

**计算机系统**

**大作业**

题 目 程序人生-Hello’s P2P

专 业 计算机科学与技术

学　　 号 1190301610

班　　 级 1903603

学 生 王家琪

指 导 教 师 刘宏伟

**计算机科学与技术学院**

**2021年6月**

**摘 要**

hello虽小，五脏俱全。本文在linux操作系统下对C语言程序hello.c的运行全过程进行了全面分析。分析了P2P（program to process）从c文件转化为可执行文件过程中的预处理、编译、汇编和链接阶段，和Z2Z（zero to zero）可执行文件执行过程中的进程管理、存储空间管理和I/O管理的原理。

**关键词：汇编；链接；进程管理；存储空间管理**

**（摘要0分，缺失-1分，根据内容精彩称都酌情加分0-1分）**

**目 录**

[第1章 概述 - 4 -](#_Toc532238396)

[1.1 Hello简介 - 4 -](#_Toc532238397)

[1.2 环境与工具 - 4 -](#_Toc532238398)

[1.3 中间结果 - 5 -](#_Toc532238399)

[1.4 本章小结 - 5 -](#_Toc532238400)

[第2章 预处理 - 6 -](#_Toc532238401)

[2.1 预处理的概念与作用 - 6 -](#_Toc532238402)

[2.2在Ubuntu下预处理的命令 - 6 -](#_Toc532238403)

[2.3 Hello的预处理结果解析 - 6 -](#_Toc532238404)

[2.4 本章小结 - 7 -](#_Toc532238405)

[第3章 编译 - 8 -](#_Toc532238406)

[3.1 编译的概念与作用 - 8 -](#_Toc532238407)

[3.2 在Ubuntu下编译的命令 - 8 -](#_Toc532238408)

[3.3 Hello的编译结果解析 - 8 -](#_Toc532238409)

[3.4 本章小结 - 11 -](#_Toc532238410)

[第4章 汇编 - 12 -](#_Toc532238411)

[4.1 汇编的概念与作用 - 12 -](#_Toc532238412)

[4.2 在Ubuntu下汇编的命令 - 12 -](#_Toc532238413)

[4.3 可重定位目标elf格式 - 12 -](#_Toc532238414)

[4.4 Hello.o的结果解析 - 14 -](#_Toc532238415)

[4.5 本章小结 - 16 -](#_Toc532238416)

[第5章 链接 - 17 -](#_Toc532238417)

[5.1 链接的概念与作用 - 17 -](#_Toc532238418)

[5.2 在Ubuntu下链接的命令 - 17 -](#_Toc532238419)

[5.3 可执行目标文件hello的格式 - 17 -](#_Toc532238420)

[5.4 hello的虚拟地址空间 - 18 -](#_Toc532238421)

[5.5 链接的重定位过程分析 - 19 -](#_Toc532238422)

[5.6 hello的执行流程 - 22 -](#_Toc532238423)

[5.7 Hello的动态链接分析 - 23 -](#_Toc532238424)

[5.8 本章小结 - 24 -](#_Toc532238425)

[第6章 hello进程管理 - 25 -](#_Toc532238426)

[6.1 进程的概念与作用 - 25 -](#_Toc532238427)

[6.2 简述壳Shell-bash的作用与处理流程 - 25 -](#_Toc532238428)

[6.3 Hello的fork进程创建过程 - 25 -](#_Toc532238429)

[6.4 Hello的execve过程 - 26 -](#_Toc532238430)

[6.5 Hello的进程执行 - 26 -](#_Toc532238431)

[6.6 hello的异常与信号处理 - 27 -](#_Toc532238432)

[6.7本章小结 - 28 -](#_Toc532238433)

[第7章 hello的存储管理 - 29 -](#_Toc532238434)

[7.1 hello的存储器地址空间 - 29 -](#_Toc532238435)

[7.2 Intel逻辑地址到线性地址的变换-段式管理 - 29 -](#_Toc532238436)

[7.3 Hello的线性地址到物理地址的变换-页式管理 - 30 -](#_Toc532238437)

[7.4 TLB与四级页表支持下的VA到PA的变换 - 32 -](#_Toc532238438)

[7.5 三级Cache支持下的物理内存访问 - 33 -](#_Toc532238439)

[7.6 hello进程fork时的内存映射 - 34 -](#_Toc532238440)

[7.7 hello进程execve时的内存映射 - 34 -](#_Toc532238441)

[7.8 缺页故障与缺页中断处理 - 34 -](#_Toc532238442)

[7.9动态存储分配管理 - 35 -](#_Toc532238443)

[7.10本章小结 - 37 -](#_Toc532238444)

[第8章 hello的IO管理 - 38 -](#_Toc532238445)

[8.1 Linux的IO设备管理方法 - 38 -](#_Toc532238446)

[8.2 简述Unix IO接口及其函数 - 38 -](#_Toc532238447)

[8.3 printf的实现分析 - 39 -](#_Toc532238448)

[8.4 getchar的实现分析 - 41 -](#_Toc532238449)

[8.5本章小结 - 42 -](#_Toc532238450)

[结论 - 42 -](#_Toc532238451)

[附件 - 44 -](#_Toc532238452)

[参考文献 - 45 -](#_Toc532238453)

# 第1章 概述

## 1.1 Hello简介

程序员通过编辑器创建并保存一个文本文件，文件名为hello.c，这叫源程序，源程序是以字节序列（ASLL字符）的方式储存在文件中的。

在预处理阶段，将头文件的内容插入到程序文本中，生成hello.i。

在编译阶段，编译器把hello.i文件翻译成hello.s，这是汇编语言程序文件，由低级机器语言指令组成。

在汇编阶段，汇编器把hello.s翻译成机器语言指令，并把指令打包成一种可重定位目标程序，并将结果保存在hello.o中，此时hello.o是一个二进制文件。

在链接阶段，链接器把hello.o与它调用的目标文件相合并，得到hello文件，这是可执行目标文件，可以被加载到内存中，直接执行。



图1-1 hello程序流程图

hello被存放在磁盘上，要想要它被执行，要在linux系统的shell（终端）输入./hello，shell认定他不是内核程序，是一个可执行文件，为其fork产生进程，然后Hello就成为了一个进程，完成了P2P: From Program to Process；

shell为其execve，mmap将其的代码段，数据段等加入到虚拟内存中，然后进行内存访问，在经历了缺页之后成功访问了代码段、数据段的内存，然后存储分级系统将其一步一步往上传输，在经历了缓存不命中之后成功读取了Hello, 然后在Hello运行完之后，shell令其父进程回收Hello，将他的数据以及一切痕迹全部清除干净实现了O2O: From Zero-0 to Zero-0;

## 1.2 环境与工具

ubuntu64位 vmware虚拟机 gcc codeblocks gedit

## 1.3 中间结果

列出你为编写本论文，生成的中间结果文件的名字，文件的作用等。

|  |  |  |
| --- | --- | --- |
| 中间结果文件 | 文件作用 | 使用时间 |
| Hello.i | 预处理得到的中间文件 | 2-预处理 |
| Hello.s | 编译得到的ASLL汇编语言文本文件 | 3-编译 |
| Hello.o | 汇编得到的二进制可重定位文件 | 4-汇编 |
| Hello.asm | Hello.o反汇编文件 | 4-汇编 |
| Hello.elf | Hello.o的ELF文件 | 4-汇编 |
| Hello | 链接得到的可执行文件 | 5-链接 |
| Hello\_asm | Hello反汇编文件 | 5-链接 |
| Hello\_elf | Hello的ELF文件 | 5-链接 |

## 1.4 本章小结

本章简单介绍了hello文件P2P和O2O的过程，P2P是hello从程序到进程，O2O是hello从创建到被内核回收释放。在这两个过程中，涉及到了预处理器，编译器，汇编器，链接器等协调工作，计算机系统的内核，存储，虚拟内存，I/O等多种机制。本章只是一个概述，详细的内容会在后面进行讲解。

**（第1章0.5分）**

# 第2章 预处理

## 2.1 预处理的概念与作用

预处理：在程序编译之前通过C Preprocessor（cpp）预处理器进行处理，得到另一个c程序，通常以.i作为文件扩展名。

作用：

1. 宏定义：又称为宏替换，即将#define里的东西转换为文本，带有参数的宏还需要进行参数代换。
2. 文件包含：将#include的系统头文件的源代码插入程序文本。
3. 条件编译：对于满足if条件的代码进行筛选，只有满足的代码才进行编译。
4. 删除注释

## 2.2在Ubuntu下预处理的命令

执行命令：gcc -m64 -no-pie -fno-PIC -E hello.c -o hello.i

得到：



图2-1 hello.i部分代码截图

## 2.3 Hello的预处理结果解析

Hello.i文件相比于Hello.c文件内容有20多行变为了3000多行，其中上述所说的#include头文件中包含的库已经复制到了文件中。

## 2.4 本章小结

深化了预处理的概念和作用，并在hello.c文件上进行了实践，通过观察具体的代码变化，明白了预处理的工作原理。

**（第2章0.5分）**

# 第3章 编译

## 3.1 编译的概念与作用

编译：将预处理后的文件生成汇编语言程序，生成.s文件。

作用：为不同的高级语言、不同的编译器提供了通用的输出语言——汇编语言，更贴近机器底层运行的语言。

## 3.2 在Ubuntu下编译的命令

执行：gcc -m64 -no-pie -fno-PIC -S hello.i -o hello.s

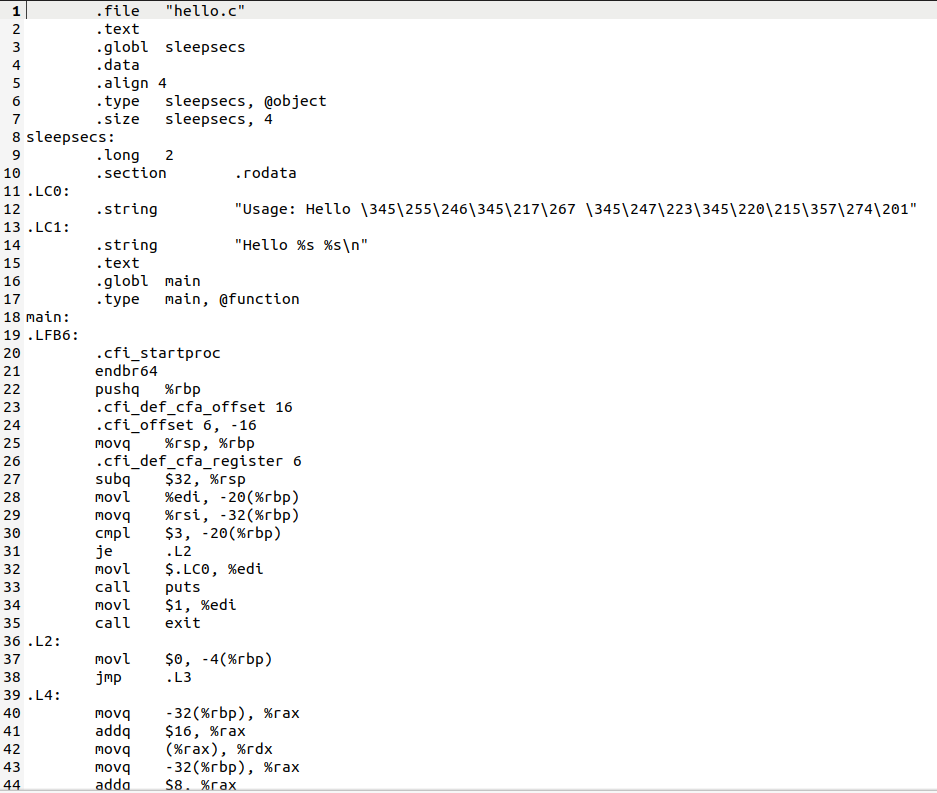


图3-1 hello.s部分代码截图

## 3.3 Hello的编译结果解析

3.3.1

int sleepsecs=2.5;

.globl sleepsecs是全局变量

.type sleepsecs是一个object类型的数据

.size sleepsecs的大小是4个字节

.long 2 sleepsecs是一个长整形的数，初始值为2。

.section .rodata sleepsecs是一个只读的数据

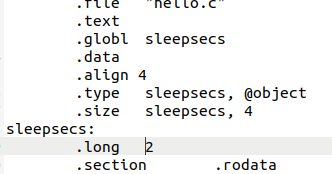


图3-2 sleepsecs汇编代码

3.3.2

main函数

.text main函数在text中定义的

.globl main函数是一个全局变量

.type main是一个函数类型的数据

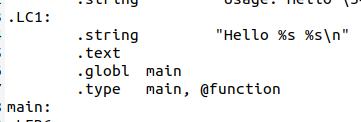


图3-3 main汇编代码

3.3.3

int i;

i是一个未初始化的局部变量，他存放在栈里面，地址是-4（%rbp），被随机赋初值0。



图3-4 int i汇编代码

3.3.4

if(argc!=3)

argc作为参数，存储在-20（%rbp）里面，利用cmpl函数与3比较，如果相等就执行.L2的代码段。

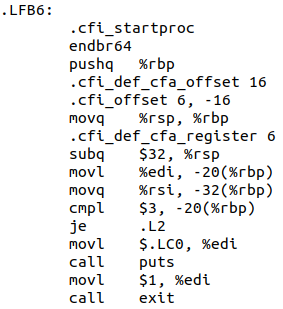


图3-5 if（argc！=3）汇编代码

3.3.5

printf("Usage: Hello 学号 姓名！\n");

