第一部分　初始化、进程初始化

第5章　两个文件

本章从易处着手介绍第一部分中的两个文件，它们可与本书其他部分较清晰地隔离出来。

对这些文件的讨论补充了第 3章中对“ C ”语言的概括介绍，包括了若干与“ C ”语言语法和语义有关的注解。

5.1　文件malloc.c

此文件仅由两个过程组成：

malloc(2528),mfree(2556)

这两个过程涉及两类存储资源的分配和释放，这两类存储资源是：

• 主存：它以32个字(64字节)为单位。

• 盘交换区(disk swap area)：它以256字(512字节)为单位。

对于这两类资源的每一类，各有一些资源图 (coremap或swapmap)记录可用区列表。指向相应资源图的一个指针作为参数传递给“ malloc ”和“ mfree ”，于是这两个过程无需了解它们正处理的资源的类型。

“ coremap ”和“ swapmap ”都是类型为“ map ”的结构数组，“ map ”在2515行说明。该结构由两个字符指针，亦即无符号整型组成。

对“ coremap ”和“ swapmap ”的说明分别位于0203行和0204行。在这里，完全忽略了“ map ”结构——这是一种程序设计捷径，由于装入程序并不检测这一点，所以可以进行这种形式的说明。在“ coremap ”和“ swapmap ”中的实际列表元素数分别是“ CMAPSIZ/2 ”和“ SMAPSIZ/2 ”。

5.1.1　列表维护规则

1) 每个可用存储区由其长度和相对地址定义 (按相应资源的单位计算)。

2) 每个列表中的各元素总是按相对地址从低到高排列。任何两个相邻列表元素所描述的可用存储区如果构成一个连续可用区，则总是立即将它们合并成一个区。

3) 从第一个元素开始顺序逐个查看就能扫描整个列表，当查看到一个零长元素时结束扫描。此最后一个元素并不是有效的列表部分，而是该列表有效部分的结束标志。

上述规则提供了对“ mfree ”完整的规格说明，提供了对“ malloc ”完整规格说明，但有

一个方面例外。此例外方面是：

当有多种方式可进行资源分配时，我们须说明在具体实现时选择哪一种。

“ malloc ”所采用的方法是“ 首次适配 ”算法 (“ First Fit ”)。采用这种算法的理由读者以后会体会到。

下面举一个例子说明资源图是如何维护的，假定下列 3个资源区是可用的：

- 一个可用区的长度为15，在位置47处开始，在61处结束。
- 一个可用区的长度为13，在位置27处开始，在39处结束。
- 一个可用区的长度为7，在位置65处开始。

于是，资源图应当包含：

项	长 度	地 址
0	13	27
1	15	47
2	7	65
3	0	??
4	??	??

如果接到一个长度为7的存储空间请求，则从位置27处开始分配，分配后资源图变成：

项	长 度	地 址
0	6	34
1	15	47
2	7	65
3	0	??
4	??	??

如果从地址40开始到46结束的存储区返回至可用区列表，则资源图变成：

项	长 度	地 址
0	28	34
1	7	65
2	0	?
3	??	??

注意，虽然可用存储资源的总量增加了，但是由于3个可用存储区(其中1个是刚释放的区)合并，该列表中的元素数反而减少 1。

现在让我们转过头来考虑实际源代码。

5.1.2 malloc(2528)

此过程体由一“ for ”循环组成，它搜索“ map ”数组，直至：

1) 到达可用资源列表的尾端；或

2) 搜索到一个区，其长度能满足当前请求。

2534：“ for ”语句首先对“ bp ”赋初值，使其指向资源图的第一个元素。在每一次循环时，“ bp ”增1以指向下一个“ map ”结构。

注意：循环继续条件“ bp－＞m_size ”是一表达式，当引用资源图结束标志元素时，该表达式值为0。该表达式也可等效地写成更明显的形式“ bp－＞m_size＞0 ”。

还要注意：对数组的结尾没有进行显式测试。(假定CMAPSIZ，SMAPSIZ＞＝2*NPROC，那么这种测试是不必要的!)

