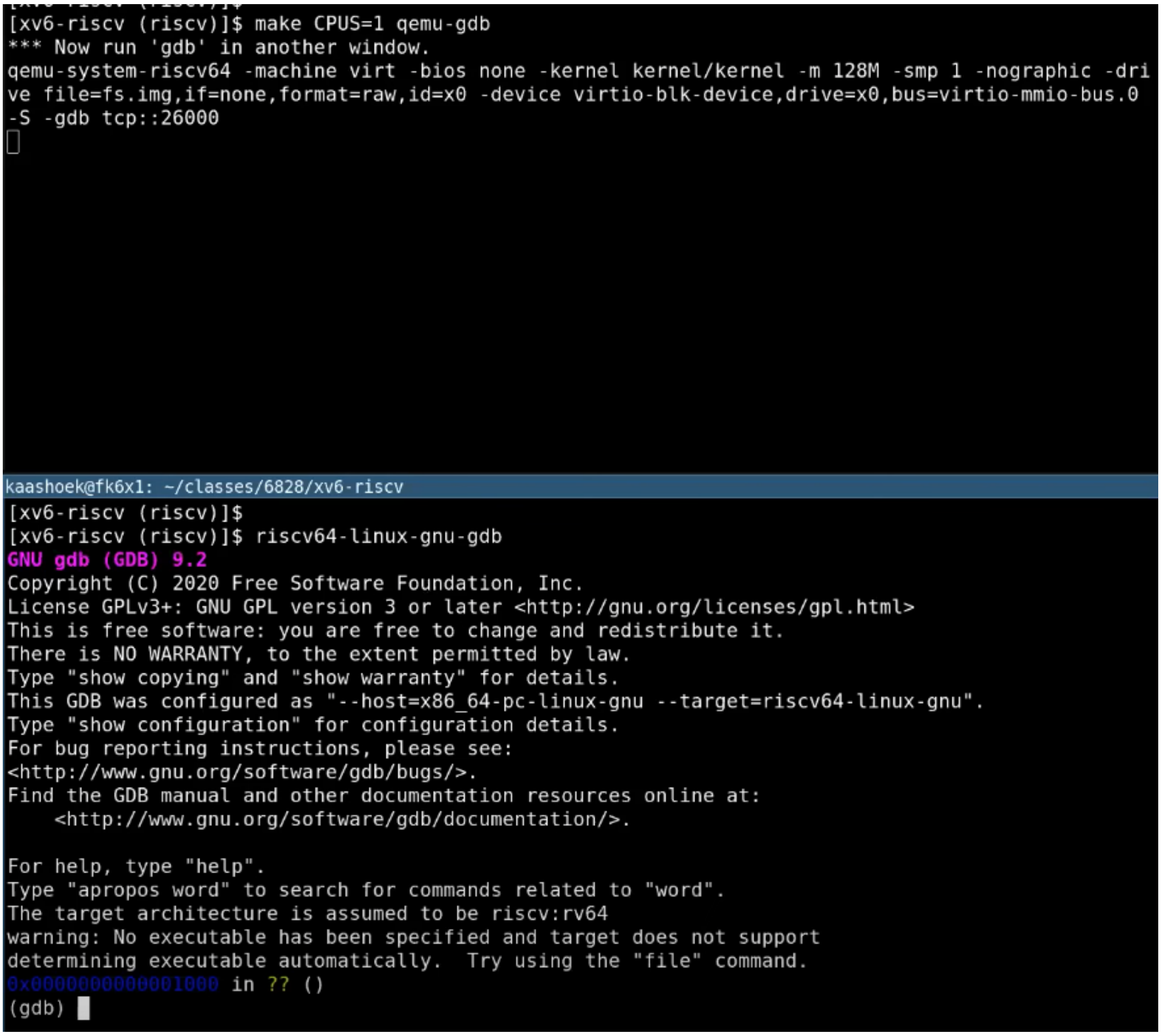
4.6 kvminit 函数

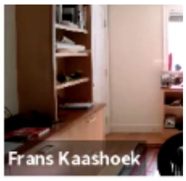
接下来，让我们看一看代码，我认为很多东西都会因此变得更加清晰。

首先，我们来做一个的常规操作，启动我们的XV6，这里QEMU实现了主板，同时我们打开gdb。



上一次我们看了boot的流程，我们跟到了main函数。main函数中调用的一个函数是kvminit（3.9），这个函数会设置好kernel的地址空间。kvminit的代码如下图所示：

```
void
kvminit()
{
  kernel_pagetable = (pagetable_t) kalloc();
  memset(kernel_pagetable, 0, PGSIZE);

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

  vmprint(kernel_pagetable);

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

  // CLINT
  kvmmap(CLINT, CLINT, 0x10000, PTE_R | PTE_W);

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

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

  // map kernel data and the physical RAM we'll make use of.
  kvmmap((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(TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);

  vmprint(kernel_pagetable);
}
```



