一、编译和链接

预处理

预处理中会展开以#起始的行，包括#if、#ifdef、#ifndef、 #else 、#elif 、 # endif、#define、#include、#line、 #error、#pragma 以及单独的#。其实就是把除了函数(包括 main)以外的东西都展开成指定的形式，比如加上行号等，并将头文件里包含的东西所对应的文件(包括子文件夹)也都写入进去以便调用。同时也会把写在文件里的注释都删掉。具体操作有：

1）将所有的#define 删除，并展开所有的宏定义；

2）处理所有的预编译指令，例如：#if,#elif,#else,#endif;

3）处理#include 预编译指令，将被包含的文件插入到预编译指令的位置；

4）添加行号信息文件名信息，便于调试；

5）删除所有的注释：// /**/;

6）保留所有的#pragma 编译指令

对 test.c **运行预处理**，在命令行输入 gcc -E test.c -o test.i，**得到预处理后的文件** test.i，发现 test.i 可以用文本编辑器打开，得到的是把#include 中的头文件展开。直接拷贝到原来的 test.c 的文件头部，在其中可以找到 printf，scanf 等函数的申明。

在程序前**加上**#define test_char char，把代码中的 char 改为 test_char，再次预处理后，得到的文件不包含 test_char 和#define 语句，但是所有的 test_char 都已经被修改为 char 了。

在程序中**加入几行注释**，发现在预处理之后得到的文件中不包含注释。

修改.c 文件，**产生语法上的错误**，比如少加一个分号，发现仍然可以得到预处理的.i 文件。这说明预处理的过程并不会检查语法错误。

编译

编译器将 c 语言程序翻译成汇编语言程序，具体操作有：

1）扫描，语法分析，语义分析，源代码优化，目标代码生成，目标代码优化；

2）生成汇编代码；

3）汇总符号；

4）生成.s 文件；

在命令行中**运行** gcc -S test.c，**得到编译后产生的文件** test.s。

分析**文件内容**。首先是一个只读数据分区，由.section .rodata 得到，其中有 2 个字符串，分别是输入和输出的格式控制字符串。之后由.globl main 声明一个全局符号 main，再由.type main, @function 声明其类型为函数,然后从.LFB0 开始时函数的主体,由汇编语言表示。可以发现在调用 scanf 和 printf 之前，都从 LC0 或 LC1 中得到格式控制字符串，作为函数的第一个参数。

在 test.c 中制**造语法错错误**，发现会无法生成.s 文件，无法经行编译的步骤。这说明编译时会对语法做出分析。

汇编

汇编语言通过汇编器编译成可重定位目标程序.o，把生成的汇编指令逐条翻译成机器可以识别的形式，即机器码，这一步会产生平台相关性，即决定了在哪种平台下运行，具体操作是：

1）根据汇编指令和特定平台，把汇编指令翻译成二进制形式；

2）合并各个 section，合并符号表；

3）生成.o 文件；

使用 objdump -h test.o 查看**各段的信息**，得到 7 个段：

.text :代码段(存放函数的二进制机器指令)

.data :数据段(存已初始化的局部/全局静态变量、未初始化的全局静态变量)

.bss :bss 段(声明未初始化变量所占大小)

.rodata :只读数据段(存放两个输入输出格式控制字符串)

.comment :注释信息段

.node.GUN-stack :堆栈提示段

输入 gcc -c test.c **得到汇编结果** test.o。

再使用 objdump -t test.o 查看**符号表**，可以看到有全局函数 main，未在本文件定义的符号 scanf，printf，__stack_chk_fail，类型不确定。

再使用 objdump -s test.o 将**所有段的内容**以 16 进制方式打印出来，可以清楚地看到两个输入输出格式控制字符串被放在.rodatada 段，这与编译产生的.s 文件中的内容相符合。