2535：若该列表元素定义了一个区域，其长度至少与所要求的相等，则……

2536：记住该区第一个单元的地址。

2537：增加存放在该数组元素中的地址。

2538：减少存放在该数组元素中的长度，并将结果与 0比较(也就是检查是否确好适配)。

2539：如果精确适配，则将所有后续元素(直至并包括结束标志元素)都下移一位置。

注意：“ (bp-1) ”指向“ bp ”所引用的前一个结构。

2542：“ while ”的继续条件并不测试“ (bp-1)－＞m_size ”和“ bp－＞m_size ”是否相等。被测试的值是由“ bp－＞m_size ”赋予“ (bp-1)－＞m_size ”的值。

(你如果没有立即识别这一点，也是可以宽恕的。)

2543：返回该区域的地址。 return语句结束了“ malloc ”过程，因此非常肯定也就结束了“ for ”循环。

注意：返回值0意味着“ 不幸 ”。这基于下列事实：没有一个有效区会在单元 0处开始。

5.1.3 mfree(2556)

此过程将一个存储区返回给由“ mp ”指定的“ 资源图
地址为“ aa ”。此过程的体由一行“ for ”语句和占多行的一

2564：本行尾端的分号是特别重要的，它终止了一条空
分号单独占1行，则会更清晰一些。)

此语句例示了“ C ”语言的能力或其晦涩。请试着用其
此语句等效的代码。

步进“ bp ”直至遇到一元素：

- 该元素的地址大于所返回区的地址，亦即不满足条件

```
i.e.not“bp－＞m_addr＜＝a”;
```

- 该元素是列表结束标志元素，亦即它不满足条件：

```
i.e.not“bp－＞m_size!=0”;
```

2565：我们已找到在其之前应插入新列表元素的元素。
还是由于合并，该列表仍保持原先的元素数，甚至其元素数
如果“ bp＞mp ”，那么我们就不会在列表开始处插入。如果

```
(bp-1)－＞m_addr+(bp-1)－＞m_size==a
```

那么正被返回的存储区与列表中前一元素描述的存储区紧相邻接。

2566：将前1列表元素的长度增加正被返回区的长度。

2567：正被返回区是否与列表中下一个元素紧相邻接。如果是这样……

2568：将下一列表元素的长度加至前一元素的长度。

2569：将所有余下的列表元素向下移动1个位置，直至并包括该列表结束标志元素。

注意：若2567行的测试在“ bp－＞m_size ”为0时意外地产生结果为真的情况，那也不会造成任何损害。

2576：若2565行的测试失败，则执行此语句，亦即正被返回的区不能与列表中的前一元素区相合并。

这能否与下一元素区合并呢?注意本语句中对下一元素是否为空(null)的检查。

2579：假定正被返回的区非空(在进入本过程时就应对此作检查)，则在列表中加一个新元素，然后将所有余下的元素向上(也就是向数组下标增大方向)移动一个位置。

5.1.4 结论

这两个过程的代码编写得非常紧凑。几乎没有可以删除之处以提高运行时的效率。但是，有可能将这些过程写得更清晰一些。

如果你对此抱有同感，那么作为练习请改写“ mfree ”使其功能更易于辨别出来。

也应注意，“ malloc ”和“ mfree ”的正确功能依赖于“ coremap ”和“ swapmap ”的正确初始化。执行此种初始化的代码在过程“ main ”中(1568行，1583行)。

```
(2416)
(2433)
ror (2447)
```

了一种向系统控制台终端发送消息的方法，这种方法直接、

不复杂、不带缓存。在系统初启、报告硬件出错和系统突然崩溃时使用此过程。

此处的“ printf ”和“ putchar ”在核心态下运行，它们类似于但并不等同于由“ C ”程序调用的库函数“ printf ”和“ putchar ”，后者在用户态下运行，位于库“ /lib/libc.a ”中。此时阅读UPM中的“ PRINTF(III) ”和“ PUTCHAR(III) ”是有益的。

2340：当程序员为此过程说明各参数时，他一定是走了神。事实上该过程体只引用“ x1 ”和“ fmt ”。

从此可以看出“ C ”程序设计在过程调用和过程说明时参数匹配的规则，该规则不进行两者之间参数的匹配检查，两者的参数数甚至也可以不同。

参数以逆序放到栈上。当调用 printf时，“ fmt ”与“ x1 ”比较，其位置更接近于栈顶，其他参数的情况依此类推 (见图5-2)。

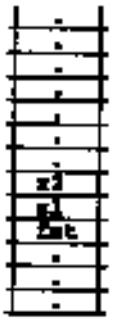

图 5-2

“ x1 ”的地址高于“ fmt ”，但低于“ x2 ”，其原因是在PDP11中栈向下生长，亦即向低地址方向扩展。

2341：“ fmt ”可解释为常数字符指针。此说明也 (几乎)等同于：

“ `char *fmt;` ”

