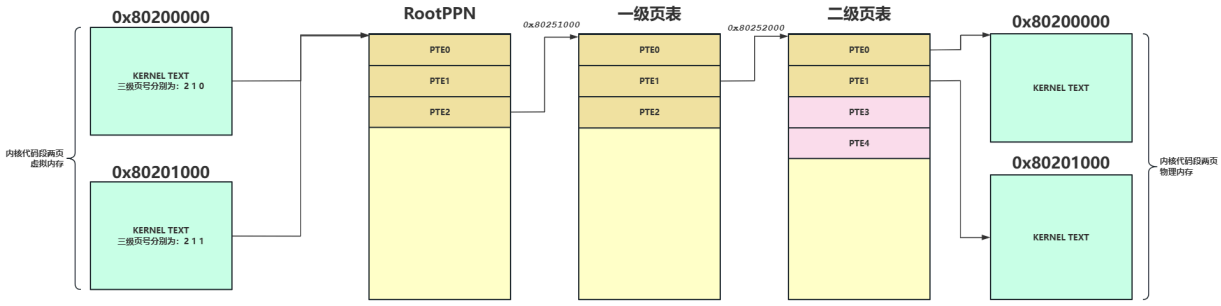
内核和用户程序的映射逻辑 | TimerのBlog

yanglianoo.github.io/2023/09/12/内核和用户程序的映射逻辑

2023年9月12日

1. 内核映射补充

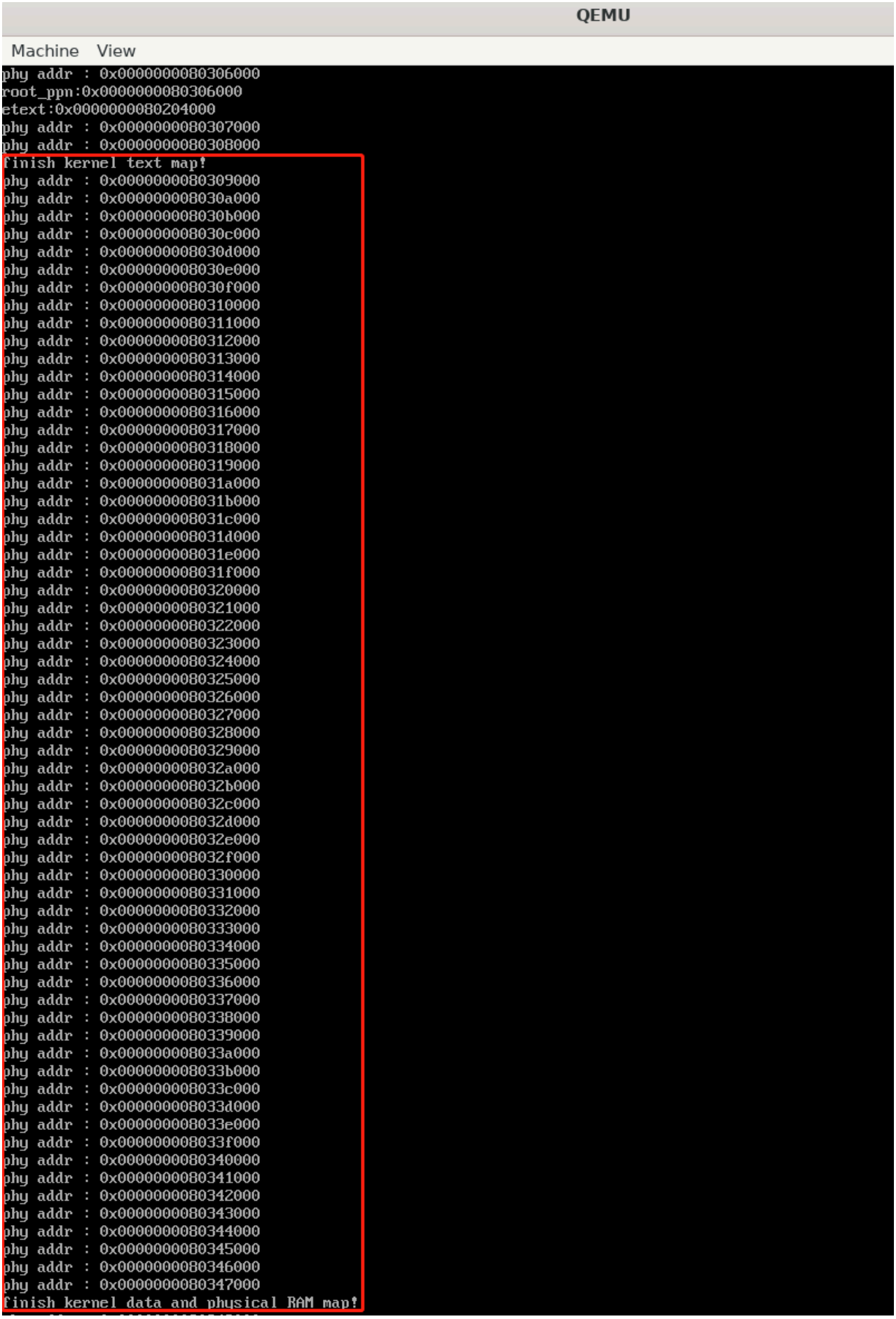
在说应用程序的内核栈映射之前先说我之前陷入的一个误区，我之前想当然的认为在进行内核映射时我们不是把从`etext ~ PHYSTOP`所在的内存都映射了吗，每一页内存的映射需要根据虚拟地址三级映射去查表，如果根据页表项的索引发现页表项为空则去分配一页内存，我就以为这后面所有的内存都被使用了，我岂不是没有空闲的物理内存页可用了，但实际不是。一页内存可以存放512个页表项，以内核的两页为例子，我们的页表使用了三页内存，虚拟地址的前两个索引都是一样的，所以在二级页表中一次往下排，依次类推`0x80202000`会放在二级页表的三号框内，`0x80203000`会放在二级页表的4号框内，所以虽然从`etext ~ PHYSTOP`的内存都被映射了，但实际占用的物理内存页并不多。我们来打印看一下:




```
PhysPageNum StackFrameAllocator_alloc(StackFrameAllocator *allocator) {
    PhysPageNum ppn;
    if (allocator->recycled.top >= 0) {
        ppn.value = pop(&(allocator->recycled));
    } else {
        if (allocator->current == allocator->end) {
            ppn.value = 0; // Return 0 as None
        } else {
            ppn.value = allocator->current++;
        }
    }
    /* 清空此页内存 ：  注意不能覆盖内核代码区，分配的内存只能是未使用部分*/
    PhysAddr addr = phys_addr_from_phys_page_num(ppn);
    memset(addr.value,0,PAGE_SIZE);
    printk("phy addr : %p\n",addr.value);
    // uint8_t* ptr = get_bytes_array(ppn);
    // for (size_t i = 0; i < PAGE_SIZE; i++)
    // {

    //     printk("%d",ptr[i]);
    //    // ptr++;
    // }
    return ppn;
}
```

