

20220913 常用的基本命令&Shell 语言编程

命令和指令

硬件给我们实现指令，软件是程序，把指令一条条排下来

What's shell?

Under Unix, KT met with the problem of too small storage to run a programme..

So, develop *Language Shell* to get several small programs work together.

Shell shows the characteristic of 调度 and manage, also shows 复用性.

- Explanatory language
- bash is a interpreter under Linux for Shell

```
jonas@ubuntu:~/Desktop$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/bash
/usr/bin/bash
/bin/rbash
/usr/bin/rbash
/bin/dash
/usr/bin/dash
```

ls

- ls -a 列出目录所有文件，包括隐藏文件；
- ls -l 列出除了文件名之外的权限、所有者、文件大小等。

```
drwxr-xr-x 2 jonas jonas 4096 Oct 21 2021 Pictures
-rwxrwxr-x 1 jonas jonas 40 Sep 12 23:11 prog
```

Pictures 是一个目录，owner 可读可写可执行，群组可读可执行，其他人只能执行，链接数为 2，owner 为 jonas，群组为 jonas，容量 4096Byte，创建时间为 2021 年 10 月 21.

文档权限信息	共 10 位： 第一位：文件类型 d 目录 - 文件 l 链接文件 后九位：依次对应三种身份的权限 身份顺序 owner,group,others 权限顺序 readable, writable, executable
链接数	表示有多少个文件链接到 inode
拥有者	
所属群组	
文件容量大小（B）	单位是字节
文件最后修改时间	不是创建时间
文件名称	.开头的是隐藏文件

cat

显示文件内容;

cat > filename 创建 filename 文件;

cat file1 file2 > file 将文件 file1 file2 合并成一个文件

cd aka change directory

更改当前目录。

cd / 到系统根目录

cd ~ 到当前用户主目录

cd - 到上一次工作路径，往往用于在工作路径下需要查看 home 下的某个配置文件后回来

cd .. 到上一级目录

pwd

当前工作目录路径

mkdir

mkdir -p 创建多级目录; mkdir -p tmp/bin/build

cp

cp 源文件 目的文件

cp -r 当目录底下有内容，需要-r 参数复制整个目录

cp -i 提示信息

cp -a 复制的文件与原文件时间一致

mv

移动和改名

ps aka process status

PID	TTY	TIME	CMD
2596	pts/0	00:00:00	bash
5859	pts/0	00:00:00	ps

引用变量

variable=123456

Export variable

Echo \$variable

环境变量？

一些系统设置

Path 环境变量：执行命令时在 path 下一个个找。

Exp: ls /bin/ls, ls /bin/ps

```
254 variable=123456
255 export variable
256 echo $variable
```

Export 会把变量放到 env 中，全局可见。Echo 会显示出来

命令行参数：echo \$#

```
jonas@ubuntu:~$ ./prog 123 abc hhh
3
./prog
123
abc
jonas@ubuntu:~$ cat prog
echo $#
echo $0
echo $1
echo $2
jonas@ubuntu:~$
```

为什么程序结束还要返回一个整型？

Shell 语言结束后，需要得到一个结束的信号。将整个整型传递给 Exit()，0 表示正常返回。

SHELL 语法成分：\$ 访问变量

内部变量，linux 提供了一种特殊变量，常用的是

\$# 传送给 shell 程序的位置参数个数

\$? 最后命令的代码或 shell 程序内所调用的 shell 程序

\$0 shell 程序的名称

\$* 调用 shell 程序时所传送全部参数的字符串

```
if [ -d Desktop ]
then
    echo $#
    echo $?
    echo $0
    echo $*

    echo $1
    echo $2
    echo $3
else
    echo "none"
fi

jonas@ubuntu:~$ cat prog_loop
for filename in `ls`
do
    echo $filename
done
```

20220920 main 函数

Vi 操作

:n1,n2 co n3 : 将 n1 至 n2 行复制到 n3 行的下面

:n1,n2 m n3 : 将 n1 至 n2 行剪切至 n3 行的下面

:n1,n2 d : 将 n1 至 n2 行删除

GCC

NAME	gcc - GNU project C and C++ compiler collections 百宝箱！套件！
SYNOPSIS	gcc [-o <u>outfile</u>] infile
DESCRIPTION	<p>When invoking GCC, it normally does preprocessing, compilation, assembly and linking. -> a.out</p> <p>... ..</p> <ul style="list-style-type: none">- 在预处理阶段，gcc 会把需要调用的头文件包含进来，替换宏常量和宏代码段- 在编译阶段，gcc 会检查代码的规范性、是否有语法错误等，在检查无误后，gcc 会把文件翻译成 .s 后缀的汇编文件- 在汇编阶段，gcc 会把 .s 后缀的汇编文件 翻译成 .o 后缀的目标文件(机器可识别的二进制文件)- 在链接阶段，gcc 会把目标文件链接到库中，生成可执行文件 <p>gcc -E test.c -o outputfile.i</p> <p>.i preprocess</p> <p>gcc -S outputfile.i</p> <p>.s not assemble</p> <p>gcc -c outputfile.i</p> <p>.o, not link</p>

main 函数

完整格式：

```
int main(int argc, char *argv[]){  
    return 0;  
}
```

argc: argument count 命令行中的字符串个数；

argv: argument value 一个指针数组，对各个参数依次赋值，argv[0] 一般为程序名。

命令行参数

由下述 `execl` 为例:

对于命令行 `ls /home/jonas/Desktop` 而言, 执行的相当于是:

```
execl("/bin/ls","ls","/home/jonas/Desktop",NULL)
```

执行的文件是 `/bin/ls` 路径下的 `ls` 文件, 也作为命令行参数, 同时还有 `/home/jonas/Desktop`

NAME	Execl, execlp, execl, execv, execvp, execvpe - execute a file
SYNOPSIS	<pre>#include<unistd.h> int execl (const char *pathname, const char *arg, ... /*(char *) NULL*/); int execlp (const char *file, const char *arg, ... /*(char *) NULL*/) int execl (const char *pathname, const char *arg, ... /*(char *) NULL, char *const envp[]*/); int execv (const char *pathname, char const*argv[]); int execvp (const char *file, char const*argv[]); int execvpe (const char *file, char const*argv[], char *const envp[]);</pre>
DESCRIPTION	<p>The <code>exec()</code> family of functions replaces the current process image with a new process image. The functions described in this manual page are front-ends for <code>execve(2)</code>.改变（替换）进程图像（进程地址空间）那些数据，程序中所有内容都在地址空间里放着。硬盘里的可执行文件。执行的程序改变了，进程并没有变</p> <div><pre>SYNOPSIS #include <unistd.h> int execve(const char *pathname, char *const argv[], char *const envp[]); DESCRIPTION execve() executes the program referred to by <code>pathname</code>. This causes the program that is currently being run by the calling process to be replaced with a new program, with newly initialized stack, heap, and (initialized and uninitialized) data segments. <code>pathname</code> must be either a binary executable, or a script starting with a line of the form:</pre></div> <p>The initial argument for these functions is the name of a file that is to be executed. The function can be grouped based on the letters following the <code>exec</code></p>

	<p>prefix.</p> <p>或许只有 <code>execve</code> 才是真正的系统调用，上述六者都是经过封装的。</p> <p>作用就是根据指定的文件名找到可执行文件，用它来取代调用进程的内容，<code>exec</code> 函数族执行成功后不会返回，因为调用的进程实体已经被取代了，只留下 PID 这种表面的信息。</p> <p><code>Int execvpe(const char *file, char const*argv[], char *const envp[])</code></p> <p>用于获取环境变量</p>
--	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

环境变量

对于 `main` 函数而言，环境变量会存放在 `envp` 中，通过 `getenv()` 使用。

环境变量是操作系统中一个具有特定名字的对象，它包含了一个或者多个应用程序所需要用的信息。当被要求执行的程序没有告诉它完整路径时，系统除了在当前目录下找此程序，还应到 `PATH` 中指定路径去找。

main 函数返回值

用于传递给操作系统表示程序退出，可以用 `echo` 查看返回的代码。

动手做

用 c 语言写小程序，通过装配，构成大程序。

```
jonas@ubuntu:~/Desktop$ cat test.c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    int i;
    printf("argc = %d\n", argc);
    for (i=0;i<argc;++i)
        printf("argument[%d]=%s\n", i, argv[i]);
    return 12;
}
```

```
jonas@ubuntu:~/Desktop/20220920$ echo $?
12
jonas@ubuntu:~/Desktop/20220920$ cat test2.c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[], char **envp[])
{
    int i;
    for (i=0;;i++)
    {
        if(envp[i]==NULL)
            break;
        printf("%s\n", envp[i]);
    }
    return 12;
}
```

Show 命令行参数，argc: argument count, argv: argument variable, 指针的数组

```
}
jonas@ubuntu:~/Desktop$ ./a.out abc def efg
argc = 4
argument[0]=./a.out
argument[1]=abc
argument[2]=def
argument[3]=efg
jonas@ubuntu:~/Desktop$
```

简介形式:

```
jonas@ubuntu:~/Desktop$ cat tidy_show_env.c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[], char **envp[])
{
    printf("%s\n", getenv(argv[1]));
    return 12;
}
```

```
jonas@ubuntu:~/Desktop/20220920$ ./a.out PATH
/home/jonas/.local/bin:/opt/ros/foxy/bin:/home/jonas/.local/bin:/home/jonas/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
```

P. S **a.out -- assembler output**

命令行来自中断 **shell** 程序的执行，是命令行解释器

20220927 进程管理类系统功能调用（1）

fork()

