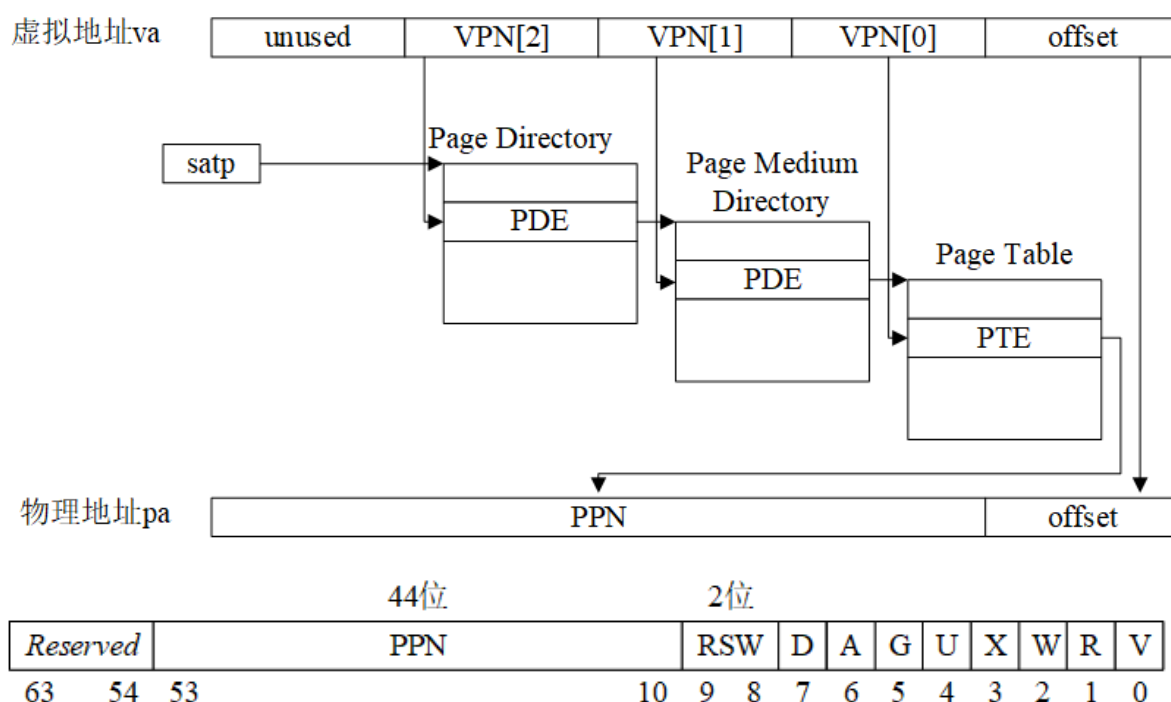


# 内存管理

## Lab2\_1 地址映射问题



其中的各个位的含意为：

- V (Valid) 位决定了该PDE/PTE是否有效 (V=1时有效) ，即是否有对应的实页。
- R (Read) 、W (Write) 和X (eXecutable) 位分别表示此页对应的实页是否可读、可写和可执行。这3个位只对PTE有意义，对于PDE而言这3个位都为0。
- U (User) 位表示该页是不是一个用户模式页。如果U=1，表示用户模式下的代码可以访问该页，否则就表示不能访问。S模式下的代码对U=1页面的访问取决于sstatus寄存器中的SUM字段取值。
- G (Global) 位表示该PDE/PTE是不是全局的。我们可以把操作系统中运行的一个进程，认为是一个独立的地址空间，有时希望某个虚地址空间转换可以在一组进程中共享，这种情况下，就可以将某个PDE的G位设置为1，达到这种共享的效果。
- A (Access) 位表示该页是否被访问过。
- D (Dirty) 位表示该页的内容是否被修改。
- RSW位 (2位) 是保留位，一般由运行在S模式的代码 (如操作系统) 来使用。
- PPN (44位) 是物理页号 (Physical Page Number，简称为PPN) 。

其中PPN为44位的原因是：对于物理地址，现有的RISC-V规范只用了其中的56位，同时，这56位中的低12位为页内位移。所以，PPN的长度=56-12=44 (位) 。