Usage: Hello 学号 姓名！\n字符串被存到了.rodata的.LC0里面。



要调用printf函数，需要将.LC0移入到%edi参数寄存器，传递给要调用的puts函数。

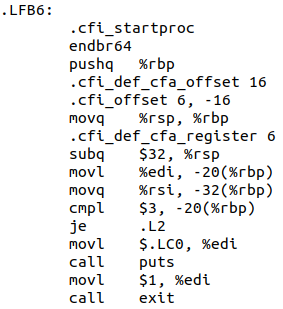


图3-5 printf函数汇编代码

3.3.6

exit(1)

将1移入参数寄存器%edi里面，call exit，调用exit函数。



图3-6 exit（1）函数汇编代码

3.3.7

for(i=0;i<10;i++)

前面说到i存储在-4（%rbp）里面，利用cmpl函数讲他与9比较，如果小于等于（jle）,则跳转到.L4代码段，.L4代码段最后有addl操作，将-4（%rbp）里面的值加一，然后执行.L3。直到jle不满足条件了，call getchar函数，进入下一阶段。

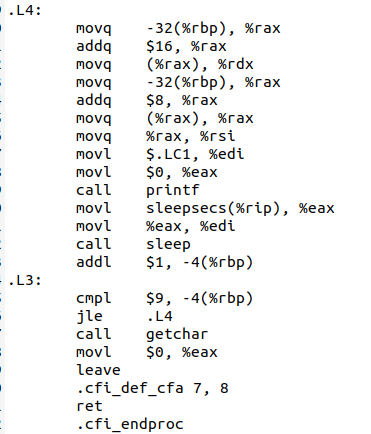


图3-7 for循环汇编代码

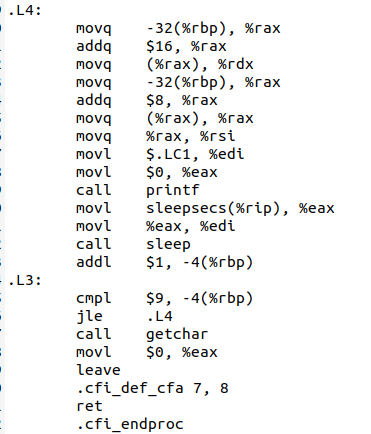
3.3.8

printf("Hello %s %s\n",argv[1],argv[2]);

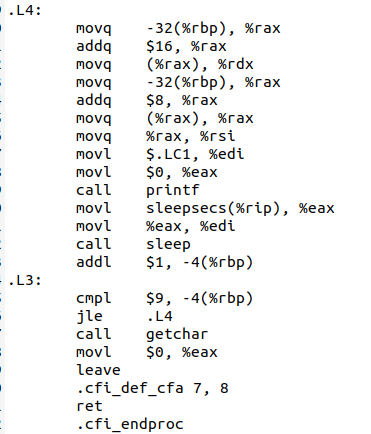
"Hello %s %s\n"字符串存储在.rodata里面，记为.LC1。



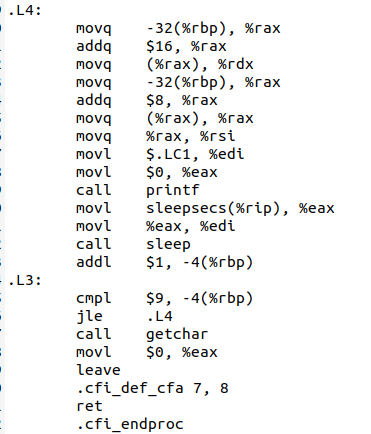
argv[1]被存储在%rdx里面



argv[2]被存储在%rax里面



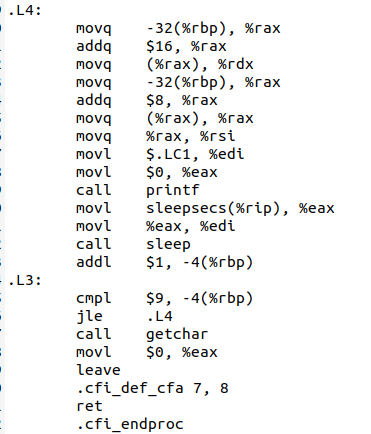
将字符串传递给%edi，调用printf函数



3.3.9

sleep(sleepsecs);

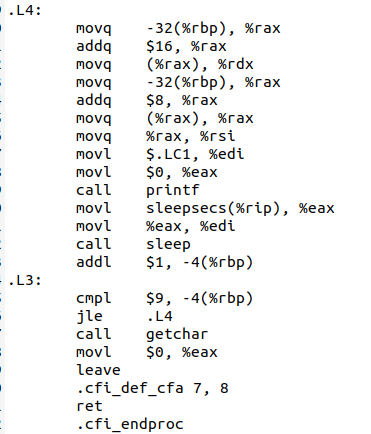
sleepsecs传递给%edi寄存器，调用sleep函数。



3.3.10

getchar();

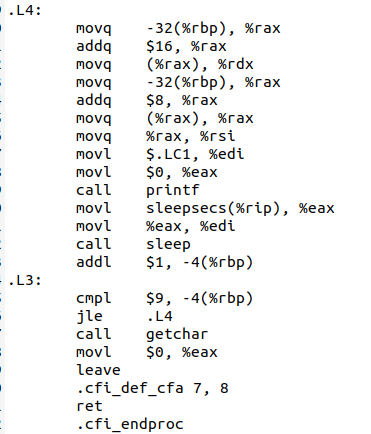
直接调用getchar函数，不需要往参数寄存器里面传递参数



3.3.11

return 0;

将0传递到%eax参数寄存器，调用ret。leave释放堆栈的空间。



## 3.4 本章小结

本章涉及到的指令全部为gun汇编程序的伪汇编指令，相比最后的汇编指令内容更为精简。通过分析汇编语言代码，熟悉了全局变量的定义，for循环，print函数等多种操作的本质，为后续的汇编和链接做准备。

**（第3章2分）**

# 第4章 汇编

## 4.1 汇编的概念与作用

汇编：把汇编语言代码翻译成机器语言代码的工程，也是从 .s 到 .o 即编译后的文件到生成机器语言二进制程序的过程。

作用：把汇编语言翻译成机器语言。把一个源程序翻译成目标程序的工作过程分为五个阶段：词法分析；语法分析；语义检查和中间代码生成；代码优化；目标代码生成。主要是进行词法分析和语法分析，又称为源程序分析，分析过程中发现有语法错误，给出提示信息。

## 4.2 在Ubuntu下汇编的命令

执行命令：gcc -m64 -no-pie -fno-PIC -c hello.s -o hello.o



图4-1 机器语言代码截图

## 4.3 可重定位目标elf格式

利用readelf -a hello.o查看hello.o的ELF格式的所有信息。（ELF文件讲解的相关博客如下https://blog.csdn.net/daide2012/article/details/73065204）

由下图ELF的头信息可知：Data是补码，小端存储；type为REL，即为可重定位文件；这个ELF header 的大小是64个字节；Section header的大小是64bytes。字符串表索引节头是13。

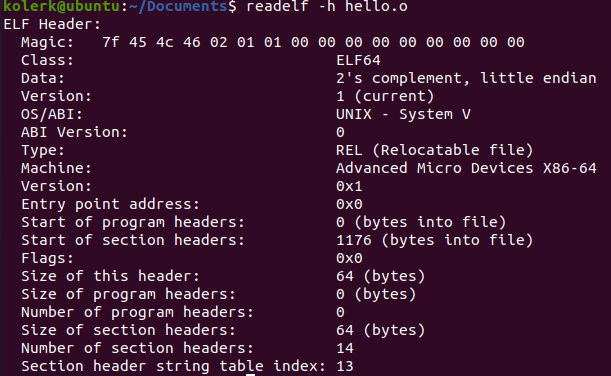


图4-2 hello的ELF头信息

Hello的节头信息如下：因为还没有链接，所以现在address（虚拟地址）都是0，offset和size指出来了各section的偏移量和大小。比如.data的offset是0xc4，大小是0x4，也就是我们定义的全局变量sleepsecs，是int类型，占用4个字节。

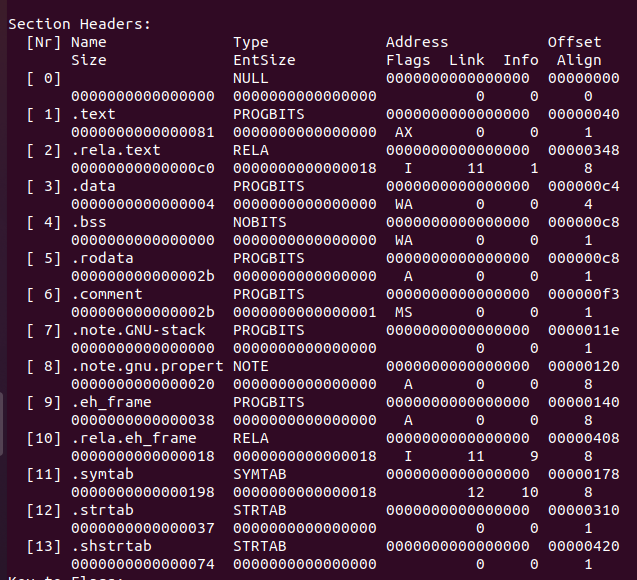


图4-3 hello的section headers信息

Hello的符号表信息如下：

Symbol table，符号表，在section header里面有详细信息。我们从Symbol table里面看到符号表一共有16 个变量。Ndx列是每个符号所在的Section编号，例如sleepsecs在第3个Section里（也就是.data），main在第一个Section里（也就是.text）各Section的编号见Section Header Table。从Bind这一列可以看出符号是GLOBAL的还是是LOCAL的。

例如：有我们自己定义的sleepsecs，type是global，全局变量，size是4个字节（int类型）。除此之外，我们调用的函数，比如main，puts，exit，printf，sleep，getchar都被记录在符号表里面。

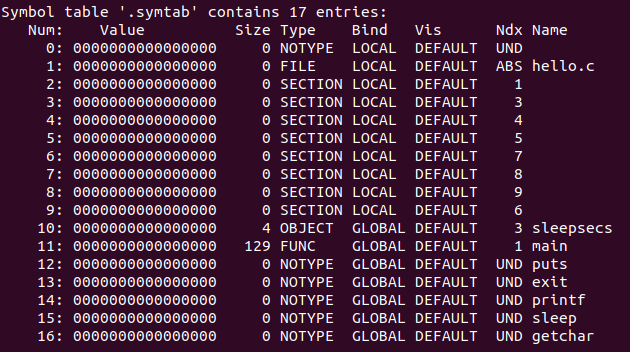


图4-4 hello的符号表信息

Hello的重定位信息如下：这部分信息是可重定位目标文件独有的，目的是告诉链接器有那部分需要重新定位。有函数，有全局变量，代码，还有我们要输出的字符串等等的定位信息。

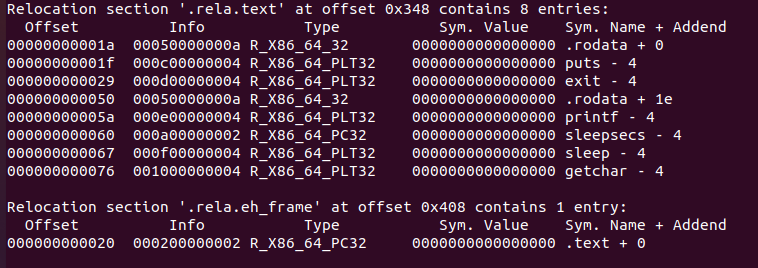


图4-5 hello的重定位信息

## 4.4 Hello.o的结果解析

objdump -d -r hello.o 分析hello.o的反汇编，并请与第3章的 hello.s进行对照分析。

总体观察图4-6后发现，除去显示格式之外两者差别不大，主要差别如下(分别举例标注在截图上)：

1.分支转移：反汇编代码跳转指令的操作数使用的不是段名称如.L2，因为段名称只是在汇编语言中便于编写的助记符，所以在汇编成机器语言之后显然不存在，而是确定的地址。

2.函数调用：在.s文件中，函数调用之后直接跟着函数名称，而在反汇编程序中，call的目标地址是当前下一条指令。这是因为hello.c中调用的函数都是共享库中的函数，最终需要通过动态链接器才能确定函数的运行时执行地址，在汇编成为机器语言的时候，对于这些不确定地址的函数调用，将其call指令后的相对地址设置为全0（目标地址正是下一条指令），然后在.rela.text节中为其添加重定位条目，等待静态链接的进一步确定。

3.全局变量访问：在.s文件中，访问rodata（printf中的字符串），使用段名称，在反汇编代码中因为rodata中数据地址也是在运行时确定，故访问也需要重定位。所以在汇编成为机器语言时，将操作数设置为全0并添加重定位条目。

|  |  |
| --- | --- |
|  |  |

图4-6 hello.o反汇编代码与hello.s对比（左hello.o反汇编代码，有hello.s）

## 4.5 本章小结

通过对汇编代码的ELF文件进行分析，深入了解了可重定位文件的结构和每部分所代表的意义，明确了反编汇语言和一般的.s文件的区别与联系，深入理解了一个程序文件的运行过程。

