操作系统　Operating System

第三章 内存管理(2)

沃天宇
woty@buaa.edu.cn
2021年3月16日

存储分配的三种方式

- **1.直接指定方式**：程序员在编程序时,或编译程序(汇编程序)对源程序进行编译(汇编)时,所用的是实际地址。
- **2.静态分配(Static Allocation)**：程序员编程时,或由编译程序产生的目的程序,均可从其地址空间的零地址开始；当装配程序对其进行连接装入时才确定它们在主存中的地址。
- **3.动态分配(Dynamic Allocation)**：作业在存储空间中的位置,在其装入时确定,在其执行过程中可根据需要申请附加的存储空间,而且一个作业已占用的部分区域不再需要时,可以要求归还给系统。

REVIEW: 分区管理方案

- 固定分区：把内存划分为若干个固定大小的连续分区
  - 优点：没有外碎片。缺点：有内碎片。
- 可变式分区：分区的边界可以移动，即分区的大小可变。
  - 优点：没有内碎片。缺点：有外碎片。

动态分区的操作和数据结构

序号P	大小	起址	状态
1	8	20K	已分配
2	32	28K	已分配
3	--	--	
4	120	92K	已分配
5	--	--	
…	…	…	…

已分配分区（P表）

序号F	大小	起址	状态
1	32	60K	空闲
2	300	212K	空闲
3	--	--	
4	--	--	
5	--	--	
…	…	…	…

空闲分区（F表）

可变式分区的分配策略

- (1)首次适应算法（First Fit）：每个空白区按其在存储空间中地址递增的顺序连在一起，在为作业分配存储区域时，从这个空白区域链的始端开始查找，选择第一个足以满足请求的空白块。
- (2)下次适应算法（Next Fit）：把存储空间中空白区构成一个循环链，每次为存储请求查找合适的分区时，总是从上次查找结束的地方开始，只要找到一个足够大的空白区，就将它划分后分配出去。
- (3)最佳适应算法（Best Fit）：总是寻找能够容纳作业的最小存储区域。
- (4)最坏适应算法（Worst Fit）：总是寻找能够容纳作业的最大的空白区。
- 伙伴系统

伙伴系统

伙伴系统示例（1M 内存）

Action	Memory					
Start	1M					
A请求150kb	A	256k			512k	
B请求100kb	A	B	128k		512k	
C请求50kb	A	B	C	64k	512k	
释放B	A	128k	C	64k	512k	
D请求200kb	A	128k	C	64k	D	256k
E请求60kb	A	128k	C	E	D	256k
释放C	A	128k	64k	E	D	256k
释放A	256k	128k	64k	E	D	256k
释放E	512k				D	256k
释放D	1M					

伙伴系统

1.是否会产生内碎片？

2.是否会产生外碎片？

3.最大内碎片多大？

4.如果既会产生内碎片，又会产生外碎片，那伙伴系统有什么优势？

分配效率高

伙伴系统特点

伙伴系统利用计算机二进制数寻址的优点，加速了相邻空闲分区的合并。

当一个 2^i 字节的块释放时，只需搜索 2^i 字节的块，而其它算法则必须搜索所有的块，伙伴系统速度更快。

伙伴系统的缺点：不能有效地利用内存。进程的大小不一定是 2 的整数倍，由此会造成浪费，内部碎片严重。例如，一个 257KB 的进程需要占用一个 512KB 的分配单位，将产生 255KB 的内部碎片。