其区别是：这里的“ fmt ”值是不能更改的。

2346：将“ adx ”设置成指向“ x1 ”。表达式“ &x1 ”是“ x1 ”的地址。注意，“ x1 ”占用的是栈单元，所以在编译时不能对此表达式求值。

随处可见的很多涉及变量或数组的表达式地址是很有效的，它们可以在编译或装入时求值。

2348：从格式字符串顺次取出字符并送入寄存器类变量“ c ”。

2349：若“ c ”不是“ % ”，则……

2350：若“ c ”是一null字符(“ \ 0 ”)，则表示格式字符串以正常方式结束，于是“ printf ”终止。

2351：否则调用“ putchar ”将该字符发送至系统控制台终端。

2553：见到一个“ % ”字符。取下一个字符(最好取得的字符不是“ / 0 ”!)。

2354：若此字符是“ d ”、“ l ”或“ o ”，则调用“ printn ”，传送给它的参数有两个，一个是“ adx ”所引用的值，另一个则取决于“ c ”的值，若其值为“ o ”，则该参数值为“ 8 ”，否则为“ 10 ”。(从此可见对于“ d ”和“ l ”的代码是完全一样的。)

“ printn ”将一个二进制数按第二个参数所表示的基数转换成一组数字字符。

2356：若格式编辑字符是“ s ”，则以null终止字符串中的所有字符 (除null外)都被传送至终端。在此种情况下，adx应当指向一字符指针。

2361：增加“ adx ”，使其指向栈上的下一个字，亦即指向传送给“ printf ”的下一个参数。

2362：回到2347行，继续扫描格式字符串。热衷于结构化程序设计的程序员将会把 2347行和本行代换成：

“ `while”(l) {“ and ”}` ”

5.2.2 printn(2369)

为了按所要求的顺序产生相应数字字符，“ printn ”递归调用自身。将本过程的代码编写

得更有效一些还是有可能的，但不会是更加完善。(无论如何，从实现“ putchar ”的观点分析，效率在这里几乎是不必考虑的。)

假设n=A*b+B,其中：

• A=ldiv(n,b)，而且

• B=lrem(n,b),0＜＝B＜b。

那么，为了显示n的值，我们需要先显示A的值，其后跟随B的值。

对于b=8或b=10，后者是容易做到的：它只是一个单一字符。如果 A=0，那么前者是容易的。如果递归调用“ printn ”，那么总能达到A=0的情况。因为A＜n，递归调用链一定会终止。

2375：对应于数字的算术值，可方便地变换成相应的字符表示，方法是将算术值与字符“ 0 ”相加。

过程“ ldiv ”和“ lrem ”将它们的第一个参数处理为一个无符号的整型数 (亦即，在实际除法操作之前将16位值扩展为32位时，不进行符号位扩展)。这两个过程分别从1392行和1400行开始。

5.2.3 putchar(2386)

本过程将作为参数的字符传送至系统控制台。这是一个小的实例，从中可以观察到PDP11计算机i/o操作的基本特征。

2391：“ SW ”在0166行定义的值是“ 0177570 ”。这是一个只读处理器寄存器的地址，在该寄存器中存放控制台开关寄存器的设定值。

此语句的意思是清晰的：取得 0177570单元的内容，然后检查它们是否为 0。问题是将其表示成“ C ”语言形式。代码

```
if(SW==0)
```

并没有实现上述要求。很清楚， SW是一个指针的值，应当用其进行间接访问。编译程序是否能接收下列形式的语句呢？

```
if(SW－＞==0)
```

但这在语法上是不正确的。在 0175行说明了一个伪结构，它是一个元素“ integ ”，使用此结构程序员找到了一个解决上述问题的方法。

在这一过程和其他地方可以找到许多其他类似的实例。

使用硬件术语进行描述，系统控制台终端接口由 4个16位控制寄存器组成，它们在单总线上连续编址，其起始地址是核心态地址 0177560(参见0165行对KL的说明)。关于这些寄存器的说明和使用方法请参见《PDP11外设手册》第24章。