在内存分配函数中加一个使用物理页的打印。



可以看见映射128M的内存占用的实际的物理内存并不多，映射只是为了告诉MMU真的要去访问一个虚拟地址时如何去查表，并不代表真的占用了，当然最后你想要去访问这128M的虚拟地址最终对应的还是128M的物理地址。

2. xv6 的应用程序内核栈映射

在timer os的设计中，之前是使用一个二维数组来作为应用程序的内核栈，我们在trap时会将应用的trap上下文保存到内核栈中

c

```
uint8_t KernelStack[MAX_TASKS]
[KERNEL_STACK_SIZE];
```

在xv6-riscv中是单独为每个应用映射了一个内核栈：

```c
// Make a direct-map page table for the kernel.
pagetable_t
kvmmake(void)
{
  pagetable_t kpgtbl;

  kpgtbl = (pagetable_t) kalloc();
  memset(kpgtbl, 0, PGSIZE);

  // uart registers
  kvmmap(kpgtbl, UART0, UART0, PGSIZE, PTE_R | PTE_W);

  // virtio mmio disk interface
  kvmmap(kpgtbl, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);

  // PLIC
  kvmmap(kpgtbl, PLIC, PLIC, 0x400000, PTE_R | PTE_W);

  // map kernel text executable and read-only.
  kvmmap(kpgtbl, KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R | PTE_X);

  // map kernel data and the physical RAM we'll make use of.
  kvmmap(kpgtbl, (uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext, PTE_R | PTE_W);

  // map the trampoline for trap entry/exit to
  // the highest virtual address in the kernel.
  kvmmap(kpgtbl, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);

  // allocate and map a kernel stack for each process.
  // 为每个进程映射一个内核栈
  proc_mapstacks(kpgtbl);

  return kpgtbl;
}
```

我们来看看这个函数：

c

```
// Allocate a page for each process's kernel stack.
// Map it high in memory, followed by an invalid
// guard page.
void
proc_mapstacks(pagetable_t kpgtbl)
{
  struct proc *p;

  for(p = proc; p < &proc[NPROC]; p++) {
    char *pa = kalloc();
    if(pa == 0)
      panic("kalloc");
    uint64 va = KSTACK((int) (p - proc));
    kvmmap(kpgtbl, va, (uint64)pa, PGSIZE, PTE_R |
PTE_W);
  }
}
```

