实验一：Linux基础与系统调用——Shell工作原理(part2)

实验目的

- 学习如何使用Linux系统调用：实现一个简单的shell
    - 学习如何编写makefile：实现一个简单的makefile来测试shell

实验环境

- 虚拟机：VMware
- 操作系统：Ubuntu 24.04.2 LTS

> 系统的安装形式可以自由选择，双系统，虚拟机都可以，系统版本则推荐使用本文档所用版本。注意：由于Linux各种发行版非常庞杂且存在较大差异，因此本试验在其他Linux发行版可能会存在兼容性问题。如果想使用其他环境（如vlab）或系统（如Arch、WSL等），请根据自己的系统**自行**调整实验步骤以及具体指令，达成实验目标即可，但其中出现的兼容性问题助教**无法**保证能够一定解决。

实验时间安排

> 注：此处为实验发布时的安排计划，请以课程主页和课程群内最新公告为准

- 3.28 晚实验课，讲解实验第一部分、第二部分，检查实验
- 4.4 清明节放假
- 4.11 晚实验课，讲解实验第三部分，检查实验
- 4.18 晚实验课，检查实验
- 4.25 晚及之后实验课，补检查实验

> 补检查分数照常给分，但会**记录**此次检查未按时完成，此记录在最后综合分数时作为一种参考（即：最终分数可能会低于当前分数）。

检查时间、地点：周五晚18: 30~22: 00，电三楼406/408。

如何提问

- 请同学们先阅读《提问的智慧》。 **原文链接**
- 提问前，请先**阅读报错信息**、查询在线文档，或百度。 **在线文档链接**；
- 在向助教提问时，请详细描述问题，并提供相关指令及相关问题的报错截图；
- 在QQ群内提问时，如遇到长时未收到回复的情况，可能是由于消息太多可能会被刷掉，因此建议在在线文档上提问；
- 如果助教的回复成功地帮你解决了问题，请回复“问题已解决”，并将问题及解答更新到在线文档。这有助于他人解决同样的问题。

为什么要做这个实验

- 为什么要学会使用Linux?
  - Linux的安全性、稳定性更好，性能也更好，配置也更灵活方便，所以常用于服务器和开发环境。实验室和公司的服务器一般也都用Linux；
  - Linux是开源系统，代码修改方便，很多学术成果都基于Linux完成；
  - Windows是闭源系统，代码无法修改，无法进行后续实验。
- 为什么要使用虚拟机?
  - 虚拟机对你的电脑影响最低。双系统若配置不正确，可能导致无法进入Windows，而虚拟机自带的快照功能也可以解决部分误操作带来的问题。
  - 本实验并不禁止其他环境的使用，但考虑其他环境（如WSL）变数太大，比如可能存在兼容性或者其他配置问题，会耽误同学们大量时间浪费在实验内容以外的琐事，因此建议各位同学尽量保持与本试验一致或类似的环境。
- 为什么要学会编译Linux内核?
  - 这是后续实验的基础。在后续实验中，我们会让大家通过阅读Linux源码、修改Linux源码、编写模块等方式理解一个真实的操作系统是怎么工作的。

其他友情提示

- **合理安排时间，强烈不建议在ddl前赶实验。**
- 本课程的实验实践性很强，请各位大胆尝试，适当变通，能完成实验任务即可。
- pdf上文本的复制有时候会丢失或者增加不必要的空格，有时候还会增加不必要的回车，有些指令一行写不下分成两行了，一些同学就漏了第二行。如果出了bug，建议各位先仔细确认自己输入的指令是否正确。要**逐字符**比对。每次输完指令之后，请观察一下指令的输出，检查一下这个输出是不是报错。**请在复制文档上的指令之前先理解一下指令的含义。** 我们在检查实验时会抽查提问指令的含义。
- 如果你想问“为什么PDF的复制容易出现问题”，请参考 **此文章** 。
- 如果同学们遇到了问题，请先查询在线文档。在线文档地址： **链接**

章节一 续·Linux、Shell指令教学 及 Makefile 编写和使用

1.0 环境准备

本次实验二三部分需要用到较多依赖工具，请使用包管理器安装以下包：

- qemu-system-x86 (Ubuntu 20.04以上) 或 qemu (Ubuntu 其他版本)
- git
- build-essential （里面包含了make/gcc/g++等，省去了单独安装的麻烦）
- libelf-dev
- xz-utils
- libssl-dev
- bc
- libncurses5-dev
- libncursesw5-dev

为了方便同学们复制粘贴，将所有的依赖放在这里：`qemu-system-x86 git build-essential libelf-dev xz-utils libssl-dev bc libncurses5-dev libncursesw5-dev`

1.1 Linux指令

1.1.1 echo

输出一个字符串。用法：`echo string`。

同样也可以输出一个文件的内容，如`echo file`将文件file的内容以纯文本形式读并输出。

经常在Shell脚本中使用该指令，以打印指定信息。

1.1.2 top

ps 命令可以一次性给出当前系统中进程状态，但使用此方式得到的信息缺乏时效性，并且，如果管理员需要实时监控进程运行情况，就必须不停地执行 ps 命令，这显然是缺乏效率的。

为此，Linux 提供了top命令。top 命令动态地持续监听进程地运行状态，与此同时，该命令还提供了一个交互界面，用户可以根据需要，人性化地定制自己的输出，进而更清楚地了进程的运行状态。直接输入`top`即可使用。

参数	含义
-d 秒数	指定 top 命令每隔几秒更新。默认是 3 秒。
-b	使用批处理模式输出。一般和 `-n` 选项合用，用于把 top 命令重定向到文件中。
-n 次数	指定 top 命令执行的次数。一般和 `-b` 选项合用。
-p 进程PID	仅查看指定 ID 的进程。
-u 用户名	只监听某个用户的进程。

在 top 命令的显示窗口中，还可以使用如下按键，进行以下交互操作：

按键 (注意大小写)	含义
? 或 h	显示交互模式的帮助。
P	按照 CPU 的使用率排序，默认就是此选项。
M	按照内存的使用率排序。
N	按照 PID 排序。
k	按照 PID 给予某个进程一个信号。一般用于中止某个进程，信号 9 是强制中止的信号。
q	退出 top 命令。

1.1.3 sleep

`sleep n`可以使当前终端暂停n秒。常见于shell脚本中，用于实现两条指令间的等待。

1.1.4 grep

筛选并高亮指定字符串。尝试在终端下运行`grep a`，它会等待用户输入。若输入一行不带a的字符串并按回车，它什么都不会输出。若输入多行字符串，其中某些行带a，它会筛选出带a的行，并将该行的a高亮后输出。