链接

链接会将目标文件和所需的库函数用链接器进行链接；

1）合并各个.obj 文件的 section，合并符号表，进行符号解析；

2）符号地址重定位；

3）生成可执行文件；

输入 gcc test.c -o with15 **得到可执行文件** with15，输入./with15 可以正常运行。由于已经把缓冲区大小规定，读入的时候也采用格式控制符读入 15 个字符，所以无论输入什么，输入多长，只会输出前 15 个字符。

使用 objdump -t with15 **查看符号表**，可以发现全局函数 main 的类型不变，但是 scanf，printf，__stack_chk_fail 都是被解析成函数了，是外部链接符号，未在本目标文件定义。

链接方式可分为两种：

（1）静态链接 在这种链接方式下，函数的代码将从其所在地静态链接库中被拷贝到最终的可执行程序中。这样该程序在被执行时这些代码将被装入到该进程的虚拟地址空间中。静态链接库实际上是一个目标文件的集合，其中的每个文件含有库中的一个或者一组相关函数的代码。

（2）动态链接 在此种方式下，函数的代码被放到称作是动态链接库或共享对象的某个目标文件中。链接程序此时所作的只是在最终的可执行程序中记录下共享对象的名字以及其它少量的登记信息。在此可执行文件被执行时，动态链接库的全部内容将被映射到运行时相应进程的虚地址空间。动态链接程序将根据可执行程序中记录的信息找到相应的函数代码。

二、进程的创建

进程的经典定义就是一个执行中程序的实例。系统中的每个程序都运行在某个进程的上下文中。上下文是由程序正确运行所需的状态组成的。这个状态包括存放在内存中的程序的代码和数据，它的栈、通用目的寄存器的内容、程序计数器、环境变量以及打开文件描述符的集合。

操作系统对进程的识别采用的是唯一的进程标识符（pid），pid 通常是

一个整数值。系统内的每个进程都有一个唯一 pid，它可以用作索引，以便访问内核中的进程的各种属性。

每次用户通过向 shell 输入一个可执行目标文件的名字（在本实验中为 with15），运行程序时，shell 就会创建一个新的进程，然后在这个新进程的上下文中运行这个可执行目标文件。应用程序也能够创建新进程，并且在这个新进程的上下文中运行它们自己的代码或其他应用程序。

一旦系统启动后，进程 init 可以创建各种用户进程。当在命令行输入./with15（由 gcc 编译得到的可执行文件），就创建了一个用户进程。因为 with15 不是一个内置的 shell 命令，所以 shell 会认为 with15 是一个可执行目标文件，通过调用加载器（loader）来运行它。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本程序。

Linux 系统中的每个程序都运行在一个进程上下文中，有自己的虚拟地址空间。当 shell 运行 with15 时，父 shell 进程生成一个子进程，它是父进程的一个复制。子进程通过 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。新的栈和堆段被初始化为零。当加载器运行时，它创建一个内存映像，通过将虚拟地址空间中的页映射到可执行文件的页大小的片（chunk），在程序头部表的引导下，载器将 with15 的片（chunk)复制到代码段和数据段。新的代码和数据段被初始化为可执行文件 with15 的内容。

最后，加载器跳转到 start 地址，它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。

上下文切换与多任务

操作系统内核使用一种称为上下文切换的较高层形式的异常控制流来实现多任务。

上下文切换机制是建立在较低层异常机制之上的，内核为每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。

上下文切换的具体步骤是

1）保存当前进程的上下文，

2）恢复某个先前被抢占的进程被保存的上下文，

3）将控制传递给这个新恢复的进程。

上下文切换的情况有：

1）为了保证所有进程可以得到公平调度，CPU 时间被划分为一段段的时间片，这些时间片再被轮流分配给各个进程。这样，当某个进程的时间片耗尽了，就会被系统挂起，切换到其它正在等待 CPU 的进程运行。

