CSAPP概念知识点整理

第一章

*1. 编译系统

- 源程序(hello.c)通过预处理器(.cpp)得到修改了的源程序(hello.i)
- .i文件经过编译器(ccl)得到汇编程序(hello.s)。
- .s文件经过汇编器(as)得到可重定位目标文件(hello.o)。
- .o文件和库文件通过连接器(ld)得到可执行目标文件(hello)
  其中，.c、.i、.s为文本文件（即在汇编阶段之前的都为文本文件），经过汇编后转换为二进制文件。

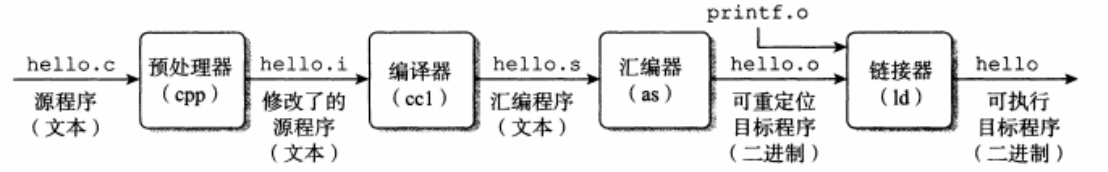


图 1-3　编译系统

2. CPU会执行的操作

- 加载：从主存复制一个字节或一个字到寄存器，以覆盖寄存器原来的内容；
- 存储：从寄存器复制一个字节或一个字到主存的某个位置，以覆盖这个位置上原来的内容；
- 操作：把两个寄存器的内容复制到ALU，ALU对这两个字做算术运算，并将结果存放到一个寄存器中，以覆盖该寄存器中原来的内容；
- 跳转：从指令本身中抽取一个字，并将这个字复制到程序计数器(PC)中，以覆盖PC中原来的值。

3. 运行hello程序

- 初始时，shell程序执行他的指令，等待我们输入一个命令。当我们在键盘上输入字符串“./hello”后，shell程序将字符逐一读入寄存器，再把他们放到主存中。
- 当我们在键盘上敲击回车键时，shell程序知道我们已经结束了命令的输入。然后shell执行一系列指令来加载可执行的hello文件，这些指令将hello目标文件中的代码和数据从磁盘复制到主存。（数据不经过处理器而直接到达主存）
- 代码和数据被加载到主存后，处理器开始执行hello程序的main程序中的机器语言指令。指令将”hello,world\n”字符串中的字节从主存复制到寄存器文件，再从寄存器文件中复制到显示设备。

4. 进程的虚拟地址空间

- 程序代码和数据。对所有的进程来说，代码是从同一固定地址开始的，紧接着是和C全局变量相对应的数据位置。在进程一开始就被指定了大小。
- 堆。代码和数据区后紧随着的是运行时堆。堆可以在运行时动态地扩展和收缩。
- 共享库。
- 栈。位于用户虚拟地址空间顶部的是用户栈，编译器用它来实现函数调用。
- 内核虚拟内存。为内核保留的，不允许应用程序读写这个区域的内容或直接调用内核代码定义的函数。

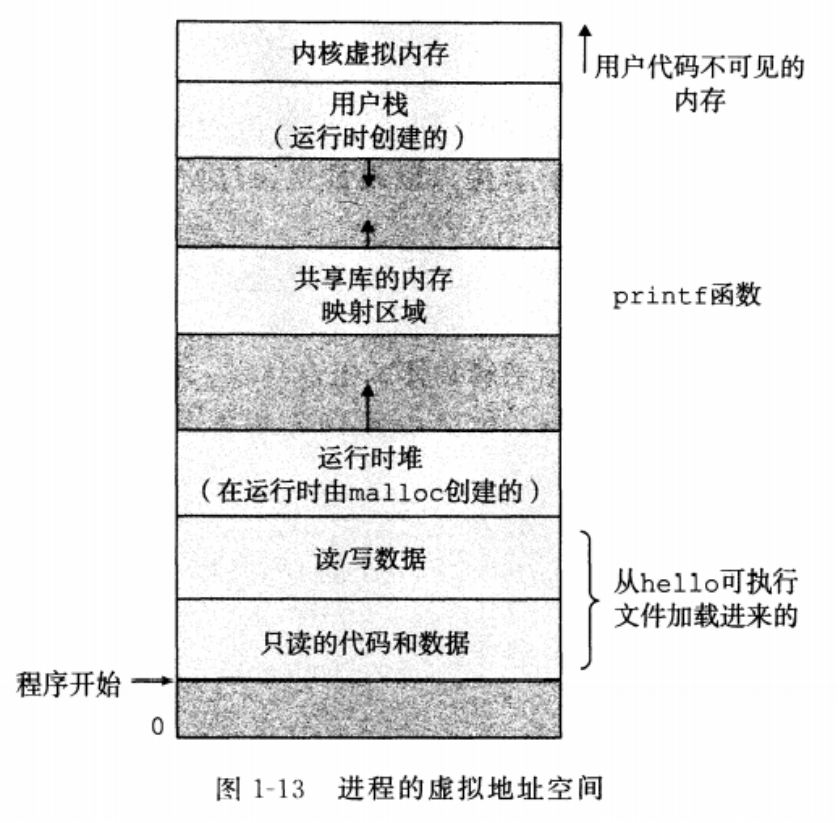


图 1-13　进程的虚拟地址空间

5. 并行和并发

- 线程级并发：构建在进程这个抽象之上，通过使一台计算机在他正在执行的进程间快速切换来实现的。
- 指令级并行：在较低的抽象层次上，处理器可以同时执行多条指令的属性被称为指令级并发。如果处理器可以达到比一个周期一条指令更快地执行速率，就称之为超标量。
- 单指令，多数据并行：在最低的层次上现代处理器拥有特殊的硬件，允许一条指令产生多个可以并行执行的操作，即SIMD。