1.1.5 wc

英文全拼：wordcount

常见用法：`wc [-lw] [filename]`，用于统计字数。

参数	含义
-l	统计行数。
-w	统计字数。
filename	统计指定文件的行数/字数。若不指定，会从标准输入(C/C++的stdin/scanf/cin)处读取数据。

1.1.6 kill

发送指定的信号到相应进程。不指定信号将发送`SIGTERM(15)`终止指定进程。如果无法终止该程序可用`"-KILL"`参数，其发送的信号为`SIGKILL(9)`，将强制结束进程。使用ps命令或者jobs 命令可以查看进程号。root用户将影响用户的进程，非root用户只能影响自己的进程。

常见用法：`kill [信号] [PID]`

1.2 Shell指令

1.2.1 管道符 |

问题：我们想统计Linux下进程的数量，应该如何解决？

一个很麻烦的解决方法是：先`ps aux`输出所有进程的列表，然后复制它的输出，执行`wc -l`，将之前`ps`输出的内容粘贴到标准输入，传递给`wc`以统计`ps`输出的行数，即进程数量。有没有方法可以免去复制粘贴的麻烦？

管道符 `|` 可以将前面一个命令的标准输出（对应C/C++的stdout/printf/cout）传递给下一个命令，作为它的**标准输入**（对应C/C++的stdin/scanf/cin）。

- 例1：在终端中运行 `ps aux | wc -l` 可以显示所有进程的数量。
- 例2：先打开Linux下的firefox，然后在终端运行 `ps aux | grep firefox` 可以显示所有进程名含firefox的进程。
- 例3：接例2，在终端运行 `ps aux | grep firefox | wc -l` 可以显示所有进程名含firefox的进程的数量。

> 思考：`wc -l` 的作用是统计输出的行数。所以 `ps aux | wc -l` 统计出的数字真的是进程数量吗？

1.2.2 重定向符 >/>>/<

重定向符 `>` ，`>>` ：它可以把前面一个命令的标准输出保存到文件内。如果文件不存在，则会创建文件。 `>` 表示覆盖文件， `>>` 表示追加文件。

- 举例： `ps -aux > ps.txt` 可以把当前运行的所有进程信息输出到ps.txt。ps.txt的原有内容会被覆盖。
- 举例： `ps -aux >> ps.txt` 可以把当前运行的所有进程信息追加写到ps.txt。

重定向符 `<` ：可以将 `<` 前的指令的标准输入重定向为 `<` 后文件的内容。

- 举例： `wc -l < ps.txt` 可以把ps.txt中记录的信息作为命令的输入，即统计ps.txt中的行数。

1.2.3 分隔符 ;

子命令：每行命令可能由若干个子命令组成，各子命令由 `;` 分隔，这些子命令会被按序依次执行。

如： `ps -a; pwd; ls -a` 表示：先打印当前用户运行的进程，然后打印当前shell所在的目录，最后显示当前目录下所有文件。

1.3 Shell 脚本

考虑到往届在检查实验时，大多数时间都浪费在了敲指令上，希望大家可以学会如何编写Shell脚本。

1.3.1 Shell脚本的编写

Shell脚本类似于Windows的.bat批处理文件。一个最简单的Shell脚本长这样：

```
#!/bin/bash
第一条命令
第二条命令
第三条命令，以此类推
```

其中，第一行的 `#!/bin/bash` 是一个约定的标记，它告诉系统这个脚本需要什么解释器来执行，即使用哪一种 Shell。Ubuntu下默认的shell是bash。脚本一般命名为 `xxx.sh` 。

1.3.2 脚本的运行

运行Shell脚本有两种方式。

1. 通过执行 `sh xxx.sh` 来直接调用解释器运行脚本。其中，sh就是我们的Shell。这种方式运行的脚本，不需要在第一行指定解释器信息。
2. 将该脚本视为可执行程序。首先，保存脚本之后，要通过 `chmod +x xxx.sh` 给脚本赋予执行权限，然后直接 `./xxx.sh` 运行脚本。

> 注意：脚本的执行和在终端下执行这些语句是完全一致的，依然需要注意绝对路径和相对路径的问题。

举例：首先编译abc.cpp并输出一个名为aabbcc的二进制可执行文件，然后执行aabbcc，最后删掉aabbcc，上述操作可以使用如下脚本来实现：

```
#!/bin/bash
gcc -o aabbcc abc.cpp
./aabbcc
rm aabbcc
```

之后的实验会介绍更多Shell脚本的语法。

1.4 Makefile 的编写

> 你可以在阅读完本文 **章节二 实现一个Linux Shell** 后，再阅读这一节。

在本实验第一部分 2.3.17 （编译指令）中，我们提到了 `make` ，在编译内核和 busybox 时，我们也使用了 `make` 、 `make install` 等命令。为什么用一个 `make` 命令就能编译好庞大的内核呢？它又是怎么工作的呢？在这节，我们就将学习如何使用 Make 工具，以及如何编写 Makefile ，使得我们的程序可以通过 `make` 命令编译。

1.4.1 Make 介绍

什么是 Make 呢？Make 是一个用于自动化编译和构建的工具。在工程需要编译的文件较多时，每次都手输编译命令较为繁琐 。而 Make工具就能帮助我们将各个步骤的编译自动化，省去了每次输入编译命令的时间，也方便了将我们写好的程序分发给他人（其他人只要使用 `make` 命令就好啦）。

什么时候我们要使用 Make 呢？简略地说，当我们的项目编译过程比较复杂，诸如涉及文件较多、依赖关系较为复杂，或者需要生成多个不同的文件等时，我们就倾向于使用 Make，把杂乱的编译统一整理起来。Make 还能优化编译速度，当源代码的某一部分发生变化时， `Make` 只会重新编译那些直接或间接依赖于已更改部分的代码，而不是重新编译整个程序（在多次编译内核时，你会发现第二次及以后编译会比第一次快得多，就是这个原因 ）。

我们如何使用 Make 呢？Make 是通过读取名为 `Makefile` 的文件来工作的，这个文件包含了关于如何构建（编译）程序的规则。我们将在下一节介绍 Makefile 的编写方式。

总的来说，Make 是一个强大的工具，它可以帮助我们管理和自动化编译过程，从而提高效率，减少错误，并确保结果的一致性。

1.4.2 Makefile

> 该部分内容只包含了本次实验所必需的部分，你也可以参考 **跟我一起写Makefile - github.io** 或 **跟我一起写Makefile - Ubuntu中文** 来了解更多内容。

