# 实验思考题

## Thinking 4.1

- 内核在保存现场的时候是如何避免破坏通用寄存器的?

  在系统调用异常处理函数 handle_sys 的开始，调用了 SAVE_ALL 宏，SAVE_ALL 将把所有的寄存器保存到指定位置，进而保证了在后续步骤不会破坏通用寄存器。在系统调用结束，handle_sys 调用 `j ret_from_exception` ，在这里，调用了 RESTORE_SOME 将所有通用寄存器的值恢复。

- 系统陷入内核调用后可以直接从当时的 `$a0-$a3` 参数寄存器中得到用户调用 msyscall 留下的信息吗?

  可以，因为 msyscall 函数调用时，寄存器 `$a0-$a3` 用于存放前四个参数。执行 syscall 并没有改变这四个寄存器。而且MIPS规范里 `$a0-$a3` 这四个寄存器就是干这事用的吧。不过跟上面说的一样，handle_sys 调用了 SAVE_ALL 宏，这几个寄存器也都保存起来了。

- 我们是怎么做到让 sys 开头的函数“认为”我们提供了和用户调用 `msyscall` 时同样的参数的?

  在 handle_sys 中先取出 `$a0` 到 `$a3` ，再从用户栈中取出其他的参数，最后将这些参数保存到内核栈中，模拟使得内核态的 sys 函数可以正常将这些参数传入到函数中。在跳转到 sys 开头的函数之前，handle_sys 将需要传递的参数都存到了栈中。在 sys 开头的函数中，会从栈中找传递的参数，就“认为”我们提供了和用户调用 msyscall 时同样的参数。

- 内核处理系统调用的过程对 Trapframe 做了哪些更改? 这种修改对应的用户态的变化是?

  handle_sys 函数在把上下文环境保存到 Trapframe 中后，取出 EPC 并将 EPC 加 4，在返回用户态后，从 syscall 的后一条指令开始执行。将返回值存入 `$v0` 寄存器，用户态可以正常获得系统调用的返回值。

## Thinking 4.2

思考下面的问题，并对这个问题谈谈你的理解：请回顾 lib/env.c 文件中 `mkenvid()` 函数的实现，该函数不会返回 0，请结合系统调用和 IPC 部分的实现与 `envid2env()` 函数的行为进行解释。

`mkenvid()` 内容如下:

```
1  #define LOG2NENV 10
2  u_int mkenvid(struct Env *e) {
3      u_int idx = e - envs;
4      u_int asid = asid_alloc();
5      return (asid << (1 + LOG2NENV)) | (1 << LOG2NENV) | idx;
6  }
```

不难看出，envid 由 asid 左移 11 位构成高 5 位，代表地址空间，第 10 位固定为 1，0-9 位为进程在相对数组 envs 基地址的偏移。高 5 位和低 10 位都有可能为 0，但因为第 10 位固定为 1，因此保证了此函数不会返回 0。

在 `envid2env()` 函数中:

```
int envid2env(u_int envid, struct Env **penv, int checkperm) {
    struct Env *e;
    /* Hint: If envid is zero, return curenv.*/
    if (envid == 0) {
        *penv = curenv;
        return 0;
    }
  ......
}
```

可以看出，如果传入的 envid 是 0，那么直接返回 curenv。因此上述 envid 为 0 的进程无法通过进程号被找到。

在 IPC 中如果要发送消息，需要通过 envid 找到对应进程，而通过 envid2env 找到的 是当前进程而不一定是想要发送到的 envid 为 0 的进程，从而造成消息错误发送（发送给自己）， 接受方也无法收到对应的消息。

此外，在 fork 函数中，父进程的返回值为子进程的 envid，子进程的返回值为 0，如果存在进程号为0的进程，系统很可能会把一个父进程误认为是子进程，从而执行错误的操作。

## Thinking 4.3

思考下面的问题，并对这两个问题谈谈你的理解：

- 子进程完全按照 `fork()` 之后父进程的代码执行，说明了什么？

  子进程和父进程的代码段共享同样的地址空间，子进程被创建后，和父进程共享代码段且具有相同的状态和数据，程序计数器就是父进程创建子进程之后的那一条指令。

- 但是子进程却没有执行 `fork()` 之前父进程的代码，又说明了什么？

  子进程恢复上下文时恢复的是 `fork()` 之后的上下文。

## Thinking 4.4

关于 fork 函数的两个返回值，下面说法正确的是：
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

选C，~~钝角，~~ 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。子进程在父进程调用 fork 时被创建，并赋予不同返回值，子进程返回值为 0，父进程返回值为子进程的进程号。