*6. 计算机系统中的抽象

- I/O设备的抽象为文件
- 内存+I/O设备抽象为虚拟内存
- 处理器的抽象为指令集架构
- 处理器+内存+I/O设备抽象为进程
- 处理器+内存+I/O设备+操作系统抽象为虚拟机

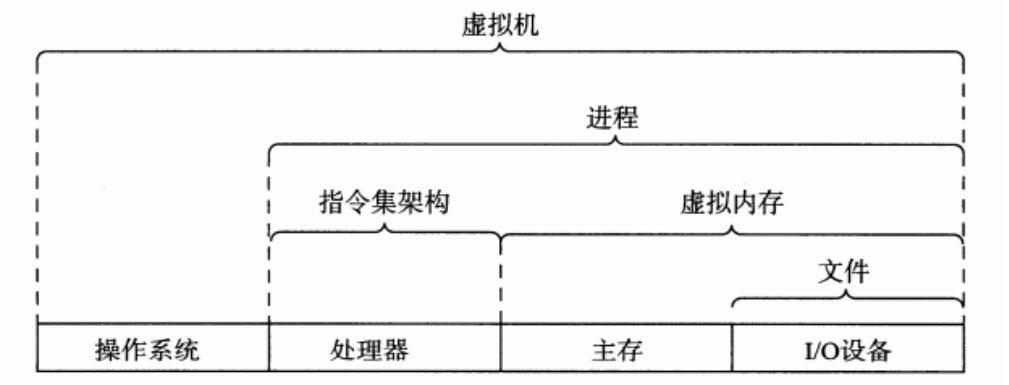


图 1-18　计算机系统提供的一些抽象。计算机系统中的一个重大主题就是提供不同层次的抽象表示，来隐藏实际实现的复杂性

第二章

7.大端法小端法

Linux32、Windows和Linux 64上使用小端法;
Sum使用大端法。
但是字符串都是用大端法。

8.C语言中的逻辑运算

逻辑运算符&&和||，对于第一个参数求值就能确定表达式的结果，那么逻辑运算符就不会对第二个参数求值。（如果有一些引用空指针和除零操作就不会导致程序终止）

9.浮点数的舍入

浮点数默认舍入方式是向偶数舍入，也称向最接近的值舍入。’[

*10.在int、float和double之间进行强制类型转换

- int转换城float，不会溢出，但可能被舍入
- 从int或float转换城double，能够保留准确的数值
- 从double转换城float，会溢出，且由于精度较小，会舍入
- 从float或double转成int，值会向零舍入。值可能会溢出。

第三章

11.gcc命令执行过程

- C预处理器扩展源代码，插入所有用#include命令指定的文件，并扩展所有用#define声明指定的宏。
- 编译器产生两个源文件的汇编代码。
- 汇编器将汇编代码转换成二进制目标代码文件。它包含所有指令的二进制表示，但是还没有填入全局值的地址。
- 链接器将目标代码文件与实现库函数的代码合并，并最终产生可执行代码文件。

12.机器代码可见的处理器状态

- 程序计数器
- 寄存器文件
- 条件码寄存器
- 向量寄存器

*13.16个寄存器的特殊作用

- 返回值：rax
- 传参：rdi、rsi、rdx、rcx、r8、r9
- 被调用者保存（调用者使用时需压栈）：rbx、rbp、r12、r13、r14、r15
- 栈指针：rsp

14.一些指令要求的寄存器

- SAR、SHR移位操作移位量只能是一个立即数或者是存放在cl中。
- 乘法操作，一个参数必须存放在rax中，计算出来的结果，高64位存放在rdx中，低64位存放在rax中。
- 除法操作，rax中的作为被除数，商存放在rax中，余数存放在rdx中。

15.条件码

除了leaq指令外所有的指令都会改变条件码。

16.惩罚时间计算

Tavg(p)=(1−p)Tok+p(Tok+Tmp)=Tok+pTmp。

17.对抗缓冲区溢出攻击

- 栈随机化：使得栈的位置在每次程序运行时都有变化。在linux系统中，栈随机化已经变成了标准行为。它属于地址空间布局随机化的一种。采用ASLR，每次运行时程序的不同部分都会被加载到内存的不同区域。“空操作雪橇”可以破除栈随机化。
- 栈破坏检测：在栈帧任意局部缓冲区与栈状态之间存储一个特殊的金丝雀值，也称为哨兵值。在恢复寄存器状态和从函数返回之前，程序检查这个金丝雀值是否被改变，如果是，程序异常中止。
- 限制可执行代码区域

第四章

*18.寄存器文件的读写及其他

- 读寄存器时，虽然寄存器不是组合电路（因为它有内部存储）。但在我们的实现中，从寄存器文件读数据就好像它是一个以地址为输入，数据为输出的组合逻辑块。
- 写寄存器时，明显用到内部存储功能，所以不被当作组合逻辑。
- 将随机访问存储器看成和组合逻辑一样的操作（在读时）（寄存器文件、指令内存和数据内存）。
- 程序计数器、条件码寄存器、数据内存和寄存器文件。写时通过一个时钟信号来控制。只有在执行整数运算指令时，才会装载条件码寄存器。只有执行rmmovq、pushq或call指令时，才会写数据内存。

