操作系统 Operation System

内存管理

原仓周
yuancz@buaa.edu.cn
TEL：82338521

内容提要

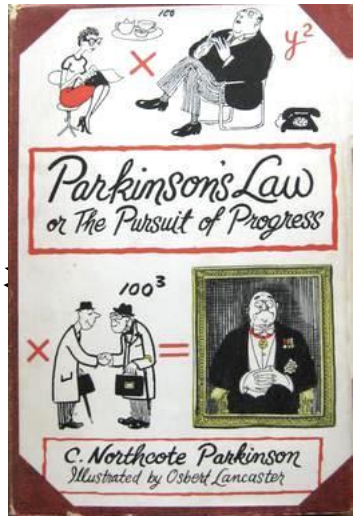
- 存储管理基础
- 页式内存管理
- 段式内存管理
- 虚拟存储管理
- 存储管理实例

内容提要

- 存储管理基础
  - 存储器硬件发展
  - 存储管理的功能
    - C程序实例分析(MIPS)
  - 存储器分配方法
    - 单道程序的内存管理
    - 多道程序的内存管理
- 页式内存管理
- 段式内存管理
- 虚拟存储管理
- 存储管理实例

存储器管理的目标

- 研究对象：以内存为中心的存储资源
  - 程序在内存中运行
- 用户角度（程序员）
  - 容量大
  - 速度快（性能）
  - 独立拥有，不受干
    （安全）
- 资源管理角度
  - 为多用户提供服务
  - 效率、利用率、能耗



帕金森定律(Parkinson)

work expands so as to fill the time available for its completion

无论存储器空间有多大，程序都能将其耗尽

存储器硬件

- RAM: Random Access Memory
  - SRAM
  - DRAM
  - SDRAM、DDR SDRAM
- ROM: Read Only Memory
  - PROM、EPROM、EEPROM等
- Flash memory
  - NOR
  - NAND
- Disk
- Tape

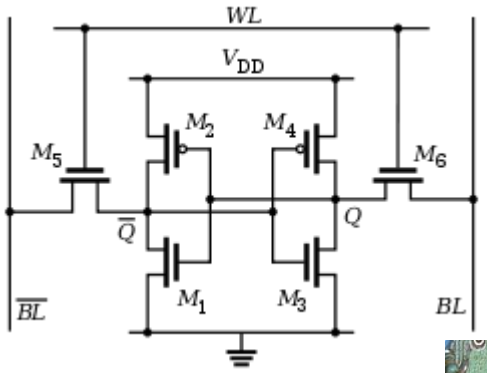
易失性

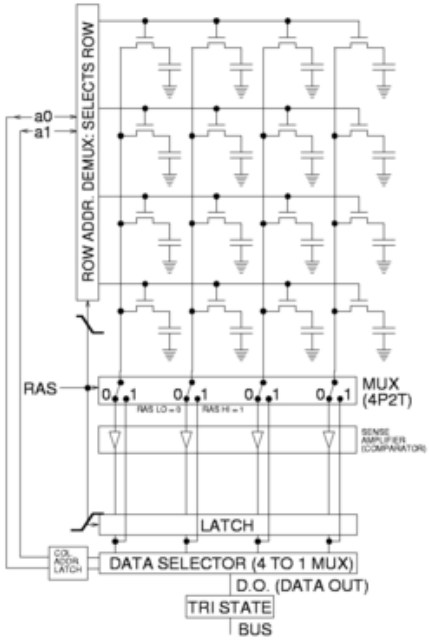
非易失性

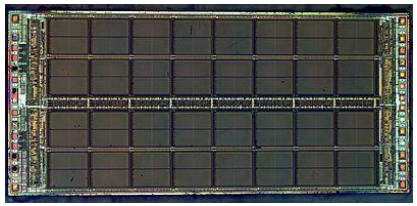
外存

存储器硬件

- 存储器的功能：保存数据，存储器的发展方向是高速、大容量和小体积。如：内存在访问速度方面的发展：DRAM、SDRAM（DDR）、SRAM等；硬盘技术在大容量方面的发展：接口标准、存储密度等；
  - DDR4理论上每根DIMM模块能达到512GiB的容量
  - DDR4-3200带宽可达51.2GB/s







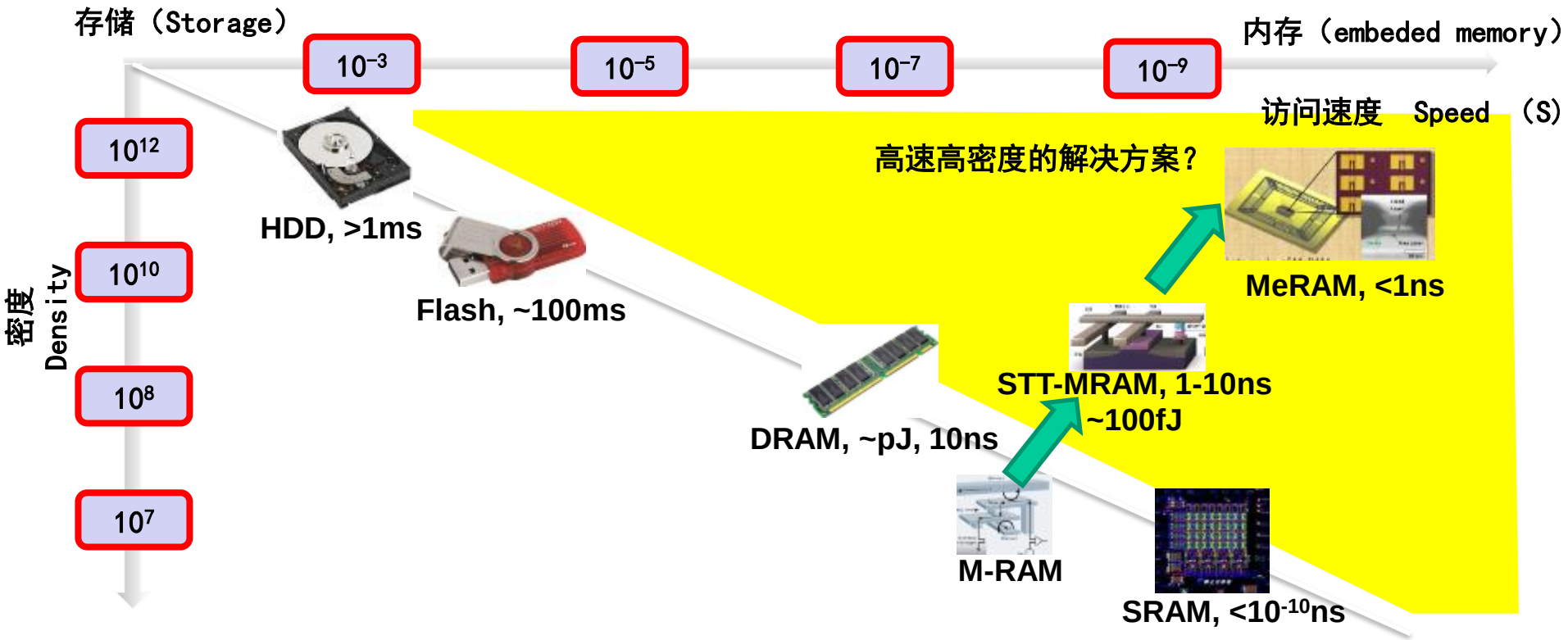
存储器硬件

- 静态存储器（SRAM）：读写速度快，生产成本高，多用于容量较小的高速缓冲存储器。
- 动态存储器（DRAM）：读写速度较慢，集成度高，生产成本低，多用于容量较大的主存储器。

	SRAM	DRAM
存储信息方式	触发器（RS）	电容
破坏性读出	否	是
定期刷新	不需要	需要
送地址方式	行列同时送	行列分两次送
运行速度	快	慢
发热量	大	小
存储成本	高	低
集成度	低	高

蕴含巨大的挑战

- 非易失性存储器件（NVM）的发展是否有突破的机会？
- SSD（Flash）、PCM、忆阻器、自旋电子器件



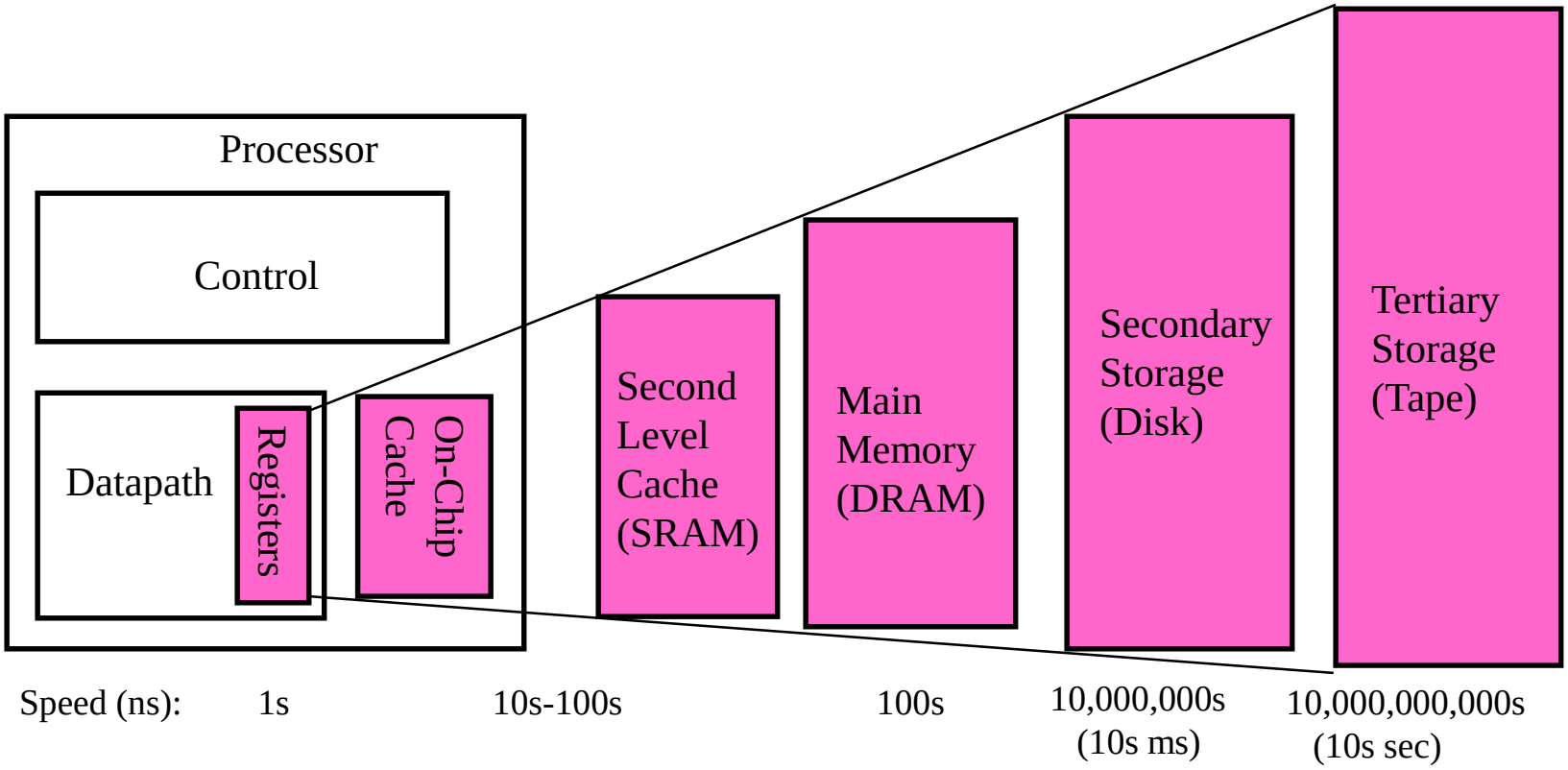
K.L.Wang, Spin orbit interaction engineering of magnetic memory for energy efficient electronics systems. NVMTS 2015

存储组织

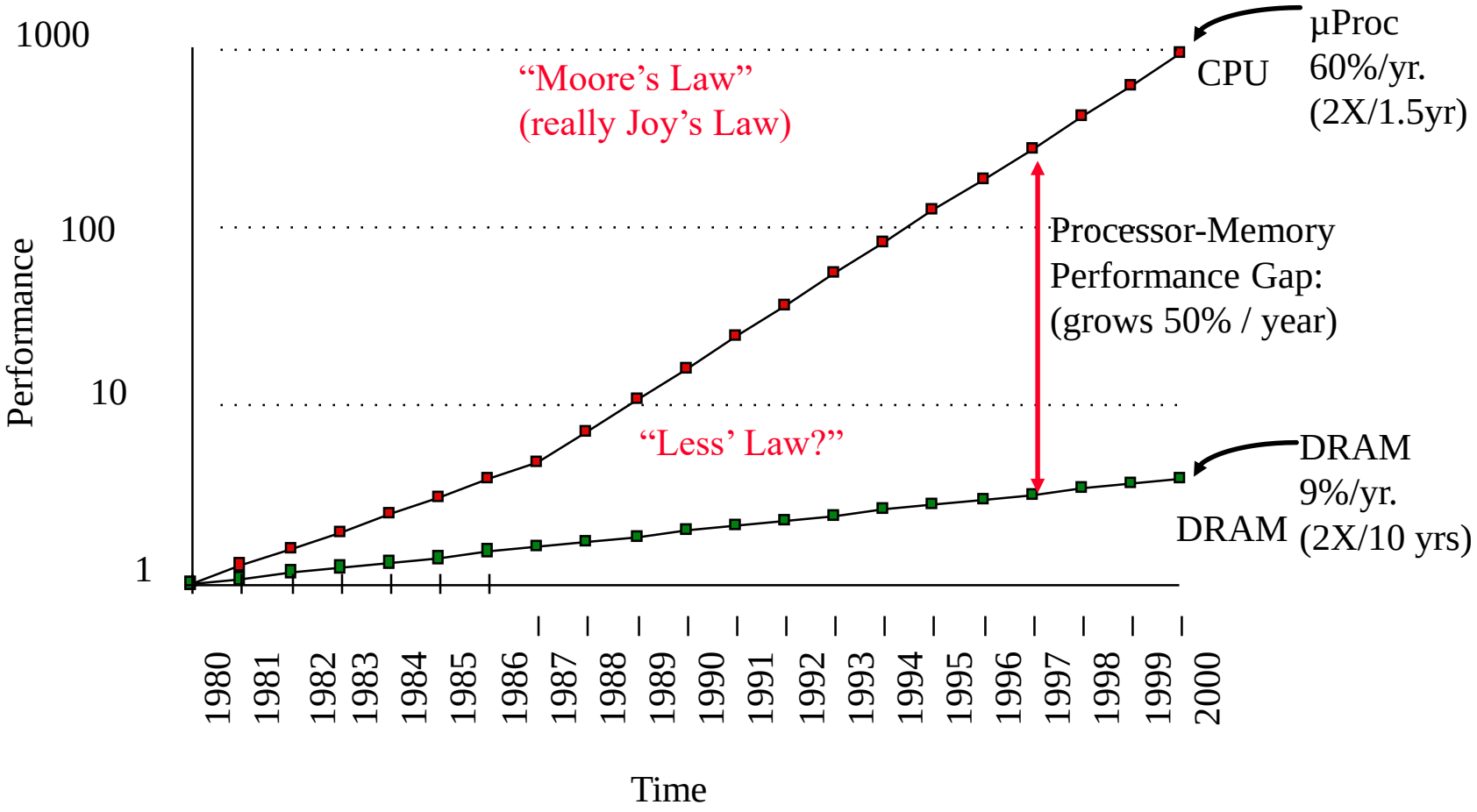
- 存储组织：在存储技术和CPU寻址技术许可的范围内组织合理的存储结构。其依据是访问速度匹配关系、容量要求和价格。
- 例如：“寄存器-内存-外存”结构和“寄存器-缓存-内存-外存”结构
- 典型的层次式存储组织：访问速度越来越慢，容量越来越大，价格越来越便宜
- 最佳状态应是各层次的存储器都处于均衡的繁忙状态（如：缓存命中率正好使主存读写保持繁忙）

存储层次结构


寄存器(register)
快速缓存(cache)
主存(primary storage)
外存(secondary storage)


Processor
Control
Datapath
Registers
On-Chip Cache
Second Level Cache (SRAM)
Main Memory (DRAM)
Secondary Storage (Disk)
Tertiary Storage (Tape)
Speed (ns):
1s
10s-100s
100s
10,000,000s (10s ms)
10,000,000,000s (10s sec)
Size (bytes):
100s
Ks-Ms
Ms
Gs
Ts

Processor-DRAM Memory Gap (latency)

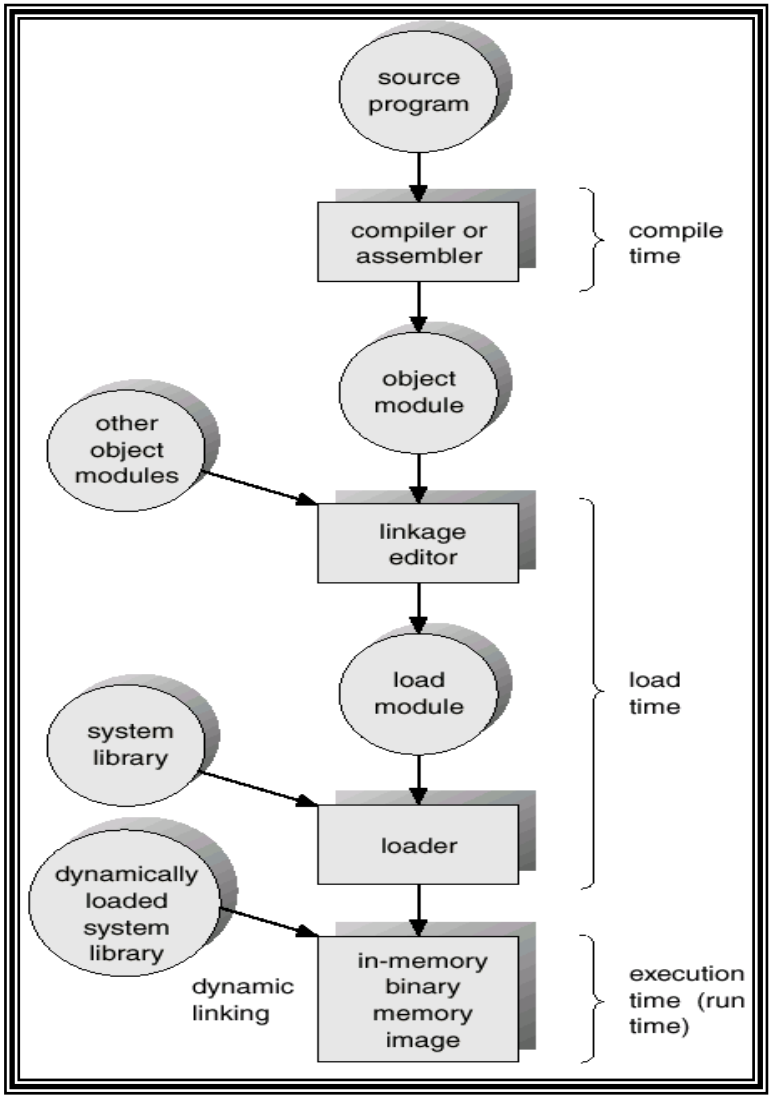


存储管理的功能

- 存储分配和回收：是存储管理的主要内容。讨论其算法和相应的数据结构。
- 地址变换：可执行文件生成中的链接技术、程序加载时的重定位技术，进程运行时硬件和软件的地址变换技术和机构。
- 存储共享和保护：代码和数据共享，对地址空间的访问权限（读、写、执行）。
- 存储器扩充：它涉及存储器的逻辑组织和物理组织；
  - 由应用程序控制：覆盖；
  - 由OS控制：交换（整个进程空间），请求调入和预调入（部分进程空间）

程序的装入和链接

- 编译
- 链接
- 装入



program.c

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

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

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;

globally visible by all modules (common, bss)

static int some_local_variable;

global/local: visible by all functions within the current module, but not outside the module (data)

```
main () {
        int some_stack_variable;
        allocated on the stack, visible only within the block
        ...
}
```

```
gcc program.c extras.c
./a.out

gcc -c program.c => program.o
gcc -c extras.c => extras.o
gcc program.o functions.o -o exe
./exe
```

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

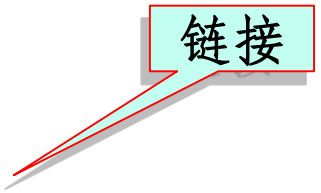
编译

```
gcc version 5.2.0 (crosstool-NG crosstool-ng-1.22.0)
COLLECT_GCC_OPTIONS='-o' 'exe' '-v' '-mabi=32' '-mllsc' '-mplt' '-mno-shared' '-EB'
 /OSLAB/compiler/mips-malta-linux-gnu/bin/../libexec/gcc/mips-malta-linux-gnu/5.2.0/cc1 -quiet -v -iprefix
/OSLAB/compiler/mips-malta-linux-gnu/bin/../lib/gcc/mips-malta-linux-gnu/5.2.0/ -isysroot
/OSLAB/compiler/mips-malta-linux-gnu/bin/../mips-malta-linux-gnu/sysroot program.c -meb -quiet -
dumpbase program.c -mabi=32 -mllsc -mplt -mno-shared -auxbase program -version -o /tmp/ccntcnFQ.s

GNU C11 (crosstool-NG crosstool-ng-1.22.0) version 5.2.0 (mips-malta-linux-gnu)
        compiled by GNU C version 4.6.3, GMP version 6.0.0, MPFR version 3.1.3, MPC version 1.0.3
GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
ignoring duplicate directory "/OSLAB/compiler/mips-malta-linux-gnu/bin/../lib/gcc/../../lib/gcc/mips-malta-
linux-gnu/5.2.0/include"
ignoring nonexistent directory "/OSLAB/compiler/mips-malta-linux-gnu/bin/../mips-malta-linux-
gnu/sysroot/home/wangluming/x-tools/mips-malta-linux-gnu/mips-malta-linux-gnu/sysroot/include"
ignoring duplicate directory "/OSLAB/compiler/mips-malta-linux-gnu/bin/../lib/gcc/../../lib/gcc/mips-malta-
linux-gnu/5.2.0/include-fixed"
ignoring duplicate directory "/OSLAB/compiler/mips-malta-linux-gnu/bin/../lib/gcc/../../lib/gcc/mips-malta-
linux-gnu/5.2.0/../../../../mips-malta-linux-gnu/include"
#include "..." search starts here:
#include <...> search starts here:
 /OSLAB/compiler/mips-malta-linux-gnu/bin/../lib/gcc/mips-malta-linux-gnu/5.2.0/include
 /OSLAB/compiler/mips-malta-linux-gnu/bin/../lib/gcc/mips-malta-linux-gnu/5.2.0/include-fixed
 /OSLAB/compiler/mips-malta-linux-gnu/bin/../lib/gcc/mips-malta-linux-gnu/5.2.0/../../../../mips-malta-linux-
gnu/include
 /OSLAB/compiler/mips-malta-linux-gnu/bin/../mips-malta-linux-gnu/sysroot/usr/include
End of search list.
```

GNU C11 (crosstool-NG crosstool-ng-1.22.0) version 5.2.0 (mips-malta-linux-gnu)
compiled by GNU C version 4.6.3, GMP version 6.0.0, MPFR version 3.1.3, MPC version 1.0.3
GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
Compiler executable checksum: a0212981a25e6bcf7c0ea0e0513f0ef0
COLLECT_GCC_OPTIONS='-o' 'exe' '-v' '-mabi=32' '-mllsc' '-mplt' '-mno-shared' '-EB'
/OSLAB/compiler/mips-malta-linux-gnu/bin/../lib/gcc/mips-malta-linux-gnu/5.2.0/../../../../mips-malta-linux-gnu/bin/as -v -EB -O1 -no-mdebug -mabi=32 -mno-shared -call_nonpic **-o /tmp/cc0eljh2.o** /tmp/ccntcnFQ.s
GNU assembler version 2.25.1 (mips-malta-linux-gnu) using BFD version (crosstool-NG crosstool-ng-1.22.0) 2.25.1
COLLECT_GCC_OPTIONS='-o' 'exe' '-v' '-mabi=32' '-mllsc' '-mplt' '-mno-shared' '-EB'
/OSLAB/compiler/mips-malta-linux-gnu/bin/../libexec/gcc/mips-malta-linux-gnu/5.2.0/cc1 -quiet -v -iprefix /OSLAB/compiler/mips-malta-linux-gnu/bin/../lib/gcc/mips-malta-linux-gnu/5.2.0/ -isysroot /OSLAB/compiler/mips-malta-linux-gnu/bin/../mips-malta-linux-gnu/sysroot **extras.c** -meb -quiet -dumpbase extras.c -mabi=32 -mllsc -mplt -mno-shared -auxbase extras -version -o /tmp/ccntcnFQ.s
GNU C11 (crosstool-NG crosstool-ng-1.22.0) version 5.2.0 (mips-malta-linux-gnu)
compiled by GNU C version 4.6.3, GMP version 6.0.0, MPFR version 3.1.3, MPC version 1.0.3
GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
ignoring duplicate directory "/OSLAB/compiler/mips-malta-linux-gnu/bin/../lib/gcc/../../lib/gcc/mips-malta-linux-gnu/5.2.0/include"
ignoring nonexistent directory "/OSLAB/compiler/mips-malta-linux-gnu/bin/../mips-malta-linux-gnu/sysroot/home/wangluming/x-tools/mips-malta-linux-gnu/mips-malta-linux-gnu/sysroot/include"
ignoring duplicate directory "/OSLAB/compiler/mips-malta-linux-gnu/bin/../lib/gcc/../../lib/gcc/mips-malta-linux-gnu/5.2.0/include-fixed"
ignoring duplicate directory "/OSLAB/compiler/mips-malta-linux-gnu/bin/../lib/gcc/../../lib/gcc/mips-malta-linux-gnu/5.2.0/../../../../mips-malta-linux-gnu/include"

