~~（X）Linux 系统调用中的功能号 n 就是异常号 n。~~
~~（X）c 函数调用过程中，调用函数的栈帧一旦被修改，被调用函数则无法正确返回。~~
~~（X）采用连接时打桩需要提供被打桩的源程序。~~
**字符 0、A、a 的 ASCII 为 0x30、0x41、0x61。**
**C=S（组）*E（行）*B（字节/块）**
矩阵乘法的循环变量顺序 **ikj，kij**（优化）
**CPU 总是执行 CS::RIP 所指向的指令**
一条 mov 指令不可以使用两个内存操作数
冷不命中，冲突不命中，容量不命中。
缺页故障。
CPU 在执行异常处理程序时其模式为**内核模式**。
在父进程中，fork 返回新创建子进程的进程 ID；在子进程中，fork 返回 0；如果出现错误，fork 返回一个负值。
将 hello.c 编译生成汇编语言的命令行 **gcc -S hello.c**。
**48 为地址也是 64 位地址空间里的！所以计算的时候要用 8B。**
**Hello World 执行程序**很小不到 4k，在其首次执行产生缺页中断次数多于两次。**代码、数据、堆栈等**至少三个页面。
C 语言整数常量都存放在程序虚拟地址空间的**代码/数据段。不能使用.data 段，因为那个是 ELF 文件里面的。**
在 Linux 下，采用 UTF-8 编码，汉字用 **3（常用）或 4（不常用）**个字节编码。
计算机是 64 位是指 CPU 中通用寄存器是 64 位的。
**函数 mmap** 可以直接将进程的虚拟地址空间映射到磁盘文件。
缺页异常处理子程序完成虚拟内存到物理内存的映射。
条件跳转指令 **JC** 是依据 **CF** 做是否跳转的判断。
非 static 局部变量可能在**栈**中或**寄存器**中。
将可执行文件 hello 反汇编结果输出到文本文件 1.text 的命令是：**objdump hello -D > 1.txt**。
Linux:lscpu（查看 CPU）；free -m（查看 MEM）；ifconfig（查看 IP）；**jobs（查看作业）**。
在 Intel CPU 中，有专门保存当前正在运行进程的页表首地址的寄存器。
多个运行中的进程可以通过私有的写时复制方式来**共享物理内存中的共享库**，节省计算机的内存开销。
Linux 下.a 结尾**静态库**用 **AR** 命令或软件生成。
C 语言中若有符号整数和无符号整数混合运算，结果解读为**无符号类型**！若混合比较则解读为**无符号**比较！这是**编译器**作出的行为。
不管是非负数还是负数，算术右移的数学意义都是除以 2 的幂再**向下取整**左移同理。
**movabsq** 将一个 64 位立即数赋给 64 位寄存器（**movq 只能操作可表示为 32 位的立即数**）
**leaq 不修改标志位。**
函数调用要注意 3 个关键点：**传递控制，传递数据，内存的分配与释放**。
rbx、rbp、r12~r15 需要**被调用者保存**（如果被调用者用到它们的话），r10、r11 需要**调用者保存**（如果被用到的话）
**通过栈传递参数时，所有的数据大小都向 8 的倍数对齐**
**对一个非静态局部变量取地址时，要在栈上为其分配空间。**
**SRAM** 将每个位存储在一个**双稳态的存储单元**里，**DRAM** 将每个位存储为一个**电容的充电**。
从**指令系统**的角度来看，x86 属于 CISC 指令集，指令执行时的行为可以较为复杂，允许多次访存
**物理页表 4KB 对齐。**
1）gcc -c myproc1.c myproc2.c
2）gcc -shared -fPIC -o mylib.so myproc1.o myproc2.o

应用程序和操作系统属于软件，处理器、主存和 I/O 设备属于硬件。操作系统的两个基本功能：（1）防止硬件被失控的应用程序滥用；（2）向应用程序提供简单一致的机制来控制复杂而又通常大不相同的低级硬件设备。操作系统通过几个基本的**抽象概念**来实现这两个功能。文件是对 I/O 设备的抽象表示。虚拟内存是对主存和磁盘 I/O 设备的抽象表示，进程是对处理器、主存和 I/O 设备的抽象表示。