**（第4章1分）**

# 第5章 链接

## 5.1 链接的概念与作用

 链接是将各种代码和数据片段收集并组合成一个单一文件的过程，这个文件可被加载到内存并执行。链接可以执行于编译时，也就是在源代码被编译成机器代码时；也可以执行于加载时，也就是在程序被加载器加载到内存并执行时；甚至于运行时，也就是由应用程序来执行。链接是由叫做链接器的程序执行的。链接器使得分离编译成为可能。

## 5.2 在Ubuntu下链接的命令

执行命令：

ld -o hello -dynamic-linker /lib64/ld-linux-x86-64.so.2 /usr/lib/x86\_64-linux-gnu/crt1.o /usr/lib/x86\_64-linux-gnu/crti.o hello.o /usr/lib/x86\_64-linux-gnu/libc.so /usr/lib/x86\_64-linux-gnu/crtn.o

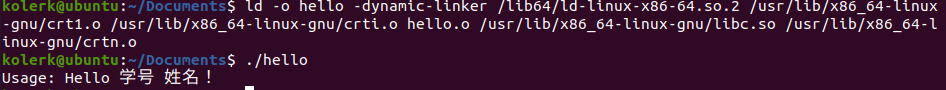


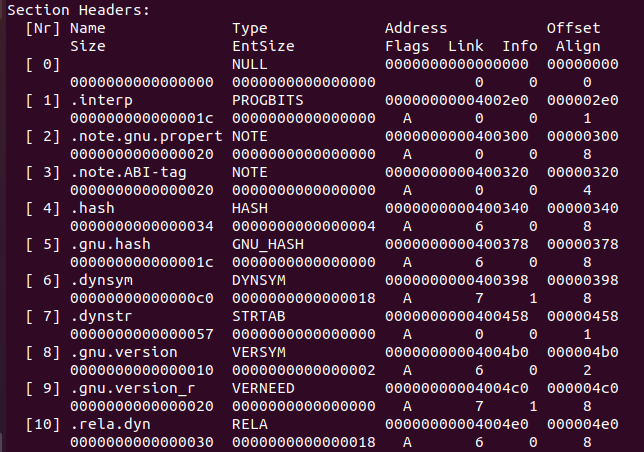
图5-1链接截图

## 5.3 可执行目标文件hello的格式

执行命令：readlf -a hello

得到Section header的信息截图如下，其中offset为各section的在程序中的地址偏移量，size为各段的大小，基本信息与可重定位文件相似，见4.3节。这里的Section header与可重定位文件有一个明显的区别就是address不再都是0了，被分配了相应的虚拟空间的地址。

大小



虚拟地址

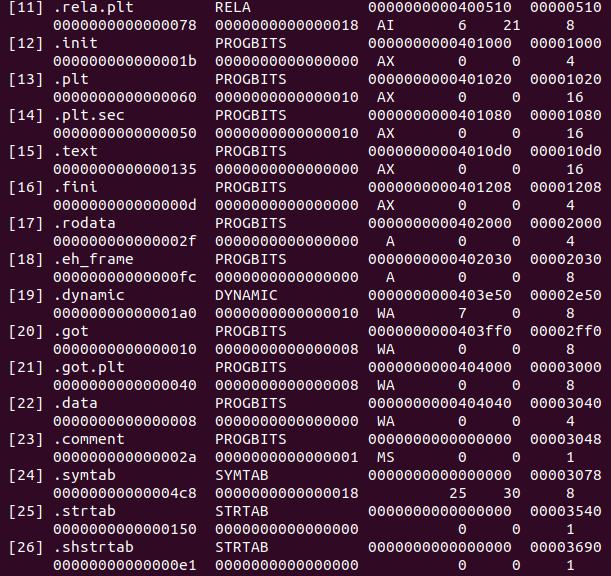


图5-2 hello的ELF文件section header

## 5.4 hello的虚拟地址空间

通过view->memory regions开业查看程序运行各个段的地址。

第一段0x400000~0x401000为1~11段的地址

第二段0x401000~0x402000为12~16段的地址

第三段0x402000~0x403000为17~28段的地址。

第四段0x403000~0x405000为19~22段的地址。

剩下的.so为共享库的地址，还有[stack]堆栈的地址。

在0x400000~0x405000段中，程序被载入，自虚拟地址0x400000开始，自0x404fff结束，这之间每个节（开始 ~ .data节）的排列即开始结束同图5.2中Address中声明。

图5-3 hello程序的虚拟内存区域



## 5.5 链接的重定位过程分析

（*以下格式自行编排，编辑时删除*）

objdump -d -r hello 分析hello与hello.o的不同，说明链接的过程。

结合hello.o的重定位项目，分析hello中对其怎么重定位的。

与hello.o反汇编文本helloo.objdump相比，在hello.objdump中多了许多节，列在下面。通过比较hello.objdump和helloo.objdump了解链接器。

1.函数个数：在使用ld命令链接的时候，指定了动态链接器为64的/lib64/ld-linux-x86-64.so.2，crt1.o、crti.o、crtn.o中主要定义了程序入口\_start、初始化函数\_init，\_start程序调用hello.c中的main函数，libc.so是动态链接共享库，其中定义了hello.c中用到的printf、sleep、getchar、exit函数和\_start中调用的\_\_libc\_csu\_init，\_\_libc\_csu\_fini，\_\_libc\_start\_main。链接器将上述函数加入。

2.函数调用：链接器解析重定条目时发现对外部函数调用的类型为R\_X86\_64\_PLT32的重定位，此时动态链接库中的函数已经加入到了PLT中，.text与.plt节相对距离已经确定，链接器计算相对距离，将对动态链接库中函数的调用值改为PLT中相应函数与下条指令的相对地址，指向对应函数。对于此类重定位链接器为其构造.plt与.got.plt。

3. rodata引用：链接器解析重定条目时发现两个类型为R\_X86\_64\_PC32的对.rodata的重定位（printf中的两个字符串），.rodata与.text节之间的相对距离确定，因此链接器直接修改call之后的值为目标地址与下一条指令的地址之差，指向相应的字符串。

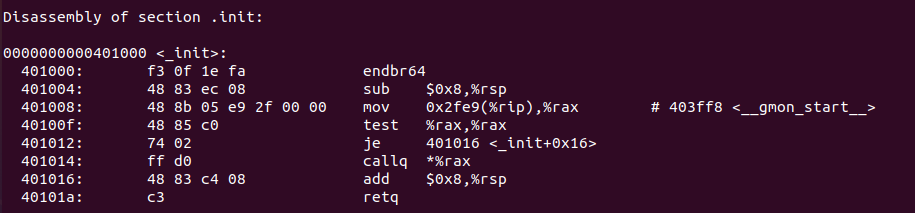


图5-4 初始化函数\_init反汇编代码

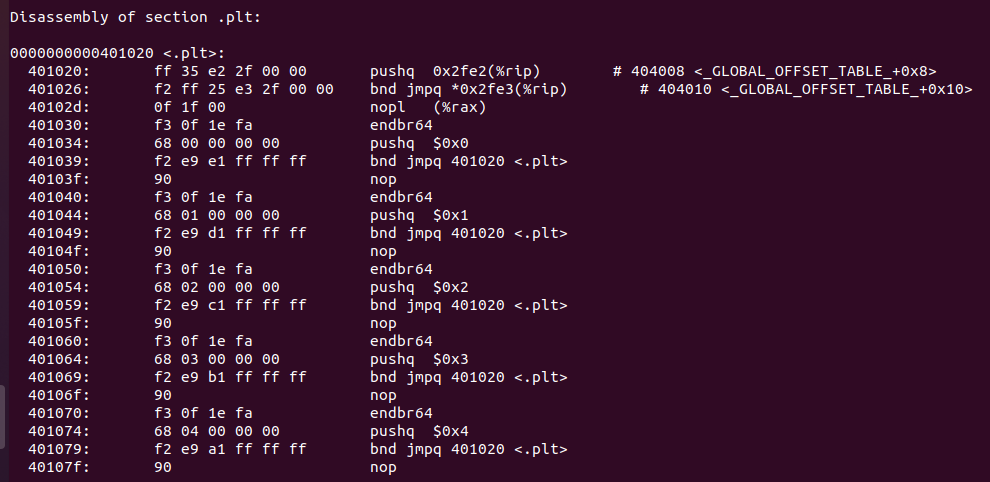


图5-5 .plt函数反汇编代码

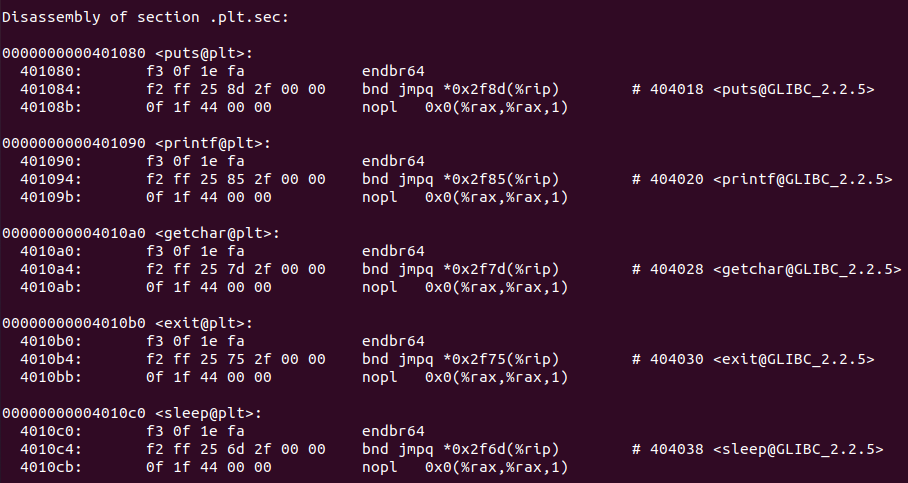
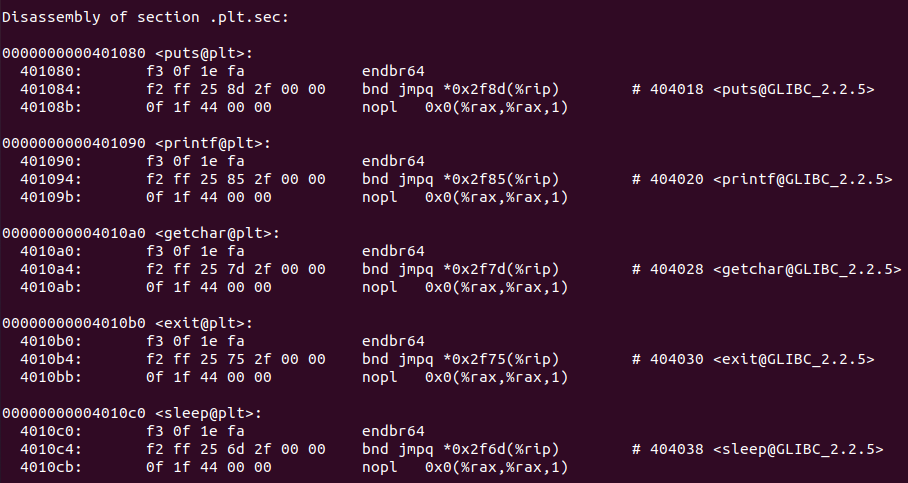


图5-6 hello调用库函数的反汇编代码

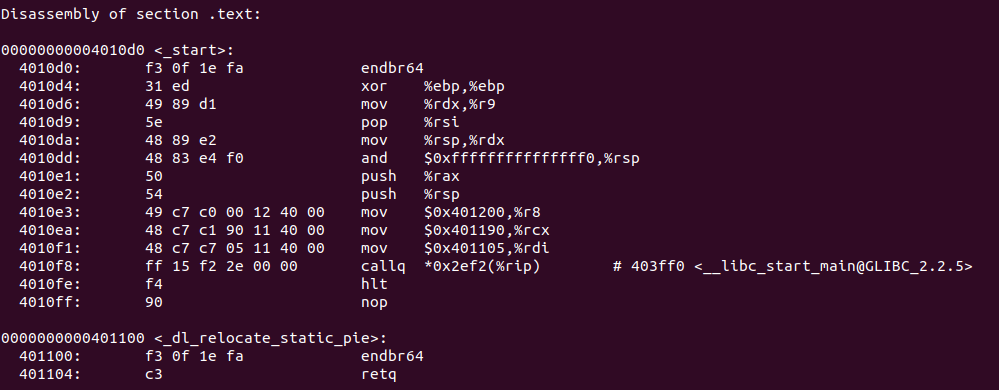


图5-7 \_start程序（调用hello.c中的main函数）反汇编代码

我们来看main函数反汇编代码，他调用的函数都已经在hello的反汇编代码中显示的定义了出来，有地址有名称，可以直接call这个函数，跳转到函数的地址。他调用的rodata也是可以直接跳转到具体地址的。

