实　　验　　报　　告

学　　号	21030031009	姓　　名	惠欣宇	专业班级	计算机 4 班
课程名称	**操作系统**			学期	**2023 年秋季学期**
任课教师	窦金凤	完成日期	2023.12.15	上机课时间	周五 3-6
实 验 名 称	**系统调用与 fork**				

Exercise 部分：

Exercise 4.1

> Exercise 4.1 填写 user/syscall_wrap.S 中的 msyscall 函数，使得用户部分的系统调用机制可以正常工作。

LEAF(msyscall): 定义一个叫做 msyscall 的叶子函数，表示这是一个函数的起始点。在这里，这个函数实际上只包含一个系统调用。
syscall: 执行系统调用指令。syscall 指令触发操作系统中断，将控制权传递给内核，执行相应的系统调用服务例程。
jr ra: 使用 jr 指令跳转到寄存器 ra 保存的地址，即返回到调用 msyscall 的地方。
nop: No Operation，空指令。在这里没有实际的操作，只是为了保证 jr 指令的正确性。
END(msyscall): 定义 msyscall 函数结束。

代码如图 1 所示。

```
#include <asm/regdef.h>
#include <asm/cp0regdef.h>
#include <asm/asm.h>

LEAF(msyscall)
    // TODO: you JUST need to execute a `syscall` instruction and return from msyscall

    syscall
    jr ra
    nop

END(msyscall)
```

图 1　msyscall 函数填写

Exercise 4.2

> Exercise 4.2 按照 lib/syscall.S 中的提示，完成 handle_sys 函数，使得内核部分的系统调用机制可以正常工作。

用于处理系统调用的异常处理程序。下面对每个部分的作用进行解释：
SAVE_ALL：保存当前进程的寄存器状态到 Trapframe 结构中，宏 SAVE_ALL 在这里负责保存寄存器状态。
CLI：关中断，通过清除 CP0_STATUS 的 IE 位来关闭中断。
Fetch EPC and Update： 从 Trapframe 结构中取出 EPC（Exception Program Counter）字段，该字段保存了进程在发生异常时的下一条指令地址。在系统调用返回时，需要将 EPC 更新为下一条指令的地址，所以将 EPC 加上 4（因为系统调用指令的长度为 4 字节）并保存回 Trapframe。
Copy Syscall Number into $a0：将 Trapframe 结构中的寄存器 reg4 的值（保存了系统调用号）复制到

寄存器 $a0。
Calculate Function Entry in sys_call_table：计算 sys_call_table 中特定系统调用号的函数入口地址。使用相对系统调用号减去 __SYSCALL_BASE 来获取在 sys_call_table 中的偏移，然后计算函数入口的地址。
Copy Arguments from User's Stack to Kernel Stack：从用户态栈中取出 msyscall 传递给系统调用的参数，这里是第 5 和第 6 个参数，分别保存到寄存器 t3 和 t4 中。接着在当前内核栈上为这六个参数分配空间，并将参数复制到相应的位置。
Invoke sys_ Function：* 通过 jalr 指令调用 sys_* 函数，其中 t2 寄存器保存了 sys_call_table 中相应系统调用号的函数入口地址。
Resume Current Kernel Stack：系统调用执行完成后，恢复当前内核栈的状态。
Store Return Value into Trapframe：将 sys_* 函数的返回值（保存在 v0 中）存储到 Trapframe 结构的 reg2 寄存器中。
Return from Exception：使用 j 指令跳转到 ret_from_exception 标签，实现异常处理的返回。
nop 指令： 空指令，用于占位，没有具体操作。

代码具体实现如图 2 所示。

```
NESTED(handle_sys,TF_SIZE, sp)
    SAVE_ALL                            // Macro used to save trapframe
    CLI                                 // Clean Interrupt Mask
    nop
    .set at                             // Resume use of $at

    // TODO: Fetch EPC from Trapframe, calculate a proper value and store it back to trapframe.
    lw t0, TF_EPC(sp)
    addiu t0, t0, 4
    sw t0, TF_EPC(sp)

    // TODO: Copy the syscall number into $a0.
    lw a0, TF_REG4(sp)

    addiu   a0, a0, -__SYSCALL_BASE     // a0 <- relative syscall number
    sll     t0, a0, 2                   // t0 <- relative syscall number times 4
    la      t1, sys_call_table          // t1 <- syscall table base
    addu    t1, t1, t0                  // t1 <- table entry of specific syscall
    lw      t2, 0(t1)                   // t2 <- function entry of specific syscall

    lw      t0, TF_REG29(sp)            // t0 <- user's stack pointer
    lw      t3, 16(t0)                  // t3 <- the 5th argument of msyscall
    lw      t4, 20(t0)                  // t4 <- the 6th argument of msyscall

    // TODO: Allocate a space of six arguments on current kernel stack and copy the six arguments to proper location
    lw a0, TF_REG4(sp)
    lw a1, TF_REG5(sp)
    lw a2, TF_REG6(sp)
    lw a3, TF_REG7(sp)
    addiu sp, sp, -24
    sw t3, 16(sp)
    sw t4, 20(sp)

    jalr    t2                          // Invoke sys_* function
    nop

    // TODO: Resume current kernel stack
    addiu sp, sp, 24
    sw      v0, TF_REG2(sp)             // Store return value of function sys_* (in $v0) into trapframe

    j       ret_from_exception          // Return from exeception
    nop
END(handle_sys)
```