19.Y86-64的顺序实现

分为六个步骤，分别是取值、译码、执行、访存、写回、更新。所有硬件单元的处理都在一个时钟周期内完成。

20.流水线冒险

- 数据相关：下一条指令会用到这一条指令计算出的结果。
- 控制相关：一条指令要确定下一条指令的位置。这些相关可能会导致流水线产生计算错误，称为冒险。冒险分为数据相关和控制相关。

措施：

- 用暂停来避免数据冒险。处理器会停止流水线中一条或多条指令，直到冒险条件不再满足。
- 用转发来避免数据冒险。将结果值从一个流水线阶段传到较早阶段的技术成为数据转发。
- 加载/使用转发需要用暂停加转发解决，因为读内存在M阶段。

- 当处理器无法根据当前取值阶段的指令来确定下一条指令的地址时，就会出现控制冒险。控制冒险只会出现在跳转指令和ret指令。ret指令通过插入三个气泡解决。。jump指令在下一周期往译码和执行阶段插入气泡，并同时取出跳转指令后面的指令，就能取消（指令排除）那两条预测错误的指令。

21.异常处理

有流水线中最深的指令引起的异常，优先级最高。

22.流水线控制逻辑

- 加载/使用冒险：在一条从内存中读出一个值的指令和一条使用该值的指令之间，流水线必须暂停一个周期。
- 处理ret：流水线必须暂停知道ret指令到达写回阶段。
- 预测错误的分支：在分支逻辑发现不应该选择分支之前，分支目标处的几条指令已经进入流水线了。必须取消这些指令，并从跳转指令后面的那条指令开始取指。
- 异常：当一条指令导致异常，我们想要禁止后面的指令更新程序员可见的状态，并且在异常指令到达写回阶段时，停止执行。

处理办法：

- 加载使用冒险：暂停F和D阶段，在E阶段插入气泡；
- ret指令：暂停取值阶段，在译码阶段插入气泡（可以不让下一条指令执行下去），重复三次，直到ret指令进行完M阶段，取出跳转地址。
- 分支跳转错误：在译码（取消第二条）和执行阶段（取消第一条）插入气泡，取消两条不正确的指令。
- 异常：当异常指令到达访存阶段，我们会采取措施防止后面的指令修改程序员可见的状态：①禁止执行阶段中的指令设置条件码②向内存阶段中插入气泡，以禁止向数据内存中写入③暂停写回阶段，因此暂停了流水线。

发现方法

- ret指令：检查译码、执行和访存阶段中指令的指令码；
- 加载使用冒险：检查执行阶段中的指令类型，并把它的目的寄存器与译码阶段中的源寄存器相比较。
- 跳转指令：在执行阶段发现错误，在访存阶段开始设置从错误中恢复所需要的条件。
- 异常：通过检查访存和写回阶段中的指令状态值。

控制条件的组合

- 一条跳转指令处于执行阶段（结果会是不跳转），跳转目标处有一条ret指令处于译码阶段。会将取指阶段暂停，并在译码和执行阶段插入气泡，因为执行阶段被插入气泡，ret指令被取消，下一阶段流水线正常运行，会将跳转指令下一条指令读入。
- ret指令处于译码阶段，访存指令处于执行阶段。将ret的执行推迟一个周期，将译码阶段暂停。

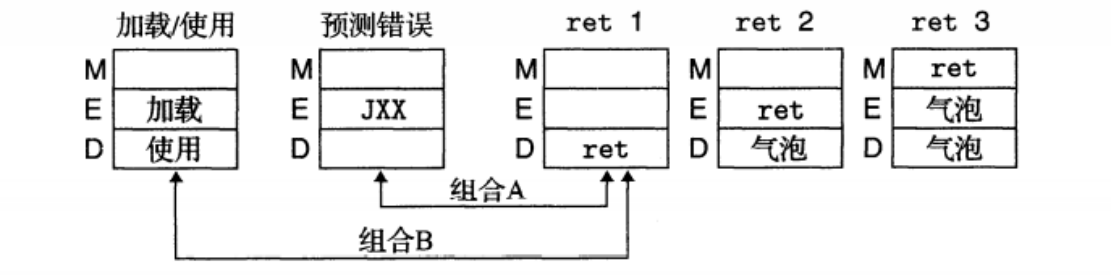


图 4-67　特殊控制条件的流水线状态。图中标明的两对情况可能同时出现

第五章

23.编译器安全的优化

遇到内存别名使用（两个指针可能指向同一个内存位置），编译器必须假设不同的指针可能会指向内存中的同一位置，从而不产生可能的优化版本，限制了优化策略。（同样的还有改变全局变量值的函数，执行多次和执行一次效果不同）

措施

- 循环不变式外提：识别要执行多次但是计算结果不会改变的计算，需要由程序员帮助编译器显式的完成代码的移动。
- 减少调用过程（函数）
- 消除不必要的内存引用
- （上面是代码层次优化，简单地降低了过程调用的开销，下面是处理器微体系结构层次优化）
- 循环展开：通过增加每次迭代计算的元素数量，减少循环的迭代次数。减少整个计算中关键路径上的操作数。
- 提高并行性：多个累计变量。通常，只有保持能够执行该操作的所有功能单元的流水线都是满的，程序才能达到这个操作的吞吐量界限。对延迟为L，容量为C的操作而言，要求循环展开因子k》C X L。
- 提高并行性：重新结合变换，改变计算的顺序。可以减少关键路径上操作的数量。

