操作系统 Lab4实验报告

实验思考题

Thinking 4.1

- 内核在保存现场时会将所有通用寄存器，CP0寄存器和当前PC寄存器的值保存在栈中，但通用寄存器中的 `k0(26)` 和 `k1(27)` 两个寄存器是保留给中断或自陷处理程序使用的，其值可能在眼皮子底下改变。因此，内核使用 `k0` 保存栈指针的值，使用 `k1` 帮助更新栈指针的值，使用 `v0` 存储非通用寄存器的值，只更改了这三个寄存器中的值，从而实现了对绝大部分通用寄存器的保护。
- 可以，内核在保存现场的过程中只修改过 `k0` 、 `k1` 和 `v0` 的值， `a0` 至 `a3` 这四个参数寄存器的值不会受到影响。
- 函数的参数通过 `a0` 至 `a3` 这四个参数寄存器传递，而在系统调用的过程中我们保证这四个参数寄存器的值不变，同时将多出来的两个参数压入栈中，使得用户栈能够完整地复制到内核栈空间中，进而保证了 `sys` 开头的函数得到了正确的参数。
- 在处理系统调用的过程中修改了 `Trapframe` 中 `cp0_epc` 的值，从而保证处理完系统调用之后能够回到用户态中正确的位置继续执行。

Thinking 4.2

`mkenvid` 函数实现如下：

```
u_int mkenvid(struct Env *e) {
    u_int idx = e - envs;
    u_int asid = asid_alloc();
    return (asid << (1 + LOG2NENV)) | (1 << LOG2NENV) | idx;
}
```

函数的返回值中 `(1 << LOG2NENV)` 一项实际上为 `(1 << 10)` ，保证了返回值非零。

在 `envid2env` 函数中，若传入的参数 `envid` 为 `0` ，则会返回当前进程的进程块 `curenv` ，方便其他源文件在无法直接访问 `curenv` 时获取当前进程的进程快。在实现 IPC 过程中，若需要当前进程块发送或接收数据，则可以在相关函数的参数 `envid` 中传入 `0` ，即可实现。

Thinking 4.3

- 说明父子进程共享代码段，且具有相同的数据和状态。
- 说明子进程在创建时保存了父进程的上下文状态、程序计数器PC等，因此会和父进程同时向下继续执行，而不会转而执行 `fork()` 之前父进程的代码。

Thinking 4.4

- 关于fork 函数的两个返回值，下面说法正确的是：**C**
  - A、fork 在父进程中被调用两次，产生两个返回值
  - B、fork 在两个进程中分别被调用一次，产生两个不同的返回值
  - C、fork 只在父进程中被调用了一次，在两个进程中各产生一个返回值
  - D、fork 只在子进程中被调用了一次，在两个进程中各产生一个返回值

Thinking 4.5

- 对于用户进程，`UTOP` 以上的空间对于用户来说都是不可写的，因此不需要使用 `duppage` 进行映射。
- `UXSTACKTOP - BY2PG` 到 `UXSTACKTOP` 之间的空间是用户进程的异常栈，用来执行页写入异常的处理程序，若在此使用 `duppage` 进行映射可能产生页写入异常不断循环。
- `USTACKTOP` 到 `USTACKTOP + BY2PG` 之间的空间是无效内存，因此也不需要使用 `duppage` 进行映射。
- 除去以上空间，需要对于剩余的具有有效位的空间使用 `duppage` 进行映射。

Thinking 4.6

- `vpd` 是指向用户页目录所在虚拟地址的指针，在使用时可以将其作为数组头，通过取数组元素来访问页目录；`vpt` 是指向用户页表所在的虚拟地址的指针，在使用时可以将其作为数组头，通过取数组元素来访问页表。对于虚拟地址 `va`，`(*vpd)[va >> 22]` 为 `va` 对应的页目录项，`(*vpt)[va >> 12]` 为 `va` 对应的页表项。
- `vpd` 和 `vpt` 在 `user/entry.S` 中被定义，`vpt` 指向 `UVPT`，即用户页表的虚拟地址；`vpd` 指向 `(UVPT+(UVPT>>12)*4)`，即用户页目录的虚拟地址。在 `include/mmu.h` 中，它们被声明为了指针数组的形式，因此进程通过数组名所代表的基地址和数组元素编号代表的偏移量即可获得自身的页表。
- `vpt` 指向 `UVPT` 而 `vpd` 指向 `(UVPT+(UVPT>>12)*4)`，即体现了自映射的设计。
- 不可以，在用户态下不能修改页表项。

Thinking 4.7

- 在用户进程发生写时复制引发的缺页中断并进行处理时，可能会再次发生缺页中断，从而“中断重入”。
- 由于MOS操作系统在用户态下实现页写入异常的处理，因此用户进程需要读取 `Trapframe` 中的数据，同时在中断结束恢复现场时也需要用到 `Trapframe` 中的数据，因此需要将其拷贝到用户空间。