## Thinking 4.5

我们并不应该对所有的用户空间页都使用 duppage 进行映射。那么究竟哪些用户空间页应该映射，哪些不应该呢？请结合本章的后续描述、mm/pmap.c 中 mips_vm_init 函数进行的页面映射以及 include/mmu.h 里的内存布局图进行思考。

从内存布局图来看，我们需要保护的用户空间页为 UTEXT 到 USTACKTOP 的这一段，因为从 USTACKTOP 再往上到 UXSTACKTOP 这一段属于异常栈和无效内存的范围。

- 每个进程的异常处理栈属于自己的，不能映射给子进程。如果允许写时复制，则会导致：进程异常栈被写 -> 触发写时复制缺页异常 -> 需要保存现场 -> 写进程异常栈 -> 触发写时复制缺页异常 -> 死循环。
- 无效内存的范围显然不需要被保护。

与此同时，在 UTEXT 到 USTACKTOP 这一段中，也并不是所有页都要被保护。

- 首先，只读的页不需要被保护。
- 其次，用 PTE_LIBRARY 标识的页为共享页，同样不需要被保护。
- 其他的页无论是否已含有 PTE_COW，都要用 PTE_COW 标记以作为保护。

## Thinking 4.6

在遍历地址空间存取页表项时你需要使用到 vpd 和 vpt 这两个“指针的指针”，请参考 user/entry.S 和 include/mmu.h 中的相关实现，思考并回答这几个问题：

- vpt 和 vpd 的作用是什么？怎样使用它们？

```
.globl vpt
vpt:
  .word UVPT

.globl vpd
vpd:
  .word (UVPT + (UVPT >> 12) * 4)
```

  `*vpt = UVPT`，`*vpd = UVPT + ((UVPT >> 12) * 4)`。UVPT 定义在 include/mmu.h 中，由 Lab3 Thinking 3.2，UVPT 是页表的起始地址，根据页目录的自映射机制，`UVPT + ((UVPT >> 12) * 4)` 是页目录的起始地址。vpt 是指向二级页表指针的指针，vpd 是指向一级页表，也即页目录指针的指针。

  使用：vpt 和 vpd 都是指针的指针，可以向使用数组一样使用。对于虚拟地址 va，`(*vpd)[va >> 22]` 为二级页表的物理地址，`(*vpt)[va >> 12]` 为 va 对应的物理页面。

- 从实现的角度谈一下为什么进程能够通过这种方式来存取自身的页表？

  vpt 与 vpd 本质上是通过宏定义的方式来对用户态的一段内存地址进行映射，因此使用这种方式实际上就是在使用 MMU 内存布局图中的地址指针，所以可以通过这种方式来存取进程自身页表。

- 它们是如何体现自映射设计的？

  vpd 本身处于 vpt 段中，说明页目录本身处于其所映射的页表中的一个页面里面。所以这两个指针的设计中运用了自映射。

- 进程能够通过这种方式来修改自己的页表项吗？

不能。进程本身处于用户态，不可以修改自身页表项，这两个指针仅供页表和页目录访问所用，这部分地址（UVPT 和 UVPT + PDMAP 之间）是只读的，用户不能修改。

## Thinking 4.7

page_fault_handler 函数中，你可能注意到了一个向异常处理栈复 制 Trapframe 运行现场的过程，请思考并回答这几个问题：

- 这里实现了一个支持类似于“中断重入”的机制，而在什么时候会出现这种“中断重入”？

  在用户发生写时复制 COW 引发的缺页中断并进行处理时，可能会再次发生缺页中断，从而出现“中断重入”。

- 内核为什么需要将异常的现场 Trapframe 复制到用户空间？

  在微内核结构中，对缺页错误的处理由用户进程完成，用户进程在处理过程中需要读取 Trapframe 的内容；同时，在处理结束后同样是由用户进程恢复现场，会用到 Trapframe 中的数据。