正常的函数调用一次**返回**一次，系统调用 execve 和 exit 在 调用成功后不返回，非系统调用函数 longjmp 从不返回，系统调用 fork 在 调用成功后返回两次，非系统调用函数 setjmp 会返回>=1 次。

**非本地跳转**和操作系统没关系（不是系统调用），纯粹是用户作出的单凭 C 语言语法无法做到的开挂式跳转行为，是用户级的异常控制流。setjmp 在第一次调用时 向某个 jmp_buf 中保存当前状态（当前代码地址、栈指针、寄存器等），之后在 某个时刻（必须是这个函数或者它嵌套调用的函数的代码，为什么？），程序通 过 longjmp 跳到那个 jmp_buf 指定的代码地址并恢复状态，表现为那个地方的 setjmp 返回了一个 longjmp 指定的参数。sigsetjmp 和 siglongjmp 还会多保存一个信号阻塞集合。

C 浮点常数 IEEE754 编码的默认舍入模式是**向偶数舍入**，它与四舍五入只有一点不同，对.5 的舍入上，采用取偶数的方式。IEEE754 还有向 0 舍入、向上舍入和向下舍入。

当在 **int，float 以及 double** 格式之间进行强制转换时，程序改变数值和位模式的原则如下（假设 int 为 32 位）：1.从 int 转换成 float，数字不会溢出，但可能被舍入。2.从 int 或 float 转换成 double，因为 double 有更大的范围（也就是可表示值得范围），也有更高得精度（即有效位数），所以能保留精确得数值。3.从 double 转换成 float，因为范围要小一些，所以值可能溢出为+∞或-∞。且由于精度较小，它还可能被舍入。4.从 float 或 double 转换成 int，值将会向 0 舍入。例如 1.999 将转换为 1。进一步说，值可能会溢出。

子进程即便运行结束，父进程也应该使用 **wait 或 waitpid** 对其进行回收。waitpid 很多情况下用在 SIGCHLD 的信号处理中，但要注意不要来一个信号就仅回收一次！

通用寄存器中，函数执行前后**必须保持原始的寄存器**有 3 个：是 rbx、rbp、rsp。rx 寄存器中，最后 4 个必须保持原值：r12、r13、r14、r15。保持原值的意义是为了让当前函数有可信任的寄存器，减小在函数调用过程中的保存&恢复操作。除了 rbp、rsp 用于特定用途外，其余 5 个寄存器可随意使用。通用寄存器中，**不必假设保存值可随意使用的寄存器**有 5 个：是 rax、rcx、rdx、rdi、rsi。其中 rax 用于第一个返回寄存器（当然也可以用于其它用途），rdx 用于第二个返回寄存器（在调用函数时也

用于第三个参数寄存器)。当参数少于 7 个时， 参数从左到右放入寄存器: rdi, rsi, rdx, rcx, r8, r9。当参数为 7 个以上时， 前 6 个与前面一样， 但后面的依次从 “右向左” 放入栈中。

Linux 系统中运行 hello world 这样的 C 程序时，标准 I/O 函数 printf 实际是通过系统级 Unix I/O 函数 **write** 实现的。相比标准 I/O，Unix I/O 函数是**异步信号安全**的，可以在信号处理程序中安全地使用。

对于 **Intel Core i7**，虚拟地址 48 位，物理地址 52 位，物理页和虚拟页大小(它们始终相等)在 Linux 下被设置为 4KB（页偏移 12 位），采取四级页表
PPO，高 40 位为页号 PPN，虚拟地址的低，物理地址的低 12 位为偏移 12 位为偏移 VPO，高 36 位为页号 VPN，VPN 会被均等分割为 4 个 9 位字段 VPN1~4，表示每一级页表里指向下一级 PTE 的索引（所以每级页表里有 2^9=512 个 PTE）。CPU 中的控制寄存器 CR3 保存页表的物理基址。

一个进程对共享对象的修改对其他把这个共享对象映射到虚拟内存的进程可见，而且这些变化也会反映在磁盘上的原始对象中；对于一个映射到私有对象的区域做的改变，对于其他进程来说是不可见的，并且进程对这个区域所做的任何写操作都不会反映在磁盘上的对象中。私有对象使用**写时复制**技术。