汇编

```
#include "..." search starts here:
#include <...> search starts here:
 /OSLAB/compiler/mips-malta-linux-gnu/bin/../lib/gcc/mips-malta-linux-gnu/5.2.0/include
 /OSLAB/compiler/mips-malta-linux-gnu/bin/../lib/gcc/mips-malta-linux-gnu/5.2.0/include-fixed
 /OSLAB/compiler/mips-malta-linux-gnu/bin/../lib/gcc/mips-malta-linux-gnu/5.2.0/../../../../mips-malta-linux-gnu/include
 /OSLAB/compiler/mips-malta-linux-gnu/bin/../mips-malta-linux-gnu/sysroot/usr/include
End of search list.
GNU C11 (crosstool-NG crosstool-ng-1.22.0) version 5.2.0 (mips-malta-linux-gnu)
        compiled by GNU C version 4.6.3, GMP version 6.0.0, MPFR version 3.1.3, MPC version 1.0.3
GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
Compiler executable checksum: a0212981a25e6bcf7c0ea0e0513f0ef0
COLLECT_GCC_OPTIONS='-o' 'exe' '-v' '-mabi=32' '-mllsc' '-mplt' '-mno-shared' '-EB'
 /OSLAB/compiler/mips-malta-linux-gnu/bin/../lib/gcc/mips-malta-linux-gnu/5.2.0/../../../../mips-malta-linux-gnu/bin/as -v -EB -O1 -no-mdebug -mabi=32 -mno-shared -call_nonpic -o /tmp/cc0fSkXd.o
/tmp/ccntcnFQ.s
GNU assembler version 2.25.1 (mips-malta-linux-gnu) using BFD version (crosstool-NG crosstool-ng-1.22.0) 2.25.1
COMPILER_PATH=/OSLAB/compiler/mips-malta-linux-gnu/bin/../libexec/gcc/mips-malta-linux-gnu/5.2.0/:/OSLAB/compiler/mips-malta-linux-gnu/bin/../libexec/gcc/:/OSLAB/compiler/mips-malta-linux-gnu/bin/../lib/gcc/mips-malta-linux-gnu/5.2.0/../../../../mips-malta-linux-gnu/bin/
LIBRARY_PATH=/OSLAB/compiler/mips-malta-linux-gnu/bin/../lib/gcc/mips-malta-linux-gnu/5.2.0/:/OSLAB/compiler/mips-malta-linux-gnu/bin/../lib/gcc/:/OSLAB/compiler/mips-malta-linux-gnu/bin/../lib/gcc/mips-malta-linux-gnu/5.2.0/../../../../mips-malta-linux-gnu/lib/:/OSLAB/compiler/mips-malta-linux-gnu/bin/../mips-malta-linux-gnu/sysroot/lib/:/OSLAB/compiler/mips-malta-linux-gnu/bin/../mips-malta-linux-gnu/sysroot/usr/lib/
```



COLLECT_GCC_OPTIONS='-o' 'exe' '-v' '-mabi=32' '-mllsc' '-mplt' '-mno-shared' '-EB'
/OSLAB/compiler/mips-malta-linux-gnu/bin/../libexec/gcc/mips-malta-linux-gnu/5.2.0/collect2 -plugin
/OSLAB/compiler/mips-malta-linux-gnu/bin/../libexec/gcc/mips-malta-linux-gnu/5.2.0/liblto_plugin.so -
plugin-opt=/OSLAB/compiler/mips-malta-linux-gnu/bin/../libexec/gcc/mips-malta-linux-gnu/5.2.0/lto-
wrapper -plugin-opt=-fresolution=/tmp/ccvlN9Dp.res -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-
through=-lgcc_s -plugin-opt=-pass-through=-lc -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-
through=-lgcc_s --sysroot=/OSLAB/compiler/lib/ld.so.1 mips-malta-linux-gnu/bin/../mips-malta-linux-
gnu/sysroot --eh-frame-hdr -EB -dynamic-linker /-melf32btsmip -o exe /OSLAB/compiler/mips-malta-
linux-gnu/bin/../mips-malta-linux-gnu/sysroot/usr/lib/crt1.o /OSLAB/compiler/mips-malta-linux-
gnu/bin/../mips-malta-linux-gnu/sysroot/usr/lib/crti.o /OSLAB/compiler/mips-malta-linux-
gnu/bin/../lib/gcc/mips-malta-linux-gnu/5.2.0/crtbegin.o -L/OSLAB/compiler/mips-malta-linux-
gnu/bin/../lib/gcc/mips-malta-linux-gnu/5.2.0 -L/OSLAB/compiler/mips-malta-linux-gnu/bin/../lib/gcc -
L/OSLAB/compiler/mips-malta-linux-gnu/bin/../lib/gcc/mips-malta-linux-gnu/5.2.0/../../../../mips-malta-
linux-gnu/lib -L/OSLAB/compiler/mips-malta-linux-gnu/bin/../mips-malta-linux-gnu/sysroot/lib -
L/OSLAB/compiler/mips-malta-linux-gnu/bin/../mips-malta-linux-gnu/sysroot/usr/lib /tmp/cc0eljh2.o
/tmp/cc0fSkXd.o -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed
/OSLAB/compiler/mips-malta-linux-gnu/bin/../lib/gcc/mips-malta-linux-gnu/5.2.0/crtend.o
/OSLAB/compiler/mips-malta-linux-gnu/bin/../mips-malta-linux-gnu/sysroot/usr/lib/crtn.o

```
        .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 am 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
        .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"
```

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 am 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
```

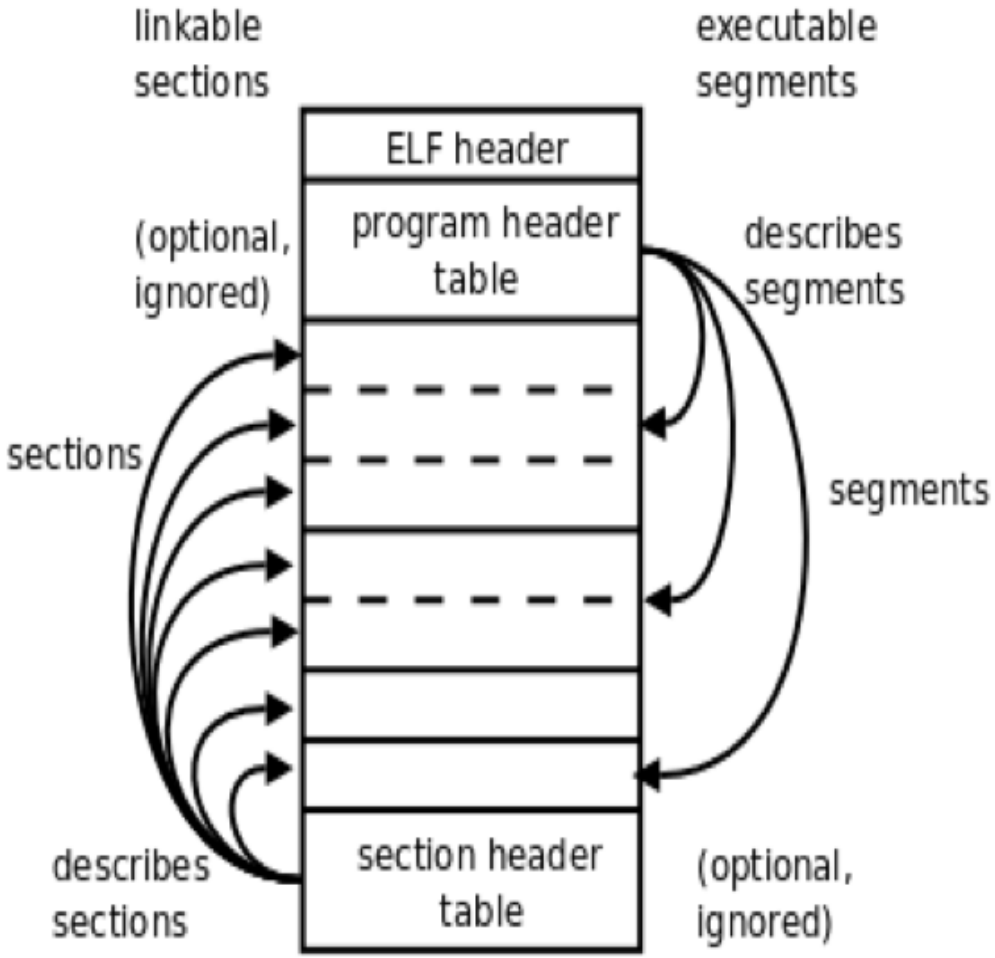
生成的汇编文件

```
.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

ELF(Executable and Linkable Format)-可执行文件格式



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.datal	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.rodatal	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

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。

符号表

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

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

Main.o

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
13:	00000004	4	OBJECT	GLOBAL	DEFAULT	COM	some_global_variable
14:	00000000	96	FUNC	GLOBAL	DEFAULT	1	main
15:	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND	read_something
16:	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND	do_something
17:	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND	write_something

extras.o

12:	00000000	68	FUNC	GLOBAL	DEFAULT	1	read_something
13:	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND	__isoc99_scanf
14:	00000044	52	FUNC	GLOBAL	DEFAULT	1	do_something
15:	00000078	84	FUNC	GLOBAL	DEFAULT	1	write_something
16:	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND	some_global_variable
17:	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND	printf

a.out

61: 00400744 52 FUNC GLOBAL DEFAULT 13 do_something

62: 00410a1c 4 OBJECT GLOBAL DEFAULT 26 some_global_variable

67: 00400700 68 FUNC GLOBAL DEFAULT 13 read_something

74: 00400778 84 FUNC GLOBAL DEFAULT 13 write_something

77: 00400990 0 FUNC GLOBAL DEFAULT [MIPS PLT] UND
__libc_start_main@@GLIBC_

80: 00410a40 0 NOTYPE GLOBAL DEFAULT 27 _end

81: 00410a1c 0 NOTYPE GLOBAL DEFAULT 26 __bss_start

82: 004006a0 96 FUNC GLOBAL DEFAULT 13 main

Main.o

13: 00000004 4 OBJECT GLOBAL DEFAULT COM some_global_variable

14: 00000000 96 FUNC GLOBAL DEFAULT 1 main

15: 00000000 0 NOTYPE GLOBAL DEFAULT UND read_something

16: 00000000 0 NOTYPE GLOBAL DEFAULT UND do_something

17: 00000000 0 NOTYPE GLOBAL DEFAULT UND write_something

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

```
.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)
```

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。
    - E重定位目标文件
    - U未解析符号
    - D已定义符号
- 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。

```
gcc -Wall -o link program.o

program.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
```

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
```

```
 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

- R_MIPS_HI16 Word32 %high(AHL + S)
- R_MIPS_LO16 Word32 AHL+S

- S=ADDR(r.symbol)=00410a1c
- AHL=0

- %high(AHL + S)=0x41
- AHL+S=0x0a1c

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

Extras.o

```
00000078 <write_something>:
  78:  27bdffe0      addiu   sp,sp,-32
  7c:  afbf001c      sw      ra,28(sp)
  80:  afbe0018      sw      s8,24(sp)
  84:  03a0f021      move    s8,sp
  88:  afc40020      sw      a0,32(s8)
  8c:  3c020000      lui     v0,0x0
  90:  8c420000      lw      v0,0(v0)
  94:  00000000      nop
  98:  00403021      move    a2,v0
  9c:  8fc50020      lw      a1,32(s8)
  a0:  3c020000      lui     v0,0x0
  a4:  24440004      addiu   a0,v0,4
  a8:  0c000000      jal     0 <read_something>
  ac:  00000000      nop
  b0:  00000000      nop
  b4:  03c0e821      move    sp,s8
  b8:  8fbf001c      lw      ra,28(sp)
  bc:  8fbe0018      lw      s8,24(sp)
  c0:  27bd0020      addiu   sp,sp,32
  c4:  03e00008      jr      ra
  c8:  00000000      nop
  cc:  00000000      nop
```

```
void write_something (const char* str) {
        printf ("%s: %d\n", str, some_global_variable);
}
```

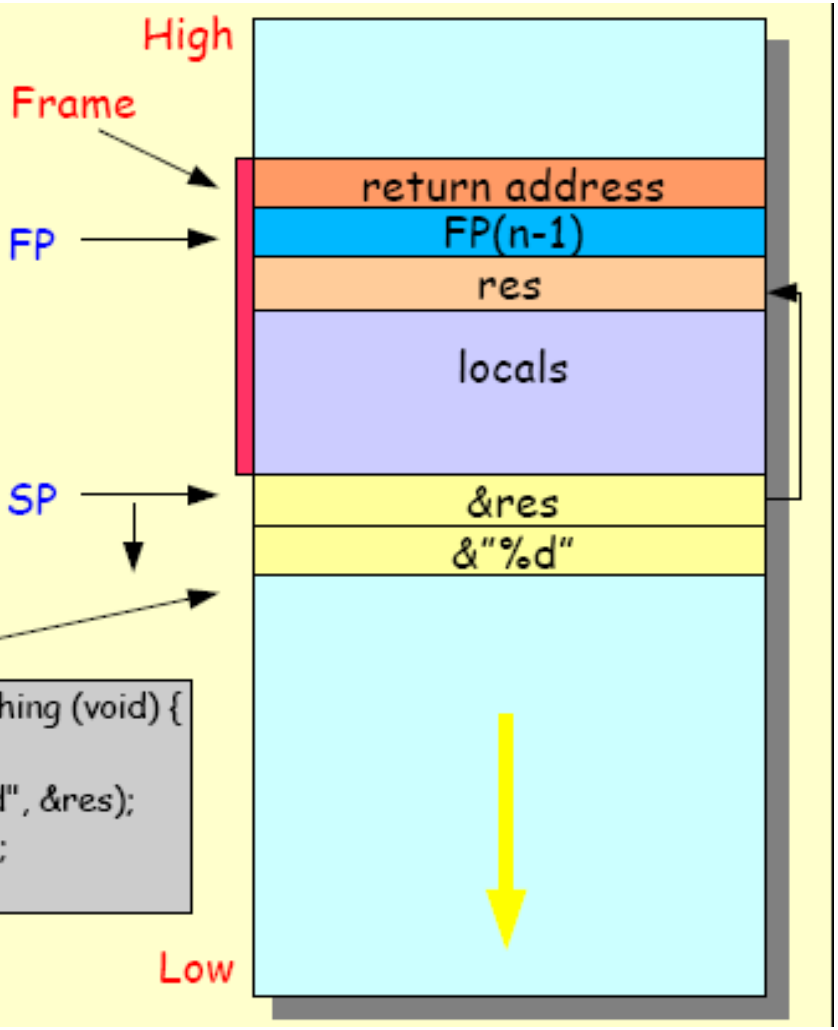
exe

62: 00410a1c 4 OBJECT GLOBAL DEFAULT 26 some_global_variable

```
00400778 <write_something>:
  400778:   27bdffe0    addiu   sp,sp,-32
  40077c:   afbf001c    sw      ra,28(sp)
  400780:   afbe0018    sw      s8,24(sp)
  400784:   03a0f021    move    s8,sp
  400788:   afc40020    sw      a0,32(s8)
  40078c:   3c020041    lui     v0,0x41
  400790:   8c420a1c    lw      v0,2588(v0)
  400794:   00000000    nop
  400798:   00403021    move    a2,v0
  40079c:   8fc50020    lw      a1,32(s8)
  4007a0:   3c020040    lui     v0,0x40
  4007a4:   24440944    addiu   a0,v0,2372
  4007a8:   0c100260    jal     400980 <printf@plt>
  4007ac:   00000000    nop
  4007b0:   00000000    nop
  4007b4:   03c0e821    move    sp,s8
  4007b8:   8fbf001c    lw      ra,28(sp)
  4007bc:   8fbe0018    lw      s8,24(sp)
  4007c0:   27bd0020    addiu   sp,sp,32
  4007c4:   03e00008    jr      ra
  4007c8:   00000000    nop
  4007cc:   00000000    nop
```

```
        .file   "extras.c"
        .version "01.01"
gcc2_compiled.:
        .section .rodata
.LC0:
        .string "%d"
.text
        .align 4
.globl read_something
        .type    read_something,@function
read_something:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        leal    -4(%ebp), %eax
        pushl   %eax
        pushl   $.LC0
        call    scanf
        movl    -4(%ebp), %eax
        leave
        ret
....
```

```
int read_something (void) {
        int res;
        scanf ("%d", &res);
        return res;
}
```



```
00000000h: 7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00 ; ELF.............
00000010h: 01 00 03 00 01 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: C4 00 00 00 00 00 00 00 34 00 00 00 00 00 28 00 ; ?.......4.....(.
00000030h: 09 00 06 00 55 89 E5 83 EC 08 83 E4 F0 B8 00 00 ; ....U夂波.稞莺..
00000040h: 00 00 29 C4 E8 FC FF FF FF B8 00 00 00 00 C9 C3 ; ..)焉?    ?...擅
00000050h: 01 00 00 00 02 00 00 00 00 47 43 43 3A 20 28 47 ; .........GCC: (G
00000060h: 4E 55 29 20 33 2E 32 2E 32 20 32 30 30 33 30 32 ; NU) 3.2.2 200302
00000070h: 32 32 20 28 52 65 64 20 48 61 74 20 4C 69 6E 75 ; 22 (Red Hat Linu
00000080h: 78 20 33 2E 32 2E 32 2D 35 29 00 00 2E 73 79 6D ; x 3.2.2-5)...sym
00000090h: 74 61 62 00 2E 73 74 72 74 61 62 00 2E 73 68 73 ; tab..strtab..shs
000000a0h: 74 72 74 61 62 00 2E 72 65 6C 2E 74 65 78 74 00 ; trtab..rel.text.
000000b0h: 2E 64 61 74 61 00 2E 62 73 73 00 2E 63 6F 6D 6D ; .data..bss..comm
000000c0h: 65 6E 74 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ent.............
000000d0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000000e0h: 00 00 00 00 00 00 00 00 00 00 00 00 1F 00 00 00 ; ................
000000f0h: 01 00 00 00 06 00 00 00 00 00 00 00 34 00 00 00 ; ............4...
00000100h: 1C 00 00 00 00 00 00 00 00 00 00 00 04 00 00 00 ; ................
00000110h: 00 00 00 00 1B 00 00 00 09 00 00 00 00 00 00 00 ; ................
00000120h: 00 00 00 00 D4 02 00 00 08 00 00 00 07 00 00 00 ; ....?...........
00000130h: 01 00 00 00 04 00 00 00 08 00 00 00 25 00 00 00 ; ............%...
00000140h: 01 00 00 00 03 00 00 00 00 00 00 00 50 00 00 00 ; ............P...
00000150h: 08 00 00 00 00 00 00 00 00 00 00 00 04 00 00 00 ; ................
00000160h: 00 00 00 00 2B 00 00 00 08 00 00 00 03 00 00 00 ; ....+...........
00000170h: 00 00 00 00 58 00 00 00 00 00 00 00 00 00 00 00 ; ....X...........
00000180h: 00 00 00 00 04 00 00 00 00 00 00 00 30 00 00 00 ; ............0...
00000190h: 01 00 00 00 00 00 00 00 00 00 00 00 58 00 00 00 ; ............X...
000001a0h: 33 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 ; 3...............
000001b0h: 00 00 00 00 11 00 00 00 03 00 00 00 00 00 00 00 ; ................
000001c0h: 00 00 00 00 8B 00 00 00 39 00 00 00 00 00 00 00 ; ....?..9........
000001d0h: 00 00 00 00 01 00 00 00 00 00 00 00 01 00 00 00 ; ................
000001e0h: 02 00 00 00 00 00 00 00 00 00 00 00 2C 02 00 00 ; ............,...
000001f0h: 90 00 00 00 08 00 00 00 06 00 00 00 04 00 00 00 ; ?...............
00000200h: 10 00 00 00 09 00 00 00 03 00 00 00 00 00 00 00 ; ................
00000210h: 00 00 00 00 BC 02 00 00 16 00 00 00 00 00 00 00 ; ....?...........
00000220h: 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 ; ................
```

```
00000000h: 7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00 ;  ELF............
00000010h: 02 00 03 00 01 00 00 00 40 83 04 08 34 00 00 00 ; ........@?.4...
00000020h: 74 2A 00 00 00 00 00 00 34 00 20 00 06 00 28 00 ; t*......4. ...(.
00000030h: 1D 00 1A 00 06 00 00 00 34 00 00 00 34 80 04 08 ; ........4...4€..
00000040h: 34 80 04 08 C0 00 00 00 C0 00 00 00 05 00 00 00 ; 4€..?..?......
00000050h: 04 00 00 00 03 00 00 00 F4 00 00 00 F4 80 04 08 ; ........?..劒..
00000060h: F4 80 04 08 13 00 00 00 13 00 00 00 04 00 00 00 ; 劒.............
00000070h: 01 00 00 00 01 00 00 00 00 00 00 00 00 80 04 08 ; ............€..
00000080h: 00 80 04 08 4D 05 00 00 4D 05 00 00 05 00 00 00 ; .€..M...M.......
00000090h: 00 10 00 00 01 00 00 00 50 05 00 00 50 95 04 08 ; ........P...P?.
000000a0h: 50 95 04 08 10 01 00 00 30 01 00 00 06 00 00 00 ; P?.....0.......
000000b0h: 00 10 00 00 02 00 00 00 64 05 00 00 64 95 04 08 ; ........d...d?.
000000c0h: 64 95 04 08 C8 00 00 00 C8 00 00 00 06 00 00 00 ; d?.?..?......
000000d0h: 04 00 00 00 04 00 00 00 08 01 00 00 08 81 04 08 ; ............?.
000000e0h: 08 81 04 08 20 00 00 00 20 00 00 00 04 00 00 00 ; .?. ... .......
000000f0h: 04 00 00 00 2F 6C 69 62 2F 6C 64 2D 6C 69 6E 75 ; ..../lib/ld-linu
00000100h: 78 2E 73 6F 2E 32 00 00 04 00 00 00 10 00 00 00 ; x.so.2.........
00000110h: 01 00 00 00 47 4E 55 00 00 00 00 00 02 00 00 00 ; ....GNU.........
00000120h: 02 00 00 00 05 00 00 00 03 00 00 00 08 00 00 00 ; ................
00000130h: 07 00 00 00 04 00 00 00 06 00 00 00 00 00 00 00 ; ................
00000140h: 00 00 00 00 01 00 00 00 02 00 00 00 03 00 00 00 ; ................
00000150h: 00 00 00 00 05 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000160h: 00 00 00 00 00 00 00 00 00 00 00 00 51 00 00 00 ; ............Q...
00000170h: E4 82 04 08 95 00 00 00 22 00 00 00 2A 00 00 00 ; 鏊..?..".*...
00000180h: F4 82 04 08 34 00 00 00 12 00 00 00 12 00 00 00 ; 劒..4...........
00000190h: 04 83 04 08 25 00 00 00 22 00 00 00 3F 00 00 00 ; .?.%..."...?...
000001a0h: 14 83 04 08 D3 00 00 00 12 00 00 00 0B 00 00 00 ; .?.?...........
000001b0h: 24 83 04 08 32 00 00 00 12 00 00 00 30 00 00 00 ; $?.2.......0...
000001c0h: 34 85 04 08 04 00 00 00 11 00 0E 00 67 00 00 00 ; 4?.........g...
000001d0h: 00 00 00 00 00 00 00 00 20 00 00 00 00 6C 69 62 ; ........ ....lib
000001e0h: 63 2E 73 6F 2E 36 00 70 72 69 6E 74 66 00 5F 5F ; c.so.6.printf.__
000001f0h: 64 65 72 65 67 69 73 74 65 72 5F 66 72 61 6D 65 ; deregister_frame
00000200h: 5F 69 6E 66 6F 00 73 63 61 6E 66 00 5F 49 4F 5F ; _info.scanf._IO_
00000210h: 73 74 64 69 6E 5F 75 73 65 64 00 5F 5F 6C 69 62 ; stdin_used.__lib
00000220h: 63 5F 73 74 61 72 74 5F 6D 61 69 6E 00 5F 5F 72 ; c_start_main.__r
```


