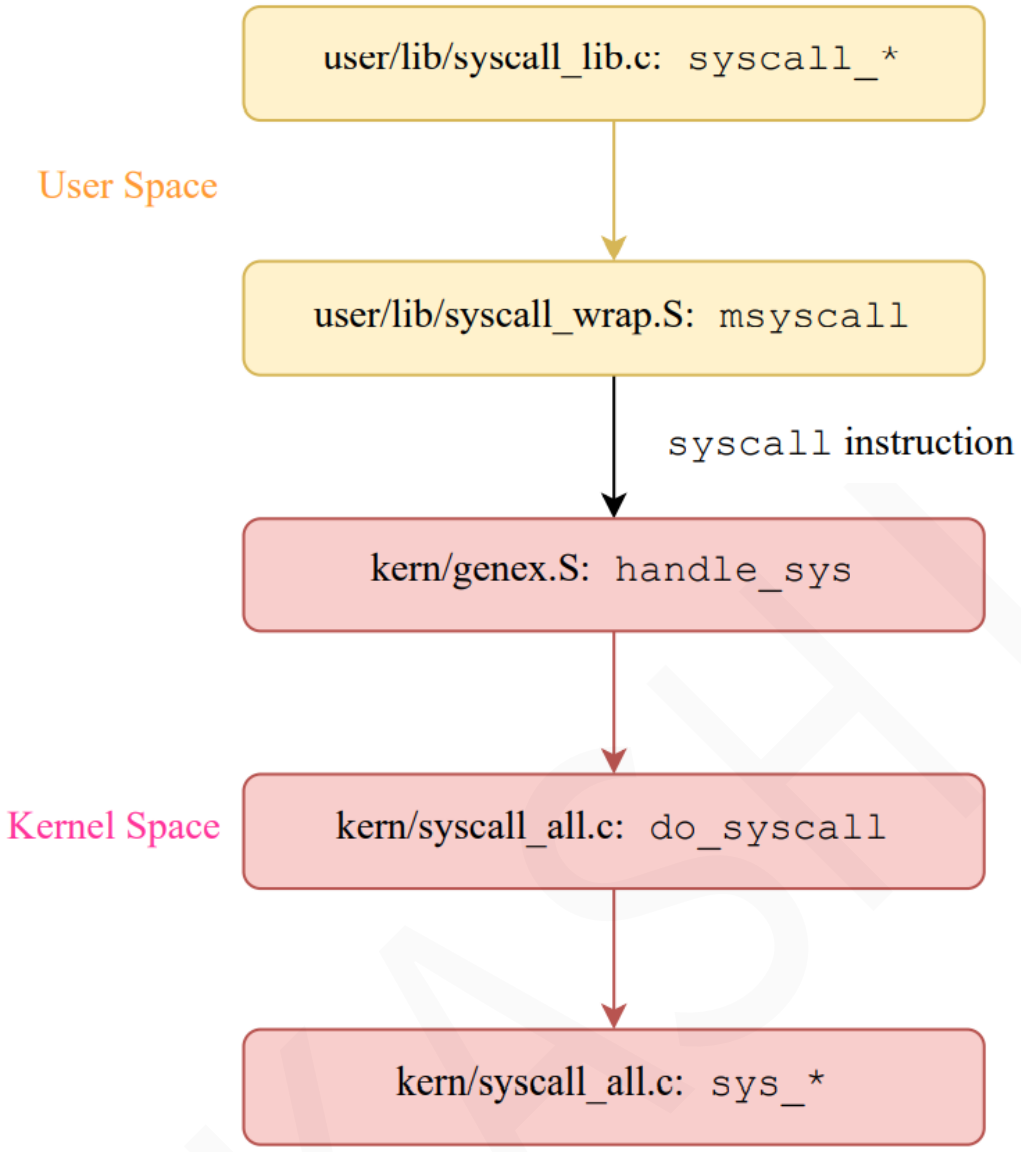
系统调用与fork

用户态程序崩溃

```
// user/lib/debugf.c

// 在用户态下替代 printk 进行格式化输出
void debugf(const char *fmt, ...);

// 在用户态下引发用户进程崩溃
void _user_panic(const char *file, int line, const char *fmt, ...);

// 在用户态下引发内核崩溃
void _user_halt(const char *file, int line, const char *fmt, ...)
```

```
// user/include/lib.h

// 对 _user_panic 和_user_halt 的封装，可以像 debugf 一样进行格式化输出
#define user_panic(...) _user_panic(__FILE__, __LINE__, __VA_ARGS__)
#define user_halt(...) _user_halt(__FILE__, __LINE__, __VA_ARGS__)

// 不为零则 panic，类似于内核态的 try，只不过这里不是 return，而是调用 user_panic 使进程退出
#undef panic_on
#define panic_on(expr) ……

// 类似于内核态的assert，为假则 panic
#define user_assert(x)
```