**程序语句 execve：** Linux 程序通过 execve 函数来调用加载器，加载器将可执行目标文件中的代码和数据从磁盘复制到内存中，然后通过跳转到程序的第一条指令或入口点来运行该程序。这个将程序复制到内存并运行的过程叫做加载。进程通过 execve 系统调用启动加载器，加载器删除子进程现有的虚拟内存段，并创建一组新的代码、数据、堆和栈段。新的栈和新的堆段被初始化为零。新的代码和数据段（.init,.text,.rodata,.data,.bss）被初始化为可执行文件的内容。最后，加载器跳转到_start 地址，他最终会调用应用程序的 main 函数。除了一些头部信息，在加载过程中没有任何从磁盘到内存的数据复制。直到 CPU 引用一个被映射的虚拟页时才会进行复制，此时操作系统利用它的页面调度机制自动将页面从磁盘传送到内存。

**UNIX I/O 的 read、write 不足值问题：**(1)read 函数从描述符为 fd 的当前文件位置复制最多 n 个字节到内存位置 buf。返回值-1 表示一个错误，返回值 0 表示 EOF，否则返回值表示的是实际传送的字节数量。(2)write 函数从内存位置 buf 复制最多 n 个字节到描述符 fd 的当前文件位置。
在某些情况下，read 和 write 传送的字节比应用程序要求的要少，这些不足值不表示有错误：(1)读时遇到 EOF。假设我们准备读一个文件，该文件从当前文件位置开始只含有 20 个字节，而我们以 50 个字节的片进行读取。这样一来，下一个 read 返回的不足值是 20，而此后的 read 将返回不足值 0 来发出 EOF 信号；(2)从终端读文本行。如果打开文件是与终端相关联的，那么每个 read 函数将一次传送一个文本行，返回的不足值 0 来等于文本行的大小；(3)读和写网络套接字(4)除了 EOF,当你读磁盘文件时，将不会遇到不足值；写磁盘文件时也不会遇到不足值。

**I/O 重定向：**dup2 函数：函数原型 int dup2(int oldfd, int newfd)；dup2 函数复制描述符表项 oldfd 到描述符表项 newfd，覆盖描述符表项 newfd 之前的内容，如果 newfd 打开了，dup2 会在复制之前关闭 newfd（会删除其文件表和 v-node 表）。

立即寻址在取指令阶段，访问一次内存；在执行阶段，不需要访问内存（一次）；直接寻址在取指令阶段，访问一次内存；在执行阶段，访问一次内存（两次）；间接寻址在取指令阶段，访问一次内存；在执行阶段，访问两次内存（三次）；寄存器直接寻址在取指令阶段，访问一次内存；在执行阶段，不需要访问内存（一次）；寄存器间接寻址在取指令阶段，访问一次内存；在执行阶段，访问一内存（两次）。

代码放在.text 段；已初始化的数据（.data）；未初始化的数据（.bss）；运行时堆（malloc 分配）；栈（局部变量）。

**发送信号：**setpid；kill 非负发送给进程，负数发送给进程组的每个进程；从键盘发送信号（只发送给前台作业）；kill 函数，若 pid 为 0 发送给自己进程组的每个进程（包括自己），若为其他值发送给另一个进程组；alarm 函数。

不同的进程是并发的，**信号处理程序和主程序也是并发的**（主要由于其异步性），如果它们共享相同的资源，容易造成数据竞争，需要小心处理。特别是信号处理程序，它与主程序共享相同的全局变量。由于信号处理程序的并发性，需要通过**信号阻塞**等方式避免与信号处理程序产生数据竞争，并且在信号处理程序中使用**可重入的**（不会使用全局数据的）**异步信号安全的**函数（Unix I/O-系统提供的 read/write，而非标准库的 scanf/printf）。

**简述缓冲区溢出攻击原理以及防范方法：**向程序输入缓冲区写入特定的数据，例如在 gets 读入字符串时，使位于栈中的缓冲区数据溢出，用特定的内容覆盖栈中的内容，例如函数返回地址等，使得程序在读入字符串，结束函数 gets 从栈中读取返回地址时，错误地返回到特定的位置，执行特定的代码，达到攻击的目的。
防范方法：（1）代码中避免溢出漏洞；（2）随机栈偏移；（3）限制可执行代码的区域（4）进行栈破坏检查——金丝雀