Thinking 4.8

- 在用户态处理页写入异常符合微内核的设计理念，尽量减少了内核出现错误的可能，即使程序崩溃也不会影响到系统自身的稳定。同时，在微内核的设计模式下，用户态进行页面的分配和映射也更加灵活方便。
- 在 `include/stackframe.h` 中的 `RESTORE_SOME` 宏中实现了现场恢复的代码，首先修改了 `CP0_STATUS` 寄存器的值，然后利用 `v0` 和 `v1` 寄存器恢复非通用寄存器的值，最后通过 `sp` 寄存器恢复通用寄存器的值，在此过程中只修改了通用寄存器中的 `v0`、`v1` 和 `sp` 寄存器，因此保证了恢复后通用寄存器值的正确性。

```
.macro RESTORE_SOME
        .set    mips1
        mfc0    t0,CP0_STATUS
        ori     t0,0x3
        xori    t0,0x3
        mtc0    t0,CP0_STATUS
        lw      v0,TF_STATUS(sp)
        li      v1, 0xff00
        and     t0, v1
        nor     v1, $0, v1
        and     v0, v1
        or      v0, t0
        mtc0    v0,CP0_STATUS
```

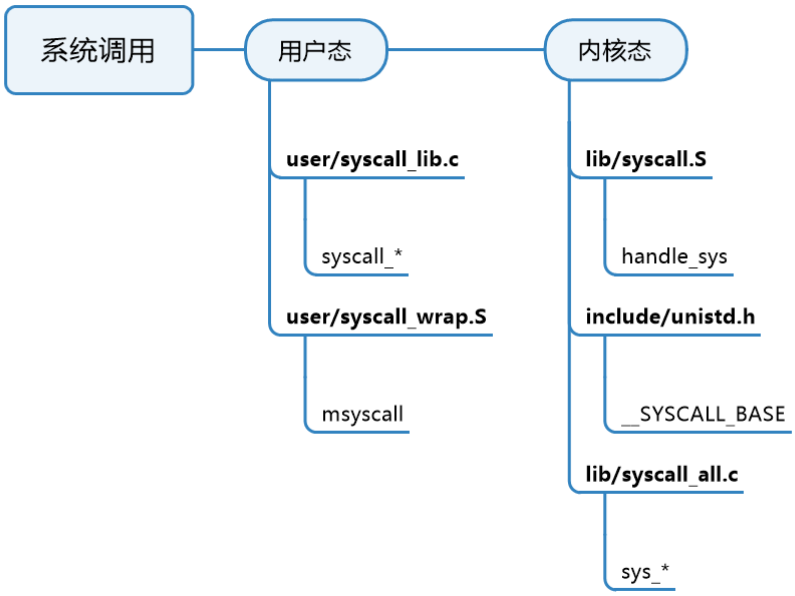
```
        lw      v1,TF_LO(sp)
        mtlo    v1
        lw      v0,TF_HI(sp)
        lw      v1,TF_EPC(sp)
        mthi    v0
        mtc0    v1,CP0_EPC
        lw      $31,TF_REG31(sp)
        lw      $30,TF_REG30(sp)
        lw      $28,TF_REG28(sp)
        lw      $25,TF_REG25(sp)
        lw      $24,TF_REG24(sp)
        lw      $23,TF_REG23(sp)
        lw      $22,TF_REG22(sp)
        lw      $21,TF_REG21(sp)
        lw      $20,TF_REG20(sp)
        lw      $19,TF_REG19(sp)
        lw      $18,TF_REG18(sp)
        lw      $17,TF_REG17(sp)
        lw      $16,TF_REG16(sp)
        lw      $15,TF_REG15(sp)
        lw      $14,TF_REG14(sp)
        lw      $13,TF_REG13(sp)
        lw      $12,TF_REG12(sp)
        lw      $11,TF_REG11(sp)
        lw      $10,TF_REG10(sp)
        lw      $9,TF_REG9(sp)
        lw      $8,TF_REG8(sp)
        lw      $7,TF_REG7(sp)
        lw      $6,TF_REG6(sp)
        lw      $5,TF_REG5(sp)
        lw      $4,TF_REG4(sp)
        lw      $3,TF_REG3(sp)
        lw      $2,TF_REG2(sp)
        lw      $1,TF_REG1(sp)
.endm
```

Thinking 4.9

- 因为在执行 `syscall_env_alloc` 的过程中也可能发生页写入异常，需要相应函数进行异常处理。
- 如果将 `set_pgfault_handler` 放置在写时复制保护机制完成之后，此时进程给 `__pgfault_handler` 赋值时会触发缺页中断，但此时中断处理函数还没有注册，因此无法正常执行。
- 不需要，因为子进程继承了父进程中对 `__pgfault_handler` 的赋值。

