操作系统第八章笔记

Zhitao Yin

2024/11/23

1 background

1.1 Memory Management Requirements 内存管理需求

1. 重定位: 计算当程序由磁盘转入主存时，虚拟地址对应的物理地址

   这里对重定位进行一个比较详细的说明：

   重定位：当一个程序转入到主存时，他真正的物理地址才开始被确定，但是由于 Logical organization 的存在，这个程序可能会被分成很多模块，分别映射到不同的物理区域，也就是说 memory not Contiguous ，好吧，这东西确实比较难讲清，让我单独分一个块出来，详见下文 2.4

2. 保护：保护不同进程的内存不被彼此访问
3. 共享：允许进程相互合作，分配一块共享的空间
4. 逻辑组织：按逻辑、功能把用户写的程序分块
5. 物理组织：

2 Contiguous Allocation

主存包括两个部分，操作系统占据一部分内存，用户进程占据高位的内存（高位的意思是：地址比较高）

2.1 固定分配:fixed partition

对于每一个进程都分配等量的内存空间，将用户态能够占领的内存部分等分。来一个进程就自动按顺序占领一块内存空间。

好处是：分配容易

坏处是：容易产生内部碎片。

内部碎片：一个进程并不需要这么多的内存空间，因此分配的存储空间中有一部分不被使用。属于该进程，但是不用，叫做内部碎片。

2.2 动态分配:dynamic partition

对每个进程根据其需要分配内存空间。

相比静态分配，这样的好处是不会产生内部碎片

但是会产生外部碎片，**外部碎片**：由于连续动态分配后，假设内存现在只剩 2kb 的内存，如果来了一个需要 3kb 的进程，那么该进程就不会被放入主存，因此就有一部分的内存是空闲出来的，称之为外部碎片。

常见算法：best fit , first fit, next fit

2.2.1 best fit

切记，main Memory 中是用链表去连接每个独立开的空闲空间块。

选择一个离 proc 索要的 memory 大小最贴近的一块空闲内存块分配

2.2.2 first fit

从链表头开始找，找到一个合适的就行。

2.2.3 next fit

分配内存时不是从链首进行查找可以分配内存的空闲分区，而是从上一次分配内存的空闲分区的下一个分区开始查找，直到找到可以为该进程分配内存的空闲分区；

2.3 Buddy system 伙伴系统

简单来说就是，对于操作系统的 main Memory，首先先不分，假设 MM 有 16kb，记录 upper bound 和 lower bound：U = 4,L = 0 。此时来了一个

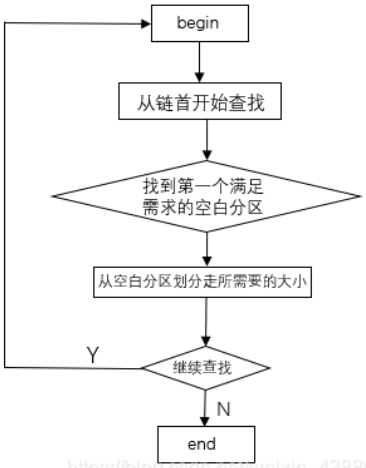

图 1: first fit

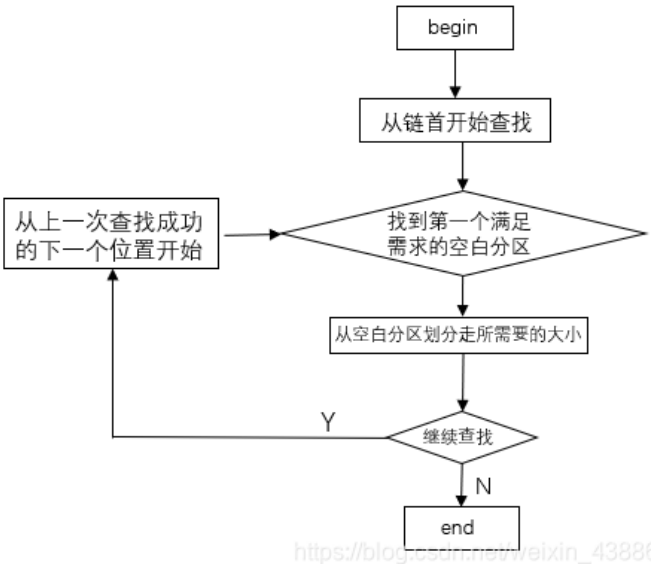

图 2: next fit

4kb = 2^2 的进程索求这么大的空间，因为 $2 < 4$ 说明此时可以给这个进程分配空间，最大的块足够。

那么此时操作系统会将 16kb 分成两块 8kb，其中一个 8kb 又会分为两个 4kb，其中一个就给这个进程使用。那么现在 L = 2。

如果再来一个 3kb 的怎么办？给他 4kb 的块！

如果来个 1kb 的怎么办？把 4kb 分成两块，再分成两块成 1kb 给他。如果没有 4kb 了怎么办？8kb 也没有？16kb 也没有？总有一个大的，给他往下划分就完了。虽然这种情况也会有一些外部碎片，but 总体来说比较好了啦。