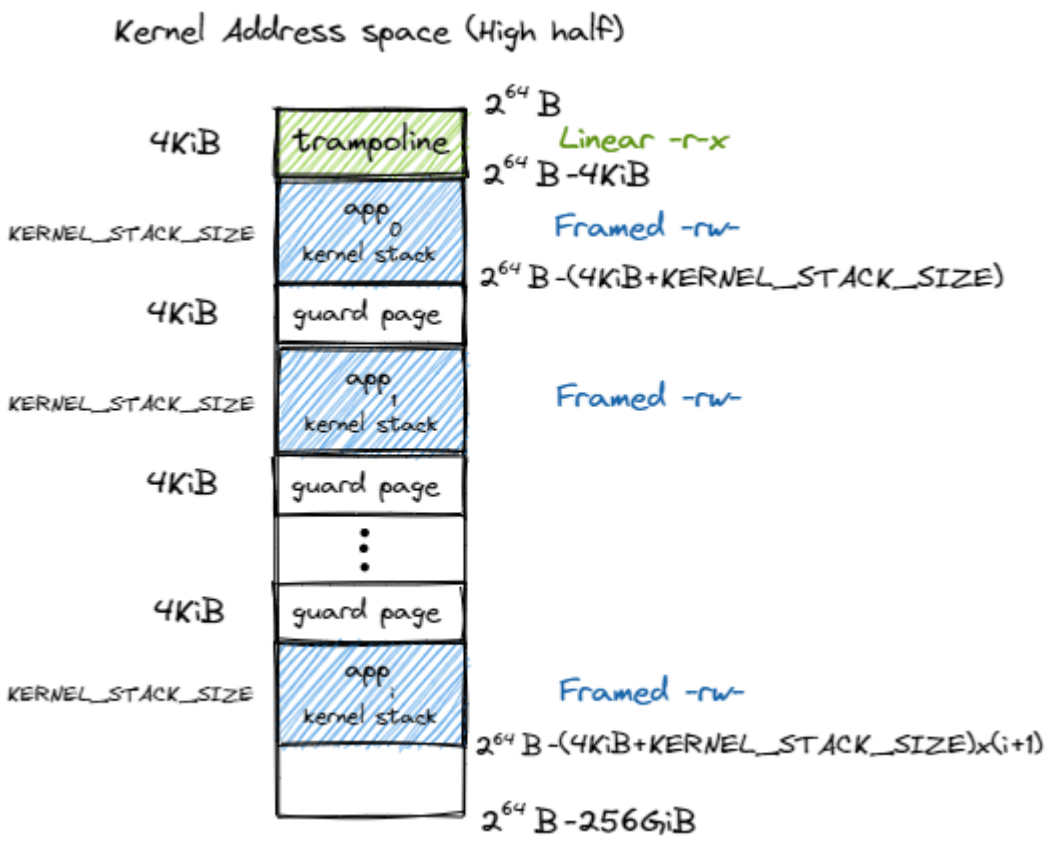
其中比较重要的一点是KSTACK这个宏：

c

```
// map the trampoline page to the highest
address,
// in both user and kernel space.
#define TRAMPOLINE (MAXVA - PGSIZE)

// map kernel stacks beneath the trampoline,
// each surrounded by invalid guard pages.
#define KSTACK(p) (TRAMPOLINE - ((p)+1)*
2*PGSIZE)
```

遍历所有的应用程序，为每个应用程序映射一个内核栈，可以看见是两页两页的跳着映射，只映射一页，另外一页用来用作栈保护，由于两页中有一页没有映射所以如果应用程序的用户栈超过了一页的大小就会触发缺页异常。并且应用程序的内核栈不是直接映射的，是映射到存放完毕页表之后的可用内存页的。