ELF头
程序头表
.init
.text
.rodata
.data
.bss
.symtab
.debug
节头表

SIGNATURE

```
000 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
    --  E  L  F 32 LE FW ------------------------
```

ELF HEADER

```
010 01 00 03 00 01 00 00 00 00 00 00 00 00 00 00 00
     0001  0003    00000001    00000000    00000000
```

0001: relocatable module

0003: target architecture

00000001

00000000: starting address for execution (executable module only)

00000000: program header (executable module only)

```
020 04 01 00 00 00 00 00 00 34 00 00 00 00 00 28 00
       00000104    00000000  0034  0000  0000  0028
```

00000104: SHT offset

00000000: CPU flags

0034: hdr length

0000: no PHT

0000

0028: SHT entry size

SHT = Section Header Table

PHT = Program Header Table

```
030 0b 00 08 00
    000b  0008
```

000b: number of entries in SHT

0008: index (into SHT) of the string section containing section names

- ./exe

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？？？

Crt1.o

- g F .text 00000000 _start

- Call __libc_csu_init
- Call __libc_start_main
- Call main
- Call __libc_csu_fini

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

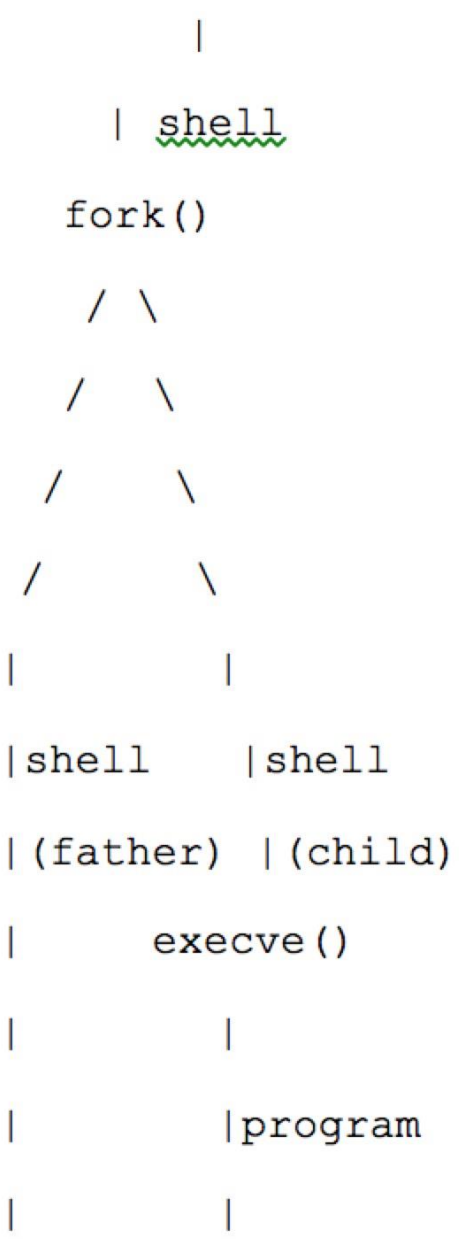
Crt1.o 的定位表：

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。

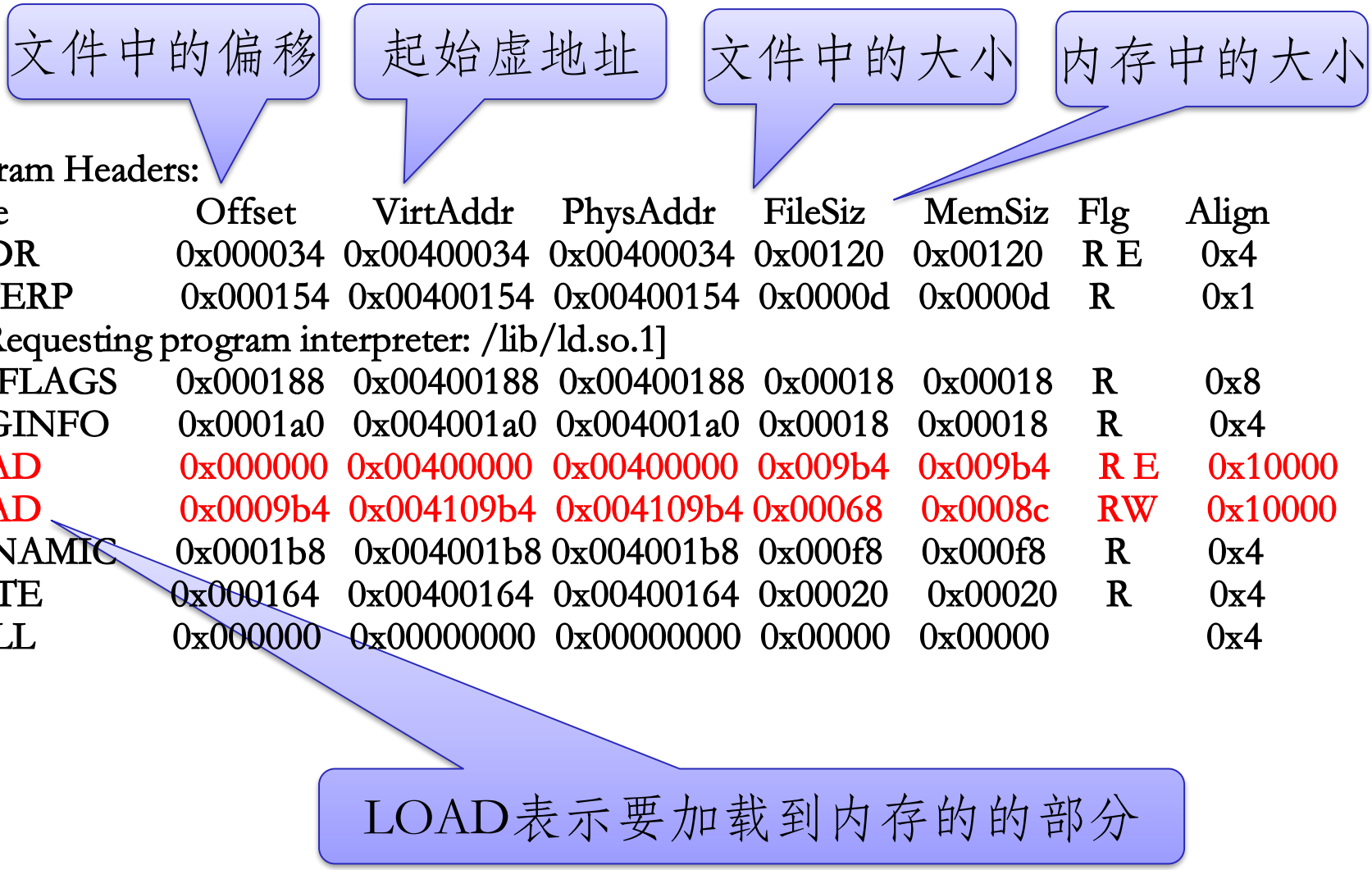
装载工作：

- 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：读出的信息分为两部分，一部分是各segment的具体信息，另一部分是section和segment之间的对应关系。
- 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:

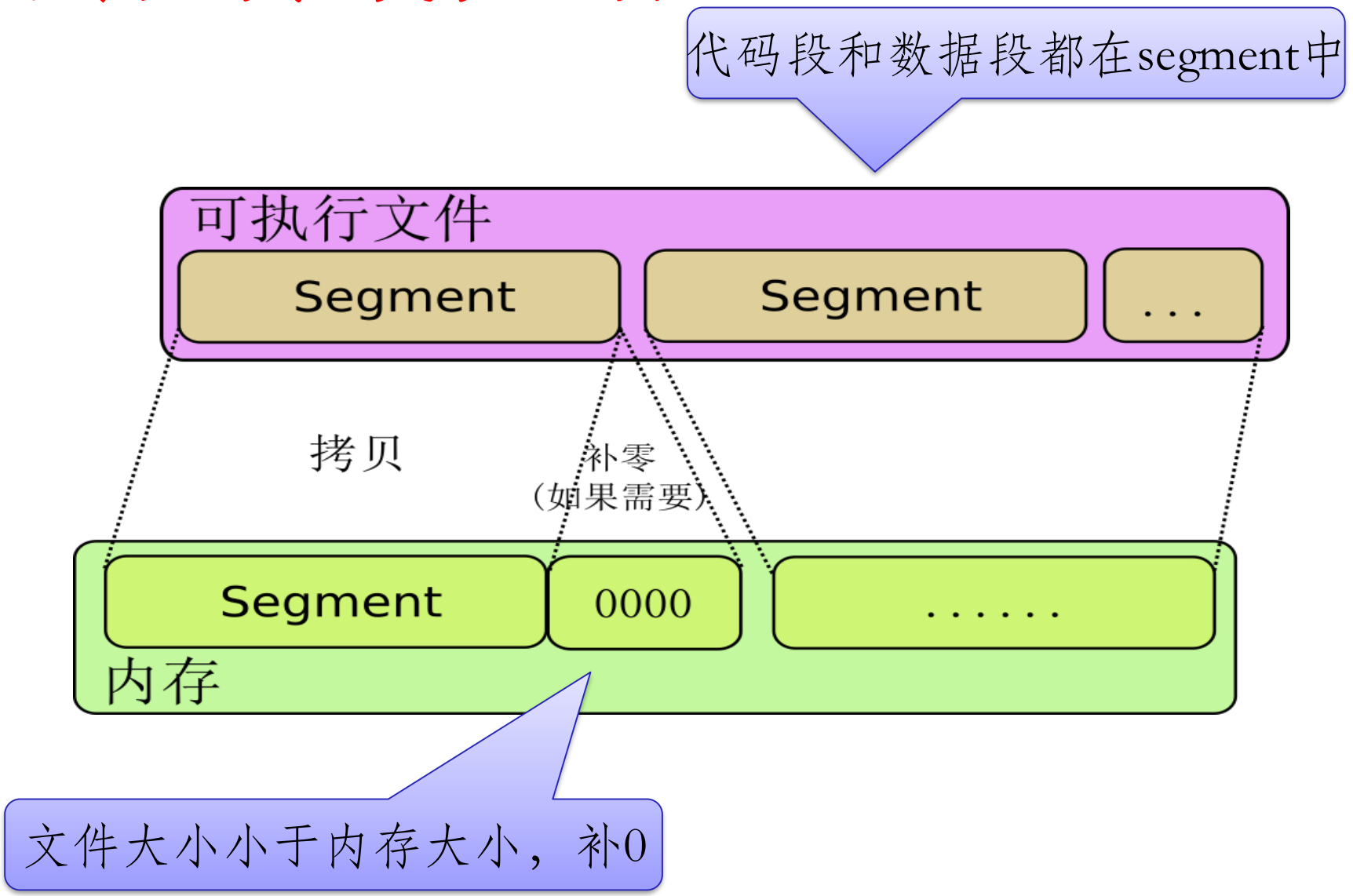
Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000034	0x00400034	0x00400034	0x00120	0x00120	R E	0x4
INTERP	0x000154	0x00400154	0x00400154	0x0000d	0x0000d	R	0x1
[Requesting program interpreter: /lib/ld.so.1]							
ABIFLAGS	0x000188	0x00400188	0x00400188	0x00018	0x00018	R	0x8
REGINFO	0x0001a0	0x004001a0	0x004001a0	0x00018	0x00018	R	0x4
LOAD	0x000000	0x00400000	0x00400000	0x009b4	0x009b4	R E	0x10000
LOAD	0x0009b4	0x004109b4	0x004109b4	0x00068	0x0008c	RW	0x10000
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。

程序的装载和运行


代码段和数据段都在segment中
可执行文件
Segment
Segment
. . .
拷贝
补零
(如果需要)
Segment
0000
. . . . . .
内存
文件大小小于内存大小，补0

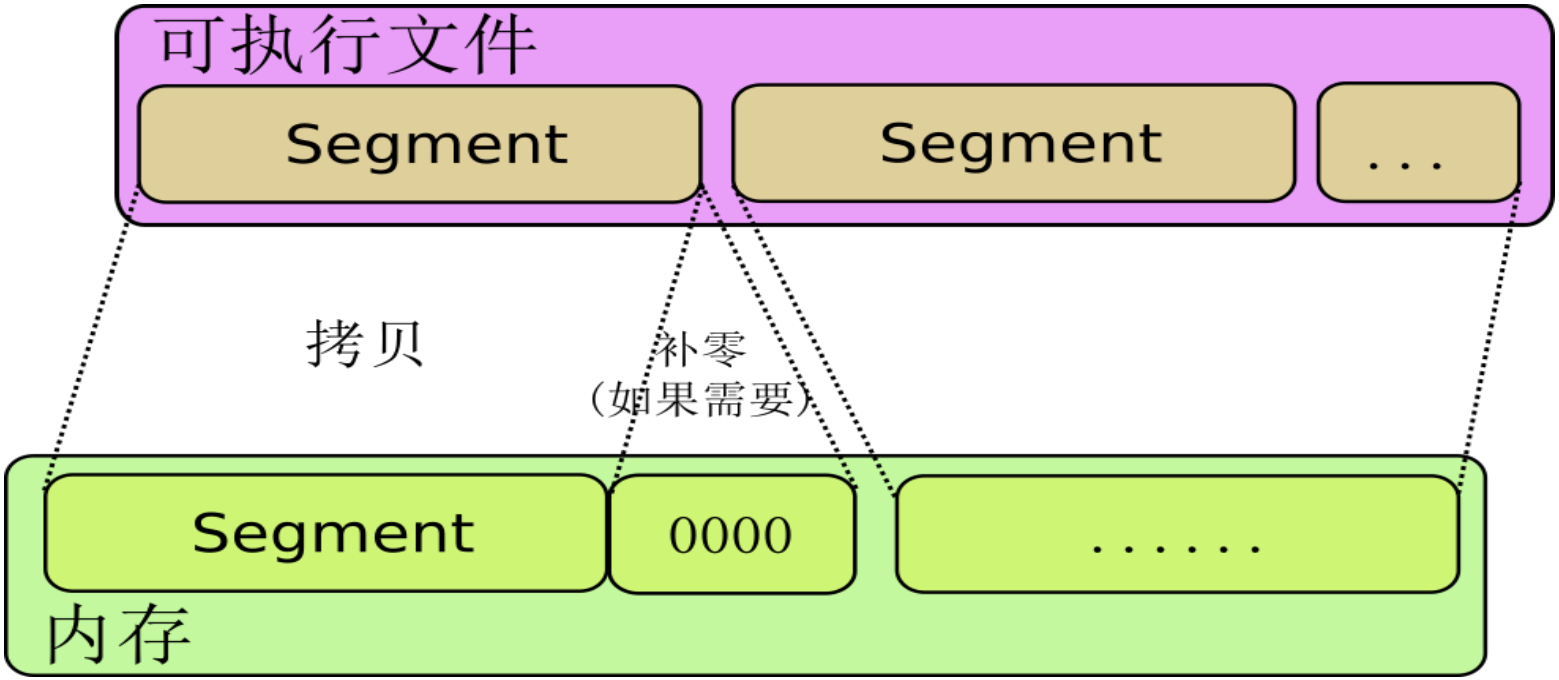
程序的装载流程

- 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！

程序的运行



各个segment都装入内存后，控制权交给应用，从应用入口开始执行

- 系统调用： execve()
- 对应函数：
  - int do_execve(char *filename, char **argv, char **envp, struct pt_regs *regs);
  - asmlinkage int sys_execve (struct pt_regs regs);
- 主要数据结构：
  - struct pt_regs在系统调用时用于保存寄存器组；
  - struct linux_binprm用于存储执行该文件的一些参数；
  - struct linux_binfmt其中定义了一些用以载入二进制文件的函数。

```
struct linux_binprm{
    char buf[128];
    unsigned long page[MAX_ARG_PAGES];
    unsigned long p;
    int sh_bang;
    struct inode * inode;
    int e_uid, e_gid;
    int argc, envc;
    char * filename;            /* Name of binary */
    unsigned long loader, exec;
    int dont_iput;              /* binfmt handler has put inode */
};
```

```
struct linux_binfmt {
    struct linux_binfmt * next;
    long *use_count;
    int (*load_binary)(struct linux_binprm *, struct  pt_regs *
    regs);
    int (*load_shlib)(int fd);
    int (*core_dump)(long signr, struct pt_regs * regs);
};
```

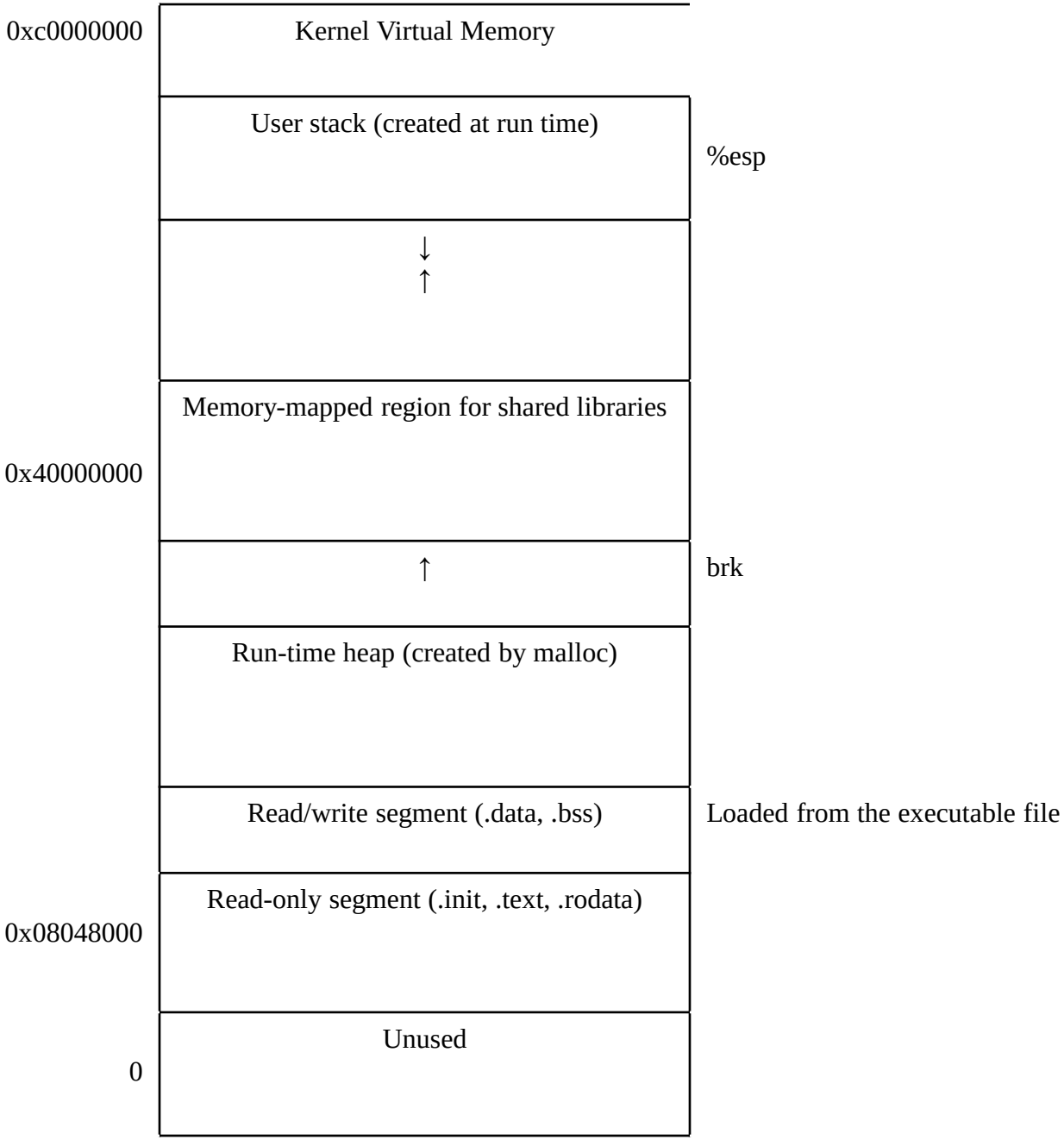
- sys_execve()只是函数do_execve()的一个界面，实际的处理动作在do_execve()中完成。
- regs.ebx：指向程序文件名的指针；
- regs.ecx：指向传递给程序的参数的指针；
- regs.edx：程序运行的环境的地址。

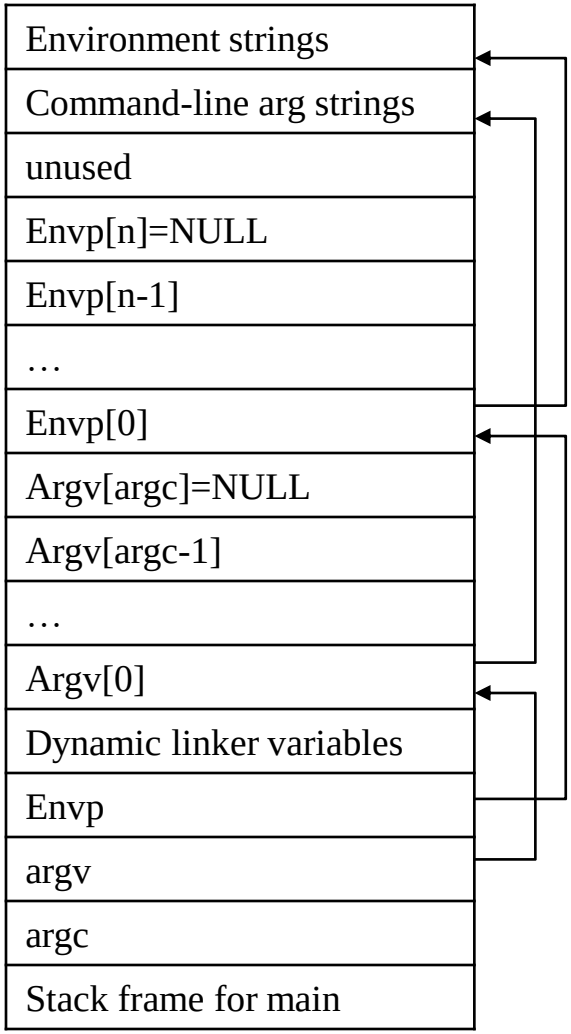
do_execve

- 通过这个程序的文件名，找到该程序对应的可执行文件即i结点；
- 检查对该文件的接触权限（在打开i结点的函数open_namei()中实现）；
- 设置程序所需的参数结构struct linux_binprm bprm；
- 以bprm为参数，调用函数prepare_binprm()，进行一些其他的检查并读入该文件的前128字节；
- 使用函数search_binary_handler()，试着用多种方式将该可执行文件载入。Linux为它所支持的每一种格式的可执行文件都提供了一个函数（由函数指针load_binary指向）以载入文件。轮流调用这些函数，并判断其返回值是否为成功标识。如果成功，则可以判断该可执行文件的格式。如果都不成功，则返回错误标识NOEXEC；

do_load_elf_binary

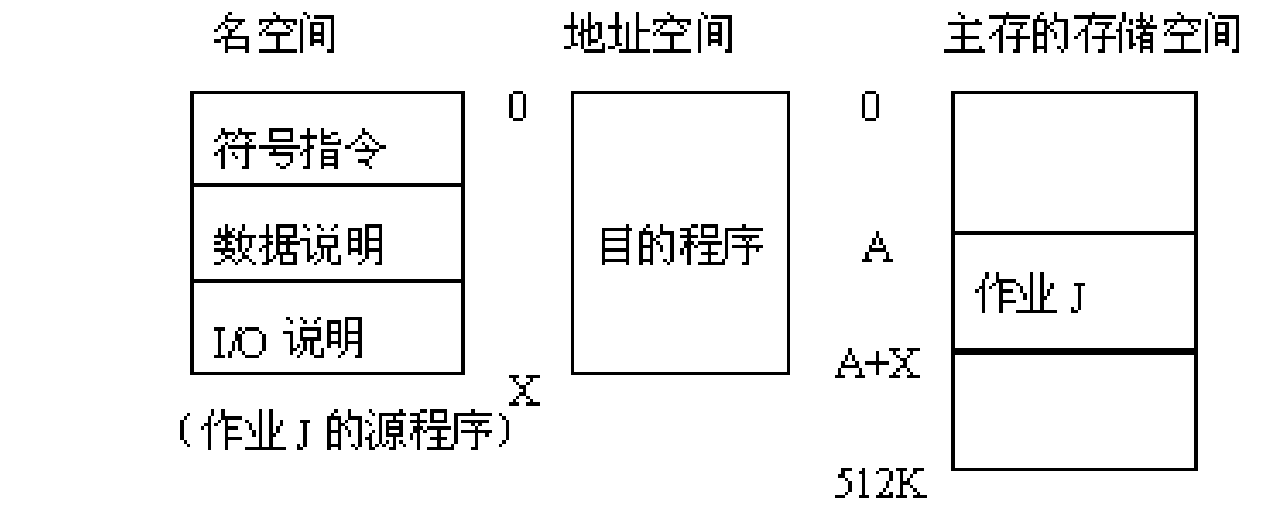
- 在参数bprm->buf中存储了文件的前128字节；
- 该函数先对欲载入文件的格式进行检查，判断是否正确。如果不正确，返回错误标识ENOEXEC，于是函数do_execve()可以继续进行检查。
- 如果判断结果是正确的，这个文件将被载入。然后，将原进程所分配的内存释放。
- 将文件的代码段和数据段使用do_mmap()函数载入内存。
- 使用set_brk()函数载入BSS段，接着将寄存器特别是指令寄存器初始化。
- 在系统调用execve结束时，使用start_thread()开始让进程从新的地址开始执行。