2）进程在系统资源不足（比如内存不足）时，要等到资源满足后才可以运行，这个时候进程也会被挂起，并由系统调度其他进程运行。

3）当进程通过睡眠函数 sleep 这样的方法将自己主动挂起时，自然也

会重新调度。

4）当有优先级更高的进程运行时，为了保证高优先级进程的运行，当前进程会被挂起，由高优先级进程来运行

5）发生硬件中断时，CPU 上的进程会被中断挂起，转而执行内核中的中断服务程序。

三、执行指令

指令顺序执行的具体步骤（不考虑流水线）：

取指（fetch)：取指阶段从内存读取指令字节，地址为程序计数器（PC）的值。从指令中抽取出指令指示符字节的两个四位部分，称为 icode(指令代码）和 ifun（指令功能)。它可能取出一个寄存器指示符字节，指明一个或两个寄存器操作数指示符 rA 和 rB。它还可能取出一个 8 字节常数字 valC。它按顺序方式计算当前指令的下一条指令的地址 valP。也就是说，valeP 等于 PC 的值加上已取出指令的长度。

译码（decode)：译码阶段从寄存器文件读入最多两个操作数，得到值 valA 和/或 valB。通常，它读入指令 valA 和 valB 字段指明的寄存器，不过有些指令是读寄存器%rsp 的。

执行（execute)：在执行阶段，算术/逻辑单元（ALU）要么执行指令根据 ifun 的值指明的操作，计算内存引用的有效地址，要么增加或减少栈指针。得到的值我们称为 valE。在此，也可能设置条件码。对一条条件传送指令来说，这个阶段会检验条件码和传送条件（由 ifun 给出），如果条件成立，则更新目标寄存器。同样，对一条跳转指令来说，这个阶段会决定是不是应该选择跳转到对应位置。

访存（memory)：访存阶段可以将数据写入内存，或者从内存读出数据。读出的值为 valM.

写回（write back)：写回阶段最多可以写两个结果到寄存器文件。

更新 PC(PC update)：将 PC 设置成下一条指令的地址。

处理器无限循环执行这些阶段。在我们简化的实现中，发生任何异常时，处理器就会停止，比如执行 halt 指令或非法指令，或试图读或者写非法地址。在更完整的设计中，处理器会进入异常处理模式，开始执行由异常的类型决定的特殊代码。

四、运行 main 函数

逐语句分析见附件的代码文件注释。

汇编语言的操作：

首先开辟一个栈帧，大小是 32 位，即 0x20 的大小，作为栈空间。

然后将栈底设置一个“金丝雀”，用于检测是否发生缓冲区溢出。

接着把栈顶部地址作为字符串变量 s 的开始地址，放在代表第二个参数的寄存器中，然后从.rodata 区取得一个字符串，作为输入格式控制字符串，放在代表第一个参数的寄存器中。于是调用 scanf 函数，得到输入的内容。

然后把栈顶部地址作为字符串变量 s 的开始地址，放在代表第二个参数的寄存器中，然后从.rodata 区取得另外一个字符串，作为输出格式控制字符串，放在代表第一个参数的寄存器中。于是调用 printf 函数，得到输出的内容。

最后将栈底的“金丝雀”与%fs:40 中的寄存器相比较，判断是否发生缓冲区溢出。如果相等，就说明没有发生缓冲区溢出，就正常结束函数，否则就调用__stack_chk_fail，代表发生 stack smashing，即栈溢出。
根据汇编语言进行分析：

但是实际并不会发生这种情况，由于在输入时加上了格式控制符号，最多只会读入 15 个字符，所以并不会发生缓冲区溢出。即 main 函数一定会正常返回。

删除输入格式控制中的 15，改成任意读取。重新编译，得到可执行文件 without15，运行，在输入 15 个字符以下时，可以正常运行。但是超出一定范围后就会发生栈溢出的错误。根据对汇编语言的分析，栈帧大小为 32 字节，在栈底的 8 个字节放着校验是否溢出的“金丝雀”，猜测最多可以容纳 24 个字符。输入 24 个字符，可以得到正常运行的结果，但是只要输入 25 个或者更多的字符，就会发生栈溢出。这证明我们的猜想是正确的，原函数最多可以容纳 24 字节的内容。