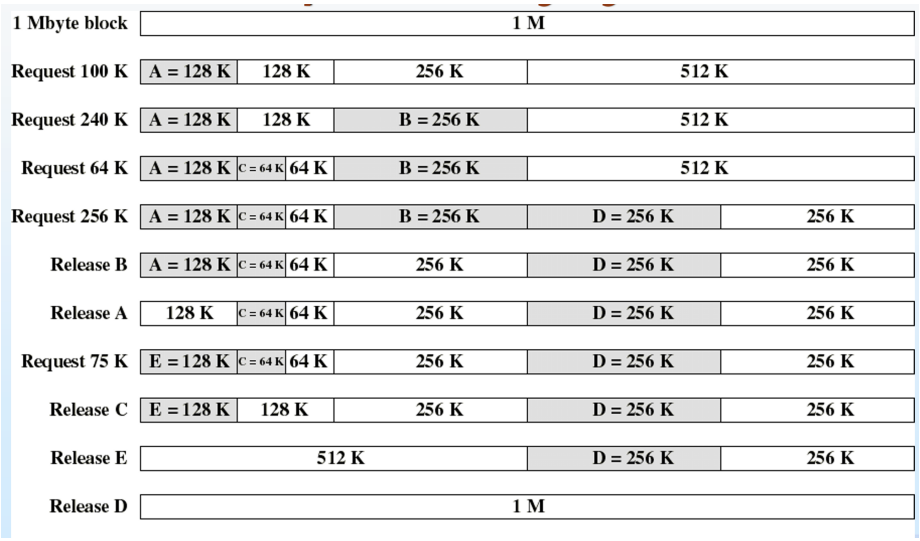

图 3: buddy system 伙伴系统

2.4 Memory 之 Relocation - Address translation(Memory protection)

2.4.1 基本术语

逻辑地址，物理地址，相对地址：

逻辑地址：虚拟的；(logical Address)

物理地址：真实的；(physical Address or saying Absolute Address)

相对地址：给你一个 base Address,base + 相对地址 = 物理地址。(relative Address)

2.4.2 要用到的寄存器

base register: 基地址寄存器，记录进程的起始地址，见 relative Address
bounds register: 限度寄存器, 记录进程的尾地址

2.4.3 Address translation Mechanism 地址转换机构

base register + relative Address > bounds register? jump|error
待补充...

3 Swapping 交换

简单来说，就是主存的存储空间毕竟是有限的，因此当必要的时候，移出一些进程以腾出主存的空间，然后移入一些更加需要放在内存的进程。接下来我举个例子。

3.1 如何记录哪些进程存放在 backing store 的哪个位置？

backing store: fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images；一个足够储存所有用户的所有进程信息的磁盘

Roll in , Roll out: 进入主存成为 Roll in, 被移出主存称为 Roll out
移入移出是根据优先级调度算法的。

3.2 Questions to be mentioned

1. 在外存的什么地方储存被换出的进程？

磁盘一般有对换区和文件区，对换区占磁盘的一小部分，用于存换出的 proc。并且对换区的 info 交换到主存的速度 » 文件区的速度

2. 什么时候 swap？

内存吃紧时交换。例如，当许多进程运行时都出现缺页的情况时，也就是说 main memory 不够了，就可以换出一些进程；如果缺页率明显下降，就暂停换出。

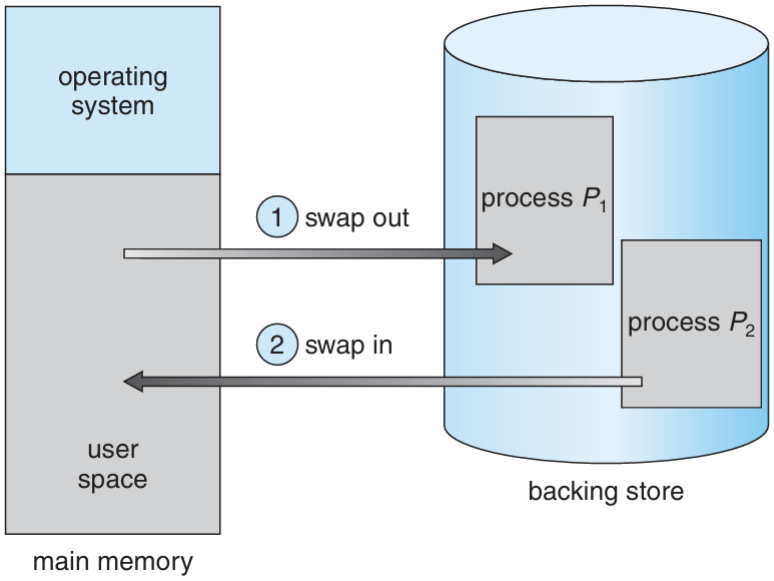

图 4: swap

3. 换出哪些进程 (already solved)