NAME	Fork - create a child process
SYNOPSIS	<pre>#include <sys/types.h> #include <unistd.h> Pid_t fork(void);</pre>
DESCRIPTION	<p>Fork()通过复制调用进程来创建一个新进程。新进程称为子进程。调用进程称为父进程。</p> <p>子进程和父进程在不同的内存空间中运行。</p> <p>在 fork()时,两个内存空间具有相同的内容。其中一个进程执行的内存写、文件映射(mmap(2))和取消映射(munmap(2))不会影响另一个进程。</p> <p>The child process is an exact duplicate of the parent process except for the following points:</p> <ul style="list-style-type: none">- The child has its own unique PID;- child's parent process ID == parent's PID- The child does not inherit its parent's memory lock.- The child inherits semaphore adjustments.- The child's set of pending signals is initially empty. <div>RETURN VALUE On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and <u>errno</u> is set appropriately.</div> <pre>Pid_t pid = fork() == 0 child Pid_t pid = fork() == -1 parent</pre>

getpid()&getppid()

NAME	Getpid, getppid - get process identification
SYNOPSIS	<pre>#include <sys/types.h></pre>

	<pre>#include <unistd.h> pid_t getpid(void); pid_t getppid(void);</pre>
DESCRIPTION	<p>Getpid() returns the process ID (PID) of the calling process.(This is often used by routines that generate unique temporary filenames)</p> <p>Getppid() returns the process ID of the parent of the calling process. This will be either the ID of the process that created this process using <i>fork()</i>, or, if that process has already terminated, the ID of the process to which this process has been reparented.</p>

wait()

Wait 等待进程结束，因此是和 main 函数的返回值有关的

用于回收子进程，或者子进程会变为僵尸态，占用系统资源

NAME	Wait, waitpid, waitid - wait for process to change state
SYNOPSIS	<pre>#include <sys/types.h> #include <sys/wait.h> pid_t wait(int *wstatus); pid_t waitpid(pid_t pid, int *wstatus, int options); int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);</pre>
DESCRIPTION	<p>All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed.</p> <p>If a child has already changed state, then these calls return immediately.</p> <p>WIFEXITED(wstatus)</p> <p>Returns true if the child terminated normally;</p> <p>WEXITSTATUS(wstatus)</p> <p>Returns the exit status of the child.</p>

Exec()执行一个程序，成功执行时，当前进程中的程序会被替代，不会继续进行；不成功才会继续；

动手做

```
#include <sys/types.h>
#include <unistd.h>
int main()
{
    //printf("Start %d\n", getpid());
    //printf("another_start %d", getpid());
    pid_t pid = fork();
    if(pid < 0)
        perror("FAILED");
    if(pid==0)
    {
        printf("child, pid: %d, ppid:%d, return pid from fork:%d\n", get
pid(), getppid(), pid);
        exit(123);
    }
    else
    {
        printf("parent, pid:%d, ppid:%d, return pid from fork:%d\n", get
pid(), getppid(), pid);
        exit(234);
    }
    return 0;
}
```

```
jonas@ubuntu:~/Desktop/20220927_ProcessManagment_1$ ./a.out
parent, pid:43354, ppid:42630, return pid from fork:43355
child, pid: 43355, ppid:43354, return pid from fork:0
```

What's 42630??

Just ps it

And confusion solved.

```
jonas@ubuntu:~/Desktop/20220927_ProcessManagment_1$ ps
  PID TTY          TIME CMD
 42630 pts/0    00:00:00 bash
 43379 pts/0    00:00:00 ps
```

20221011 进程管理类系统功能调用（2）

进程和程序一一对应吗？不是！一个程序可以创建很多个进程，一个进程可以对应很多个程序，因为可以替换

Process image

它是一个内存级的实体并由：

- 进程控制块（PCB）、
- 进程执行的程序（code） / 程序、
- 进程执行时所用的数据 / 数据集合、
- 进程执行时使用的工作区组成。

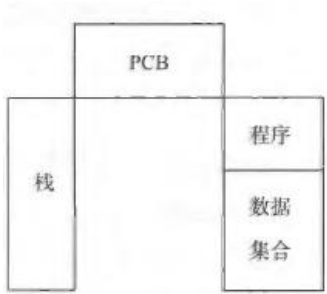


图 2-7 进程映像模型

Process image: 对 process 所在地址空间进行初始化
地址空间中的数据来源于“可执行文件” 或 “参数”

Exec()

对系统功能调用的一个封装

NAME	Execl, execlp, execl, execv, execvp, execvpe - execute a file
SYNOPSIS	<pre>#include<unistd.h> int execl (const char *pathname, const char *arg, ... /*(char *) NULL*/); int execlp (const char *file, const char *arg, ... /*(char *) NULL*/) int execl (const char *pathname, const char *arg, ... /*(char *) NULL, char *const envp[]*/);</pre>

