**2018.11.1**

1.学习了一些linux库，主要有sys/types.h、unistd.h、sched.h，了解了pid\_t、fork()、vfork()、wait()、waitpid()、execl()、execlp()、execle()、execv()、execvp()、execve()、exit()、\_exit()、getpriority()、setpriority()、getpid()、getpgrp()、setpgrp()以及僵尸进程，孤儿进程，exit()和\_exit()函数的区别，exit()和return的区别，二维数组在栈中的存储，静态变量如何访问，还有操作系统在main函数执行前做的一系列工作。

**Tips:**

1. wait函数默认是等待所有子进程执行完毕，而waitpid可以根据pid参数的不同，选择不同的等待条件。
2. 子进程完成后会成为僵尸进程，它会释放部分资源，但是仍会保留程序控制块，只有当父进程调用wait相关的函数后，才会真正的释放掉这个进程，若父进程在调用wait之前终止了，那么它的子孙进程将会成为孤儿进程，最终其父进程会变为init，即交由init进程管理，init进程会循环监督子进程是否完成，若完成则释放其PCB。僵尸进程过多会导致OS可创建进程的空间减少；孤儿进程最终将交由init进程释放，因此一般对OS是无害的。
3. fork创建的子进程与父进程共享代码，fork函数会返回两次，在子进程中返回0，在父进程中返回子进程PID，因此可以在父进程中利用if else语句来控制子进程。创建子进程后，执行顺序由操作系统调度算法决定，但可以通过setpriority函数来决定哪个进程优先执行(实验了是这样)。
4. 用vfork函数创建的子进程与父进程共享资源，由于子进程不是因父进程调用而执行，共享的栈中也不会存在其返回地址，因此vfork创建的子进程不能使用return语句返回，而只能使用\_exit函数终止。exit函数相比\_exit函数多了释放IO等资源的操作，由于子进程与父进程共享资源，若子进程调用exit函数退出，则会严重影响父进程对IO资源的使用，因此利用vfork创建的子进程必须调用\_exit来终止。
5. exit属于系统调用，return属于语言级别的关键字，exit是结束进程返回对象是操作系统，而return只是一系列弹出操作，最终返回给函数调用者，对于main函数，在return后会伴随着exit。
6. 成功调用exec类函数不会继续父进程代码，因此可以借助fork函数，将exec类函数写入子进程函数体，这样子进程就不会执行父进程的代码了(书上说是不会复制代码段了，但是我觉得不对，虽然目前无法考证)，exec类函数能传入调用参数与环境变量，因此可以借助其实现与bash命令等价的功能。
7. 目前了解到操作系统在执行main函数前会先执行\_start和\_startmain(具体名字忘了)函数，第一个函数主要是初始化栈(将EBP置0，压入ESP地址以作为最大栈地址等)，然后将命令行参数与环境变量参数的首地址压栈，再压入init函数，finit函数与，动态链接器相关函数与main函数地址，最后压入最大栈地址。在startmain中则是初始化堆，文件句柄数组等(其他还不知道)，在startmain的最后会调用main，获取其返回值，并将其作为参数调用exit函数，在startmain和exit的最后都有一个HLT停机指令，在执行HLT之前会有INT $0x80，这将导致正常情况下HLT永远不会执行，因此HLT主要是为预防突发情况而附加的指令，这样即使exit失败了，仍有HLT可以挽救。
8. 文件句柄这块目前不是太了解，书中介绍是文件句柄数组在内核区，因此用户只能通过FILE指针做相应变换访问。数组大小一般为64\*32，因此FILE结构体中的\_file参数的0-4位用于确定列，5-10位用于确定行。该数组一般采用指向一维数组的指针存储，这样可以根据需要扩展，而不必在一开始就浪费2048个空间。
9. 最后是一个有趣的问题。实验可证明C++的throw语句可以跳出深度递归，而且在递归语句后的语句都不会执行，那么跳出后如何恢复栈为调用前的状态呢？try catch throw又是怎么通过汇编实现的(应该可以通过绝对地址转移来实现)？对于GCC编译器，也许可以采用保留调用前的%RBP，然后反复MOVL %RSP, %RBP; pop %RBP;来恢复栈，但对于CL这种没有使用%RBP，而是采用每次通过加减一定大小的%RSP来恢复栈的编译器，是如何解决这个问题的？