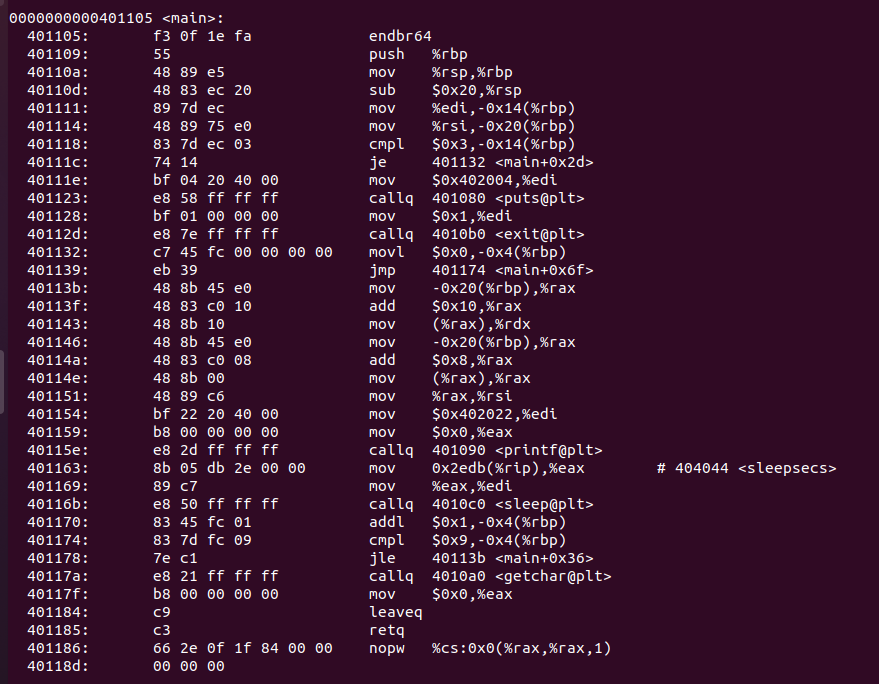
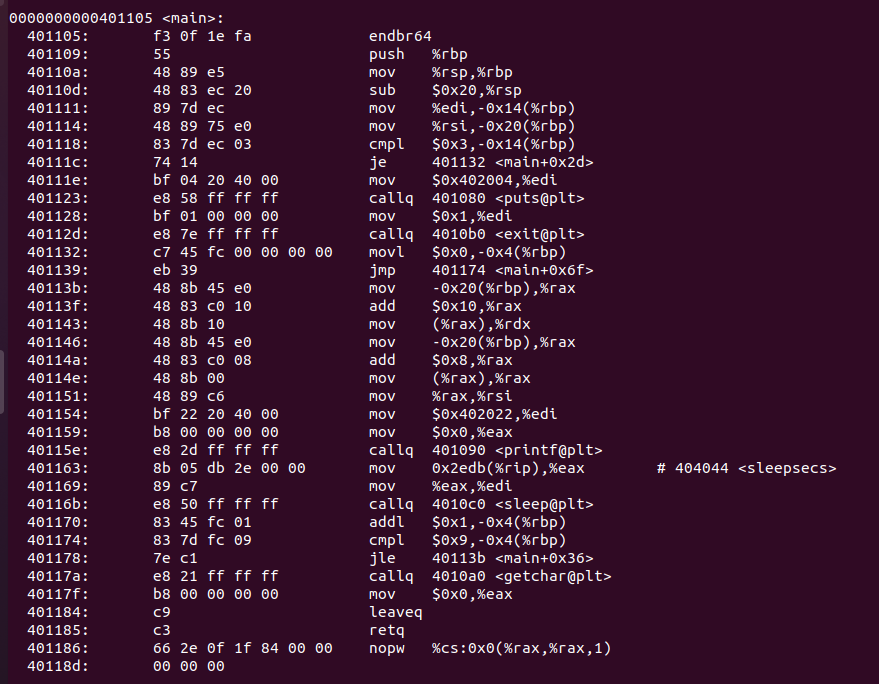


图5-8 main函数反汇编代码

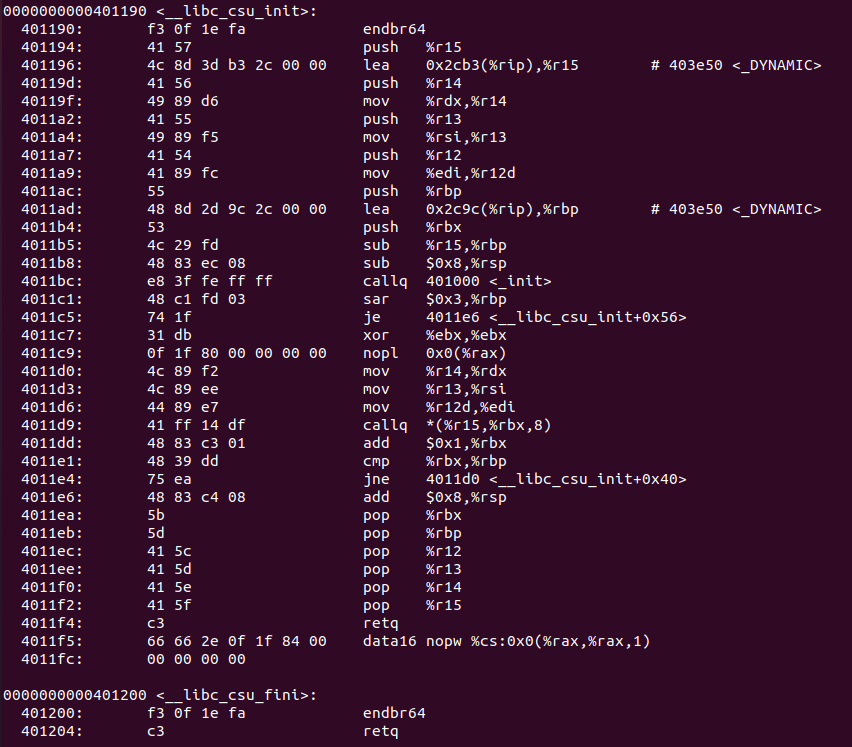
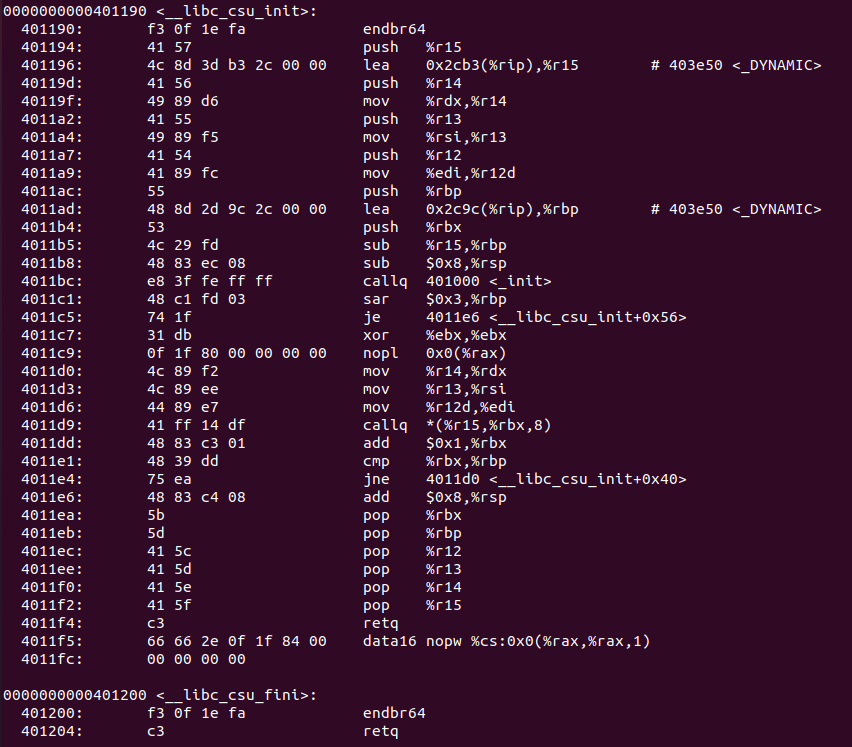
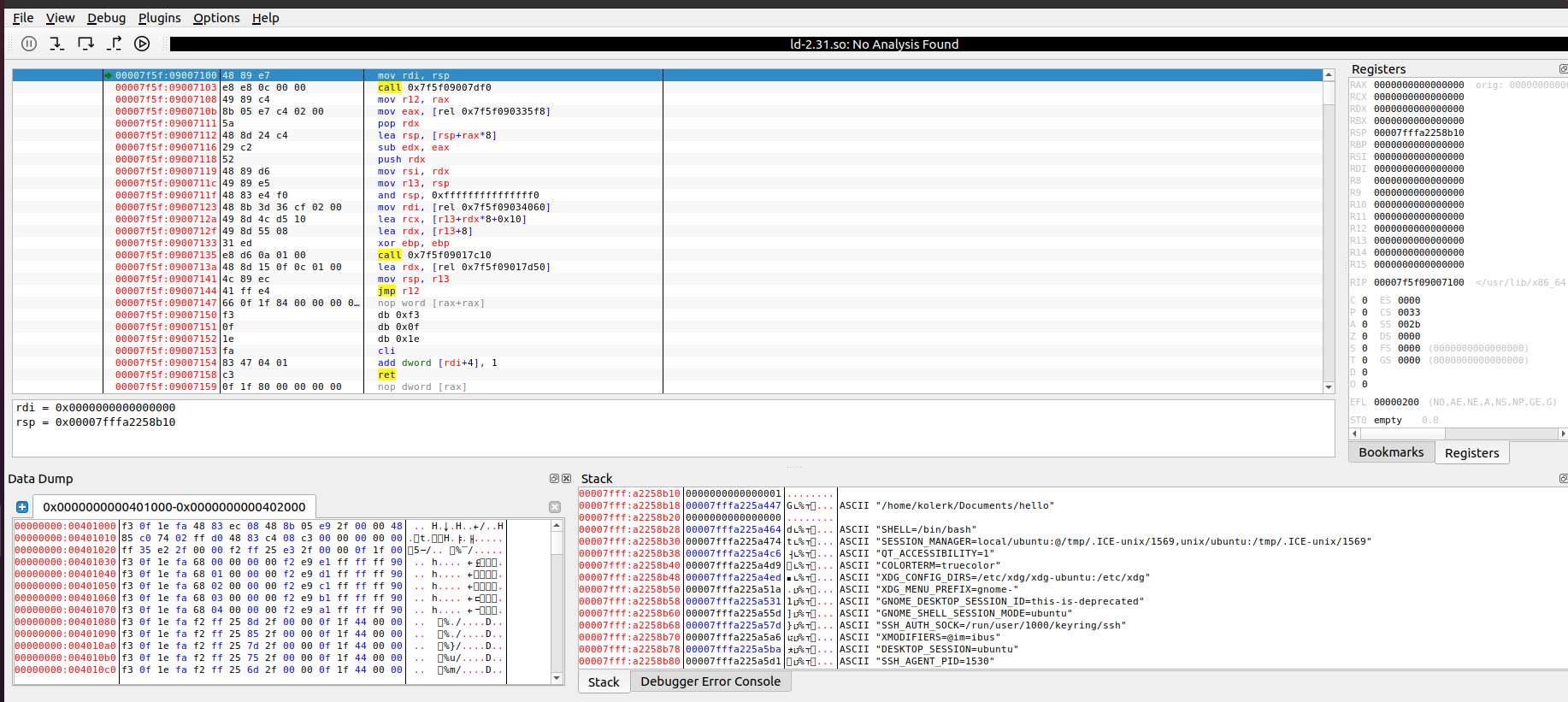


图5-9 调用的动态库的反汇编代码。

## 5.6 hello的执行流程

1. 在edb中加载hello可执行文件



1. 列出所有过程(终端输入 hello 1190301610 wangjiaqi)

|  |  |
| --- | --- |
| 子程序名 | 地址 |
| ld-3.31.so!\_dl\_start | 7efbff4d8ea0 |
| ld-3.31.so!\_dl\_init | 7efbff4e7630 |
| Hello!\_start | 400500 |
| libc-3.31.so!\_libc\_start\_main | 7efbff100ab0 |
| Hello!printf@plt(10) | 4004c0 |
| Hello!sleep@plt(10) | 4004f0 |
| Hello!getchar@plt | 4004d0 |
| Libc-3.31.so!exit | 7efbff122120 |

1. 列出所有过程（未输入）

|  |  |
| --- | --- |
| 子程序名 | 地址 |
| ld-3.31.so!\_dl\_start | 7efbff4d8ea0 |
| ld-3.31.so!\_dl\_init | 7efbff4e7630 |
| Hello!\_start | 400500 |
| libc-3.31.so!\_libc\_start\_main | 7efbff100ab0 |
| Hello！puts@plt | 4004b0 |
| Hello！exit@plt | 4004e0 |

## 5.7 Hello的动态链接分析

对于动态共享库里面的PIC函数，编译器没有办法预测函数运行时候的地址，所以需要为他天骄重定位记录，并等待动态链接器来处理。为避免运行时修改调用模块的代码段，链接器采用延迟绑定的策略。动态链接器使用过程链接表PLT和全局偏移量表GOT实现函数的动态链接。其中GOT中存在函数目标地址，PLT使用GOT中的地址跳转到目标函数。

## 5.8 本章小结

本章详细讲述了hello在执行的时候的链接操作，包括链接的概念，作用，在Unbuntu下怎么链接。并对hello的ELF文件进行了详细的解析，同时结合了虚拟地址的空间知识。通过反汇编hello文件，将它与hello.o反汇编文件进行对比，加深了对重定位的理解。最后对hello的动态链接过程进行了分析。