## Thinking 4.8

到这里我们大概知道了这是一个由用户程序处理并由用户程序自身来恢复运行现场的过程，请思考并回答以下几个问题：

- 在用户态处理页写入异常，相比于在内核态处理有什么优势？

  陷入内核会增添操作系统内核的工作量；且让用户进程实现内核功能体现了微内核思想，全方位保证操作系统正常运行。将异常处理交给用户进程，可以让内核做更多其他的事情。

- 从通用寄存器的用途角度讨论，在可能被中断的用户态下进行现场的恢复，要如何做到不破坏现场中的通用寄存器？

  在发生异常时，在中断执行前使用 SAVE_ALL 保存所有寄存器的值，在从异常中返回时，使用 RESTORE_SOME 恢复寄存器的值，最后再恢复栈指针从未完成现场的恢复。

## Thinking 4.9

请思考并回答以下几个问题：

- 为什么需要将 set_pgfault_handler 的调用放置在 syscall_env_alloc 之前？

  在调用 syscall_env_alloc 的过程中也可能需要进行异常处理，在调用 fork 时，有可能当前进程已是之前进程的子进程，从而需要考虑是否会发生写时复制的缺页中断异常处理，如果这时还没有调用过 set_pgfault_handler 则无法处理异常。

- 如果放置在写时复制保护机制完成之后会有怎样的效果？

  进程给 __pgfault_handler 变量赋值的时候会触发缺页中断，但因中断处理未设置好导致无法进行正常处理。

- 子进程是否需要对在 entry.S 定义的字 __pgfault_handler 赋值？

  不需要。父进程在之前设置过 `__pgfault_handler` 的值，而子进程复制了父进程中 `__pgfault_handler` 变量值，因此无需再次设置。

# 实验难点

## 内核态与用户态

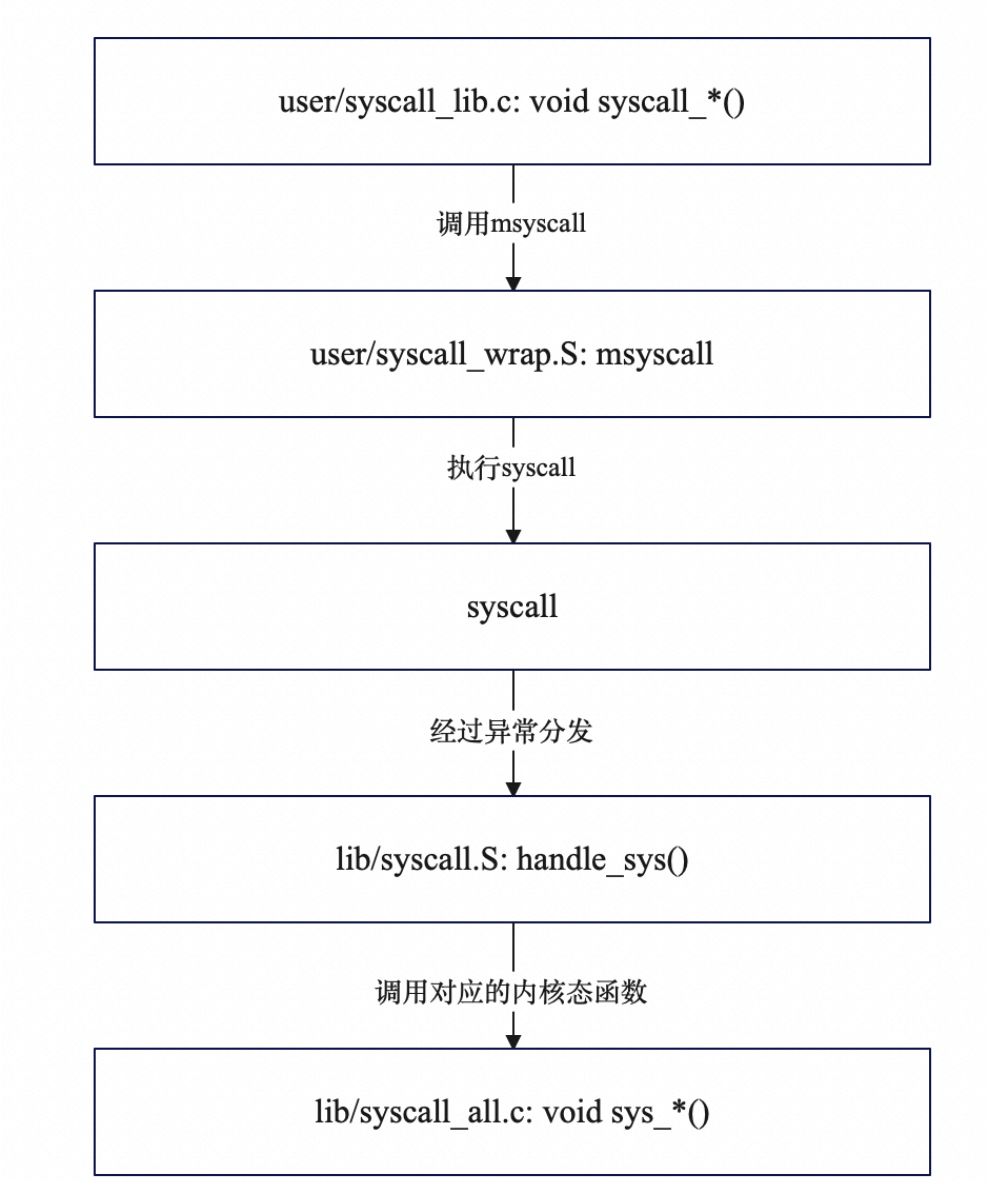
在我们的小操作系统中，对内核态与用户态最直观的区分就是：user 文件夹中的内容（如 fork.c，syscall_lib.c）为用户态的内容，而其他文件夹（如 lib）为内核态的内容。在内存布局图中，以 ULIM 为分界线，其上的内存空间属于内核态，其下的内存空间属于用户态。
我们应该理解每一步是在内核态运行还是用户态运行的。

