|  |  |  |  |
| --- | --- | --- | --- |
| **评语:** | **课中检查完成的题号及题数：** |  | |
|  | **课后完成的题号与题数：** | **完成两个：1-2、3** | |
|  |  |  | |
| **成绩:** |  | **指导教师:** | **谷建华** |

|  |
| --- |
| **实验报告九** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **实验名称：** | | **完成缺页中断处理函数** | | | | **日期：** | **2019.12.23** |
| **班级：** | **10011704** | | **学号：** | **2017303030** | **姓名：** | | **刘丰毓** |

1. **实验目的**

完成缺页中断处理函数。

1. **具体实现**

什么是内存管理？对计算机的一般用户来说，内存的是一条条价值不菲的内存条，他们的内存管理是对内存条的管理，这关系到他们的钱包；对于应用程序开发者来说，得益于进程概念的广泛普及，内存管理是对自己所写程序的进程空间的管理，他们不必关心其他进程的存在；而这种抽象一定离不开的则是操作系统，对于一个现代操作系统来说，内存管理四个字则意味着，充分利用有限的内存资源，统筹规划，做好精准的管理与合理的抽象，为进程提供一个良好的“生存环境”。

操作系统的任何操作都离不开硬件的支持，i386为我们提供了很多硬件层面的内存管理的机制，得益于这些机制，进行某些抽象才成为可能。这些机制比如：分段、分页、中断，这三者、尤其是后面两个，让操作系统拥有进程的概念成为可能。

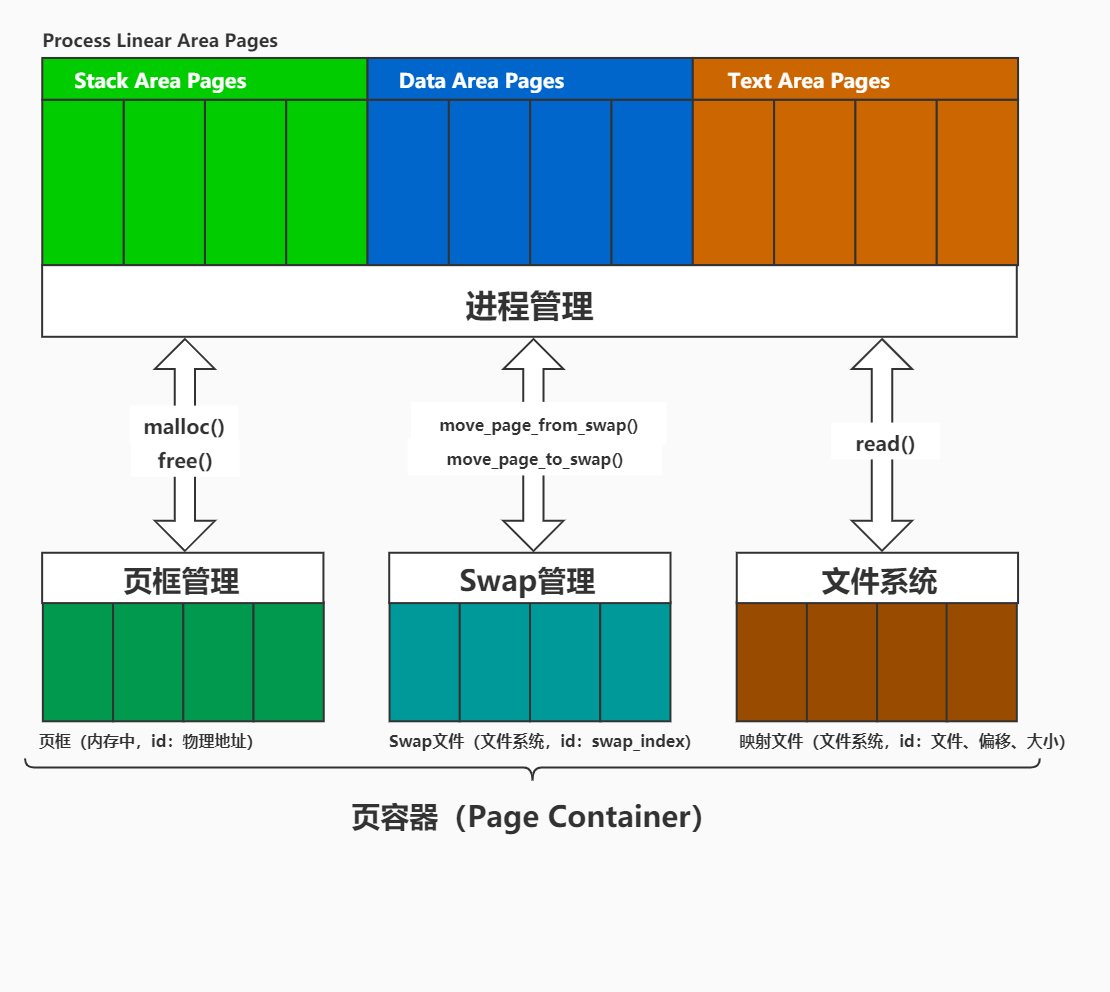
分页机制将内存分为一个一个4K大小的页，同时在物理地址的基础上建立了线性地址的概念。线性地址是一种虚拟的地址，它和物理地址不同，不会直接从CPU通过地址总线送至内存，但它与物理地址一样都是32位的，分页机制就是用来完成这种虚拟的线性地址到实际物理地址的映射。虽说线性地址是虚拟的，但关键在于CPU可以识别线性地址，这个地址会在硬件中自动转化为物理地址这个过程对进程完全是透明的，正是有了这种映射，操作系统可以为进程提供统一的内存模型。