图 2　handle_sys 函数实现

Exercise 4.3

Exercise 4.3 实现 lib/syscall_all.c 中的 int sys_mem_alloc(int sysno,u_int envid, u_int va, u_int perm) 函数

为指定的进程分配一个物理页面，并将该物理页面映射到进程的虚拟地址空间中的指定位置。这个系统调用的参数包括目标进程的环境 ID (envid)、欲映射的虚拟地址 (va)，以及映射权限 (perm)。函数首先进行参数的合法性检查，然后通过 envid2env 函数获取目标进程的结构体。接着，调用 page_alloc 分配一个物理页面，并使用 page_insert 将该物理页面映射到目标进程的虚拟地址空间中的指定位置，使用给定的映射权限。如果整个过程都执行成功，函数返回 0 表示成功执行系统调用。

代码如图 3 所示。

```
305 int sys_mem_alloc(int sysno, u_int envid, u_int va, u_int perm)
306 {
307     // Your code here.
308     struct Env *env;
309     struct Page *ppage;
310     int ret;
311     ret = 0;
312     perm = perm & (BY2PG - 1);
313     if(va >= UTOP){
314         return -E_INVAL;
315     }
316     if((perm & PTE_COW) || (!(perm & PTE_V))){
317         return -E_INVAL;
318     }
319     ret = envid2env(envid, &env, 1);
320     if(ret < 0){
321         return ret;
322     }
323     ret = page_alloc(&ppage);
324     if(ret < 0){
325         return ret;
326     }
327     ret = page_insert(env->env_pgdir,ppage, va, perm);
328     if(ret < 0){
329         return ret;
330     }
331     return 0;
332 }
```

图 3　sys_mem_alloc 函数实现

Exercise 4.4

Exercise 4.4 实现 lib/syscall_all.c 中的 int sys_mem_map(int sysno,u_int srcid, u_int srcva, u_int dstid, u_dstva, u_int perm) 函数

sys_mem_map 的功能是将一个进程的某个虚拟地址空间中的页面映射到另一个进程的虚拟地址空间中的指定位置。该系统调用的参数包括源进程的环境 ID (srcid)、源进程的虚拟地址 (srcva)、目标进程的环境 ID (dstid)、目标进程的虚拟地址 (dstva)，以及映射权限 (perm)。

函数首先进行参数的合法性检查，包括检查虚拟地址是否超出用户地址空间、检查是否有有效的映射权限等。然后，通过 envid2env 函数获取源进程和目标进程的结构体。接着，调用 page_lookup

函数查找源进程中指定虚拟地址的页面，并获取该页面的物理页框和页表项。

最后，调用 page_insert 将源进程的页面映射到目标进程的虚拟地址空间中的指定位置，使用给定的映射权限。如果整个过程都执行成功，函数返回 0 表示成功执行系统调用。

代码如图 4 所示。

```
347 int sys_mem_map(int sysno, u_int srcid, u_int srcva, u_int dstid, u_int dstva,
348                  u_int perm)
349 {
350     int ret;
351     u_int round_srcva, round_dstva;
352     struct Env *srcenv;
353     struct Env *dstenv;
354     struct Page *ppage;
355     Pte *ppte;
356 
357     ppage = NULL;
358     ret = 0;
359     round_srcva = ROUNDDOWN(srcva, BY2PG);
360     round_dstva = ROUNDDOWN(dstva, BY2PG);
361     //your code here
362     if(round_srcva >= UTOP || round_dstva >= UTOP){
363         return -E_INVAL;
364     }
365     if(!(perm & PTE_V)){
366         return -E_INVAL;
367     }
368     ret = envid2env(srcid, &srcenv,0);
369     if(ret < 0){
370         return ret;
371     }
372     ret = envid2env(dstid, &dstenv,0);
373     if(ret < 0){
374         return ret;
375     }
376     ppage = page_lookup(srcenv->env_pgdir, round_srcva, &ppte);
377     if(ppage == NULL){
378         return -E_INVAL;
379     }
380     if(((*ppte & PTE_R) == 0) && ((perm & PTE_R) != 0 )){
381         return -E_INVAL;
382     }
383     ret = page_insert(dstenv->env_pgdir, ppage, round_dstva, perm);
384     if(ret < 0){
385         return ret;
386     }
387     return ret;
388 }
```