这里在虚拟地址空间的最顶端有一页叫做：`TRAMPOLINE`，这一页是跳板页，我们后面来分析。

这时候再来看xv6的内存映射图就很清楚了：

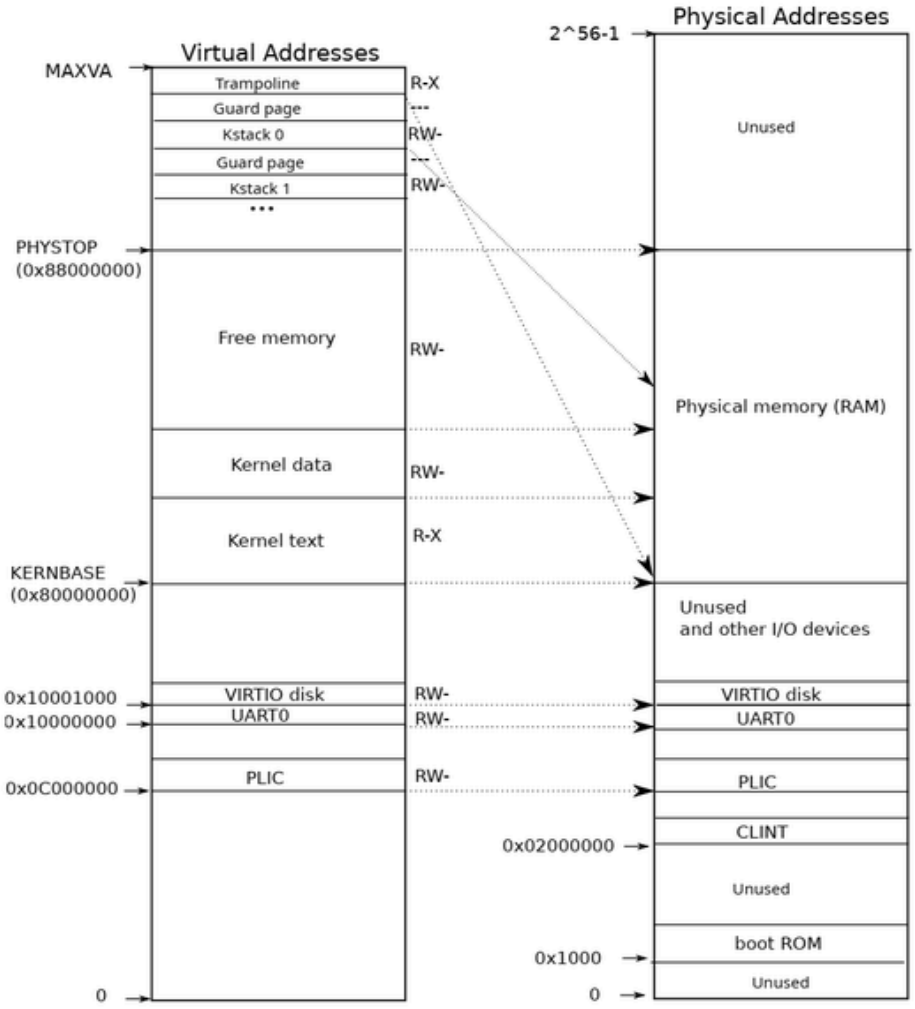
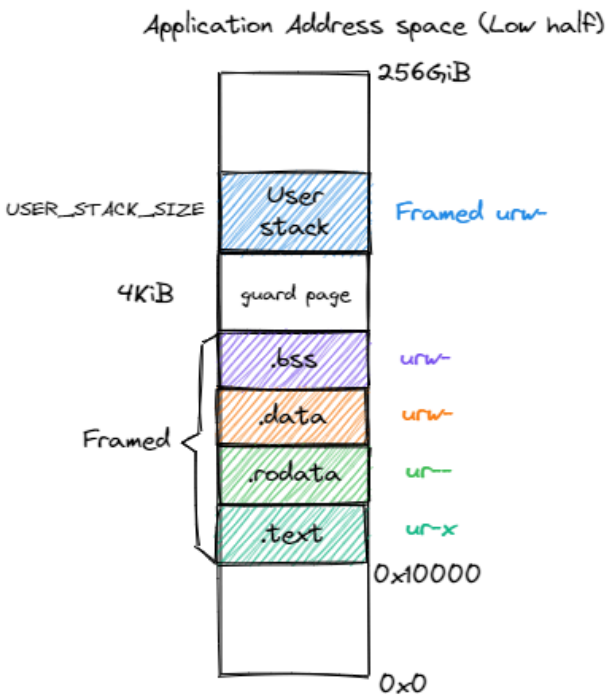
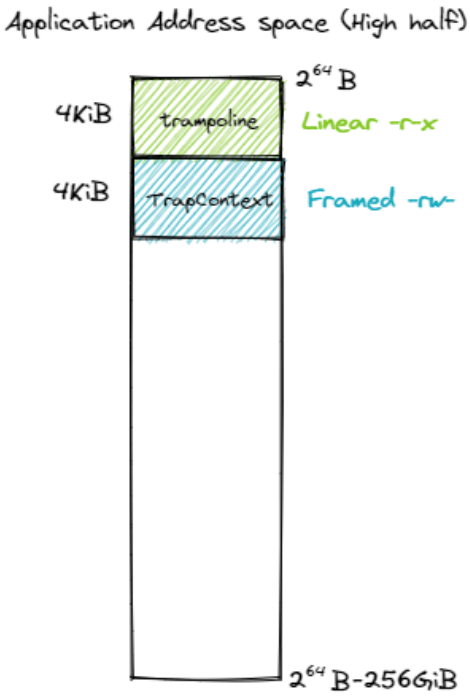


Figure 3.3: On the left, xv6's kernel address space. RWX refer to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see.

3. 应用地址空间

应用地址空间的内存布局如下图所示:





左侧链接脚本中指明了应用连接到`0x10000`，从 `0x10000` 开始向高地址放置应用内存布局中的各个逻辑段，最后放置带有一个保护页面的用户栈。

右侧则给出了最高的 256GiB ，可以看出它只是和内核地址空间一样将跳板放置在最高页，还将 Trap 上下文放置在次高页中。为啥要在应用空间的最高出映射这两页，我们后续再来分析。

