该文件中的某些宏，填写异常向量表。
- 完成exception.c中的exception_init函数，使得 kernel 启动后能够正确设置异常向量表。
- 修改异常处理函数，使得当发生异常指令异常时，让内核使用 kinfo 打印在esr.h中定义的宏UNKNOWN的信息，并调用sys_exit函数中止用户进程。

  见代码

第三部分：系统调用和缺页异常

练习4：和其他异常不同，ChCore中的系统调用是通过使用汇编代码直接跳转到syscall_table中的相应条目来处理的。请阅读kernel/exception/exception_table.S中的代码，并简要描述ChCore是如何将系统调用从异常向量分派到系统调用表中对应条目的

```
EXPORT(el1_vector)
        exception_entry sync_el1t
        exception_entry irq_el1t
        exception_entry fiq_el1t
        exception_entry error_el1t

        exception_entry sync_el1h
        exception_entry irq_el1h
        exception_entry fiq_el1h
        exception_entry error_el1h

        exception_entry sync_el0_64
        exception_entry irq_el0_64
        exception_entry fiq_el0_64
        exception_entry error_el0_64

        exception_entry sync_el0_32
        exception_entry irq_el0_32
        exception_entry fiq_el0_32
        exception_entry error_el0_32
```

os根据el1_vector中的不同值，会进行不同的操作。当异常类型为sync_el0_64时，会调用到el0_syscall，在el0_syscall中会把系统调用从异常向量分派到系统调用表中对应条目

练习5：在user/lib/syscall.c中完成syscall这一用户库函数，在其中使用SVC指令进入内核态并执行相应的系统调用

见代码

练习6：在ChCore中完成以下系统调用

- sys_putc：printf所使用的基本系统调用，需要使用uart_send标准输出一个字符
- sys_exit：具有退出当前用户线程的功能，需要将对应编号的系统调用分派到sys_exit这一函数上
- sys_create_pmo：用于测试缺页异常，实现在vm_syscall.c
- sys_map_pmo：用于测试缺页异常，实现在vm_syscall.c
- sys_handle_brk：用户线程将使用sys_handle_brk创建或扩展用户堆。通过这一系统调用，当前进程的堆将被扩大至虚拟地址addr

见代码

练习7：请使用 GDB 检查START函数执行结束后程序计数器的值，解释为何会发生这一现象，并尝试在 ChCore 的异常处理器中处理对应类型的异常

```
(gdb) b START
Breakpoint 1 at 0x400130
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, 0x0000000000400130 in START ()
(gdb) n
Single stepping until exit from function START,
which has no line number information.
0x0000000000400148 in _start_c ()
(gdb) n
Single stepping until exit from function _start_c,
which has no line number information.
0x0000000000400110 in main ()
(gdb) n
Single stepping until exit from function main,
which has no line number information.
0x0000000000000000 in ?? ()
(gdb) n
Cannot find bounds of current function
(gdb) display /20i $pc
1: x/20i $pc
=> 0x0: <error: Cannot access memory at address 0x0>
```

- START函数执行结束后程序计数器的值为0x0
- 发生这一现象的原因可能是：START调用了_start_c，_start_c继续调用了main，当调用结束后应该返回父函数或调用sys_exit，但是既没有父函数也没有调用exit，于是pc指向了0x0
- 修改了kernel/mm/vm_syscall.c中的sys_handle_brk函数，以及kernel/syscall/syscall.c中的sys_putc函数和常量syscall_table

练习8：在合适的地方添加对sys_exit的调用，使用户主线程能够正常退出。一般而言，退出当前线程后，内核中的调度器会将当前线程移出调度队列，并继续从调度队列中选择下一个可执行的线程进行执行。然而，由于目前 ChCore 中仅包含一个线程，因此，在唯一的用户线程退出后，ChCore将中止内核的运行并直接退出。后续实验会对这一问题进行修复

见代码

练习9：完成缺页异常处理，具体而言，在函数handle_entry_c中将缺页异常转发给函数do_page_fault()处理，并完成do_page_fault()和handle_trans_fault()函数

见代码