系统调用



syscall_*与内核中的sys_* 函数一一对应。

系统调用号

```
enum {
    SYS_putchar,            // 0：用于在控制台输出一个字符
    SYS_print_cons,         // 1：用于在控制台输出字符串
    SYS_getenvid,           // 2：获取当前环境的ID
    SYS_yield,              // 3：让出CPU，允许其他环境运行
    SYS_env_destroy,        // 4：销毁指定环境
    SYS_set_tlb_mod_entry,  // 5：设置TLB修改项
    SYS_mem_alloc,          // 6：分配内存
    SYS_mem_map,            // 7：映射内存
    SYS_mem_unmap,          // 8：取消内存映射
    SYS_exofork,            // 9：创建一个新的环境，复制当前环境
    SYS_set_env_status,     // 10：设置环境状态
    SYS_set_trapframe,      // 11：设置陷阱帧
    SYS_panic,              // 12：使环境进入panic状态
    SYS_ipc_try_send,       // 13：尝试发送IPC消息
    SYS_ipc_recv,           // 14：接收IPC消息
    SYS_cgetc,              // 15：从控制台获取一个字符
```

```
    SYS_write_dev,          // 16：向设备写入数据
    SYS_read_dev,           // 17：从设备读取数据
    MAX_SYSNO,              // 18：系统调用的最大编号（非系统调用）
};
```

系统调用（sys中断）分发

详见Lab3。

syscall

注意调用时的参数类型和sys_*处理时并不相同。

```
// user/lib/syscall_lib.c
void syscall_putchar(int ch);
int syscall_print_cons(const void *str, u_int num);
u_int syscall_getenvid(void);
void syscall_yield(void);
int syscall_env_destroy(u_int envid);
int syscall_set_tlb_mod_entry(u_int envid, void (*func)(struct Trapframe *));
int syscall_mem_alloc(u_int envid, void *va, u_int perm);
int syscall_mem_map(u_int srcid, void *srcva, u_int dstid, void *dstva, u_int
perm);
int syscall_mem_unmap(u_int envid, void *va);
int syscall_set_env_status(u_int envid, u_int status);
int syscall_set_trapframe(u_int envid, struct Trapframe *tf);
void syscall_panic(const char *msg);
int syscall_ipc_try_send(u_int envid, u_int value, const void *srcva, u_int
perm);
int syscall_ipc_recv(void *dstva);
int syscall_cgetc();
// int syscall_write_dev(void *va, u_int dev, u_int len);
// int syscall_read_dev(void *va, u_int dev, u_int len);
```

msyscall

这个函数有很多参数，最多有 6 个，第一个是系统调用号。并且这个函数需要把参数从用户态传递到内核态，实现的方式是，在用户态正常压栈，在内核态中时通过内核栈中的寄存器快照访问参数，包括 $a0 - $a3 的值和其中 sp 指向的用户栈上保存的参数。

```
# user/lib/syscall_wrap.S
LEAF(msyscall)
    syscall
    jr ra
END(msyscall)
```

do_syscall

```
// kern/syscall_all.c

/* 从用户栈中取出 msyscall 的参数，分发系统调用
从寄存器快照中取 a0 系统调用号，判断是否合法，不合法则将返回值写入寄存器快照中 v0
将 EPC += 4 跳过 syscall 指令，使能正确返回
从系统调用函数表中取出相应函数
从寄存器快照和用户栈中读取参数
调用刚刚取出的函数
*/
void do_syscall(struct Trapframe *tf);
```

辅助函数

```
// kern/syscall_all.c
static inline int is_illegal_va(u_long va) {
    return va < UTEMP || va >= UTOP;
}

static inline int is_illegal_va_range(u_long va, u_int len) {
    if (len == 0) {
        return 0;
    }
    return va + len < va || va < UTEMP || va + len > UTOP;
}
```