图 4　sys_mem_map 函数实现

Exercise 4.5

Exercise 4.5 实现 lib/syscall_all.c 中的 int sys_mem_unmap(int sysno,u_int envid, u_int va) 函数

sys_mem_unmap 的功能是取消一个进程虚拟地址空间中的某个虚拟地址的页面映射。该系统调用的参数包括进程的环境 ID (envid) 和待取消映射的虚拟地址 (va)。

函数首先进行参数的合法性检查，检查虚拟地址是否超出用户地址空间。然后，通过 envid2env 函数获取进程的结构体。接着，调用 page_remove 函数将指定虚拟地址的页面从进程的页表中移除。

如果整个过程执行成功，函数返回 0 表示成功执行系统调用。

代码如图 5 所示。

```
400 int sys_mem_unmap(int sysno, u_int envid, u_int va)
401 {
402     // Your code here.
403     int ret;
404     struct Env *env;
405     if(va >= UTOP){
406         return -E_INVAL;
407     }
408     ret = envid2env(envid, &env, 0);
409     if(ret < 0){
410         return ret;
411     }
412     page_remove(env->env_pgdir,va);
413     return 0;
414 }
```

图 5　sys_mem_unmap 函数实现

Exercise 4.6

Exercise 4.6 实现 lib/syscall_all.c 中的 void sys_yield(void) 函数

sys_yield 将当前进程的 Trapframe 复制到时钟中断栈的顶部，并调用 sched_yield 函数进行进程切换。

函数使用 bcopy 将当前进程的 Trapframe 复制到时钟中断栈的顶部。这是因为在发生时钟中断时，内核会切换到中断栈，并在中断栈的顶部保存当前进程的 Trapframe。通过将当前进程的 Trapframe 复制到时钟中断栈的顶部，就可以在切换到下一个进程时使用这个 Trapframe，从而实现进程切换。
接着，函数调用 sched_yield，该函数负责选择下一个要运行的进程，并进行上下文切换。这样，当前进程的执行就被暂停，系统会选择另一个可运行的进程来执行。

代码如图 6 所示。

```
224 void sys_yield(void)
225 {
226     bcopy((void *)KERNEL_SP - sizeof(struct Trapframe), (void *)TIMESTACK - sizeof(struct Trapframe), sizeof(struct Trapframe));
227     sched_yield();
228 }
```

图 6　sys_yield 函数实现

Exercise 4.7

Exercise 4.7 实现 lib/syscall_all.c 中的 void sys_ipc_recv(int sysno,u_int dstva) 函数和 int sys_ipc_can_send(int sysno,u_int envid, u_int value, u_int srcva, u_int perm) 函数。

sys_ipc_recv 用于设置当前进程为接收 IPC（进程间通信）的状态。具体来说，它做了以下几个主要的事情：

检查目标虚拟地址 dstva 是否在有效用户地址范围内。

将当前进程的 env_ipc_recving 置为 1，表示当前进程正在等待接收 IPC。

设置当前进程的 env_ipc_dstva 为参数 dstva，表示当前进程期望接收 IPC 数据的虚拟地址。

将当前进程的状态 env_status 设置为 ENV_NOT_RUNNABLE，表示当前进程处于不可运行状态。

调用 sys_yield 进行进程切换，让其他可运行的进程有机会执行。

sys_ipc_recv 函数如图 7 所示。

```
533 void sys_ipc_recv(int sysno, u_int dstva)
534 {
535     if(dstva >= UTOP){
536         return;
537     }
538     curenv->env_ipc_recving = 1;
539     curenv->env_ipc_dstva = dstva;
540     curenv->env_status = ENV_NOT_RUNNABLE;
541     sys_yield();
542 }
```

图 7　sys_ipc_recv 函数实现

sys_ipc_can_send 用于判断当前进程是否可以向目标进程发送 IPC。

检查源虚拟地址 srcva 是否在有效用户地址范围内（UTOP 以下）。

调用 envid2env 函数获取目标进程的 struct Env 结构体。

检查目标进程是否正在等待接收 IPC（env_ipc_recving 是否为 0）。如果目标进程不处于等待接收状态，返回错误码 -E_IPC_NOT_RECV。

将目标进程的相关 IPC 字段设置为发送方的信息：env_ipc_recving 置为 0，env_ipc_from 设置为发送方的进程 ID，env_ipc_value 设置为参数 value，env_ipc_perm 设置为参数 perm。