由此为每个应用程序的映射逻辑就清晰了：

- 解析每个应用程序的`ELF`文件，从中解析出各逻辑段的信息
- 依次映射各逻辑段
- 映射应用程序用户栈

4. 跳板机制的实现

4.1 为什么需要跳板机制

为啥需要跳板机制，rCore中是下面这样解释的

> 在上面的内核地址空间和应用程序地址空间中我们看到无论是内核还是应用的地址空间，最高的虚拟页面都是一个跳板。同时应用地址空间的次高虚拟页面还被设置为用来存放应用的 `Trap` 上下文。那么跳板究竟起什么作用呢？为何不直接把 `Trap` 上下文仍放到应用的内核栈中呢？
>
> 在之前的设计中，当一个应用 Trap 到内核时，`sscratch` 已指向该应用的内核栈栈顶，我们用一条指令即可从用户栈切换到内核栈，然后直接将 `Trap` 上下文压入内核栈栈顶。当 `Trap` 处理完毕返回用户态的时候，将 `Trap` 上下文中的内容恢复到寄存器上，最后将保存着应用用户栈顶的 `sscratch` 与 `sp` 进行交换，也就从内核栈切换回了用户栈。在这个过程中， `sscratch` 起到了非常关键的作用，它使得我们可以在不破坏任何通用寄存器的情况下，完成用户栈与内核栈的切换，以及位于内核栈顶的 `Trap` 上下文的保存与恢复。
>
> 然而，一旦使能了分页机制，一切就并没有这么简单了，我们必须在这个过程中同时完成地址空间的切换。具体来说，当 `__alltraps` 保存 Trap 上下文的时候，我们必须通过修改 `satp` 从应用地址空间切换到内核地址空间，因为 `trap handler` 只有在内核地址空间中才能访问；同理，在 `__restore` 恢复 Trap 上下文的时候，我们也必须从内核地址空间切换回应用地址空间，因为应用的代码和数据只能在它自己的地址空间中才能访问，应用是看不到内核地址空间的。这样就要求地址空间的切换不能影响指令的连续执行，即要求应用和内核地址空间在切换地址空间指令附近是平滑的。
>
> 我们为何将应用的 Trap 上下文放到应用地址空间的次高页面而不是内核地址空间中的内核栈中呢？原因在于，在保存 Trap 上下文到内核栈中之前，我们必须完成两项工作：1）必须先切换到内核地址空间，这就需要将内核地址空间的 token 写入 satp 寄存器；2）之后还需要保存应用的内核栈栈顶的位置，这样才能以它为基址保存 Trap 上下文。这两步需要用寄存器作为临时周转，然而我们无法在不破坏任何一个通用寄存器的情况下做到这一点。因为事实上我们需要用到内核的两条信息：内核地址空间的 token ，以及应用的内核栈栈顶的位置，RISC-V却只提供一个 `sscratch` 寄存器可用来进行周转。所以，我们不得不将 Trap 上下文保存在应用地址空间的一个虚拟页面中，而不是切换到内核地址空间去保存。

总结一下就是要是想要在内核栈中保存trap上下文，则需要先切换到内核的页表，需要一个寄存器去存内核地址空间`satp`的值，在进行地址空间切换后，才能将trap上下文压入内核栈中，之前是通过`sscratch` 来保存应用的内核栈的，现在由于开启了分页机制，导致需要

多一个寄存器来实现。但是无法在不破坏任何一个通用寄存器的情况来做到这一点，因此将Trap上下文保存到应用地址空间的一页中。

这里我有个问题，刚产生trap时，此时已经进入了S模式，需要在S模式下执行__alltraps函数，但此时执行代码和访问数据还是在应用程序所处的用户态虚拟地址空间中，而不是我们通常理解的内核虚拟地址空间。从内核态的trap返回时，会去执行__restore函数，此时也是在S态，但是需要去应用地址空间拿到trap上下文来恢复。

rCore的解释是：无论是内核还是应用的地址空间，跳板的虚拟页均位于同样位置，且它们也将会映射到同一个实际存放这段汇编代码的物理页帧。也就是说，在执行 __alltraps 或 __restore 函数进行地址空间切换的时候，应用的用户态虚拟地址空间和操作系统内核的内核态虚拟地址空间对切换地址空间的指令所在页的映射方式均是相同的，这就说明了这段切换地址空间的指令控制流仍是可以连续执行的。

简单点说就是，所用应用程序的跳板页都映射到同一物理内存页上，同时内核的也需要映射一个跳板页到这同一个物理页上，这样无论内核态去访问还是用户态去访问，访问的都是同一页物理内存，这一页物理内存上放的就是之前的__alltraps 函数和__restore ，这样就能理解了。

4.2 trap修改

根据上面的分析，__alltraps 函数需要实现在保存完毕trap上下文后从应用地址空间切换到内核地址空间，而__restore函数需要先切换到应用地址空间去恢复trap上下文，然后返回用户态。

先对trap上下文进行拓展：多了三项

```c
/*S模式的trap上下文*/
typedef struct TrapContext {
    reg_t x0;
    reg_t ra;
    reg_t sp;
    reg_t gp;
    reg_t tp;
    reg_t t0;
    reg_t t1;
    reg_t t2;
    reg_t s0;
    reg_t s1;
    reg_t a0;
    reg_t a1;
    reg_t a2;
    reg_t a3;
    reg_t a4;
    reg_t a5;
    reg_t a6;
    reg_t a7;
    reg_t s2;
    reg_t s3;
    reg_t s4;
    reg_t s5;
    reg_t s6;
    reg_t s7;
    reg_t s8;
    reg_t s9;
    reg_t s10;
    reg_t s11;
    reg_t t3;
    reg_t t4;
    reg_t t5;
    reg_t t6;
    /* S模式下的寄存器 */
    reg_t sstatus;
    reg_t sepc;
    /* mmu 相关 */
    reg_t kernel_satp;      //内核地址空间的satp值
    reg_t kernel_sp;        //内核栈栈顶的虚拟地址
    reg_t trap_handler;     //内核中 trap handler 入口的虚拟地址
}TrapContext;
```

在多出的三个字段中：

- `kernel_satp` 表示内核地址空间的 token ， 即内核页表的起始物理地址；
- `kernel_sp` 表示当前应用在内核地址空间中的内核栈栈顶的虚拟地址；
- `trap_handler` 表示内核中`trap handler`入口点的虚拟地址。