**简述 shell 的主要原理与过程：**Linux 系统中，shell 是一个交互型应用级程序，代表用户运行其他程序（是命令行解释器，以用户态方式运行的终端进程）终端进程读取用户由键盘输入的命令行；分析命令行字符串，获取命令行参数，并构造传递给 execve 的 argv 变量；检查第一个命令行参数是否是一个内置的 shell 命令；如果不是内部命令，调用 fork()创建新进程/子进程；在子进程中，分析命令行字符串，调用 execve()执行指定程序；如果用户没要求后台运行（命令末尾没有&号）否则 shell 使用 waitpid（或 wait）等待作业终止后返回；如果用户要求后台运行（末尾有&号），则 shell 返回

**判断两个浮点数是否相等：**由于浮点数的 IEEE 754 编码表示存在精度、舍入、溢出、类型不匹配问题，两个浮点数不能够直接比较大小，应计算两个浮点数差的绝对值，当绝对值的小于某个可以接受的数值时认为相等。

**什么是共享库？简述动态链接的实现方法。**（a）共享库是一个.so 的目标模块（elf 文件），在运行或加载时，由动态链接器程序加载到任意的内存地址，并和一个和内存中的程序动态完全链接为一个可执行程序，使用它可以节省内存与硬盘空间，方便软件的更新升级。（b）加载时动态链接：应用第一次加载和运行时，通过 ld-linux.so 动态链接器重定位动态库的代码和数据到某个内存段，再重定位当前应用程序对共享库定义的符号的引用，然后将控制传递给应用程序（此后共享库位置固定不变）。（c）运行时动态链接：在程序执行过程中，通过 dlopen/dlssym 函数加载和链接共享库，实现符号重定位，通过 dlclose 卸载动态库。

**简述程序局部性原理，如何编写局部性好的程序？**局部性原理：程序倾向于使用与最近使用过数据的地址接近或者是相同的数据和指令。时间局部性：最近引用的项很可能在不久的将来再次被引用，如代码和变量等；空间局部性：与被引用项相邻的项有可能在不久的将来再次被引用；让通用或共享的功能或函数——最常见的情况运行的块（专注在核心函数和内循环）尽量减少每个循环内部的缓存不命中数量：反复引用变量是好的（时间局部性和空间局部性）——寄存器-编译器；参考模式是好的（空间局部性）——缓存时连续块。一旦从内存中读入数据对象，尽可能多的使用它，使得程序中时间局部性最大。

**在终端的命令行运行显示"HELLO WORLD"的执行程序 hello，结合进程创建、加载、缺页中断、到存储访问（虚存）等等，论述 hello 是怎么一步步执行的。**shell 接受命令；用 fork 创建子进程；execve 函数加载进程；执行时产生缺页异常/中断；利用 VA 访存；缺页中断后页面换入，并恢复上下文；printf 函数设计的动态链接库的动态链接；调用 printf 函数设计的 hello world 字符串的获取；hello 运行完毕后产生 SIGCHLD 的信号；父进程对其回收、资源释放。

**结合 fork，execve 函数，简述在 shell 中加载和运行 hello 程序的过程。**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。

**重定位步骤：**（1）合并相同的节；（2）对定义符号进行重定位；（3）对引用符号进行重定位（需要用到在.rel_data 和.rel_text 节中保存的重定位信息）。

**针对有符号及无符号整数的加法运算，CPU、编译器、程序员是怎么配合完成不同类型整数的数据表示、数据运算，并如何判断其结果是否超出范围的？**C 程序员用 unsigned 或 signed（缺省，可不用）来区分数据类型，常数后加 U 表示无符号数。程序中可以自由进行比较、赋值、运算等。CPU 并不知道数据类型，只是按位进行加法操作，并按照逻辑规定设置 CF、OF、ZF、SF、PF、AF 等标志位。编译器会将数据转换成相应的二进制编码（无符号数）或补码（有符号数），如果类型不一致，都转换成无符号数再进行操作。数据操作之后，编译器根据不同数据类型选择不同的分支转移指令，可按照如上标志位，无符号数用 JA/JB 等、有符号数用 JG/JL 等判断数据大小，并进行跳转。无符号溢出用 JC、有符号溢出用 JO 判断。