**2018.11.2**

1.复习了函数指针，指向一维数组的指针，以及算符优先级，词法分析等知识。

**Question:**

1.main返回值为一个结构体会发生什么情况？

2.书中scanf和printf异常的例子？

3.外部变量定义与声明的类型不一致，是统一改为最大长度类型，还是按原类型访存？

4.指向一维数组的指针的内容和对该指针取内容后的值相等（预测相等）？

**Result:**

第一问：运行时无任何异常。

第二问：整型变量的值会被修改，与书中描述基本一致。

第三问：按原类型访问，保存的符号类型是最大的那个类型。

第四问：相等。

**Tips:**

1. 一个文件不能同时被读写，读写切换中间需要插入fseek函数以改变文件状态。
2. getchar函数返回整型值，若用char类型变量接收，可能导致超过一个字节的字符被截断，对于while((c=getchar()) !=EOF)这样的语句就可能导致死循环或异常退出。另外，getchar在stdio.h中被实现为宏，若不包含stdio.h，可能会导致因多次调用getchar函数使进程运行速度下降。
3. 利用setjmp与longjmp函数可以实现try catch机制，应该也可以用于跳出深度递归（也许goto也可以，但是goto会导致栈异常）。
4. 对于switch得case中一系列相对连续的值，编译器会采用地址表的方法，将输入的数据经过一定变换，计算出其跳转位置，对于那些不在case中的点，设置其跳转地址与default相同即可，而对于范围较大，比较离散的case来说，则会生成一系列cmp，jmp式代码，效率应与if else语句块相近。
5. 子进程可继承父进程文件句柄，c可变参数[va\_list, va\_start(), va\_arg(), va\_end()]。
6. 非局部跳转不会导致栈混乱[jmp\_buf, setjmp(), longjmp()]。
7. 子进程私有资源(TLS机制，gcc中可以使用\_\_thread关键字来声明)管理[.tls段，每个线程拥有一个.TLS段的副本，由堆分配，其地址位于TEB结构体中，书中介绍偏移为0x2c]，CRT库的发展。

**2018.11.3**

1.认识了fread的实现原理，学习了一些系统调用以及运行库的由来。

2.复习了整数编码（原码，反码，补码），有了更进一步（语言级，原理级）的认识。

3.看了一个数组复制的例子，了解了register在生成高质量汇编代码上的作用，认识到-O9编译优化参数的强大。

**Tips:**

1. a[i]=b[i]劣于\*pa++=\*pb++（前者每次通过乘法运算获得元素，后者通过取内容与每次移动固定字节数获得元素），可以通过\*pa=\*pb;pa++;pb++;对\*pa++=\*pb++优化（前者先赋值，后增加，后者先保存原值，再自增，最后赋值）。
2. a[i] == i[a]，因编译器将其认作\*(a+i)（但是是如何通过词法、语法分析的呢？可能是ident[ident]？但是可以在语义分析时明确[]内是number还是ident）。
3. fread函数主要实现了缓冲机制。它利用FILE结构体中的buff相关的成员，一次性调用read函数从文件句柄获取较多资源（一般为OS文件系统页大小，不过在用户一次性读入量较大时会尝试扩展）。另一个较为棘手的问题是windows下会将"\r\n"视为"\n"，因此在每次读取到'\r'时，都要再读一个字符，若此时刚好是buff的末尾，只能再利用read函数从文件中读，若是'\n'则皆大欢喜，否则文件指针就要回退，而对于管道文件无法回退指针的情况，需要将这个字符暂存。
4. 系统调用形式复杂，且各个OS不同，于是利用加层的思想，引入运行库，让c程序员按照运行库提供的标准编码，使得代码可移植性更好，也在一定程度上降低了编程难度。
5. 二进制首位按负权计算即可得到二进制串的补码值。
6. 无符号整数与有符号整数运算时，有符号整数会转为无符号整数。转换原则：二进制位不变，解析方式改变（首部权重取负）。

**2018.11.4**

1.复习了二进制位运算，逻辑运算等。

2.了解了C++全局构造和全局析构，利用.CTOR和.DTOR两个段控制函数在main运行前后执行，并学习了其原理和在MSVC下的实现方式。

3.进一步学习了计算机启动过程，了解到地址线并不全用于内存（DRAM），还用于其他硬件设备，这也是实际可用内存小于理论内存的原因。

4.复习了寻址方式，call地址计算，栈，大端小端等知识。

5.学习了共享内存，信号量等IPC进程通信机制，了解了sys/ipc.h sys/sem.h sys/shm.h key\_t等头文件与结构，具体函数运用还不太熟练，书上讲的很模糊。大致是先使用getshm/getsem函数获取或创建共享内存/信号量，然后对于信号量，还要设定其可用资源的数量，随后通过shmid/semid，调用相应函数访问共享内存/信号量，最后调用相应函数销毁。