- msyscall：用于让程序陷入内核，属于用户态函数。
- handle_sys：汇编函数，用于将用户态的系统调用的请求正确传递至内核态的系统调用函数，从而能正确的完成。属于内核态函数。
- sys_mem_alloc：用于分配内存的系统调用函数，属于内核态函数。
- sys_mem_map：用于实现两个进程间地址空间映射的系统调用函数，属于内核态函数。
- sys_mem_unmap：用于解除某个进程空间虚拟内存与物理内存映射的系统调用函数，属于内核态函数。
- sys_yield：用于实现用户进程对CPU的放弃的系统调用函数，属于内核态函数。
- sys_ipc_recv：用于实现进程间通信的接收的系统调用函数，属于内核态函数。
- sys_ipc_can_send：用于实现进程间通信的发送的系统调用函数，属于内核态函数。
- sys_env_alloc：用于创建当前进程的子进程的系统调用函数，属于内核态函数。
- duppage：用于对用户空间页中的可写入页进行保护，属于用户态函数。
- page_fault_handler：用于将当前现场保存至异常处理栈并设置 EPC，属于内核态函数。
- sys_set_pgfault_handler：用于设置中断处理函数的系统调用函数，属于内核态函数。
- pgfault：用于处理缺页中断，属于用户态函数。
- sys_set_env_status：用于设置子进程的状态的系统调用函数，属于内核态函数。
- fork：用于创建一个子进程并分别运行，属于用户态函数。

## 系统调用的过程

实现一个系统调用需要进行下面几步：

1. 用户态进程调用位于 user/syscall_lib.c 中的 syscall_xxx 函数。
2. syscall_xxx 将相关的参数传入 msyscall 中。
3. msyscall 跳转至 handle_sys 中，将用户栈与相关寄存器转移到内核栈与寄存器中，并跳转到位于 lib/syscall_all.c 的相应的 sys_xxx 函数。
4. 执行 sys_xxx 函数并返回，并恢复栈。
5. 返回用户态。