然后是切换地址空间：修改kerneltrap.S，在kerneltrap.S文件的开头我们声明此代码段为一个名为trampoline的节，这样trampoline就指向了此段代码的起始地址，用于后面链接和映射

plaintext

```
.section .text.trampoline
.globl __alltraps
.align 3
__alltraps:
    # 从sscratch获取S模式下的SP，把U模式下的SP保存到sscratch寄存器
中
    csrrw sp, sscratch, sp
    # now sp->kernel stack, sscratch->user stack
    # allocate a TrapContext on kernel stack
    # save general-purpose registers
    sd x1, 1*8(sp)
    # skip sp(x2), we will save it later
    sd x3, 3*8(sp)
    # skip tp(x4), application does not use it
    # save x5~x31
    sd x4, 4*8(sp)
    sd x5, 5*8(sp)
    sd x6, 6*8(sp)
    sd x7, 7*8(sp)
    sd x8, 8*8(sp)
    sd x9, 9*8(sp)
    sd x10,10*8(sp)
    sd x11, 11*8(sp)
    sd x12, 12*8(sp)
    sd x13, 13*8(sp)
    sd x14, 14*8(sp)
    sd x15, 15*8(sp)
    sd x16, 16*8(sp)
    sd x17, 17*8(sp)
    sd x18, 18*8(sp)
    sd x19, 19*8(sp)
    sd x20, 20*8(sp)
    sd x21, 21*8(sp)
    sd x22, 22*8(sp)
    sd x23, 23*8(sp)
    sd x24, 24*8(sp)
    sd x25, 25*8(sp)
    sd x26, 26*8(sp)
    sd x27, 27*8(sp)
    sd x28, 28*8(sp)
    sd x29, 29*8(sp)
    sd x30, 30*8(sp)
    sd x31, 31*8(sp)

    # we can use t0/t1/t2 freely, because they were saved on kernel
stack
    csrr t0, sstatus
    csrr t1, sepc
    sd t0, 32*8(sp)
    sd t1, 33*8(sp)
    # read user stack from sscratch and save it on the kernel stack
    csrr t2, sscratch
    sd t2, 2*8(sp)
    # load kernel_satp into t0
    ld t0, 34*8(sp)
    # load trap_handler into t1
    ld t1, 36*8(sp)
    # move to kernel_sp
```

```
    ld sp, 35*8(sp)
    # switch to kernel space
    csrw satp, t0
    sfence.vma
    # jump to trap_handler
    jr t1


.globl __restore
.align 3
__restore:
    # a0: *TrapContext in user space(Constant); a1: user space
token
    # switch to user space
    csrw satp, a1
    sfence.vma
    csrw sscratch, a0
    mv sp, a0
    # now sp->kernel stack(after allocated), sscratch->user stack
    # restore sstatus/sepc
    ld t0, 32*8(sp)
    ld t1, 33*8(sp)
    csrw sstatus, t0
    csrw sepc, t1
    # restore general-purpuse registers except sp/tp
    ld x1, 1*8(sp)
    ld x3, 3*8(sp)
    ld x4, 4*8(sp)
    ld x5, 5*8(sp)
    ld x6, 6*8(sp)
    ld x7, 7*8(sp)
    ld x8, 8*8(sp)
    ld x9, 9*8(sp)
    ld x10,10*8(sp)
    ld x11, 11*8(sp)
    ld x12, 12*8(sp)
    ld x13, 13*8(sp)
    ld x14, 14*8(sp)
    ld x15, 15*8(sp)
    ld x16, 16*8(sp)
    ld x17, 17*8(sp)
    ld x18, 18*8(sp)
    ld x19, 19*8(sp)
    ld x20, 20*8(sp)
    ld x21, 21*8(sp)
    ld x22, 22*8(sp)
    ld x23, 23*8(sp)
    ld x24, 24*8(sp)
    ld x25, 25*8(sp)
    ld x26, 26*8(sp)
    ld x27, 27*8(sp)
    ld x28, 28*8(sp)
    ld x29, 29*8(sp)
    ld x30, 30*8(sp)
    ld x31, 31*8(sp)

    # back to user stack
    ld sp, 2*8(sp)

    sret
```

- 当应用 Trap 进入内核的时候，硬件会设置一些 CSR 并在 S 特权级下跳转到 `__alltraps` 保存 Trap 上下文。此时 sp 寄存器仍指向用户栈，但 `sscratch` 则被设置为指向应用地址空间中存放 Trap 上下文的位置（实际在次高页面）。随后，就像之前一样，我们 `csrrw` 交换 sp 和 `sscratch`，并基于指向 Trap 上下文位置的 sp 开始保存通用寄存器和一些 CSR，这个过程在第 28 行结束。到这里，我们就全程在应用地址空间中完成了保存 Trap 上下文的工作。