0xc0000000
Kernel Virtual Memory
User stack (created at run time)
%esp
↓
↑
Memory-mapped region for shared libraries
0x40000000
↑
brk
Run-time heap (created by malloc)
Read/write segment (.data, .bss)
Loaded from the executable file
Read-only segment (.init, .text, .rodata)
0x08048000
Unused
0

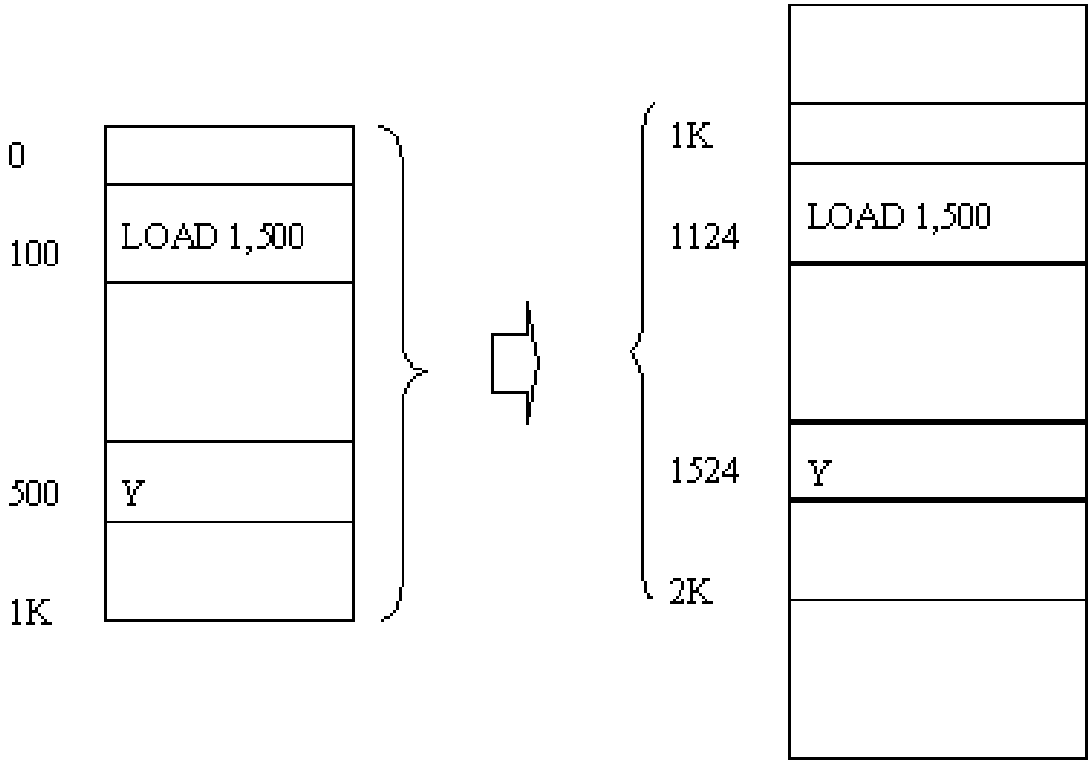

Environment strings
Command-line arg strings
unused
Envp[n]=NULL
Envp[n-1]
…
Envp[0]
Argv[argc]=NULL
Argv[argc-1]
…
Argv[0]
Dynamic linker variables
Envp
argv
argc
Stack frame for main

- 1. 地址空间：源程序经过编译后得到的目标程序，存在于它所限定的地址范围内，这个范围称为地址空间。简言之，地址空间是逻辑地址的集合。
- 2. 存储空间：存储空间是指主存中一系列存储信息的物理单元的集合，这些单元的编号称为物理地址或绝对地址。简言之，存储空间是物理地址的集合。

作业J存在于不同的空间中

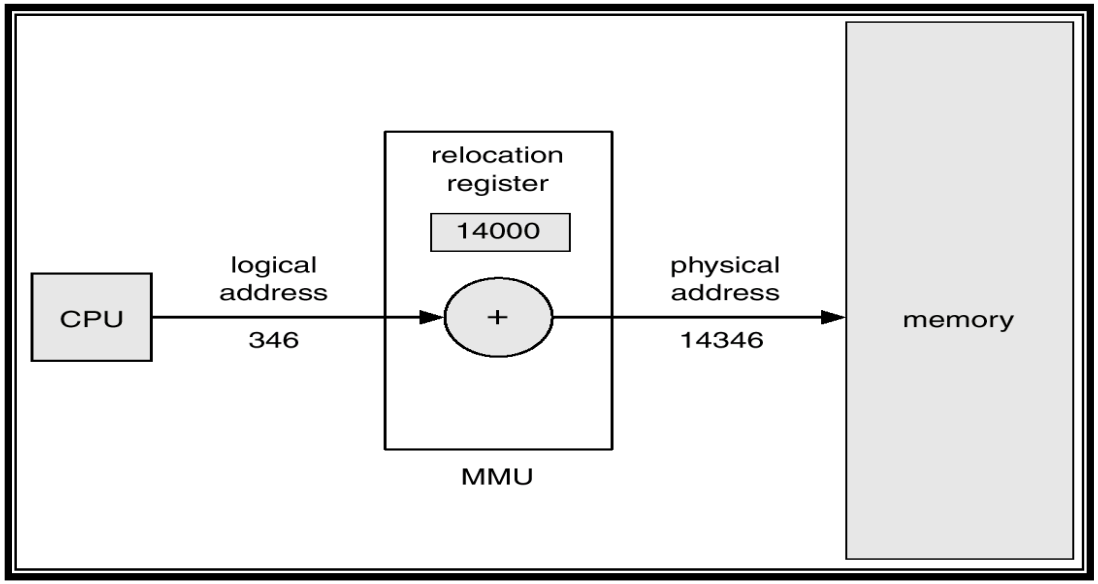


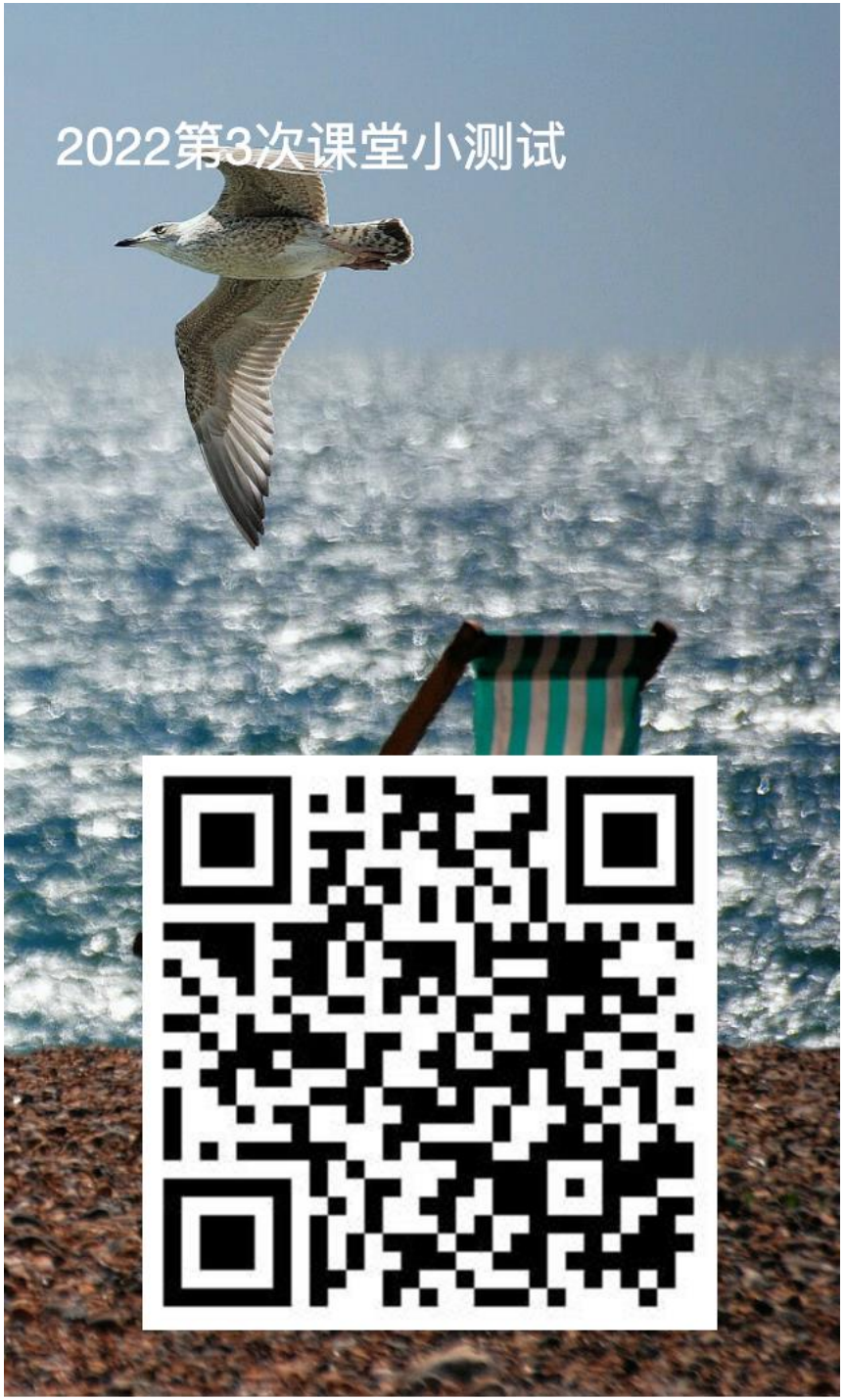
作业由地址空间装入存储空间



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




2022第3次课堂小测试

参考资料

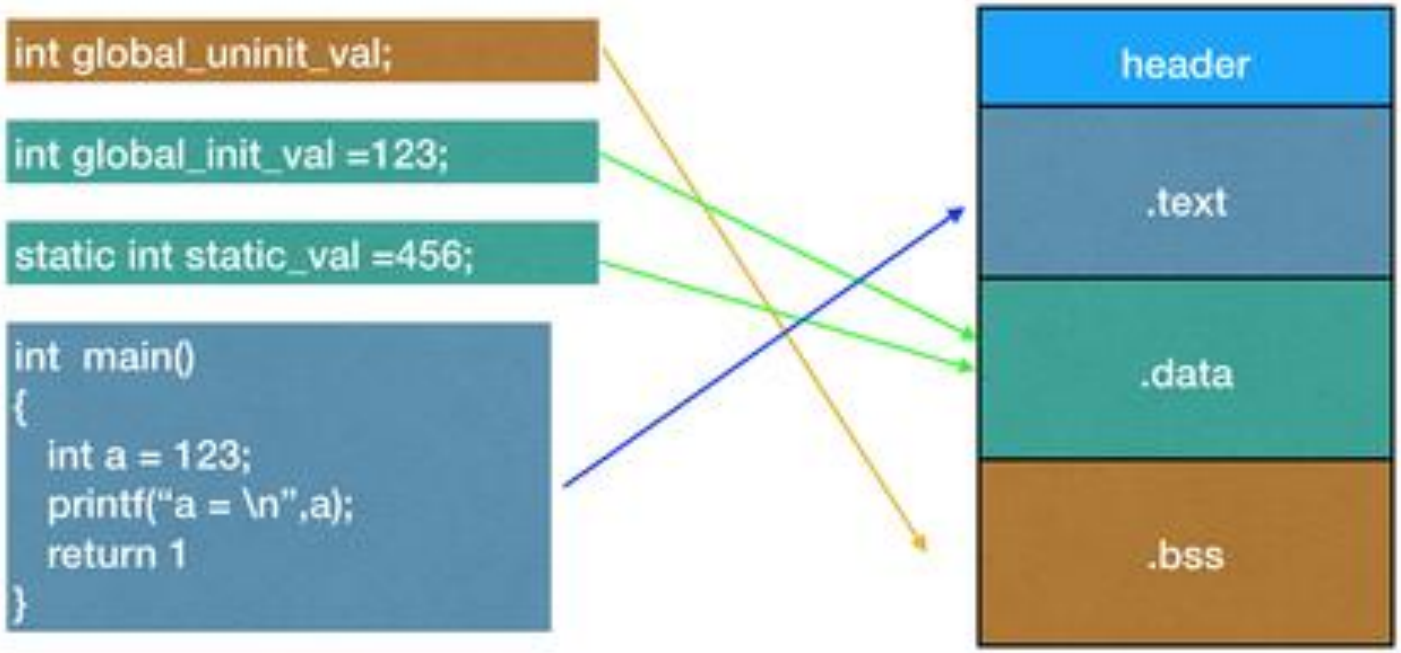
- Program Structure in GNU Linux. https://www.slideshare.net/varunmahajan06/gnu-linux-programstructure
- GCC Internals: https://gcc.gnu.org/onlinedocs/gccint/
- MIPS Relocation Types. https://dmz-portal.mips.com/wiki/MIPS_relocation_types
- 李战怀.大数据环境下存储技术发展对数据管理研究的影响.

基础知识

- 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

- 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。

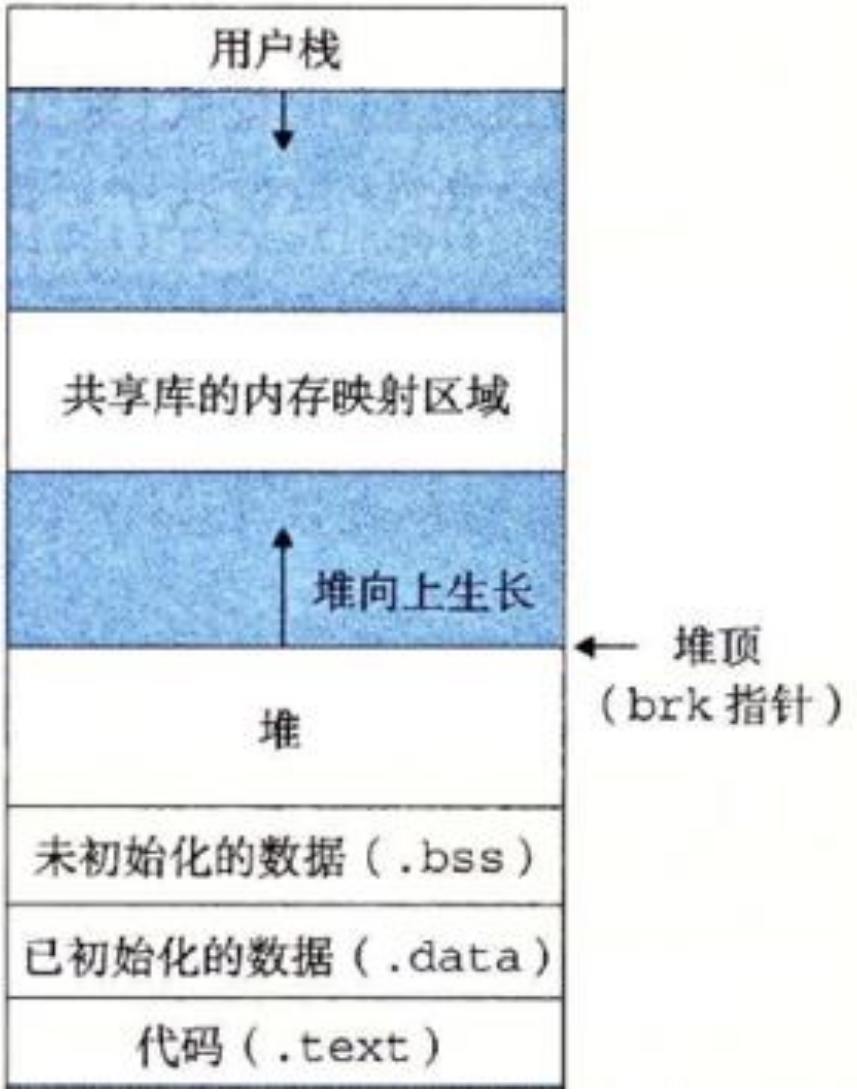


基础知识

- 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

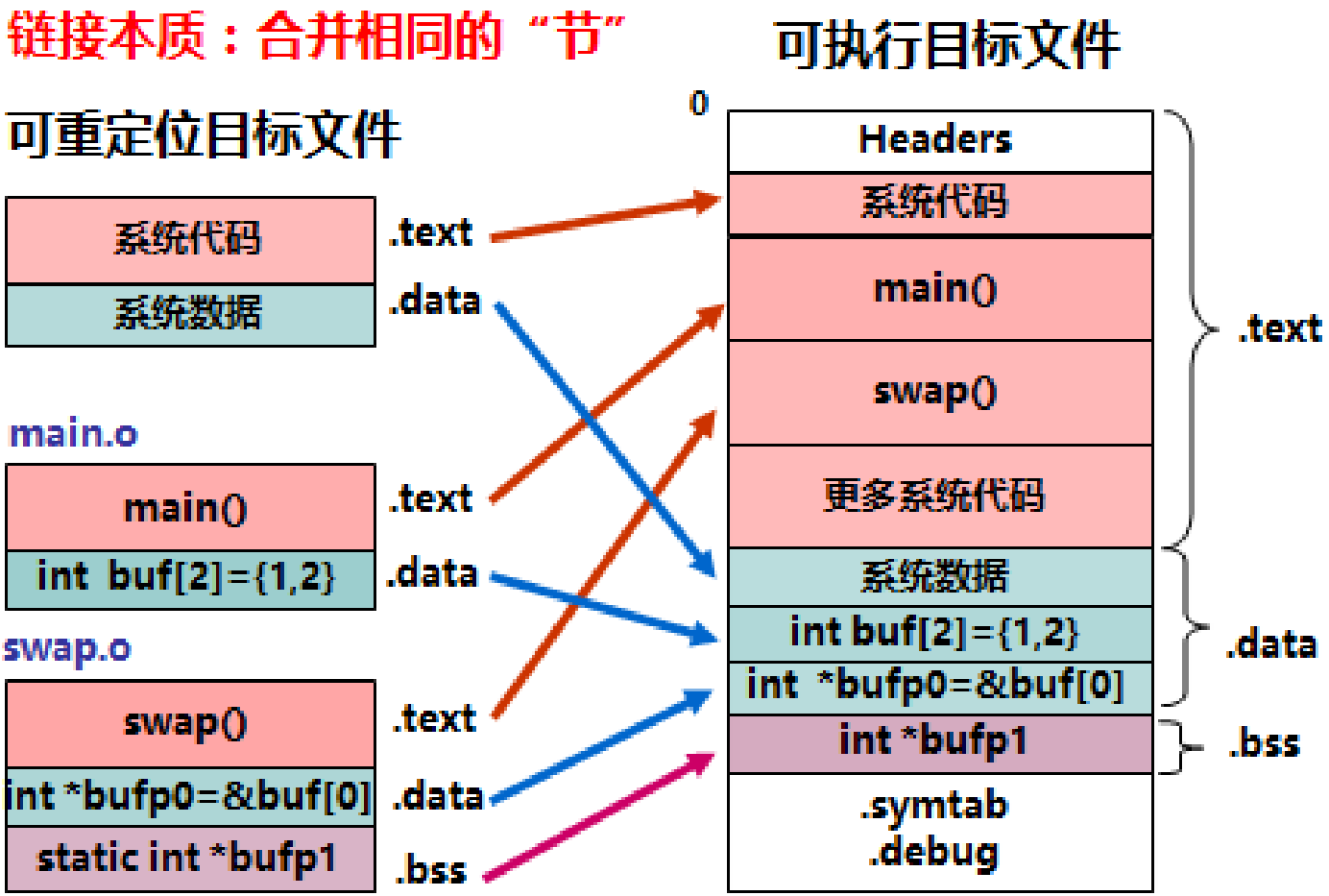
- text和data段都在可执行文件中，由系统从可执行文件中加载，而bss段不在可执行文件中，由系统初始化。
- 一个装入内存的可执行程序，除了bss、data和text段外，还需构建一个栈（stack）和一个堆（heap）。



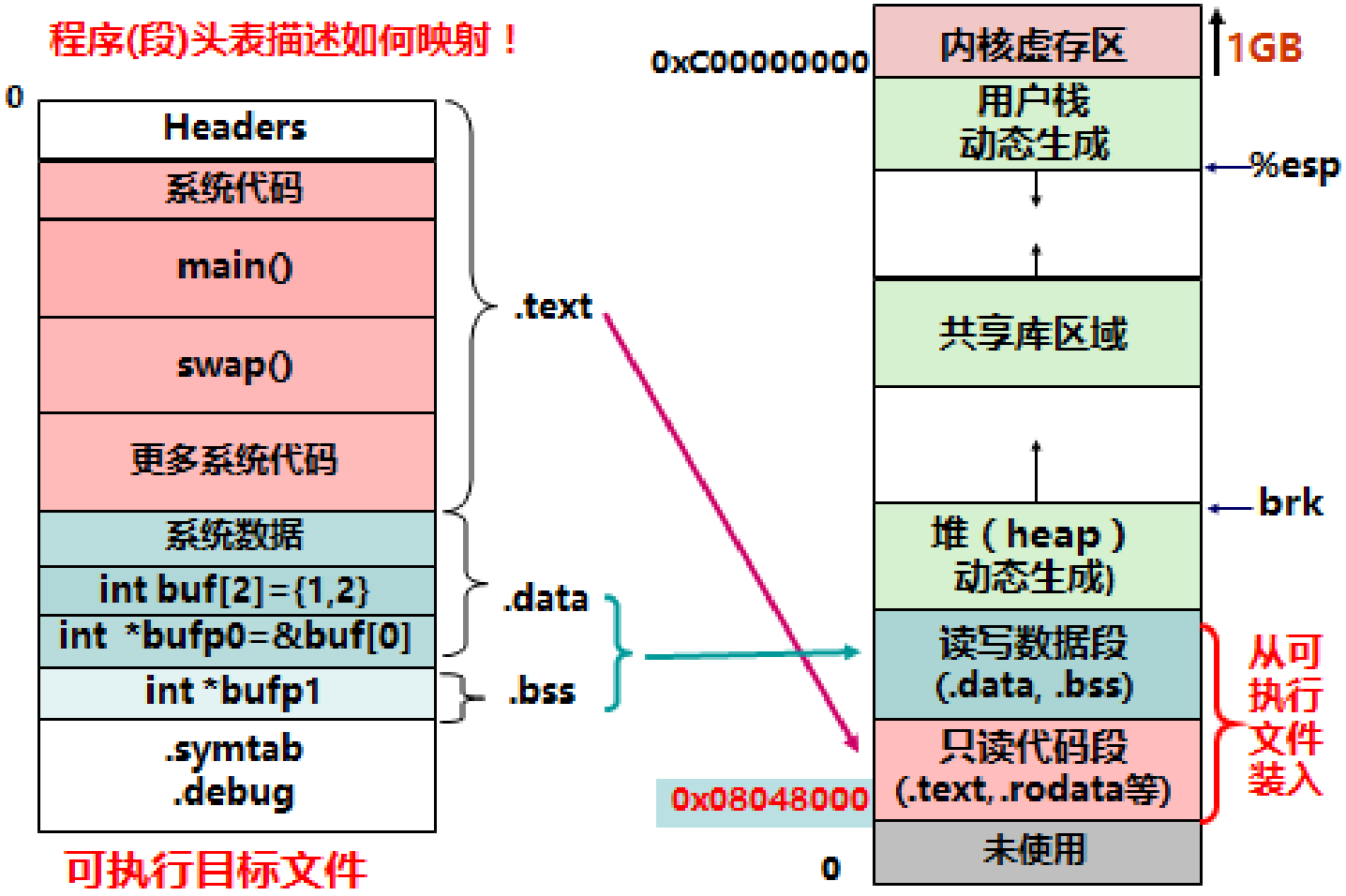
基础知识

- 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）。

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
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
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

可执行文件的内存映像(x86)



需求与存储管理的目标

需求：

- 从每个计算机使用者（程序员）的角度：
  1. 整个空间都归我使用；
  2. 不希望任何第三方因素妨碍我的程序的正常运行；
- 从计算机平台提供者的角度：
  - 尽可能同时为多个用户提供服务；