## 一些宏和结构体定义

## kernel / process.h

```
typedef struct trapframe {
    // space to store context (all common registers)
    /* offset:0    */ riscv_regs regs;

    // process's "user kernel" stack
    /* offset:248 */ uint64 kernel_sp;
    // pointer to smode_trap_handler
    /* offset:256 */ uint64 kernel_trap;
    // saved user process counter
    /* offset:264 */ uint64 epc;

    //kernel page table
    /* offset:272 */ uint64 kernel_satp;
}trapframe;
typedef struct process {
    // pointing to the stack used in trap handling.
    uint64 kstack;
    // user page table
    pagetable_t pagetable;
    // trapframe storing the context of a (User mode) process.
    trapframe* trapframe;
}process;
extern process* current;
```

## kernel / riscv.h

```
#define MAXVA (1L << (9 + 9 + 9 + 12 - 1))
#define PTE2PA(pte) (((pte) >> 10) << 12) //抹去10位标志位 在最右侧补上12个'0'作为页内偏移量
#define PA2PTE(pa) (((uint64)pa) >> 12) << 10) //抹去12位的页内偏移量 在最右侧添加10个'0'作为10个标志位
#define PXMASK 0x1FF // 9个bit的'1'串
#define PGSHIFT 12 // bits of offset within a page 先右移PGSHIFT位 去除页内偏移量
#define PXSHIFT(level) (PGSHIFT + (9 * (level))) //level每个层级的VPN有9位,level从2到1
#define PX(level, va) (((uint64)(va)) >> PXSHIFT(level)) & PXMASK //只取最低9位,即对应层级的VPN
//上图的各个bit位的检验变量
#define PTE_V (1L << 0) // valid
#define PTE_R (1L << 1)
#define PTE_W (1L << 2)
#define PTE_X (1L << 3)
#define PTE_U (1L << 4) // 1 -> user can access
#define PTE_G (1L << 5) // Global
#define PTE_A (1L << 6) // Accessed
#define PTE_D (1L << 7) // Dirty
#define PGSIZE 4096 // 4KB per page

typedef uint64 pte_t;
typedef uint64 *pagetable_t; // 512 PTES 页表变量
```

## 用户进程

### user / app\_helloworld\_no\_lds.c

```
#include "user_lib.h"
#include "util/types.h"

int main(void) {
    printu("Hello world!\n");
    exit(0);
}
```

printu()和exit()函数的调用过程与Lab1\_1基本是一致的，唯一的区别在于 Lab1\_1不需要将虚拟地址映射为物理地址，所以在Lab2\_1需要在kernel/syscall.c中的sys\_user\_print()函数中的将buf地址进行映射为物理地址

## 内核中的重要操作

### kernel / syscall.c

```
ssize_t sys_user_print(const char* buf, size_t n) {
    //buf is an address in user space on user stack,
    //so we have to transfer it into phisical address (kernel is running in direct
    mapping).
    assert( current );
    char* pa = (char*)user_va_to_pa((pagetable_t)(current->pagetable),
    (void*)buf);
    sprint(pa);
    return 0;
}
```