- 接下来该考虑切换到内核地址空间并跳转到 trap handler 了。

    - 第 53 行将内核地址空间的 `token` 载入到 `t0` 寄存器中；
    - 第 55 行将 `trap handler` 入口点的虚拟地址载入到 `t1` 寄存器中；
    - 第 57 行直接将 `sp` 修改为应用内核栈顶的地址；
    - 第 59~60 行将 satp 修改为内核地址空间的 token 并使用 `sfence.vma` 刷新快表，这就切换到了内核地址空间；
    - 第 62 行 最后通过 `jr` 指令跳转到 `t1` 寄存器所保存的`trap handler` 入口点的地址。
- 当内核将 Trap 处理完毕准备返回用户态的时候会 *调用* `__restore` （符合RISC-V函数调用规范），它有两个参数：第一个是 Trap 上下文在应用地址空间中的位置，这个对于所有的应用来说都是相同的，在 a0 寄存器中传递；第二个则是即将回到的应用的地址空间的 token，在 a1 寄存器中传递。

    - 第 70~71 行先切换回应用地址空间（注：Trap 上下文是保存在应用地址空间中）；
    - 第 72 行将传入的 Trap 上下文位置保存在 `sscratch` 寄存器中，这样 `__alltraps` 中才能基于它将 Trap 上下文保存到正确的位置；
    - 第 73 行将 sp 修改为 Trap 上下文的位置，后面基于它恢复各通用寄存器和 CSR；
    - 第 115 行最后通过 `sret` 指令返回用户态。

4.3 链接文件修改

将 `kerneltrap.S` 中的整段汇编代码放置在 `.text.trampoline` 段，并在调整内存布局的时候将它对齐到代码段的一个页面中：

plaintext

```
skernel = .;   /* 定义内核起始内存
地址 */
.text : {
        *(.text.entry)
        . = ALIGN(4K);
        trampoline = .;
        *(.text.trampoline);
        . = ALIGN(4K);
         _strampoline = .;
         *(.text .text.*)
         . = ALIGN(4K);
         PROVIDE(etext = .);
} >ram
```

这样，这段汇编代码放在一个物理页帧中，且 `__alltraps` 恰好位于这个物理页帧的开头，其物理地址被外部符号 `strampoline` 标记。在开启分页模式之后，内核和应用代码都只能看到各自的虚拟地址空间，而在它们的视角中，这段汇编代码都被放在它们各自地址空间的最高虚拟页面上，由于这段汇编代码在执行的时候涉及到地址空间切换，故而被称为跳板页面。

4.4 映射跳板页

在kvmmake中将跳板页进行映射，大小为一页，属性为可读可执行

```
extern char etext[];
extern char trampoline[];

PageTable kvmmake(void)
{
    PageTable pt;
    PhysPageNum root_ppn =  StackFrameAllocator_alloc(&FrameAllocatorImpl);
    pt.root_ppn = root_ppn;
    printk("root_ppn:%p\n",phys_addr_from_phys_page_num(root_ppn));

    printk("etext:%p\n",(u64)etext);
    // map kernel text executable and read-only.
    PageTable_map(&pt,virt_addr_from_size_t(KERNBASE),phys_addr_from_size_t(KERNBASE), \
                (u64)etext-KERNBASE , PTE_R | PTE_X ) ;
    printk("finish kernel text map!\n");
    // map kernel data and the physical RAM we'll make use of.
    PageTable_map(&pt,virt_addr_from_size_t((u64)etext),phys_addr_from_size_t((u64)etext ), \
                PHYSTOP - (u64)etext , PTE_R | PTE_W ) ;
    printk("finish kernel data and physical RAM map!\n");
    // map the trampoline for trap entry/exit to the highest virtual address in the kernel.
    PageTable_map(&pt, virt_addr_from_size_t(TRAMPOLINE), phys_addr_from_size_t((u64)trampoline), \
                PAGE_SIZE, PTE_R | PTE_X );
    printk("finish TRAMPOLINE Page map!\n");

    return pt;
}
```

5. 为timer os映射应用程序内核栈

无论是rCore还是xv6的内存方式都是一样的，我们内核的应用程序内核栈和xv6一样映射，首先在`address`中新建一个`kalloc`函数：此函数直接返回分配的物理页帧号。我们现在`address`的部分是十分臃肿的，是因为我最开始为了模仿rCore把那些`PhysPageNum PageTable VirtPageNum VirtPageNum PhysAddr`之类的定义全部搞成了结构体类型的，所以操作这些数据结构的方式全部使用了函数。其实应该定义成一个u64的类型就可以了，然后定义一些宏来操作。后面再来优化吧。

c

```
PhysPageNum kalloc(void)
{
    PhysPageNum frame =
StackFrameAllocator_alloc(&FrameAllocatorImpl);
    return frame;
}
```

