1. **空间与地址分配**
2. **链接器为目标文件分配地址和空间**

* 输出的可执行文件中的空间
* 装载后的虚拟地址中的虚拟地址空间

1. **整个链接过程分两步：**

* 空间与地址分配
* 符号解析与重定位

在链接之前，虚拟空间还没有被分配，目标文件中的所有段的VMA都是0。

1. **链接步骤：**

* 给各个段分配相应的虚拟地址空间。
* 计算各个符号的虚拟地址，各个符号在段内的相对位置是固定的，这里只需加上偏移量
* 每个要重定位的段都有一个重定位表，重定位表里是需要被调整的位置的偏移量，重定位的过程中，每个重定位入口都是对一个符号的引用，当链接器要对某个符号的引用进行重定位时，它就是要确定这个符号的目标地址，就会去查找所有输入目标文件的符号表所组成的全局符号表，找到相应的符号后进行重定位。
* 总结上述：**重定位表+全局符号表=重定位**

1. **指令修正**

* 绝对寻址修正=**（全局符号表获取的目标地址）**实际地址+**（偏移量位置的值）**被修正位置的值
* 相对寻址修正=实际地址+被修正位置的值-被修正的位置**（重定位表获取的偏移量）**

1. **多个文件出现同一变量多个定义的情况（COMMON块机制）**

两个或两个以上强符号类型；无需处理，非法；

有一个强符号，其他都是弱符号，出现类型不一致；以强符号为准；

两个或两个以上弱符号类型不一致；以输入文件中最大的为准；

1. **C++相关问题**
2. **重复代码消除**

出现重复代码的情况：模板、外部内联函数（extern inline function）、虚函数表等

模板的解决方法：每个模板的实例代码单独的放在一个段里，当别的编译单元中也生成相同的实例段后，链接器会区分这些相同的模板实例段，然后把它们合并入一个段。

函数级别链接：所有函数单独保存在一个段里，当链接器需要用到某个函数时，就将它合并到输出文件中，对于没用的函数就抛弃。（visual c++编译器提供的机制）

1. **全局构造和析构**

Main函数调用之前：初始化进程执行环境（设置栈指针+初始化DATA区+将BSS区清零+构造全局对象+将argc、argv参数传入main函数）。

全局对象的析构函数在main之后执行。

* .init段:进程的初始化指令，，当一个程序开始运行时，在main函数被调用之前，Glibc的初始化部分安排执行这个段中的代码。
* .fini段，保存着进程终止代码指令，当一个进程的main函数正常退出时，Glibc会安排执行这个段中的代码

1. **二进制兼容**

ABI(Application Binary Interface)代表与可执行代码二进制兼容相关的内容

1. **静态库链接**
2. **语言库与系统调用**

语言库就是对操作系统API的包装。

* 部分函数需要调用系统API，eg：printf
* 部分函数不需要调用系统API，eg：strlen

1. **静态链接**

静态库可简单的看成一组目标文件集合，即很多目标文件进过压缩打包后形成的文件。

Ld链接器会自动寻找所有需要的符号及它们所在的目标文件

链接过程：

* 调用cc1编译成一个临时的汇编文件
* 调用as将临时汇编文件汇编成临时目标文件
* Collect2程序完成最后的链接，collect2程序是ld链接器的一个包装

1. **链接过程控制**

控制链接过程的方法：

* 使用命令行来给链接器知道参数
* 将链接指令存放在目标文件中
* 使用链接控制脚本

1. **进程的虚拟空间**

PAE(Pysical Address Extension):地址扩展方式，扩展物理线到36位。

操作系统提供一个窗口映射的方法，把额外的内存映射到进程地址空间中。

在windows下，这种访问内存的操作方式叫做AWE（Address Windowsing Extensions）

在Unix类操作系统中则采用mmap系统调用

1. **装载方式**

动态装入方式：覆盖装入和页映射

动态装入的原理：程序用到哪个模块就将哪个模块装入内存，如果不用就暂时不装入，放入磁盘中。

1. **覆盖装入**

程序员手动程序分割成若干块，将模块按照它们之间的调用依赖关系组织成树状结构，然后编写一个小的辅助代码来管理这些模块何时驻入内存。

* 模块调用时，整个调用路径上的模块必须在内存中
* 任何模块不允许跨树状结构进行调用

1. **页映射**

将内存和所有磁盘中的数据和指令按照页为单位划分成若干个页，然后所有的装载和操作单位都是页

**页面调度算法：**

* 先进先出算法（FIFO）
* 最少使用算法（LRU）
* 最近最不常用算法（LFU）
* 最佳算法（OPT）

1. **从操作系统的角度看可执行文件的装载**
2. **进程的建立**

* 创建一个独立的虚拟地址空间，即创建映射函数所需的相应数据结构
* 读取可执行文件头，并且建立虚拟空间与可执行文件的映射关系
* 将CPU的指令寄存器设置成可执行文件的入口地址，启动运行