24.优化的限制因素

- 寄存器溢出：就需要将临时值存放在内存中。
- 分支预测和预测错误处罚。
- 读写相关。

第六章

25.增强的DRAM

- 快页模式DRAM：同一行连续的访问可以直接从行缓冲区得到服务。
- 扩展数据输出DRAM：允许各个CAS信号在时间上靠的更紧密一点。
- 同步DRAM：能够更快地输出超单元的内容。
- 双倍数据速率同步
- 视频RAM

26.磁盘操作时间

- 寻道时间Tseek：移动传送臂到目标磁道的时间。
- 旋转时间Trot：等待目标扇区的第一个字节到传送臂下的时间，最大为 $T_{\text{max rotation}} = \frac{1}{\text{RPM}} \times \frac{60\text{s}}{1\text{min}}$（磁盘转动一圈的时间），平均为最大的一半。
- 传送时间：$T_{\text{avg transfer}} = \frac{1}{\text{RPM}} \times \frac{1}{(\text{平均扇区数} / \text{磁道})} \times \frac{60\text{s}}{1\text{min}}$（就是走完一个扇区所需要的时间）

*27.局部性

程序倾向于引用邻近于其他引用过数据项的数据项，或者最近引用过的数据项本身。

- 时间局部性：被引用过一次的内存位置很可能在不远的将来再被多次引用。

- 空间局部性：如果一个内存引用位置被引用了一次，那么程序很可能在不远的将来引用附近的一个内存位置。

局部性原理的一些简单原则：

- 重复引用相同变量的程序具有好的时间局部性
- 对于具有步长为k的引用模式的程序，步长越小，空间局部性越好。
- 对于取指令来说，循环有好的时间和空间局部性，循环体越小，循环迭代的次数越多，局部性越好。

28.缓存不命中的种类

- 强制性不命中（冷不命中）：空的缓存被称为冷缓存。冷不命中只是短暂的不命中。
- 冲突不命中
- 容量不命中：工作集大小超过缓存的大小。

*29.缓存管理

- 编译器管理寄存器文件，决定当发生不命中时何时加载发射，以及确定哪个寄存器来存放数据
- L1、L2、L3层的缓存（SRAM)完全由内置在缓存中的硬件逻辑来管理。
- 在有虚拟内存的系统中，DRAM主存作为存储在磁盘上的数据块的缓存，由操作系统和CPU上的地址翻译硬件共同管理。

第七章

30.静态链接

静态链接器以一组可重定位目标文件和命令行参数为输入，生成一个完全链接的，可以加载和运行的可执行目标文件作为输出。

为了构造可执行文件，链接器必须完成两个主要任务：

- 符号解析。目标文件定义和引用符号，每个符号对应于一个函数、一个全局变量或一个静态变量。符号解析的目的是将每个符号引用正好和一个符号定义关联起来。
- 重定位。编译器和汇编器生成从地址0开始的代码和数据节。连接器通过把每个符号定义与一个内存位置关联起来，从而重定位这些节，然后修改所有对这些符号的引用，使得他们指向这个内存位置。链接器使用汇编器产生的重定位条目的详细指令，不加甄别地执行这样的重定位。

31.目标文件

目标文件有三种形式：

- 可重定位目标文件。包含二进制代码和数据，其形式可以在编译时与其他可重定位目标文件合并起来，创建一个可执行目标文件。
- 可执行目标文件：包含二进制代码和数据，其形式可以直接复制到内存并执行。
- 共享目标文件：特殊类型的可重定位目标文件，可以加载或运行时被动态的加载进内存并链接。

32.文件节的内容

- .text：已编译程序的机器代码（立即数也放这）
- .eodata：只读数据，比如printf语句串中的格式串和开关语句的跳转表。
- .data：已初始化的全局变量和静态C变量。局部C变量在运行时被保存在栈中。

- .bss：未初始化的静态C变量，以及所有被初始化为0的全局或静态变量。
- （因为.bss中不允许有重复的名字，而COMMON中可以）

*33.符号

- 全局符号：由模块m定义并能被其他模块引用的全局符号，全局链接器符号对应于非静态的C函数和全局变量。
- 外部符号：由其他模块定义并被模块m引用的全局符号。对应于在其它模块中定义的非静态C函数和全局变量。
- 本地（局部）符号：带static属性的C语言和全局变量。

34.符号表

- ABS表示不该被重定位的符号
- UNDEF代表未定义的符号，也就是在本目标模块中引用，但是却在其他地方定义的符号
- COMMOM存储未初始化的全局符号

*35.静态库

将所有相关的目标函数被编译成独立的目标模块，然后封装成一个单独的静态库文件，称为静态库。链接时，链接器将只复制被程序引用的目标模块。

36.重定位工作

- 重定位节和符号定义：链接器将所有相同类型的节合并为同一类型的新的聚合节。然后，链接器将运行时内存地址赋给新的聚合节，赋给输入模块定义的每个节，以及赋给输入模块定义的每个符号。这步完成后，程序中的每条指令和全局变量都有唯一的运行时内存地址了。
- 重定位节中的符号引用：链接器修改代码节和数据节中对每个符号的引用，使得他们指向正确的运行时地址。