我们在前一部分看了kernel的地址空间长成什么样，这里我们来看一下代码是如何将它设置好的。首先在kvminit中设置一个断点，之后运行代码到断点位置。在gdb中执行layout split，可以看到（从上面的代码也可以看出）函数的第一步是为最高一级page directory分配物理page（注，调用kalloc就是分配物理page）。下一行将这段内存初始化为0。

```
kaashoek@fk6x1: ~/classes/6828/xv6-riscv
kernel/vm.c
>26         kernel_pagetable = (pagetable_t) kalloc();
 27         memset(kernel_pagetable, 0, PGSIZE);
 28
 29         // uart registers
 30         kvmmap(UART0, UART0, PGSIZE, PTE_R | PTE_W);
 31
 32         vmprint(kernel_pagetable);
 33

>0x80001940 <kvminit+12> auipc   ra,0xfffff
 0x80001944 <kvminit+16> jalr    462(ra)
 0x80001948 <kvminit+20> auipc   s1,0x7
 0x8000194c <kvminit+24> addi    s1,s1,1736
 0x80001950 <kvminit+28> sd      a0,0(s1)
 0x80001952 <kvminit+30> lui     a2,0x1
 0x80001954 <kvminit+32> li      a1,0
 0x80001956 <kvminit+34> auipc   ra,0xfffff
 0x8000195a <kvminit+38> jalr    932(ra)
remote Thread 1.1 In: kvminit                        L26   PC: 0x80001940
```