伙伴系统不如基于分页和分段的虚拟内存技术有效。

伙伴系统目前应用于 Linux 系统和多处理机系统。

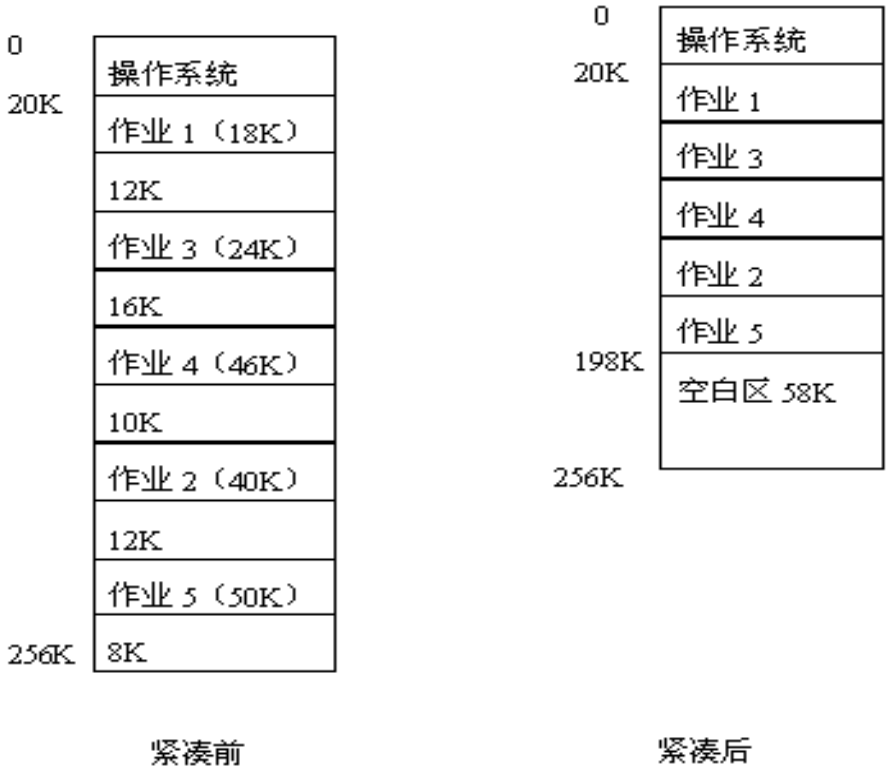
可重定位分区分配

- 可重定位分区分配（紧凑）：定时的或在内存紧张时，移动某些已分配区中的信息，把存储空间中所有的空白区合并为一个大的连续区。

可重定位分区分配

- 可重定位分区分配：定时的或在内存紧张时，移动某些已分配区中的信息，把存储空间中所有的空白区合并为一个大的连续区。

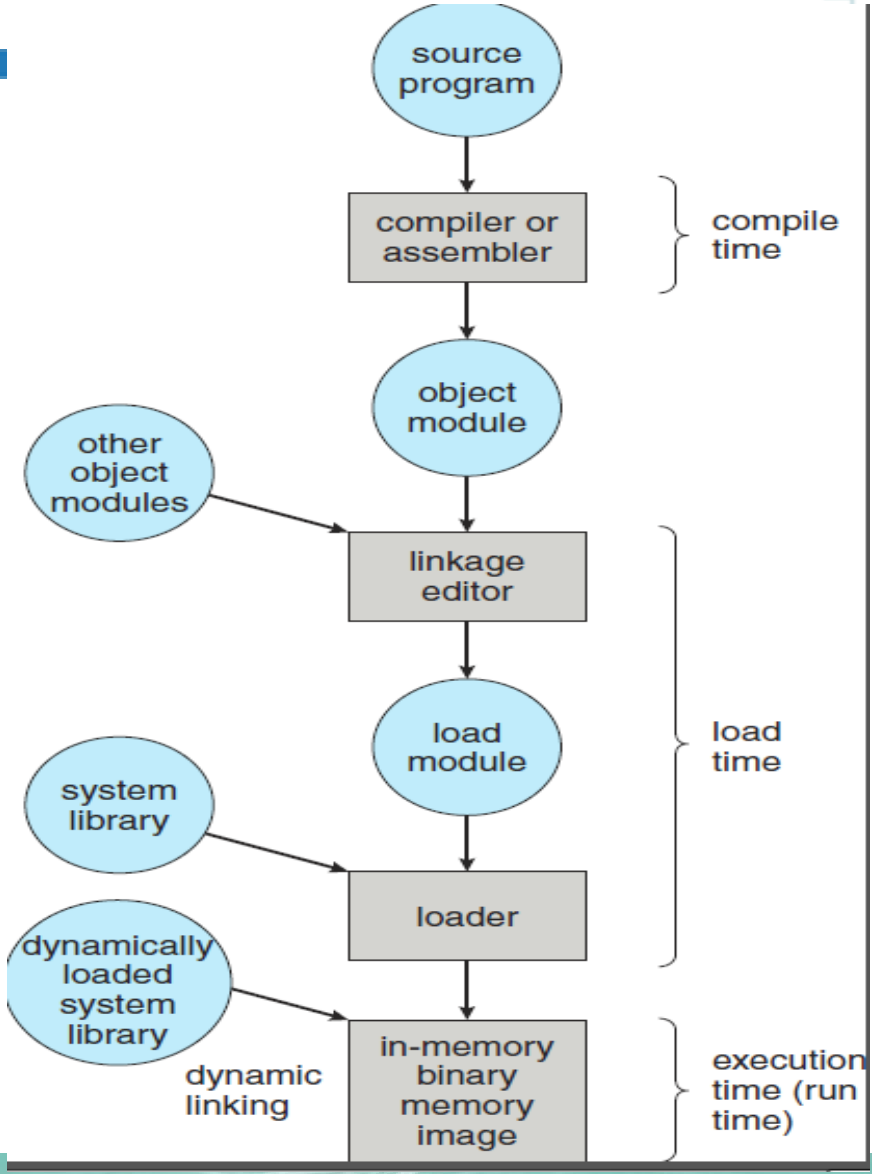
- 缺点、限制
  - 性能开销
  - 设备依赖（DMA）
  - 间接寻址



程序的链接和装入

一个用户源程序要变为在内存中可执行的程序，通常要进行以下处理：

- **编译(compile)**：由编译程序将用户源程序编译成若干个目标模块。
- **链接(linking)**：由链接程序将目标模块和相应的库函数链接成可装载模块（可执行文件）。
- **装入(loading)**：由装载程序将可装载入模块装入内存。



程序的链接

采用静态链接和动态链接方式

- **静态链接：**用户一个工程中所需的多个程序采用静态链接的方式链接在一起。当我们希望共享库的函数代码直接链接入程序代码中，也采用静态链接方式。
- **动态链接：**用于链接共享库代码。当程序运行中需要某些目标模块时，才对它们进行链接，具有高效且节省内存空间的优点。但相比静态链接，使用动态链接库的程序相对慢。

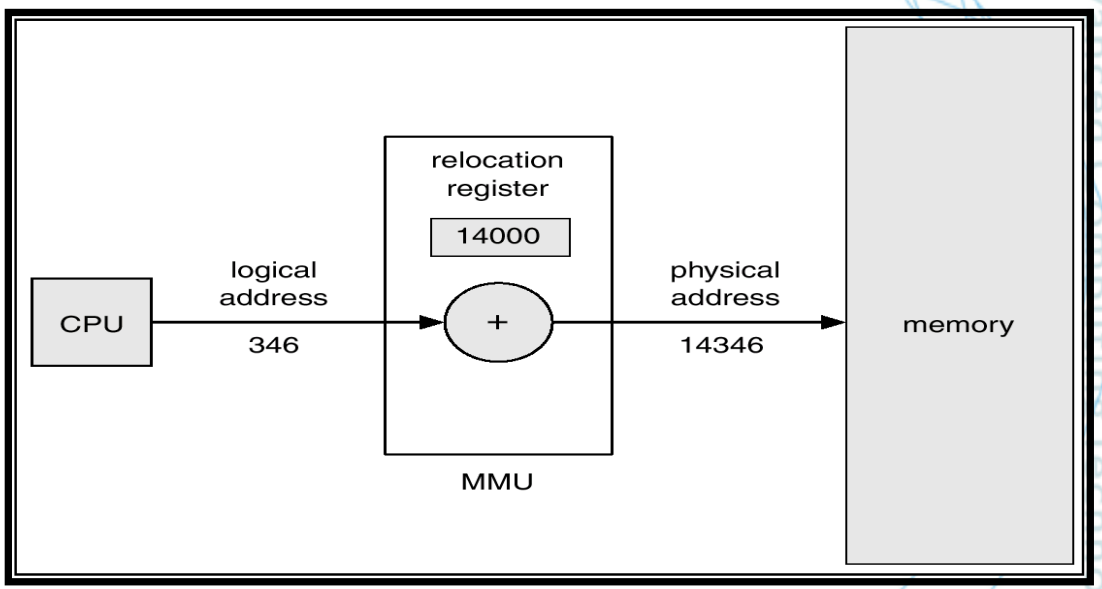
程序的装入

一般采用动态运行时装入方式

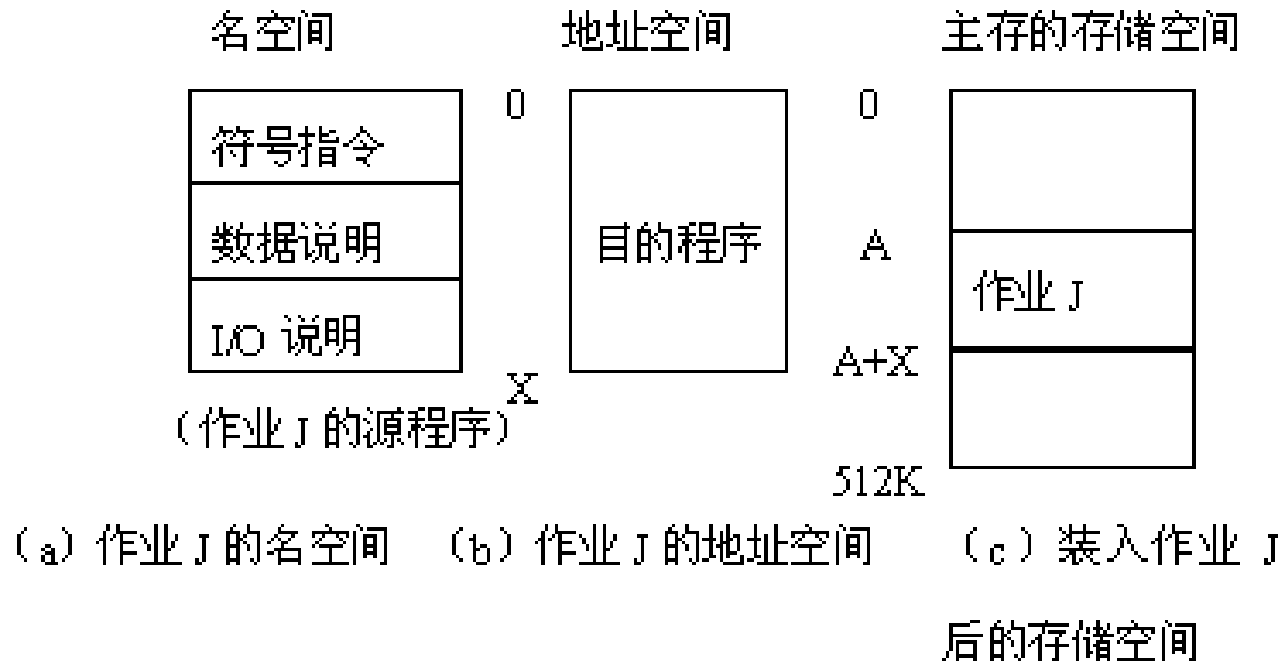
- 程序在内存中的位置经常要改变。程序在内存中的移动意味着它的物理位置发生了变化，这时必须对程序和数据的地址 (绝对地址) 进行修改后方能运行。
- 为了保证程序在内存中的位置可以改变。装入程序把装入模块装入内存后，并不立即把装入模块中相对地址转换为绝对地址，而是在程序运行时才进行。
- 这种方式需要一个重定位寄存器来支持，在程序运行过程中进行地址转换。