	<pre>int execv (const char *pathname, char const*argv[]); int execvp (const char *file, char const*argv[]); int execvpe (const char *file, char const*argv[], char *const envp[]);</pre>
DESCRIPTION	<p>The exec() family of functions replaces the current process image with a new process image. The functions described in this manual page are front-ends for <code>execve(2)</code>. 改变（替换）进程图像（进程地址空间）那些数据，程序所有内容都在地址空间里放着。硬盘里的可执行文件。执行的程序改变了，进程并没有变 金蝉脱壳</p> <pre>SYNOPSIS #include <unistd.h> int execve(const char *pathname, char *const argv[], char *const envp[]); DESCRIPTION execve() executes the program referred to by pathname. This causes the program that is currently being run by the calling process to be replaced with a new program, with newly initialized stack, heap, and (initialized and uninitialized) data segments. pathname must be either a binary executable, or a script starting with a line of the form:</pre> <p>The initial argument for these functions is the name of a file that is to be executed. The function can be grouped based on the letters following the exec prefix.</p> <p>或许只有 <code>execve</code> 才是真正的系统调用，上述六者都是经过封装的。</p> <p>作用就是根据指定的文件名找到可执行文件，用它来取代调用进程的内容，<code>exec</code> 函数族执行成功后不会返回，因为调用的进程实体已经被取代了，只留下 PID 这种表面的信息。</p> <pre>Int execvpe(const char *file, char const*argv[], char *const envp[]) 用于获取环境变量</pre>
六种系统功能调用的解释	<p><code>Execl</code> & <code>execvp</code> 用到了 <code>char *envp[]</code> 传递环境变量，其他四个把默认的环境变量不做修改地传进应用程序。而前二者会用指定的修改它们。</p> <p><code>Execlp</code> & <code>execvp</code> & <code>execvpe</code> 可以直接将 <code>file</code> 作为参数，例如 <code>ls</code>，会自动到 <code>PATH</code> 里去找。</p> <p>l - <i>execl()</i>, <i>execlp()</i>, <i>execl()</i> list of one or more pointers, terminated by null.</p> <p>v - <i>execv()</i>, <i>execvp()</i>, <i>execvpe()</i> specify the command-line args as a vector.</p> <p>e - <i>execl()</i>, <i>execvpe()</i> envp arg.</p> <p>p - <i>execlp()</i>, <i>execvp()</i>, <i>execvpe()</i> duplicate them by searching executable file.</p>

wait()

Wait 等待进程结束，因此是和 main 函数的返回值有关的

用于回收子进程，或者子进程会变为僵尸态，占用系统资源

NAME	Wait, waitpid, waitid - wait for process to change state
SYNOPSIS	<pre>#include <sys/types.h> #include <sys/wait.h> pid_t wait(int *wstatus); pid_t waitpid(pid_t pid, int *wstatus, int options); int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);</pre>
DESCRIPTION	<p>All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed.</p> <p>If a child has already changed state, then these calls return immediately.</p> <p>WIFEXITED(wstatus)</p> <p>Returns true if the child terminated normally;</p> <p>WEXITSTATUS(wstatus)</p> <p>Returns the exit status of the child.</p>

To make relations clear:

main 函数: `int main(int argc, char *argv[], char *envp[]);`

动手做

Ls, ps 等命令并不是操作系统中的，它们只是程序，操作系统提供了命令接口。

系统调用后当前程序就不存在了，即 **terminate** 不输出

```
jonas@ubuntu:~/Desktop/20221010$ cat p1.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    printf("Start\n");
    execl("/bin/ls", "ls", NULL);
    printf("Terminate\n");
    return 0;
}
jonas@ubuntu:~/Desktop/20221010$ ./a.out
Start
a.out p1.c
```

添加更多参数

```
jonas@ubuntu:~/Desktop/20221010$ cat p2.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    printf("Start\n");
    execl("/bin/ls","ls","-al" ,"/bin/ls" , NULL);
    printf("Terminate\n");
    return 0;
}
jonas@ubuntu:~/Desktop/20221010$ ./a.out
Start
-rwxr-xr-x 1 root root 142144 Sep  5 2019 /bin/ls
```

创建子进程

同时执行，而非父进程先执行。子进程工作量大，所以打印的慢。s

```
jonas@ubuntu:~/Desktop/20221010$ cat p3.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    if(fork()==0)
        execl("/bin/ls","ls","-al" ,"/bin/ls" , NULL);
    else
        printf("I'm a parent process\n");
    return 0;
}
jonas@ubuntu:~/Desktop/20221010$ ./a.out
I'm a parent process
jonas@ubuntu:~/Desktop/20221010$ -rwxr-xr-x 1 root root 142144 Sep  5 2019 /bin/ls
```