将目标进程的状态 env_status 设置为 ENV_RUNNABLE，表示目标进程已经准备好可以运行。

如果参数 srcva 不为 0，则将当前进程中位于虚拟地址 srcva 处的页面插入到目标进程的虚拟地址 env_ipc_dstva 处，并设置权限为参数 perm。

sys_ipc_can_send 函数如图 8 所示。

```
562 int sys_ipc_can_send(int sysno, u_int envid, u_int value, u_int srcva,
563                     u_int perm)
564 {
565
566     int r;
567     struct Env *e;
568     struct Page *p;
569     perm = perm & (BY2PG - 1);
570     if(srcva >= UTOP){
571         return -E_INVAL;
572     }
573     r = envid2env(envid, &e, 0);
574     if(r < 0) return r;
575     if(e->env_ipc_recving == 0){
576         return -E_IPC_NOT_RECV;
577     }
578     e->env_ipc_recving = 0;
579     e->env_ipc_from = curenv->env_id;
580     e->env_ipc_value = value;
581     e->env_ipc_perm = perm;
582     e->env_status = ENV_RUNNABLE;
583
584     if(srcva != 0){
585         p = page_lookup(curenv->env_pgdir, srcva, NULL);
586         if(p == NULL){
587             return -E_INVAL;
588         }
589         page_insert(e->env_pgdir, p, e->env_ipc_dstva, perm);
590     }
591     return 0;
592 }
```

图 8　sys_ipc_can_send 函数实现

Exercise 4.8

> **Exercise 4.8** 填写 lib/syscall_all.c 中的 sys_env_alloc 函数

sys_env_alloc 用于为当前进程分配一个新的子进程：
调用 env_alloc 函数为新的子进程分配一个 struct Env 结构体，并将其地址存储在 e 变量中。如果分配失败，则返回错误码。
使用 bcopy 函数将当前进程的 Trapframe 复制到新的子进程的 Trapframe 中。这包括复制整个寄存器状态以及 CPU 相关的信息。
将新进程的 PC 寄存器设置为 cp0_epc，以便在新进程开始执行时从正确的地址开始执行。
将新进程的寄存器 $v0 设置为 0，这是一个通用的初始化值。
将新进程的状态 env_status 设置为 ENV_NOT_RUNNABLE，表示新进程还未准备好运行。
将新进程的优先级 env_pri 设置为当前进程的优先级。
返回新进程的进程 ID。

sys_env_alloc 函数如图 9 所示。

```
429 int sys_env_alloc(void)
430 {
431     // Your code here.
432     int r;
433     struct Env *e;
434     r = env_alloc(&e, curenv->env_id);
435     if(r < 0){
436         return r;
437     }
438     bcopy((void*)KERNEL_SP - sizeof(struct Trapframe), (void *)&(e->env_tf), sizeof(struct Trapframe));
439     e->env_tf.pc = e->env_tf.cp0_epc;
440     e->env_tf.regs[2] = 0;
441     e->env_status = ENV_NOT_RUNNABLE;
442     e->env_pri = curenv->env_pri;
443     return e->env_id;
444 }
```

图 9　sys_env_alloc 函数实现

Exercise 4.9

> **Exercise 4.9** 填写 user/fork.c 中的 fork 函数中关于 sys_env_alloc 的部分和“子进程”执行的部分 ■

fork 函数实现了在当前进程的基础上创建一个新的子进程：

调用 syscall_env_alloc 系统调用函数，分配一个新的进程并获取其进程 ID (newenvid)。这个系统调用在内核中的实现就是 sys_env_alloc 函数。

在新进程创建成功的情况下，判断当前进程是否是父进程。如果是父进程，将全局变量 env 设置为指向新进程的 struct Env 结构体。这样，父子进程在后续的操作中可以根据这个变量来区分。

遍历当前进程的所有用户空间页表项，对于每一个有效的页表项，调用 duppage 函数将其复制到新进程的页表中。这样，父子进程共享相同的物理页面，实现了写时复制 (Copy-On-Write, COW)。

调用 syscall_mem_alloc 系统调用函数，在新进程的用户栈上分配一个页面，并设置相应的权限标志 (PTE_V|PTE_R)。

调用 syscall_set_pgfault_handler 系统调用函数，设置新进程的页面错误处理函数为 __asm_pgfault_handler，并指定其用户栈顶地址 (UXSTACKTOP)。