1.4.2.1 使用脚本自动化编译和其局限性

在1.3中，我们介绍了shell脚本基本用法，那么编译项目当然也可以使用shell脚本实现。假设我们有一个 C 文件 `hello.c` ，我们要将其编译为名为 `hello` 可执行文件，（正如 Lab1 的 2.3.17 节介绍的那样）可以编写以下脚本：

```
# make.sh
gcc -o hello hello.c
```

这个命令实际上读取了文件 `hello.c`，生成可执行文件 `hello`。从编译过程的角度考虑，我们可以说：生成 `hello` 的过程，是依赖于文件 `hello.c` 的。或者说，`hello` 是生成的**目标**，而 `hello.c` 是这个目标的**依赖项**。

很多时候，我们的项目不止一个源文件，要生成的目标文件也不止一个，例如我们可能有一个 `hello.h` 文件存放通用的函数，而希望把 `hello1.c` 和 `hello2.c`（都 include 了 `hello.h`）分别编译为 `hello1` 和 `hello2`，我们可能会写出这样一个脚本：

```
# make.sh
gcc -o hello1 hello1.c
gcc -o hello2 hello2.c
```

虽然我们的命令里没有出现 `hello.h`，但它被包含在两个源文件里，因此 `hello1` 实际依赖于 `hello1.c` 和 `hello.h`，而 `hello2` 则依赖于 `hello2.c` 和 `hello.h`。

一切似乎都还好，似乎用脚本也能完成自动化编译的任务。但实际上，我们的脚本存在一个关键的问题，那就是运行它时，`hello1` 和 `hello2` 一定会被同时重新编译。这看起来似乎没什么，但假设我们的 `hello1.c` 和 `hello2.c` 都很复杂，每一个都需要编译很长时间。而假设我们某次只修改了 `hello1.c`，而没有修改 `hello.h` 和 `hello2.c`，这时候我们用这个脚本，它仍然会重新编译 `hello1`，浪费了编译时间。

那我们能否拆开成两个脚本呢？大型项目里，要编译的文件可能有千千万万，我们不可能为每个文件（甚至每种组合）都写一个脚本，于是，只靠脚本管理编译过程就显得有些力不从心了。

实际上，当一个大项目编译时，可能会出现 `A` 依赖于 `B`，`B` 又依赖于 `C` 的情况。例如，多文件的项目中，为了避免重复编译，通常会把 `.c` 文件先编译为 `.o` 文件，再进行链接生成最终的可执行文件或库文件。这种情况，Make的作用就更加关键了。（本次实验不涉及这样的多级依赖，你可以参考 **附录3** 和节开头的参考链接了解更多 C编译相关知识。）

1.3.2.2 Makefile 基础规则

既然脚本没法解决所有问题，那 `make` 又是怎么解决的呢？正如前文所说，Make 是通过读取名为 `Makefile` 的文件来工作的。而 `Makefile` 里实际就记录了整个项目需要生成的所有文件的依赖关系，和生成规则。

我们先粗略的看一下 Makefile 的结构，Makefile 由以下格式的块构成，每个块代表一个或多个目标的生成方式和依赖。

```
[目标] ... : [依赖项] ...
    [生成命令]
    ...
    ...
```

- `[目标]`

  代表一个要生成的目标文件名，有时候也可以只是一个标签不对应真实文件（见后续介绍）.

- `[依赖项]`

  生成 `[目标]` 的依赖项，相当于编译所需要的“输入文件”。

- `[生成命令]`

  生成 `[目标]` 所需要运行的 Shell 指令。

例如，对上一节提到的 `hello.c` 编译为 `hello` 的依赖关系，可以用以下 `Makefile` 来描述（`#` 开头的行是 Makefile 里的注释）：

```
# hello 依赖于 hello.c ，通过 gcc -o hello hello.c 生成。
hello: hello.c
    gcc -o hello hello.c
```

当我们运行 `make` 命令时，make会尝试读取当前目录名为 `Makefile` 的文件，并按其中描述运行命令。（显然，上面的例子中，它会运行 `gcc -o hello hello.c` 。）

一个 `Makefile` 里可以有多个目标，一个目标可以有多个依赖项（也可以没有依赖），如之前 `hello1` 和 `hello2` 编译的依赖关系可表示为：

```
hello1: hello1.c hello.h
    gcc -o hello1 hello1.c

hello2: hello2.c hello.h
    gcc -o hello2 hello2.c
```

当我们运行 `make` 时，Make 会尝试生成 `Makefile` 中第一个目标，即 `hello1` 。如果要生成其它目标，可以在 `make` 后面添加目标名，如 `make hello2` 。 `make` 运行时，会自动检查生成目标是否已经存在，如果存在，会自动判断目标和依赖项的修改时间，如果目标生成之后，依赖项有过修改， `make` 会重新生成目标，否则，说明上次生成后，依赖项没有被修改过，这意味着没必要重新编译， `make` 将什么都不做。这样，Make 就节省了宝贵的编译时间。 （在多级依赖的场合，如果依赖项不存在， `make` 会尝试首先生成依赖项，这也非常符合直觉。）

1.3.2.3 伪目标和清空目录的规则

编译时，除了生成某些东西，有时我们还想做些不会生成对应文件的任务。例如，我们有时可能想删除所有之前编译的结果（如 `hello1` 和 `hello2` ）。我们可以这样写：

```
clean:
    rm -rf hello1 hello2
```

这个规则中 `[目标]` 为clean， `[依赖项]` 为空， `[生成命令]` 为 `rm -f ...` 。但是，我们并不打算实际生成一个名为 `clean` 的文件。这时候，我们称类似 `clean` 这样的目标为“**伪目标**”。“伪目标”并不是一个文件，只是一个标签，由于“伪目标”不是文件，所以 make 无法通过它的依赖关系和决定它是否要执行。我们只有通过显式地指明这个“目标”才能让其生效（以 `make clean` 来使用该目标）。

为了避免和文件重名导致的错误（例如我们目录下真的有名为 `clean` 的文件），我们可以使用一个特殊的标记 `.PHONY` 来显式地指明一个目标是“伪目标”，向 Make 说明，不管是否有这个文件，这个目标就是“伪目标”。

```
.PHONY : clean
```

只要有这个声明，不管是否有 `clean` 文件，以及文件修改时间如何， `make clean` 都会执行对应的命令。于是整个过程可以这样写：

```
.PHONY : clean
clean:
    rm -rf hello1 hello2
```