Wait 等待子进程状态结束（变化）

```
#include<stdlib.h>
#include<unistd.h>
#include<wait.h>
#include<sys/types.h>

int main()
{
    int status,pid;

    if(fork()==0)
    {
        printf("I'm a child process\n");
        execl("/bin/ls","ls","-al","/bin/ls",NULL);
    }
    else{
        printf("I'm a parent process and begin wait\n");
        pid=wait(&status);
        printf("end wait,pid from fork:%d,%d, %d\n",pid,WEXITSTATUS(status), WIFEXITED(status));
    }

    return 0;
}
```

```
jonas@ubuntu:~/Desktop/20221010_ProcessManagment_2$ ./a.out
I'm a parent process and begin wait
I'm a child process
-rwxr-xr-x 1 root root 142144 Sep  5 2019 /bin/ls
end wait,pid from fork:44041,0, 1
```

20221018 C 语言程序执行 shell 程序&管道实现进程通信

Shell 语言是个解释性的语言，Shell 语言的解释器是/bin/bash；

最快生成文本文件：Cat>x.c；

所有程序都是文本文件；交给/bin/bash

Shell 的执行过程

```
jonas@ubuntu:~/Desktop$ strace ls -l
execve("/usr/bin/ls", ["ls", "-l"], 0x7ffe255b0ac8 /* 56 vars */) = 0
brk(NULL)                               = 0x55e976ad9000
```

简单读写

```
jonas@ubuntu:~/Desktop/20221018_CShell_ProcessCommunication$ cat p1.c
#include <stdlib.h>
#include <stdio.h>

int main()
{
    char buff[100];
    scanf("%s", buff);
    printf("%s\n", buff);
    return 0;
}
```

Ls 和 ls | sort 是两个命令；
如何在两个进程之间假设一个管道线，使前一个程序的输出作为下一个程序的输入；

实现 ls|sort

- 用的工具叫管道；
 - Pipe, pipe2 – create a pipe

NAME	Pipe, pipe2 - create pipe
SYNOPSIS	<pre>#include <unistd.h> int pipe(int pipefd[2]); int pipe2(int pipefd[2], int flags);</pre>
DESCRIPTION	<p>Pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication.</p> <p>Pipefd[0] refers to the read end of the pipe, while pipefd[1] refers to the write</p>

使用管道实现两个进程通信，父进程的输出作为子进程的输入，并输出。

```
jonas@ubuntu:~/Desktop/20221018_CShell_ProcessCommunication$ cat process_communication_pipe.c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int p[2];          // For pipe ends;
    char buffer[100];  // Also for pipe;
    pipe(p);           // Equal to open two ends:read and write

    // Aim is to have one write and one read what was written
    pid_t pid = fork();
    // printf("%d", getpid());
    if(pid == 0)       // Child process
    {
        close(0); // close standard input
        dup(p[0]); // dup pipe read port to standard input
        close(p[1]); // close original pipe port
        close(p[0]);
        scanf("%s", buffer);
        printf("I received %s\n", buffer);
    }
    else
    {
        close(1); // close standard output, will not show following
        dup(p[1]); // dup write port to standard output
        close(p[0]);
        close(p[1]);
        printf("hahahahah\n");
    }

    return 0;
}
```

```
jonas@ubuntu:~/Desktop/20221018_CShell_ProcessCommunication$ ./a.out
I received hahahahah
```

类似地，实现 `ls | sort`，将 `ls` 的结果送给 `sort`，得到最终结果

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int p[2];          // For pipe ends;
    char buffer[100];  // Also for pipe;
    pipe(p);           // Equal to open two ends:read and write

    // Aim is to have one write and one read what was written
    pid_t pid = fork();
    // printf("%d", getpid());
    if(pid == 0)       // Child process
    {
        close(0); // close standard input
        dup(p[0]); // dup pipe read port to standard input
        close(p[1]); // close original pipe port
        close(p[0]);
        execl("/usr/bin/ls", "sort", NULL);
    }
    else
    {
        close(1); // close standard output, will not show following
        dup(p[1]); // dup write port to standard output
        close(p[0]);
        close(p[1]);
        execl("/bin/ls", "ls", NULL);
    }

    return 0;
}
```



```
jonas@ubuntu:~/Desktop/20221018_CShell_ProcessCommunication$ ./a.out
a.out DIY_ls_sort.c p1.c process_communication_pipe.c
jonas@ubuntu:~/Desktop/20221018_CShell_ProcessCommunication$ ls | sort
a.out
DIY_ls_sort.c
p1.c
process_communication_pipe.c
```

我们写的软件是软件流水线上的小程序

```
#include<stdlib.h>
#include<stdio.h>
#include<unistd.h>
```

```
Int main()
{
    Char buff[100];