1. **页错误**

* 执行完上面的步骤后，可执行文件的真正指令和数据还没有载入内存
* 所以当开始执行程序时，CPU会发现访问的是一个空页面，发生页错误，这时CPU将控制器交给操作系统；
* 操作系统查询映射关系，然后找到空页面所在的VMA，计算出相应页面在可执行文件中的偏移，
* 然后在物理内存中分配一个物理页面，将进程中该虚拟页与分配的物理页之间建立映射关系，然后把控制权返回给进程，
* 进程从刚才页错误的位置重新开始执行。

1. **减少映射时的内存浪费**

ELF文件被映射时，是以系统的页长度为单位，每个段在映射时的长度应该是系统页长度的整数倍，如果不是，多余部分也将占用一个页。所以为了避免这种浪费，对于相同权限的段，把他们合并在一起当做一个段进行映射。

1. **堆和栈**

在虚拟空间（VMA）上

* 堆，权限可读写，可执行；无映像文件，匿名，可向上扩展
* 栈，权限可读写，不可执行；无映像文件，匿名，可向下扩展

**堆的最大申请数量（malloc）：**

受操作系统、程序大小、程序栈数量、大小等影响，每次都不一样

**进程栈的初始化**

进程刚开始启动的时候，需要知道一些进程运行的环境，最基本的就是系统环境变量和运行参数。

操作系统会在程序启动前将系统环境变量和运行参数提前保持到进程的虚拟空间栈中。

栈顶寄存器esp执行的位置就是初始化之后堆栈的顶部。

程序启动后，程序的库部分会把堆栈里的初始化信息中的参数传递给main（）函数。

1. **动态链接**
2. **静态链接的缺点**

* 空间浪费
* 程序更新、部署、发布麻烦

1. **动态链接的思想**

将程序安装模块拆分成各个相对独立部分，在程序运行时，才将他们链接在一起形成一个完整的程序，而不是像静态链接一样把所有的程序模块都链接成一个单独的可执行文件。

Linux中，ELF动态链接文件被称为动态共享对象,”.so”

Windows中，动态链接文件被称为动态链接库，”.dll

程序与lib.so之间由动态链接器进行连接，而不是静态链接器ld。”

1. **内存**
2. **程序的内存分布**

* 内核空间
* 栈
* 动态链接库
* 堆
* 可执行文件映像（.text、.data、.rodata、.bss）
* 保留区

1. **栈与调用惯例**

栈保存了一个函数调用所需要维护的信息

* 函数的返回地址和参数
* 临时变量：函数的非静态局部变量以及编译器生产的其他临时变量
* 保存的上下文：函数调用之前需要保持不变的寄存器

I386中，一个函数活动记录用ebp和esp两个寄存器划定返回。Esp指向栈的顶部，ebp之前是函数返回地址，ebp直接指向的数据是调用函数前ebp的值，这样函数返回时，ebp可以通过读取这个值回复到调用前的值。

**调用惯例：**

* 函数参数的传递顺序和方式：

函数传递有多种方式，最常见的是栈传递

函数调用方将参数压入栈，函数自己从栈中取出

* 栈的维护方式

函数参数压入栈后，函数体会被调用，之后函数调用方需要将栈中参数全部弹出，使得栈的调用前后保持一致

* 名字修饰的策略

函数本身的名字的修饰

**函数返回值传递：**

函数与调用方的交互：参数+返回值

* 返回值小于等于4byte，函数将返回值存储在eax寄存器
* 返回5~8byte，低字节存储在eax，高字节存储在adx
* 返回值大于8byte，eax存储指针