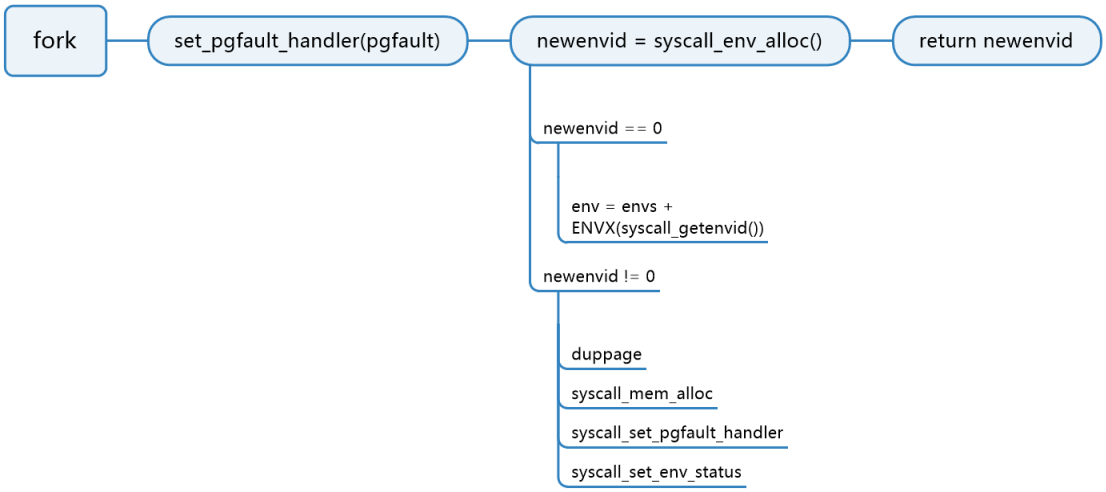
体会感想

系统调用



- 系统调用是用户实现内核级操作的方式，也是用户和内核之间的通信机制之一。
- 在MOS中，用户进程通过调用 `user/syscall_lib.c` 中的函数进行系统调用，用户态的系统调用函数通过调用msyscall函数陷入内核态，之后通过 `handle_sys` 函数向内核传递信息并分发内核态的系统调用函数，最后进入对应函数处理用户请求。

`fork`



- `fork` 函数为当前进程创建了一个子进程，子进程开始运行时的大部分上下文状态与原进程相同，包括程序镜像、通用寄存器和程序计数器 PC 等。
- 在新进程（子进程）中， `fork()` 调用的返回值为0，而在旧进程（父进程）中，同一调用的返回值是子进程的进程ID，即 `env_id` 。这一设计实质上是在 `fork()` 函数中所调用的 `syscall_env_alloc()` 实现的：

```c
int sys_env_alloc(void)
{
    // Your code here.
    int r;
    struct Env *e;

```

```
    r = env_alloc(&e, curenv->env_id);
    if (r)
        return r;
    bcopy((void *)KERNEL_SP - sizeof(struct Trapframe),
          (void *)(&(e->env_tf)),
          sizeof(struct Trapframe));
    e->env_status = ENV_NOT_RUNNABLE;
    e->env_pri = curenv->env_pri;
    e->env_tf.pc = e->env_tf.cp0_epc;
    e->env_tf.regs[2] = 0;

    return e->env_id;
    //  panic("sys_env_alloc not implemented");
}
```

  - `syscall_env_alloc()` 首先将当前进程的运行现场（进程上下文）`Trapframe` 复制到到子进程的进程控制块中，同时设置了子进程的进程控制块中 `env_tf.pc` 的值，使子进程被唤醒时也能够从父进程陷入异常的后一条开始执行，从而使得父子进程同时开始执行 `fork` 之后的代码段。
  - `syscall_env_alloc()` 还将子进程中该函数的返回值人为设置为0，这样在 `fork` 函数中便能够区分父子进程，从而对进程进行不同的设置。
- `fork` 函数中父进程在创建了一个子进程后还执行了如下操作，为子进程设置好了相应的环境和状态：
  - 为子进程分配异常处理栈。
  - 设置子进程的异常处理函数，确保页写入异常可以被正常处理。
  - 设置子进程的运行状态。

残留难点

`set_pgfault_handler`

- `set_pgfault_handler` 函数是父进程为子进程设置页错误处理函数的函数，其具体实现原理我还没有完全理解。

```
void
set_pgfault_handler(void (*fn)(u_int va))
{
    if (__pgfault_handler == 0) {
        // Your code here:
        // map one page of exception stack with top at UXSTACKTOP
        // register assembly handler and stack with operating system
        if (syscall_mem_alloc(0, UXSTACKTOP - BY2PG, PTE_V | PTE_R) < 0 ||
            syscall_set_pgfault_handler(0, __asm_pgfault_handler,
UXSTACKTOP) < 0) {
            writef("cannot set pgfault handler\n");
            return;
        }

        //      panic("set_pgfault_handler not implemented");
    }

    // Save handler pointer for assembly to call.
    __pgfault_handler = fn;
}
```