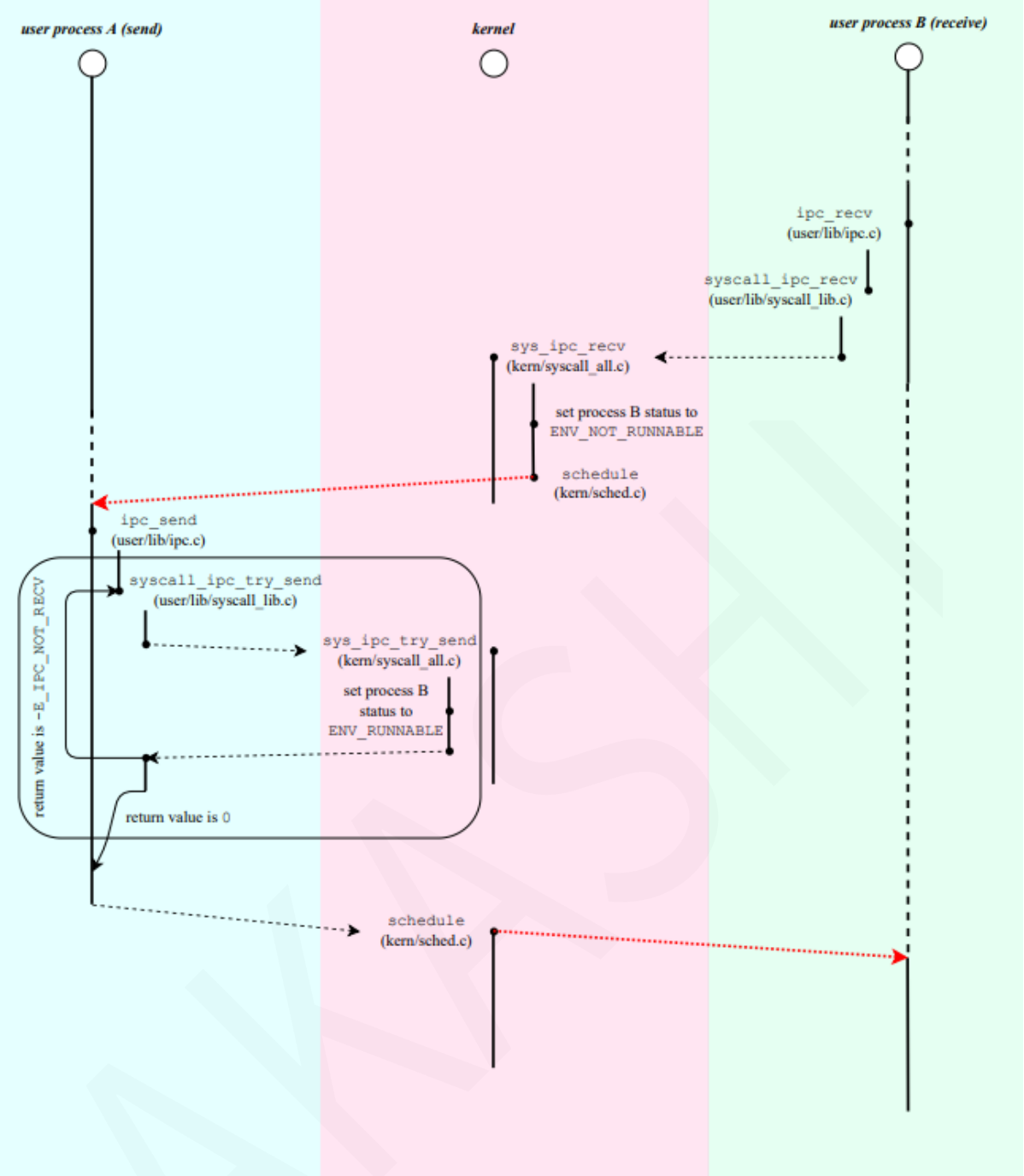
```
// kern/env.c

/* 通过 envid 获取相应的进程 PCB，同时可选进行权限检查
如果 envid 为0，直接获取 curenv 并返回
否则使用 &envs[ENVX(envid)] 获取指向相应 PCB 的指针
检查当前当前 PCB 是否有效，并检查当前 PCB 指代的进程是否与 envid 相对应，失败则返回 -E_BAD_ENV
如果 checkprem 置位，则检查指向的进程是否是当前进程或其直接子进程，不是则返回 -E_BAD_ENV
为 *e 复制并返回 0
*/
int envid2env(u_int envid, struct Env **penv, int checkperm);
```