4 paging 分页

4.1 pages and frames 页和帧

frames:Divide physical memory into fixed-sized blocks

pages: Divide logical memory into blocks of same size 也就是说，把进程的逻辑空间地址分为很多页。例如，你写个 helloworld，效果是在大屏幕上展现 helloworld 的烟花字样，那么你需要实现 print 和 paint，假设这两个功能各 1kb，那么如果一页是 1kb 的话，你这个 program 就要被分为 2 页。

4.2 page table 页表

首先我要告诉你的是，每一个进程都有一个 Pagetable，并不是 cpu 中只有一个 pagetable。

每一个 proc 的 pgtbl 的内容只有：frame num。是的，没错！pgtbl 就像一个 int 数组一样，里面 a[i] 存放的就是页框号 (页框号 = 内存块号 = 物理块号...)！

我现在随机假设一个进程，其可以分成 4 页，一页是 4kb 大小，现在假设 pagetable = [1,3,5,7], 那么第 1 页对应的物理页框就是 1，第 2 页对应 3 号页框

然后如何得到进程第 i 页中 offset = 50 的 PA(physical Address) 呢？frame num = pagetable[i], Memory(per frame) · frame num = 起始地址。然后起始地址 + offset = PA。

4.3 基本地址转换机构 (paging hardware)

PTR:page table register 页表寄存器，PTR 寄存器由两部分组成，32 位 bit 被分为两部分，前一部分是页表在内存中的起始地址 F，后一部分记录页表的长度 M。注意，PTR 是存于 OS 的 main Memory 中的。并不移出！

在进程被移入到主存前，F 和 M 都由 PCB（进程控制块）存储，当进程被调度的时候，OS 会把他们放到页表寄存器中。

我好像忽略了什么...我有点忘记了，稍后再补充吧。

5 Segmentation 分段

分段和分页很像，但只是像噢，哪里不一样呢？

5.1 Structure of a program 程序结构

如果让你去看一个 pj，我相信你是有这个能力去判断哪些函数是用于计算数据，哪些函数是用于 debug，哪些函数是用于保障基本输入输出的。

现在我们让计算机也有这个能力 (实际上计算机也确实有这能力)。计算机按照自己的理解将程序分为了很多个部分（按功能划分的），这些部分成为 program's Segmentation. 见下图：

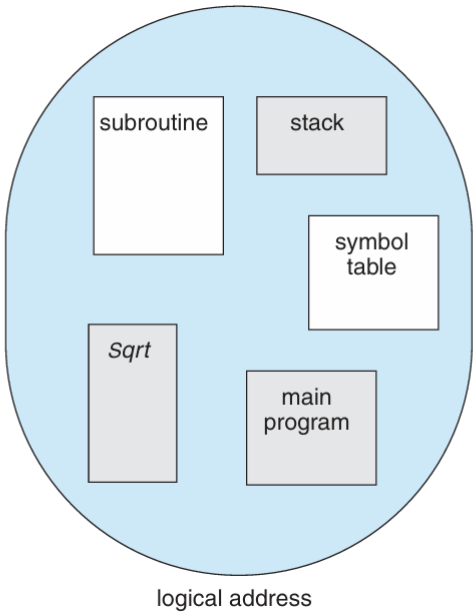

图 5: segmentation

5.2 Segmentation table 段表

段表也是一个列表，但不是数组，因为列表中每一个元素都由两部分组成：base and limit segment table = [<base,limit>，…]

base–contains the starting physical address where the segments reside in memory

limit–specifies the length of the segment

Segment-table base register (STBR) points to the segment table's location in memory

Segment-table length register (STLR) indicates number of segments used by a program

至于程序的虚拟地址中，哪一部分属于哪一段呢？因为只有知道属于哪一段你才能查表然后找到物理地址呀！有趣的部分来咯：因为虚拟地址是没有意义的！因此对于 32bit 的虚拟地址而言，我找到你这个 program 内存占比最大的一段，我称为 Seg1, 那么 seg1 假设占比 2^30 的内存, 那么前两个 bit 的意义我们将其认为段号！因为 offset 只需要 30 位，实际上前 2 位没什么用，我们可以随意用来区分段号！