之后，通过kvmmap函数，将每一个I/O设备映射到内核。例如，下图中高亮的行将UART0映射到内核的地址空间。

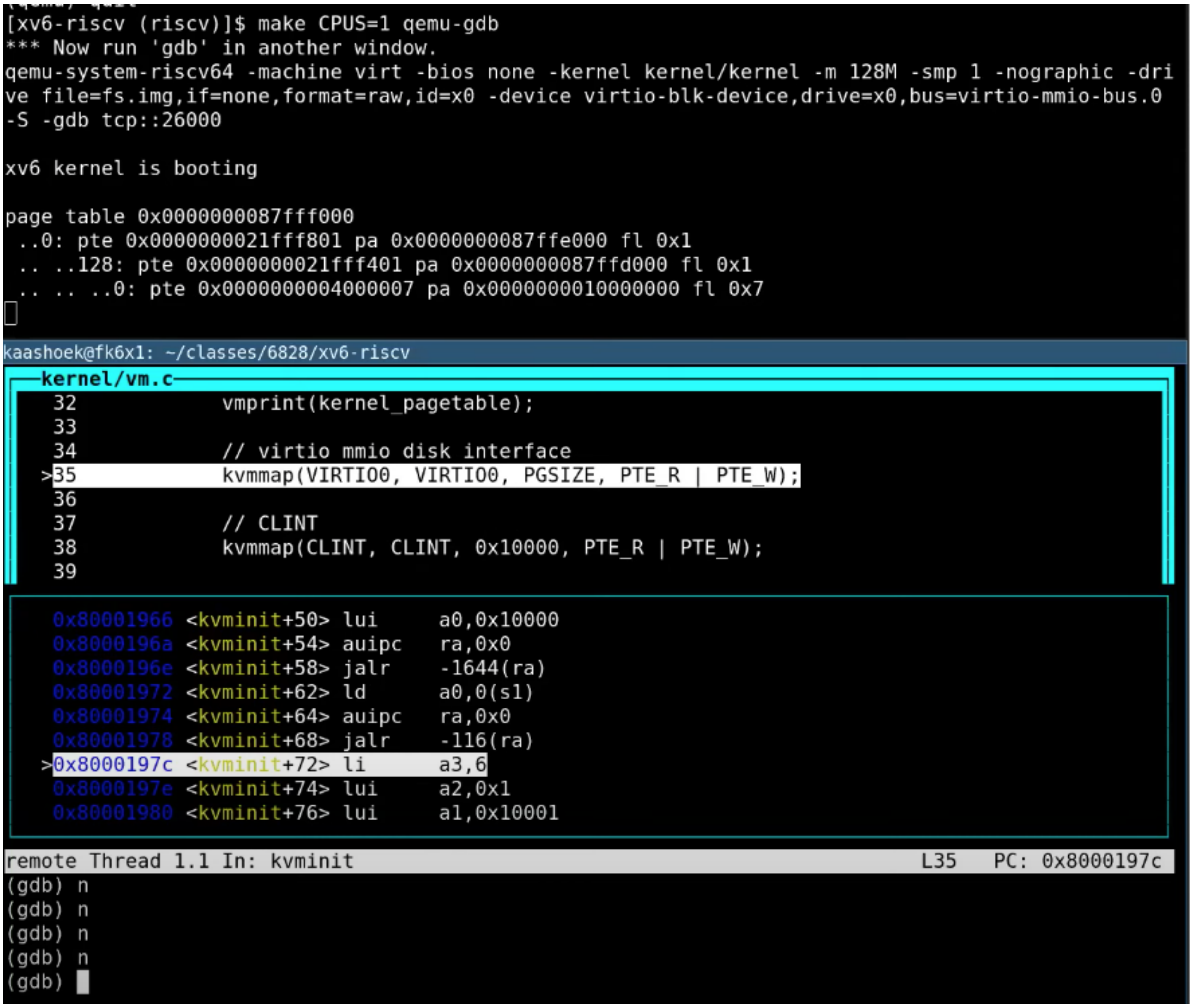
```
kernel/vm.c
 26          kernel_pagetable = (pagetable_t) kalloc();
 27          memset(kernel_pagetable, 0, PGSIZE);
 28
 29          // uart registers
>30          kvmmap(UART0, UART0, PGSIZE, PTE_R | PTE_W);
 31
 32          vmprint(kernel_pagetable);
 33

 0x80001954 <kvminit+32> li      a1,0
 0x80001956 <kvminit+34> auipc   ra,0xfffff
 0x8000195a <kvminit+38> jalr    932(ra)
>0x8000195e <kvminit+42> li      a3,6
 0x80001960 <kvminit+44> lui     a2,0x1
 0x80001962 <kvminit+46> lui     a1,0x10000
 0x80001966 <kvminit+50> lui     a0,0x10000
 0x8000196a <kvminit+54> auipc   ra,0x0
 0x8000196e <kvminit+58> jalr    -1644(ra)

remote Thread 1.1 In: kvminit                          L30   PC: 0x8000195e
(gdb) n
(gdb) n
(gdb)
```

我们可以查看一个文件叫做memlayout.h，它将4.5中的文档翻译成了一堆常量。在这个文件里面可以看到，UART0对应了地址0x10000000（注，4.5中的文档是真正SiFive RISC-V的文档，而下图是QEMU的地址，所以4.5中的文档地址与这里的不符）。

```
// Physical memory layout

// qemu -machine virt is set up like this,
// based on qemu's hw/riscv/virt.c:
//
// 00001000 -- boot ROM, provided by qemu
// 02000000 -- CLINT
// 0C000000 -- PLIC
// 10000000 -- uart0
// 10001000 -- virtio disk
// 80000000 -- boot ROM jumps here in machine mode
//             -kernel loads the kernel here
// unused RAM after 80000000.

// the kernel uses physical memory thus:
// 80000000 -- entry.S, then kernel text and data
// end -- start of kernel page allocation area
// PHYSTOP -- end RAM used by the kernel

// qemu puts UART registers here in physical memory.
#define UART0 0x10000000L
#define UART0_IRQ 10
```

所以，通过kvmmap可以将物理地址映射到相同的虚拟地址（注，因为kvmmap的前两个参数一致）。

在page table实验中，第一个练习是实现vmprint，这个函数会打印当前的kernel page table。我们现在跳过这个函数，看一下执行完第一个kvmmap时的kernel page table。



我们来看一下这里的输出。第一行是最高一级page directory的地址，这就是存在SATP或者将会存在SATP中的地址。第二行可以看到最高一级page directory只有一条PTE序号为0，它包含了中间级page directory的物理地址。第三行可以看到中间级的page directory只有一条PTE序号为128，它指向了最低级page directory的物理地址。第四行可以看到最低级的page directory包含了PTE指向物理地址。你们可以看到最低一级 page directory中PTE的物理地址就是0x10000000，对应了UART0。

