第24讲 进程的实现

2024.04.16

之前花了很长一段时间讲虚拟化、讲进程的时候，其实我们讲的是应用视角的操作系统，就是我们一方面我们说程序是状态机，然后操作系统是状态机的管理者。然后操系统就为这些状态机提供系统调用这样的一个API，然后所有的进程都可以通过系统调用 API 去访问操作系统内部的对象。

那么今天我就会用相对比较简略的讲一讲我们的操作系统上面的进程到底是怎么实现出来的。

1. 进程的实现

1. what虚拟内存

Thread-OS/Lab2

实现了在多个线程之间切换

```
Context *on_interrupt(Event ev, Context *ctx) {
  if (!current) {
    current = &tasks[0];  // First trap
  } else {
    current->context = ctx;
    current = current->next;
  }
  return current->context;
}
```

内核线程 v.s. 进程，还差了什么？

还差了一个VR眼镜！

一个 VR 眼镜！

我们看到的世界未必真实

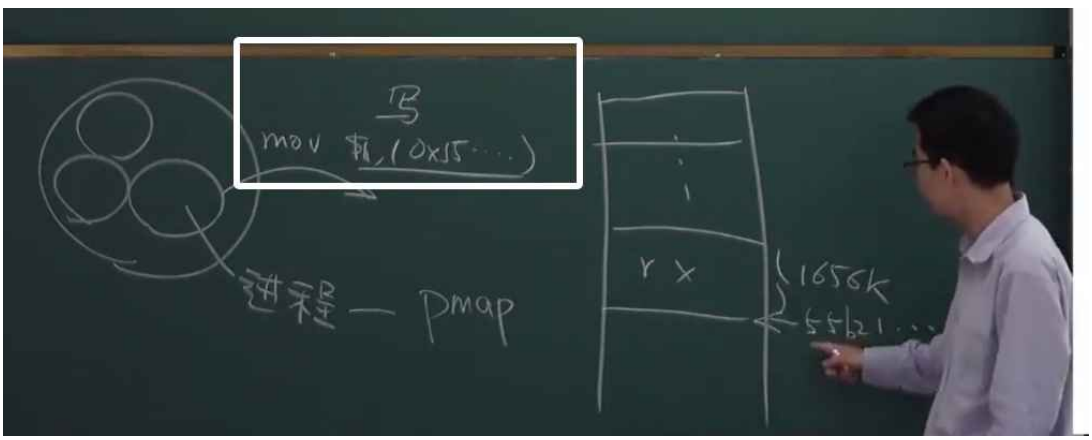
- (进程也是)

那你接下来就想，我们难道没有某种程度上戴着 VR 眼镜吗？就比如说我现在看到同学们在教室里，然后同学们看到我在教室里，我后面有一个黑板，你看到的真的是存在的吗？没有人知道是存在还是不存在。比如说你是一个大脑然后放在一个在容器里，然后给你那样的电信号做刺激，还是它是真的存在的？没有人知道。

某种程度上来说，如果我们能够给这个线程带上一个 VR 眼镜，那这个线程他看到的内存就不再是我们的整个所有的物理内存了。

在这里面为什么叫线程，为什么叫 THREAD OS？就是因为我们所有的线程都是共享内存的，你可以访问任何一个全局变量。

但是如果我们能够给这个线程戴上一个眼镜的话，他看到不再是一个真实的物理内存，而是一个虚拟的内存的话，我们再通过一个地址翻译的部件，能够把我们每一次内存访问的值改成另外一个地址。我们都知道，Linux 我们的进程是由一段一段连续、带权限访问的内存组成的，那么我们就可以通过戴上 VR 眼镜这样的一个手段来创建出这样的地址空间。



这个写就需要经过地址翻译的函数。你就知道你现在是戴上了一个 VR 眼镜。这个 VR 眼睛就是一个地址翻译部件，它会把所有的虚拟的内存地址都翻译成物理内存地址。这里的所有是指包括你在取指令

的时候，你想你的程序要执行这条指令从哪来？这条指令是从 PC 指针的寄存器那个地方取出来的，然后这个 PC 也是在这个地址空间里的，所以对于取指令来说，它也要经过这个地址翻译。

进程的地址空间

pmap: 进程拥有 $[0, 2^{64})$ 的地址空间

- 但这个地址空间是“虚拟”的
- 我们的物理内存也就是从 0 开始的几个 GB

地址翻译

- 计算机系统中的一个数据结构 $f(x)$
  - 把虚拟内存地址翻译成物理内存地址
  - 由寄存器 (例如 x86 的 CR3) 和内存共同描述
- 地址翻译 (VR 眼镜) 是强制的
  - 用户程序 mov CR3 将导致 #GP 异常
  - 越权访问/非法访问将导致 #PF 异常 (Page Fault)
  - 你看到的世界完全是操作系统的安排 (你害怕了吗)

CR3：指向页表。

总而言之，进程看到的世界是操作系统完全安排好的。进程也没有任何反抗的权力。

2. why虚拟内存

虚假的地址空间 (1)

“映射了，但没有真的映射”

- VR 眼镜：眼睛还没看到的地方可以先不显示

还记得 mmap 分配空间/映射磁盘的例子吗？

- 操作系统只要用一个数据结构记录哪段内存是什么
  - 因此进程可以使用超过物理内存的虚拟内存 (swap)
- Page Fault 的时候再分配
  - 文件 → 映射文件内容后返回
  - 匿名数据 → 分配物理页面后返回
  - 非法访问 → SIGSEGV
- 用什么样的数据结构实现？

```
mmap.mmap(fd, prot=mmap.PROT_READ, length=128 << 30)
```

虚假的地址空间 (2)

“映射了很多次，但又只有一个”

- 比 VR 眼镜高级的功能：多个位置共享同一块屏幕

我们相信：厉害的操作系统，同一份代码仅有一个副本

- 如何间接证实这一点？
- 如何直接确认这一点？
  - AskGPT: How to print the corresponding physical address of a given virtual address in Linux user space in C?

虚假的地址空间 (3)

“复制了，但没有完全复制”

- 即便使用了很多内存，fork() 依然可以在瞬间完成
  - 复制 GB 级内存也需要数百 ms
  - 怎么做到的？

复制，很有可能白复制

- fork-execve 非常常见
- 刚复制完 1GB 内存，就销毁了
  - 操作系统：我不干这种事
  - 除了万不得已，能不复制就不复制
  - “写时复制” (copy-on-write)

2. 处理器调度原理

Take away message

进程在操作系统中的实现是简单又复杂的。从简单来说，进程就是带有独立地址空间的线程；通过硬件提供的分页机制，就能给线程戴上 “VR 眼镜”，使得看到的内存并不是真实的内存。同时，进程也是复杂的：我们可以借助虚拟内存实现 demand paging、copy-on-write fork 等有趣的机制；而如何调度系统中的进程，在现代多处理器时代也显得愈加复杂。