那么如果这个时候段 >4 呢？2 位 bit 不够怎么办？首先呢，这种情况很少见；出现了的话，用两个寄存器不就完了，64 位总是够的。

5.3 paging and Segmentation

各有优劣。

paging 很方便。segmentation 很聪明。

6 Load and Link：装入与链接

链接很有意思，也是一种映射方式吧，有点像分段，但是不如分段那么格式化，比较随意。

6.1 装入

试想一个 MIPS code: add s1,s2,s3

s1,s2,s3 实际上都是一个地址，敢问：是 PA 还是 LA？是 LA。

装入就是，当你把这种地址操作的程序装到 main Memory 里面的时候，这些地址怎么处理？

6.1.1 dynamic loading 动态装入

这里只介绍一种，还有另外两种：重定位装入 + 静态装入。

我用两句话给你解释完，绝对装入：装到主存之前，这些所有的相对地址就改为绝对地址。这种是糖丸了的。

重定位装入：装入之后，根据程序的 base（起始地址），把所有相对地址加上 base。这就要求你分配的地址空间必须连续。因为 base 只记录一个值。

dynamic loading 动态装入: 用一个寄存器记录某个模块 (segmentation) 的开始地址，只把需要用到的 module 和 data 放进主存，然后用 base reg（可以用多个）记录起始地址。

6.2 Linking

6.2.1 从写程序到运行程序，Linking 起到什么作用？

见下图。

6.2.2 Link 的三种方式

1. 静态链接 static Link：直接将各个 module 连在一块成为一个完整的可执行文件，以后再不分开。缺点很显然，你一个 module 原本可以重复利用，你把他连在一起了，怎么重复用？

2. 装入时动态链接，program 要用的时候，里面的这些 module 移入主存的时候边装入边 link 起来。

3.dynamic Linking: 装入 program 的时候，如果要用这个 module，才装入时 link，否则都不连起来。

6.2.3 dynamic Linking 的运行方式

编译时，先得知哪些段需要被链接。为每个进程保持一个用于记录已连接段的表目，称为段名-段号对照表。段名 ['main'] 段号 1。

编译时生成一个符号表，为将外段的符号名转为对应的段内地址，每个段在编译时，生成一个符号表

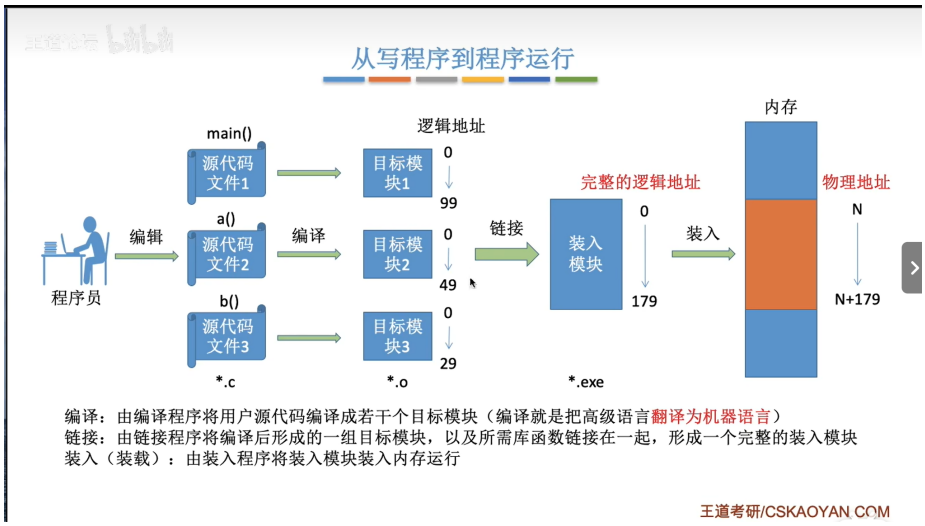

图 6: process of program being executed

所以现在有三个表：段名-段号符号名-段内地址段表。

编译的时候，如果一个段中需要访问另外一个段。那么这时候会跳到一个间接字：L|D 32bit 的一个数，前面 L 是 1bit。如果 L 是 1，代表段号未定，并且此时的 D 可以分为两部分 [X]|Y，X 表示段名，Y 表示段内 offset。那么此时你可以把段名-段号加入到表中。并且段表更新该段对应的首地址。L 是 1 也可以代表段号已经被别的 program 用了。那么此时你把 D 变为段号-offset 即可。

如果 L 是 0，此时 D 就是段号-offset，你可以取段号根据段表得到 base，base+offset 得到 PA。