**Tips:**

1. 两个相同的数传入利用异或交换的函数，会置零，且使用异或实现swap产生的汇编代码质量劣于用临时变量产生的，实际上临时变量产生的汇编代码并不会额外使用内存。
2. 整型值有大小端问题，而字符串没有，因此字符串在数据传输中更通用。同时，不同设备，不同协议（尤其是数据传输协议）都有其规定的字节序，在其中进行数据传输时要注意数据格式转换。
3. 链接器会按顺序合并各个目标文件的.CTOR和.DTOR段，运行时CRT库函数通过这两个段获取函数指针后依次调用，从而实现了全局构造和全局析构，因此只要显式将函数指针声明到相应的段，便可实现在main函数前后执行。全局析构较为特殊，因为它的调用顺序要和全局构造相反，为了不影响链接器的工作效率，编译器在编译构造函数时，会在内部插入利用atexit函数注册其析构函数的代码（好像是在构造函数内部插入，有点忘了，但是确实是通过atexit实现），因atexit先注册后调用（这点与堆栈相似）的性质以达到按构造的反次序析构的目的。
4. fread是对ReadFile系统API的封装，预测fread通过一次性读取n页（大量数据）至缓冲区的工作，待调用时直接从缓冲区返回以提高效率。
5. 加电->BIOS->MBR->...（还没探索到）中机器运行在实模式状态，目前学习到实模式状态下仅可访问1M内存（我是不太信的。。）。实模式下可直接访问物理内存，就像8086那样，采用段地址\*16+偏移地址的方式访问。
6. 机器加电后会强制修改CS和IP寄存器，让它们指向能跳转到BIOS入口地址的指令，随后BIOS进行硬件检测，中断向量表构建等工作后，扫描硬盘的第一扇区（就是第零个，只不过从一开始编号），找到510字节后以0x55aa结尾的部分，也就是MBR，将其载入0x7C00，随后CS置0，IP置7C00，跳转到MBR的第一条指令，开始执行MBR。
7. ASM文件中的SECTION，vstart等都是伪指令，即OS无法识别，但可由支持它们的汇编器识别转换的指令。其中vstart告诉汇编器从vstart指定的值开始对标号、变量编址（这一点和ORG伪指令类似）。在MBR中会讲vstart置为0x7C00，因其会被载入内存中该地址处，这样经过汇编，引用标号或变量的指令，其相应位置的符号会变为0x7C00+OFFSET从而使得MBR能正确访问这些数据。
8. 对于CPU来说，一切数据皆指令（个人理解），若不控制CPU的IP寄存器，待其执行完进程，IP自增到数据部分时，会发生难以预料的错误。
9. CPU通过操作码即可确定指令长度，IP通过当前值加上当前指令长度得到下一条指令地址。
10. 一些寄存器只能用作基址寻址的基址寄存器部分，主要是硬件上做了限制。
11. 纯二进制文件没有文件头、段表、符号表等辅助信息，是CPU可以直接执行的指令序列。ELF、PE等可执行文件则需要通过一个加载器才能为CPU所执行（也就是创建进程的那一套吧）。
12. 部分寄存器对程序员可见，如AX、BX、CX等，部分不可见，如IP、FLAG等，但是那些程序员不可见的寄存器也需要程序员编写指令去做初始化或修改，如利用PUSHF和POPF可以修改FLAG，而CALL、RET、RETF可以修改IP寄存器。

**2018.11.6-2018.11.7**

1.复习了浮点数（完全没看懂），汇编，了解了64位汇编的新指令，以及编译器将if-else，while，do-while，for，switch语句块转汇编的一般写法，对驱动程序，IO接口，BOCHS调试，硬盘控制有了初步了解。

**Tips:**

1. 硬盘与CPU之间隔着硬盘驱动器，控制器，CPU主要和驱动器交互数据。通过柱面-磁头-扇区可定位磁盘，开启逻辑块寻址后，CPU可直接利用逻辑块号访问硬盘资源。
2. 浮点数和整数在许多运算下有相同的位行为（姑且先记住吧，乘法和除法一定是不同的，这一点从MUL【无符号数乘法】，IMUL[【有符号数乘法】与相应的两个除法指令即可看出，其他指令应该无区别）。

**2018.11.8**

1.看书完善了MBR代码，初探利用指令访问硬盘。

2.复习了数组，结构体，联合体的内存分布，并重温数据对齐，过程调用。

**Tips:**

1. c99引入了变长数组，但是变量需要在引用前被定义，变长数组在寻找元素时只能利用乘法计算位置，因为数组大小在运行时确定。
2. 数据对齐是为CPU服务，假设CPU一次读取八个字节内存，那按八字节对齐可以在访问变量时只访存一次（变量小于八字节）。
3. 伪指令.align n可以保证其后的符号地址是n的倍数，即按n字节对齐。