### kernel / vmm.c

```
pte_t *page_walk(pagetable_t page_dir, uint64 va, int alloc) //寻找某个虚拟地址所对
应的最后一级PTE所在地址
{//page_dir为某个进程的页目录起始地址，va为目标虚拟地址，alloc==1表示允许在页面未调入时申请一
块内存当做新页面
    if (va >= MAXVA) panic("page_walk");//当虚拟地址超过MAXVA时 越界了 不用继续下面的操作

    pagetable_t pt = page_dir;//该级页表的起始地址

    for (int level = 2; level > 0; level--) {
        //PX()可以获取当前层级的VPN，pt（该级页表的起始地址）+PX（VPN类似于数组下标）
        pte_t *pte = pt + PX(level, va);//确定对应PDE的物理地址 使用*pte即可访问这一个PDE
        （+PX时，由于PX的返回值是int pte的类型是指针 所以会把PX自动左移3位再加上pt（即
        pte=pt+sizeof（一个页表项）*index）

        if (*pte & PTE_V) { //检查该页面是否valid 即：是否已经调入
            pt = (pagetable_t)PTE2PA(*pte);//通过PDE获取下一级页表的起始地址
        }
        else { //页面还没调入
```

```

        if( alloc && ((pt = (pte_t *)alloc_page(1)) != 0) ){//如果alloc==1 或者申请成功 就把这一块内存给它（一个页面的大小）
            memset(pt, 0, PGSIZE);//将这个页面的内容清空
            *pte = PA2PTE(pt) | PTE_V;//将新申请的内存物理地址填入PDE中，并且把Valid位设为1
        }else //如果申请失败（没内存了）或者alloc==0 就返回0（NULL）
            return 0;
    }
}
return pt + PX(0, va);//获取PTE的地址
}

uint64 lookup_pa(pagetable_t pagetable, uint64 va) //{//通过某个虚拟地址寻找其所在的页面的起始地址
    pte_t *pte;
    uint64 pa;

    if (va >= MAXVA) return 0;//越界则返回NULL

    pte = page_walk(pagetable, va, 0);
    if (pte == 0 || (*pte & PTE_V) == 0 || ((*pte & PTE_R) == 0 && (*pte & PTE_W) == 0))//若目标页面未调入或不可读写则返回NULL
        return 0;
    pa = PTE2PA(*pte);//通过PTE获取对应页面的起始地址

    return pa;
}

void *user_va_to_pa(pagetable_t page_dir, void *va) //{//将虚拟地址转换为物理地址
    // TODO (lab2_1): implement user_va_to_pa to convert a given user virtual address "va"
    // to its corresponding physical address, i.e., "pa". To do it, we need to walk
    // through the page table, starting from its directory "page_dir", to locate the PTE
    // that maps "va". If found, returns the "pa" by using:
    // pa = PYHS_ADDR(PTE) + (va - va & (1<<PGSHIFT -1))//PGSHIFT=12
    // Here, PYHS_ADDR() means retrieving the starting address (4KB aligned), and
    // (va - va & (1<<PGSHIFT -1)) means computing the offset of "va" in its page.
    // Also, it is possible that "va" is not mapped at all. in such case, we can find
    // invalid PTE, and should return NULL.
    if(lookup_pa(page_dir, (uint64)va)){//若页面寻找成功 则返回页面起始地址+页内偏移量
        {
            return (void *)lookup_pa(page_dir, (uint64)va)+ ((uint64)va & ((1<<PGSHIFT)-1));
        }
        else return NULL;
    }
}

```

**大致流程：**

1.用户进程调用printu()函数来输出某个字符串

2.printu()函数经过层层调用来到sys\_user\_print()函数，通过buf变量来访问缓存区读取字符串并输出

3.sys\_user\_print()函数中需要调用user\_va\_to\_pa()函数，来实现虚拟地址对物理地址的转换（因为此处传进的buf变量的地址还是相对于用户进程的虚拟地址，并非在内存中的实际物理地址）

4.在user\_va\_to\_pa()函数中调用lookup\_pa()获取该虚拟地址va所对应的物理块首地址，再加上页内偏移量即可获得va对应的物理地址

5.lookup\_pa()中调用了page\_walk()函数来解析前两级页表（会返回最后一级PTE的地址），再根据PTE获得对应物理块的地址

Q1:在page\_walk () 函数中，pte=pt+PX(level,va) PX只是取出了VPN[level] 代表第VPN[level]个页表项 而一个页表项有8B，为什么不需要左移3位：由于pte是指针类型的变量，PX的返回值是整型变量，所以在相加时会自动把PX当成一个下标处理，自动左移三位再相加

Q2:在page\_walk () 函数中，发现某个页面还未被调入后，为什么只是申请了一个全0的页面，却没有去外存中调取对应的页面进来：因为这里的缺页并不是实际的进程页，只是中间页面，它只负责记录更底层页表的起始地址和其他信息，这个页面在第一次被发现未调入时，它所管辖的其他页面肯定还没有被访问过，自然也没有对应的物理地址和标志位，所以全0即可

这行注释有问题：页内偏移量的计算方式应该是：va&((1<<PGSHIFT)-1)

```
// pa = PYHS_ADDR(PTE) + (va - va & (1<<PGSHIFT - 1))//PGSHIFT=12
```