重定位

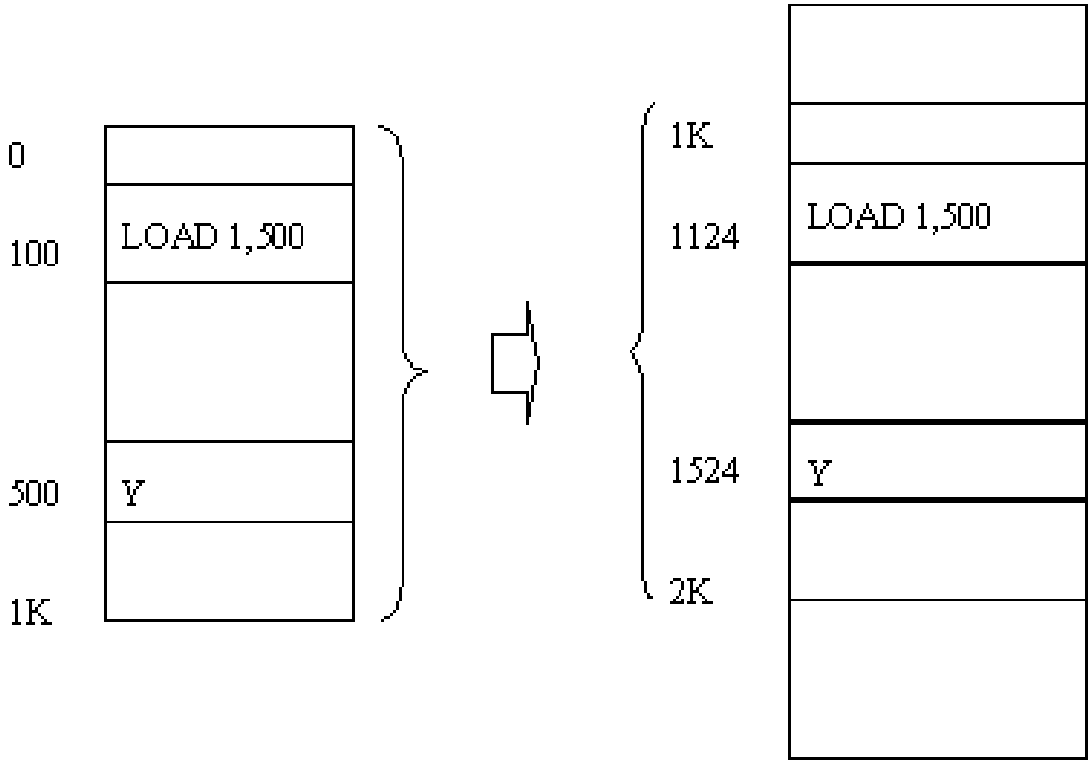
- 在装入时对目标程序中的指令和数据地址的修改，或映射过程。
  - 静态重定位
  - 动态重定位



作业J存在于不同的空间中

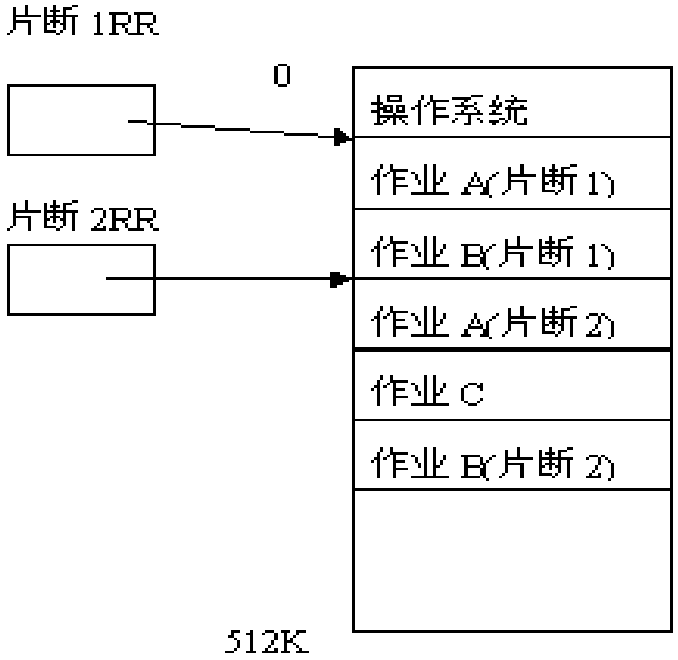


作业由地址空间装入存储空间



多重分区分配

- 多重分区分配：一个作业往往由相对独立的程序段和数据段组成，将这些片断分别装入到存储空间中不同的区域内的分配方式。

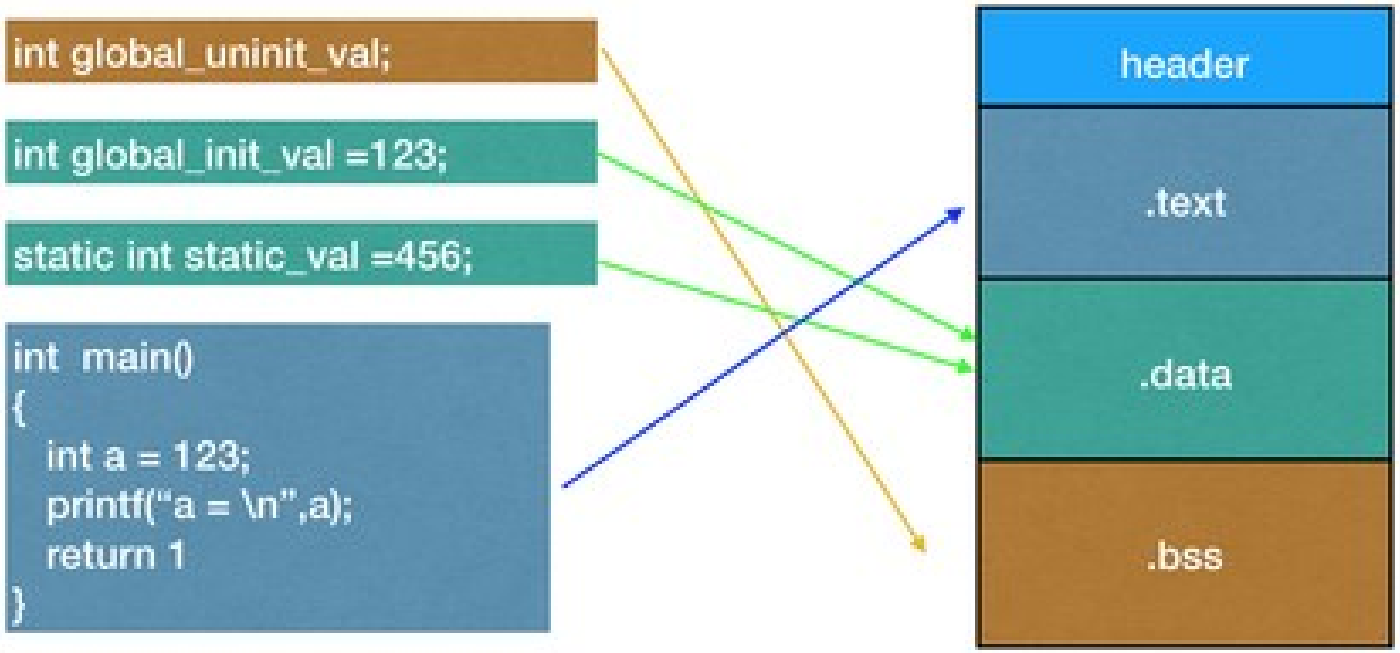


基础知识：程序段

- 一个程序本质上都是由 bss段、data段、text段三个组成的。
- 在C语言之类的程序编译完成之后，已初始化的全局变量保存在data 段中，未初始化的全局变量保存在bss 段中。
  - bss段：（bss segment）用来存放程序中未初始化的全局变量的一块内存区域。bss是英文Block Started by Symbol的简称。bss段属于静态内存分配。
  - data段：数据段（data segment）用来存放程序中已初始化的全局变量的一块内存区域。数据段属于静态内存分配。