`.PHONY` 后可以有多个目标名，也可以写在伪目标的后面。例如，我们可能想用一个命令 `make` 同时生成 `hello1` 和 `hello2` ，可以写一个伪目标 `all` ，放在 `makefile` 的开头：

```
all: hello1 hello2

hello1: hello1.c hello.h
    gcc -o hello1 hello1.c

hello2: hello2.c hello.h
    gcc -o hello2 hello2.c

.PHONY : all clean

clean:
    rm -rf hello1 hello2
```

注意这实际上是个多级依赖，`all` 依赖于 `hello1` 和 `hello2`，所以这两个依赖不存在时，`make` 会尝试生成它们。

在我们的实验中，需要使用伪目标 `test` 来自动运行测试，方法也类似如此，大家可以自行尝试。

1.3.2.4 Makefile的文件名

默认情况下，make命令会在当前目录下按顺序寻找名为 `GNUmakefile`、`makefile` 和 `Makefile` 的文件。在这三个文件名中，最好使用 `Makefile` 这个文件名，因为这个文件名在排序上靠近其它比较重要的文件，比如 `README`。最好不要用 `GNUmakefile`，因为这个文件名只能由GNU `make`，其它版本的 `make` 无法识别，但是基本上来说，大多数的 `make` 都支持 `makefile` 和 `Makefile` 这两种默认文件名。

章节二 实现一个Linux Shell

> 注意：
>
> - 本部分的主要目的是锻炼大家**使用** Linux系统调用编写程序的能力，不会涉及编写一个完整的系统调用。
> - 本部分主要是代码填空，费力的部分已由助教完成，代码量不大，大家不必恐慌。**需要大家完成的代码已经用注释 `TODO:` 标记，可以通过搜索轻松找到**，使用支持TODO高亮编辑器（如vscode装TODO highlight插件）的同学也可以通过高亮找到要添加内容的地方。

2.1 Shell的基本原理

我们在各种操作系统中会遇到各式各样的用于输入命令、基于文字与系统交互的终端，Windows下的cmd和powershell、Unix下的sh、bash和zsh等等，这些都是不同的shell，我们可以看到它们的共同点——基于“命令” (command) 与系统交互，完成相应的工作。

Shell处理命令的方式很好理解，每条输入的“命令”其实都对应着一个可执行文件，例如我们输入 `top` 时，**shell程序会创建一个新进程并在新进程中运行可执行文件。具体而言，shell程序在特定的文件夹中搜索名为"top"的可执行文件，搜索到之后运行该可执行文件，并将进程的输出定向到指定的位置(如终端设备)**，这一过程也是我们本次实验要实现的内容。具体来讲：

- **创建新进程：**用到我们课上讲过的 `fork()` 。
- **运行可执行文件：**使用课上讲的 `exec()` 系列函数。
  - exec开头的函数有很多，如 `execvp` 、 `execl` 等，可以使用man查看不同exec函数的区别及使用方法，最后选取合适的函数。
  - exec中需要指定可执行文件，因为用户的命令给出的只有文件名本身，没有给出文件的所在位置，所以在系统中通常会定义一个环境变量PATH来记录一组文件夹，所有命令对应的可执行文件通常存在于某个文件夹下，找不到则返回不存在。可以使用 `echo $PATH` 来查看你现在使用的shell都会去哪些文件夹下查找可执行程序。
- **结果输出：**命令默认会从STDIN_FILENO (默认值为0) 中读输入，并输出到STDOUT_FILENO (默认值为1) 中。在shell中也会有其他的输入来源和输出目标，如将一个命令的输出作为另一个命令的输入，在这种场合下就需要进行进程间的通信，比如采用我们课上讲过的管道pipe，执行两个命令的进程一个在写端输出内容，一个在读端读取输入。此外，命令还可以设置从文件输入和输出到文件，比如将执行命令程序中的STDOUT_FILENO使用文件描述符覆盖后，命令的执行结果就输出到文件中。

2.2 Shell内建指令

`cd` 命令可以用来切换shell的当前目录。但需要指出的是，不同于其他命令（比如 `ls` ，我们可以在/bin下面找到一个名为 `ls` 的可执行文件）， `cd` 命令其实是一个shell内置指令。由于子进程无法修改父进程的参数，所以若不使用内建命令而是fork出一个子进程并且在子进程中exec一个 `cd` 程序，因为子进程执行结束后会回到了父shell环境，而父shell的路径根本没有被改变，最终无法得到期望的结果。同理，不仅是 `cd` ，**改变当前shell的参数（如 `source` 命令、 `exit` 命令、 `kill` 命令）基本都是由shell内建命令实现的** 。

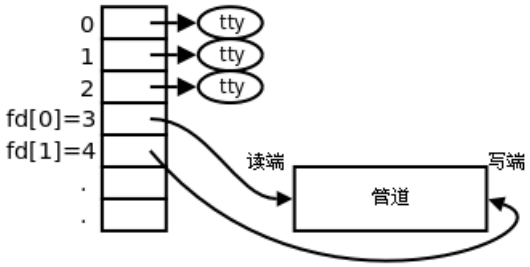
2.3 有关管道的背景知识

“一切皆文件”是Unix/Linux的基本哲学之一。普通文件、目录、I/O设备等在Unix/Linux都被当做文件来对待。虽然他们的类型不同，但是linux系统为它们提供了一套统一的操作接口，即文件的open/read/write/close等。当我们调用Linux的系统调用打开一个文件时，系统会返回一个文件描述符，每个文件描述符与一个打开的文件唯一对应。 之后我们可以通过文件描述符来对文件进行操作。管道也是一样，我们可以通过类似文件的read/write操作来对管道进行读写。为便于理解，本次实验使用匿名管道。匿名管道具有以下特点：