## Lab2\_2 内存空间的分配和回收

### 一些宏和结构体定义

kernel / process.c

```
uint64 g_ufree_page = USER_FREE_ADDRESS_START;
```

kernel / memlayout.h

```
#define USER_FREE_ADDRESS_START 0x00000000 + PGSIZE * 1024// 即虚拟地址空间从4MB开始分配
```

### 用户进程

## user / app\_naive\_malloc.c

```
#include "user_lib.h"
#include "util/types.h"

struct my_structure {
    char c;
    int n;
};

int main(void) {
    struct my_structure* s = (struct my_structure*)naive_malloc();
    s->c = 'a';
    s->n = 1;

    printf("s: %lx, {%c %d}\n", s, s->c, s->n);

    naive_free(s);
    exit(0);
}
```

## 内核中的重要操作

### user / user\_lib.c

```
uint64 do_user_call(uint64 sysnum, uint64 a1, uint64 a2, uint64 a3, uint64 a4,
uint64 a5, uint64 a6, uint64 a7) {
    int ret;
    // before invoking the syscall, arguments of do_user_call are already loaded
    into the argument
    // registers (a0-a7) of our (emulated) risc-v machine.
    asm volatile(
        "ecall\n" // 会跳转到 smode_trap_handler() 函数
        "sw a0, %0" // returns a 32-bit value
        : "=m"(ret)
        :
        : "memory");

    return ret;
}

void* naive_malloc() {
    return (void*)do_user_call(SYS_user_allocate_page, 0, 0, 0, 0, 0, 0,
0); // do_user_call() 会跳转到 smode_trap() 函数
}

//
// lib call to naive_free
//
void naive_free(void* va) {
    do_user_call(SYS_user_free_page, (uint64)va, 0, 0, 0, 0, 0, 0);
}
```

## kernel / strap.c

```
void smode_trap_handler(void) {
    // make sure we are in User mode before entering the trap handling.
    // we will consider other previous case in lab1_3 (interrupt).
    if ((read_csr(sstatus) & SSTATUS_SPP) != 0) panic("usertrap: not from user mode");

    assert(current);
    // save user process counter.
    current->trapframe->epc = read_csr(sepc);

    // if the cause of trap is syscall from user application
    uint64 cause = read_csr(scause);

    if (cause == CAUSE_USER_ECALL) {
        handle_syscall(current->trapframe);
    } else if (cause == CAUSE_MTIMER_S_TRAP) { //soft trap generated by timer
interrupt in M mode
        handle_mtimer_trap();
    } else {
        sprint("smode_trap_handler(): unexpected scause %p\n", read_csr(scause));
        sprint("          sepc=%p stval=%p\n", read_csr(sepc), read_csr(stval));
        panic( "unexpected exception happened.\n" );
    }

    // continue the execution of current process.
    switch_to(current);
}

static void handle_syscall(trapframe *tf) {
    // tf->epc points to the address that our computer will jump to after the trap handling.
    // for a syscall, we should return to the NEXT instruction after its handling.
    // in RV64G, each instruction occupies exactly 32 bits (i.e., 4 Bytes)
    tf->epc += 4;

    // TODO (lab1_1): remove the panic call below, and call do_syscall (defined in
    // kernel/syscall.c) to conduct real operations of the kernel side for a syscall.
    // IMPORTANT: return value should be returned to user app, or else, you will encounter
    // problems in later experiments!
    tf->regs.a0=do_syscall(tf->regs.a0,tf->regs.a1,tf->regs.a2,tf->regs.a3,tf->regs.a4,tf->regs.a5,tf->regs.a6,tf->regs.a7);
}
```

## kernel / syscall.c

```
long do_syscall(long a0, long a1, long a2, long a3, long a4, long a5, long a6, long a7) {
    switch (a0) {
        case SYS_user_print:
            return sys_user_print((const char*)a1, a2);
        case SYS_user_exit:
            return sys_user_exit(a1);
        case SYS_user_allocate_page://这两个是本次实验需要用到的
```

```

        return sys_user_allocate_page();
    case SYS_user_free_page: //这两个是本次实验需要用到的
        return sys_user_free_page(a1);
    default:
        panic("Unknown syscall %ld \n", a0);
    }
}

uint64 sys_user_allocate_page() {
    void* pa = alloc_page(); //申请一个页面并返回其首地址
    uint64 va = g_ufree_page; //g_ufree_page为当前还未使用到的最小虚拟地址（以一页4KB为跨度
    一次+4KB）
    g_ufree_page += PGSIZE;
    user_vm_map((pagetable_t)current->pagetable, va, PGSIZE, (uint64)pa,
        prot_to_type(PROT_WRITE | PROT_READ, 1)); //prot_to_type()会返回一个二进制数
    用来标志某个PTE的权限

    return va;
}

uint64 sys_user_free_page(uint64 va) {
    user_vm_unmap((pagetable_t)current->pagetable, va, PGSIZE, 1);
    return 0;
}