**2018.11.9**

1.看书实现了loader，对MBR运行机制，vstart伪指令有了进一步了解。

2.进军保护模式，kernel。

**2018.11.10**

1.学习了缓冲区溢出的危害与预防措施以及浮点数的相关指令。

2.初探实模式与保护模式。

**Tips:**

1. 缓冲区溢出会破坏堆栈，可能造成对帧指针和返回地址的修改，出现返回异常。有些人利用栈溢出漏洞，注入恶意代码，随后利用将返回地址修改为恶意代码的入口，使得插入的代码得以运行，以此来入侵计算机。
2. 由于上述攻击方式需要明确栈地址，因此编译器和操作系统做了一系列防护措施：  
   一是栈随机化，即程序装入后，使栈以下的n个数字节不可用，使实际栈顶地址在每次运行时无法确定，但是这种方式仍然可以通过遍历破解（不知道怎么做到的）。对于Linux操作系统来说，还会使用地址空间布局随机化技术，在程序装入时，将各个段的首地址随机化。  
   二是栈破坏检测。通过在栈帧和局部变量顶部之间的栈空间中插入金丝雀值（一个位于.RODATA段随机的值，书中取fs:40），在过程调用结束前先检测金丝雀值是否改变以判断是否发生缓冲区溢出。  
   三是限制可执行代码区域。顾名思义，即赋予各个段可读，可写，可执行的权利。通过这个技术将段空间设置为不可执行以预防缓冲区溢出攻击。
3. 浮点数指令有多个标准，在AVX2标准下设立XMM寄存器（128位）和YMM寄存器（XMM的扩展，256位），对标量浮点数据操作时，只是用低32位（float）和低64位（double）。
4. 约定所有的XMM寄存器由调用者保存，XMM0用于返回。
5. AVX浮点操作不能以立即数为操作数，只能为所有常量分配和初始化存储空间，然后将值从内存读入，即只支持内存到寄存器，寄存器到寄存器，寄存器到内存三中方式。
6. 实模式与保护模式下的寄存器编码不同，bits n伪指令可以使其以下的指令编译为n位的机器码。实模式下可以使用保护模式下扩展的寄存器，同理保护模式下也可以使用实模式的。若在指令前端加上0x66，则会使操作数大小反转（如32位模式下将立即数传入BX寄存器，或是访存时使用word关键字），若是加上0x67，则会使寻址方式反转（如32位模式下使用BX寄存器间接寻址）。
7. 保护模式下引入全局描述符表，表项为段描述符，以此将内存分段约束，达到保护目的。

**Questions:**

1. 若想进行缓冲区溢出攻击，则需要知道程序运行时相应位置的栈地址，否则将无法跳转到恶意代码的入口，那这个地址是怎么知道的？

**2018.11.11-2018.11.12**

1.学习了CPU指令集体系结构，HCL以及Y86-64的顺序实现原理，弄清了push %rsp和pop %rsp的原理和结果。

2.初识全局描述符表与段描述符各个字段的意义。

**Tips:**

1. 指令集体系结构包括指令集及其编码、各种状态单元、异常事件处理和编程规范。
2. Y86-64顺序实现可以分为五个步骤：取指、译码、执行、访存、写回五个步骤。各个步骤中通过多个组合电路实现不同指令的不同需求。指令遵循从不回读的原则，即处理器从来不需要为完成一条指令的执行而去读该指令更新了的状态。
3. 对于push %rsp，在译码时会将当前%rsp的值给valB，将-8给valA，在执行阶段将valB-8，也就是执行加法计算，在访存阶段将%rsp的值写入地址为valB的内存中，最后将valB写回%rsp，也就是利用valB保存中间值，造成存入%rsp原值的情况。pop也是相同道理，唯一不同的是采取valA和valB都存%rsp，前者用于访存，后者用于恢复，因先恢复，后将访存内容写入寄存器导致%rsp接收到内存中的值（不会再-8）。
4. ALU单元会根据指令的功能码确定valA要和valB如何计算，并且输出valE。如push和pop操作，在译码阶段会根据操作码将valA设置为-8/8，因此这个阶段只需要把ALU设置为加法，而像OPX这样的指令，则根据x选择适合的运算模式，CPU的各个步骤的物理实现都会根据操作码环环相扣，因此得以实现不同的指令功能。
5. 进入保护模式前，需要构建GDT（全局描述符表），打开A20GATE（也就是打开CR0的PE位，使得CPU打破20位地址线的寻址限制）。

**2018.11.13-2018.11.14**

1.初探CPU的流水线设计方法，理解欧拉筛法，在一秒左右的时间内完成一亿以内素数个数的统计。

**Tips:**

1. 利用流水线技术可以达到同时执行多条指令的目的，主要实现原理是将指令的执行过程分为合适的段，在时间段之间（也就是CPU时钟周期了，为了使得指令同步，会选择最长段所需的延迟）插入一组寄存器缓冲数据，这样每次经历一个CPU时钟（上边沿），寄存器将结果输入到下一阶段的同时，接受上一个阶段所产生的结果。流水线CPU以吞吐量和延迟（执行完一条指令的时间）为指标。流水线CPU存在数据冒险和控制冒险，在当前指令尝试读取（前面的指令）未完成写入的寄存器/内存时，会发生数据冒险，控制冒险则是由条件转移和ret指令引起。对数据冒险，目前学习到可采用暂停当前指令来实现同步，对条件转移引起的控制冒险，可采用尝试的方法，发生错误时牺牲一定代价恢复，由于ret一般和call配对，因此可以通过增设一个硬件栈的方式来预测ret的返回地址。