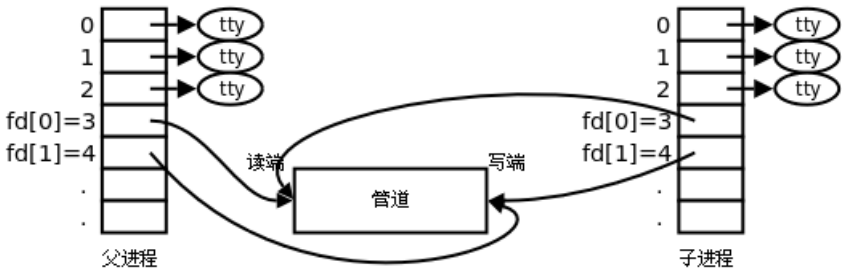
1. 只能用于父子进程等有血缘关系的进程；
2. 写端不关闭，并且不写，读端读完，继续等待，此时阻塞，直到有数据写入才继续（就好比你的C程序在scanf，但你一直什么都不输入，程序会停住）；
    - 尤其地，假如一条管道有多个写端，那么只有在所有写端都关闭之后（管道的引用数降为0），读端才会解除阻塞状态。
3. 读端不关闭，并且不读，写端写满管道buffer，此时阻塞，直到管道有空位才继续；
4. 读端关闭，写端在写，那么写进程收到信号SIGPIPE，通常导致进程异常中止；
5. 写端关闭，读端在读，那么读端在读完之后再读会返回0；
6. 匿名管道的通信通常是一次性的，如果需要反复通信，可以使用命名管道。

一般来说，匿名管道的使用方法是：

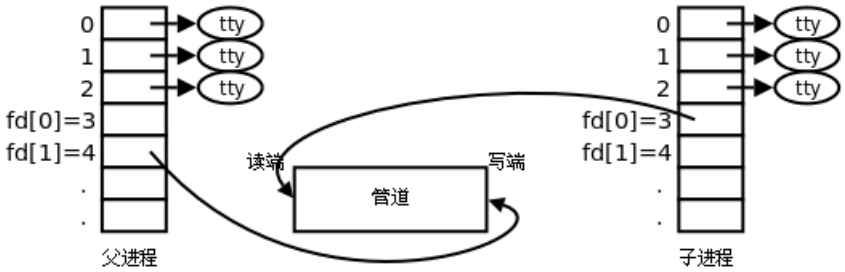
1. 父进程创建管道



2. 父进程fork出子进程



3. 父进程关闭fd[0]，子进程关闭fd[1]



- 首先，父进程调用pipe函数创建一个匿名管道。pipe的函数原型是 `int pipe(int pipefd[2])` 。我们传入一个长为2的一维数组 `pipefd` ，Linux会将 `pipefd[0]` 设为**读端**的文件描述符，并将 `pipefd[1]` 设为**写端**的文件描述符。（注：此时管道已被打开，相当于已调用了 `open` 函数打开文件）但是需要注意：此时管道的读端和写端接在了同一个进程上。如果你此时往 `pipefd[1]` 里写入数据，这些数据可以从 `pipefd[0]` 里读出来。不过这种“我传我自己”(原地tp) 通常没什么意义，我们接下来要把管道应用于进程通信。

- 其次，使用 `fork` 函数创建一个子进程。 `fork` 完成之后，数组 `pipefd` 也会被复制。此时，子进程也拥有了对管道的控制权。若目的是父进程向子进程发送数据，那么父进程就是写端，子进程就是读端。我们应该把父进程的读端关闭，把子进程的写端关闭，进而便于数据从父进程流向子进程。
    - 如果不关闭子进程的写端，子进程会一直等待（参考2.3.2）。
- 因为匿名管道是单向的，所以如果想实现从子进程向父进程发送数据，就得另开一个管道。
- 父子进程调用 `write` 函数和 `read` 函数写入、读出数据。
    - `write` 函数的原型是： `ssize_t write(int fd, const void * buf, size_t count);`
    - `read` 函数的原型是： `ssize_t read(int fd, void * buf, size_t count);`
    - 如果你要向管道里读写数据，那么这里的 `fd` 就是上面的 `pipefd[0]` 或 `pipefd[1]` 。
    - 这两个函数的使用方法在此不多赘述。如有疑问，可以百度。

> **注意：如果你的数组越了界，或在read/write的时候count的值比buf的大小更大，则会出现很多奇怪的错误（如段错误（Segmentation Fault）、其他数组的值被改变、输出其他数组的值或一段乱码（注意，烫烫烫是Visual C的特性，Linux下没有烫烫烫）等）。提问前请先排查是否出现了此类问题。**

2.4 输入/输出的重定向

> 注：本节描述的是如何将一个程序产生的标准输出转移到其他非标准输出的地方，不特指 `>` 和 `>>` 符号。

在使用 `|` ，`>`，`>>` `<` 时，我们需要将程序输出的内容重定向为文件输出或其他程序的输入。在本次实验中，为方便起见，我们将shell作为重定向的中转站。

- 当出现前三种符号时，我们需要把前一指令的标准输出重定向为管道，让父进程（即shell）截获它的标准输出，然后由shell决定将前一指令的输出转发到下一进程、文件或标准输出（即屏幕）。
- 当出现 `|` 和 `<` 时，我们需要把后一指令的标准输入重定向为管道，让父进程（即shell）把前一进程被截获的标准输出/指定文件读出的内容通过管道发给后一进程。

重定向使用 `dup2` 系统调用完成。其原型为： `int dup2(int oldfd, int newfd);` 。该函数相当于将 `newfd` 标识符变成 `oldfd` 的一个拷贝，与 `newfd` 相关的输入/输出都会重定向到 `oldfd` 中。如果 `newfd` 之前已被打开，则先将其关闭。举例：下述程序在屏幕上没有输出，而在文件输出"hello!goodbye!"。屏幕上不会出现"hello!"和"goodbye"。

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>

int main(void)
{
    int fd;

    fd = open("./test.txt", O_RDWR | O_CREAT | O_TRUNC, 0666);

    // 此代码运行在dup2之前，本应显示到屏幕，但实际上是暂时放到屏幕输出缓冲区
    printf("hello!");

    // 程序默认是输出到STDOUT_FILENO即1中，现在我们让1重定向到fd。
    // 也就是将标准输出的内容重定向到文件，因此屏幕输出缓冲区以及后面的printf
    // 语句本应输出到屏幕（即标准输出，fd为STDOUT_FILENO）的内容重定向到文件中。
    dup2(fd, 1);
```

```
    printf("goodbye!\n");
    return 0;
}
```

> 易混淆的地方：向文件/管道内写入数据实际是程序的一个输出过程。

2.5 一些shell命令的例子和结果分析

> 本部分旨在帮助想要进一步理解真实shell行为的同学，希望下述示例及结果分析可以为同学们带来启发。

这一节中，我们给出一些shell命令的例子，并分析命令的输出结果，让大家能够更直观的理解shell的运行。**我们并不要求实验中实现的shell行为和真实的shell完全一致，但你自己需要能够清楚地解释自己实现的处理逻辑。**但在实现shell时，可以参考真实shell的这些行为。这些实验都是在bash下测试的，大家也可以自己在Ubuntu等的终端中重复这些实验。在这一节的代码中，每行开头为 `$` 的，是输入的shell命令，剩下的行是shell的输出结果。如：