    Int p[2];
    Pipe(p); // 相当于打开了两个文件，一个输入端，一个输出端；
             // p[0] is a read port, p[1] is a write port

    If(fork()==0){ // child process
        Close(0); // close standard input
        Dup(p[0]); // duplicate pipe input port to standard input
        Close(p[0]); // close original pipe port
        Close(p[1]);
        Execl("/usr/bin/ls", "sort", NULL);
    } else { // parent process
        Close(1); // close standard output
        Dup(p[1]); // duplicate pipe output port to standard output
        Close(p[0]); // close original pipe port
        Close(p[1]);
        Execl("/bin/ls", "ls", NULL);
    }
    Return 0;
}
```

20221025 信号的使用

Kill

NAME	Kill - send a signal to a process
SYNOPSIS	Kill [optionsp <pid> [...]]
DESCRIPTION	Use kill -l to see all signals. <div>1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP 6) SIGABRT 7) SIGBUS 8) SIGFPE 9) SIGKILL 10) SIGUSR1 11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM 16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP 21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ 26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO 30) SIGPWR 31) SIGSYS 34) SIGRTMIN 35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3 38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8 43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7 58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2 63) SIGRTMAX-1 64) SIGRTMAX</div>

程序运行时往往可以通过 ctrl+c 停止，这就是向进程发送了一个信号 SIGINT。

```
jonas@ubuntu:~/Desktop/20221018_CShell_ProcessCommunication$ ps aux | grep man
root      159  0.0  0.0   0   0 ?        I<   Nov13   0:00 [charger_manager]
jonas    100470  0.0  0.1  9844  4060 pts/1    S+   06:28   0:00 man dup
jonas    101471  0.0  0.0   9040   660 pts/0    S+   07:23   0:00 grep --color=auto man
```

对于 man dup ，想杀死它，可以直接 kill 100470

```
jonas@ubuntu:~/Desktop/20221018_CShell_ProcessCommunication$ ps aux | grep man
root      159  0.0  0.0   0   0 ?        I<   Nov13   0:00 [charger_manager]
jonas    101473  0.0  0.0   9040   724 pts/0    S+   07:25   0:00 grep --color=auto man
```

Signal

NAME	Signal - ANSI C signal handling
SYNOPSIS	#include <signal.h> sighandler_t signal(int signum, sighandler_t handler);
DESCRIPTION	Signal() sets the disposition of the signal signum to handler. 注册了一个信号捕捉函数，想要设置捕捉的信号编号，回调函数，捕捉后要执行的函数

断点？

1. 怎么进入调试状态，靠中断，指令指针和某个值对上了，引起异常，断点就和断点寄存器指针对上，就挂起，受 IDE 控制，受信号控制，ptues，使进程停下、改变量、继续进行....变量在 os 里对应地址，-g—符号表，原理就是信号。各个进程之间不能相互影响，—

都是独立的 balabala, 。

动手做

对于一个正在执行的程序，根据其捕获的信号编号发送给它。

首先，使用 ps 查看进程号：

```
jonas@ubuntu:~$ ps aux | grep a.out
jonas      101572  98.8  0.0   2376   580 pts/0    R+   07:36   1:08 ./a.out
jonas      101575   0.0  0.0   9040   660 pts/2    S+   07:37   0:00 grep --color=
auto a.out
```

Pid 是 101572，查看会被捕获的信号编号，当前进程处于运行状态：

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void f(int sig_id)
{
    printf("received signal %d\n", sig_id);
    return;
}

int main()
{
    signal(31,f());
    for(;;);
}
```

```
jonas@ubuntu:~/Desktop/20221025_Signal$ ./a.out
```

通过 kill，将信号发送给该进程查看结果，

```
jonas@ubuntu:~$ kill -31 101572
jonas@ubuntu:~$ 
jonas@ubuntu:~/Desktop/20221025_Signal$ ./a.out
received signal 31
```

信号被成功捕获，

现在想要杀死改进程：kill -9 101572

```
jonas@ubuntu:~$ kill -9 101572
jonas@ubuntu:~$ 
jonas@ubuntu:~/Desktop/20221025_Signal$ ./a.out
received signal 31
Killed
```

```
#include <stdio.h>
#include <stdlib.h>
Void f(int sig_id)
{
    Printf("received signal\n", sig_id);
    Return;}
Int main(){
    Signal(31, f);
    For(;;)
}
```

20221101 存储映射文件 mmap

文件操作:

NAME	Open, openat, creat - open and possibly create a file
SYNOPSIS	<pre>#include <sys/types.h> #include <sys/stat.h> #include <fcntl.h> Int open(const char *pathname, int flags); Int open(const char *pathname, int flags, mode_t mode); Int openat(int dirfd, const char *pathname, int flags, mode_t mode)</pre> <div><pre>SYNOPSIS #include <sys/types.h> #include <sys/stat.h> #include <fcntl.h> int open(const char *pathname, int flags); int open(const char *pathname, int flags, mode_t mode); int creat(const char *pathname, mode_t mode); int openat(int dirfd, const char *pathname, int flags); int openat(int dirfd, const char *pathname, int flags, mode_t mode);</pre></div>
DESCRIPTION	<p>The open() system call opens the file specified by pathname. Not existed, create.</p> <p>文件不存在创建它，并指定模式</p> <div><p>The argument <code>flags</code> must include one of the following <u>access modes</u>: <code>O_RDONLY</code>, <code>O_WRONLY</code>, or <code>O_RDWR</code>. These request opening the file read-only, write-only, or read/write, respectively.</p></div>

动手做

用 `pathname`，`exec` 中用的是 `pass`；`Ls>x.dat` 使显示内容输入到文件中；
程序：把文件读到进程而非内存

```
jonas@ubuntu:~/Desktop/20221101$ cat p1.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main()
{
    int fd=open("./x.date", O_RDONLY);
    char buf[2000];
    read(fd, buf, 2000);
    printf("%s", buf);
    close(fd);

    return 0;
}
```

结果:

```
jonas@ubuntu:~/Desktop/20221101$ ./a.out
20220920
20220927
20221010
20221101
Internal short circuit detection in Li-ion batteries using supervised machine learning.pdf
mytext2.txt
mytext.txt
PyEdition-framework-v2-main
table
x.date
一些压缩包
```

MMAP:整个存储管理的核心

Memory mapping 内存映射

将一个虚拟内存区域与一个磁盘上的对象关联起来，来初始化这个虚拟内存区域的内容。减少拷贝次数。不需要再拷贝到内核。

NAME	Mmap, munmap - map or unmap files or devices into memory
SYNOPSIS	<pre>#include <sys/mman.h> void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset); int munmap (void *addr, size_t length);</pre> <div><pre>NAME mmap, munmap - map or unmap files or devices into memory SYNOPSIS #include <sys/mman.h> void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset); int munmap(void *addr, size_t length); See NOTES for information on feature test macro requirements. DESCRIPTION mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in addr. The length argument specifies the length of the mapping (which must be greater than 0).</pre></div>
DESCRIPTION	mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in addr . The length argument specifies the length of the mapping (which must be greater than 0).

在虚地址空间创建一个新的映射；prot 是权限

进程里一段地址对应外存中某个文件，一旦产生缺页中断，知道的是丢失的地址信息。

根据虚地址，知道落在哪个区域内；

进程中所有的信息都来自外存，能放内存就尽量放

```

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

int main()
{
    int fd=open("./x.date", O_RDONLY);    //open file
    size_t len = lseek(fd, 0, SEEK_END); //set file R/W pointer
    char *p=mmap(NULL, len, PROT_READ, MAP_SHARED, fd, 0); //map a file to v
    irtual address space
    printf("%s", p);
    munmap(p, len);
    close(fd);

    return 0;
}

```

取文件长度 lseek

```

NAME
    lseek - reposition read/write file offset

SYNOPSIS
    #include <sys/types.h>
    #include <unistd.h>

    off_t lseek(int fd, off_t offset, int whence);

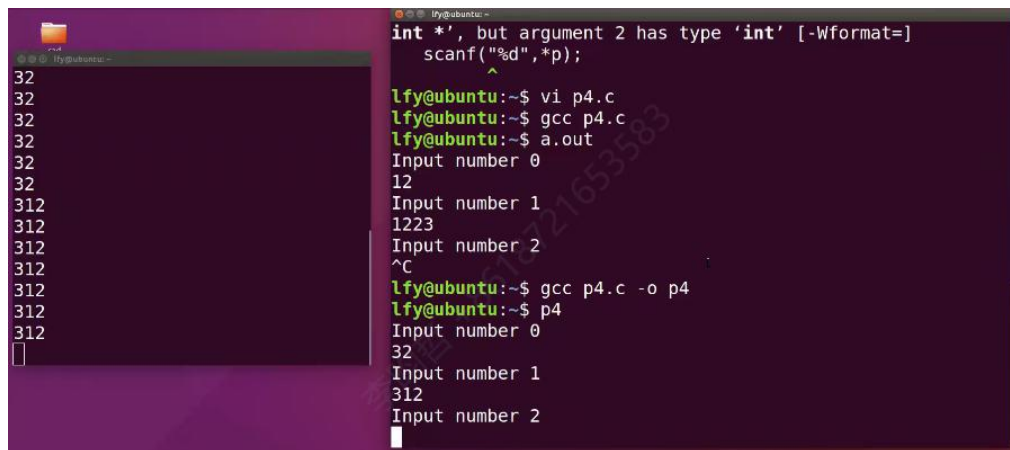
DESCRIPTION
    lseek() repositions the file offset of the open file description asso-
    ciated with the file descriptor fd to the argument offset according to
    the directive whence as follows:

    SEEK_SET
        The file offset is set to offset bytes.

    SEEK_CUR
        The file offset is set to its current location plus offset
        bytes.