基础知识：程序段

- 一个程序本质上都是由 bss段、data段、text段三个组成的。在C语言之类的程序编译完成之后，已初始化的全局变量保存在**data 段**中，未初始化的全局变量保存在**bss 段**中。

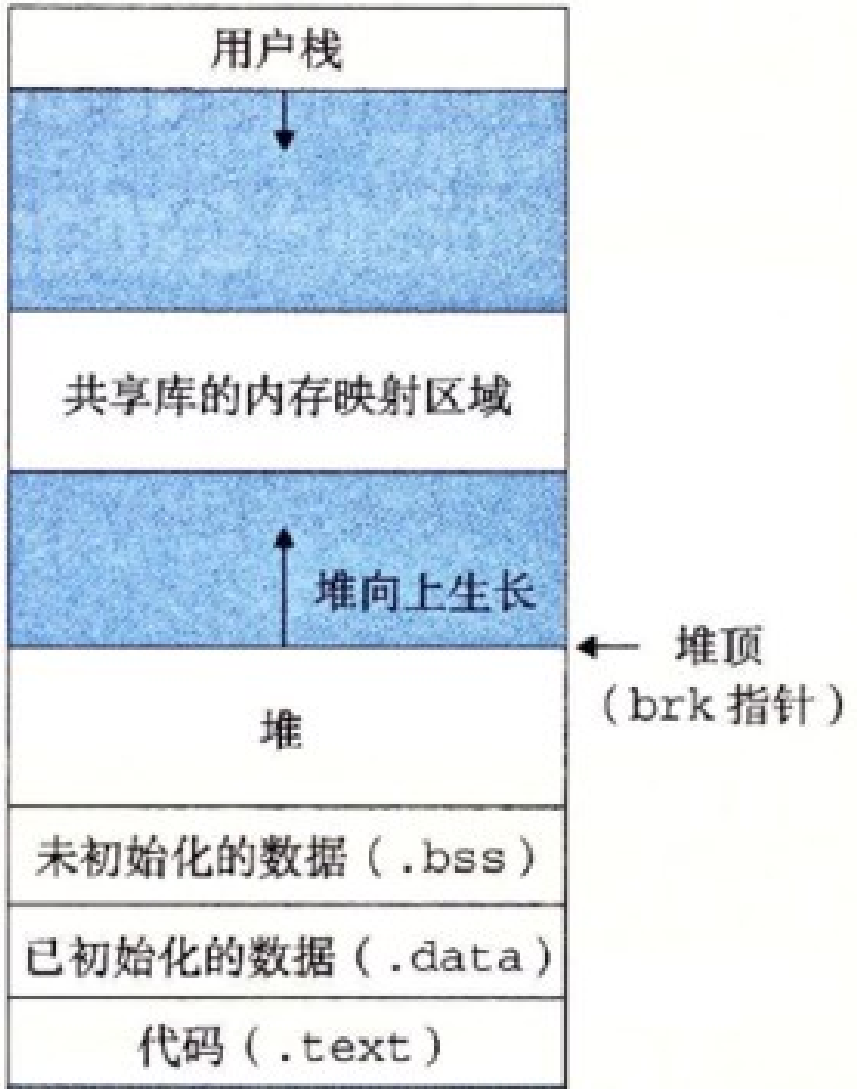


基础知识：程序段

- bss段：（bss segment）用来存放程序中未初始化的全局变量的一块内存区域。bss是英文Block Started by Symbol的简称。bss段属于静态内存分配。
- data段：数据段（data segment）用来存放程序中已初始化的全局变量的一块内存区域。数据段属于静态内存分配。
- text段：代码段（code segment/text segment）用来存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于只读(某些架构也允许代码段为可写，即允许修改程序)。在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。

基础知识

- text和data段都在可执行文件中，由系统从可执行文件中加载，而bss段不在可执行文件中，由系统初始化。
- 一个装入内存的可执行程序，除了bss、data和text段外，还需构建一个栈（stack）和一个堆（heap）。



基础知识

- 栈(stack)：存放、交换临时数据的内存区
  - 用户存放程序局部变量的内存区域，（但不包括static声明的变量，static意味着在数据段中存放变量）。
  - 保存/恢复调用现场。在函数被调用时，其参数也会被压入发起调用的进程栈中，并且待到调用结束后，函数的返回值也会被存放回栈中。
- 堆（heap）：存放进程运行中动态分配的内存段
  - 它的大小并不固定，可动态扩张或缩减。当进程调用malloc等函数分配内存时，新分配的内存就被动态添加到堆上（堆被扩张）；当利用free等函数释放内存时，被释放的内存从堆中被剔除（堆被缩减）。

具体实例：program.c

```
int read_something(void);
int do_something(int);
void write_something(const char*);
int some_global_variable;
static int some_local_variable;

int main () {
        int some_stack_variable;
        some_stack_variable = read_something();
        some_global_variable = do_something(some_stack_variable);
        write_something("I have done");
}
```

program.c

直接编译，报错：

```
gcc -o program program.c
/tmp/ccvuB6VX.o: In function `main':
program.c:(.text+0x10): undefined reference to `read_something'
program.c:(.text+0x20): undefined reference to `do_something'
program.c:(.text+0x3c): undefined reference to `write_something'
collect2: error: ld returned 1 exit status
```

extras.c

```
#include <stdio.h>
extern int some_global_variable;
int read_something (void) {
        int res;
        scanf("%d", &res);
        return res;
}
int do_something(int var) {
        return var + var;
}
void write_something (const char* str) {
        printf ("%s: %d\n", str, some_global_variable);
}
```

```
int some_global_variable;
    //全局变量，项目所有的源文件都可以访问
static int some_local_variable;
    //静态全局变量，仅仅当前源文件可以访问

main() {
        int some_stack_variable;
        //栈分配，仅仅当前函数可以访问
        …
}
```

Gcc的编译和链接

//默认是编译+链接

gcc -o program program.c extras.c

./program

//只编译　　-c 仅仅编译不链接

gcc -c program.c 结果生成 **program.o**

gcc -c extras.c 结果生成 **extras.o**

//只链接

gcc program.o functions.o -o program

./program

gcc调用包含的几个工具

cc1：预处理器和编译器

as：汇编器

collect2：链接器

请观察gcc的详细输出（用-v参数），找出gcc调用的各个编译环节相应程序

生成的汇编文件

```
.file   1 "program.c"
.section .mdebug.abi32
.previous
.nan    legacy
.module fp=32
.module nooddspreg
.abicalls
.option pic0

.comm   some_global_variable,4,4
.local  some_local_variable
.comm   some_local_variable,4,4
.rdata
.align  2
```

```
$LC0:
        .ascii  "I have done\000"
        .text
        .align  2
        .globl  main
        .set    nomips16
        .set    nomicromips
        .ent    main
        .type   main, @function
main:
        .frame  $fp,40,$31
         # vars= 8, regs= 2/0, args= 16, gp= 8
        .mask   0xc0000000,-4
        .fmask  0x00000000,0
```

生成的汇编文件

```
.set    noreorder
.set    nomacro
addiu   $sp,$sp,-40
sw      $31,36($sp)
sw      $fp,32($sp)
move    $fp,$sp
jal     read_something
nop

sw      $2,24($fp)
lw      $4,24($fp)
jal     do_something
nop
move    $3,$2
lui     $2,%hi(some_global_variable)
sw      $3,%lo(some_global_variable)($2)
lui     $2,%hi($LC0)
addiu   $4,$2,%lo($LC0)
```

```
jal     write_something
nop
move    $2,$0
move    $sp,$fp
lw      $31,36($sp)
lw      $fp,32($sp)
addiu   $sp,$sp,40
j       $31
nop

.set    macro
.set    reorder
.end    main
.size   main, .-main
.ident  "GCC: (crosstool-NG
        crosstool-ng-1.22.0) 5.2.0"
```