**（第5章1分）**

# 第6章 hello进程管理

## 6.1 进程的概念与作用

**进程是一个正在执行的程序的实例**，每一个进程都有它自己的地址空间，一般情况下，包括文本区域、数据区域、和堆栈。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储区着活动过程调用的指令和本地变量。

进程为用户提供了以下假象：我们的程序好像是系统中当前运行的唯一程序一样，我们的程序好像是**独占的使用处理器和内存**，处理器好像是无间断的执行我们程序中的指令，我们程序中的代码和数据好像是系统内存中唯一的对象。

## 6.2 简述壳Shell-bash的作用与处理流程

Shell的作用：Shell是一个用C语言编写的程序，他是用户使用Linux的桥梁。Shell 是指一种应用程序，Shell应用程序提供了一个界面，用户通过这个界面访问操作系统内核的服务。

处理流程：1.从终端读入输入的命令。2.将输入字符串切分获得所有的参数。3.如果是内置命令则立即执行。4.否则调用相应的程序为其分配子进程并运行。5.shell应该可以处理异常，比如外部信号，并对这些信号进行相应处理。

## 6.3 Hello的fork进程创建过程

终端执行./hello 1190301610 wangjiaqi命令，shell将输入的命令分割解析，当shell检测到./hello不是一个内置命令的时候，就默认这是当前目录下的一个可执行文件，利用fork创建一个子进程给hello执行自己的程序。子进程和父进程拥有完全相同的内容，但是是彼此独立的，拥有不同的PID。在hello执行期间，父进程等待hello结束，利用wait函数进行回收。

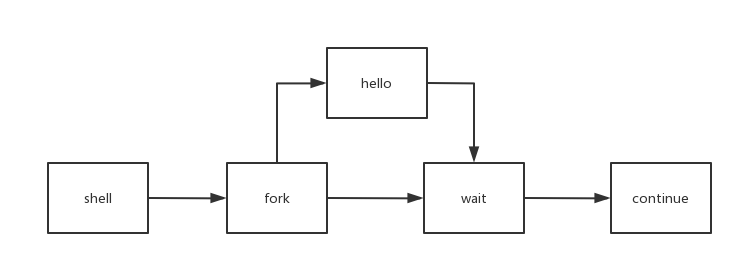


图6-1 hello的fork创建过程

## 6.4 Hello的execve过程

Shell在fork之后，子进程调用execve函数在当前进程的上下文中加载并运行一个新程序即hello程序，execve调用驻留在内存中的被称为**启动加载器**的操作系统代码来执行hello程序，加载器删除子进程现有的虚拟内存段，并创建一组新的代码、数据、堆和栈段（这些内容被放到磁盘里）。

新的栈和堆段被初始化为零，新的代码和数据段被初始化为可执行文件中的内容。最后加载器设置PC指向\_start地址，\_start最终调用hello中的main函数。

Hello的虚拟内存结构如下：

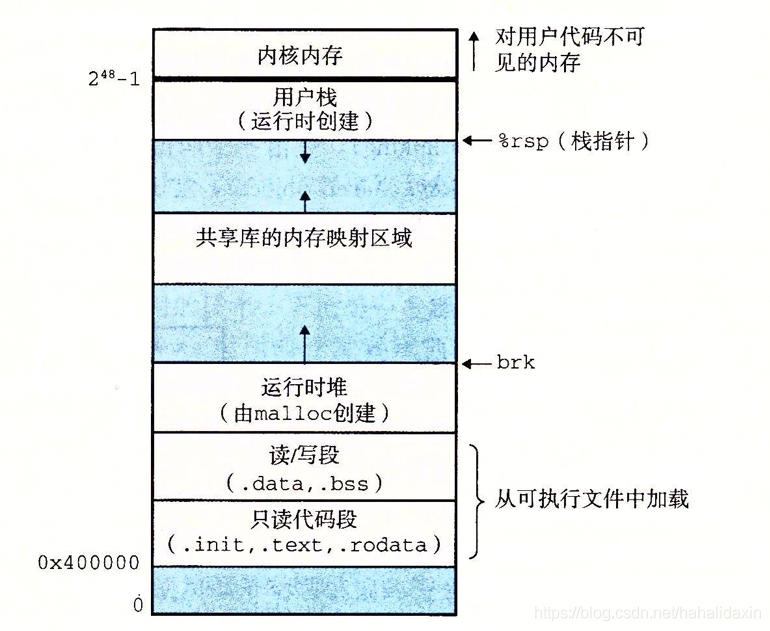


图6-2 虚拟内存映像

## 6.5 Hello的进程执行

（*以下格式自行编排，编辑时删除*）

结合进程上下文信息、进程时间片，阐述进程调度的过程，用户态与核心态转换等等。

**上下文信息**：上下文就是内核重新启动一个被抢占的进程所需要的状态，它由通用寄存器、浮点寄存器、程序计数器、用户栈、状态寄存器、内核栈和各种内核数据结构等对象的值构成。

**逻辑控制流**：一系列程序计数器PC的值的序列叫做逻辑控制流，进程是轮流使用处理器的，在同一个处理器核心中，每个进程执行它的流的一部分后被抢占（暂时挂起），然后轮到其他进程。

**时间片**：一个进程执行它的控制流的一部分的每一时间段叫做时间片。

**用户模式和内核模式：**处理器通常使用一个寄存器提供两种模式的区分，该寄存器描述了进程当前享有的特权，当没有设置模式位时，进程就处于用户模式中，用户模式的进程不允许执行特权指令，也不允许直接引用地址空间中内核区内的代码和数据；设置模式位时，进程处于内核模式，该进程可以执行指令集中的任何命令，并且可以访问系统中的任何内存位置。

**上下文切换：**上下文切换是由内核中调度器完成的，当内核调度新的进程运行后，它就会抢占当前进程，并进行1）保存以前进程的上下文。2）恢复新恢复进程被保存的上下文。3）将控制传递给这个新恢复的进程 。来完成上下文切换。

例如，当hello在用户模式执行完printf之后，调用sleep进入内核模式，休眠2秒，这期间，hello从运行队列进入等待队列，由内核统一调度进行上下文切换，安排别的进程运行，2秒时间结束，进行上下文切换，由别的进程变成hello进程。如此反复执行10次。

当调用getchar函数时，触发异常syscall，用户模式转变为内核模式，调用read处理键盘传来的内容。需要注意的是，由于从键盘缓冲区到内存耗时长，在传输键盘内容期间，内核又可以进行上下文切换，调度别的进程来占用CPU等资源，当完成键盘缓冲区到内存的数据传输时，发出一个信号给内核，内核进行上下文切换，重新执行hello。

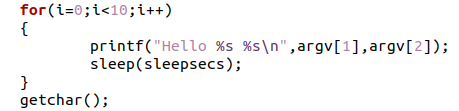


图6-3 hello进程分析代码截图

## 6.6 hello的异常与信号处理

Ctrl-C命令，hello被终止，如下图。



图6-4 Ctrl-C

Ctrl-z命令，hello被暂停，利用ps查看，还可以看到hello仍在进程里面。

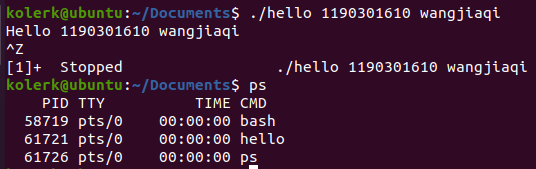


图6-5 Ctrl-z后运行ps

Ctrl-z命令，hello被暂停，利用jobs查看，还可以看到hello状态为stopped。

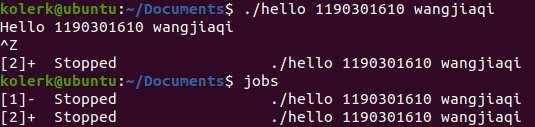


图6-6 Ctrl-z后运行jobs

Ctrl-z命令，hello被暂停，执行fg，hello继续执行，jobs里面没有hello，hello被父进程回收。

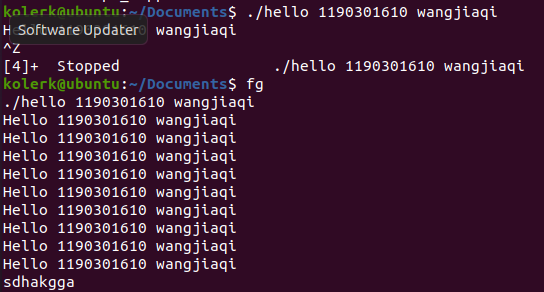


图6-7 Ctrl-z后运行fg

## 6.7本章小结

本章分析了hello在进程里面的过程，首先是shell执行fork命令，创建一个子进程，execve加载hello。然后分析hello的执行过程，内核和用户模式不断切换，内核调度一切资源。当遇到异常与信号的时候，观察hello的运行变化，深入了解linux的异常控制流的处理机制。

**（第6章1分）**

# 第7章 hello的存储管理

## 7.1 hello的存储器地址空间

**逻辑地址**：汇编语言中的地址都是逻辑地址。逻辑地址分为两部分：选择符和偏移量。hello反汇编代码里面的地址都是逻辑地址。如下图的0x402004就是逻辑地址。

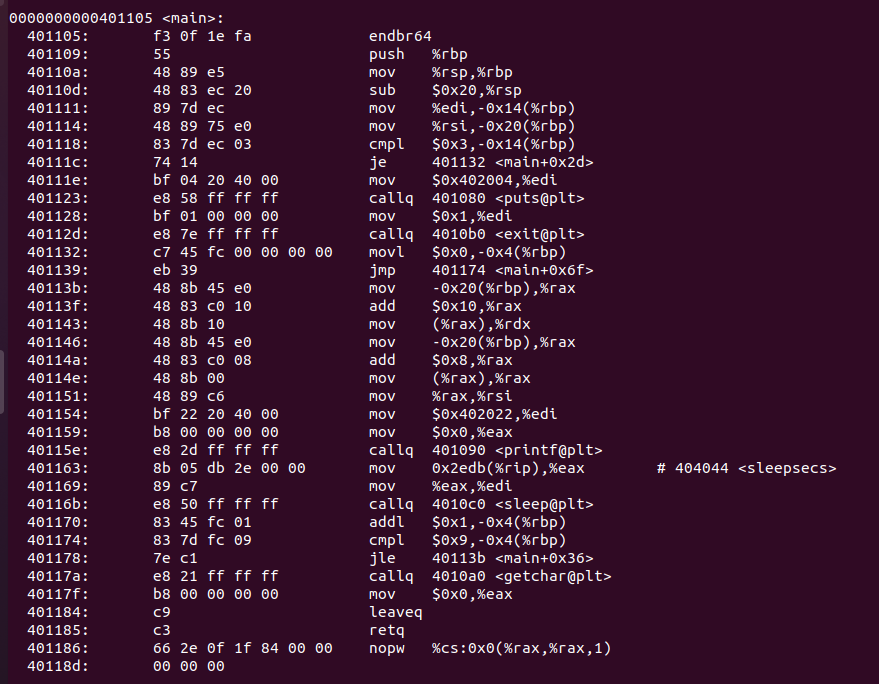


图7-1 hello可执行文件的反汇编代码

**线性地址（虚拟地址）**：逻辑地址经过段机制后转化为线性地址。Linux中逻辑地址等于线性地址。因为Linux所有的段（用户代码段、用户数据段、内核代码段、内核数据段）的线性地址都是从 0x0 开始，长度4G，而线性地址=逻辑地址+ 0x0，也就是说逻辑地址等于线性地址了。

**物理地址**：CPU通过地址总线的寻址，真实的存储在计算机系统的物理内存对应地址。这个概念最好理解。虚拟地址通过页表的方式通过MMU映射到物理地址中。

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

段式管理，是指把一个程序分成若干个段进行存储，每个段都是一个逻辑实体，它的产生是与程序的模块化直接有关的。段式管理是通过段表进行的，它包括段号或段名、段起点、装入位、段的长度等。此外还需要主存占用区域表、主存可用区域表。  
 线性地址是由逻辑地址加上隐含的DS 数据段的基地址而得到。也就是说，逻辑地址就是在DS中的偏移量。线性地址的基本组成方式是段号+段内偏移地址。

在x86保护模式下，段的信息（段基线性地址、长度、权限等）即段描述符占8个字节，而段寄存器只有2字节，段信息无法直接存放在段寄存器中。Intel的设计是段描述符集中存放在GDT或LDT中，而段寄存器存放的是段描述符在GDT或LDT内的索引值(index)。

首先给定一个完整的逻辑地址[段选择符：段内偏移地址]，接下来：

1.看段选择描述符中的T1字段是0还是1，可以知道当前要转换的是GDT中的段，还是LDT中的段，再根据指定的相应的寄存器，得到其地址和大小，我们就有了一个数组了。

2.拿出段选择符中的前13位，可以在这个数组中查找到对应的段描述符，这样就有了Base，即基地址就知道了。

3.把基地址Base+Offset,就是要转换的下一个阶段的地址。

## 7.3 Hello的线性地址到物理地址的变换-页式管理

线性地址（虚拟地址VA）到物理地址（PA）之间的转换通过分页机制完成。分页机制简单来说是对虚拟内存进行分页，并利用页表映射物理页。

如下图所示，当一个访问虚拟内存地址的请求发送给MMU的时候，MMU就会翻译这个地址，找到页表中对应的特定条目。

每个页表条目由一个有效位和物理页号（PPN）组成（还有其他的一些权限位），当有效位是1时，表明虚拟地址的页面储存在物理页面中，访问该条目的PPN，结合页偏移量（VPO），就可以知道页在物理内存中的地址，直接进行访问即可；当有效位是0时，这个虚拟地址的页面不在存储器中，触发缺页异常，内核来进行缺页异常处理，详见7.8。