分页机制还拥有很多其他功能，比如权限控制、读写控制等，其中就包括了我们最近在做的缺页中断。

分页机制其实就是建立一张张表，操作系统只需要向表中填写合适的值，就可以完成想要的32位地址到32位地址的转换。这套机制最大的一个好处是，我们不需要将所有的页表和页面在一开始就分配好，可以将页表或者页目录的某一项暂时置为空，也就不用分配实际的物理地址，等到了真正需要访问某个地址时，分页机制通过中断来告知操作系统，这个中断就是我们所说的缺页中断。页表项和页目录项的最后一位就表示了这一页是否存在，如果这一位为0，而CPU试图访问这一页的某个地址，那么硬件就会触发缺页中断。

在现代操作系统的内存管理中，有一个贯彻始终的理念，就是所有的分配都是拖到不能再拖才进行，也就是所谓的“懒加载”、“延迟分配”。因为操作系统不是为一个两个进程服务的，它要尽可能保证自己能应付更多的任务，这时候“懒惰”就换来了整个系统的“勤快”。在i386机器上，内存管理实现“懒加载”所利用的基本硬件机制就是分页提供的缺页中断。

基于“延迟分配”的基本理念，我们为MiniOS提出了一套内存管理的抽象，整体思路是站在进程的角度上，围绕着缺页中断，完善其周围各组件的管理与协调。



这是我们的架构图。

我们首先确定的一个概念是“页”。这里借鉴了Linux的设计。对一个进程来说，它所拥有的页是与物理内存无关的，页指的是数据，是进程线性结构中的一部分。进程的线性区指的就是我们为进程提供的一种抽象，让进程看起来自己拥有全部的4G内存。

与此同时，我们将实际内存中的物理空间称为“页框”。将页与页框分开来看是很重要的，页框指的是一个地方，它能容纳一个页。

在这套架构中，页框属于“页容器”的一种。页容器指的是能够存放页数据的容器，比如上述的页框，又比如这里的swap文件、映射文件。Swap文件是硬盘上的一块保留的区域，可以在系统内存不足的时候，将页框中的页面放入其中以节省内存。而映射文件指文件系统中的普通文件，只不过这些文件的一部分与进程线性地址中的一部分关联起来了。

页容器中的单位空间我们称为“页空间”。每种页容器都有能够标识其中一块页空间的id。对于物理内存来说，这个id是4K对齐的物理地址；对于swap文件来说，id是按4K划分的块的序号；映射文件的id则是一个三元组：文件描述符、文件偏移以及大小。

页容器的概念可以将这三者统一起来。根据每种页容器特性与用途，它们又有一些不同的属性，比如能否写回、页与页空间的大小关系等等。

页框其实是一种特殊的页容器。因为CPU架构的原因，所有需要访问的数据最终都必须放到内存中，也就是页框中，页框也就成为了整个架构的一个重心。

有了这个架构，我们来试着用这个架构完成一下我们的懒加载机制。

懒加载实际上可以根据进程线性区域不同性质以及其他因素被分成几个相对独立的功能：代码段（文本段）和数据段的懒加载、栈段的懒加载以及页面的交换。

第一个功能，代码段和数据段的懒加载。我们要运行一个程序，首先需要把代码段和数据段加载进内存中，而这两个段的内容是位于文件系统当中的，这刚刚好与上图的映射文件页容器对应。