使用软件术语进行描述，该接口是一个在 2313行至2319行定义的无名结构，它具有四个元素，它们的名字是4个控制寄存器的名字。因为对这种结构无需分配任何实例，所以该结构没有名字并无妨碍。(我们关心的是在KL所给予的地址上预定义的东西。)

2393：在发送器状态寄存器(“ XST ”)的第7位为0时，不做任何事情，其原因是该接口并没有为接收下一个字符准备就绪。

这是一个“ 忙等待 ”(“ busy waiting ”)的经典实例，在这里处理机无用地反复循环执行

一组指令，直至发生某个外部事件。这种处理机能力的浪费通常是不可容忍的，但在本特例中却可接受。

2395：本语句的作用与2405行的语句紧密相关。

2397：保存发送器状态寄存器的当前内容。

2398：清除发送器状态寄存器，为发送下一个字符作准备。

2399：清控制状态寄存器的第7位，将要发送的下一个字符送入发送器缓冲寄存器，这启动了下一个输出操作。

2400：一个“新行”符需伴随一“回车”符，这由递归调用“ putchar ”实现。

也加进了2个额外的“删除”字符，其作用是提供在终端上完成回车操作所需的延迟。

2405：对“ putchar ”以一个参数0进行调用有效地造成2391行至2394行的再执行。

很难看出为什么程序员在这里选用递归调用，而不选用简单地重复 2393行和2394行。即使不考虑清晰程度，仅考虑代码的效率和紧凑性方面，在这里使用递归调用法也是有害而无益的。

5.2.4 panic(2419)

UNIX操作系统中有很多位置都要调用本过程。当继续系统的操作似乎是并无希望时将调用本过程。

UNIX并不自称是一个“容错”或“个别部件发生故障时仍能可靠工作但性能有所下降”的系统。在很多情况下，针对某些很明显的问题调用“ panic ”是一种行之有效的解决方法。比较复杂的解决方法需要增加大量的代码，这与UNIX遵循的哲学“尽量使其简化”是背道而驰的。

2419：本语句的作用在2323行开始的注释中说明。

2420：“ update ”使所有大的块缓存的内容都写到相应块

2421：以一个格式字符串和一个参数调用“ printf ”，该

2422：此“ for ”语句构成了一个无限循环，该循环中

“ idle ”(1284)。

“ idle ”将处理器优先级降为 0，然后执行“ wait ”。这续无限长时间。当发生一硬件中断时，“ wait ”终止。

对“ idle ”进行无限次调用优于执行一条“ halt ”指令得以完成，并且系统时钟能继续工作。操作员可以采用的从新启动系统(如果愿意，则可在内存转储操作之后)……

5.2.5 prdev(2433)、deverror(2447)

在i/o操作中出错时，这两个过程提供警告消息。在此“ printf ”的实例。

5.3 包含的文件

应当注意的是：“ malloc.c ”文件并不要求包含其他文件包含4个文件。

细心的读者会注意到，在文件系统的层次结构中，这 4个文件所在的层次较“ prf.c ”高一层。

2304行语句可被理解为在其位置上代换成“ param.h ”文件的全部内容。这样也就增补了标识符“ SW ”、“ KL ”以及“ integ ”的定义，它们都在“ putchar ”中出现。

如前所述，“ KL ”、“ SW ”和“ integ ”的说明分别位于0165、0166和0175行，但是如果在“ prf.c ”中没有包含“ param.h ”，那么它们对“ prf.c ”中的各过程是没有任何意义的。

“ prf.c ”中包含了“ buf.h ”和“ conf.h ”文件，它们提供了“ d_major ”、“ d_minor ”、“ b_dev ”和“ b_blkno ”的说明，在“ prdev ”和“ deverror ”中使用到它们。

包含第4个文件的理由较难发现。从代码本身分析这并不需要，“ prf.c ”的编写者应当为此对读者表示歉意。在编辑这一部源代码时，很可能将“ integ ”的说明从“ seg.h ”搬到了“ param.h ”。

注意，变量“ panicstr ”(2328)是全局型的，但在“ prf.c ”之外并不引用它，所以没有将其说明放在“ .h ”文件中。