一些系统调用的处理

```
// kern/syscall_all.c

/* 显示为用户可操作的用户地址空间分配物理内存
检查用户虚地址合法性，不合法返回 -E_INVAL
尝试使用 envid 获取对应结构体，并检查权限，失败返回 -E_BAD_ENV
尝试使用 page_alloc 分配新的物理页，失败返回 -E_NO_MEM
尝试使用 page_insert 建立映射关系，失败返回 -E_NO_MEM
*/
int sys_mem_alloc(u_int envid, u_int va, u_int perm);

/* 将源进程地址空间中的相应内存映射到目标进程的相应地址空间的相应虚拟内存中去\
```

```
检查源和目标两个用户虚地址合法性，不合法返回 -E_INVAL
尝试使用 envid 获取两个对应结构体，并检查权限，失败返回 -E_BAD_ENV
尝试使用 page_lookup 在源进程页表中寻找对应物理页，失败返回 -E_INVAL
尝试使用 page_insert 建立映射关系，失败返回 -E_NO_MEM
*/
int sys_mem_map(u_int srcid, u_int srcva, u_int dstid, u_int dstva, u_int perm);

/* 是解除某个进程地址空间虚拟内存和物理内存之间的映射关系
检查用户虚地址合法性，不合法返回 -E_INVAL
尝试使用 envid 获取对应结构体，并检查权限，失败返回 -E_BAD_ENV
使用 page_remove 解除映射
*/
int sys_mem_unmap(u_int envid, u_int va);

/* 用户进程对 CPU 的放弃
调用 schedule 强行引发切换
*/
void sys_yield(void)
```