分析：

- 计算机至少同时存在两个程序：一个用户程序和一个服务程序（操作系统）
- 每个程序具有的地址空间应该是相互独立的；
- 每个程序使用的空间应该得到保护；

需求与存储管理的目标

存储管理的基石：

1. 地址独立：程序发出的地址与物理地址无关
2. 地址保护：一个程序不能访问另一个程序的地址空间

存储管理要解决的问题：分配和回收

内容提要

- 存储管理基础
  - 单道程序的内存管理
  - 多道程序的内存管理
- 页式内存管理
- 段式内存管理
- 虚拟存储管理
- 存储管理实例

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。

单道程序的内存管理

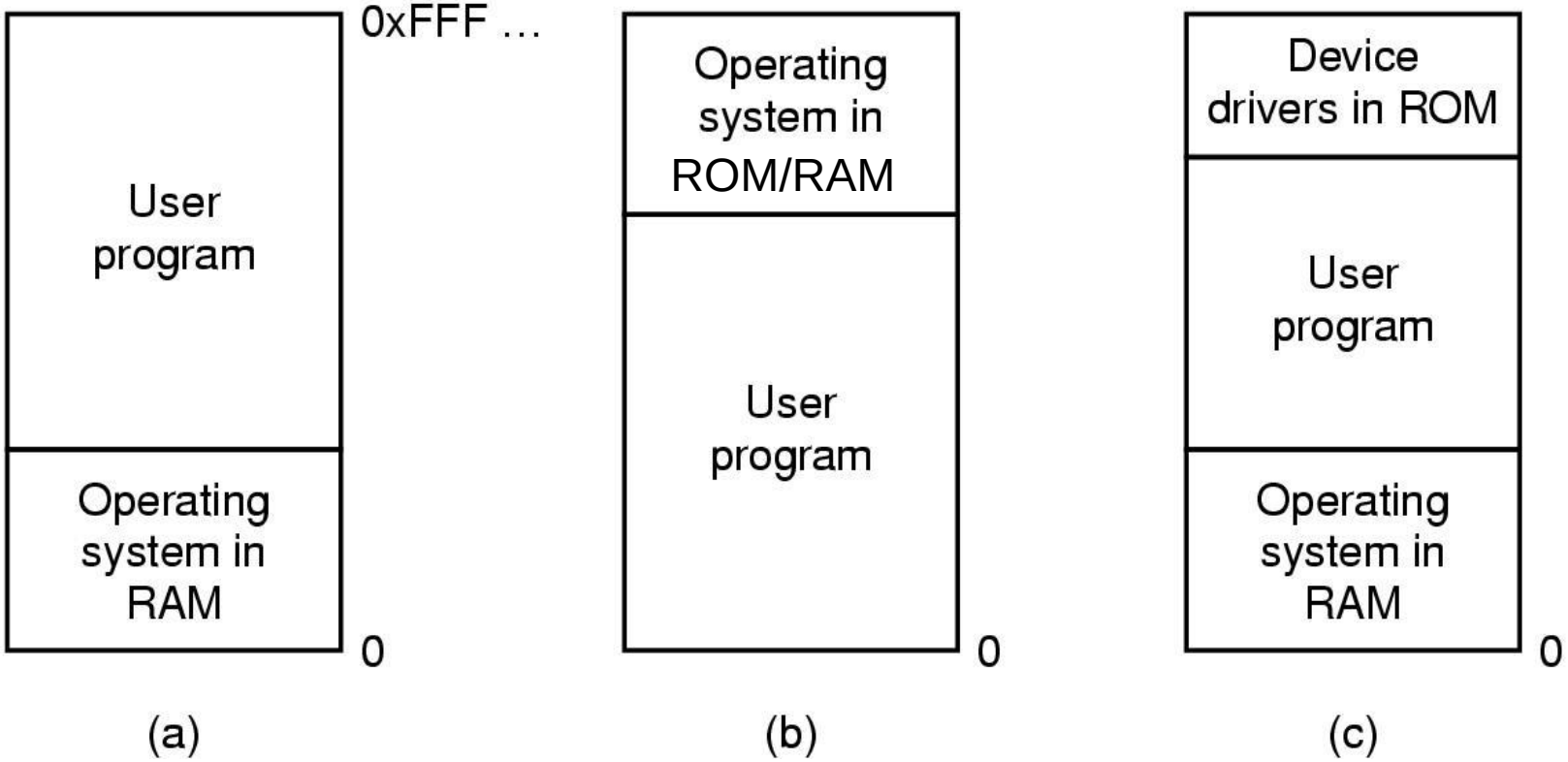
条件：

- 在单道程序环境下，整个内存里只有两个程序：一个用户程序和操作系统。
- 操作系统所占的空间是固定的。
- 因此可以将用户程序永远加载到同一个地址，即用户程序永远从同一个地方开始运行。

结论：

- 用户程序的地址在运行之前可以计算。

操作系统在内存中的位置



单道程序的内存管理

方法：

- 静态地址翻译：即在程序运行之前就计算出所有物理地址。
- 静态翻译工作可以由加载器实现。

分析：

- **地址独立？** YES. 因为用户无需知道物理内存的相关知识。
- **地址保护？** YES. 因为没有其它用户程序。

单道程序的内存管理

优点：

- 执行过程中无需任何地址翻译工作，程序运行速度快。

缺点：

- 比物理内存大的程序无法加载，因而无法运行。
- 造成资源浪费（小程序会造成空间浪费；I/O时间长会造成计算资源浪费）。

思考：

- 程序可加载到内存中，就一定可以正常运行吗?
- 用户程序运行会影响操作系统吗?

多道程序的存储管理

空间的分配：分区式分配

- 把内存分为一些大小相等或不等的分区(partition)，每个应用程序占用一个或几个分区。操作系统占用其中一个分区。
- 适用于多道程序系统和分时系统，支持多个程序并发执行，但难以进行内存分区的共享。

方法：

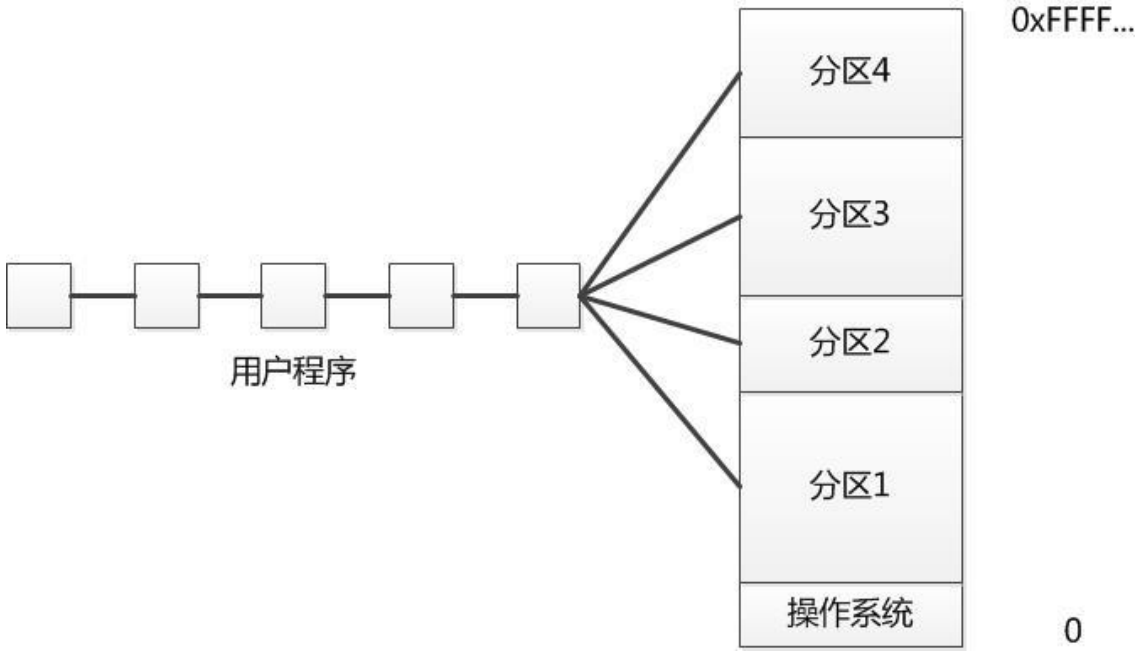
- 固定（静态）式分区分配，程序适应分区。
- 可变（动态）式分区分配，分区适应程序。

固定式分区

- 把内存划分为若干个固定大小的连续分区。
  - 分区大小相等：只适合于多个相同程序的并发执行（处理多个类型相同的对象）。
  - 分区大小不等：多个小分区、适量的中等分区、少量的大分区。根据程序的大小，分配当前空闲的、适当大小的分区。

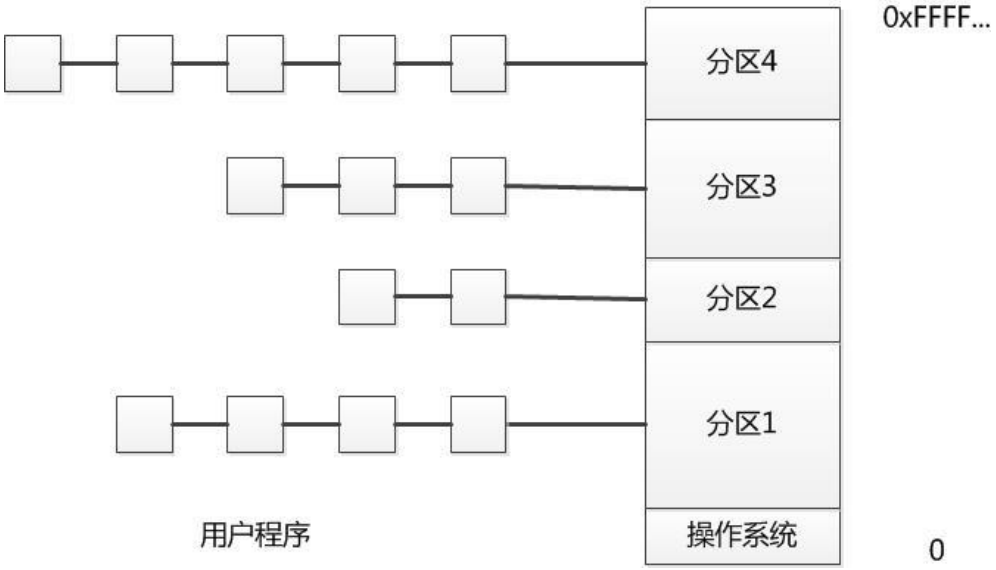
单一队列的分配方式

- 当需要加载程序时，选择一个当前闲置且容量足够大的分区进行加载，可采用共享队列的固定分区（多个用户程序排在一个共同的队列里面等待分区）分配。



多队列分配方式

- 由于程序大小和分区大小不一定匹配，有可能形成一个小程序占用一个大分区的情况，从而造成内存里虽然有小分区闲置但无法加载大程序的情况。这时，可以采用多个队列，给每个分区一个队列，程序按照大小排在相应的队列里。

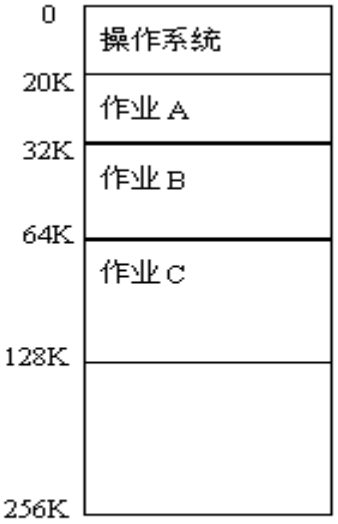


固定式分区的管理

- 优点：易于实现，开销小。
- 缺点：内碎片造成浪费，分区总数固定，限制了并发执行的程序数目。
- 采用的数据结构：分区表——记录分区的大小和使用情况

分区号	大小	起址	状态
1	12K	20K	已分配
2	32K	32K	已分配
3	64K	64K	已分配
4	128K	128K	未分配

分区说明表

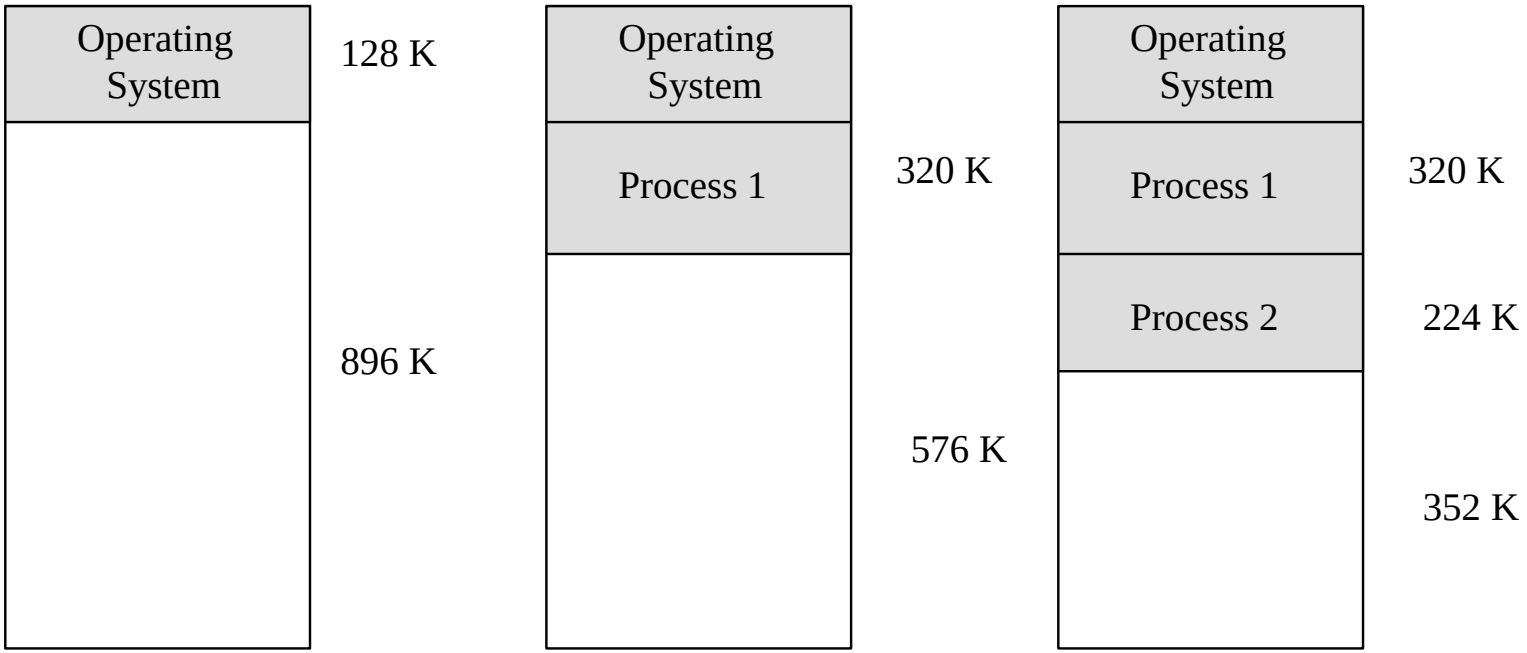


存储空间分配情况

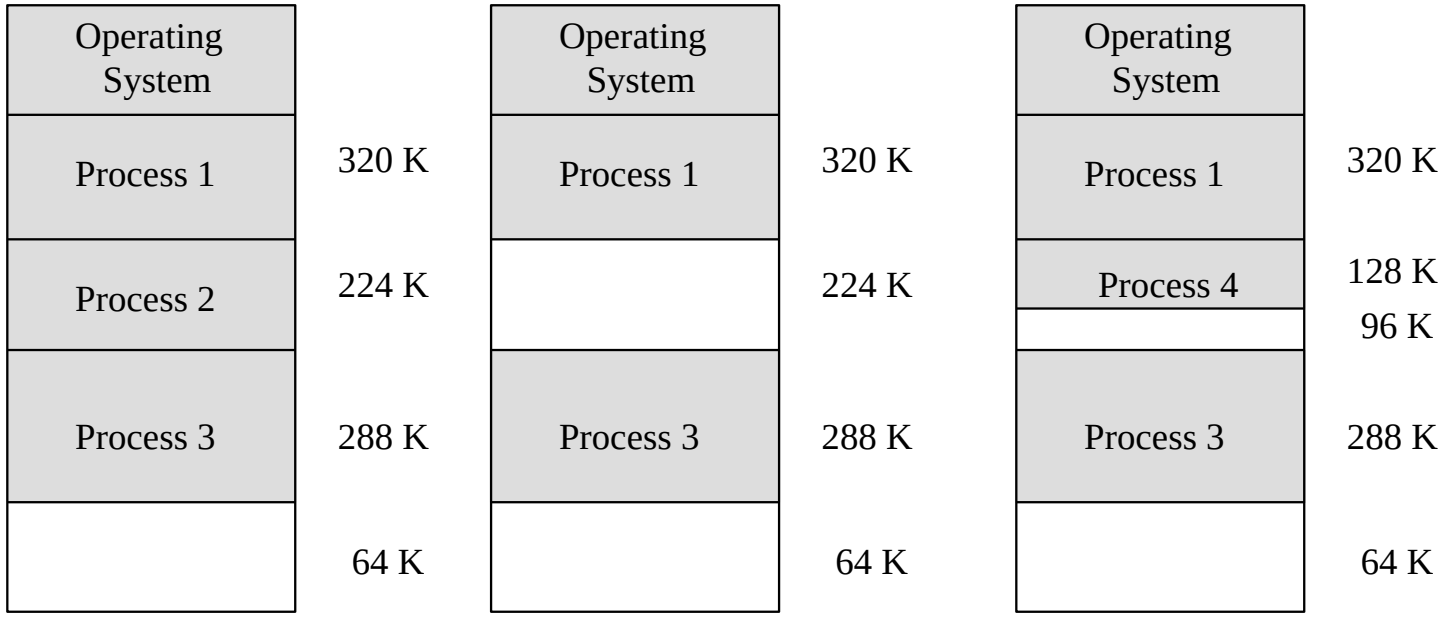
可变式分区：

- 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。

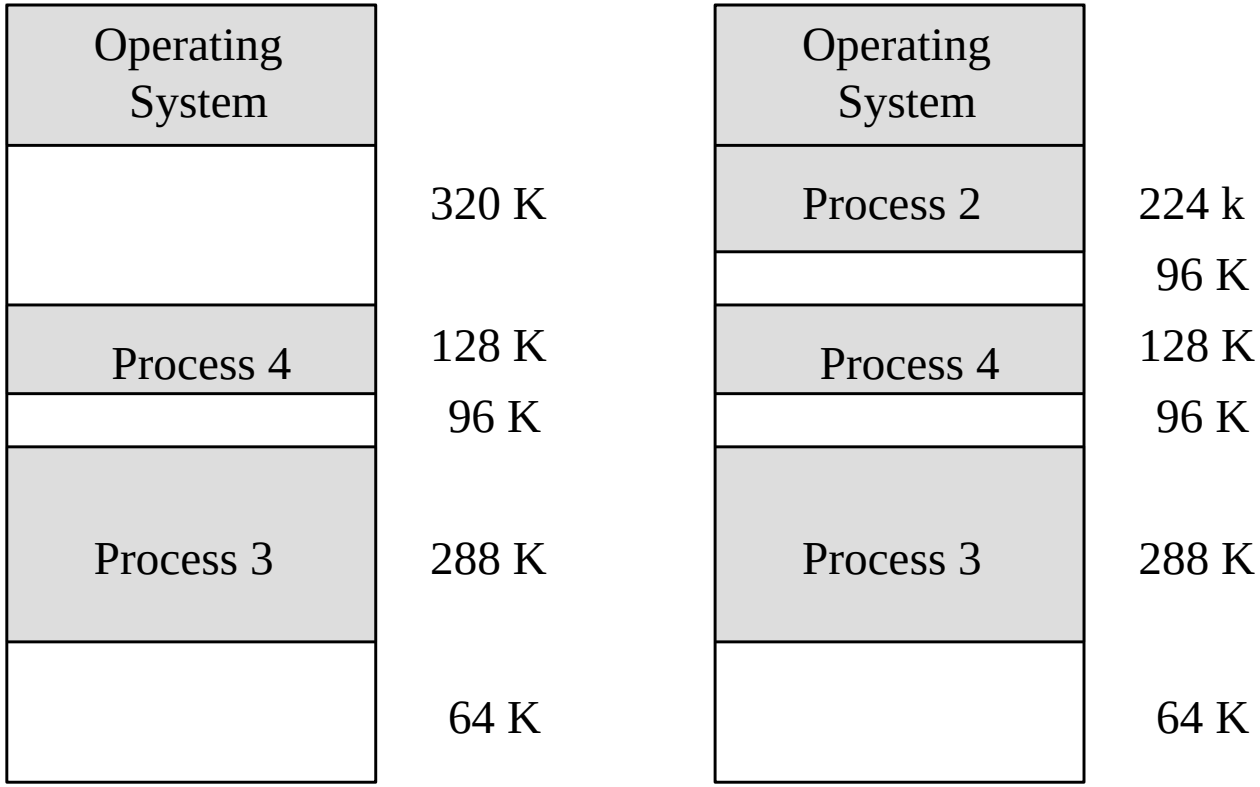
可变式分区：


Operating System
128 K
896 K
Operating System
Process 1
320 K
576 K
Operating System
Process 1
320 K
Process 2
224 K
352 K

可变式分区：

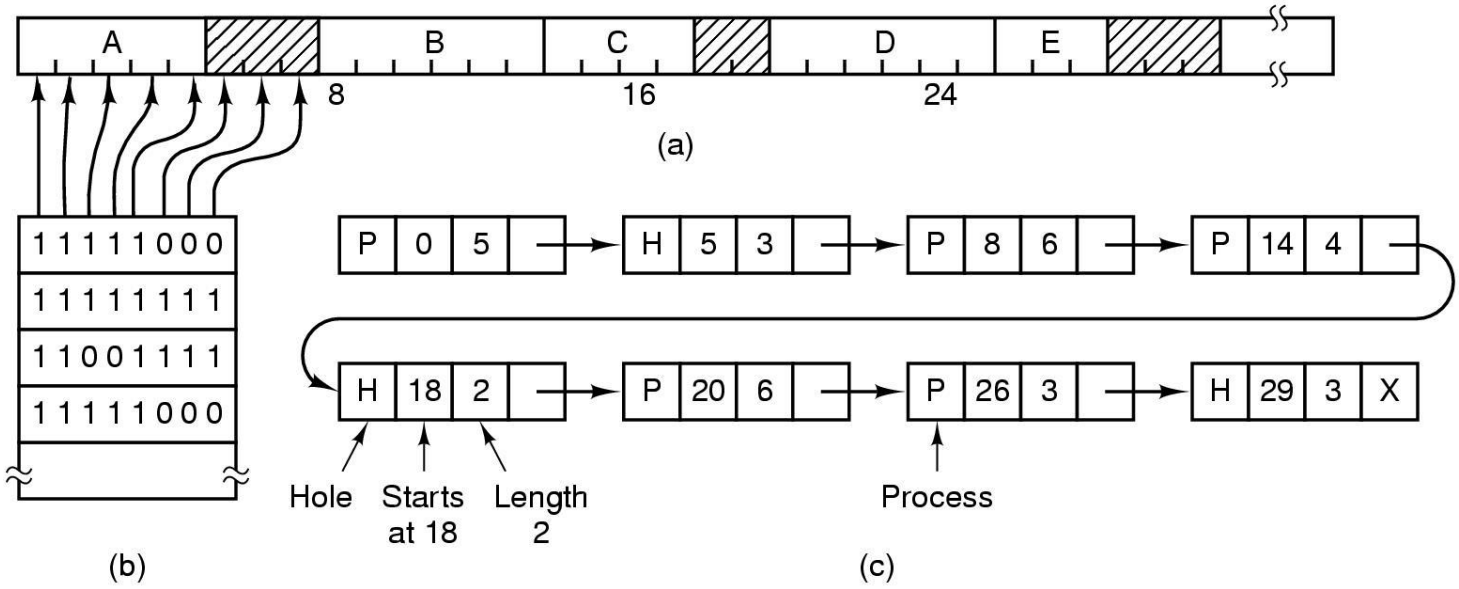

Operating System
Process 1
Process 2
Process 3
320 K
224 K
288 K
64 K
Operating System
Process 1
Process 3
320 K
224 K
288 K
64 K
Operating System
Process 1
Process 4
Process 3
320 K
128 K
96 K
288 K
64 K

可变式分区：


Operating System
320 K
Process 4
128 K
96 K
Process 3
288 K
64 K
Operating System
Process 2
224 k
96 K
Process 4
128 K
96 K
Process 3
288 K
64 K