**2018.11.15-2018.11.18**

1.完成对Y86-64流水线CPU的学习，初步探索C程序面向目标机的优化。

2.开启了分页模式，学习了二级页表及其具体实现，了解到二级页表的巧妙构造方法

**Tips:**

1. 流水线CPU利用分解指令周期，插入缓冲寄存器以提高指令执行的效率。它主要面临数据冒险与控制冒险两类问题。对于数据冒险，可采用转发逻辑进行处理，特别地，对于译码-访存（书中是加载-使用）类数据冒险，需暂停取指、译码阶段，并在执行阶段插入一个气泡，之后再利用转发逻辑解决。对于控制冒险，分JXX、RET两类，前者可利用暂停与插入气泡来恢复寄存器状态，后者只能利用插入两个气泡来恢复。对于多种冒险并行与多个异常同时发生这两类问题，较复杂，不太清楚。
2. 目前学到C代码主要有三个优化方向。减少过程调用，减少对不必要内存的应用，减少条件转移，多使用三目运算符。这些优化主要通过改变C代码来诱导编译器做出优化。需注意由于存在内存共享的情况（即函数中有两指针，编译器无法得知它们是否指向相同的地址空间）。
3. OS加电后，将BIOS载入内存，开始执行BIOS，BIOS做了一系列初始化操作，之后BIOS从磁盘的0号扇区开始寻找并装载MBR，最后跳转到MBR入口地址完成使命，MBR从磁盘中读取LOADER，随后跳转到LOADER入口地址完成MBR使命，LOADER在读取内存大小、开启保护模式（初始化GDT表、打开A20地址线、打开CR0的PE位）、分页模式（初始化页目录表、内核空间页表、打开CR0的PG位、将页目录表基址赋给CR3）后，将KERNEL载入内存，最后跳转到KERNEL入口地址（KERNEL开始大部分代码由C构成），将控制权交给操作系统内核。
4. 在开启分页模式之前，需要重新设置RSP、视频段基址、GDT表基址为相应的虚拟地址，并在开启分页后重新加载GDT基址（为何要在开启分页后呢？之前可以不？）
5. 32位CPU模式下，内存共4G，分页模式靠二级页表实现。页目录表的256\*3=768项开始到1022项归属内核空间，共1GB-4MB，页目录表的最后一项指向页目录表基址，在访问页目录表的最后一项时，会起到将页目录表当作页表来访问的效果，因此在保护模式下可利用页目录表的最后一项来更新页目录表与页表，具体原理：页目录表表项里存的是页表的基址，且第1023项存的是页目录表的基址，将二级地址（第二个高10位）设置为0-1022可指向页表基址，设置为1023指向页目录表基址，最后利用偏移地址完成对表项的更新）。

**2018.11.20-2018.11.21**

1.完成了CSAPP第五章的学习，对程序性能优化有了大致认识，复习了SRAM与DRAM。

2.成功在BOCHS中运行kernel，初步探索任务的.TSS段与CPU特权级。

**Tips:**

1. 第五章优化策略总结（从编译器与目标机CPU出发）:

* 减少过程调用
* 减少不必要的内存引用
* 根据CPU功能单元数目采取循环展开、多个累计变量（可交换且可结合）、重写结合变换（可结合）等优化措施（小心过度展开导致的寄存器溢出，这反而会使性能下降！）
* 适当利用条件传送代替条件转移，由于CPU已经有足够优秀的投机执行能力，对于循环条件判别这类判别结果分布不均（多正确）的指令，条件传送不会有太大效果
* 考虑写/读相关，尽可能修改调整
* 对于多个同类浮点运算指令来说，可以将一定的需要运算的浮点数同时装入YMM（128位寄存器，可存储多个浮点数）寄存器进行处理以提高指令并行度

1. 从程序的局部性原理出发，还可以利用存储器的高速缓存来优化程序，如尽量访问一组连续的数据等。
2. 32位elf文件格式与64位elf文件格式相似，若要解析32位elf文件，需将源程序编译链接生成32位目标文件（GCC -m32 && LD -m elf\_i386）。
3. 各个任务都有其TSS段，低28位保存三类特权级堆栈信息：SS、ESP。特别地，对于特权级为3的任务，有0、1、2三种，对于特权级为2的任务，只有0、1两种，依此类推。
4. CPU当前特权级(CPL)即当前执行的指令所属代码段段描述符中的DPL，这个值也保存在CS选择子的RPL部分。为防止堆栈混乱，任务的每个都有独立的堆栈。另外，特权级转移分为两类，一类是由中断门、调用门等实现低特权级转向高特权级，另一类是由调用返回指令从高特权级返回到低特权级，这也是唯一能让处理器降低特权级的方法。因此每个任务最多具有三种堆栈（不存在向第3特权级转移的情况），由于向高特权级转移时，高特权级的堆栈会保存低特权级的堆栈信息（也就是返回时候所用了），因此各任务无需记录自己的特权级堆栈信息。
5. TSS也是一种数据结构，其基址由TR寄存器保存。

**Experiment:**

1. 长数组复制，指针和下标两种实现方式，比较效率。
2. 循环体内printf与循环体外printf缓冲内容，比较效率。
3. 多个累计变量实现循环展开与不进行循环展开的效率对比。
4. 验证重新结合变换能取得更优的运算效率。
5. 比较条件转移和条件传送在条件判断结果分布均匀情况下的效率。
6. 对同一内存与对不同内存反复写读，比较效率。