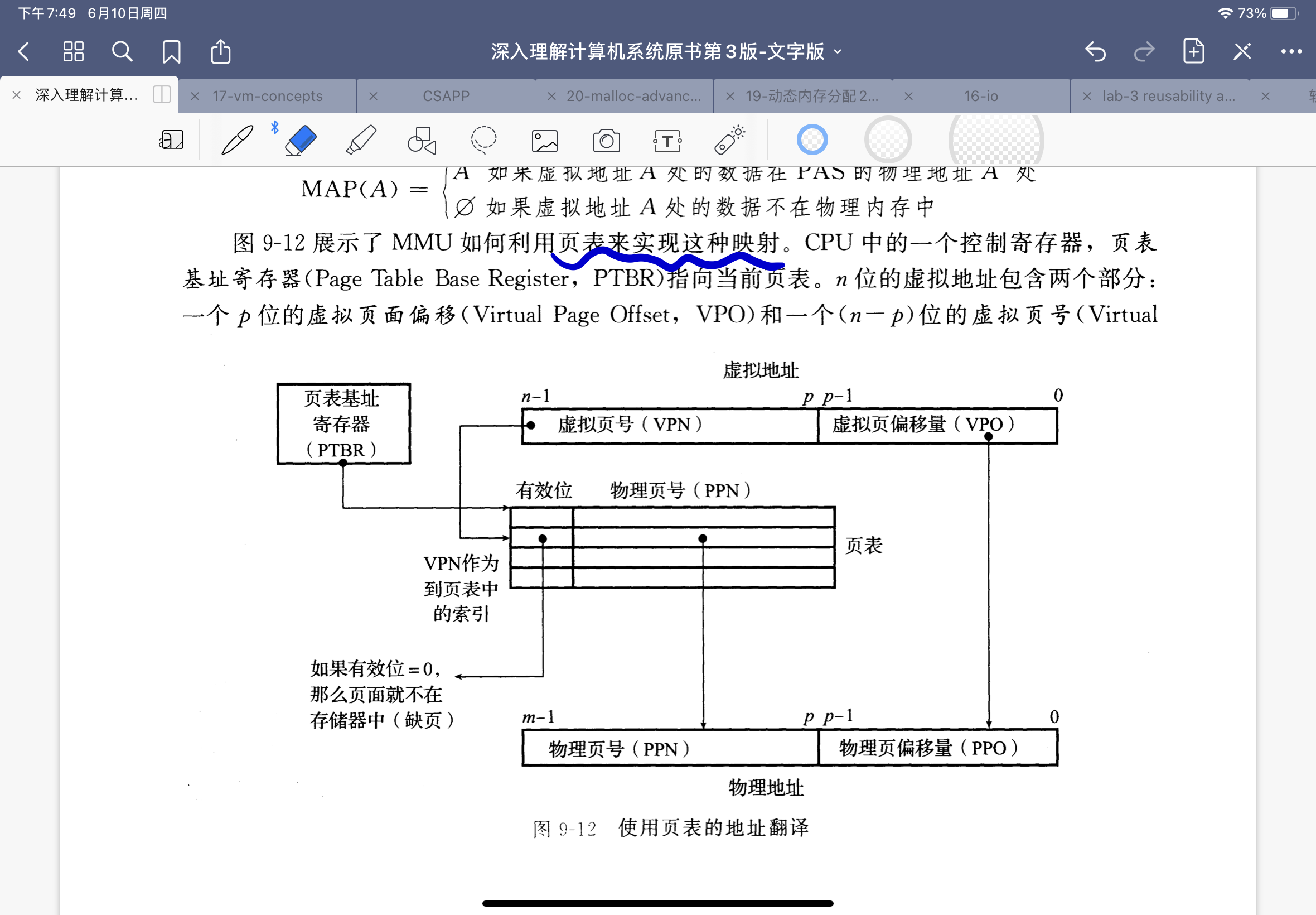


图7- 页表地址翻译

Linux系统虚拟内存组织形式如图所示：

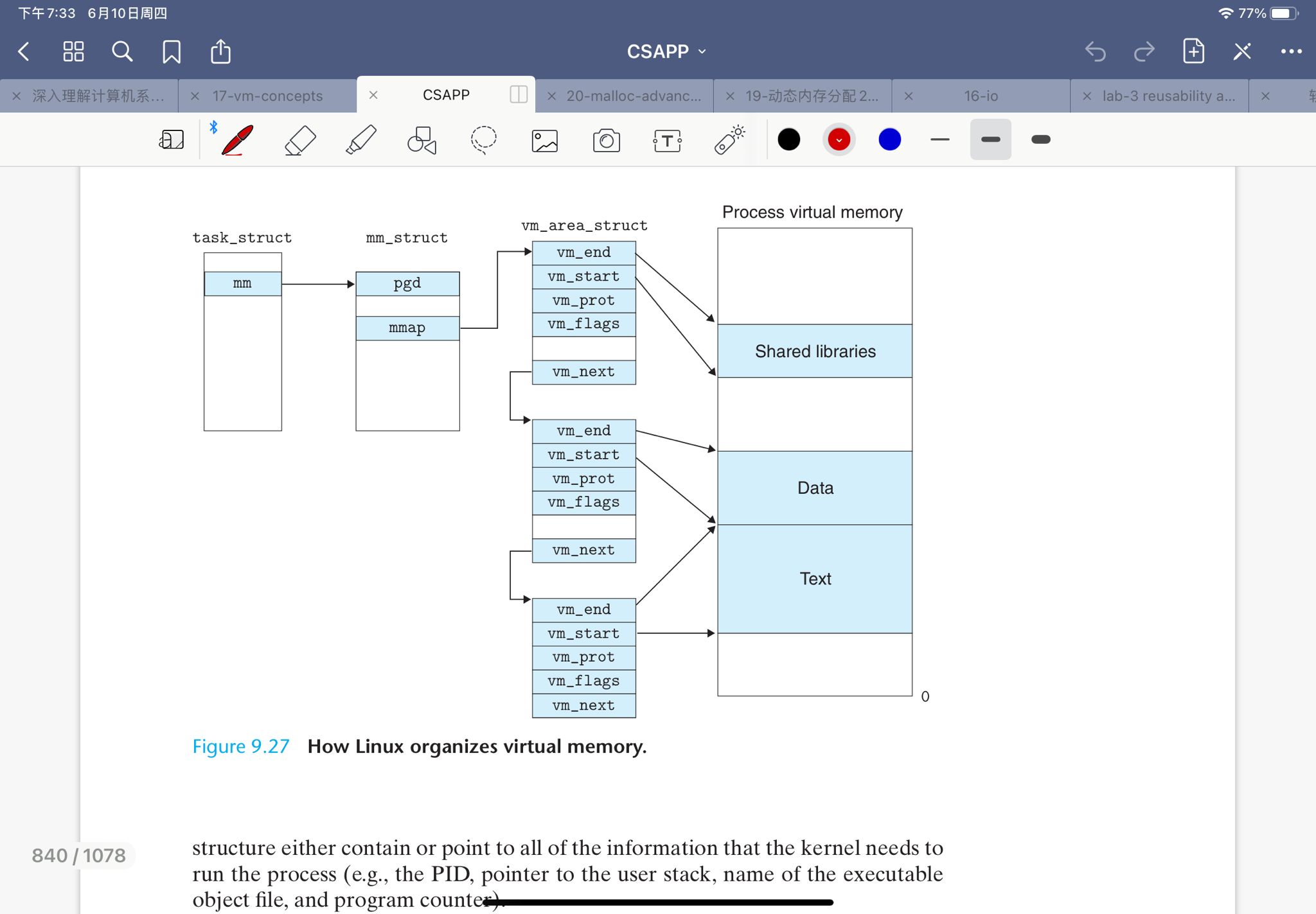


图7- 虚拟内存组织形式

Linux将虚拟内存组织成一些段的集合，一个段就是已经存在的（已分配的）虚拟内存的连续片。不存在的虚拟内存不保存在段中。

内核为hello进程维护一个段的任务结构即图中的task\_struct，它存储了内核运行该进程的所有信息（PID，堆栈指针等），其中一个条目指向mm\_struct，它描述了虚拟内存的当前状态，pgd指向第一级页表的基地址，mmap指向一个vm\_area\_struct的链表，一个vm\_area\_struct链表条目描述了虚拟地址空间的一个区域，链表相连指出了hello进程虚拟内存中的所有段。

vm\_start：指向这个区域开始处。

vm\_end：指向这个区域结束处。

vm\_prot：描述这个区域内包含的所有页的读写许可权限。

vm\_flags：描述这个区域内的页面是与其他进程共享的，还是私有的，还有一些其他信息。

## 7.4 TLB与四级页表支持下的VA到PA的变换

TLB可以理解为页表的一个小的，虚拟寻址的高速缓存（类似cache是内存的一个缓存），用于组选择和行匹配的索引和标记字段是从虚拟地址的页号中提取出来的，如下图所示，如果TLB有2^t组，TLB索引（TLBI）由VPN的t个低位组成的，TLB标记（TLBG）是由VPN中剩余的位组成的。

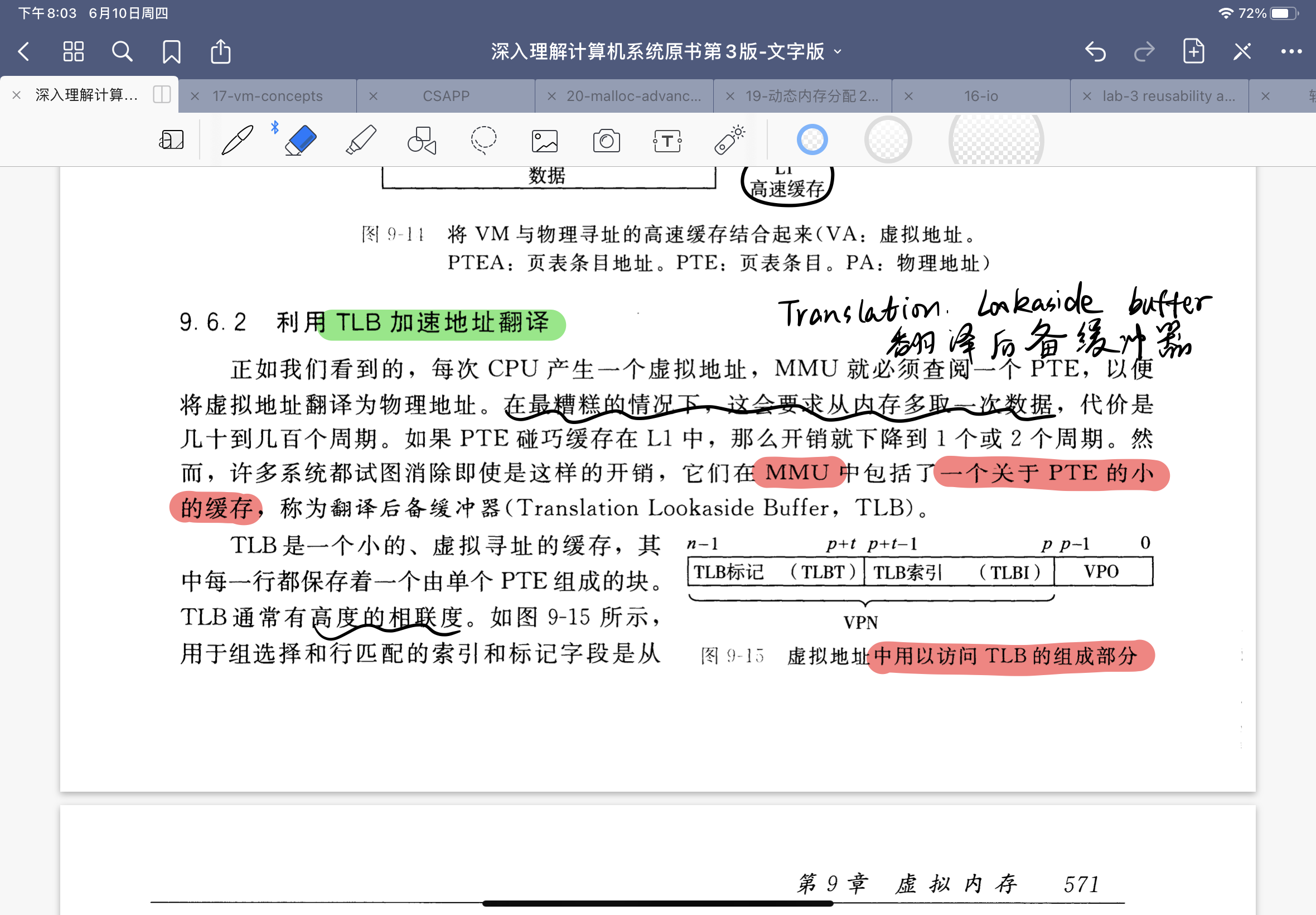


图7- TLB寻址

为了减少页表在内存空间的占用，引入了多级页表的机制，k级页表示意图如下。如果虚拟内存不存在，则不分配页表来记录这段内存。

1级页表指向2级页表，2级页表存储的是3级页表基地址，只有最后一级页表的内容是PTE和物理页号。



图7- K级页表示意图

前提如下：虚拟地址空间48位，物理地址空间52位，页表大小4KB，4级页表。TLB 4路16组相联。CR3指向第一级页表的起始位置（上下文一部分）。

解析前提条件：由一个页表大小4KB，一个PTE条目8B，共512个条目，使用9位二进制索引，一共4个页表共使用36位二进制索引，所以VPN共36位，因为VA 48位，所以VPO 12位；因为TLB共16组，所以TLBI需4位，因为VPN 36位，所以TLBT 32位。