进程间通讯



一个进程使用系统调用将数据和相关信息存放在进程控制块中，另一个进程同样使用系统调用的方式读取。

定义

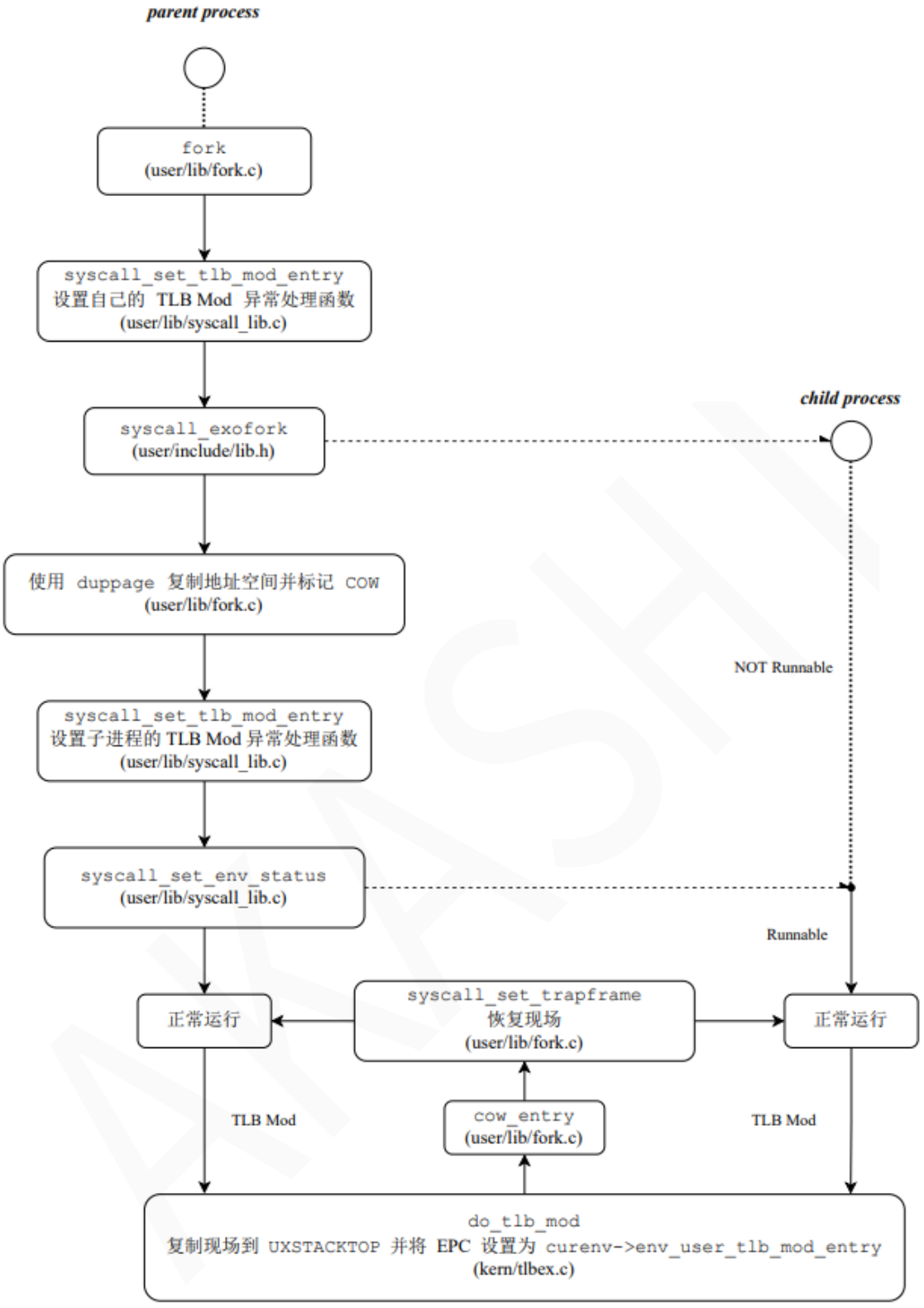
```
// include/env.h

struct Env {
    // Lab 4 IPC
    u_int env_ipc_value;   // 进程传递的具体数值
    u_int env_ipc_from;    // 发送方的进程 ID
    u_int env_ipc_recving; // 1：等待接受数据中；0：不可接受数据
    u_int env_ipc_dstva;   // 接收到的页面需要与自身的哪个虚拟页面完成映射
    u_int env_ipc_perm;    // 传递的页面的权限位设置
```

```
    // Lab 4 fault handling
    u_int env_user_tlb_mod_entry; // user tlb mod handler
};
```

相关系统调用处理

```
// kern/syscall_all.c

/* 阻塞当前进程，准备接受数据
检查目标虚地址是否是 0 或用户合法地址，不是返回 -E_INVAL
设置当前进程控制块，表明准备接收数据，并设置接收到的页面的映射目标虚地址
设置当前进程状态为未就绪，从调度队列中移除
设置内核栈底的寄存器快照，将返回值设置为 0，强制进程调度
*/
int sys_ipc_recv(u_int dstva);

/* 尝试向目标进程发送数据
检查源虚地址是否是 0 或用户合法地址，不是返回 -E_INVAL
尝试获取目标进程进程控制块，不检查亲缘关系，失败返回 -E_IPC_NOT_RECV
将要发送的数据写入目标进程控制块，标记来源，设置权限位，结束目标进程等待接收的状态
设置目标进程就绪，插入调度队尾
如果源虚地址不为 0，则将源虚地址在当前页表下对应的物理页映射到目标进程页表的目标虚地址上，失败返回 -E_NO_MEM
*/
int sys_ipc_try_send(u_int envid, u_int value, u_int srcva, u_int perm);
```

fork和写时复制



相关宏定义

```
// user/include/lib.h

#define vpt ((volatile Pte *)UVPT) // 映射到用户空间的页表的基地址
#define vpd ((volatile Pde *)(UVPT + (PDX(UVPT) << PGSHIFT))) // 映射到用户空间的页目录的基地址
#define envs ((volatile struct Env *)UENVS) // 映射到用户空间的进程控制块数组
#define pages ((volatile struct Page *)UPAGES) // 映射到用户空间的页控制块数组
```