```

## kernel / vmm.c

```

void user_vm_map(pagetable_t page_dir, uint64 va, uint64 size, uint64 pa, int
perm) {
    if (map_pages(page_dir, va, size, pa, perm) != 0) {
        panic("fail to user_vm_map .\n");
    }
}

void user_vm_unmap(pagetable_t page_dir, uint64 va, uint64 size, int free) {
    // TODO (lab2_2): implement user_vm_unmap to disable the mapping of the
    virtual pages
    // in [va, va+size], and free the corresponding physical pages used by the
    virtual
    // addresses when if free is not zero.
    // basic idea here is to first locate the PTEs of the virtual pages, and then
    reclaim
    // (use free_page() defined in pmm.c) the physical pages. lastly, invalidate
    the PTEs.
    // as naive_free reclaims only one page at a time, you only need to consider
    one page
    // to make user/app_naive_malloc to produce the correct behavior.
    if(free==0) return ;
    uint64 first, last;
    pte_t *pte;
    uint64 pa; // 首地址pa

    for (first = ROUNDDOWN(va, PGSIZE), last = ROUNDDOWN(va + size - 1,
    PGSIZE); first <= last; first += PGSIZE, pa += PGSIZE) { //需要检查每一个与
    [va,va+size-1]有重合部分的虚拟页
        if ((pte = page_walk(page_dir, first, 0)) != 0) { // 找到va对应的PTE
            pa = lookup_pa(page_dir, (uint64)va); // 找到va对应的pa
            if ( pa != 0 && free != 0 ) { //若pa是合法地址且free!=0 则可以释放地址了

```



```

        free_page((void *)pa);
    }
    *pte = 0;
}
}
}

int map_pages(pagetable_t page_dir, uint64 va, uint64 size, uint64 pa, int perm)
{
    //判断从va开始是否有足够大的（size大小）未使用的虚拟空间
    uint64 first, last;
    pte_t *pte;

    for (first = ROUNDDOWN(va, PGSIZE)/*取整*/, last = ROUNDDOWN(va + size - 1, PGSIZE); first <= last; first += PGSIZE, pa += PGSIZE) {
        if ((pte = page_walk(page_dir, first, 1)) == 0) return -1; //已经没有空间分配了
        if (*pte & PTE_V) //这个虚拟空间所对应的页面已经被其他东西使用了
            panic("map_pages fails on mapping va (0x%lx) to pa (0x%lx)", first, pa);
        *pte = PA2PTE(pa) | perm | PTE_V;
    }
    return 0;
}

```

## 大致流程：

1.用户进程使用naive\_malloc()和naive\_free()函数

2.这两个函数经过层层跳转来到sys\_user\_allocate\_page()和sys\_user\_free\_page()两个函数

3.sys\_user\_allocate\_page()已经实现好了，大致过程为从内存中获取一个页面并返回其首地址，检查想要与其对应的虚拟地址是否符合要求，若符合则修改该PTE的权限和地址段

4.sys\_user\_free\_page()则要调用user\_vm\_unmap()函数来释放空间

5.user\_vm\_unmap()通过va找到其对应的所有页面PTE和物理地址，若找到的是合法的地址就可以直接释放了

## Lab2\_3 缺页异常处理

mideleg	Machine Interrupt Delegation Registers。中断代理寄存器。
medeleg	Machine Exception Delegation Registers。异常代理寄存器。

#### 1.4.4 RISC-V的中断代理机制

RISC-V在中断处理上有一个很有意思的设计，就是可以将系统中的特定中断或者异常，通过设置较高特权级的CSR寄存器，“代理给”某个更低的特权级处理。例如，我们可以设置机器模式的`mideleg`以及`medeleg`中的某些位，将系统中的部分中断（对应`mideleg`）或异常（对应`medeleg`）“代理”给较低特权级的监管模式来处理；同理，我们也可以设置监管模式的`sideleg`以及`sedeleg`中的某些位，将系统中的部分中断或异常“代理”给用户特权级的代码来处理。