*37.动态链接共享库

共享库是一个目标模块，在运行或加载时，可以加载到任意的内存地址，并和一个在内存中的程序链接起来。这个过程称为动态链接，是由一个叫做动态链接器的程序来执行的。
特点：任何给定的文件系统中，对于一个库只有一个.so文件。所有引用该库的可执行目标文件共享这个.so文件中的代码和数据（而不用复制到自己的可执行文件中）。在内存中，一个共享库的.text节的一个副本可以被不同正在运行的进程共享。

从应用程序中加载和连接共享库

```
#include <dlfcn.h>

void *dlopen(const char *filename, int flag);
                                        返回：若成功则为指向句柄的指针，若出错则为 NULL。
```

dlopen函数加载和链接共享库filename。
flag参数要么包含RTLD_NOW（告诉链接器立刻解析对外部符号的引用）
要么包含RTLD _LAZY（指示链接器推迟符号解析直到执行来自库中的代码）

```
#include <dlfcn.h>

void *dlsym(void *handle, char *symbol);
                                        返回：若成功则为指向符号的指针，若出错则为 NULL。
```

dlsym 函数的输入是一个指向前面已经打开了的共享库的句柄和一个 symbol 名字，如果该符号存在，就返回符号的地址，否则返回 NULL。

（就是获取动态库中的函数用的）

38.位置无关代码

可以加载而无需重定位的代码称为位置无关代码。
延迟绑定：只需要重定位被使用的目标代码而不重定位无需使用的。

PIC数据引用

每个引用全局目标的目标模块都有自己的GOT表。在GOT中，每个被这个目标模块引用的全局数据目标（过程或全局变量）都有一个8字节条目。编译器还为每个GOT中每个条目生成一个重定位记录。在加载时，动态链接器会重定位GOT中的每个条目，使得它包含目标的正确的绝对地址。

39.库打桩机制

- 编译时打桩：在编译时用指令 `linux> gcc -DCOMPILETIME -c mymalloc.c`

  `linux> gcc -I. -o intc int.c mymalloc.o` 由于用 -I.参数，所以会进行打桩。它告诉C预处理器在搜索通常的系统目录之前，先在当前目录中查找malloc.h。
- 链接时打桩

  `linux> gcc -Wl,--wrap,malloc -Wl,--wrap,free -o intl int.o mymalloc.o`

  （是对int.o文件进行的操作，先将int.c文件经过预处理和编译变成int.c文件和，在链接时打桩）

  用下述方法把这些源文件编译成可重定位目标文件：

  ```
  linux> gcc -DLINKTIME -c mymalloc.c
  linux> gcc -c int.c
  ```
- 运行时打桩：只需要访问可执行目标文件。基于动态链接器的LD_PRELOAD环境变量。如果该环境变量被设置为一个共享库路径名的列表，当你加载和执行一个程序，需要解析未定义的引用时，动态链接器会先搜索LR _PRELOAD库，然后才搜索任何其他库!

  ```
  linux> gcc -DRUNTIME -shared -fpic -o mymalloc.so mymalloc.c -ldl
  ```

  这是如何编译主程序：

  ```
  linux> gcc -o intr int.c
  ```

  下面是如何从 bash shell 中运行这个程序[⊖]：

  ```
  linux> LD_PRELOAD="./mymalloc.so" ./intr
  ```

第八章

40.异常号

系统中的可能的每种类型的异常都分配了一个唯一的非负整数得异常号。包括被零除、缺页、内存访问违例、断点以及算数运算溢出在内的异常是由处理器的设计者分配的，其他包括系统调用和来自外部I/O设备的信号是由操作系统内核的设计者分配的。

*41.异常的类别

- 中断：来自I/O设备的信号，是异步的，且总是返回到下一条指令
- 陷阱：有意的异常（系统调用），是同步的，总是返回到下一条指令
- 故障：潜在可恢复的错误（如缺故异常），是同步的，可能返回到当前指令，也可能不返回
- 终止：不可恢复的错误，是同步的，不会返回。

异常

- 0：除法错误。报告为浮点异常
- 13：一般保护故障：引用一个未定义的虚拟内存区域或写只读的文本段。报告为段异常
- 14：缺页
- 18：机器检查：终止。

42.系统调用

所有系统调用的参数都是通过寄存器而不是栈传递的。寄存器rax包含系统调用号，寄存器rdi、rsi、rdx、rcx、r8、r9传递参数。调用结束后，rcx和r11会被破坏，rax包含返回值。如果是负数则表明发生了错误。

*43.上下文信息

内核为每一个进程维持一个上下文。上下文就是内核重新启动一个被抢占的进程所需的状态。它由一些对象的值组成，这些对象包括**通用目的寄存器、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，比如描述地址空间的页表、包含有关当前进程信息的进程表、以及包含进程已打开文件的信息的文件表。**

*44.上下文切换

- 当内核代表用户执行系统调用时，可能发生上下文切换。
- 中断可能引发上下文切换

45.进程状态

进程总是处于下面三种状态之一：

- 运行：进程要么在CPU上运行，要么在等待被执行且最终会被内核调度。
- 停止：进程的执行被挂起且不会被调度。
- 终止：进程永远的停止了。造成终止有三种原因：①收到行为为终止进程的信号②从主程序返回③调用exit