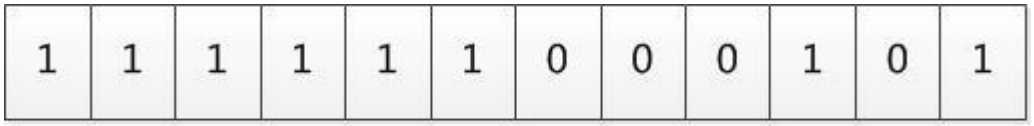
闲置空间的管理

- 在管理内存的时候，OS需要知道内存空间有多少空闲？这就必须跟踪内存的使用，跟踪的办法有两种：位图表示法（分区表）和链表表示法（分区链表）



位图表示法

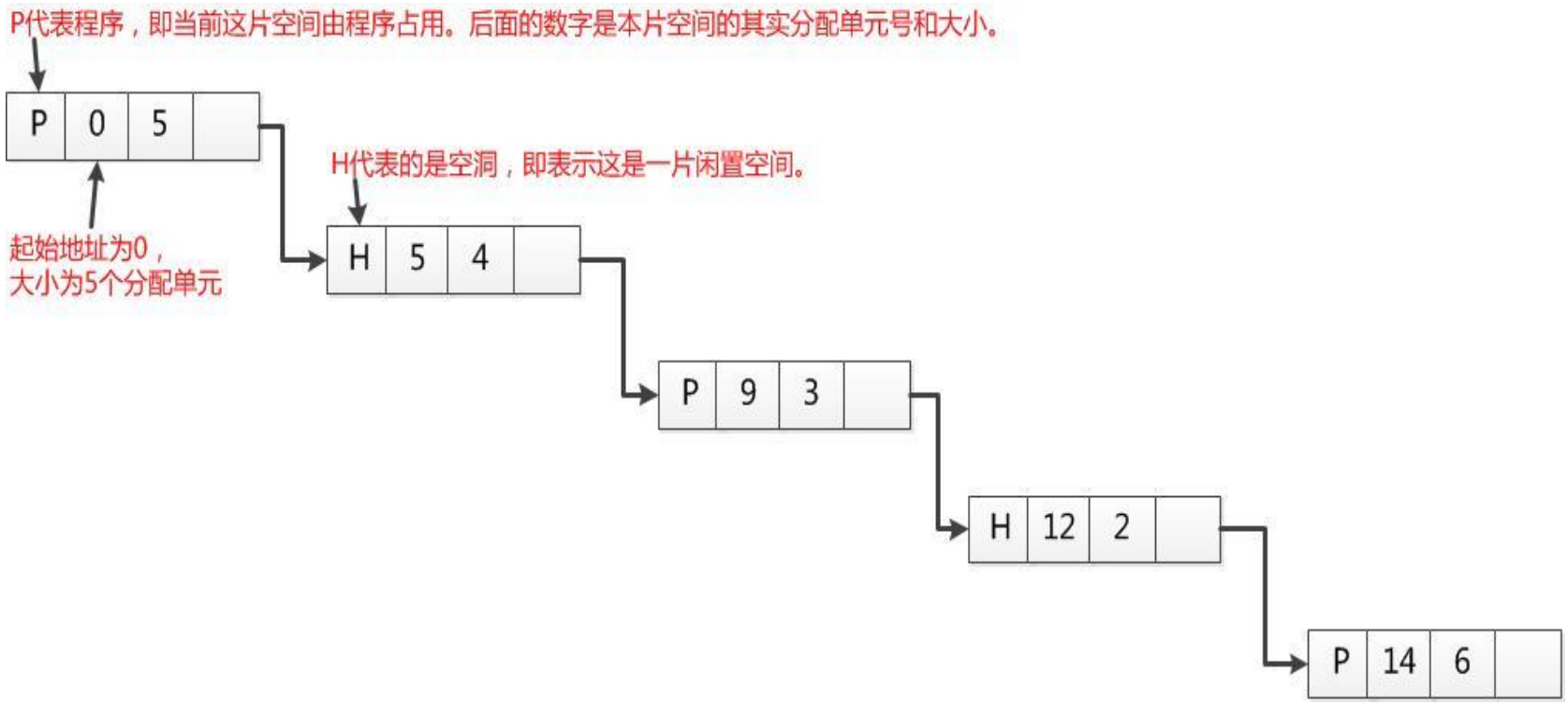
- 给每个分配单元赋予一个字位，用来记录该分配单元是否闲置。例如，字位取值为0表示单元闲置，取值为1则表示已被占用，这种表示方法就是位图表示法。



内存分配位图表示

链表表示法

- 将分配单元按照是否闲置链接起来，这种方法称为链表表示法。如上图所示的的位图所表示的内存分配状态，使用链表来表示的话则会如下图所示



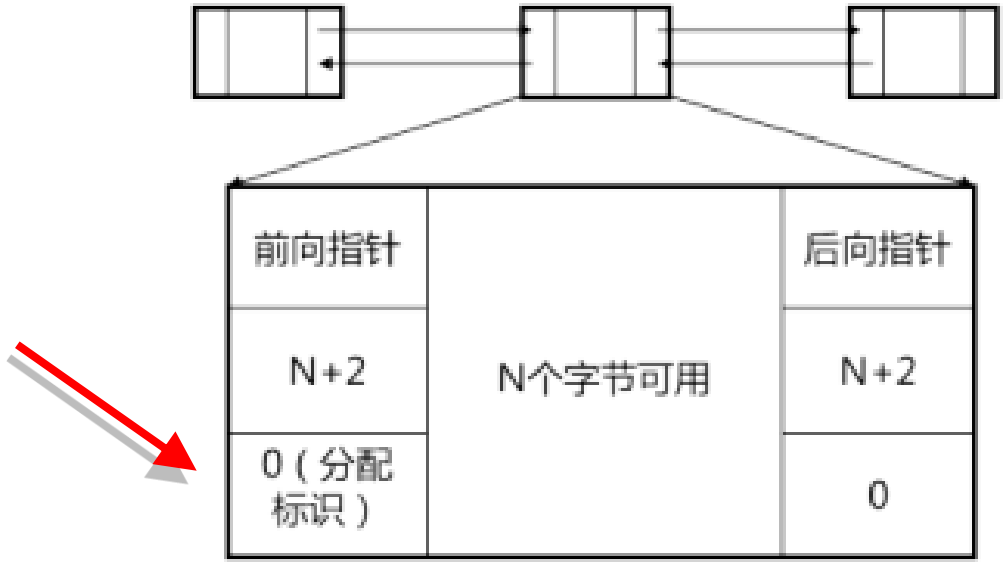
两种方法的特点

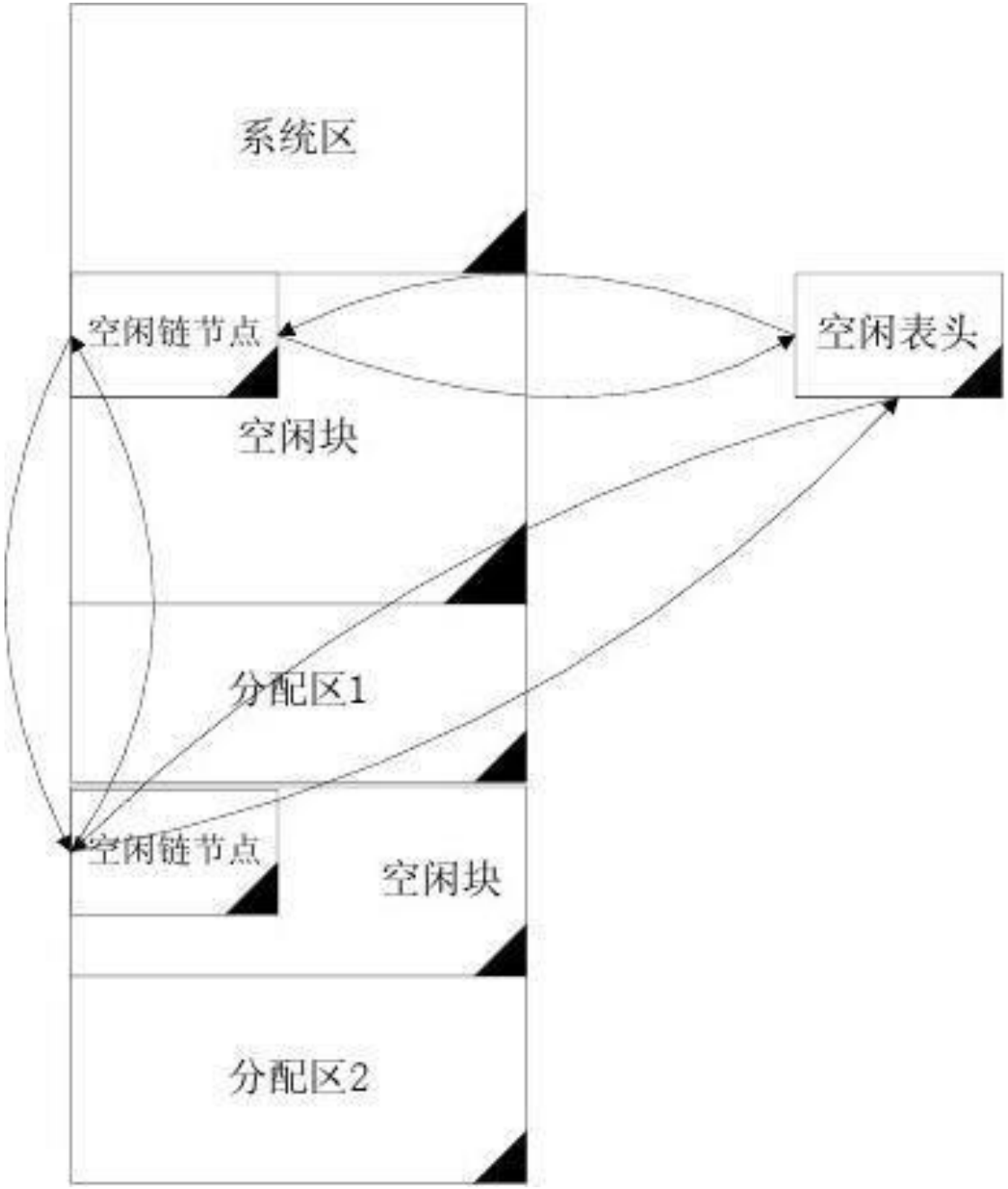
- 位图表示法：
  - 空间成本固定：不依赖于内存中的程序数量。
  - 时间成本低：操作简单，直接修改其位图值即可。
  - 没有容错能力：如果一个分配单元为1，不能肯定应该为1还是因错误变成1。
- 链表表示法：
  - 空间成本：取决于程序的数量。
  - 时间成本：链表扫描通常速度较慢，还要进行链表项的插入、删除和修改。
  - 有一定容错能力：因为链表有被占空间和闲置空间的表项，可以相互验证。

可变分区的管理

- 内存分配采用两张表：已分配分区表和未分配分区表。
- 每张表的表项为存储控制块MCB（Memory Control Block），包括AMCB（Allocated MCB）和FMCB（Free MCB）
- 空闲分区控制块按某种次序构成FMCB链表结构。当分区被分配出去以后，前、后向指针无意义。

分配标识：
0：未分配
1：已分配




系统区
空闲链节点
空闲表头
空闲块
分配区1
空闲链节点
空闲块
分配区2

分区分配操作（分配内存）

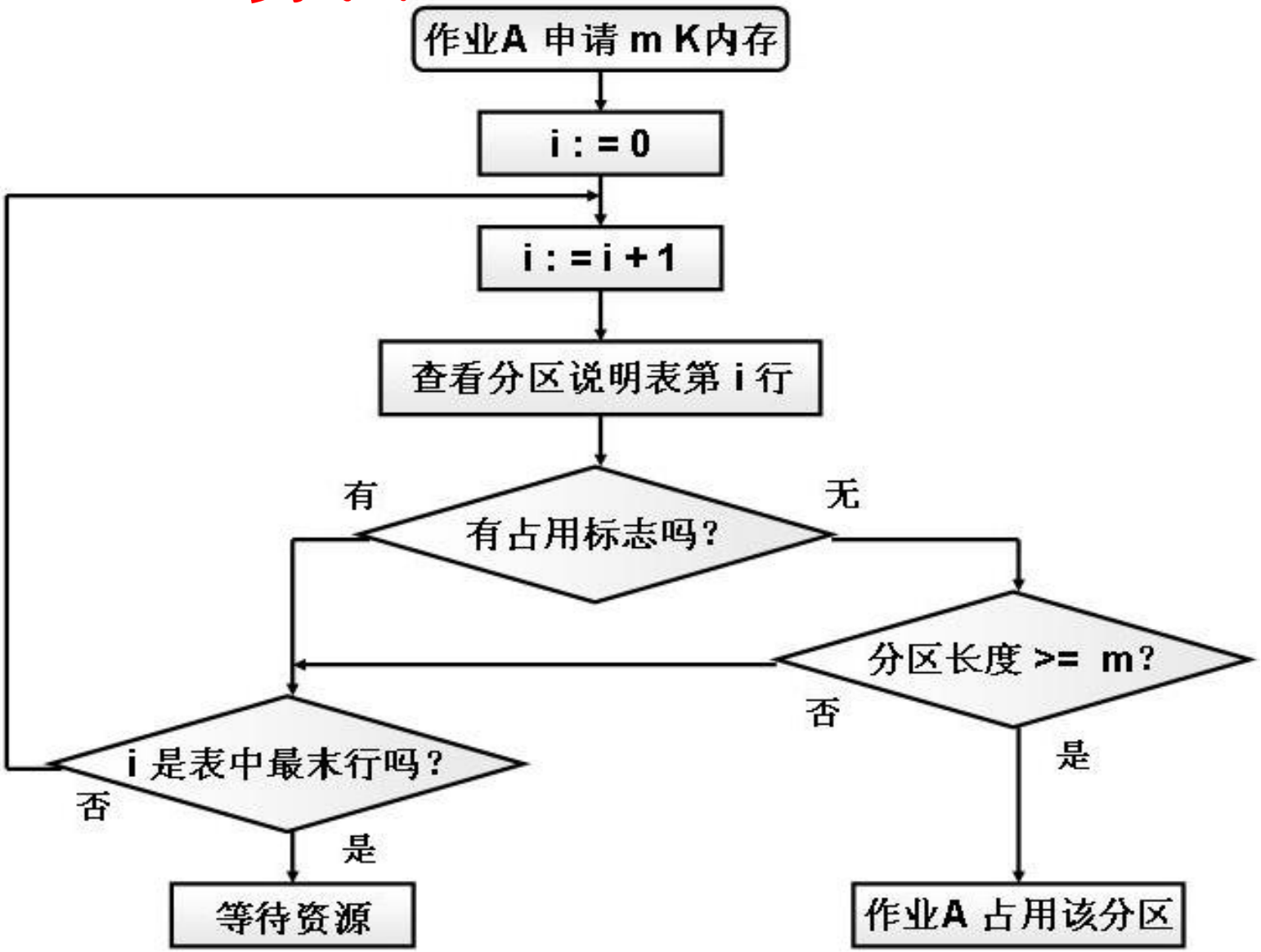
分配内存

- 事先规定 size 是不再切割的剩余分区的大小。
- 设请求的分区大小为 u.size，空闲分区的大小为 m.size。
- 若 m.size-u.size≤size，将整个分区分配给请求者。
- 否则，从该分区中按请求的大小划分出一块内存空间分配出去，余下的部分仍留在空闲分区表/链中。

基于顺序搜索的分配算法：

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）：为一个作业选择分区时，总是寻找其大小最接近于作业所要求的存储区域。
4. 最坏适应算法（Worst Fit）：为作业选择存储区域时，总是寻找最大的空白区。

FirstFit算法


作业A 申请 m K内存
i : = 0
i : = i + 1
查看分区说明表第 i 行
有
有占用标志吗?
无
分区长度 >= m?
否
是
i 是表中最末行吗?
否
是
等待资源
作业A 占用该分区

FirstFit算法

- 优点：
  - 分配和释放的时间性能较好
  - 较大的空闲分区保留在内存的高端
- 缺点：随着低端内存被不断分配，会产生很多小分区，开销会增大。

算法举例

- 例：系统中的空闲分区表如下表示，现有三个作业分配申请内存空间100K、30K及7K，给出按首次适应算法、下次适应算法、最佳适应算法和最坏适应算法的内存分配情况及分配后空闲分区表。

区号	大小	起始地址	状态
1	32K	20K	未分配
2	8K	52K	未分配
3	120K	60K	未分配
4	331K	180K	未分配

首次适应算法

- 申请作业 100k，分配 3 号分区，剩下分区为 20k，起始地址 160K 。
- 100K、30K 、7K

区号	大小	起始地址	状态
1	32K	20K	未分配
2	8K	52K	未分配
3	20K	160K	未分配
4	331K	180K	未分配

首次适应算法

- 申请作业 30K，分配 1 号分区，剩下分区为 2K，起始地址 50K 。
- 100K、30K 、7K

区号	大小	起始地址	状态
1	2K	50K	未分配
2	8K	52K	未分配
3	20K	160K	未分配
4	331K	180K	未分配

首次适应算法

- 申请作业 7K，分配 2 号分区，剩下分区为 1K，起始地址59K。
- 100K、30K 、7K

区号	大小	起始地址	状态
1	2K	50K	未分配
2	1K	59K	未分配
3	20K	160K	未分配
4	331K	180K	未分配

下次适应算法

- 按下次适应算法，申请作业100k，分配3号分区，剩下分区为20k，起始地址160K；
- 100K、30K、7K

区号	大小	起始地址	状态
1	32K	20K	未分配
2	8K	52K	未分配
3	20K	160K	未分配
4	331K	180K	未分配

下次适应算法

- 按下次适应算法，申请作业 30K，分配 4 号分区，剩下分区为301K;
- 100K、30K 、7K

区号	大小	起始地址	状态
1	32K	20K	未分配
2	8K	52K	未分配
3	20K	160K	未分配
4	301K	210K	未分配

下次适应算法

- 按下次适应算法，申请作业 7k，分配 4 号分区
- 100K、30K 、7K

区号	大小	起始地址	状态
1	32K	20K	未分配
2	8K	52K	未分配
3	20K	160K	未分配
4	294K	217K	未分配

算法特点

- 首次适应：优先利用内存低地址部分的空闲分区。但由于低地址部分不断被划分，留下许多难以利用的很小的空闲分区（碎片或零头），而每次查找又都是从低地址部分开始，增加了查找可用空闲分区的开销。
- 下次适应：使存储空间的利用更加均衡，不致使小的空闲区集中在存储区的一端，但这会导致缺乏大的空闲分区。

最佳适应算法

按容量大小递增的次序排列

分配前的空闲分区表

区号	大小	起址
1	8k	52k
2	32k	20k
3	120k	60k
4	331k	180k

作业30K分配后

区号	大小	起址
2	2k	50k
1	8k	52k
3	20k	160k
4	331k	180k

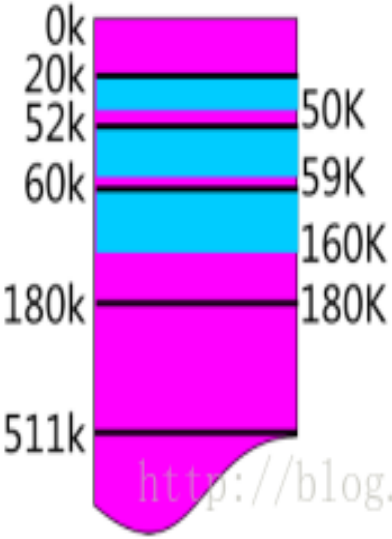
按容量递增的次序重新排列

作业100K分配后

区号	大小	起址
1	8k	52k
3	20k	160k
2	32k	20k
4	331k	180k

作业7K分配后

区号	大小	起址
1	1k	59k
2	2k	50k
3	20k	160k
4	331k	180k



区号	大小	起址
1	1k	59k
2	2k	50k
3	20k	160k
4	331k	180k

最坏适应算法

按容量大小递减的次序排列

分配前的空闲分区表

区号	大小	起址
1	331k	180k
2	120k	60k
3	32k	20k
4	8k	52k

作业30K分配后

区号	大小	起址
1	201k	310k
2	120k	60k
3	32k	20k
4	8k	52k

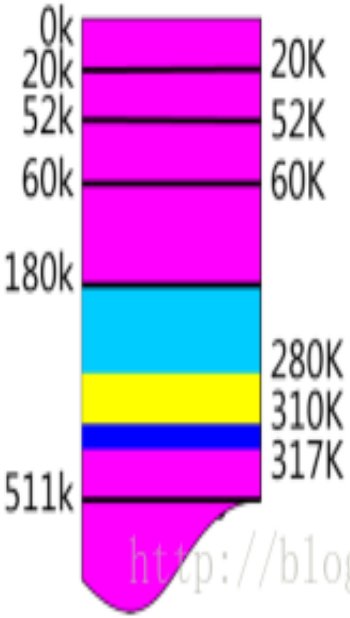
按容量递减的次序重新排列

作业100K分配后

区号	大小	起址
1	231k	280k
2	120k	60k
3	32k	20k
4	8k	52k

作业7K分配后

区号	大小	起址
1	194k	317k
2	120k	60k
3	32k	20k
4	8k	52k



区号	大小	起址
1	194k	317k
2	120k	60k
3	32k	20k
4	8k	52k

算法特点

- 最佳适应：若存在与作业大小一致的空闲分区,则它必然被选中，若不存在与作业大小一致的空闲分区，则只划分比作业稍大的空闲分区，从而保留了大的空闲分区。最佳适应算法往往使剩下的空闲区非常小，从而在存储器中留下许多难以利用的小空闲区（碎片）。
- 最坏适应算法的特点：总是挑选满足作业要求的最大的分区分配给作业。这样使分给作业后剩下的空闲分区也较大，可装下其它作业。由于最大的空闲分区总是因首先分配而划分，当有大作业到来时，其存储空间的申请往往会得不到满足。

练习题

- 在下列存储管理算法中，内存分配和释放平均时间之和为最大的是:

A 首次适应 B 下次适应 C 最佳适应 D 最差适应

练习题

- 在下列存储管理算法中，内存分配和释放平均时间之和为最大的是:

A 首次适应 B 下次适应 C 最佳适应 D 最差适应

需要寻找满足需要且最小的空闲块，
释放要找到上下邻空闲区，修改插入链表

练习题

- 可变分区又称为动态分区，它是在系统运行过程中___动态建立的:

A 作业未装入　　B 在作业装入

C 在作业创建　　D 在作业完成

练习题

- 可变分区又称为动态分区，它是在系统运行过程中___动态建立的:

A 作业未装入　　B 在作业装入

C 在作业创建　　D 在作业完成

分区大小在程序装入时大小动态确定，量身定制

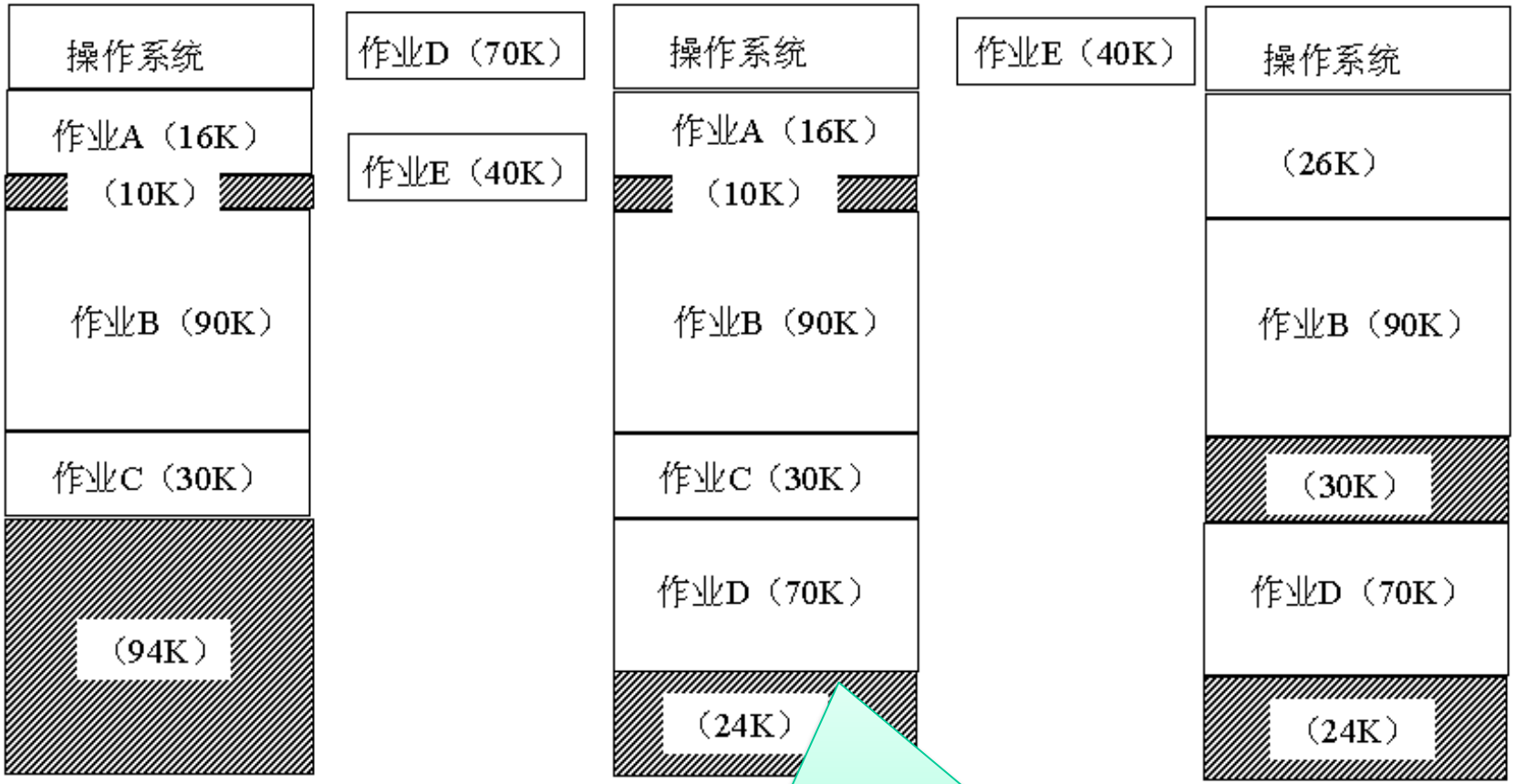
练习：可变分区的内存分配

- 假设一个可变分区系统中内存按照顺序包含如下空闲区： 10 KB, 4 KB, 20 KB, 18 KB, 7 KB, 9 KB, 12 KB, and 15 KB.
- 假设后续的连续内存分配请求是： (a) 12 KB (b) 10 KB (c) 9 KB
- 对于首次适应那个空闲区会被分配?对于 最佳适应, 最差适应, 和 下次适应 又会分配那些空闲区?