**Result:**

1. 指针方式效率更高，且利用register关键字能达到和O3级编译优化相近的水平
2. 未完成
3. 运算效率在优化后得到较大改善，但对于编译优化后的整数加法来说，运算效率反而降低。
4. 同上。
5. 在条件判别结果分布均匀的情况下，条件传送性能远优于条件转移。
6. 没有明显差别。

**2018.11.22**

1.完成了CSAPP第六章的学习，复习了内存访问、磁盘访问机制、高速缓存组织方法（直接相联、全相联、组相联）、高速缓存读操作未命中时的置换策略（LRU、LFU），进一步把握局部性原理（时间局部性、空间局部性）在程序性能优化上的应用，从存储器系统结构的角度学习了程序性能优化策略。

**Tips:**

1. 指令也有指令高速缓存，因此局部性原理不仅仅存在于数据，也存在于指令。
2. 当写操作的对象未命中时，有写分配和非写分配两种策略，前者是将对象加载入高一级的缓存后覆写，后者是直接将值写入低一级的缓存。当命中时，有直写和写回两种策略，前者是直接将数据写入低一级的缓存，后者是先将数据写入高一级的缓存，并设置修改位以标识，待该对象需要被换出时写回低一级的缓存。
3. 利用局部性原理的优化策略（针对存储器，目标是尽可能让缓存命中）：

* 让最常见的情况运行得快，即对核心函数、核心循环细致考虑其时空局部性
* 尽量将循环步长设置为1，这样不仅能提高程序的空间局部性，在某些情况下还能发挥处理器的预读入功能（即将将要访问的数据块预先载入缓存，这种功能一般在步长较小时启用）
* 测试并分析计算机存储器层次的“存储器山”
* 考虑重新排列循环。如矩阵乘法，同样次数的乘法运算，可以采用不同的循环策略使之尽可能多的以小步长访问元素
* 可将数据结构组织成一个片，使之刚好为缓存大小（一般是一级高速缓存），这样对该结构的访问可完全在cache中进行以提高cache命中率
* 集中注意于内循环
* 一旦读入一个数据，尽可能多地访问它，这样可以有效提高时间局部性

**2018.11.24-2018.11.25**

1.OS会利用CPL（当前CPU特权级）、RPL（请求特权级）、DPL（目标特权级）共同实现特权数据保护，同时，一些特权指令(POPF、LGDT等)只有在最高特权级才可以使用。

2.学会了汇编与C的混合编程，并正确实现了部分print功能。

**Tips:**

1. 用户调用内核功能，如read时，可能会伪造目的缓冲区基地址的选择子为某内核数据段选择子以试图破坏内核数据，因此发生门调用时，CPU会将用户传入的选择子的RPL改为用户（任务）的CPL，这样在进入内核代码后，同时检测CPL与RPL是否在数值上小于DPL可以达到预防的目的。
2. 为提高访问效率，CPU对于特权级的检查发生在对各个选择子修改的时期，当选择子被赋值后，无论如何引用都不会引发特权级检查。
3. CALL可以实现向高特权级转移，而JMP只能平级转移，因JMP属于“一去不复返”类型的指令，会造成低特权级的代码无法执行。
4. 用户进程调用门时需满足RPL>门RPL，门RPL>CPL>DPL，即便用户伪造RPL，其访问特权也会被用户进程CPL所限制，因此RPL只用检查一次即可。
5. 一致性代码段可以使得用户进程调用高特权级的段，但是不会发生特权级转移，即CS选择子的RPL不会发生改变。
6. 在用户进程（特权级为3）调用某系统调用（代码段特权级为0）后执行返回指令时会检查各个选择子的RPL是否在数值上低于返回后的CPL，若低于，为保护高特权级的数据，会将相应的选择子（段寄存器）置0，因此在实现内核函数时（如print），若想使用内核数据段（如GS），必须重新赋值。
7. 32位ELF目标文件发生过程调用时，默认为CDECL，会将参数从右到左入栈，而不会采用EDI、ESI之类的寄存器传参。
8. 显存的文本模式下一屏可显示25\*80个字符，每个字符占两个字节，低字节为字符ASCII码，高字节为字符显示方式，由于利用中断获取的光标CURSOR为字符的序号，因此需要将其乘以二（左移一位）真正需填入字符数据的地址。

**2018.11.27**

1.复习了编译、链接、装载、运行、共享库、重定位等概念，深入了解PLT（延迟绑定）与GOT的关系以及链接时因链接顺序带来的符号决议错误，最后学习到编译时、链接时、运行时的函数打桩方法。

2.初探异常控制流(ECF)，C++、JAVA的异常依赖C函数setjmp()、longjmp()实现。异常分四类：中断（异步）、陷阱（同步）、故障（同步）、终止（同步）。其中中断和陷阱会返回下一条指令，故障若被正常解决则会返回到当前指令，否则终止进程，而终止则会直接终止进程。