函数调用变成了汇编函数调用指令，do_something只是标记

Linux下可执行文件的格式

- 在Linux下可执行文件的格式为ELF（Executable and Linkable Format），ELF文件分为三类：
  - 1.可重定位（relocatable）文件，保存着代码和适当的数据，用来和其他的object文件一起来创建一个可执行文件或者是一个共享文件。
  - 2.可执行（executable）文件，保存着一个用来执行的程序，该文件指出了exec（BA_OS）如何来创建程序进程映像。
  - 3.共享object文件，保存着代码和合适的数据，用来被下面的两个链接器链接。第一个是链接编辑器（静态链接），可以和其他的可重定位和共享object文件一起来创建object文件；第二个是动态链接器，联合一个可执行文件和其他的共享object文件来创建一个进程映象。

Linux下可执行文件的格式

- 注意，ELF文件是二进制兼容的文件（ABI，应用程序二进制接口），也就是说ELF文件已经是适应到某一种CPU体系结构的二进制文件了。可以这样来理解：ELF文件是经过编译或链接生成的文件，而编译或链接必须指定具体的CPU体系结构，
- 故ELF文件是针对某一种CPU体系结构（即与具体体系结构相关）的二进制文件。

ELF文件格式

ELF头
程序头表（可省略）
.text
.rodata
.data
……
节头表

.bss	此节存放用于程序内存映象的未初始化数据。此节类型是SHT_NOBITS,因此不占文件空间。
.comment	此节存放版本控制信息。
.data和.data1	此节存放用于程序内存映象的初始化数据。
.debug	此节存放符号调试信息。
.dynamic	此节存放动态连接信息。
.dynstr	此节存放动态连接所需的字符串，在大多数情况下，这些字符串代表的是与符号表项有关的名字。
.dynsym	此节存放的是“符号表”中描述的动态连接符号表。
.fini	此节存放与进程中指代码有关的执行指令。
.got	此节存放全程偏移量表。
.hash	此节存放一个符号散列表。
.init	此节存放组成进程初始化代码的执行指令。
.interp	此节存放一个程序解释程序的路径名。
.line	此节存放符号调试中使用的行号信息，主要描述源程序与机器指令之间的对应关系。
.note	此节存放供其他程序检测兼容性，一致性的特殊信息。
.plt	此节存放过程连接表。
.rel*name*和.rela*name*	此节存放重定位信息。
.rodata和.rodata1	此节存放进程映象中不可写段的只读数据。
.shstrtab	此节存放节名。
.strtab	此节存放的字符串标识与符号表项有关的名字。
.symtab	此节存放符号表。
.text	此节存放正文，也称程序的执行指令。

ELF文件头的定义（52个字节）

ELF头描述文件组成。

```
typedef struct {
        unsigned char     e_ident[16];     /* 标志本文件为目标文件，提供
                                              机器无关的数据，可实现对文
                                              件内容的译码与解释*/
        unsigned char     e_type[2];       /* 标识目标文件类型 */
        unsigned char     e_machine[2];    /* 指定必需的体系结构 */
        unsigned char     e_version[4];    /* 标识目标文件版本 */
        unsigned char     e_entry[4];      /* 指向起始虚地址的指针 */
        unsigned char     e_phoff[4];      /* 程序头表的文件偏移量 */
        unsigned char     e_shoff[4];      /* 节头表的文件偏移量 */
        unsigned char     e_flags[4];      /* 针对具体处理器的标志 */
        unsigned char     e_ehsize[2];     /* ELF 头的大小 */
        unsigned char     e_phentsize[2];  /* 程序头表每项的大小 */
        unsigned char     e_phnum[2];      /* 程序头表项的个数 */
        unsigned char     e_shentsize[2];  /* 节头表每项的大小 */
        unsigned char     e_shnum[2];      /* 节头表项的个数 */
        unsigned char     e_shstrndx[2];   /* 与节名字符串表相关的节头表
                                              项的索引 */
} Elf32_Ehdr;
```

ELF文件头的定义

e_ident:	这一部分是文件的标志，用于表明该文件是一个ELF文件。ELF文件的头四个字节为magic number。
e_type:	用于标明该文件的类型，如可执行文件、动态连接库、可重定位文件等。
e_machine:	表明体系结构，如x86，x86_64，MIPS，PowerPC等等。
e_version:	文件版本
e_entry:	程序入口的虚拟地址
e_phoff:	程序头表在该ELF文件中的位置(具体地说是偏移)。ELF文件可以没有程序头表。

ELF文件头的定义

e_shoff：	节头表的位置。
e_eflags：	针对具体处理器的标志。
e_ehsize：	ELF 头的大小。
e_phentsize：	程序头表每项的大小。
e_phnum：	程序头表项的个数。
e_shentsize：	节头表每项的大小。
e_shnum：	节头表项的个数。
e_shstrndx：	与节名字符串表相关的节头表。

一个具体的ELF文件头

```
ELF Header:
  Magic:   7f 45 4c 46 01 02 01 00 01 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, big endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       1
  Type:                              EXEC (Executable file)
  Machine:                           MIPS R3000
  Version:                           0x1
  Entry point address:               0x4004c0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          5520 (bytes into file)
  Flags:                             0x1005, noreorder, cpic, o32, mips1
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         35
  Section header string table index: 32
```

使用objdump反汇编ELF文件

```
program.o:     file format elf32-tradbigmips
Disassembly of section .text:
00000000 <main>:
  0:   27bdffd8        addiu   sp,sp,-40
  4:   afbf0024        sw      ra,36(sp)
  8:   afbe0020        sw      s8,32(sp)
  c:   03a0f021        move    s8,sp
 10:   0c000000        jal     0 <main>
 14:   00000000        nop
 18:   afc20018        sw      v0,24(s8)
 1c:   8fc40018        lw      a0,24(s8)
 20:   0c000000        jal     0 <main>
 24:   00000000        nop
 28:   00401821        move    v1,v0
 2c:   3c020000        lui     v0,0x0
 30:   ac430000        sw      v1,0(v0)
 34:   3c020000        lui     v0,0x0
 38:   24440000        addiu   a0,v0,0
 3c:   0c000000        jal     0 <main>
```

偏移

汇编指令

机器码

使用objdump反汇编ELF文件

- 在源文件三处函数调用，对应到汇编文件里，就是三处jal指令。
- 三条jal所对应的机器码，头六位二进制数(000011)代表jal，而后面的一串0是操作数，也就是要跳转到的地址。