涉及到的系统调用的处理

```
// kern/syscall_all.c

/* 申请一个新的进程控制块，设置一些信息；复制父进程内核栈的寄存器快照，修改返回值
尝试申请新的进程控制块，失败返回 -E_NO_MEM 或 -E_NO_FREE_ENV
复制内核栈底的寄存器快照给子进程进程控制块中的 env_tf
在子进程的寄存器快照中设置v0为0，即设置返回值
标记子进程未就绪，继承父进程优先级
*/
int sys_exofork(void);

/* 为目标进程注册页写入异常处理函数
尝试使用 envid 获取对应 Env 结构体，并检查权限，失败返回 -E_BAD_ENV
将该进程控制块的 env_user_tlb_mod_entry 设置为传入的 func
*/
int sys_set_tlb_mod_entry(u_int envid, u_int func);

/* 修改目标进程的进程状态
检查传入的状态是否是就绪或阻塞，不是返回 -E_INVAL
尝试使用 envid 获取对应 Env 结构体，并检查权限，失败返回 -E_BAD_ENV
维护调度链表
修改相应 PCB 的进程状态
*/
int sys_set_env_status(u_int envid, u_int status);

/* 将用户空间的寄存器快照赋予相应进程 PCB
判断快照是否位于 UTEMP 到 UTOP 之间，不处于则返回 -E_INVAL
尝试使用 envid 获取对应 Env 结构体，并检查权限，失败返回 -E_BAD_ENV
如果修改当前进程，则将传入的快照复制到内核栈底，以供展开，返回 tf->regs[2]，之际上之后这个值还
会被写入 tf->regs[2]，相当于没动
如果修改其他进程，则将传入的快照给相应 PCB 并返回
*/
int sys_set_trapframe(u_int envid, struct Trapframe *tf);
```

```
// user/lib/fork.c

/* 复制页面的映射，并修改相应权限位
获取 vpt[vpn] 的权限位
如果可写，非共享，非写时复制，则取消写，置写时复制，使用 syscall_mem_map 先后将该地址映射到子
进程，重新映射到当前进程
如果不是则直接映射到子进程
*/
static void duppage(u_int envid, u_int vpn);

/* 在用户态处理页写入异常
获取问题访问的虚拟地址，并通过它获得权限，如果不是写时复制页，引发进程崩溃
使用 syscall_mem_alloc 为分配 UCOW 分配物理页
将原页内容拷贝到 UCOW 区域
使用 syscall_mem_map 将自己的 UCOW 区域映射回原来的虚拟地址
使用 syscall_mem_unmap 解除 UCOW 的映射
通过调用 syscall_set_trapframe，展开异常处理栈保存的寄存器快照，返回处理前的状态
*/
static void __attribute__((noreturn)) cow_entry(struct Trapframe *tf);
```

```
/* 用户态进程创建新进程
首先检查当前进程的写时复制处理函数是否是 cow_entry，不是则设置
调用 syscall_exofork 申请新 PCB 复制寄存器快照并设置父进程和优先级
如果是子进程，设置 env 为当前进程进程控制块并返回
父进程将有效的页 duppage 给子进程
父进程为子进程设置写时复制处理函数 cow_entry，设置子进程就绪
*/
int fork(void);
```

```
// kern/tlbex.c

/* 处理“页写入异常”，修改内核栈中保存的寄存器快照，以便异常返回时将写时复制交给用户注册的处理程序
使用异常处理栈处理，并在用户异常处理栈中保存初始传入的寄存器快照
如果保存的寄存器快照中用户态的 sp 指针不在异常处理栈中，则将其置于异常处理栈顶部
移动寄存器快照中的 sp 指针，在异常处理栈内开辟空间，将**初始**传入的寄存器快照复制一份到异常处理栈中
如果当前进程没有设置写时复制函数则引发内核崩溃
将指向初始传入的未经修改的寄存器快照在用户异常处理栈的拷贝的指针作为参数，保存到内核栈中寄存器快照的 a0，位置，并移动相应 sp
设置 cp0_epc 为当前进程控制块中保存的 env_user_tlb_mod_entry
（当从异常返回时，相当于以指向这份快照的指针作为参数调用 env_user_tlb_mod_entry）
*/
void do_tlb_mod(struct Trapframe *tf);
```