**Tips:**

1. PLT的实现由PLT与GOT共同完成。GOT中保存了共享库全局符号的地址，这包括函数与变量（代码与数据）。在调用共享库函数时，会从PLT中相应位置（每个库函数在PLT中对应一个入口）执行一条跳转指令，该指令会以GOT中该函数的项为目标地址进行转移。若是二次调用，则成功转移（GOT中相应地址已重定位），若是首次调用，则会跳转到装载该库函数的入口代码（将GOT中对应项初始化为该地址即可），随后携参数调用动态加载器，实现延迟绑定。
2. 链接时，链接器从左至右扫描可重定向文件，对每个可重定向文件，链接器将其未定义的全局符号加入U，将U中的且在该文件中已定义的全局符号加入D（同时从U中去除）。如此，若包含函数功能的文件卸载了调用者左侧，会发生已定义的函数无法加入D，最终使得U非空，产生链接错误（觉着不太对，完全可以把定义的符号直接加入D）。
3. 重定向主要完成了符号地址的确定和对符号引用重定位两个工作。因此即使无外部符号，也需要重定向。重定位依赖重定位表，表项给出了待重定位的位置信息与重定位方式等，协助重定位的进行。
4. 打桩的本质时对函数符号引用规则做了修改（我认为）。

* 编译时打桩依赖对源程序的重新编译。利用预处理器的宏机制，将源程序中对库函数的调用转向自己实现的函数（利用-I .优先搜索当前目录下的头文件）。
* 链接时打桩依赖对源程序目标文件的重新链接。利用链接参数将对f（假设f是库函数）的引用转为\_\_wrap\_f，对\_\_real\_f的引用转为f来实现，因此在链接时再加入自己的\_\_wrap\_f即可（\_\_wrap\_f中利用\_\_real\_f访问库函数）。
* 运行时打桩依赖源程序的可执行文件。它利用动态链接库运行时加载函数与动态链接库优先搜索LD\_PRELOAD环境变量中的路径这两种机制实现。利用LD\_PRELOAD让可执行程序调用自己动态库中的函数，自己的动态库中利用运行时加载函数动态加载库函数完成功能（这个运行时加载不会受LD\_PRELOAD环境变量的影响吗？）。

**Experiment:**

1. extern声明的变量存于.data段？
2. 将静态链接库置于main.o前会因符号决议规则发生链接错误？（应加-static参数）
3. 比较链接时加-static参数与不加时可执行文件的大小。
4. 同一目标文件中两个不同函数声明（定义）的同名static局部变量是如何做符号修饰的？
5. 验证引用目标文件中已定义符号的指令是否需要重定位。
6. 调用execve后还会返回到调用者继续执行吗?

**Result:**

1. Windows下不是。
2. Windows下和Linux下皆不会。
3. 动态链接比静态链接要小（动态链接库是自己创建的一个较小的链接库，可能影响结果）。
4. Windows下GNUC会将局部变量（假设是a）修饰为a.x（x为某个数）。
5. 需要重定位。
6. 不会。

**2018.11.28-2018.11.29**

1.学习了可编程中断控制器8295A的使用，利用内联汇编实现了针对端口读写的IO函数，构造了IDT表、中断处理例程，实现了中断的初始化。

**Tips:**

1. 外部设备通过IRQ引脚连接到8295A，当外部设备引发中断后，8295A根据IMR寄存器过滤被屏蔽的中断信号后（一个8295A有8个引脚，IMR是8位寄存器，为1的位表示屏蔽），通过其决策机制，决定将哪些中断优先通过INTR接口传输给CPU，CPU接收到中断信号（中断号）后，暂停当前的任务，根据中断信号以及IDT寄存器（中断描述符表基址）跳转到相应中断向量，随后进入中断处理例程。注意，中断处理例程是一次性完成的，也就是具有原子性。
2. 可以利用ICW（初始化命令寄存器组）与OCW（操作命令寄存器组）两类寄存器控制8295A。对于每个8295A，只有两个IO端口，部分OCW可以通过写入的控制字中某两位来识别，其他端口则规定了的写入次序。主片IR0引脚上的是时钟中断，可以通过将8295A的起始中断号初始化为不同的值来避免与CPU初始化的中断号的冲突。
3. 部分中断会压入错误码，为保证中断处理程序在恢复栈时有一致的操作，在利用宏汇编初始化中断处理程序时，对于压入错误码的中断，在开头插入NOP指令占位，否则插入PUSH 0指令，充当错误码。