如图 ，CPU产生虚拟地址VA，VA传送给MMU，MMU使用前36位VPN作为TLBT（前32位）+TLBI（后4位）向TLB中匹配，如果命中，则得到PPN（40bit）与VPO（12bit）组合成PA（52bit）。如果TLB中没有命中，MMU向页表中查询，CR3确定第一级页表的起始地址，VPN1（9bit）确定在第一级页表中的偏移量，查询出PTE，如果在物理内存中且权限符合，确定第二级页表的起始地址，以此类推，最终在第四级页表中查询到PPN，与VPO组合成PA，并且向TLB中添加条目。 如果查询PTE的时候发现不在物理内存中，则引发缺页故障。如果发现权限不够，则引发段错误。



## 图7- TLB与四级页表支持下的VA到PA的变换

## 7.5 三级Cache支持下的物理内存访问

前提：只讨论L1 Cache的寻址细节，L2与L3Cache原理相同。L1 Cache是8路64组相联。块大小为64B。

解析前提条件：因为共64组，所以需要6bit CI进行组寻址，因为共有8路，因为块大小为64B所以需要6bit CO表示数据偏移位置，因为VA共52bit，所以CT共40bit。

在上一步中我们已经获得了物理地址VA，使用CI进行组索引，每组8路，对8路的块分别匹配CT（前40位）如果匹配成功且块的valid标志位为1，则命中（hit），根据数据偏移量CO（后六位）取出数据返回。

如果没有匹配成功或者匹配成功但是标志位是1，则不命中（miss），向下一级缓存中查询数据（L2 Cache->L3 Cache->主存）。查询到数据之后，一种简单的放置策略如下：如果映射到的组内有空闲块，则直接放置，否则组内都是有效块，产生冲突（evict），则采用最近最少使用策略LFU进行替换。



## 图7- 三级Cache支持下的物理内存访问

## 7.6 hello进程fork时的内存映射

当shell进程调用fork函数时，内核为新进程创建虚拟内存，子进程和父进程拥有完全相同的内容，但是是彼此独立的，拥有不同的PID。它创建了当前进程的mm\_struct、区域结构和页表的原样副本，并将这两个进程的每个页面都标记为只读。

## 7.7 hello进程execve时的内存映射

execve函数调用驻留在内核区域的启动加载器代码，在当前进程中加载并运行包含在可执行目标文件hello中的程序，用hello程序有效地替代了当前程序。加载并运行hello需要以下几个步骤：

1.删除已存在的用户区域，删除当前进程虚拟地址的用户部分中的已存在的区域结构。

2.映射私有区域，为新程序的代码、数据、bss和栈区域创建新的区域结构，所有这些新的区域都是私有的、写时复制的。代码和数据区域被映射为hello文件中的.text和.data区，bss区域是请求二进制零的，映射到匿名文件，其大小包含在hello中，栈和堆地址也是请求二进制零的，初始长度为零。

3.映射共享区域， hello程序与共享对象libc.so链接，libc.so是动态链接到这个程序中的，然后再映射到用户虚拟地址空间中的共享区域内。

4.设置程序计数器（PC），execve做的最后一件事情就是设置当前进程上下文的程序计数器，使之指向代码区域的入口点。

## 7.8 缺页故障与缺页中断处理

这部分其实在之前的6.6中已经提到了，对于缺页进程是当做一个异常来处理的，此时进入内核模式，将缺页的虚拟地址写入物理内存中，如果有空闲的物理页，则直接写入，并将PPN传回页表即可，如果没有空闲的物理页，则选择一个牺牲页，如果这个牺牲页面被修改过，那么就将它交换出去，换入新的页面，更新页表。然后内核模式转为用户模式，重新执行刚刚的访存指令。

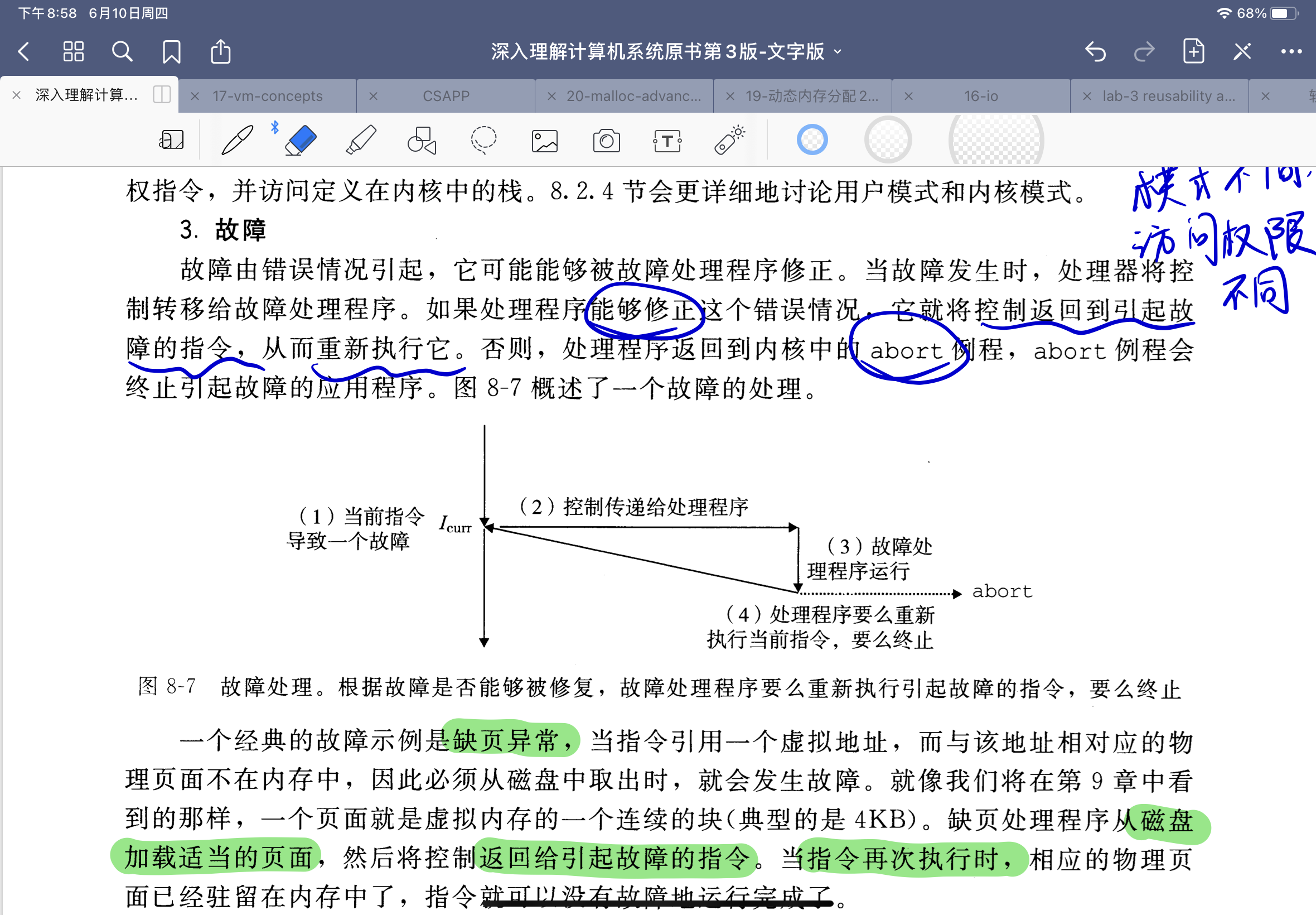


图7- 缺页故障处理

## 7.9动态存储分配管理

动态内存分配器维护着一个进程的虚拟内存区域，称为堆。分配器将堆视为一组不同大小的块的集合来维护。每个块就是一个连续的虚拟内存片，要么是已分配的，要么是空闲的。一个已分配的块保持已分配状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。

分配器分为两种：显式分配器、隐式分配器。显式分配器：要求应用显式地释放任何已分配的块，例如malloc（）函数。隐式分配器（垃圾收集器）：要求分配器检测一个已分配块何时不再使用，那么就释放这个块；自动释放未使用的已经分配的块的过程叫做垃圾收集。

**隐式空闲链表：**所谓隐式空闲链表，对比于显式空闲链表，代表并不直接对空闲块进行链接，而是将对内存空间中的所有块组织成一个大链表，其中Header和Footer中的block大小间接起到了前驱、后继指针的作用。

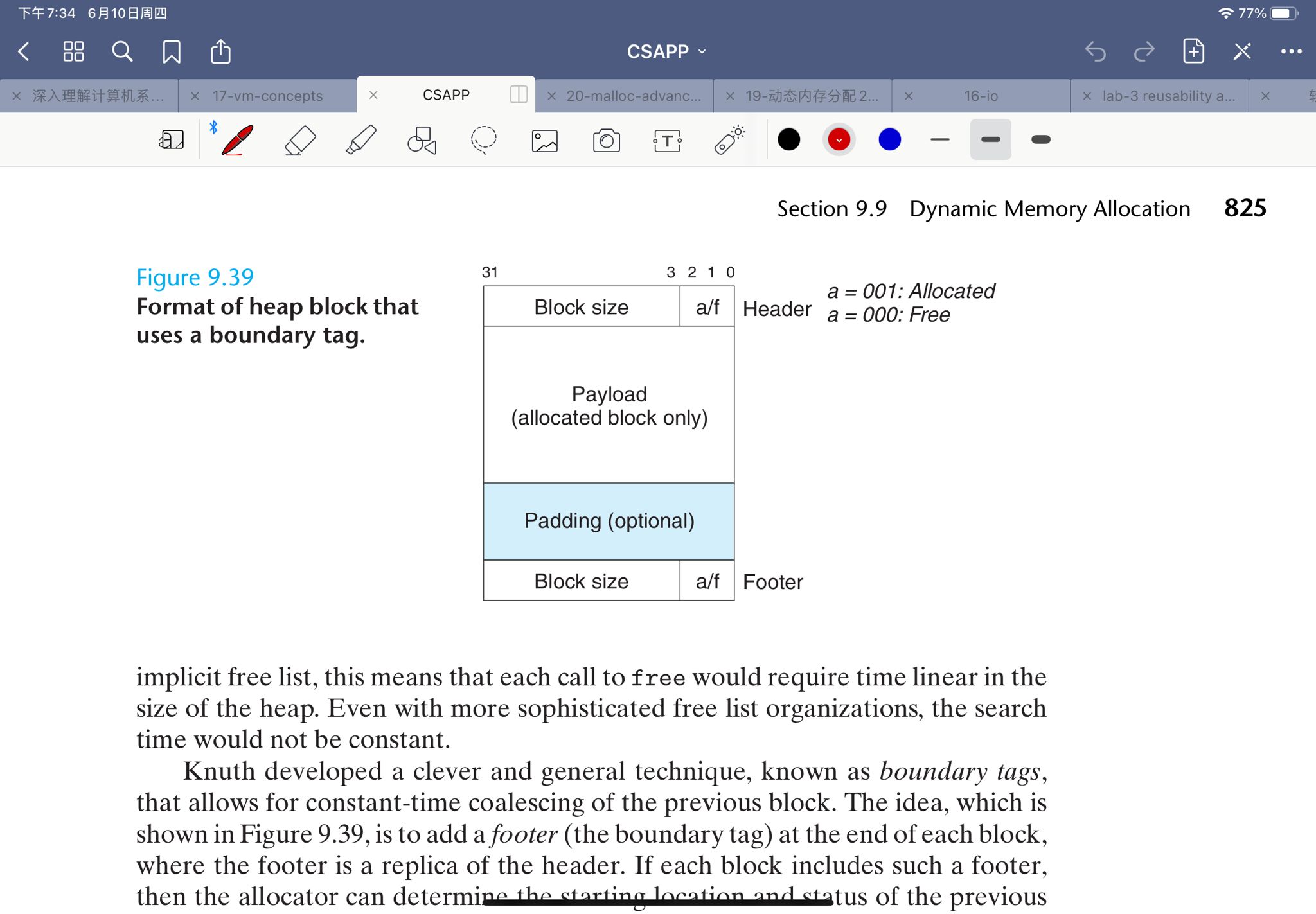


图7- 隐式空闲链表结构

涉及到的操作包括查找空闲块，malloc一个块，合并空闲块等等。其中比较复杂的就是合并空闲块这个操作，因为合并空闲块，不仅要看此块之后是不是空闲块，还要看之前是不是空闲块，这也是加入头部和角标的原因之一。对于此块前后是否为空闲块，分为以下四种情况，对每种情况要分别考虑。