```
$ cd /bin
$ pwd
/bin
```

表示在shell中了先运行了 `cd /bin` ，然后运行了 `pwd` ，第一条命令没有任何输出，第二条命令输出为 `/bin` 。 `pwd` 命令会显示shell的当前目录，我们先用 `cd` 命令进入了 `/bin` 目录，所以 `pwd` 输出 `/bin` 。

2.5.1 多个子命令

由 `;` 分隔的多个子命令，和在多行中依次运行这些子命令效果相同。

```
$ cd /bin ; pwd
/bin
$ echo hello ; echo my ; echo shell
hello
my
shell
```

`echo` 会将它的参数打印到标准输出，如 `echo hello world` 会输出 `hello world` 。这个实验用 `;` 分隔了三个shell命令，结果和在三行中分别运行这些命令一致。

2.5.2 管道符

这个实验展示了shell处理管道时的行为，相同的命令在管道中行为可能和单独运行时不同。

```
$ echo hello | echo my | echo shell
shell
```

如上，虽然管道中的上一条命令的输出被重定向至下一条命令的输入，但是因为 `echo` 命令本身不接受输入，所以前两个 `echo` 的结果不会显示。如果实验检查中，你提供了这样的测试样例，是不能证明正确实现了管道的。

```
$ cd /bin ; pwd
/bin
$ cd /etc | pwd
/bin
```

如上，`cd /etc` 并没有改变shell的当前目录，这是因为**管道中的内置命令也是在新的子进程中运行的**，所以不会改变当前进程（shell）的状态。而在**不包含管道的命令中，内置命令在shell父进程中运行，外部命令在子进程中运行**，所以，不包含管道的内置命令能够改变当前进程（shell）的状态。可以用 `type` 命令检查命令是否是内部命令。

```
$ type cd
cd is a shell builtin
$ type cat
cat is hashed (/bin/cat)
```

说明 `cd` 是内置命令，而 `cat` 则是调用 `/bin/cat` 的外置命令。接下来我们测试管道中命令的运行顺序。

```
$ sleep 0.02 | sleep 0.02 | sleep 0.02 | ps | grep sleep
15840 tty1     00:00:00 sleep
$ sleep 0.02 | sleep 0.02 | sleep 0.02 | ps | grep sleep
15845 tty1     00:00:00 sleep
15846 tty1     00:00:00 sleep
15847 tty1     00:00:00 sleep
```

可以看到，运行了两次相同的命令，却得到了不同的结果，多次运行该命令，每次输出的sleep行数不同，从0行到3行都有可能（根据电脑速度和核数，可能需要将sleep后面的数字增大或缩小来重复该实验）。这说明**管道中各个命令是并行执行的**，`ps` 命令运行时，前面的 `sleep` 命令可能执行结束，也有可能仍在执行。

2.5.3 重定向

以下命令均在bash运行，如果使用zsh可能会产生不同结果，详见附录1

```
$ cd /tmp ; mkdir test ; cd test
$ echo hello > a >> b > c
$ ls
a  b  c
$ cat a
$ cat b
$ cat c
hello
```

当一个命令中出现多个输出重定向时，虽然所有文件都会被建立，但是只有最后一个文件会真正被写入命令的输出。

```
$ cd /tmp ; mkdir test ; cd test
$ echo hello > out | grep hello
$ cat out
hello
```

当输出重定向和管道符同时使用时，命令会将结果输出到文件中，而管道中的下一个命令将接收不到任何字符。