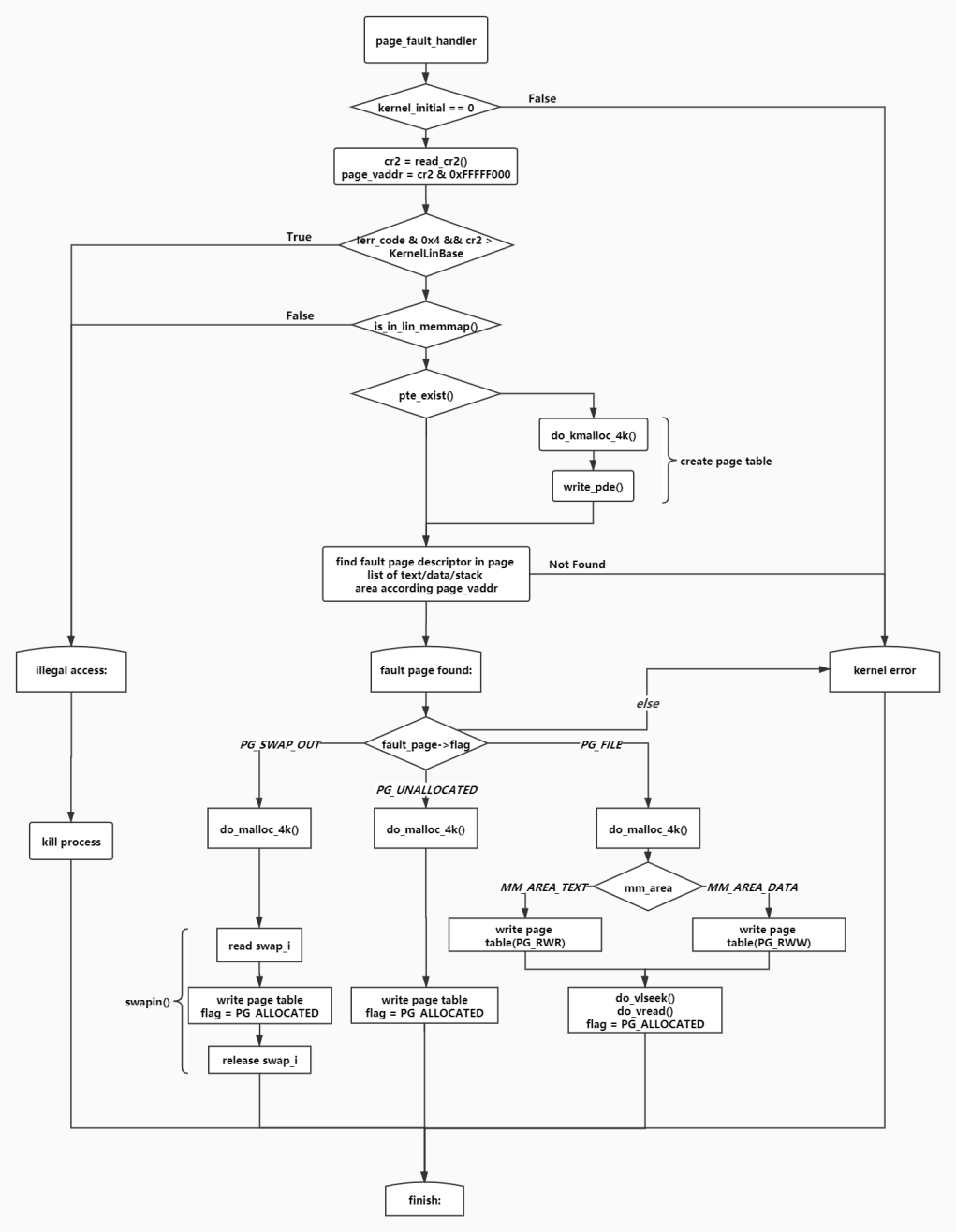
第二个功能，栈段的懒加载。栈段是不存在程序相关的初始化数据的，因此我们只需要为页面在物理内存中寻找一块空间即可，这与上图页框页容器对应。