**2018.12.1-2018.12.5**

1.浅识MakeFile，利用8253加快了时钟中断的频率，学习了可变宏参数，并构造了断言函数库与字符串函数库。

**Tips:**

1. 8253为可编程计数器，计数器开始工作需两个条件，一是GATE为高电平，这是硬件控制的，二是计数初值已写入8253中的减法计数器，这是由软件的out指令控制的。最终由硬件控制来启动硬件的方式称硬件启动，而最终通过软件控制来启动硬件的方式为软件启动。8253利用晶振产生的时钟信号计数，我们主要通过8253的第二种工作方式（比特发生器）让8253在经过特定数量的时钟信号后发出一个定时信号，以实现定时功能。
2. \_\_VAR\_ARGS为编译器内置宏参数，等价于可变宏参数中的所有参数，#可以将宏参数转为字符串。
3. Linux中文件具有三类时间，一是atime（访问文件数据部分时间），cat会修改此信息，而ls不会；二是ctime（文件属性或数据的改变时间）；三是mtime（文件数据部分修改时间）。MakeFile根据文件的mtime来判断是否需要执行操作。

**2018.12.6-2018.12.9**

1.实现了位图，并利用位图尝试实现内存管理系统。

**Tips:**

1. 位图中的指针最好设置为单字节指针，这样字节偏移量就是位图的索引。在内存管理系统中，我们用位图的一位代表一页，一页为4KB内存。
2. 内存管理系统需要获取总内存，我们在loader中曾计算过，并将其保存在物理地址0xb03处（可以反汇编loader代码查看），由于0xb03处于低端1MB内存，因此虚拟地址0xb03与物理地址0xb03相对应，可凭此嵌入内联汇编代码获取总内存。
3. 内存管理系统利用位图实现，主要依赖内核地址池与用户地址池两个结构体成员维护。由于地址池主要管理未分配的地址，因此已被占用的低端1M地址与已初始化的页目录和页表需除去（这两者刚好是连续分布在1M地址后），剩余的空间按4K每页，以一定比例分配个内核与用户（目前是1:1）。由于分配的是未被占用的内存，而位图也需要占用内存，因此将位图放入低端1MB内存处。在loader分页后，我们将栈指针初始化为0x9f000，在引入进程后，我们需要用4KB，即一页的内存容量保存PCB（进程控制块），而主进程的PCB就占用0x9e000起始处的页框（由于栈指针会下移，因此主进程PCB可用的实际容量不到4KB），因此，本系统中将0x9a000-0x9e000的内存划给位图，共四个页框，一个页框可以映射4K \* 8 \* 4KB = 128MB大小的内存，因此本系统共支持512MB的位图。内核地址池与用户地址池位图依次放在以0x9a000起始的内存单元。

**2018.12.10**

1.复习fork，execve，wait等函数的使用以及main函数入口时栈中参数与环境变量分布，对其有了更深的理解，并初步了解Linux的信号机制。

Tips:

1. fork函数调用一次，返回两次，在子进程中返回0，在调用者进程中返回子进程PID，fork生成的子进程会继承父进程打开的资源，也会继承父进程屏蔽、解锁的信号。
2. execve会覆盖当前进程的地址空间，并继承了调用execve时函数已打开的所有文件句柄，execve并不创建新的进程，因此新的程序仍然拥有与调用者相同的PID。
3. pause函数会阻塞当前进程，直到当前进程接收一个信号。sleep函数会让当前进程休眠，其返回值是sleep剩余的时间，一般sleep返回0，但sleep若被某信号中断，则会返回非0值。
4. 调用waitpid或wait函数可回收已完成的子进程（僵死进程）资源。
5. printf在函数执行返回或退出或输出缓冲区满时才会输出信息，因此对于多进程的程序，在printf后要注意使用fflush刷新输出缓冲区，否则会造成输出异常。
6. 进程从内核态返回用户态时会检查其信号集，并按一定策略（一般选最小的信号）接收、处理信号。进程在处理信号（假设是K）的同时，会阻塞对信号K的接收，随后若再有信号K发送至该进程，而处理信号的进程仍未结束时，该信号将被丢弃。因此，进入信号处理例程代表至少有一个信号被接收了。
7. 利用signal函数可以绑定信号与信号处理例程到当前进程，还可利用sigprocmask函数（书中未介绍用法）屏蔽、解锁信号，也可用于让进程阻塞直至某信号发生（其中可利用sigsuspend来解决CPU忙等问题）。

**2018.12.12**

1.完成了CSAPP第八章的学习，了解了C的非本地跳转机制。

**Tips:**

1. 非本地跳转主要通过setjmp与longjmp两个函数实现，针对信号中的非本地跳转可利用sigsetjmp与siglongjmp实现。
2. setjmp返回值不能赋值给变量，并可以返回多次，第一次是在setjmp调用时，会返回0，随后每当longjmp调用一次，env中最后保存的返回地址处的setjmp将返回longjmp参数中的返回值。
3. longjmp函数不返回。
4. 由于siglongjmp函数不是异步信号安全的函数，因此siglongjmp可达的代码中要调用异步信号安全的函数。

**Question:**

1. 为什么setjmp的返回值不允许赋值给变量？
2. longjmp是如何平衡堆栈的？
3. 为什么要为信号中的非本地跳转机制定义专门的sigsetjmp与siglongjmp函数？
4. 什么是异步信号安全？
5. 为什么siglongjmp不是异步信号安全的函数，其可达的代码中就只能调用异步信号安全的函数？