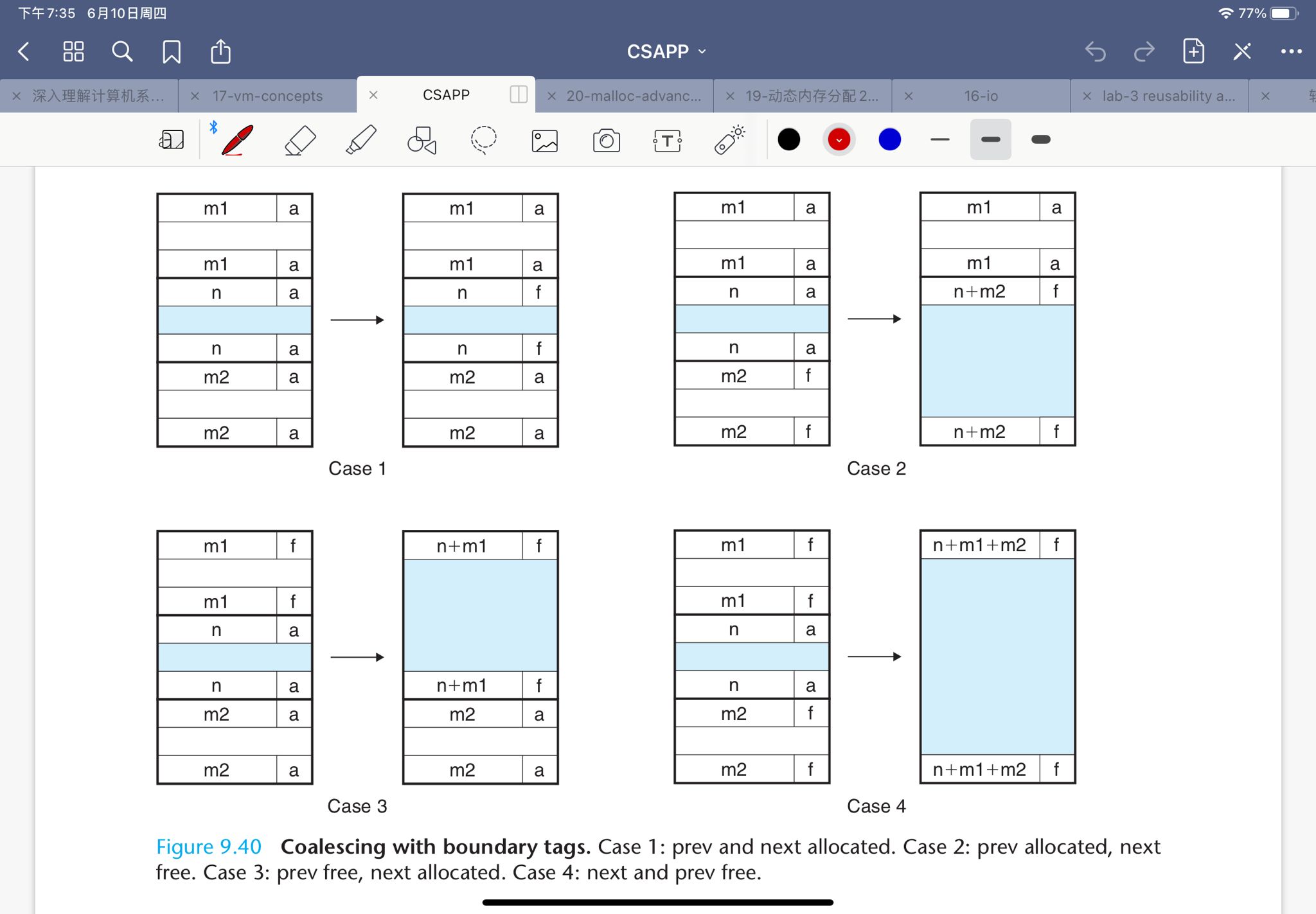


图7- 隐式空闲链表合并空闲块

## 7.10本章小结

本章深入解析进程的存储管理。hello运行的时候，所需要的一切数据代码共享库都要在物理内存里面，这就要求在磁盘里面的虚拟内存到物理内存之间有一个较好的映射关系，其中涉及到了地址变换，TLB，页表等多种操作，还要处理缺页的异常，这是一系列复杂的操作。hello还调用了动态内存分配，利用堆和显示分配器来完成自己的printf功能。

**（第7章 2分）**

# 第8章 hello的IO管理

## 8.1 Linux的IO设备管理方法

（*以下格式自行编排，编辑时删除*）

**设备的模型化：**文件——所有的I/O设备都被模型化为文件，甚至内核也被映射成文件。

**设备管理：**unix io接口——这种将设备优雅的映射为文件的方式，运行Linux内核引出一个简单的、低级的应用接口，成为UnixI/O。

我们可以对文件的操作有：打开关闭操作open、close，读写操作read，write，改变当前文件位置lseek等

## 8.2 简述Unix IO接口及其函数

（*以下格式自行编排，编辑时删除*）

接口：

1. **打开文件：**一个应用程序通过要求内核打开相应的文件，来宣告太想要访问一个I/O设备，内核返回一个小的非负整数，叫做描述符，他在后续对此文件的所有操作中标识这个文件，内核记录有关这个打开文件的所有信息，应用程序只需要记住这个描述符。
2. **Linux shell创建每个进程开始都有三个打开的文件：**标准输入（描述符为0），标准输出（描述符为1）和标准错误（描述符为2）。头文件<unistd.h>定义了常量STDIN\_FILENO,STOOUT\_FILENO,STDERR\_FILENO,他们可以代替显示的描述符值。
3. **改变当前文件位置：**对呀每个打开的文件，内核保持着一个当前文件位置k，初始值为0，这个文件位置是从文件开头起始的字节偏移量，应用程序能够通过执行seek。显示的改变这个k。
4. **读写文件：**一个读操作就是从文件复制n个自己到内存，从当前文件位置k开始，然后将k增加到k+n。给定一个大小为m字节的文件，如果m<k+n，执行读操作的时候会触发一个EOF（end of file）条件，应用程序可以检测到这个条件，但是文件没有明确的EOF标志。类似的写操作就是从内存复制n个字节到一个文件，从k开始，更新k。
5. **关闭文件：**应用程序完成了对文件的访问之后，就会通知内核关闭文件。作为相应，内核释放文件打开的时候建立的数据结构，并将这个描述符恢复到可用的描述符池里面。无论如何终止文件，内核都会释放。

函数

1. **打开和关闭文件**

打开：int open（char \*filename，int flag，mode\_t mode）

返回：如成功返回文件的描述符，不成功返回-1

Flags:O\_RDONLY(只读)，O\_WRONLY（只写），O\_RDWR(可读写)

Mode：访问新文件的权限位

关闭：int close（fd）

返回值：如果成功返回0，否则为-1

1. **读和写文件**

读：ssize\_t read( int fd, void \*buf, size\_t n)

返回值：如果成功，返回读的字节数，EOF为0，否则为-1

描述：从描述符为fd的当前文件位置赋值最多n个字节到内存位置buf

写：ssize\_t write( int fd, const void \*buf , size\_t n)

返回值：成功返回字节数，否则返回1

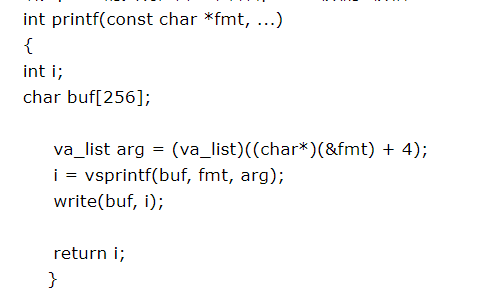
描述：从buf里面赋值至多n个字节到描述符为fd的文件的当前文件位置。

## 8.3 printf的实现分析

（*以下格式自行编排，编辑时删除*）

<https://www.cnblogs.com/pianist/p/3315801.html>

1. 首先观察printf函数的实现



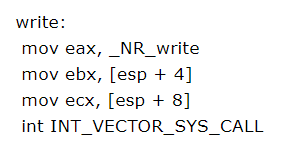
我们发现printf函数中，定义了一个字符指针va\_list类型的arg，这个函数 调用了vsprintf，继续看一下他：



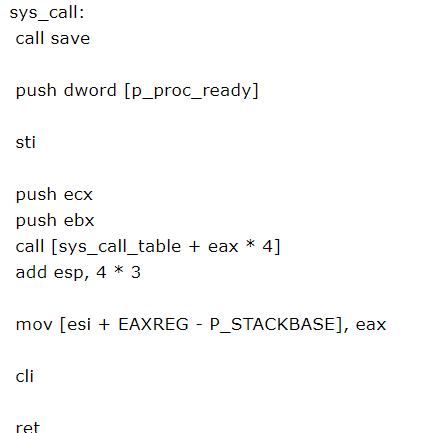
函数主要实现的功能：格式化。接受确定输出格式的字符串fmt，勇哥是字符串对个数变化参数进行格式化，产生格式化输出。

1. 系统函数write

反汇编write函数，发现要调用INT\_VECTOR\_SYS\_CALL，他通过系统调用sys\_call这个函数。



再来看sys\_call这个函数，通过分析，知道这个函数的主要功能是显示格式化的字符串，将要输出的字符串从总线复制到显卡的显存里面



1. 字符显示驱动子程序：从ASCII到字模库到显示vram（存储每一个点的RGB颜色信息）。
2. 显示芯片按照刷新频率逐行读取vram，并通过信号线向液晶显示器传输每一个点（RGB分量）。我们要显示的“hello 1190301610 王家琪”就被打印输出到了显示器上。

## 8.4 getchar的实现分析

1. 运行getchar函数的时候，用户模式转变为内核模式，输入时，显示屏上面有显示并进入缓存。按回车，通知内核完成，在切换为用户模式。

2. 异步异常-键盘中断的处理：键盘中断处理子程序。接受按键扫描码转成ascii码，保存到系统的键盘缓冲区。

3. getchar等调用read系统函数，通过系统调用读取按键ascii码，直到接受到回车键才返回。

## 8.5本章小结

在本章，我们知道了文件的操作。文件是一个抽象的概念，磁盘，显示器，适配器，网络都可以被计算机看成文件，甚至内核也是一个文件。

1. Linux提供了系统级的I/O函数。可以open，close，read，write，以及I/O重定向。
2. Printf函数是我们最常使用的函数，了解了他的内部代码，比较复杂，调用了vsprintf和系统调用write等函数，还有sys\_call等
3. Getchar函数用来中断处理，接受来找键盘的输入，他的实现也是基于系统调用函数write。

**（第8章1分）**

# 结论

用计算机系统的语言，逐条总结hello所经历的过程。

1. Hello虽小，五脏俱全，他的执行几乎涉及到了我们本学期学习的所有知识
2. Hello.c经过预处理，生成hello.i文本文件
3. Hello.i经过编译，生成了hello.s的文本文件，这是汇编语言的文本文件
4. Hello.s经过汇编，生成了hello.o的二进制文件，这是可重定位目标文件
5. Hello.o经过链接，生成了可执行文件hello，这个可以直接被计算机执行
6. Bash进程调用fork函数创建一个子进程，子进程execute可执行文件hello。
7. Hello的文件一开始是放在磁盘里面的，我们看的只是他的虚拟地址，通过操作系统的分页，我们最终将它放入内存，真正执行的时候我们是把虚拟地址转化为物理地址。
8. Hello可能调用I/O函数，这部分与linuxI/O系统函数密切相关
9. 最后hello执行完，被父进程回收，内核回收清除他的信息

你对计算机系统的设计与实现的深切感悟，你的创新理念，如新的设计与实现方法。

1. 学习是一个反复的过程。计算机是一个完整的体系，我们是分章节学习的，一开始很迷惑，有很多不太懂的地方，当学习完之后再看，不得不感叹数据结构设置的巧妙之处。
2. 在CSAPP学习的过程中，一开始是觉得简单，后来觉得越学越难，再后来又开始归纳复习，进行总结，但是现在又觉得博大精深。知识的学习是永无止境的，我们总是在不断的复习，回顾，但是一边学习要一边思考，为什么要这样设置？可不可以换成别的？如果是我，我应该怎么做？计算思维的培养和训练也很重要。

**（结论0分，缺失 -1分，根据内容酌情加分）**

# 附件

列出所有的中间产物的文件名，并予以说明起作用。

**（附件0分，缺失 -1分）**

# 参考文献

**为完成本次大作业你翻阅的书籍与网站等**

[1] 林来兴. 空间控制技术[M]. 北京：中国宇航出版社，1992：25-42.

[2] 辛希孟. 信息技术与信息服务国际研讨会论文集：A集[C]. 北京：中国科学出版社，1999.

[3] 赵耀东. 新时代的工业工程师[M/OL]. 台北：天下文化出版社，1998 [1998-09-26]. http://www.ie.nthu.edu.tw/info/ie.newie.htm（Big5）.

[4] 谌颖. 空间交会控制理论与方法研究[D]. 哈尔滨：哈尔滨工业大学，1992：8-13.

[5] KANAMORI H. Shaking Without Quaking[J]. Science，1998，279（5359）：2063-2064.

[6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL]. Science，1998，281：331-332[1998-09-23]. http://www.sciencemag.org/cgi/ collection/anatmorp.

**（参考文献0分，缺失 -1分）**