10: 0c000000 jal 0 <main>

该条机器指令的二进制表示：

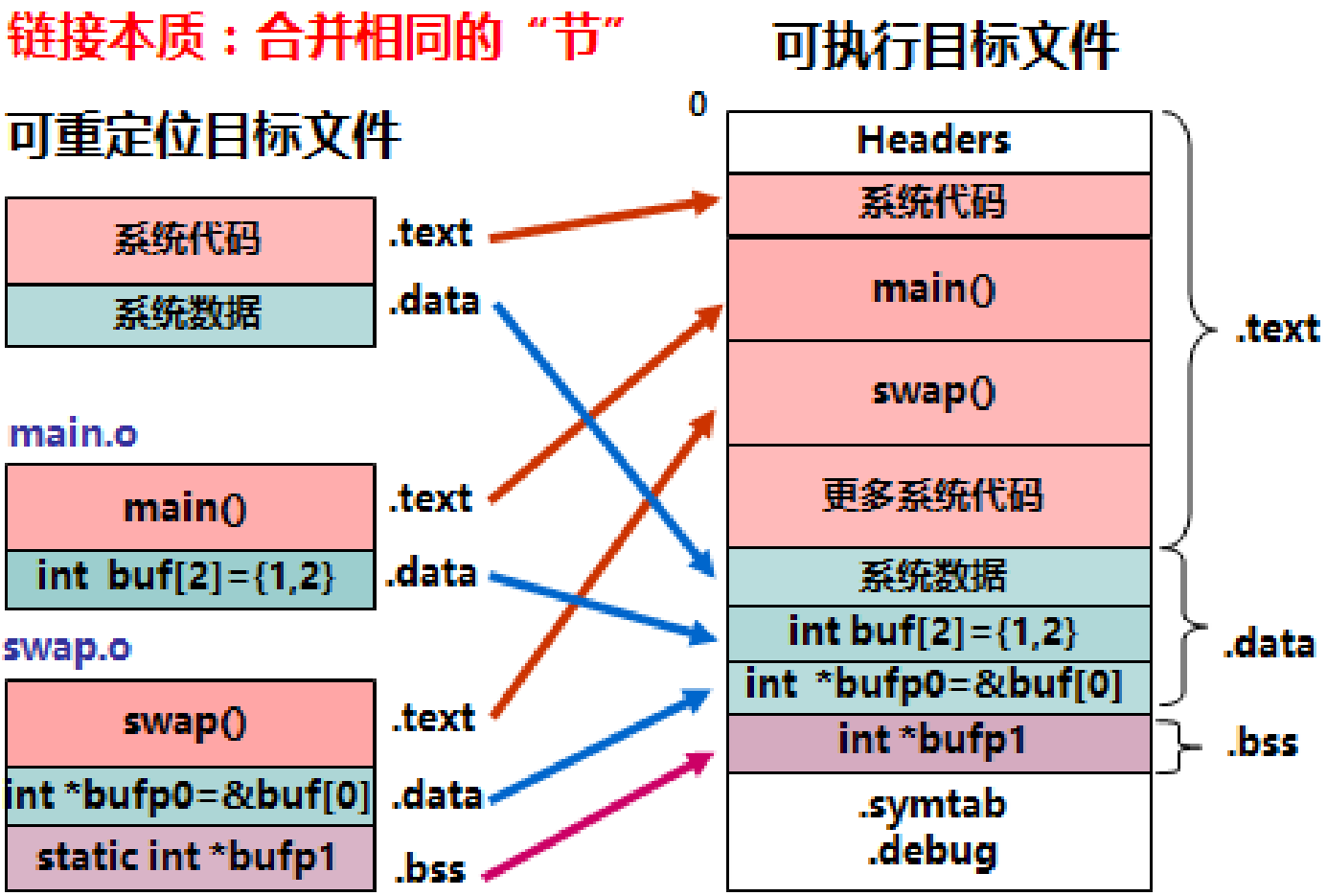
$(0c000000)_{16}=(0000\ 1100\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000)_2$

要跳转的函数地址都是0？！

链接的过程

- 编译C程序的时候，是以.c文件作为编译单元的。
  - 编译：.c→.o；编译时函数定义在不同文件，无法知道地址。在我们的例子中，会产生两个.o文件，分别是program.o和extras.o。
- 链接的过程：
  - 将这些.o文件链接到一起，形成最终的可执行文件。
  - 在链接时，链接器会扫描各个目标文件，将之前未填写的地址填写上，从而生成一个真正可执行的文件。
- 重定位(Relocation)
  - 将之前未填写的地址填写的过程。在符号解析的基础上将所有关联的目标模块合并，并确定运行时每个定义符号在虚拟地址空间中的地址，在定义符号的引用处重定位引用的地址。

链接过程的本质


链接本质：合并相同的“节”
可执行目标文件
可重定位目标文件
0
系统代码
系统数据
.text
.data
main.o
main()
int buf[2]={1,2}
.text
.data
swap.o
swap()
int *bufp0=&buf[0]
static int *bufp1
.text
.data
.bss
Headers
系统代码
main()
swap()
更多系统代码
系统数据
int buf[2]={1,2}
int *bufp0=&buf[0]
int *bufp1
.symtab
.debug
.text
.data
.bss

Relocation entry

typedef struct {

/*给出了使用重定位动作的地点。对重定位文件来说，它的值是从节起始处到受重定位影响的存储单元的字节偏移量；对可执行文件或共享目标文件来说，它的值是受重定位影响的存储单元的虚拟地址*/

Elf32_Addr r_offset;

/*给出了与重定位修改地点有关的符号表索引和所使用的重定位的类型*/

Elf32_Word r_info;(symbol:24; type:8)

} Elf32_Rel;

Readelf读取重定位节

- Relocation section '.rel.text' at offset 0x348 contains 7 entries:

Offset	Info	Type	Sym.Value	Sym. Name
00000010	00000f04	R_MIPS_26	00000000	read_something
00000020	00001004	R_MIPS_26	00000000	do_something
0000002c	00000d05	R_MIPS_HI16	00000004	some_global_variable
00000030	00000d06	R_MIPS_LO16	00000004	some_global_variable
00000034	00000705	R_MIPS_HI16	00000000	.rodata
00000038	00000706	R_MIPS_LO16	00000000	.rodata
0000003c	00001104	R_MIPS_26	00000000	write_something

10: 0c000000 jal 0 <main>

链接后…

```
004006a0 <main>:
 4006a0:    27bdffd8    addiu   sp,sp,-40
 4006a4:    afbf0024    sw      ra,36(sp)
 4006a8:    afbe0020    sw      s8,32(sp)
 4006ac:    03a0f021    move    s8,sp
 4006b0:    0c1001c0    jal     400700 <read_something>
 4006b4:    00000000    nop
 4006b8:    afc20018    sw      v0,24(s8)
 4006bc:    8fc40018    lw      a0,24(s8)
 4006c0:    0c1001d1    jal     400744 <do_something>
 4006c4:    00000000    nop
 4006c8:    00401821    move    v1,v0
 4006cc:    3c020041    lui     v0,0x41
 4006d0:    ac430a1c    sw      v1,2588(v0)
 4006d4:    3c020040    lui     v0,0x40
 4006d8:    24440930    addiu   a0,v0,2352
 4006dc:    0c1001de    jal     400778 <write_something>
 4006e0:    00000000    nop
 4006e4:    00001021    move    v0,zero
 4006e8:    03c0e821    move    sp,s8
 4006ec:    8fbf0024    lw      ra,36(sp)
 4006f0:    8fbe0020    lw      s8,32(sp)
 4006f4:    27bd0028    addiu   sp,sp,40
 4006f8:    03e00008    jr      ra
 4006fc:    00000000    nop
400700:  ……
```

```
 0:   27bdffd8    addiu   sp,sp,-40
 4:   afbf0024    sw      ra,36(sp)
 8:   afbe0020    sw      s8,32(sp)
 c:   03a0f021    move    s8,sp
10:   0c000000    jal     0 <main>
14:   00000000    nop
18:   afc20018    sw      v0,24(s8)
1c:   8fc40018    lw      a0,24(s8)
20:   0c000000    jal     0 <main>
24:   00000000    nop
28:   00401821    move    v1,v0
2c:   3c020000    lui     v0,0x0
30:   ac430000    sw      v1,0(v0)
34:   3c020000    lui     v0,0x0
38:   24440000    addiu   a0,v0,0
3c:   0c000000    jal     0 <main>
……
```

重定位时链接地址的计算

Name	Symbol	Calculation
R_MIPS_26	Local	((A | ((P + 4) & 0xf0000000)) + S) >> 2
	External	(sign_extend(A) + S) >> 2
R_MIPS_HI16	Any	%high(AHL + S) The %high(x) function is (x - (short)x) >> 16
R_MIPS_LO16	Any	AHL+S

A　　附加值(addend)。
S　　符号的地址。
AHL　地址的附加量(addend)。

链接地址的计算read_something

10: 0c000000 jal 0 <main> 编译后main.o

Offset	Info	Type	Sym.Value	Sym. Name
00000010	00000f04	R_MIPS_26	00000000	read_something

Symbol table '.symtab' contains 93 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
67:	00400700	68	FUNC	GLOBAL	DEFAULT	13	read_something

计算的公式为(sign_extend(A) + S) >> 2，其中，A=0，S=00400700，所以结果为1001c0，填写到jal指令的操作数的位置，得到的结果正是0c1001c0，与汇编器给出的一致。