```

文件共享存储



```

lfy@ubuntu:~$ vi p4.c
lfy@ubuntu:~$ gcc p4.c
lfy@ubuntu:~$ a.out
Input number 0
12
Input number 1
1223
Input number 2
^C
lfy@ubuntu:~$ gcc p4.c -o p4
lfy@ubuntu:~$ p4
Input number 0
32
Input number 1
312
Input number 2

```

Scanf 不仅做了写，还做了读，因此权限上出问题了


```

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

int main()
{
    int fd=open("./x.date", O_RDWR);    //open file
    size_t len = lseek(fd, 0, SEEK_END); //set file R/W pointer
    volatile int *p, i;

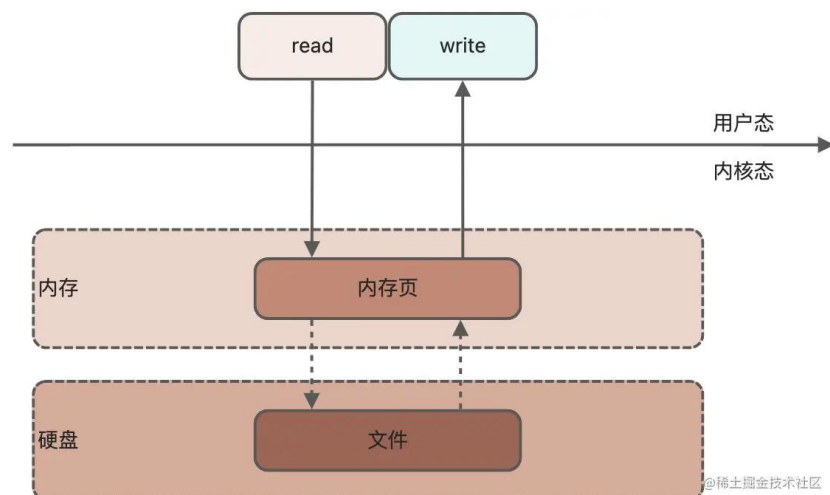
    if(fork()==0)
    {
        for(p=mmap(NULL, len, PROT_READ, MAP_SHARED, fd, 0);;)
        {
            sleep(1);
            printf("%d\n",*p);
        }
    }
    else{
        for(i = 0;p=mmap(NULL, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);i++)
        {
            printf("Input number %d\n", i);
            scanf("%d", p);
        }
    }
    munmap((void *)p, len);
    close(fd);

    return 0;
}

```

volatile 是一个特征修饰符（type specifier）。volatile 的作用是作为指令关键字，确保本条指令不会因编译器的优化而省略，且要求每次直接读值。volatile 的变量是说这变量可能会被意想不到地改变，这样，编译器就不会去假设这个变量的值了。

说到 MMAP 的保护神，首页了解下内存页：在页式虚拟存储器中，会在虚拟存储空间和物理主存空间都分割为一个个固定大小的页，为线程分配内存也是以页为单位。比如：页的大小为 4K，那么 4GB 存储空间就需要 $4GB/4KB=1M$ 条记录，即有 100 多万 4KB 的页，内存页中，当用户发生文件读写时，内核会申请一个内存页与文件进行读写操作，如图：



这时如果内存页中没有数据，就会发生一种中断机制，它就叫缺页中断，此中断就是 MMAP 的保护神，为什么这么说呢？我们知道 mmap 函数调用后，在分配时只是建立了进程虚拟地址空间，并没有分配虚拟内存对应的物理内存，当访问这些没有建立映射关系的虚拟内存时，CPU 加载指令发现代码段是缺失的，就触发了缺页中断，中断后，内核通过检查虚拟地址的所在区域，发现存在内存映射，就可以通过虚拟内存地址计算文件偏移，定位到内存所

缺的页对应的文件的页，由内核启动磁盘 IO，将对应的页从磁盘加载到内存中。最终保护 mmap 能顺利进行，无私奉献。了解完缺页中断，我们再来细聊下 mmap 四种场景下的内存分配原理

.信号：（signal）是一种处理异步事件的方式。信号是比较复杂的通信方式，用于通知接受进程有某种事件发生，除了用于进程外，还可以发送信号给进程本身。

2.信号量：（Semaphore）进程间通信处理同步互斥的机制。是在多线程环境下使用的一种设施，它负责协调各个线程，以保证它们能够正确、合理的使用公共资源。

简单地讲，信号就是一种异步通信，通知进程某种事件的发生；信号量是进程/线程同步与互斥的一种机制，保证进程/线程间之间的有序执行或对公共资源的有序访问