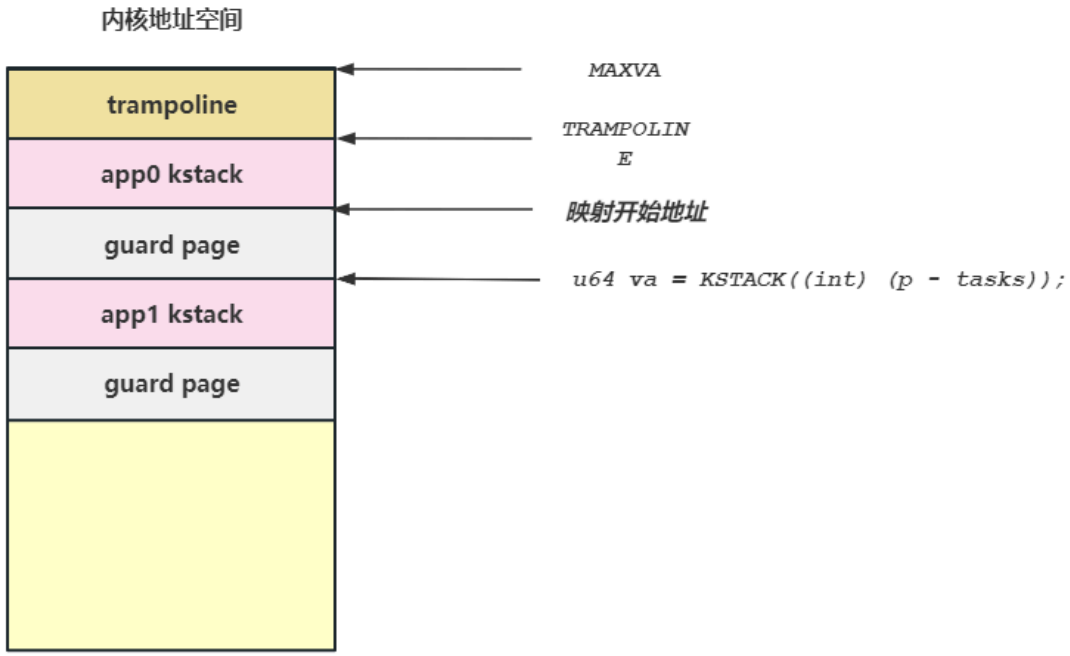
在`task.c`中新建一个函数，来进行应用程序内核栈的映射：就是把xv6中的那个函数抄了过来，然后修改了一下

c

```
/* 为每个应用程序映射内核栈,内核空间以及进行了映射 */
void proc_mapstacks(PageTable* kpgtbl)
{
  struct TaskControlBlock *p;

  for(p = tasks; p < &tasks[MAX_TASKS]; p++) {
    char *pa = (char*)phys_addr_from_phys_page_num(kalloc()).value;
    if(pa == 0)
      panic("kalloc");
    u64 va = KSTACK((int) (p - tasks));
    PageTable_map(kpgtbl, virt_addr_from_size_t(va + PAGE_SIZE),
phys_addr_from_size_t((u64)pa), \
                  PAGE_SIZE, PTE_R | PTE_W);
  }
}
```

这里需要说明一下`u64 va = KSTACK((int) (p - tasks));`得到的`va`实际上是`guard page`的最低地址，因此实际开始映射的地址是：`va + PAGE_SIZE`，如下图：



然后在kvmmake中来调用此函数：

```
PageTable kvmmake(void)
{
    PageTable pt;
    PhysPageNum root_ppn =  StackFrameAllocator_alloc(&FrameAllocatorImpl);
    pt.root_ppn = root_ppn;
    printk("root_ppn:%p\n",phys_addr_from_phys_page_num(root_ppn));

    printk("etext:%p\n",(u64)etext);
    // map kernel text executable and read-only.
    PageTable_map(&pt,virt_addr_from_size_t(KERNBASE),phys_addr_from_size_t(KERNBASE), \
                (u64)etext-KERNBASE , PTE_R | PTE_X ) ;
    printk("finish kernel text map!\n");
    // map kernel data and the physical RAM we'll make use of.
    PageTable_map(&pt,virt_addr_from_size_t((u64)etext),phys_addr_from_size_t((u64)etext ), \
                PHYSTOP - (u64)etext , PTE_R | PTE_W ) ;
    printk("finish kernel data and physical RAM map!\n");
    // map the trampoline for trap entry/exit to the highest virtual address in the kernel.
    PageTable_map(&pt, virt_addr_from_size_t(TRAMPOLINE), phys_addr_from_size_t((u64)trampoline), \
                PAGE_SIZE, PTE_R | PTE_X );
    printk("finish TRAMPOLINE Page map!\n");

    //allocate and map a kernel stack for each process.
    proc_mapstacks(&pt);
    printk("finish kernel stack map!\n");

    return pt;
}
```

至此我们完成了内核的映射、跳板页的映射、应用程序内核栈的映射，下一步就是读取应用程序的elf文件，完成对应用程序的各逻辑段的映射、跳板页的映射、用户栈的映射、trap上下文的映射，然后在task_create函数中设置每个任务的trap上下文，包括应用程序入口地址、用户栈指针、内核空间的satp值、内核栈顶的虚拟地址、内核中trap handler入口的虚拟地址。然后就可以实现开启虚拟地址的多任务了。

参考链接

- Chapter 3: Page Tables - 知乎 (zhihu.com)
- xv6: a simple, Unix-like teaching operating system (mit.edu)

- 基于地址空间的分时多任务 - rCore-Tutorial-Book-v3 3.6.0-alpha.1 文档 (rcore-os.cn)

- 3.2 内核地址空间 · 6.S081 All-In-One (dgs.zone)

相关推荐