第三个功能，页面的交换。当系统物理内存不足时，将页面从页框中拿出来放到swap页容器中，等到需要的时候，再取回来即可。

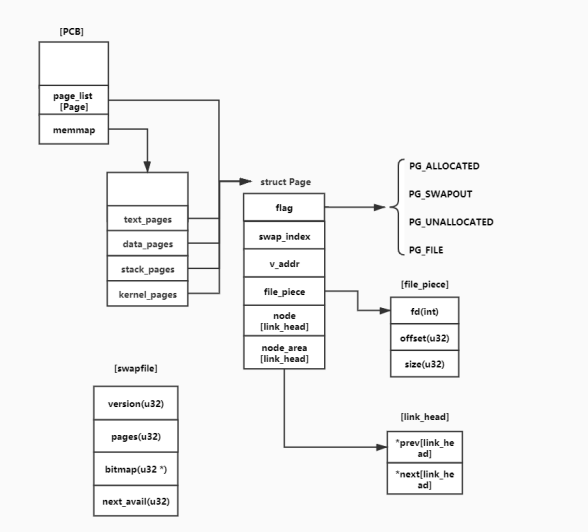
到这里，我们的懒加载机制就完成了，还顺便引入了页面的换入换出。

这一整套架构的实现依靠的就是缺页中断，所以我之前说我们是围绕着缺页中断做的内存管理，所有的分配与加载都是在缺页中断中真正完成的。

缺页中断具体流程是这样的。



我们先来看看数据结构的部分。



这里主要的思路是在进程的PCB中维护当前进程的所有存在于线性区中的页的信息，比如页的线性地址、页的类型、在交换分区中的索引、在文件中的位置。这是我们的页描述符结构体，这里的flag字段表明了这个页当前的类型，每种类型表明下面某个字段有效。在发生缺页中断的时候，就可以查询这张表。同时，为了权限控制以及快速定位某一页，我们把这个链表根据线性区拆成多个链表，串联起每个线性区中的页描述符。

另外，swap的管理使用了struct swapfile 这个结构体，其中维护了一个位图，表明swap分区或文件中的某一个页空间是可用的。

有了这些数据结构，就可以完成这个缺页中断处理函数了。

可以看出，每个页容器都有着自己的操作函数，使用者只需要使用它们的id与他们进行交流即可。

为了配合缺页中断处理函数，我们对系统的exec进行了改造，思路也很简单，就是在读取elf头之后，把各段的页面设置为不存在，并在页描述符表中添加相应的项，等到这个进程被调度到，就会触发缺页中断，上面的处理函数能够获取到足够的信息以解决这个缺页问题。

Swap的实现也很相似。因为我们没有时间去做一个换入换出算法，所以我们只保留了接口函数swapout和swapin，通过这两个函数就可以对进程的页面进行换入换出，至于其中的实现是对上层隐藏的，之后如果要写换入换出算法的话，就可以利用这两个函数。

这些就是我们主要做的工作，作为辅助，我们还实现了几个内核断言和调试宏，一个灵活的链表，一个位图算法。

内核中的日志很重要，从调试器中一步一步跟踪问题是很费时费力的事情，很多时候有个日志打印就可以解决大多数问题。另外，断言的作用也很大，在某些必须成立的条件上就可以下个断言，出了问题我们自然能看到，不必每次都去if-disp\_str这样子写。这是我们的kassert函数实现。类似的还有kdebug、kfatal等。

链表参考了Linux的实现。相关的数据结构只有一个简单的struct link\_head。可以实现多个链表连接向同一个结构体的功能。

不足：

因为目前我们的disp\_str函数没有同步锁，多个执行流同时进行打印会使得输出交错，尤其是与用户进程的输出相互覆盖。我觉得好点的解决方案是在内核中也是用tty设备进行输出，并且对输出进行加锁，这样的话，上面两个问题都可以避免。

第二个问题就是，因为时间仓促，我们的数据结构都没来得及进行细致的设计。比如上面的页描述符中的数据字段就可以使用一个共用体来实现。我们还可以对线性区描述进行重构，把线性区页描述成一张表，这样就可以把一些通用的属性抽象出来，而不用在代码中特意区分。

第三个问题是，我们没能完整实现swap分区的读写，因为我们目前还没有找到比较合适的读写方案，有几个可行的方案也因为时间仓促没写出来。

1. **所遇问题及解决方法**

TTY设备会向显存中写入数据，在分页机制启动之后，显存应该就被映射到了0xc0000000以上的地址，但miniOS现有的代码中直接使用了原有的0xb800，导致出现错误的缺页中断。修改VM\_MEM\_BASE之后解决了这个问题。