调用 syscall_set_env_status 系统调用函数，将新进程的状态设置为 ENV_RUNNABLE，表示新进程已准备好运行。

返回新进程的进程 ID (newenvid)。

代码如图 10 所示。

```
163 int fork(void)
164 {
165     // Your code here.
166     u_int newenvid;
167     extern struct Env *envs;
168     extern struct Env *env;
169     u_int i;
170     set_pgfault_handler(pgfault);
171     newenvid = syscall_env_alloc();
172     if(newenvid == 0){
173         env = envs + ENVX(syscall_getenvid());
174     }
175     if(newenvid){
176         for(i = 0; i < VPN(USTACKTOP); i++){
177             if((((Pde *)(*vpd))[i >> 10] & PTE_V ) && (((Pte *)(*vpt))[i] & PTE_V) ){
178         duppage(newenvid, i);
179             }
180         }
181 
182         syscall_mem_alloc(newenvid, UXSTACKTOP - BY2PG, PTE_V|PTE_R);
183         syscall_set_pgfault_handler(newenvid, __asm_pgfault_handler, UXSTACKTOP);
184         syscall_set_env_status(newenvid, ENV_RUNNABLE);
185     }
186     return newenvid;
187 }
```

图 10　fork 函数实现

Exercise 4.10

Exercise 4.10 结合注释，填写 user/fork.c 中的 duppage 函数

duppage 函数实现了对页表项的复制，主要用于实现写时复制：

计算页面的起始虚拟地址 addr，通过将页号左移 PGSHIFT 位得到。

通过查找页表项 (*vpt)[pn] 获取该页面的权限标志 (perm)。

检查页面是否可写 (PTE_R 标志为置位) 且不是共享库 (PTE_LIBRARY 标志未置位)。如果是，则将权限标志中加入 PTE_COW 标志，表示启用写时复制机制，并设置 flag 为 1。

调用 syscall_mem_map 系统调用函数，将当前进程中的页面映射到目标进程 (envid 所指定的进程) 的相同虚拟地址处，共享物理页面。这里实际上是将两个进程的页表项指向同一个物理页面。
如果启用了写时复制机制 (flag 为 1)，再次调用 syscall_mem_map，将当前进程中的页面映射到当前进程的相同虚拟地址处，同时保留 PTE_COW 标志。这一步的目的是为了避免写时复制过程中引起的递归调用。

代码如图 11 所示。

```
132 static void duppage(u_int envid, u_int pn)
133 {
134     u_int addr;
135     u_int perm;
136     int flag = 0;
137     addr = pn << PGSHIFT;
138     perm = (*vpt)[pn] & (BY2PG - 1);
139     if((perm & PTE_R) && !(perm & PTE_LIBRARY)){
140         perm |= PTE_COW;
141         flag = 1;
142     }
143     syscall_mem_map(0, addr, envid, addr, perm);
144     if(flag){
145         syscall_mem_map(0, addr, 0, addr, perm);
146     }
147 }
```

图 11　duppage 函数实现

Exercise 4.11

Exercise 4.11　完成 lib/traps.c 中的 page_fault_handler 函数

page_fault_handler 函数是用于处理页错误异常的：

将当前的 Trapframe 复制一份到 PgTrapFrame 中。

检查发生页错误的地址是否在用户异常栈中。如果是，说明页错误发生在用户态的异常栈上。

如果是用户态异常栈上的页错误，将用户态异常栈指针 regs[29] 向下移动一页大小，为异常栈的下一帧保存一个新的 Trapframe。

使用 bcopy 将原始的 Trapframe 复制到用户态异常栈上。

如果页错误发生在内核态，或者不在用户态异常栈上，将用户态异常栈指针 regs[29] 设置为用户异常栈的顶部减去一个 Trapframe 大小。

使用 bcopy 将原始的 Trapframe 复制到用户态异常栈的顶部。

将异常处理例程的入口地址 cp0_epc 设置为当前进程的 env_pgfault_handler，表示将控制权转移到用户定义的页错误处理函数。

代码如图 12 所示。

```c
void page_fault_handler(struct Trapframe *tf)
{
    struct Trapframe PgTrapFrame;
    extern struct Env *curenv;

    bcopy(tf, &PgTrapFrame, sizeof(struct Trapframe));

    if (tf->regs[29] >= (curenv->env_xstacktop - BY2PG) &&
        tf->regs[29] <= (curenv->env_xstacktop - 1)) {
            tf->regs[29] = tf->regs[29] - sizeof(struct  Trapframe);
            bcopy(&PgTrapFrame, (void *)tf->regs[29], sizeof(struct Trapframe));
        } else {
            tf->regs[29] = curenv->env_xstacktop - sizeof(struct  Trapframe);
            bcopy(&PgTrapFrame,(void *)curenv->env_xstacktop - sizeof(struct  Trapframe),sizeof(struct Trapframe));
        }
    // TODO: Set EPC to a proper value in the trapframe
    tf->cp0_epc = curenv->env_pgfault_handler;
    return;
}
```