前面是物理地址，我们可以从虚拟地址的角度来验证这里符合预期。我们将地址0x10000000向右移位12bit，这样可以得到虚拟地址的高27bit（index部分）。之后我们再对这部分右移位9bit，并打印成10进制数，可以得到128，这就是中间级page directory中PTE的序号。这与之前（4.4）介绍的内容是符合的。

```
(gdb) p /x (0x10000000 >> 12)
$1 = 0x10000
(gdb) p /x (0x10000 >> 9)
$2 = 0x80
(gdb) p 0x80
$3 = 128
```

从标志位来看（fl部分），最低一级page directory中的PTE有读写标志位，并且Valid标志位也设置了（4.3底部有标志位的介绍）。

内核会持续的按照这种方式，调用kvmmap来设置地址空间。之后会对VIRTIO0、CLINT、PLIC、kernel text、kernel data、最后是TRAMPOLINE进行地址映射。最后我们还会调用vmprint打印完整的kernel page directory，可以看出已经设置了很多PTE。

```
xv6 kernel is booting

page table 0x0000000087fff000
 ..0: pte 0x0000000021fff801 pa 0x0000000087ffe000 fl 0x1
 .. ..128: pte 0x0000000021fff401 pa 0x0000000087ffd000 fl 0x1
 .. .. ..0: pte 0x0000000004000007 pa 0x0000000010000000 fl 0x7
page table 0x0000000087fff000
 ..0: pte 0x0000000021fff801 pa 0x0000000087ffe000 fl 0x1
 .. ..16: pte 0x0000000021fff001 pa 0x0000000087ffc000 fl 0x1
 .. .. ..0: pte 0x0000000000800007 pa 0x0000000002000000 fl 0x7
 .. .. ..15: pte 0x0000000000803c07 pa 0x000000000200f000 fl 0x7
 .. ..96: pte 0x0000000021ffec01 pa 0x0000000087ffb000 fl 0x1
 .. .. ..0: pte 0x0000000003000007 pa 0x000000000c000000 fl 0x7
 .. .. ..511: pte 0x000000000307fc07 pa 0x000000000c1ff000 fl 0x7
 .. ..97: pte 0x0000000021ffe801 pa 0x0000000087ffa000 fl 0x1
 .. .. ..0: pte 0x0000000003080007 pa 0x000000000c200000 fl 0x7
 .. .. ..511: pte 0x00000000030ffc07 pa 0x000000000c3ff000 fl 0x7
kaashoek@fk6x1: ~/classes/6828/xv6-riscv
-kernel/main.c-
   18              printf("\n");
   19              kinit();         // physical page allocator
   20              kvminit();       // create kernel page table
  >21              kvminithart();   // turn on paging
   22              procinit();      // process table
   23              trapinit();      // trap vectors
   24              trapinithart();  // install kernel trap vector
   25              plicinit();      // set up interrupt controller

    0x80000f4a <main+162>    jalr    -1140(ra)
    0x80000f4e <main+166>    auipc   ra,0x1
    0x80000f52 <main+170>    jalr    -1562(ra)
   >0x80000f56 <main+174>    auipc   ra,0x0
    0x80000f5a <main+178>    jalr    432(ra)
    0x80000f5e <main+182>    auipc   ra,0x1
    0x80000f62 <main+186>    jalr    -1272(ra)
    0x80000f66 <main+190>    auipc   ra,0x2
    0x80000f6a <main+194>    jalr    -2038(ra)
remote Thread 1.1 In: main                                         L21   PC: 0x80000f56
$3 = 128
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) n
main () at kernel/main.c:21
(gdb)
```

这里就不过细节了，但是这些PTE构成了我们在4.5中看到的地址空间对应关系。

（下面问答来自课程结束部分，因为内容相关就移到这里。）

> 学生：下面这两行内存不会越界吗？

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

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

> Frans：不会。这里KERNBASE是0x80000000，这是内存开始的地址。kvmmap的第三个参数是size，etext是kernel text的最后一个地址，etext - KERNBASE会返回kernel text的字节数，我不确定这块有多大，大概是60-90个page，这部分是kernel的text部分。PHYSTOP是物理内存的最大位置，PHYSTOP-text是kernel的data部分。会有足够的DRAM来完成这里的映射。