**Unix I/O 函数与 C 标准 I/O 函数能否混合使用？为什么？并说明各自的适用范围。**不能。因为 UnixI/O 函数是不带缓冲的，C 标准 I/O 函数是带缓冲的。如混合使用会
导致数据输出顺序与程序中发送顺序不一致的情况，从而出错。Unix I/O 函数是异步信号安全的函数，可用于信号处理程序中，并适用于一些实时性要求高（高性能）的 I/O 应用场合。C 标准 I/O 函数带缓冲，能减少对 I/O 设备的访问次数，大大提高 I/O 的效率，如磁盘文件和终端文件等。

**以 Intel64 位现代处理器为例，简述加快页表 PTE 访问、大大降低页表占用空间的相关技术。**采用 TLB 加快页表 PTE 的访问：采用高速缓冲存储器作为页表的 Cache。TLB 中保存最近常用的虚拟页号对应的页表条目 PTE（含物理页号）。对于虚拟页数较少的进程，它的页表可以完全放在 TLB 中。
采用多级页表大大降低页表占用的空间：由于 VM 空间的页面有大量的连续页面都是未分配的，Intel64 位 CPU 采用 4 级页表后，一级页表的大量条目其内容为 NULL，他们对应的二、三、四级页表项就不用存储。相应的，二、三级页表也是如此，会有值为 NULL 的 PTE，他们的子页表也都不用存储。，这样只有已分配页表条目对应的 4、3、2、1 级页表项才需要存储，大大节省了页表空间。

**程序执行 int x=y/c 语句时，当 c=0 时程序执行结果是什么？并请结合异常、信号的概念及处理机制解释原因。**异常是指为响应某个事件将控制权转移到操作系统内核中的情况，每种异常都有一个异常号及对应的异常处理子程序（内核态）信号是一条消息，它通知进程系统中发生了一个某种类型的事件，在进程进入运行态前由 OS 内核检查信号并执行其对应的信号处理子程序（用户态）。程序执行到这条语句时会产生整数除法出错异常（异常号 0），执行 Divide_Eror( )异常处理子程序，在异常处理子程序中，向当前进程发送一个 SIGFPE 信号（8 号信号），而 SIGFPE（信号处理子程序）的默认行为就是显示“Floating point exception (core dumped)”，终止并转储内存。

**简述链接器在将目标文件链接生成可执行文件时，如何处理符号引用（符号解析）的。**1）根据强弱符号类别，确定被引用的符号在哪里；2）根据各目标文件的段落大小、系统代码等信息，合并同类型的段，并计算可执行文件各段的大小；3）根据程序的内存映像，确定被引用的符号在程序运行时的内存地址；4）根据在磁盘上的可执行文件中，各段落的大小，确定符号引用的位置；5）根据上述信息，计算符号引用处应该采用的数值，并写入文件。

**组合逻辑与时序逻辑：**组合逻辑包括 ALU、控制逻辑、读(内存、数据/指令内存、寄存器文件)时序逻辑包括写寄存器、写内存。组合逻辑的状态不用等待时钟上升沿触，就能快速更新；时序逻辑，例如寄存器，在数据更新/写入时，需要时钟上升沿触发，

才能将更新的数值写入。

**固态硬盘的存储容量不建议用尽，主要有以下几个原因：**1. 写入速度下降：固态硬盘使用闪存记录数据，如果固态硬盘用的太满，会影响写入速度。因为闪存需要先擦除然后才能重新写入，擦除的速度比写入要慢很多。当固态硬盘用的越满，留给主控擦除腾挪的难度越大，写入速度就会下降。2. 影响寿命：固态硬盘用的太满后，写入放大率会大幅提升，这意味着对闪存的磨损加剧，从而影响固态硬盘的寿命。