图 12　page_fault_handler 函数实现

Exercise 4.12

> Exercise 4.12 完成 lib/syscall_all.c 中的 sys_set_pgfault_handler 函数

sys_set_pgfault_handler 用于设置进程的页错误处理函数和异常栈的栈顶：

通过 envid2env 函数获取指定 envid 对应的进程结构体 env。

如果找不到对应的进程，返回错误码。

将进程的页错误处理函数 env_pgfault_handler 设置为参数 func，表示将控制权转交给这个函数来处理页错误。

将异常栈的栈顶 env_xstacktop 设置为参数 xstacktop，表示用户可以提供一个自定义的异常栈。

返回 0 表示成功设置页错误处理函数和异常栈。

代码如图 13 所示。

```c
int sys_set_pgfault_handler(int sysno, u_int envid, u_int func, u_int xstacktop)
{
    // Your code here.
    struct Env *env;
    int ret;
    ret = envid2env(envid, &env, 0);
    if(ret  < 0){
        return ret;
    }
    env->env_pgfault_handler = func;
    env->env_xstacktop = xstacktop;
    return 0;
}
```

图 13　sys_set_pgfault_handler 函数实现

Exercise 4.13

Exercise 4.13 填写 user/fork.c 中的 pgfault 函数

pgfault 函数是一个页面错误处理程序，当发生写时拷贝页面的页面错误时调用。以下是其功能的概述：

提取权限： 从页表获取虚拟地址 va 处页面的权限，并将其存储在 perm 变量中。

检查 COW： 验证页面是否设置了写时拷贝（COW）标志。如果未设置 COW 标志，则表示该页面不适用于写时拷贝，将报告错误。

分配临时页面： 使用 syscall_mem_alloc 在用户空间堆栈中分配一个临时页面。分配的页面具有与原始页面相同的权限，但 COW 标志被清除。

将数据复制到临时页面： 使用 user_bcopy 将原始页面内容（向下舍入到页面边界）复制到新分配的临时页面。

将临时页面映射到原始虚拟地址： 使用 syscall_mem_map 将临时页面映射到原始虚拟地址。这个操作实际上用新分配的页面替换了原始页面。

取消映射临时页面： 使用 syscall_mem_unmap 从用户空间堆栈取消映射临时页面。

目的是实现写时拷贝语义。当进程尝试修改共享页面时，会发生页面错误。页面错误处理程序不会立即提供页面的私有副本，而是分配一个新页面，复制数据，然后将新页面重新映射到原始虚拟地址。这样，多个进程最初可以共享相同的只读页面，并且仅在尝试修改时才创建私有副本。

代码如图 14 所示。

```
82  static void pgfault(u_int va)
83  {
84      u_int *tmp;
85      u_long perm;
86      perm = ((Pte *)(*vpt))[VPN(va)] & (BY2PG -1);
87      tmp = (u_int *)USTACKTOP;
88      int r;
89      if((perm & PTE_COW) == 0){
90          user_panic("this is not a COW page");
91      }
92
93      perm = perm & (~PTE_COW);
94      r = syscall_mem_alloc(0, (u_int)tmp,perm);
95      if(r < 0){
96          return r;
97      }
98      user_bcopy(ROUNDDOWN(va,BY2PG), tmp, BY2PG);
99      r = syscall_mem_map(0, tmp, 0, va, perm);
100     if(r < 0){
101         user_panic("failed to mem map");
102     }
103     r  = syscall_mem_unmap(0, tmp);
104     if(r < 0){
105         return r;
106     }
107 }
```

图 14　pgfault 函数实现

Exercise 4.14

Exercise 4.14 填写 lib/syscall_all.c 中的 sys_set_env_status 函数

sys_set_env_status 函数用于修改指定环境的状态：

验证状态参数：首先，函数验证 status 参数是否为合法的环境状态，即 ENV_RUNNABLE、ENV_NOT_RUNNABLE 或 ENV_FREE。

获取目标环境：使用 envid2env 函数根据给定的环境 ID（envid）获取目标环境的指针。

更新环境状态：将目标环境的状态更新为新的状态值。

调整调度链表：如果目标环境之前处于不可运行状态，且新状态为可运行状态 (ENV_RUNNABLE)，则将其添加到可运行环境链表（env_sched_list[0]）。如果目标环境之前处于可运行状态，但新状态不是可运行状态，就从可运行链表中移除。

返回 0 表示成功修改环境状态。

代码如图 15 所示。

```
457 int sys_set_env_status(int sysno, u_int envid, u_int status)
458 {
459     // Your code here.
460     struct Env *env;
461     int ret;
462     if(status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE && status != ENV_FREE)
463     {
464         return -E_INVAL;
465     }
466     ret = envid2env(envid, &env, 0);
467     if(ret != 0) return ret;
468     if(env->env_status != ENV_RUNNABLE && status == ENV_RUNNABLE){
469         LIST_INSERT_HEAD(&env_sched_list[0], env, env_sched_link);
470     }
471     if(env->env_status == ENV_RUNNABLE && status != ENV_RUNNABLE){
472         LIST_REMOVE(env, env_sched_link);
473     }
474     env->env_status = status;
475     return 0;
476 }
```