*46.函数的返回值

- getpid：返回一次，返回调用者的pid
- exit：不返回
- fork：返回两次，子进程返回0，父进程返回子进程的pid
- waitpid：返回一次，返回子进程的pid；或者statusp被设置为WNOHANG时，返回0
- sleep：返回还要休眠的秒数
- pause：总是返回-1
- execve：不返回

*47.shell运行的过程

- 解析命令行函数解析以空格分割的命令行参数，并构造最终会传递给execve的argv向量。第一个参数被假设为要么

是一个内置的shell命令名，马上就会解释这个命令，要么是一个可执行目标文件，会在一个新的子进程的上下文加载并运行这个文件。

- 如果最后一个参数是一个”&“字符，解析函数返回1，表示应该在后台执行该程序（shell不会等待它完成）。否则返回0，表示应该在前台执行（shell会等待他完成）。
- 解析命令行完成后，eval函数调用builtin_comman函数，检查第一个命令行参数是否是一个内置的shell命令，如果是，理解是这个命令并返回1，否则返回零。
- 如果builtin command函数返回0，shell创建一个子进程，并在子进程中执行所请求的程序。如果用户要求后台执行，则shell返回到循环顶部，等待下一个命令行，否则使用waitpid函数等待作业终止。

48.信号

- 进程试图除0，内核发送给给它一个8.SIGFPE
- 进程试图执行非法指令，发送4.SIGILL
- 进程试图进行非法内存引用，11.SIGSEGV
- 键入Ctrl+c，发送2.SIGINT
- 9.SIGKILL会强制终止一个进程
- 一个子进程终止或者停止时，内核会发送一个17.SIGCHLD给父进程

序号	名称	默认行为	相应事件
1	SIGHUP	终止	终端线挂断
2	SIGINT	终止	来自键盘的中断
3	SIGQUIT	终止	来自键盘的退出
4	SIGILL	终止	非法指令
5	SIGTRAP	终止并转储内存①	跟踪陷阱
6	SIGABRT	终止并转储内存①	来自 abort 函数的终止信号
7	SIGBUS	终止	总线错误
8	SIGFPE	终止并转储内存①	浮点异常
9	SIGKILL	终止②	杀死程序
10	SIGUSR1	终止	用户定义的信号 1
11	SIGSEGV	终止并转储内存①	无效的内存引用（段故障）
12	SIGUSR2	终止	用户定义的信号 2
13	SIGPIPE	终止	向一个没有读用户的管道做写操作
14	SIGALRM	终止	来自 alarm 函数的定时器信号
15	SIGTERM	终止	软件终止信号
16	SIGSTKFLT	终止	协处理器上的栈故障
17	SIGCHLD	忽略	一个子进程停止或者终止
18	SIGCONT	忽略	继续进程如果该进程停止
19	SIGSTOP	停止直到下一个 SIGCONT②	不是来自终端的停止信号
20	SIGTSTP	停止直到下一个 SIGCONT	来自终端的停止信号
21	SIGTTIN	停止直到下一个 SIGCONT	后台进程从终端读
22	SIGTTOU	停止直到下一个 SIGCONT	后台进程向终端写
23	SIGURG	忽略	套接字上的紧急情况
24	SIGXCPU	终止	CPU 时间限制超出
25	SIGXFSZ	终止	文件大小限制超出
26	SIGVTALRM	终止	虚拟定时器期满
27	SIGPROF	终止	剖析定时器期满
28	SIGWINCH	忽略	窗口大小变化
29	SIGIO	终止	在某个描述符上可执行 I/O 操作
30	SIGPWR	终止	电源故障

49.阻塞和解除阻塞信号

- 隐式阻塞机制：阻塞同类型
- 显式阻塞机制：用sigprocmask来明确地阻塞和接触阻塞选定的信号

*50.异步信号安全的函数

产生输出的唯一安全的方法是使用write函数。printf和sprintf是不安全的。（malloc和exit也是不安全的）

_Exit	fexecve	poll	sigqueue
_exit	fork	posix_trace_event	sigset
abort	fstat	pselect	sigsuspend
accept	fstatat	raise	sleep
access	fsync	read	sockatmark
aio_error	ftruncate	readlink	socket
aio_return	futimens	readlinkat	socketpair
aio_suspend	getegid	recv	stat
alarm	geteuid	recvfrom	symlink
bind	getgid	recvmsg	symlinkat
cfgetispeed	getgroups	rename	tcdrain
cfgetospeed	getpeername	renameat	tcflow
cfsetispeed	getpgrp	rmdir	tcflush
cfsetospeed	getpid	select	tcgetattr
chdir	getppid	sem_post	tcgetpgrp
chmod	getsockname	send	tcsendbreak
chown	getsockopt	sendmsg	tcsetattr
clock_gettime	getuid	sendto	tcsetpgrp
close	kill	setgid	time
connect	link	setpgid	timer_getoverrun
creat	linkat	setsid	timer_gettime
dup	listen	setsockopt	timer_settime
dup2	lseek	setuid	times
execl	lstat	shutdown	umask
execle	mkdir	sigaction	uname
execv	mkdirat	sigaddset	unlink
execve	mkfifo	sigdelset	unlinkat
faccessat	mkfifoat	sigemptyset	utime
fchmod	mknod	sigfillset	utimensat
fchmodat	mknodat	sigismember	utimes
fchown	open	signal	wait
fchownat	openat	sigpause	waitpid
fcntl	pause	sigpending	write
fdatasync	pipe	sigprocmask	