**逻辑地址：**机器语言指令中用来指定一个操作数或一条指令的地址。每一个逻辑地址都由一个段和偏移量组成，偏移量指明了从段开始的地方到实际地址之间的距离。例如，c 语言取指针的操作（&），这个值是逻辑地址，它是相对于你当前进程数据段的地址，不是物理地址。逻辑地址由两个地址分量构成，一个为段基值，另一个为偏移量。两个分量均为无符号数编码。**线性地址：**也称虚拟地址，是逻辑地址和物理地址之间的中间层，在分段部件中，逻辑地址就是段中的偏移地址，加上基地址就是线性地址。

**物理地址：**用于内存芯片级内存单元寻址，它们从微处理器的地址引脚发送到内存总线上的电信号相对应。

内存控制单元 MMU 通过分段单元的硬件电路把一个逻辑地址转换成线性地址，接着经过分页单元把线性地址转换成物理地址 。

**简述浮点数在数轴上的分布特点，按步骤分析计算 float 数 -1.2 在内存各字节内容。**浮点数在数轴上成对称分布，越靠近原点 0 则分布的密度越高，越远离原点 0 则分步越稀疏。

float：32 bits，1 个符号位，8 位指数（127 移码），23 位尾数（先导为 1 的规格化） (1)转换成二进制： -1.00110011[0011]…2； (2)科学记数法，先导为 1: -1.00110011[0011] … E 0；(3)指数的 127 移码： 0+127=127 其二进制形式为 01111111；(4)尾数 23 位，向偶数舍入：0011 0011 0011 0011 0011 0011 00… 0011 0011 0011 0011 0011 010；(5)IEEE754 编码： 1 0111 1111 0011 0011 0011 0011 0011 010，其 16 进制 BF 99 99 9A；(6)内存小端存储：9A 99 99 BF。

**longjmp 的执行：**从缓冲区 j 中恢复寄存器内容（栈指针、基址指针、程序计数器）；返回值 i 在**%eax** 中；跳转至保存在缓冲区 j 中的 PC 所指示的位置。

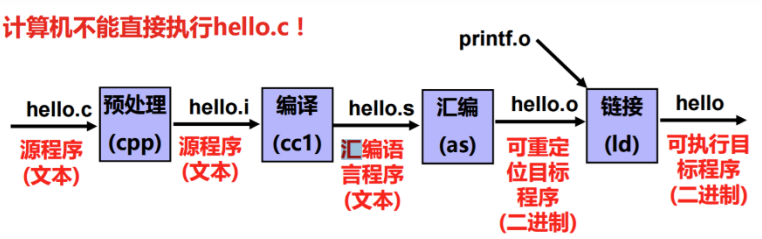
**什么是静态库？使用静态库的优点是什么？**编译时将所有相关的目标模块打包成为一个单独的文件用作链接器的输入，这个文件称为静态库。优点：使用时，通过把相关函数编译为独立模块，然后封装成一个单独的静态库文件，应用程序可以通过在命令行上指定单独的文件名来使用这些在库中定义的函数，链接器只需复制被程序引用的目标模块，减少了可执行文件在磁盘和内存中的大小；同时，应用程序员只需包含较少的库文件的名字。

**信号处理程序的编写原则：**1） 处理程序尽可能简单；2） 在处理程序中只调用异步信号安全的函数；3） 确保其他处理程序不会覆盖当前的 errno；4） 阻塞所有信号保护对共享全局数据结构的访问；5） 用 volatile 声明全局变量；6） 用 sig_atomic_t 声明标志。

| 描 述 | exp | frac | 单精度 | | 双精度 | |
|---|---|---|---|---|---|---|
| | | | 值 | 十进制 | 值 | 十进制 |
| 0 | 00…00 | 0…00 | 0 | 0.0 | 0 | 0.0 |
| 最小非规格化数 | 00…00 | 0…01 | $2^{-23}\times2^{-126}$ | $1.4\times10^{-45}$ | $2^{-52}\times2^{-1022}$ | $4.9\times10^{-324}$ |
| 最大非规格化数 | 00…00 | 1…11 | $(1-\varepsilon)\times2^{-126}$ | $1.2\times10^{-38}$ | $(1-\varepsilon)\times2^{-1022}$ | $2.2\times10^{-308}$ |
| 最小规格化数 | 00…01 | 0…00 | $1\times2^{-126}$ | $1.2\times10^{-38}$ | $1\times2^{-1022}$ | $2.2\times10^{-308}$ |
| 1 | 01…11 | 0…00 | $1\times2^{0}$ | 1.0 | $1\times2^{0}$ | 1.0 |
| 最大规格化数 | 11…10 | 1…11 | $(2-\varepsilon)\times2^{127}$ | $3.4\times10^{38}$ | $(2-\varepsilon)\times2^{1023}$ | $1.8\times10^{308}$ |