练习：可变分区的内存分配

- 首次适应 20 KB, 10 KB, 18 KB.
- 最佳适应 12 KB, 10 KB, 9 KB.
- 最差适应 20 KB, 18 KB, 15 KB.
- 下次适应 20 KB, 18 KB, 9 KB.

可变分区分配和回收的例子

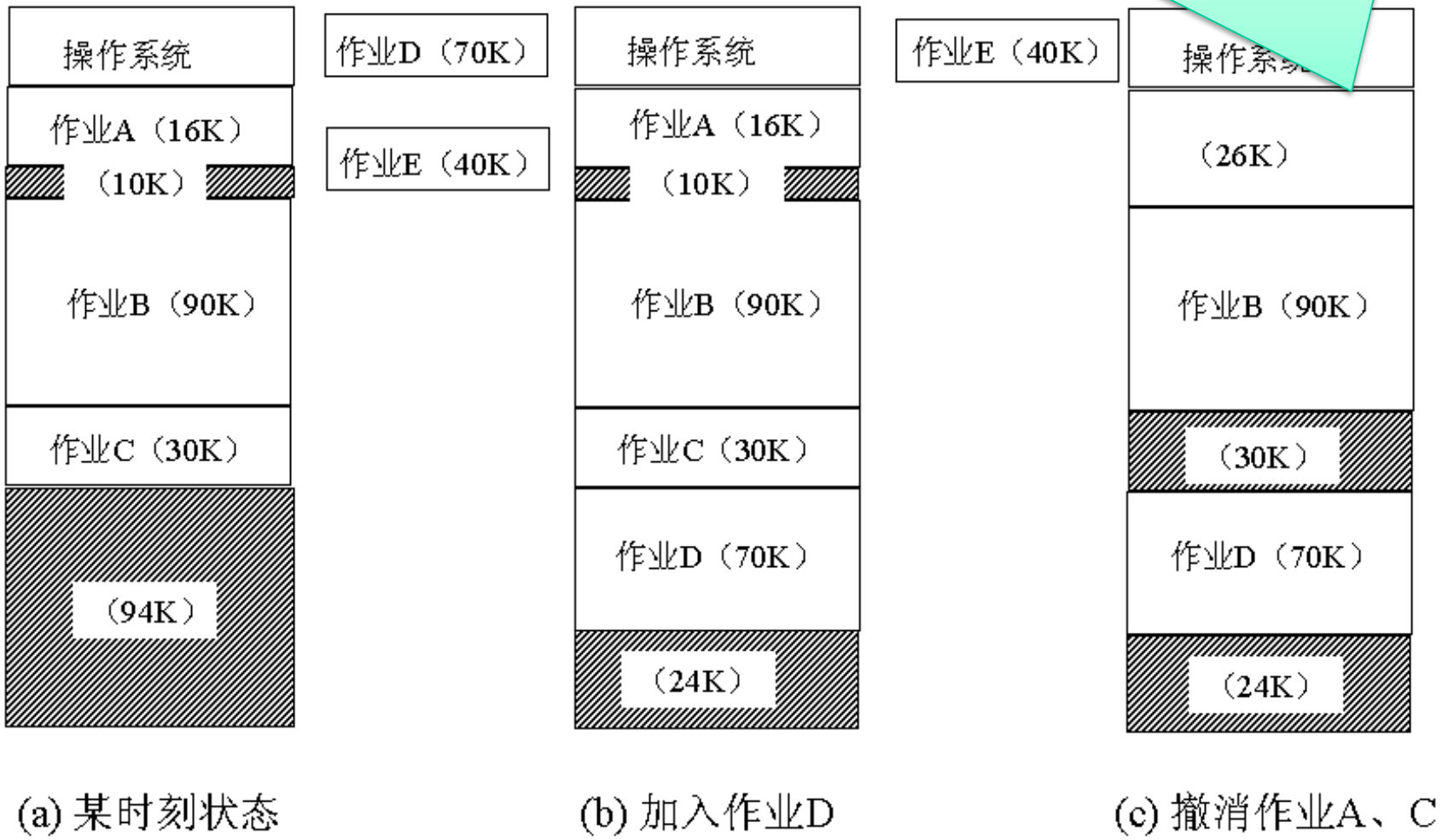


(a) 某时刻状态

作业D分配，产生24K空闲区

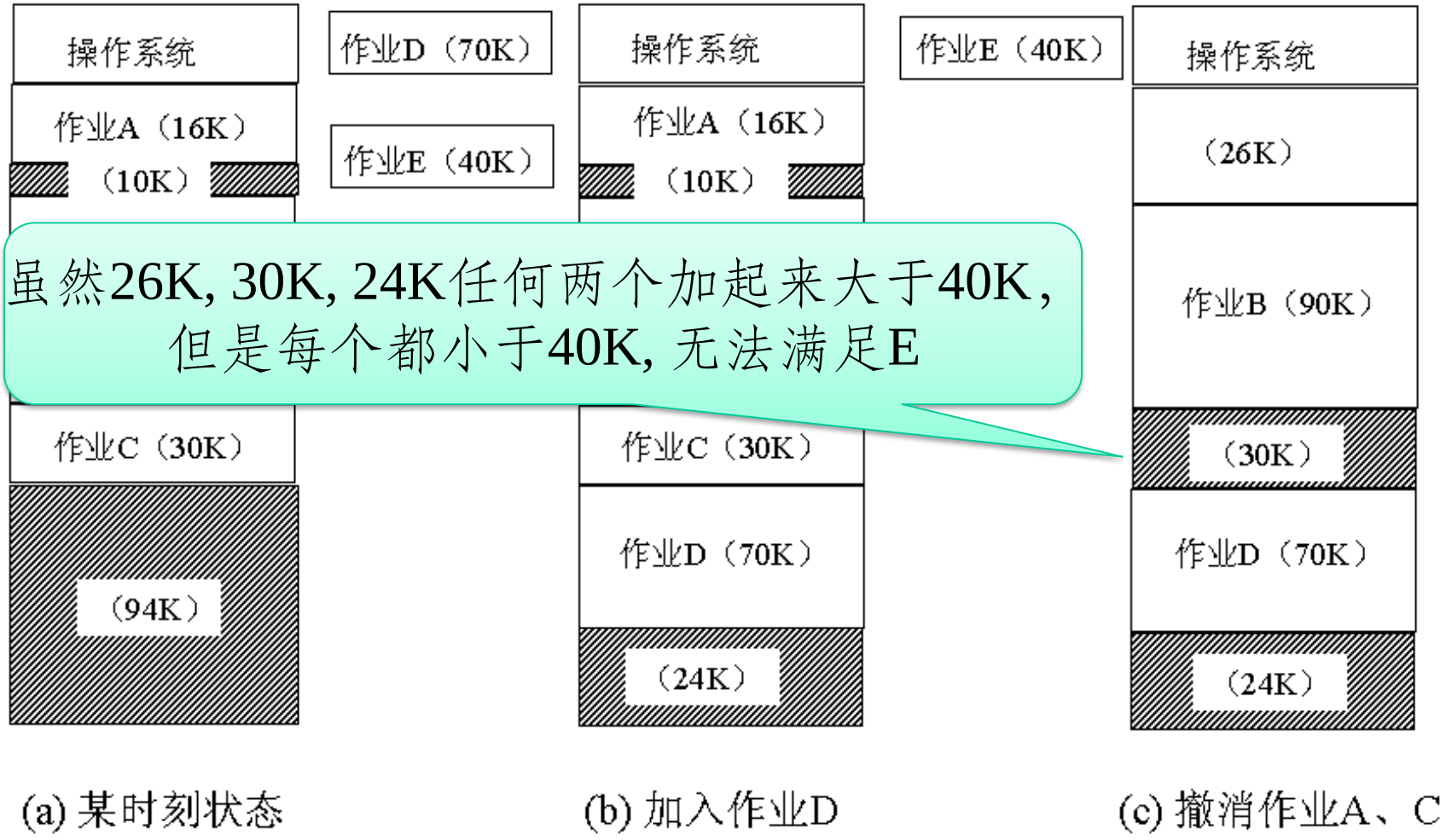
可变分区分配

作业A撤销，产生16K空闲区，合并成26K

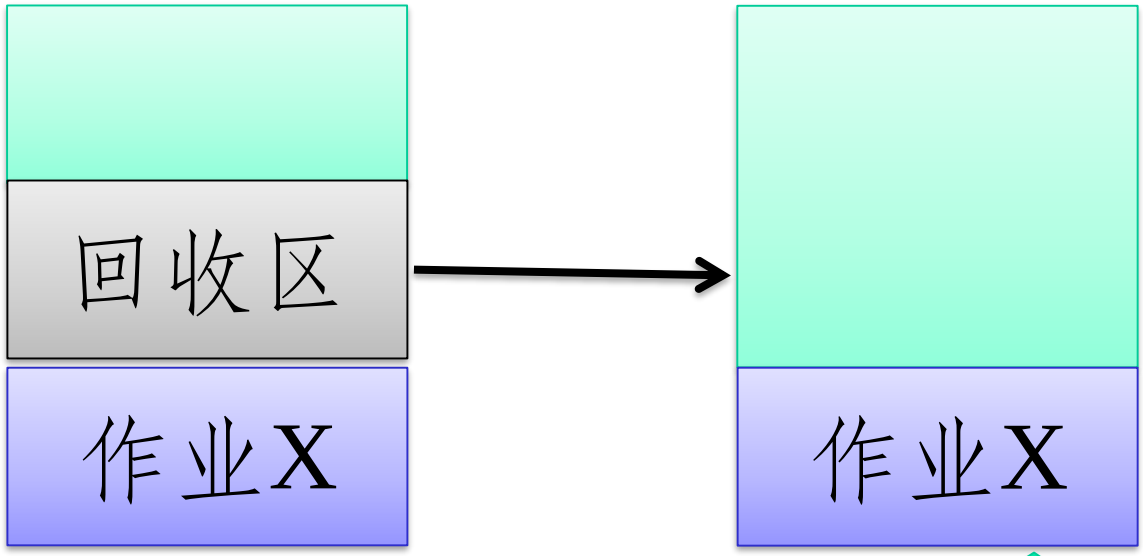


(a) 某时刻状态　(b) 加入作业D　(c) 撤消作业A、C

可变分区分配和回收的例子



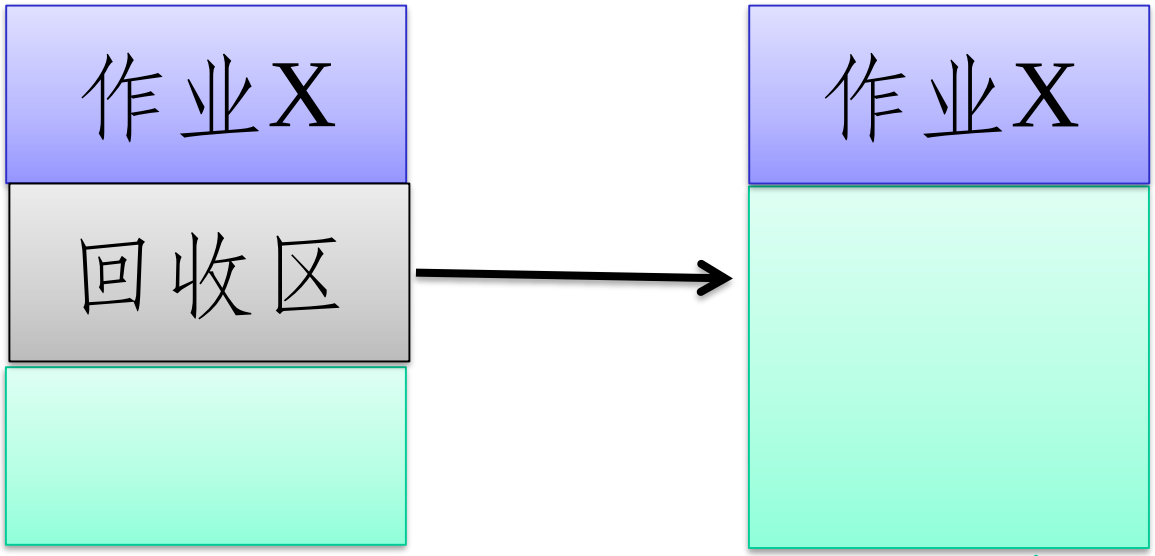
回收区域空白区邻接的四种情况



合并空闲区域，修改空闲区表

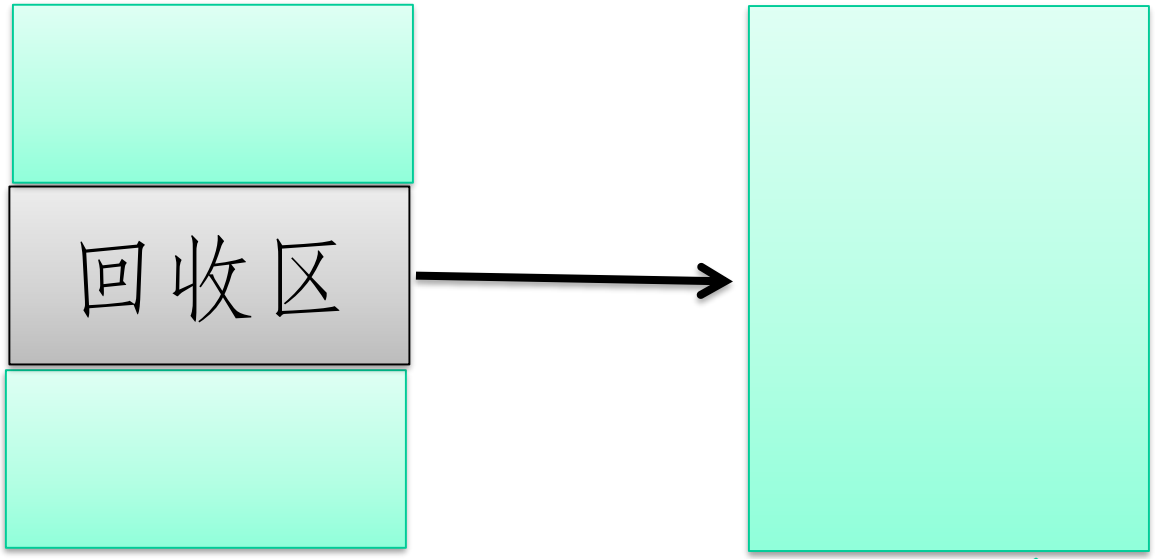
绿色表示空闲区

回收区域空白区邻接的四种情况



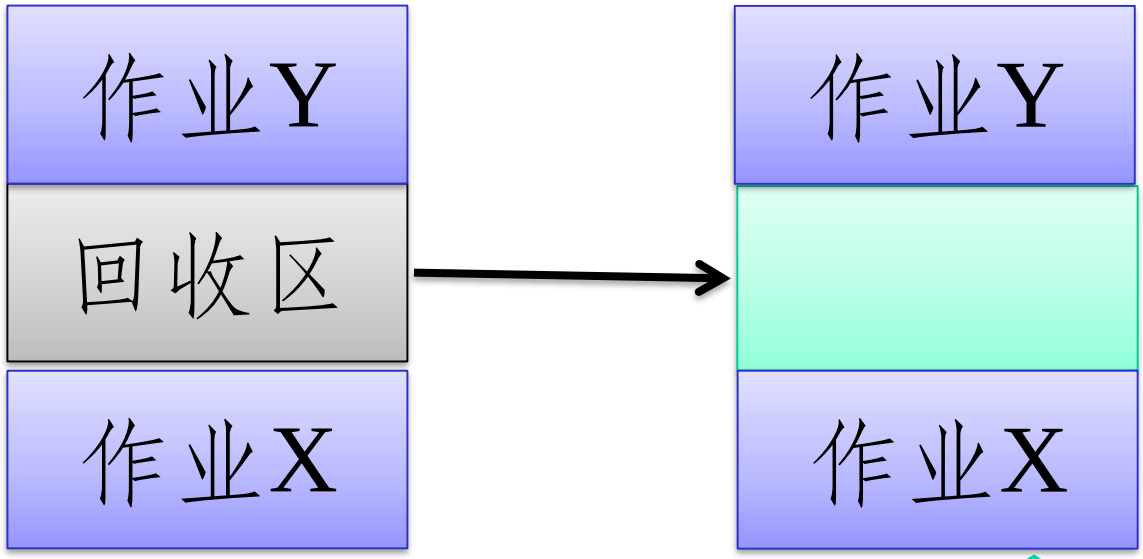
合并空闲区域，修改空闲区表

回收区域空白区邻接的四种情况



合并空闲区域，修改空闲区表

回收区域空白区邻接的四种情况



绿色表示空闲区

直接修改空闲区表

基于索引搜索的分配算法

- 基于顺序搜索的动态分区分配算法一般只是适合于较小的系统，如果系统的分区很多，空闲分区表（链）可能很大（很长），检索速度会比较慢。为了提高搜索空闲分区的速度，大中型系统采用了基于索引搜索的动态分区分配算法。

快速适应算法

- 快速适应算法，又称为分类搜索法，把空闲分区按容量大小进行分类，经常用到长度的空闲区设立单独的空闲区链表。系统为多个空闲链表设立一张管理索引表。

优点：

- 查找效率高，仅需要根据程序的长度，寻找到能容纳它的最小空闲区链表，取下第一块进行分配即可。该算法在分配时，不会对任何分区产生分割，所以能保留大的分区，也不会产生内存碎片。

缺点：

- 在分区归还主存时算法复杂，系统开销较大。在分配空闲分区时是以进程为单位，一个分区只属于一个进程，存在一定的浪费。空间换时间。

伙伴系统

- 固定分区方式不够灵活，当进程大小与空闲分区大小不匹配时，内存空间利用率很低。
- 动态分区方式算法复杂，回收空闲分区时需要进行分区合并等，系统开销较大。
- 伙伴系统 (buddy system)是介于固定分区与可变分区之间的动态分区技术。
- 伙伴：在分配存储块时将一个大的存储块分裂成两个大小相等的小块，这两个小块就称为“伙伴”。

Linux系统采用，同学们回去自学

伙伴系统

- 伙伴系统规定，无论已分配分区或空闲分区，其大小均为 2 的 k 次幂，k 为整数，$n \leq k \leq m$，其中：2^n 表示分配的最小分区的大小，2^m 表示分配的最大分区的大小，通常 2^m 是整个可分配内存的大小。
- 在系统运行过程中，由于不断的划分，可能会形成若干个不连续的空闲分区。
- 内存管理模块保持有多个空闲块链表，空闲块的大小可以为 1，2，4，8，…，2^m 字节。

伙伴系统的内存分配

系统初启时，只有一个最大的空闲块（整个内存）。

当一个长度为 n 的进程申请内存时，系统就分给它一个大于或等于所申请尺寸的最小的 2 的幂次的空闲块。

如果 $2^{i-1} < n \leq 2^i$，则在空闲分区大小为 2^i 的空闲分区链表中查找。

例如，某进程提出的 50KB 的内存请求，将首先被系统向上取整，转化为对一个 64KB 的空闲块的请求。

若找到大小为 2^i 的空闲分区，即把该空闲分区分配给进程。否则表明长度为 2^i 的空闲分区已经耗尽，则在分区大小为 2^{i+1} 的空闲分区链表中寻找。

伙伴系统的内存分配

若存在 2^{i+1} 的一个空闲分区，把该空闲分区分为相等的两个分区，这两个分区称为一对伙伴，其中的一个分区用于分配，另一个加入大小为 2^i 的空闲分区链表中。

若大小为 2^{i+1} 的空闲分区也不存在，需要查找大小为 2^{i+2} 的空闲分区，若找到则对其进行两次分割：第一次，将其分割为大小为 2^{i+1} 的两个分区，一个用于分割，一个加入到大小为 2^{i+1} 的空闲分区链表中；第二次，将用于分割的空闲区分割为 2^i 的两个分区，一个用于分配，一个加入到大小为 2^i 的空闲分区链表中。

若仍然找不到，则继续查找大小为 2^{i+3} 的空闲分区，以此类推。

伙伴系统的内存释放

首先考虑将被释放块与其伙伴合并成一个大的空闲块，然后继续合并下去，直到不能合并为止。

例如：回收大小为 2^i 的空闲分区时，若事先已存在 2^i 的空闲分区时，则应将其与伙伴分区合并为大小为 2^{i+1} 的空闲分区，若事先已存在 2^{i+1} 的空闲分区时，又应继续与其伙伴分区合并为大小为 2^{i+2} 的空闲分区，依此类推。

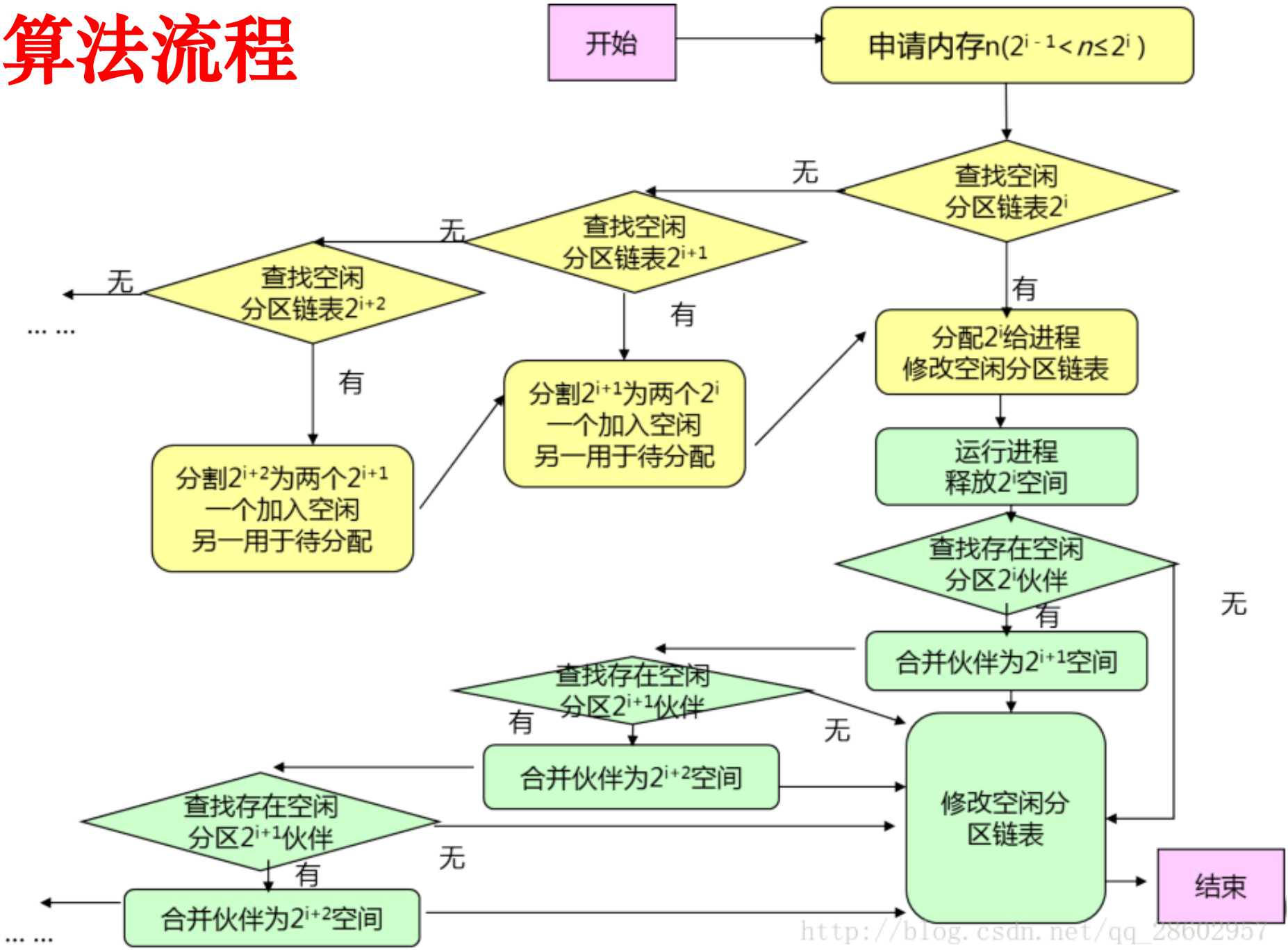
如果有两个存储块大小相同，地址也相邻，但不是由同一个大块分裂出来的（不是伙伴），则不会被合并起来。