五、调用库函数

函数调用的过程：

当 x86-64 过程需要的存储空间超出寄存器能够存放的大小时，就会在栈上分配空间，这个部分称为过程的栈帧，当前正在执行的过程的帧总是在栈顶。当过程 P 调用过程 Q 时，会把返回地址压入栈中，指明当 Q 返回时，要从 P 程序的哪个位置继续执行。Q 的代码会扩展当前栈的边界，分配它的栈帧所需的空间。在这个空间中，它可以保存寄存器的值，分配局部变量空间，为它调用的过程设置参数。大多数过程的栈帧都是定长的，在过程的开始就分配好了。如果 Q 需要更多的参数，P 可以在调用。之前在自己的栈帧里存储好这些参数。

为了提高空间和时间效率，x86-64 过程只分配自己所需要的栈帧部分。例如，许多过程有 6 个或者更少的参数，那么所有的参数都可以通过寄存器传递。因此，某些栈帧部分可以省略。实际上，许多函数甚至根本不需要栈帧。当所有的局部变量都可以保存在寄存器中，而且该函数不会调用任何其他函数时，就可以这样处理。

将控制从函数 P 转移到函数 Q 只需要简单地把程序计数器（PC）设置为 Q 的代码的起始位置。从 Q 返回的时候，处理器必须记录好它需要继续的执行的代码位置。在 x86-64 机器中，这个信息是用指令 callQ 调用过程 Q 来记录的。压入的地址 A 称为返回地址，是紧跟在 call 指令后面的那条指令的地址。该指令会把地址 A 压入栈中，并将 PC 设置为 Q 的起始地址。对应的指令 ret 会从栈中弹出地址 A，并把 PC 设置为 A。

这种把返回地址压入栈的简单的机制能够让函数在稍后返回到程序中正确的点。C 语言标准的调用—返回机制刚好与栈提供的后进先出的内存管理方法吻合。

当调用一个过程时，除了要把控制传递给它并在过程返回时再传递回来之外，过程调用还可能包括把数据作为参数传递，而从过程返回还有可能包括返回一个值。x86-64 中,大部分过程间的数据传送是通过寄存器实现的。当过程 P 调用过程 Q 时，P 的代码必须首先把参数复制到适当的寄存器中。当 Q 返回到 P 时，P 的代码可以访问寄存器各 rax 中的返回值。

x86-64 中，可以通过寄存器最多传递 6 个整型参数。寄存器的使用是有特殊顺序的，寄存器使用的名字取决于要传递的数据类型的大小。会根据参数在参数列表中的顺序为它们分配寄存器。

如果一个函数有大于 6 个整型参数,超出 6 个的部分就要通过栈来传递。假设过程 P 调用过程 Q，有 n 个整型参数，且 n>6。那么 P 的代码分配的栈帧必须要能容纳 7 到 n 号参数的存储空间。要把参数 1~6 复制到对应的寄存器，把参数 7~n 放到栈上，而参数 7 位于栈顶。通过栈传递参数时，所有的数据大小都向 8 的倍数对齐。参数到位以后，程序就可以执行 call 指令将控制转移到过程 Q 了。过程 Q 可以通过寄存器访问参数，有必要的话也可以通过栈访问。相应地，如果 Q 也调用了某个有超过 6 个参数的函数，它也需要在自己的栈帧中为超出 6 个部分的参数分配空间。

系统调用和库函数调用：

系统调用：

系统调用是操作系统内核提供的函数，在内核态运行，是操作系统为用户提供的一些接口。它通过软中断向内核态发出一个明确的请求。有一些任务需要进程跑在内核态才能执行，比如和硬件打交道。所以进程调用系统调用就能让自己运行在内核态从而执行这些类似的任务。printf() 就需要调用 write() 系统调用来完成输出。