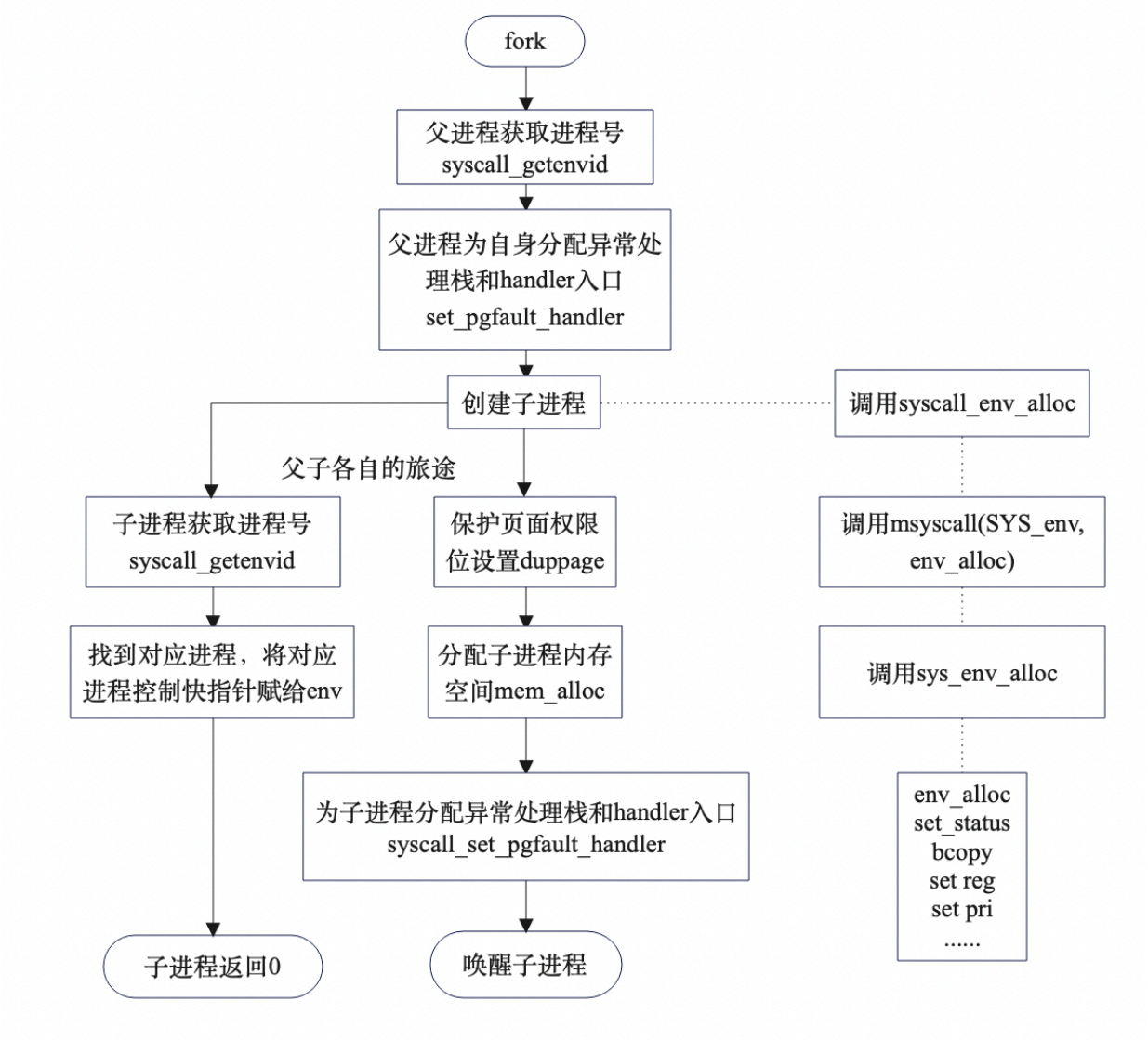



若要增加新的系统调用，需要更改的位置有：

- `lib/syscall.S` 中最后的 `.word` 字段。
- `include/unistd.h` 中的系统调用号。
- `lib/syscall_all.c` 中的 `sys_xxx`。
- `user/syscall_lib.c` 中的 `syscall_xxx`。
- 参数增加时需要修改 `msyscall` 的声明（`user/lib.h`）。

## fork

`fork` 函数的过程如图所示




# 体会与感想

感觉 lab4 的难度并不小，主要是一会用户态一会内核态，整晕了。一开始没弄清哪些函数在内核态哪些在用户态，导致填写函数时有点吃力，在 debug 的过程中逐渐整明白了一点。整个过程横跨用户态内核态，用了一堆文件，一步步找，感觉也挺麻的。而且软院人写汇编代码，真的痛苦。