一个例子

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					

算法流程



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。

系统中的碎片

- 内存中无法被利用的存储空间称为碎片。

内部碎片：

- 指分配给作业的存储空间中未被利用的部分，如固定分区中存在的碎片。
- 单一连续区存储管理、固定分区存储管理等都会出现内部碎片。
- 内部碎片无法被整理，但作业完成后会得到释放。它们其实已经被分配出去了，只是没有被利用。

系统中的碎片

外部碎片：

- 指系统中无法利用的小的空闲分区。如分区与分区之间存在的碎片。这些不连续的区间就是外部碎片。动态分区管理会产生外部碎片。
- 外部碎片才是造成内存系统性能下降的主要原因。外部碎片可以被整理后清除。
- 消除外部碎片的方法：紧凑技术。

紧凑技术（Compaction）

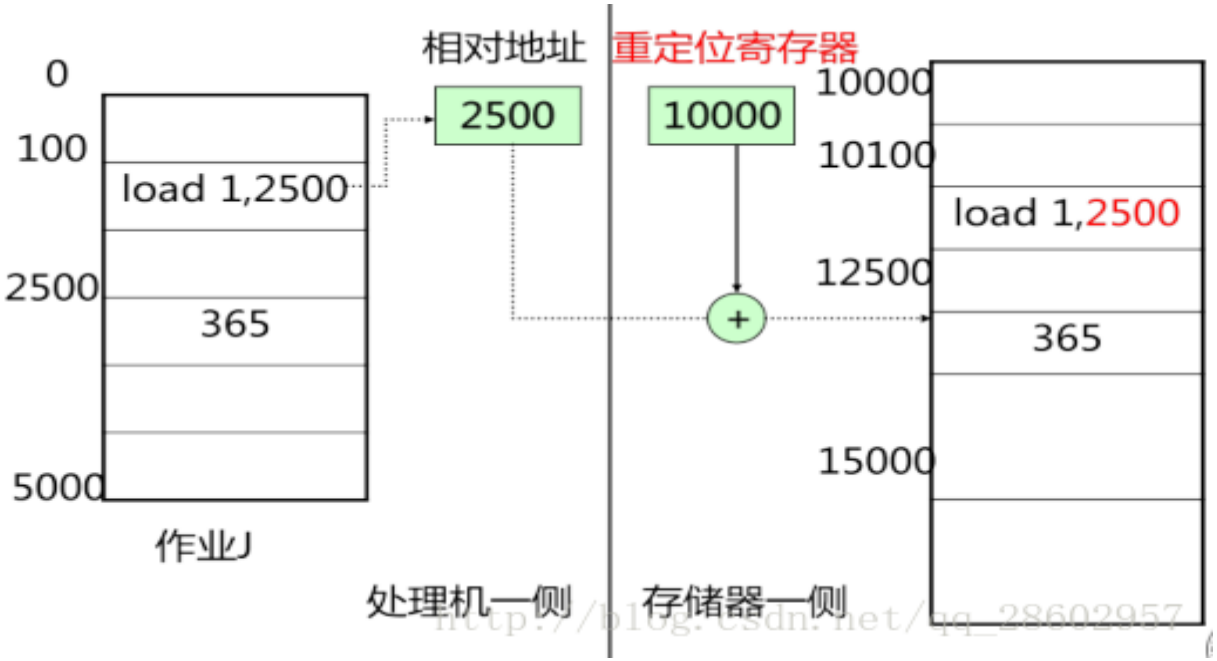
- 通过移动作业从把多个分散的小分区拼接成一个大分区的方法称为紧凑（拼接或紧缩）。
- 目标：消除外部碎片，使本来分散的多个小空闲分区连成一个大的空闲区。
- 紧凑时机：找不到足够大的空闲分区且总空闲分区容量可以满足作业要求时。

实现支撑

动态重定位：作业在内存中的位置发生了变化，这就必须对其地址加以修改或变换。

可重定位分区分配

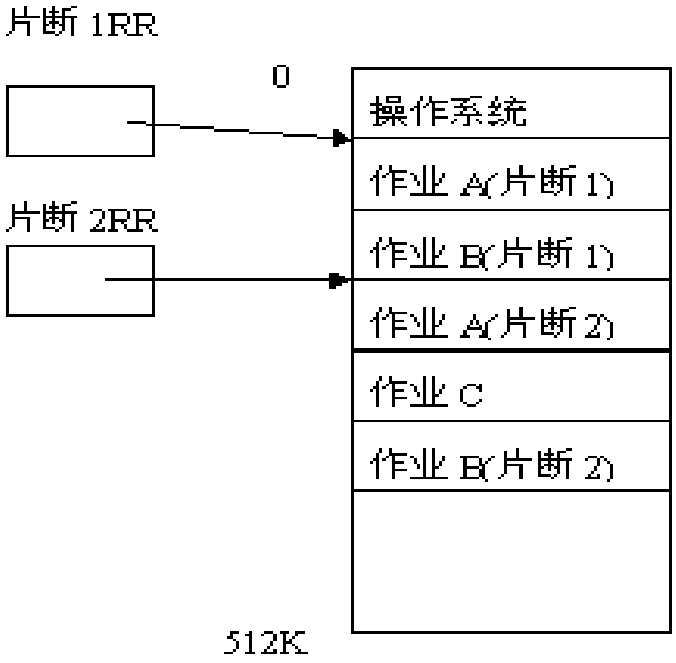
结合紧凑技术的动态分区技术



动态重定位的实现

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。

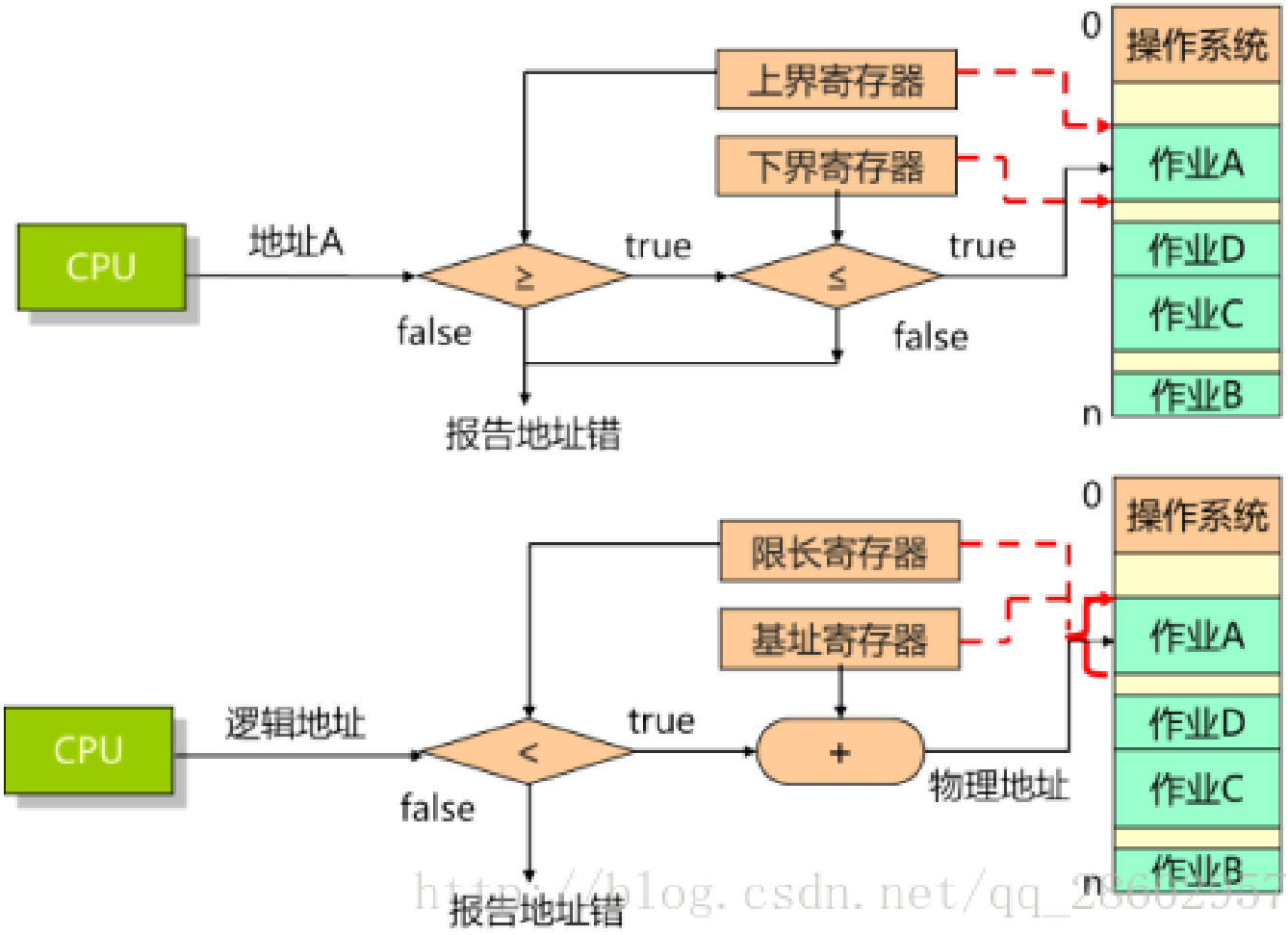


分区的存储保护

存储保护是为了防止一个作业有意或无意地破坏操作系统或其它作业。常用的存储保护方法有

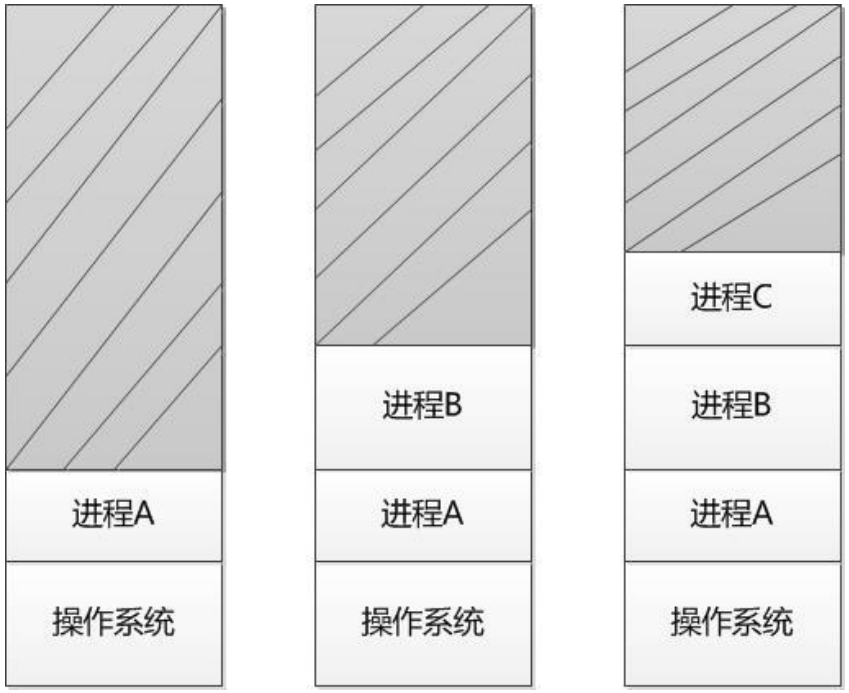
- 界限寄存器方法：
  - 上下界寄存器方法
  - 基址、限长寄存器 (BR,LR) 方法
- 存储保护键方法：
  - 给每个存储块分配一个单独的保护键，它相当于一把锁。进入系统的每个作业也赋予一个保护键，它相当于一把钥匙。当作业运行时，检查钥匙和锁是否匹配，如果不匹配，则系统发出保护性中断信号，停止作业运行。

界限寄存器法



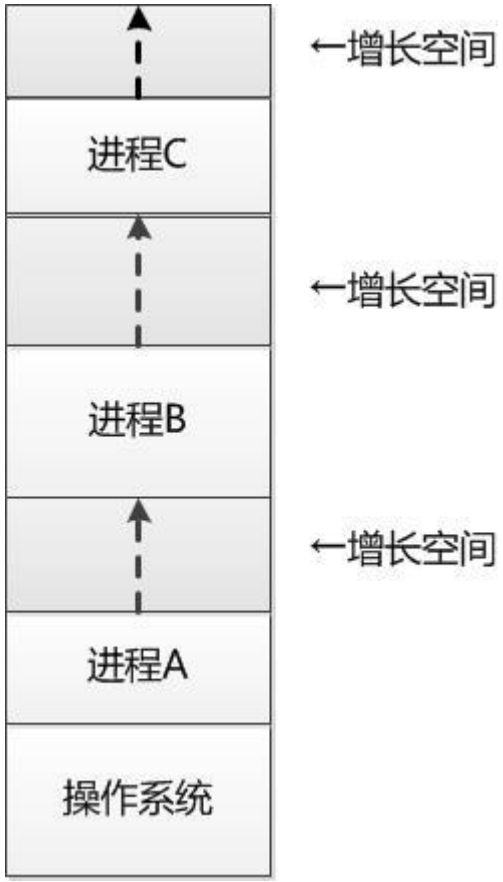
分区管理的问题

- 例如，一开始内存中只有OS，这时候进程A来了，于是分出一片与进程A大小一样的内存空间；随后，进程B来了，于是在进程A之上分出一片给进程B；然后进程C来了，就在进程B上面再分出一片给C。



问题

- 每个程序像叠罗汉一样累计，如果程序B在成型过程中需要更多空间怎么办？（例如在实际程序中，很多递归嵌套函数调用的时候回造成栈空间的增长）
- 预留一定的空间？OS怎么知道应该分配多少空间给一个程序呢？分配多了，就是浪费；而分配少了，则可能造成程序无法继续执行。



必须解决大作业在小内存中运行的问题。

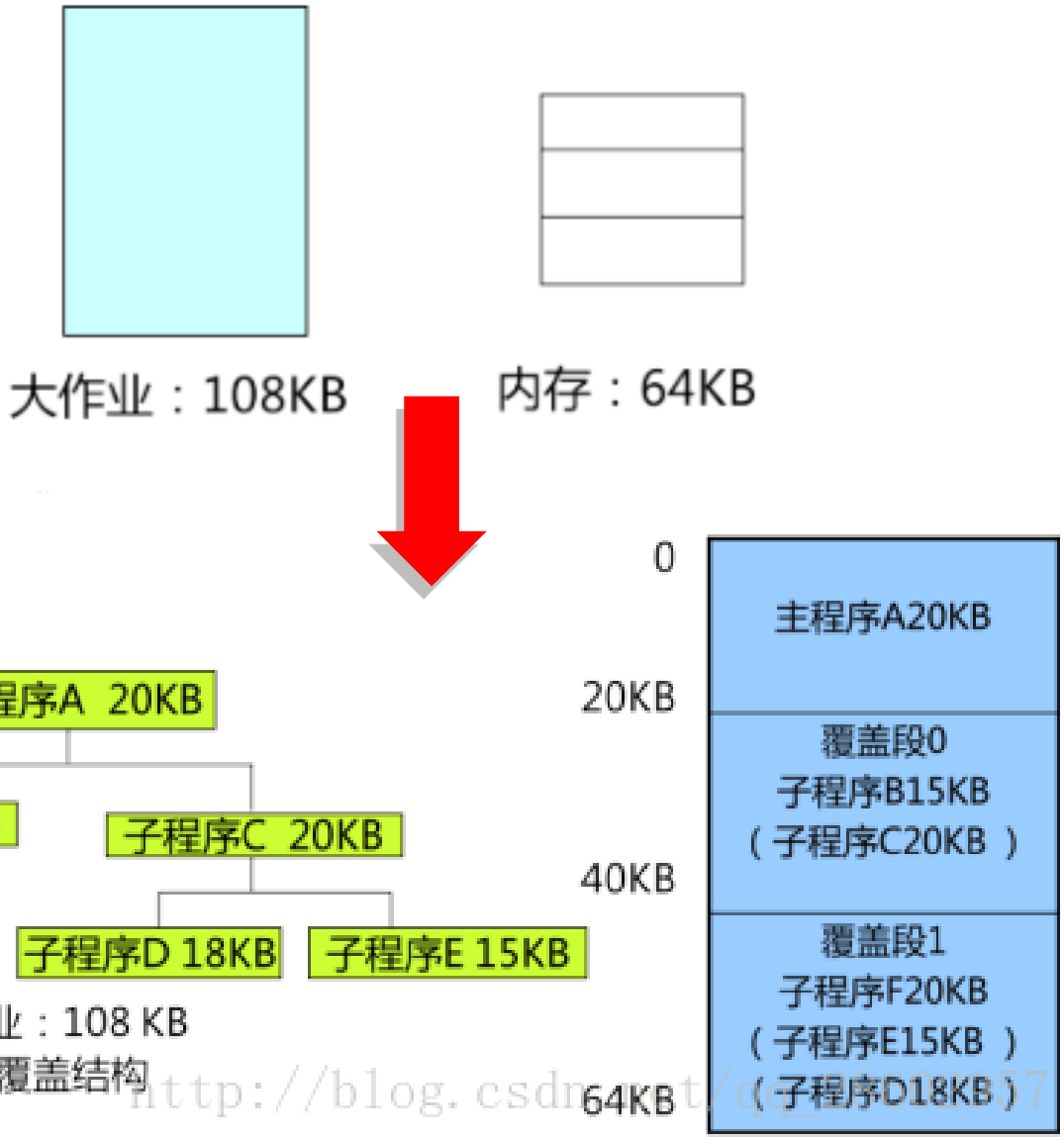
解决的方法

- **覆盖**与**交换**技术是在多道程序环境下用来扩充内存的两种方法。
- 覆盖与交换可以解决在小的内存空间运行大作业的问题，是“扩充”内存容量和提高内存利用率的有效措施。
- 覆盖技术主要用在早期的 OS 中，交换技术则用在现代OS 中。

覆盖 (Overlay)

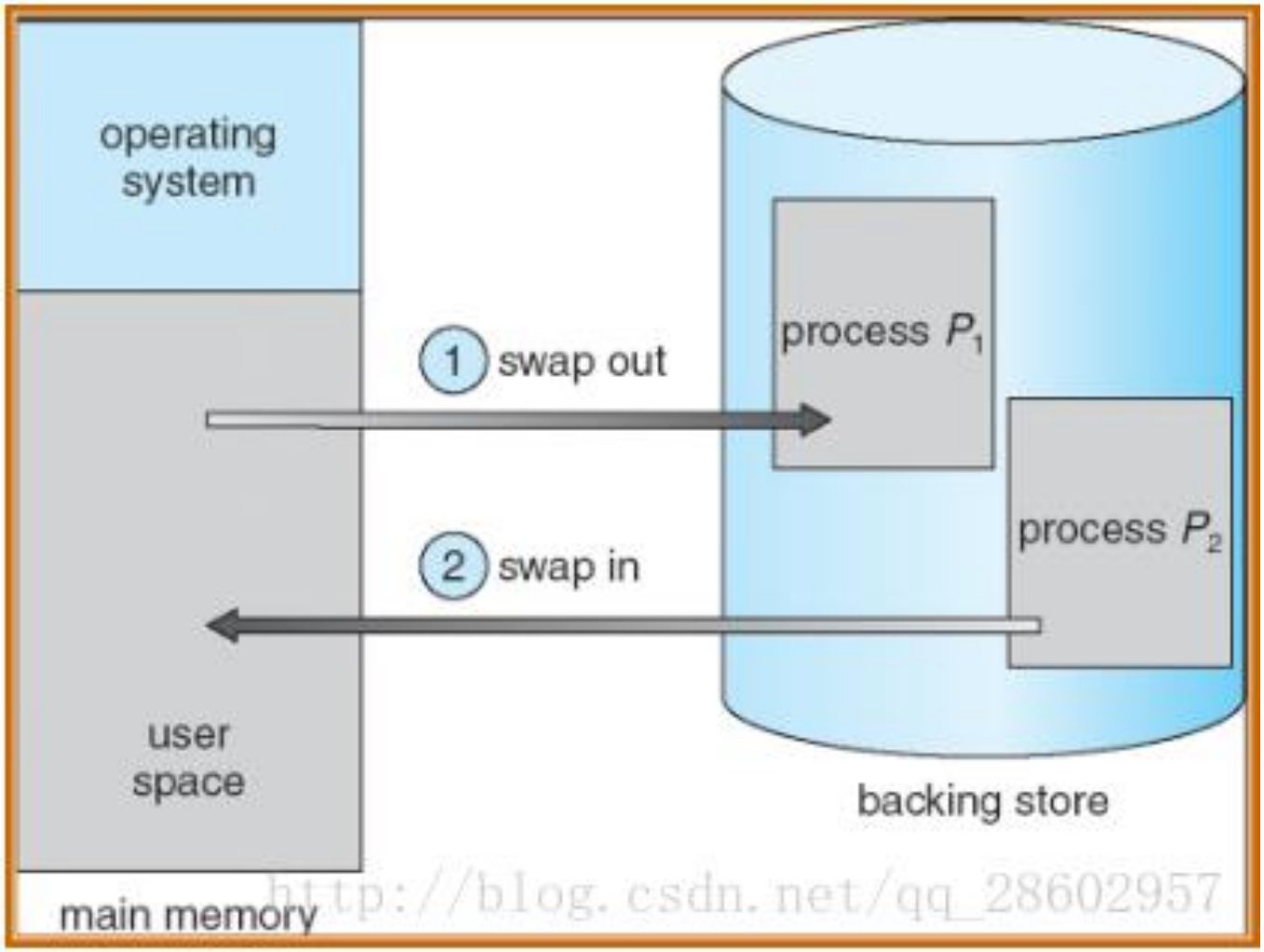
- 覆盖技术主要用在早期的 OS 中（内存 <64KB），可用的存储空间受限，某些大作业不能一次全部装入内存，产生了大作业与小内存的矛盾。
- 覆盖：把一个程序划分为一系列功能相对独立的程序段，让执行时不要求同时装入内存的程序段组成一组（称为覆盖段），共享主存的同一个区域，这种内存扩充技术就是覆盖。
- 程序段先保存在磁盘上，当有关程序段的前一部分执行结束，把后续程序段调入内存，覆盖前面的程序段（内存“扩大”了）。
- 一般要求作业各模块之间有明确的调用结构，程序员要向系统指明覆盖结构，然后由操作系统完成自动覆盖。
- 缺点：对用户不透明，增加了用户负担。

示例



交换（Swapping）

- 交换：广义的说，所谓交换就是把暂时不用的某个（或某些）程序及其数据的部分或全部从主存移到辅存中去，以便腾出必要的存储空间；接着把指定程序或数据从辅存读到相应的主存中，并将控制转给它，让其在系统上运行。
- 优点：增加并发运行的程序数目，并且给用户提供适当的响应时间；编写程序时不影响程序结构
- 缺点：对换入和换出的控制增加处理机开销；程序整个地址空间都进行传送，没有考虑执行过程中地址访问的统计特性。


operating system
process P_1
1 swap out
process P_2
2 swap in
user space
backing store
main memory
http://blog.csdn.net/qq_28602957

交换技术的几个问题

- 选择原则，即将哪个进程换出/内存?
  - 等待I/O的进程
- 交换时机的确定，何时需发生交换?
  - 只要不用就换出（很少再用）；只在内存空间不够或有不够的危险时换出
- 交换时需要做哪些工作?
- 换入回内存时位置的确定

覆盖与交换技术的区别

- 覆盖可减少一个程序运行所需的空间。交换可让整个程序暂存于外存中，让出内存空间。
- 覆盖是由程序员实现的，操作系统根据程序员提供的覆盖结构来完成程序段之间的覆盖。交换技术不要求程序员给出程序段之间的覆盖结构。
- 覆盖技术主要对同一个作业或程序进行。交换换主要在作业或程序间之间进行。

讨论

是否满足：

- 地址独立？
- 地址保护？

- 碎片问题、小内存运行大程序问题