*51.非本地跳转

setjump函数在env缓冲区中保存当前调用环境，以供后面的longjump使用，并返回0.调用环境包括程序计数器、栈指针和通用目的寄存器。

```
#include <setjmp.h>

int setjmp(jmp_buf env);
int sigsetjmp(sigjmp_buf env, int savesigs);
```

返回：setjmp返回0，longjmp返回非零。

longjump函数从env缓冲区中恢复调用环境，然后触发一个从最近一次初始化env的setjump的调用的返回。然后setjump返回，并带有非零的返回值retval

```
#include <setjmp.h>

void longjmp(jmp_buf env, int retval);
void siglongjmp(sigjmp_buf env, int retval);
```

setjump处的返回值

从不返回。

第九章

52.虚拟页面的种类

- 未分配的
- 缓存的
- （已分配）未缓存的

53.页面置换

判断虚拟页是否在DRAM中、在哪个物理页、替换策略等步骤，都是由软硬件联合提供的，包括操作系统软件、MMU（内存管理单元）中的地址翻译硬件和一个存放在物理内存中的叫做页表的数据结构，页表将虚拟页映射到物理页。
页表是一个页表条目（PTE）的数组。

54.虚拟内存作为内存管理的工具

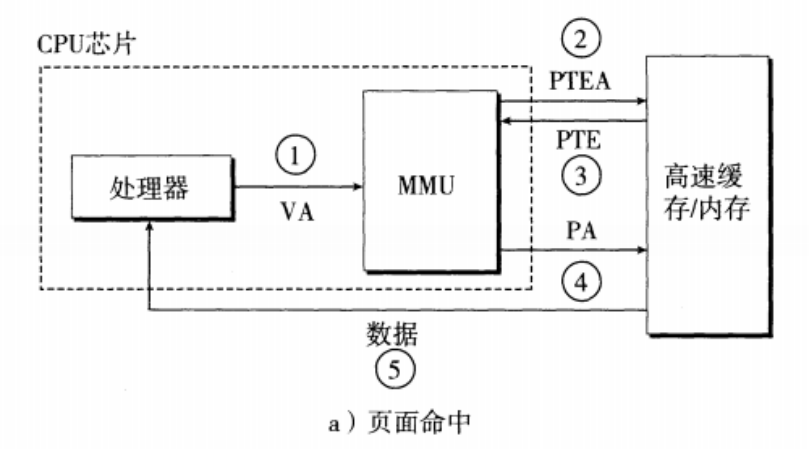
按需页面调度和独立的虚拟地址空间的结合，对系统中内存的使用和管理造成了深远的影响。

- 简化链接：独立的地址空间允许每个进程的内存映像使用相同的基本格式，而不管代码和数据实际存放在物理内存的何处。
- 简化加载：容易向内存中加载可执行文件和共享对象文件。
- 简化共享：独立地址空间为操作系统提供了一个管理用户进程和操作系统自身之间共享的一致机制。
- 简化内存分配。

*55.地址转换操作（异常由MMU触发）

页命中（完全是由硬件控制的）

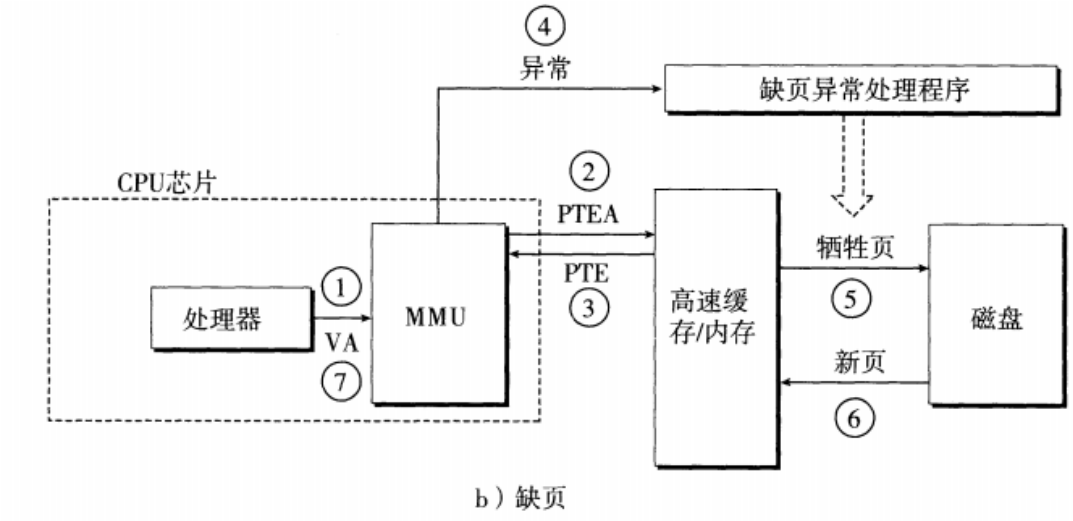
- CPU生成一个虚拟地址，传送给MMU
- MMU生成PTE地址，从高速缓存/内存请求得到它（就是去搜索页表中的页表项）
- 高速缓存/主存向MMU返回PTE
- MMU构造物理地址，并把它传送给高速缓存/主存
- 高速缓存/主存返回所请求的数据字给CPU



a）页面命中

页不命中（由硬件和操作系统内核协作完成）

- 1-3步和命中相同
- PTE有效位是0，所以MMU触发一次异常，传递CPU中的控制到操作系统内核中的缺页异常处理程序。
- 缺页处理程序确定出物理内存中的牺牲页，如果这个页面已经被修改了，则把它患处到磁盘。
- 缺页处理程序调用新的页面，并更新内存中的PTE。
- 缺页处理程序返回到原来的进程，再次执行导致缺页的指令。



b）缺页

图 9-13 页面命中和缺页的操作图（VA：虚拟地址。PTEA：页表条目地址。PTE：页表条目。PA：物理地址）

56.TLB

在MMU中包括了一个关于PTE的小的缓存，称为翻译后备缓冲器(TLB)