图 2-36 非负浮点数的示例

2^-23*2^-126；1.4*10^-45；2^-52*2^-1022；4.9*10^-324；(1-ep)*2^-126；1.2*10^-38；(1-ep)*2^-1022；2.2*10^-308；1*2^-126；1.2*10^-38；1*2^-1022；2.2*10^-308；(2-ep)*2^127；3.4*10^38；(2-ep)*2^1023；1.8*10^308；




**gcc -o hello hello.c**

-o：指定生成的输出文件；-E：仅执行编译预处理（.i）；-S：将 C 代码转换为汇编代码（.s）；-c：仅执行编译操作，不进行链接操作（.o）；-wall：显示警告信息；-w：不输出任何警告信息。多个-O 参数默认最后一个。

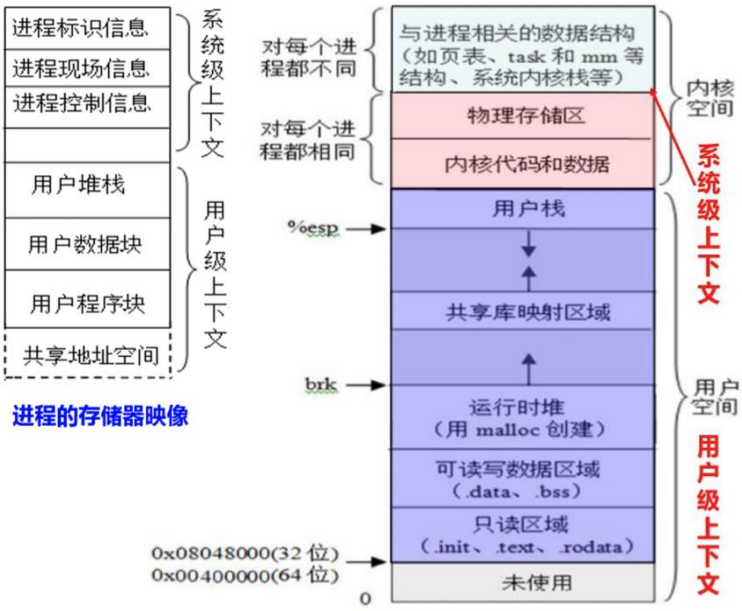
**对齐数据的动机：**内存按 4 字节或 8 字节（对齐的）块来访问（4/8 依赖于系统）：不能高效地装载或存储跨越四字边界的数据；当一个数据跨越 2 个页面时，虚拟内存比较棘手。

**编译器优化的局限性：**(1)难以克服优化障碍（潜在的内存别名使用，潜在的函数副作用）；(2)在基本约束条件下运行；(3)大多数分析只在过程中执行；(3)大多数分析都是基于静态信息的；(4)当有疑问时，编译器必须是保守的

**为什么使用链接器？**1.模块化；2. 效率：时间：分开编译（更改一个源文件不需要重新编译其他源文件）；空间：库（可以将公共函数聚合为单个文件，且可执行文件和运行内存映像只包含它们实际使用的函数的代码）

**低层机制（硬件层）**1.异常（操作系统和硬件共同实现）。

**高层机制（操作系统、应用）**1. 进程切换（操作系统和硬件计时器实现）；2.信号（操作系统实现）；3.非本地跳转（C 运行库实现）。




| ID | 名称 | 默认行为 | 相应事件 |
|---|---|---|---|
| 2 | SIGINT | 终止 | 来自键盘的中断 |
| 9 | SIGKILL | 终止 | 杀死程序(该信号不能被捕获不能被忽略) |
| 11 | SIGSEGV | 终止 | 无效的内存引用（段故障） |
| 14 | SIGALRM | 终止 | 来自alarm函数的定时器信号 |
| 17 | SIGCHLD | 忽略 | 一个子进程停止或者终止 |