图 15　sys_set_env_status 函数实现

Exercise 4.15

Exercise 4.15 填写 user/fork.c 中的 fork 函数中关于“父进程”执行的部分

Thinking 部分：

Thinking 4.1

> **Thinking 4.1** 思考并回答下面的问题：
>
> - 内核在保存现场的时候是如何避免破坏通用寄存器的？
> - 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？
> - 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？
> - 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？

1.内核保存现场避免破坏通用寄存器：在系统调用发生时，内核会将通用寄存器的值保存到内核栈或其他指定的位置，以保护这些寄存器的值。通过保存现场，内核确保在处理系统调用期间，通用寄存器的值不会被破坏，而在系统调用完成后可以正确地恢复用户进程的执行状态。

2.从 Sa0Sa3 参数寄存器中获取信息：在 MIPS 架构中，Sa0-Sa3 寄存器保存了系统调用的参数。当系统陷入内核调用时，内核可以直接从这些寄存器中获取用户调用 msyscall 留下的信息，例如系统调用号和参数。

3.sys 开头的函数参数一致性：内核通过将用户空间的 Trapframe 结构传递给内核，在 Trapframe 中保存了进入内核时的所有寄存器状态。在 sys 开头的函数中，这个 Trapframe 结构被认为保存了和用户调用 msyscall 时相同的参数信息，因此通过访问 Trapframe 结构，可以获取用户进程在发起系统调用时的寄存器状态和其他相关信息。

4.内核处理系统调用对 Trapframe 的更改，内核在处理系统调用时会对 Trapframe 进行修改，主要体现在以下几个方面：

保存寄存器状态：内核保存了用户进程在进行系统调用时的寄存器状态，以便在系统调用执行完毕后正确还原。

设置 PC 和 CP0_EPC：内核将程序计数器 (PC) 设置为系统调用服务例程的入口地址，同时设置 CP0_EPC 以保存用户进程的下一条指令地址，以便在系统调用返回时回到正确的用户态地址。

更新 CP0_STATUS： 可能会修改 CP0_STATUS 寄存器的值，以调整内核的运行环境，例如开启或关闭中断。

对应用户态的变化是，在系统调用执行完毕后，恢复了用户进程的寄存器状态和程序计数器，使得用户进程在返回到用户态时可以继续执行。

Thinking 4.2

> **Thinking 4.2** 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？
> - 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？

子进程完全按照 fork() 之后父进程的代码执行，说明了什么？

这说明子进程是通过复制父进程的地址空间来创建的。fork 系统调用会创建一个新的进程（子进程），并将父进程的整个地址空间复制到子进程中，包括代码段、数据段、堆、栈等。因此，子进程在创建时与父进程具有相同的程序和数据，从而实现了代码共享。

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？

这说明 fork 系统调用后，父进程和子进程分别开始执行，而不是在调用 fork 之前的那一刻。在 fork 调用后，父进程和子进程在同一执行点继续执行，但它们是独立的执行实体。因此，子进程并不会执行 fork 调用之前父进程的代码。

Thinking 4.3

> **Thinking 4.3** 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

选择 C

Thinking 4.4

> **Thinking 4.4** 如果仔细阅读上述这一段话，你应该可以发现，我们并不是对所有的用户空间页都使用 duppage 进行了保护。那么究竟哪些用户空间页可以保护，哪些不可以呢，请结合 include/mmu.h 里的内存布局图谈谈你的看法。 ■

用户栈区：

用户栈区包含了用户进程的运行时栈，它的地址范围是 [USTACKBOTTOM, USTACKTOP)。

用户栈区是每个用户进程独有的，不需要保护。因为每个进程都有自己的栈，它们的栈是独立的，不会相互影响。

用户堆区：

用户堆区是用于动态内存分配的，它的地址范围是 [UHEAPBOTTOM, UHEAPTOP)。

用户堆区是需要保护的，因为它是由多个用户进程共享的内存区域。一个进程对堆的修改可能会影响

其他进程。

代码区和数据区：
代码区和数据区包含程序的代码和静态数据，它们的地址范围是 [UTEXT, DATA]。
代码区和数据区也是共享的，因此需要保护。多个进程可以共享相同的代码，但不能同时写入。

Thinking 4.5

Thinking 4.5 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？
- 从实现的角度谈一下为什么能够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？
- 进程能够通过这种存取的方式来修改自己的页表项吗？

1. vpd 和 vpt 的作用：
vpd（Virtual Page Directory）指向当前进程的页目录表的起始地址。
vpt（Virtual Page Table）是一个二维数组，每个元素是一个指向当前进程的某个页表的指针。