1. **堆与内存管理**

**Malloc是怎么实现的？**

运行库向操作系统申请一个适当大小的堆空间，然后分配给程序用，当空间不够时，再向操作系统申请空间。运行库需要一个算法来管理堆空间。

**Linux进程堆管理**

Linux提供两个堆空间分配的方式，即两个系统调用

* Brk()：设置进程数据段的结束地址，即它可以扩大或缩小数据段（.data+.bss+.rodata）

Glibc有一个sbrk函数，功能与brk类似，实际上是对brk系统调用的包装

* Mmap()：向操作系统申请一段虚拟地址空间，当这段空间不将地址映射到某个文件时，即称为匿名空间，就可以拿来作为堆空间。

**Windows进程堆管理**

* VirtualAlloc()：类似于mmap()，但是申请的空间不限于堆，应用程序可随意使用。
* HeapCreate系列函数：HeapCreate+HeapAlloc+HeapFree+HeapDestroy

**堆分配算法**

* 空闲链表：把堆上各个空闲的块安装链表的方式连接起来，当用户请求一块空间时，遍历整个列表，直到找到合适大小的块并且将他拆分，当用户释放空间时将它合并到空闲链表中。
* 位图：将整个堆划分为大量的块，每个块的大小相同。当用户请求内存时，分配整数个块的空间给用户
* 对象池：如果每一次分配的空间大小都一样，就将其作为一个单元，把整个堆划分为大量的小块，每次请求的时候只需要找到一个小块就可以了。

1. **运行库**
2. **入口函数和程序初始化**

**入口函数：**

* 操作系统在创建进程后，把控制权交给程序的入口，这个入口往往是运行库的某个入口函数
* 入口函数对运行库和程序运行环境进行初始化，包括堆、IO、线程、全局变量构造等待
* 入口函数完成初始化之后，调用main函数，开始执行程序主体部分
* main函数执行完毕后，返回到入口函数，入口函数进行清理工作，包括全局变量析构，堆销毁，关闭IO等。然后进行系统调用结束进程。

PS：启动过程，静态和动态的情况不一样

**GLIBC入口函数：**

Glibc的程序入口为\_start,

\_start

程序正常结束有两种情况：

* main函数正常返回，exit被调用
* 程序中用exit退出

由此可见exit是程序结束的必经之路，因此可把调用atexit注册函数的任务交给exit。

**MSCV CRT入口函数：**

默认入口函数名为mainCRTStartup

* 初始化和OS版本有关的全局变量
* 初始化堆：\_heap\_init调用HeapCreate()
* 初始化IO;
* 获取命令行参数和环境变量
* 初始化C库的一些数据
* 调用main并记录返回值
* 检查错误并将main的返回值返回

**IO初始化的职责**

* C语言文件操作是通过一个FILE结构的指针来进行。
* 在操作系统层面上，文件操作也有FILE的概念，LINUX的是文件描述符，Windows的是句柄。
* IO初始化的职责：IO初始化函数需要在用户空间中建立stdin、stdout、stderr以及对应的FILE结构，使得程序进入main之后可以直接使用printf、scanf等函数。

1. **C/C++运行库**

C语言运行库包含：

* 启动与退出：包括入口函数及入口函数所依赖的其他函数
* 标准函数：由C语言标准规定的C语言标准库所拥有的函数实现
* IO：IO功能的封装和实现
* 堆：堆的封装和实现
* 语言实现
* 调试：实现调试功能的代码

**GLIBC与MSVC CRT**

* Linux上的C语言运行库位glibc；windows的是MSCV CRT；
* 运行库是平台相关的，与操作系统结合的非常紧密。它将不同操作系统的API抽象成相同的库函数。比如fread()，其实不同操作系统上fread的实现都是不同的。

虽然各个平台上的C语言运行库提供了很多功能，但是它们是有限的，比如用户权限控制、线程创建第二个都不属于标准的C语言运行库。

所以，我们需要绕过C语言程序库，直接调用系统API或其他库。

1. **运行库与多线程**

**线程的访问权限**

线程可以访问进程内存里的所有数据，甚至包括其他线程的堆栈（如果知道其他线程的堆栈地址的话）

线程拥有自己的私有存储空间

* 栈：并非完全不能被其他线程访问
* 线程局部存储（TLS）：线程局部存储是某些操作系统为线程单独提供的私有空间

可使用\_thread或\_declspec关键字定义全局变量为TLS变量

* 寄存器：寄存器存放的数据是执行流的基本数据，所以为线程私有

线程私有：局部变量、函数参数、TLS数据

线程公有：全局变量、堆上的数据、函数里的静态变量、程序代码、打开文件（A线程打开的文件可以由B线程读写）

1. **C++全局构造和析构**
2. **Fread实现**