```
$ cd /tmp ; mkdir test ; cd test
$ > out2 echo hello ; cat out2
hello
```

重定向符可以写在命令前。

2.6 总结：本次实验中shell执行命令的流程

- 第一步：打印命令提示符（类似shell ->）。
- 第二步：把分隔符 `;` 连接的各条命令分割开。（多命令选做内容）
- 第三步：对于单条命令，把管道符 `|` 连接的各部分分割开。
- 第四步：如果命令为单一命令没有管道，先根据命令设置标准输入和标准输出的重定向（重定向选做内容）；再检查是否是shell内置指令：是则处理内置指令后进入下一个循环；如果不是，则fork出一个子进程，然后在fork出的子进程中exec运行命令，等待运行结束。（如果不fork直接exec，会怎么样？）
- 第五步：如果只有一个管道，创建一个管道，并将子进程1的**标准输出重定向到管道写端**，然后fork出一个子进程，根据命令重新设置标准输入和标准输出的重定向（重定向选做内容），在子进程中先检查是否为内置指令，是则处理内置指令，否则exec运行命令；子进程2的**标准输入重定向到管道读端**，同子进程1的运行思路。（注：2.4的样例分析中我们得出，管道的多个命令之间，虽然某个命令可能会因为等待前一个进程的输出而阻塞，但整体是没有顺序执行的，即并发执行。所以我们为了让多个内置指令可以并发，需要在fork出子进程后才执行内置指令）
- 第六步：如果有多个管道，参考第三步，n个进程创建n-1个管道，每次将子进程的**标准输出重定向到管道写端**，父进程保存**对应管道的读端**（上一个子进程向管道写入的内容），并使得下一个进程的**标准输入重定向到保存的读端**，直到最后一个进程使用标准输出将结果打印到终端。（多管道选做内容）
- 第七步：根据第二步结果确定是否有剩余命令未执行，如果有，返回第三步执行（多命令选做内容）；否则进入下一步。（分隔符多命令和管道连接的多命令实现方式上有什么区别？为什么？）
- 第八步：打印新的命令提示符，进入下一轮循环。

2.7 任务目标

代码填空实现一个shell。这个shell不要求在qemu下运行。功能包括：

必做部分

- 实现运行单条命令（非shell内置命令）。
- 支持一条命令中有单个管道符 `|` 。
- 实现exit, cd, kill三个shell内置指令。
- 在shell上能显示当前所在的目录。如：{ustc}shell: [/home/ustc/exp2/] > （后面是用户的输入）

> kill内置命令：涉及SIGTERM、SIGKILL等多种信号量，命令执行方式可以**自定义**，输入的信号量用编号表示即可，例如：kill 1234 9，表示强制终止PID为1234的进程（9为SIGKILL信号量编号）；kill 1234，表示以默认的方式（SIGTERM信号量）终止PID为1234的进程。注意(kill 1234 9只是自定义的命令执行方法，bash中kill的执行方式见1.1.6)

Makefile部分

补全助教提供的 Makefile 文件，实现自动编译和测试。要求包括：

- 可以正确编译 `testsh` ， `simple_shell` 。
- 可以使用 `test` 为目标使用 `testsh` 测试 `simple_shell` 。

二选一部分

- 实现子命令符 `;` 和重定向符 `>` , `>>` , `<` 。
- 支持一条命令中有多个管道符 `|` 。如果你确信你的多管道功能可以正常实现单管道功能，代码填空里的单管道可以不做。

其他说明

- 实验文件中提供了一个testsh.c作为本次shell实验的测试脚本，testdata作为测试脚本所需要的文件(不可修改或删除，否则测试样例不能通过)，测试脚本中前7条为必做实验部分，8-12为选做1，13-14测试选做2，使用方法为先编译出可执行文件，再 `testsh shell_name` ，你可以在testsh中得到本次实验的部分提示。(详细测试样例说明，请查看附录2)
- 在实现cd命令时，不要求实现 `cd -` , `cd ~` 等方式
- 我们使用的是Linux系统调用，请不要尝试在Windows下运行自己写的shell程序。那是不可能的。
- 需要自行设计测试样例以验证你的实现是正确的。如测试单管道时使用 `ps aux | wc -l` ，与自带的shell输出结果进行比较。
- 不限制分隔符、管道符、重定向符的符号优先级。你可以参考我们代码框架中实现、提示的优先级。
- 不要求实现不加空格的重定向符，不要求实现使用'~'表示家目录。
- 请尽量使用系统调用完成实验，不准用system函数。
- 我们提供了本次实验使用的系统调用API的范围，请同学们自行查询它们的使用方法。你在实验中可能会用到它们中的一部分。 你也可以使用不属于本表的系统调用。
    - read/write
    - open/close
    - pipe
    - dup/dup2
    - getpid/getcwd
    - fork/vfork/clone/wait/exec
    - mkdir/rmdir/chdir
    - exit/kill
    - shutdown/reboot
    - chmod/chown
- 如果你想问如何编译、运行自己写的代码，请参考lab1的2.3.17和2.2.3,以及本实验的1.3.2。
- 得分细则见文档3.1。

章节三： 检查要求

本部分实验无实验报告。

本部分实验共4分。

3.1 “实现一个Shell”部分检查标准 (4')

- 支持基本的单条命令运行、支持两条命令间的管道 `|` 、内建命令（只要求 `cd` / `exit` / `kill` ），得2分。
- 选做：
  - 支持多条命令间的管道 `|` 操作，得1分。
  - 支持重定向符 `>` 、 `>>` 、 `<` 和分号 `;` ，得1分。
- 你需要流畅地说明你是如何实现实验要求的（即，现场讲解代码），**并且使用make命令展示测试结果**。本部分共1分。若未能完成任何一个功能，则该部分分数无效。

附录

1. Unix不同Shell的区别和联系

我们在2.1中提到了Shell的基本原理，其中提到了Unix中不同的Shell,本部分先说明不同shell的区别，最后讲解为什么在重定向中Bash和Zsh的输出并不完全相同

1.1 基本shell

Unix系统中主要要两种类型的shell:“Bourne Shell”和“C Shell”。并且这两种类别的shell分别有几种子类别

Bourne Shell 的子类别有：

- **Bourne Shell (sh)**
- **KornShell (ksh)**
- **Bourne Again Shell (Bash)**

C-Type shells的子类别有:

- **C Shell (csh)**
- **TENEX/TOPS C shell (tcsh)**

其中Bourne-Again Shell即Bash，是用于取代默认Bourne Shell 的 Unix shell。它在各方面都融合了Korn和C Shell的功能。这意味着用户可以获得 Bourne Shell 的语法兼容性和跨平台性，并在此基础上使用这些其他 shell 的功能进行扩展。

> 查看本机上已经安装的shell可使用 `cat /etc/shells` 来查看，安装使用和修改默认shell方法自行查询

1.2 当前热门的Shell

1. Fish(Friendly Interactive Shell)

   Fish是2005年发布的开源Shell，专门开发为易于使用，并具有开箱即用功能的shell。其风格化的颜色编码对新程序员也很有帮助，因为它突出了语法，使其更易于阅读。Fish Shell 的功能包括制表符补全、语法突出显示、自动补全建议、可搜索的命令历史记录等。(**该Shell的语法与Bash差别较大，有些bash脚本可能无法正常使用，需要修改为特定语法**)

   > Ubuntu安装方法：
   >
   > ```
   > $ sudo apt install fish
   > $ fish
   > ```

2. Zsh(Z shell)

   Z-shell 被设计为现代创新和交互式外壳。Zsh 在其他 Unix 和开源 Linux shell （包括 tcsh、ksh、Bash 等）之上提供了一组独特的功能。这个开源 shell 易于使用、可定制，提供拼写检查、自动完成和其他生产力功能。

   > Ubuntu安装方法：
   >
   > ```
   > $ sudo apt install zsh
   > $ zsh
   > ```

1.3 重定向中zsh和bash的表现不同的原因

在文档2.5.3节中阐述了shell中重定向的使用方法，如果使用bash，输出结果如下所示

```
$ cd /tmp ; mkdir test ; cd test
$ echo hello > a >> b > c
$ ls
a  b  c
$ cat a
$ cat b
$ cat c
hello
```

但是如果使用zsh，输出结果则如下所示

```
$ cd /tmp ; mkdir test ; cd test
$ echo hello > a >> b > c
$ ls
a  b  c
$ cat a
hello
$ cat b
hello
$ cat c
hello
```

首先，先看一个简单的例子

```
$ echo hello world > a
$ cat a
hello world
```

该例子即使用 `>` 重定向符号将echo的标准输出重定向到文件a，我们可以查看 `/proc/PID_OF_PROCESS/fd` 来查看具体情况，由于echo命令执行速度较快，使用如下例子

```
$ cat > a

# Open another Shell
$ ls -l /proc/$(pidof cat)/fd
total 0
lrwx------ 1 ridx ridx 64 Mar 28 15:31 0 -> /dev/pts/0
l-wx------ 1 ridx ridx 64 Mar 28 15:31 1 -> ~/a
lrwx------ 1 ridx ridx 64 Mar 28 15:31 2 -> /dev/pts/0
```

cat等待用户输入后重定向给文件a直到Ctrl-D结束，可以看到cat的标准输出已经指向a。0 是标准输入（即用户输入端），1 是标准输出（即正常情况的输出端），2 是错误输出（即异常情况的输出端），想要重定向错误输出端，可以使用 `2>`，本实验不要求实现该方法。