库函数：

库函数是高层的，是在系统调用上的一层包装，运行在用户态，为程序员提供调用真正的在幕后完成实际事务的系统调用的更为方便的接口。有的库函数使用了系统调用，有的库函数没有用到系统调用，本程序中的库函数 scanf()和 printf()就用到了系统调用。

系统调用与库函数调用的区别：

系统调用通常用于底层文件访问，库函数调用通常用于应用程序中对一般文件的访问。

系统调用发生在内核空间，因此如果在用户空间的一般应用程序中使用系统调用来进行文件操作，会有用户空间到内核空间切换的开销。事实上，即使在用户空间使用库函数来对文件进行操作，因为文件总是存在于存储介质上，因此不管是读写操作，都是对硬件（存储器）的操作，都必然会引起系统调用。也就是说，库函数对文件的操作实际上是通过系统调用来实现的。

系统调用是操作系统相关的，因此一般没有跨操作系统的可移植性。库函数调用是系统无关的，因此可移植性好。

为何不直接使用系统调用？

读写文件通常是大量的数据（这种大量是相对于底层驱动的系统调用所实现的数据操作单位而言），这时使用库函数就可以大大减少调用系统调用的次数。

再加上缓冲区技术的作用，在用户空间和内核空间，对文件操作都使用了缓冲区，都是先将内容写到用户空间缓冲区，当用户空间缓冲区满或者写操作结束时，才将用户缓冲区的内容写到内核缓冲区，同样的道理，当内核缓冲区满或写结束时才将内核缓冲区内容写到文件对应的硬件媒介。

对于程序员来说，使用库函数不但极大地方便了人们编写程序库函数，而且可以很方便的调试，而系统调用因为运行在内核，调试很麻烦。

六、程序退出

进程会因为三种原因终止：

1)收到一个信号，该信号的默认行为是终止进程，

2)从主程序返回，

3)调用 exit 函数。

其中，本次实验是通过从主程序退出的方式终止进程的。当所有语句执行完毕后，进程返回状态值（本实验为 0）到父进程（通过系统调用 wait()）。所有进程资源，如物理和虚拟内存、打开文件和 I/O 缓冲区等，会由操作系统释放。

对于资源的释放，采取“谁申请谁释放”的原则。进程自身申请的信号量、文件描述符等，需要进程自己释放。而进程描述符、内核栈这些资源则需要父进程来回收。

当一个进程终止时，操作系统会释放其资源。不过，它位于进程表中的条目还是在的，直到它的父进程调用 wait()；这是因为进程表包含了进程的退出状态。当进程已经终止，但是其父进程尚未调用 wait()，这样的进程称为僵尸进程。

所有进程终止时都会过渡到这种状态，但是一般而言僵尸只是短暂存在。一旦父进程调用了 wait()，僵尸进程的进程标识符和它在进程表中的条目就会释放。

如果父进程没有调用 wait() 就终止，使得子进程成为孤儿进程。对这种情况，系统就会将 init 进程作为孤儿进程的父进程。进程 init 定期调用 wait()，以便收集任何孤儿进程的退出状态，并释放孤儿进程标识符和进程表条目。

七、总结

本次实验通过对 test.c 作为样例，分析了在编写完程序后发生的事情，按照时间顺序，从编译和链接、进程的创建、CPU 指令的执行过程、运行 main 函数、调用库函数、进程的终止这六个阶段分别做了详细深入的分析。

在编译和预处理阶段，分为四个小部分。第一部分是预处理，通过实验，观察到预处理会将源文件中的#include 项展开，将宏定义替换，将注释删除，并且不会检查语法错误。第二部分是编译，通过实际操作，验证了 c 语言程序在这一步会翻译成汇编语言程序，分析了得到的编译文件各部分的内容以及与 test.c 的关联性，还证明在这一步就会检查语法的正确性。第三部分是汇编，验证了在这一步会把汇编语言编译成可重定位目标程序.o，把生成的汇编指令逐条翻译成机器可以识别的形式，通过查看符号表、检视段内容来验证猜想。第四步是链接，在链接完成后通过手动检查符号表并与第三步汇编中的符号表对比，分析链接与汇编的不同。