0000 0000 0100 0000 0000 0111 0000 00~~00~~ 右移2位→

0000 0000 0001 0000 0000 0001 1100 0000

4006b0: 0c1001c0 jal 400700 <read_something> 链接后

链接地址的计算some_global_variable

2c: 3c020000 lui v0, 0x0

30: ac430000 sw v1,0(v0) 编译后main.o

Offset	Info	Type	Sym.Value	Sym. Name
0000002c	00000d05	R_MIPS_HI16	00000004	some_global_variable
00000030	00000d06	R_MIPS_LO16	00000004	some_global_variable

Symbol table '.symtab' contains 93 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
62:	00410a1c	4	OBJECT	GLOBAL	DEFAULT	26	some_global_variable

高16位的类型为R_MIPS_HI16，计算公式为((AHL + S)-(short)(AHL + S))>>16，此处AHL为0，S为00410a1c，结果为41

低16位地址的类型为R_MIPS_LO16，计算公式为AHL+S，此处AHL为0，S为00410a1c。这里只保留16位，因此，结果为0a1c

4006cc: 3c020041 lui v0, 0x41

4006d0: ac430a1c sw v1, 2588(v0) 链接后

程序入口点

```
ELF Header:
  Magic:   7f 45 4c 46 01 02 01 00 01 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, big endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       1
  Type:                              EXEC (Executable file)
  Machine:                           MIPS R3000
  Version:                           0x1
  Entry point address:               0x4004c0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          5520 (bytes into file)
  Flags:                             0x1005, noreorder, cpic, o32, mips1
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         35
  Section header string table index: 32
```

但是不等于Main的地址：
004006a0？？？

程序入口点-_start 函数

```
/usr/lib/crt1.o:     file format elf32-tradbigmips
Disassembly of section .text:
00000000 <__start>:
   0:   3c1c0000        lui     gp,0x0
   4:   279c0000        addiu   gp,gp,0
   8:   0000f821        move    ra,zero
   c:   3c040000        lui     a0,0x0
  10:   24840000        addiu   a0,a0,0
  14:   8fa50000        lw      a1,0(sp)
  18:   27a60004        addiu   a2,sp,4
  1c:   2401fff8        li      at,-8
  20:   03a1e824        and     sp,sp,at
  24:   27bdffe0        addiu   sp,sp,-32
  28:   3c070000        lui     a3,0x0
  2c:   24e70000        addiu   a3,a3,0
  30:   3c080000        lui     t0,0x0
  34:   25080000        addiu   t0,t0,0
  38:   afa80010        sw      t0,16(sp)
  3c:   afa20014        sw      v0,20(sp)
  40:   afbd0018        sw      sp,24(sp)
  44:   3c190000        lui     t9,0x0
  48:   27390000        addiu   t9,t9,0
  4c:   0320f809        jalr    t9
```

程序入口点

start_入口函数调用了main

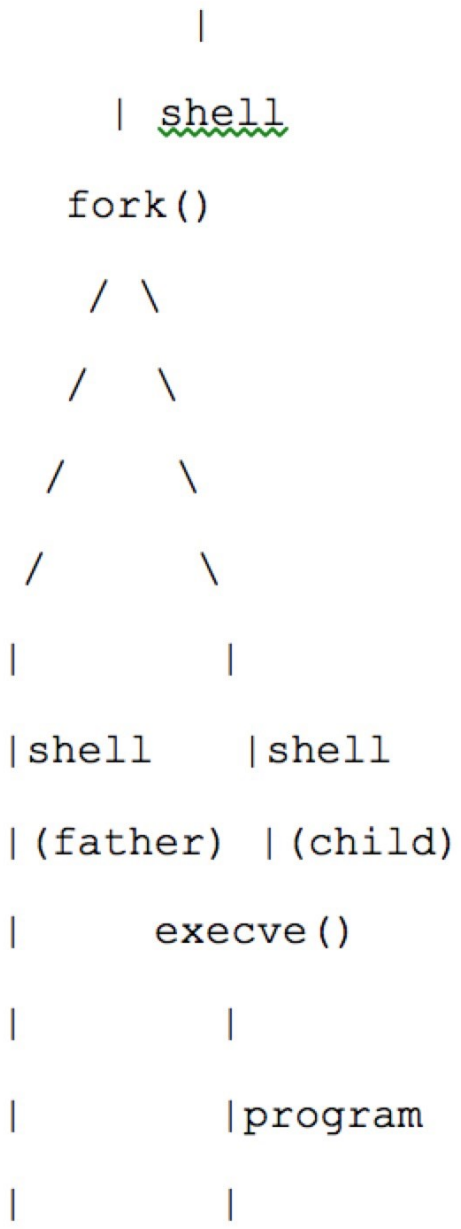
Crt1.o 的定位表:

Relocation section '.rel.text' at offset 0x42c contains 10 entries:

Offset	Info	Type	Sym.Value	Sym. Name
00000000	00000f05	R_MIPS_HI16	00000000	_gp
00000004	00000f06	R_MIPS_LO16	00000000	_gp
0000000c	00001305	R_MIPS_HI16	00000000	main
00000010	00001306	R_MIPS_LO16	00000000	main
00000028	00001205	R_MIPS_HI16	00000000	__libc_csu_init
0000002c	00001206	R_MIPS_LO16	00000000	__libc_csu_init
00000030	00001005	R_MIPS_HI16	00000000	__libc_csu_fini
00000034	00001006	R_MIPS_LO16	00000000	__libc_csu_fini
00000044	00001605	R_MIPS_HI16	00000000	__libc_start_main
00000048	00001606	R_MIPS_LO16	00000000	__libc_start_main

程序的装载和运行

- 执行程序的过程
  - shell调用fork()系统调用，
  - 创建出一个子进程
  - 子进程调用execve()加载program
- Fork()
- Execve(char *filename, char *argv[], char *envp)