例如，我们的PKE代码中，有以下的等效代码：

```
csrwr mideleg, 1<<1 | 1<<5 | 1<<9
csrwr medeleg, 1<<0 | 1<<3 | 1<<8 | 1<<12 | 1<<13 | 1<<15
```

这段代码的作用是将M模式中interrupt中的1、5和9号，分别对应Supervisor software interrupt, Supervisor timer interrupt和Supervisor external interrupt代理出去，到S模式处理；再将M模式中的exception（或trap）中的0、3、8、12、13和15号，分别对应Instruction address misaligned，调试中断Breakpoint（3号），用户态系统调用Environment call from U-mode（8号），缺页或访存异常（12、13和15号）代理出去，到S模式处理。实际上，将这些重要的中断代理出去后，系统中产生的绝大部分中断事件将都在S模式处理。所以，在其后的PKE实验中，读者主要跟U模式以及S模式的代码打交道，除启动过程和一些简单的设置（如访存、中断代理等），实验也基本不涉及M模式的代码。

## 一些宏和结构体定义

### kernel / riscv.h

```
#define IRQ_S_EXT 9                // s-mode external interrupt
#define IRQ_S_TIMER 5              // s-mode timer interrupt
#define IRQ_S_SOFT 1               // s-mode software interrupt
#define IRQ_M_SOFT 3              // m-mode software interrupt

// fields of mip, the Machine Interrupt Pending register
#define MIP_SEIP (1 << IRQ_S_EXT) // s-mode external interrupt pending
#define MIP_SSIP (1 << IRQ_S_SOFT) // s-mode software interrupt pending
#define MIP_STIP (1 << IRQ_S_TIMER) // s-mode timer interrupt pending
#define MIP_MSIP (1 << IRQ_M_SOFT) // m-mode software interrupt pending

#define CAUSE_MISALIGNED_FETCH 0x0 // Instruction address misaligned
#define CAUSE_FETCH_ACCESS 0x1     // Instruction access fault
#define CAUSE_ILLEGAL_INSTRUCTION 0x2 // Illegal Instruction
#define CAUSE_BREAKPOINT 0x3       // Breakpoint
#define CAUSE_MISALIGNED_LOAD 0x4   // Load address misaligned
#define CAUSE_LOAD_ACCESS 0x5       // Load access fault
#define CAUSE_MISALIGNED_STORE 0x6   // Store/AMO address misaligned
#define CAUSE_STORE_ACCESS 0x7      // Store/AMO access fault
#define CAUSE_USER_ECALL 0x8        // Environment call from U-mode
#define CAUSE_SUPERVISOR_ECALL 0x9   // Environment call from S-mode
#define CAUSE_MACHINE_ECALL 0xb     // Environment call from M-mode
#define CAUSE_FETCH_PAGE_FAULT 0xc   // Instruction page fault
#define CAUSE_LOAD_PAGE_FAULT 0xd    // Load page fault
#define CAUSE_STORE_PAGE_FAULT 0xf   // Store/AMO page fault
```

## 用户进程

```
#include "user_lib.h"
#include "util/types.h"

uint64 sum_sequence(uint64 n) {
    if (n == 0)
```

```

    return 0;
else
    return sum_sequence( n-1 ) + n;
}

int main(void) {
    // we need a large enough "n" to trigger pagefaults in the user stack
    uint64 n = 1000;

    printf("Summation of an arithmetic sequence from 0 to %ld is: %ld \n", n,
sum_sequence(1000) );
    exit(0);
}