在进程的创建阶段，首先分析了进程的定义和进程的标识符 pid，然后从在命令行里输入./with15 开始，逐步分析进程创建的步骤，首先父进程复制自己，然后通过 execve 启动加载器，赋值可执行文件的代码段与数据段到该进程，最后加载器跳转到 start 地址，调用该程序的 main 函数，成功创建一个目标进程。随后又分析了操作系统如何在系统层面实现多任务的，即在运行./with15 的“同时”还运行其它进程。从上下文切换这一种异常控制流出发，分析了上下文切换的具体步骤、上下文切换的情况这两个方面，最终阐明系统是如何通过上下文切换来实现多任务的。

在 CPU 执行阶段，我从底层 CPU 出发，详细分析了指令顺序执行的步骤，即取指、译码、执行、访存、写回、更新 PC 这六个部分。由于本学期课程对于流水线并未深入展开讲解，所以在分析 CPU 执行过程时并未考虑流水线的影响，以及对于不同异常状态的不同处理方式。

在运行 main 函数阶段，我在汇编语言层面，逐句分析了每一句指令的操作以及目的，然后从整体入手，分析汇编语言下，到底执行了什么操作，即开辟栈帧，设置“金丝雀”，参数传递、调用函数、校验“金丝雀”，程序返回这些操作。然后发现“金丝雀”在源程序下是多余的，因为输入格式控制在 15 个字符之内。于是进行下一步实验，删除输入格式控制字符串中的 15，编译得到 without15，发现在输入 24 个字符之内是可以正常执行，一旦多于 24 个字符，就会极大概率破坏金丝雀，发生栈溢出。字符数与栈帧大小减去金丝雀大小剩余的空间刚好吻合，再次证明 24 个字符的上限。

在调用库函数阶段，首先我分析了调用函数的具体步骤，从空间分配、控制转移、参数传递等方面详细分析了调用过程。之后注意到了库函数调用与系统调用的区别，随后分析了系统调用和库函数的定义，以及两种调用方式的区别。最后，分别从调用次数、缓冲区技术、程序员视角分析了为什么大多数情况下为何使用库函数调用，而很少采用系统调用的原因。

在程序退出阶段，首先分析了程序退出的三个原因，即收到信号、主程序返回、调用 exit 函数。以本程序为例，分析了当所有语句执行完毕后，“谁申请谁释放”执行 ret 指令终止程序时发生了哪些内容的释放。除此之外，我还分析了对于僵尸进程和孤儿进程这两种特殊的情况的发生条件，以及操作系统会做些什么防止僵尸进程和孤儿进程的大量存在。

本报告通过动手实验、理论分析、查阅资料等方式，结合本学期课程的内容与自己的理解，以 test.c 为样例，较为完整全面地阐述清楚其背后的逻辑，对于深入理解程序从编写、编译、运行到最后终止的各个过程发生了什么有着较大的帮助。

实践是检验真理的唯一标准，在深入了解程序从编写、编译、运行到最后终止的各个过程发生了什么之后，在以后完成项目、编写程序中才能想得更深更远，对于各种可能存在的错误信息有一个全面的把握，利于以后修正程序的漏洞。除此之外，在了解各个部分的工作方式之后，我们才成写出更友好的代码，在不损失正确率的前提下优化程序性能，提高执行效率、减少复杂度、减少开销。

参考文献：

《深入理解计算机系统》原书第三版
https://blog.csdn.net/gt1025814447/article/details/80442673
https://blog.csdn.net/w1050321758/article/details/108529233
https://segmentfault.com/a/1190000040798939
http://c.biancheng.net/view/1207.html
https://blog.csdn.net/dreamispossible/article/details/89389181
https://blog.csdn.net/qq_41489540/article/details/109255409
https://blog.csdn.net/qq_34228570/article/details/72997248
https://zhuanlan.zhihu.com/p/117626599