程序的装载

装载前的工作：

- shell调用fork()系统调用，创建出一个子进程。

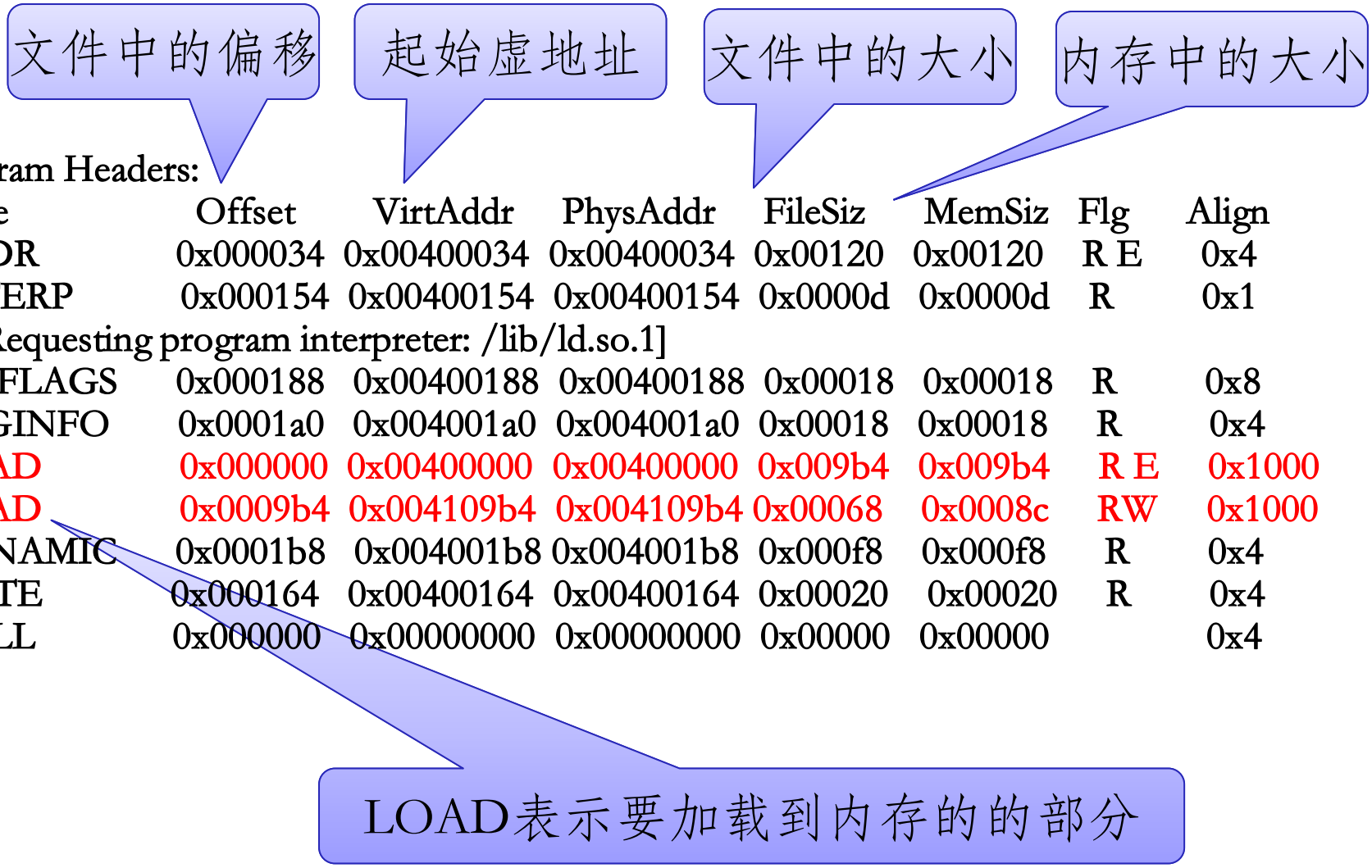
装载工作：

- 子进程调用execve()加载program(即要执行的程序)。

程序如何被加载：

- 加载器在加载程序的时候只需要看ELF文件中和segment相关的信息即可。我们用readelf工具将segment读取出来。
- 其中Type为Load的segment是需要被加载到内存中的部分。

程序的装载



Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000034	0x00400034	0x00400034	0x00120	0x00120	R E	0x4
INTERP	0x000154	0x00400154	0x00400154	0x0000d	0x0000d	R	0x1
[Requesting program interpreter: /lib/ld.so.1]							
ABIFLAGS	0x000188	0x00400188	0x00400188	0x00018	0x00018	R	0x8
REGINFO	0x0001a0	0x004001a0	0x004001a0	0x00018	0x00018	R	0x4
LOAD	0x000000	0x00400000	0x00400000	0x009b4	0x009b4	R E	0x1000
LOAD	0x0009b4	0x004109b4	0x004109b4	0x00068	0x0008c	RW	0x1000
DYNAMIC	0x0001b8	0x004001b8	0x004001b8	0x000f8	0x000f8	R	0x4
NOTE	0x000164	0x00400164	0x00400164	0x00020	0x00020	R	0x4
NULL	0x000000	0x00000000	0x00000000	0x00000	0x00000		0x4

程序的装载

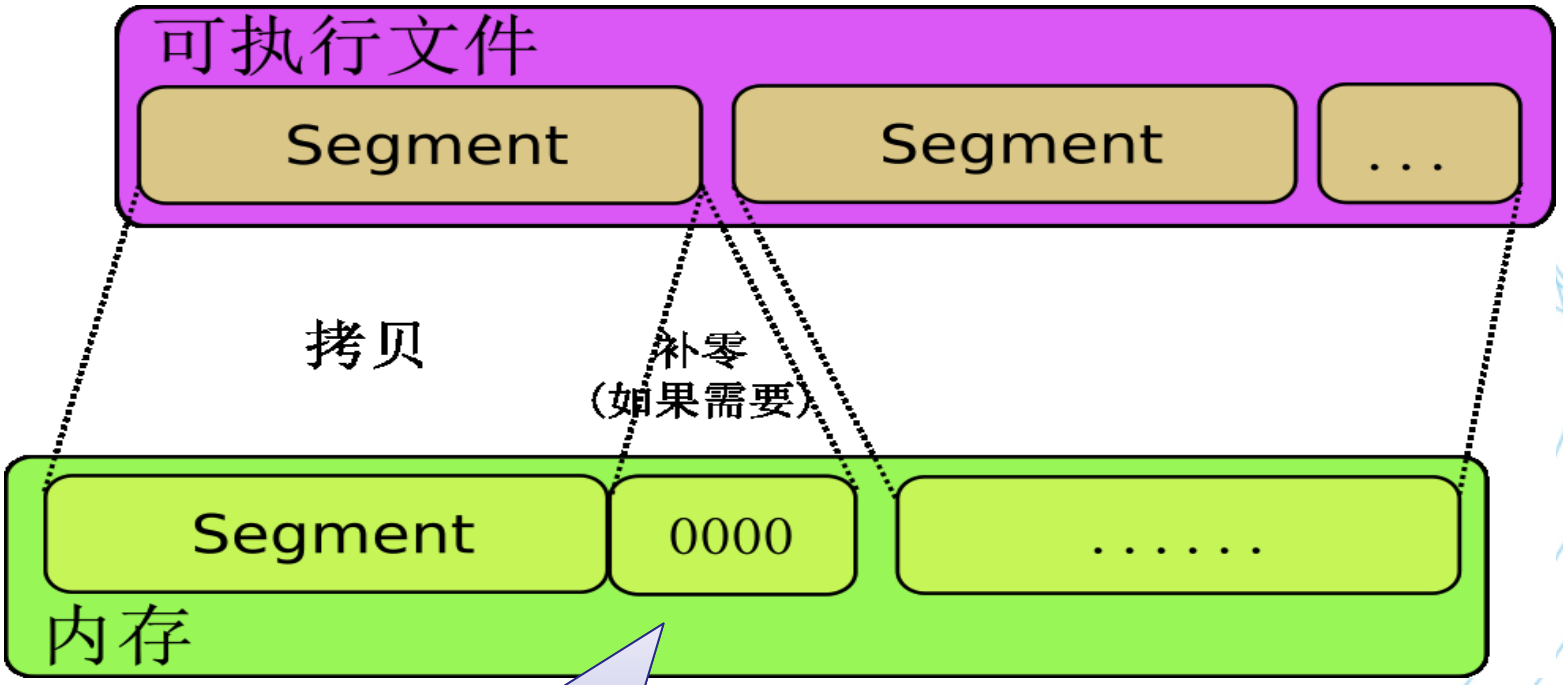
细节：

- 一个segment在文件中的大小是小于等于其在内存中的大小。
- 如果在文件中的大小小于在内存中的大小，那么在载入内存时通过补零使其达到其在内存中应有的大小。

Member	Text	Data
p_type	PT_LOAD	PT_LOAD
p_offset	0x100	0x2bf00
p_vaddr	0x8048100	0x8074f00
p_paddr	unspecified	unspecified
p_filesz	0x2be00	0x4e00
p_memsz	0x2be00	0x5e24
p_flags	PF_R+PF_X	PF_R+PF_W+PF_X
p_align	0x1000	0x1000

程序的装载和运行

代码段和数据段都在segment中



文件大小小于内存大小，补0

程序的装载流程

- 读取ELF头部的魔数(Magic Number)，以确认该文件确实是ELF文件。
  - ELF文件的头四个字节依次为’0x7f’、’E’、‘L’、‘F’。
  - 加载器会首先对比这四个字节，若不一致，则报错。
- 找到段表项。
  - ELF头部会给出的段表起始位置在文件中的偏移，段表项的大小，以及段表包含了多少项。根据这些信息可以找到每一个段表项。
- 对于每个段表项解析出各个段应当被加载的虚地址，在文件中的偏移。以及在内存中的大小和在文件中的大小。（段在文件中的大小小于等于内存中的大小）。

程序的装载流程

- 对于每一个段，根据其在内存中的大小，为其分配足够的物理页，并映射到指定的虚地址上。再将文件中的内容拷贝到内存中。
- 若ELF中记录的段在内存中的大小大于在文件中的大小，则多出来的部分用0进行填充。
- 设置进程控制块中的PC为ELF文件中记载的入口地址。
- 控制权交给进程开始执行！

可执行文件的内存映像(x86)