使用方法：
vpd 用于访问当前进程的页目录表，vpt 用于访问当前进程的页表。
对于一个虚拟地址 va，可以通过 vpd[PDX(va)] 定位到页目录表中的某一项，然后通过 vpt[PTX(va)] 定位到页表中的某一项。

2. 实现原理：
这种方式能够通过自映射的设计来存取进程自身页表。在 MIPS 体系结构中，页表的结构是一种自映射结构，页目录表和页表的布局是相同的。每个页目录项和页表项的虚拟地址都指向了相同的页表。vpd 和 vpt 就是通过直接使用当前进程的页目录表和页表，利用页表的自映射特性，从而实现了对自身页表的访问。
这种设计在复制页表时非常方便，因为它可以直接通过共享页目录表的方式，将整个页表进行复制。
自映射设计体现：

3. 自映射设计的核心思想是将页目录表和页表的布局设置成相同，使得每个页目录项和页表项的虚拟地址都指向相同的页表。
在这个设计中，vpd 和 vpt 都直接指向了当前进程的页目录表和页表，使得对于自身页表的访问变得非常简便，也方便了页面的共享和复制。

4. 进程通过 vpd 和 vpt 存取的是自身的页表和页目录表的副本，并且这些表在用户态是只读的，因此进程不能直接通过这种方式来修改自己的页表项。修改页表项需要在内核态进行，通过系统调用来完成。

Thinking 4.6

> **Thinking 4.6** 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”？
> - 内核为什么需要将异常的现场 Trapframe 复制到用户空间？

1. 在多任务操作系统中，可能会出现中断重入的情况。中断重入是指在处理一个中断的过程中，又发生了新的中断。这种情况可能导致当前中断处理被打断，转而处理新的中断，然后再返回继续处理原来的中断。在这个过程中，需要保存和恢复现场，以确保各个中断处理过程之间不会相互干扰。
2. 在 page fault handler 中，可能会在异常处理的过程中触发新的异常，例如缺页异常可能会导致页面置换，而页面置换的过程中又可能发生新的缺页异常。为了支持这种中断重入的情况，需要将当前的 Trapframe 复制到用户空间，以便下次异常处理继续执行。这样可以确保在异常处理的过程中，多次发生异常时能够正确保存和恢复现场。

Thinking 4.7

> **Thinking 4.7** 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：
>
> - 用户处理相比于在内核处理写时复制的缺页中断有什么优势？
> - 从通用寄存器的用途角度讨论用户空间下进行现场的恢复是如何做到不破坏通用寄存器的？

1. 在用户处理缺页中断时，相比于在内核处理写时复制的缺页中断，主要的优势在于用户程序可以更灵活地处理自身的异常情况，而不需要每次都陷入内核。这样可以减少内核态和用户态之间的切换开销，提高系统的性能。用户程序能够直接处理缺页异常，进行页面置换或其他相关操作，而不必通过内核的中间层。
2. 从通用寄存器的用途角度来看，用户空间下进行现场的恢复是通过将 Trapframe 复制到用户栈上实现的。在异常处理的过程中，由于用户程序的运行是在用户态，通用寄存器中的值是用户程序自身的状态。通过将 Trapframe 复制到用户栈上，可以保存当前用户程序的运行状态，而不会破坏通用寄存器的值。这样，在异常处理完成后，用户程序可以从保存的 Trapframe 中恢复运行现场，确保通用寄存器的状态不被破坏。

Thinking 4.8

> **Thinking 4.8** 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？
> - 如果放置在写时复制保护机制完成之后会有怎样的效果？
> - 子进程需不需要对在 entry.S 定义的字 ___pgfault_handler 赋值？

1. 在设置 set_pgfault_handler 之前调用 syscall_env_alloc 的原因是确保子进程在开始执行时已经设置了页错误处理函数。syscall_env_alloc 函数中会为子进程创建一个新的地址空间，包括页表等，而在这个过程中可能会发生缺页异常。如果在 syscall_env_alloc 之前调用 set_pgfault_handler，那么子进程在创建地址空间的过程中，如果发生了缺页异常，将会陷入内核，但由于还没有设置好页错误处理函数，可能导致异常处理不当，甚至系统崩溃。

2. 如果将 set_pgfault_handler 放置在写时复制保护机制完成之后，那么在子进程执行时，如果发生缺页异常，将会陷入内核，但此时写时复制保护已经完成，父子进程共享的页面已经建立，可能会导致异常处理混乱，例如修改了共享的页面。

3. 子进程需要对在 entry.S 定义的 pgfault 处理函数赋值。在子进程复制父进程的页表时，需要复制 entry.S 中定义的 pgfault 处理函数地址，以保证在子进程中也能够正确处理缺页异常。

本次实验耗时 9 小时