```

## 内核中的重要操作

### kernel / machine / minit.c

```

static void delegate_traps() {
    if (!supports_extension('S')) {
        // confirm that our processor supports supervisor mode. abort if not.
        printf("S mode is not supported.\n");
        return;
    }

    uintptr_t interrupts = MIP_SSIIP | MIP_STIP | MIP_SEIP;
    uintptr_t exceptions = (1U << CAUSE_MISALIGNED_FETCH) | (1U <<
CAUSE_FETCH_PAGE_FAULT) |
                        (1U << CAUSE_BREAKPOINT) | (1U <<
CAUSE_LOAD_PAGE_FAULT) |
                        (1U << CAUSE_STORE_PAGE_FAULT) | (1U <<
CAUSE_USER_ECALL);

    write_csr(mideleg, interrupts);
    write_csr(medeleg, exceptions);
    assert(read_csr(mideleg) == interrupts); //将某些中断代理给S模式
    assert(read_csr(medeleg) == exceptions); //将某些异常代理给S模式（其中包括了缺页异常）
}

```

### kernel / strap.c

```

void smode_trap_handler(void) {
    // make sure we are in User mode before entering the trap handling.
    // we will consider other previous case in lab1_3 (interrupt).
    if ((read_csr(sstatus) & SSTATUS_SPP) != 0) panic("usertrap: not from user
mode"); //判断是否是来自用户模式的trap

    assert(current);
    // 保存当前的PC
    current->trapframe->epc = read_csr(sepc);

    // 查看trap_cause
    uint64 cause = read_csr(scause);

    switch (cause) {

```

```

case CAUSE_USER_ECALL:
    handle_syscall(current->trapframe);
    break;
case CAUSE_MTIMER_S_TRAP:
    handle_mtimer_trap();
    break;
case CAUSE_STORE_PAGE_FAULT:
case CAUSE_LOAD_PAGE_FAULT: //此处为本次实验需要完善的地方
    // the address of missing page is stored in stval
    // call handle_user_page_fault to process page faults
    handle_user_page_fault(cause, read_csr(sepc), read_csr(stval));
    break;
default:
    sprint("smode_trap_handler(): unexpected scause %p\n", read_csr(scause));
    sprint("          sepc=%p stval=%p\n", read_csr(sepc), read_csr(stval));
    panic( "unexpected exception happened.\n" );
    break;
}

// sepc: the pc when fault happens;
// stval: the virtual address that causes pagefault when being accessed.
void handle_user_page_fault(uint64 mcause, uint64 sepc, uint64 stval) {
    sprint("handle_page_fault: %lx\n", stval);
    switch (mcause) {
        case CAUSE_STORE_PAGE_FAULT:
            // TODO (lab2_3): implement the operations that solve the page fault to
            // dynamically increase application stack.
            // hint: first allocate a new physical page, and then, maps the new page
            // to the
            // virtual address that causes the page fault.
            //panic( "You need to implement the operations that actually handle the
            //page fault in lab2_3.\n" );
            {
                void * pa = alloc_page(); // 分配一个新的物理页
                pte_t * pte;
                // 通过 stval(va) 找到对应的 pte
                if ((pte = page_walk((pagetable_t)current->pagetable, stval, 1)) == 0)
                    panic("get pte failed"); // 没找到 pte
                if (*pte & PTE_V) // 找到的 pte 所对应的虚拟地址已经被用了
                    panic("map_pages fails on mapping va (0x%lx) to pa (0x%lx)", stval, pa);
                uint64 perm = prot_to_type(PROT_WRITE | PROT_READ, 1);
                *pte = PA2PTE(pa) | perm | PTE_V; //给这个PTE赋予权限
                break;
            }
        default:
            sprint("unknown page fault.\n");
            break;
    }
}
}

```

**大致流程：**

- 1.在递归过程中，用户栈的使用已经超过了页面范围，缺页异常被系统捕捉到。
- 2.在kernel / machine / minit.c的delegate\_traps()函数中，已经将该异常代理给S模式，于是系统开始跳转到smode\_trap\_handler()函数
- 3.根据异常号，跳转到handle\_user\_page\_fault()函数
- 4.在handle\_user\_page\_fault()函数，先通过alloc\_page()函数分配一个新的物理页，并返回其首地址。再对stval（异常发生时访问的虚拟地址）使用page\_walk()函数寻找其对应的PTE的物理地址(实际上是尝试去构造一个PTE，因为理论上该页面还没被调入)
- 5.若无法创建这个PTE或者这个虚拟地址暂时还在处于被占用状态（valid位==1）则报告异常
- 6.否则就根据这个获得的物理地址和perm（即想要给予这个页面的权限变量）给这个PTE赋值