```
#Use Bash Shell
$ cat > a >> b >c

# Open another Shell
$ ls -l /proc/$(pidof cat)/fd
total 0
lrwx------ 1 ridx ridx 64 Mar 28 15:44 0 -> /dev/pts/0
l-wx------ 1 ridx ridx 64 Mar 28 15:44 1 -> ~/c
lrwx------ 1 ridx ridx 64 Mar 28 15:44 2 -> /dev/pts/0
```

回到最开始的重定向例子中，由于例子中echo命令过快，我们使用cat来代替.使用bash执行命令 `cat > a >> b > c` 。可以看到cat标准输出只有c，所以cat程序只往文件c中写入数据

```
#Use Zsh
$ cat > a >> b >c

# Open another Shell
$ pstree -p | grep cat
|-sh(3365094)---node(3365156)-+-node(3365181)-+-zsh(3416851)---cat(3416921)---
zsh(3416922)
$ ls -l /proc/3416921/fd
total 0
lrwx------ 1 ridx ridx 64 Mar 28 15:48 0 -> /dev/pts/0
l-wx------ 1 ridx ridx 64 Mar 28 15:48 1 -> 'pipe:[457492940]'
lrwx------ 1 ridx ridx 64 Mar 28 15:48 2 -> /dev/pts/0
$ ls -l /proc/3416922/fd
total 0
l-wx------ 1 ridx ridx 64 Mar 28 15:50 11 -> ~/a
l-wx------ 1 ridx ridx 64 Mar 28 15:50 16 -> ~/b
l-wx------ 1 ridx ridx 64 Mar 28 15:50 18 -> ~/c
lr-x------ 1 ridx ridx 64 Mar 28 15:50 17 -> 'pipe:[457492940]'
```

可以看到 cat 的标准输出是重定向到管道，管道对面是 zsh 进程，然后 zsh 打开了那三个文件。实际将内容写入文件的是 zsh，而不是 cat。所以在zsh中 `echo hello > a >> b > c` ，三个文件均更新.

> 在fish中，重定向的表现和bash相同，不过fish无法使用$(...)语法，感兴趣可以直接使用 `ls -l /proc/(pidof cat)/fd` 或者通过pstree查看cat进程号手动验证

2. shell实验测试样例详细说明

2.1编译运行

testsh.c为标准c程序代码，编译测试请参考请参考本实验第一部分的2.3.17和2.2.3。测试程序接受一个字符串参数 `shell_name` ,请先编译出自己的shell程序，然后编译testsh，运行命令请使用 `testsh shell_name` 。(当然我们鼓励使用makefile或者sh脚本来编译运行测试文件)

如果同时完成选做1和选做2，运行结果如下

```
kill: PASS
cd: PASS
current path: PASS
simple echo: PASS
simple grep: PASS
two commands: PASS
simple pipe: PASS
more test: PASS
output redirection(use >): PASS
output redirection(use >>): PASS
input redirection: PASS
both redirections: PASS
pipe and redirection: PASS
multipipe: PASS
two commands with pipes: PASS
passed all tests
```

2.2testsh详细说明

测试程序中，每一个测试样例为一个函数，下面表格为所有的测试样例。testsh使用的前提是你的shell先完成**exit**命令的功能

函数名	测试样例/功能	命令
t0	kill	`kill [pid]`
t1	cd	`cd /sys/class/net`
t2	current path	查看shell是否打印了当前目录
t3	simple echo	`echo hello goodbye`
t4	simple grep	`grep professor testdata`
t5	two commands	`echo x\necho goodbye\n`
t6	simple pipe	`cat file | cat`
t7	more test	`ps | grep ps`
t8	output redirection(use >)	`echo data > file`
t9	output redirection(use >>)	`echo data > file;echo data >> file`
t10	input redirection	`cat < file`
t11	both redirections	`grep USTC < testdata > testsh.out`
t12	pipe and redirections	`grep system < testdata | wc > testsh.out`
t13	multipipe	`cat testdata | grep system | wc -l`
t14	two commands with pipes	`echo hello 2025 | grep hello ; echo nihao 2025 | grep 2025`

2.3可能出现的问题

1. 运行该程序时，可能会出现 `unexpected wait() return` 的报错，如果出现该报错，请尝试再次运行程序
2. 在测试 `simple pipe` 功能时可能会有卡顿，属于正常情况，请耐心等待
3. 本程序必须要求你的simple_shell具有exit功能，否则无法正常工作
4. 如果你的ßshell无关信息输出过多，可能会有 `testsh: saw expected output, but too much else as well` 出现，请调整代码
5. 如果shell测试失败，可能会产生临时乱码文件，属于正常现象，测试完毕后把临时文件删除即可
6. 本程序如果出现各种问题，请及时在QQ群/文档/ 提供问题描述或建议

3.C语言程序编译流程

1. 预处理（Preprocessing）
   预处理是编译过程的第一步，主要使用预处理器来处理源代码文件。在预处理阶段，源代码中的宏定义、条件编译指令以及头文件的包含等将被展开和处理，生成经过处理的中间代码。预处理的结果是一个经过宏定义替换、头文件包含等操作后的中间代码文件。
2. 编译（Compiling）
   编译是将预处理之后的中间代码翻译成汇编代码的过程。编译器会将中间代码文件转化为与特定平台相关的汇编语言代码文件。在编译阶段，进行语法分析、语义分析等操作，生成与特定平台相关的汇编代码。
3. 汇编（Assembling）
   汇编是将汇编语言代码翻译为机器代码的过程。汇编器将汇编代码转换为目标文件，其中包含了与特定平台相关的机器指令和数据。目标文件中包含了汇编语言代码转换而成的机器码。

4. 链接（Linking）
   链接是将各个目标文件（包括自己编写的文件和库文件）合并为一个可执行文件的过程。链接器将目标文件中的符号解析为地址，并将它们连接到最终的可执行文件中。链接过程包括符号解析、地址重定位等步骤，确保各个模块能够正确地相互调用。
5. 可执行文件（Executable）
   最终的输出是一个可执行文件，其中包含了所有必要的指令和数据，可以在特定平台上运行。这个可执行文件经过编译链接过程，包含了源代码的所有功能和逻辑，可以被操作系统加载并执行。