实验报告部分

4.1

- 保存现场的工作主要由汇编宏 SAVE_ALL 完成。主要容易被破坏的通用寄存器是 $sp，在该宏定义中，通过先将 $sp 的值转移到 $k0 中，再移动 sp 进行保存，来避免破坏其值。
- 在特殊的情况下，如果 msyscall 和 handle_sys 函数不传递参数，可能是可以的，因为在用户调用 msyscall 时传递的参数可能仍保留在 $a1 - $a3 中，$a0 保存的是用户态的 sp，这是由 handle_sys 传递给 do_* 的参数。正确的做法是通过该参数，即异常分发后 sp 的值，指向一个寄存器快照结构体，应该从其中获取参数值，包括 $a0 - $a3 的值和其中 sp 指向的用户栈上保存的参数。
- 我们在 do_syscall 函数中把参数从用户栈中取了出来，传递给相应的处理函数，编译器会将他们放在相应的寄存器和内核栈的相应位置上，让sys函数认为我们提供了用户调用 msyscall 时同样的参数。
- 我们让 CP0_EPC 的值指向了原本指向的指令的下一条指令。这样做的原因是让返回后 PC 指向 syscall 的下一条指令，使得异常处理在完成后能够正确返回。

4.2

相应的进程控制块可能已经被释放，或者已经被分配给了其它新的进程。如果不进行这个判断，就可能把新的环境和一个已经被释放的环境混淆，进而导致错误的行为。

4.3

在 mkenvid() 有一个静态变量 i，只会初始化一次，之后不断自增，在生成 envid 时，i 会被放在其高位。在这种状况下，mkenvid() 不会返回 0 ，并且除了极端情况，即使是对于相同的进程控制块，当其对应不同的进程时，也不会给出相同的 id。由此便可以在系统调用时进行验证，某个系统调用操作的目标进程，还是否用户希望操作的进程，如果目标进程对应的进程控制块已经被 free 或者拥有了新的 envid，即标志着它不再能代表用户希望操作的进程了，此时 envid2env 会返回 -E_BAD_ENV。此外，这一特性在之后系统调用中也会用到。

4.4

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。

4.5

应该映射 UTEMP 到 USTACKTOP 之间有效的页。

4.6

- vpt 和 vpd 的作用是使用户能够在用户空间下访问页表。使用方式是在用户程序中通过这两个指针访问表项。
- 因为用户能够在用户态下使用这两个地址，通过MMU的转换，正确的找到页表所在的物理地址，进而进行访问。原来页表所在的物理地址仅仅映射到了内核虚拟地址，故无法在用户态访问，通过把页表映射到用户空间，使得用户可以使用用户虚地址访问页表所在的物理地址。
- 在将所有的页表页映射到 UVPT 对应的空间上时，我们仅仅操作了页目录中的一个特殊项，即自映射的页目录项。通过建立这一项的映射，并赋予它自映射页表页目录的含义，完成了将所有的页表页映射到 UVPT 的目标。在使用时，用户可以通过 UVPT 找到对应的自映射页目录基地址，进而访问全部的页表。vpt[PDX(UVPT)] 指向的页面，既是页目录，也是管理 UVPT 中所有页表页的页表。
- 我们使用以下代码完成页目录自映射 `e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_V;` 不难看出在进行自映射时页目录只给予了有效的权限位，并没有给予可写权限位，故不能修改自己的页表项，对页表项的写入操作会被MMU拒绝。

4.7

- 在异常处理的时候如果再次出现了异常，会发生异常重入。
- 因为当 cow_entry 处理完写时复制的内容后，需要利用异常处理栈中保存的寄存器快照恢复之前的现场。内核栈中除了栈底的内容会在返回用户态，即进入 cow_entry 函数时被丢弃，所以不能放在内核栈中。而内核栈底的寄存器快照用于在返回用户态的时候进入 cow_entry。

4.8

相比于在内核态处理页写入异常，在用户态处理安全风险也较小，因为用户态具有的权限更低；系统的稳定性更高，当在用户态处理时，如果发生错误，只会让当前进程产生问题，如果在内核态进行处理，产生错误可能会导致内核崩溃。

4.9

- 在 syscall_exofork 之前调用的 syscall_set_tlb_mod_entry 的目的是设置父进程自己的写时复制处理函数，如果放在 syscall_exofork 之后，则会在子进程中再次被执行。
- 如果放置在写时复制完成之后，则可能会遇到触发写时复制保护时，还没有注册写时复制处理函数的问题。