57.多级页表的好处

- 减少了内存要求。一、如果一级页表中的一个一个PTE是空的，那么相应的二级页表根本就不会存在。代表一种巨大的潜在节约。二、只有一级页表才需要总是在主存中。

*58.内存映射

虚拟内存区域可以映射到两种类型的对象中的一种：

- Linux文件系统中的普通文件
- 匿名文件

59.execve函数运行过程

execve函数在当前进程中加载并运行包含可执行目标文件Aa.out中的程序，用a.out程序有效的代替了当前程序。加载并运行a.out的步骤：
* 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已存在的区域结构。
* 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。所有这些新的区域结构都是私有的、写时复制的。代码和数据区域被映射为a.out文件中的.text和.data区。bss区是请求二进制零的，映射到匿名文件，其大小包含在a.out种。栈和堆区域也是请求二进制零的，初始长度为0。
* 映射共享区域。如果a.out程序与共享对象（或目标）链接，那么这些对象是动态链接到这个程序的，然后再映射到用户虚拟地址空间中的共享区域内。
* 设置程序计数器(PC)。使之指向代码区域的入口点。

60.使用mmap函数的用户级内存映射

```
#include <unistd.h>
#include <sys/mman.h>

void  *mmap(void *start, size_t length, int prot, int flags,
            int fd, off_t offset);
```

虚拟内存起始点 (仅仅是一个暗示，通常被定义为NULL)

从文件描述符fd指定的对象offset处的length字节对象映射

返回：若成功时则为指向映射区域的指针，若出错则为 MAP_FAILED(−1)。

*61.动态内存分配

对于每个进程，内核维护这一个变量brk，指向堆的顶部。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。

- 显示分配器：要求应用显式地释放任何已分配的块
- 隐式分配器：（垃圾收集器）分配器检测一个一分配块何时不再被程序所使用，就释放这个块。

*62.垃圾收集器

垃圾收集器将内存视为一张有向可达图。

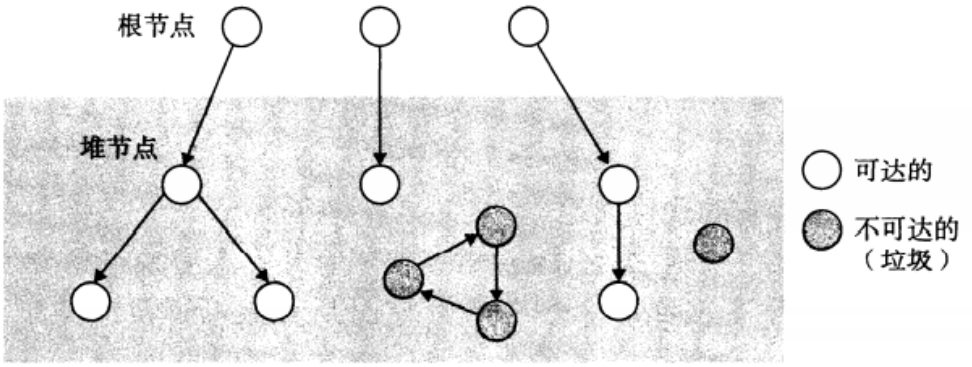


图 9-49　垃圾收集器将内存视为一张有向图

每个堆节点对应于堆中的一个已分配块。有向边P->Q意味着块P中地某个位置指向块Q中地某个位置。根节点对应于这样一种不在堆中的位置，它们中包含指向堆中的指针。这些位置可以是寄存器、栈里的变量，或者是细腻内存中读写数据区域的全局变量。

63.块分配策略

- 首次适配：从头开始搜索空闲链表，选择第一个合适的空闲块。优点是趋于将大的空闲块保留在链表的后面。缺点

是它趋向于在靠近的链表起始处留下小空闲块碎片，增加了较大块的搜索时间

- 下一次适配：从上一次查询结束的地方开始。内存利用率低
- 最佳适配：利用率高，缺点是要求对堆进行彻底的搜索。

第十章

64.文件权限

每个进程自己都有一个umask，关于打开文件的权限是文件自己的权限mode & ~ umask。

65.元数据

应用程序能通过调用stat和fstat函数，检索到关于文件的信息（也称为文件的元数据）。
st_mode成员编码了文件访问许可位和文件类型。

shell运行

结合fork，execve函数，简述在shell中加载和运行hello程序的过程。
每步骤1分
① 在shell命令行中输入命令：$./hello
② shell命令行解释器构造argv和envp；
③ 调用fork()函数创建子进程，其地址空间与shell父进程完全相同，包括只读代码段、读写数据段、堆及用户栈等
④ 调用execve()函数在当前进程（新创建的子进程）的上下文中加载并运行hello程序。将hello中的.text节、.data节、.bss节等内容加载到当前进程的虚拟地址空间
⑤ 调用hello程序的main()函数，hello程序开始在一个进程的上下文中运行。