## 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?

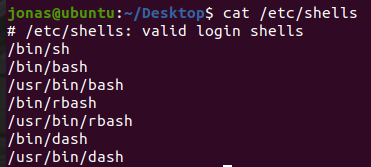
Under Unix, KT met with the problem of too small storage to run a programme..

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



### 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。



|  |
| --- |
| 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 合并称一个文件

### 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



引用变量

variable=123456

Export variable

Echo $variable

环境变量？

一些系统设置

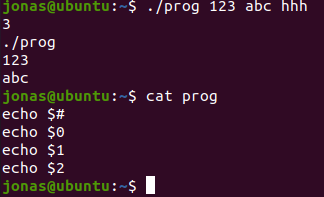
Path环境变量：执行命令时在path下一个个找。

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



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 $#



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

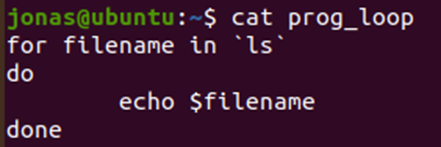
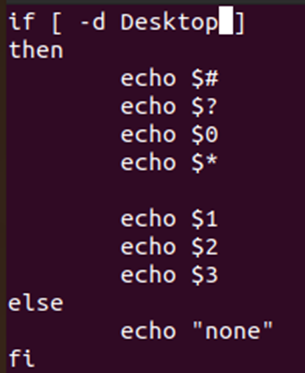
内部变量，linux提供的一种特殊变量，常用的是

$# 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



## 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 outfile] ... ... infile |
| DESCRIPTION | When invoking GCC, it normally does preprocessing, compilation, assembly and linking. -> a.out  ... ...  - 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  ***gcc -E test.c -o outputfile.i***  .i preprocess  ***gcc -S outputfile.i***  .s not assemble  ***gcc -c outputfile.i***  .o，not link |

### 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, execle, execv, execvp, execvpe - execute a file |
| SYNOPSIS | #include<unistd.h>  int execl (const char \***pathname**, const char \*arg, ... /\*(char \*) NULL\*/)；  int execlp (const char \***file**, const char \*arg, ... /\*(char \*) NULL\*/)  int execle (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[]); |
| DESCRIPTION | 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 execve(2).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    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.  或许只有execve才是真正的系统调用，上述六者都是经过封装的。  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，exec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。  Int execvpe(const char \*file, char const\*argv[], char \*const envp[])  用于获取环境变量 |

### 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。

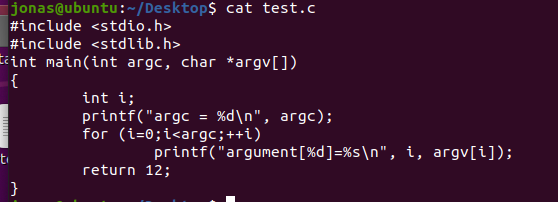
环境变量是操作系统中一个具有特定名字的对象，它包含了一个或者多个应用程序所需要用到的信息。当被要求执行的程序没有告诉它完整路径时，系统除了在当前目录下找此程序，还应到PATH中指定路径去找。

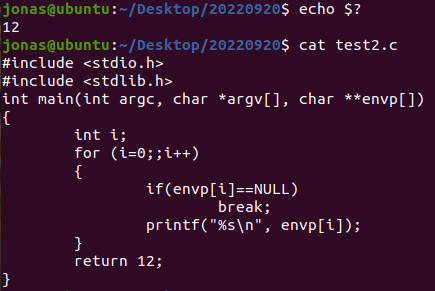
### main函数返回值

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。

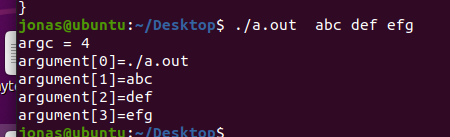
### 动手做

用c语言写小程序，通过装配，构成大程序。

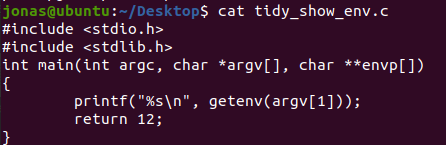




Show 命令行参数，argc: argument count, argv: argument variable,指针的数组



简介形式：





1. **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 | #include <sys/types.h>  #include <unistd.h>  Pid\_t fork(void); |
| DESCRIPTION | 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。  子进程和父进程在不同的内存空间中运行。  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。  The child process is an exact duplicate of the parent process except for the following points:  - 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.    Pid\_t pid = fork() == 0 child  Pid\_t pid = fork() == -1 parent |

### getpid()&getppid()

|  |  |
| --- | --- |
| NAME | Getpid, getppid - get process identification |
| SYNOPSIS | #include <sys/types.h>  #include <unistd.h>  pid\_t getpid(void);  pid\_t getppid(void); |
| DESCRIPTION | Getpid() returns the process ID (PID) of the calling process.(This is often used by routines that generate unique temporary filenames)  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 ***fork()***, or, if that process has already terminated, the ID of the process to which this process has been reparented. |

### wait()

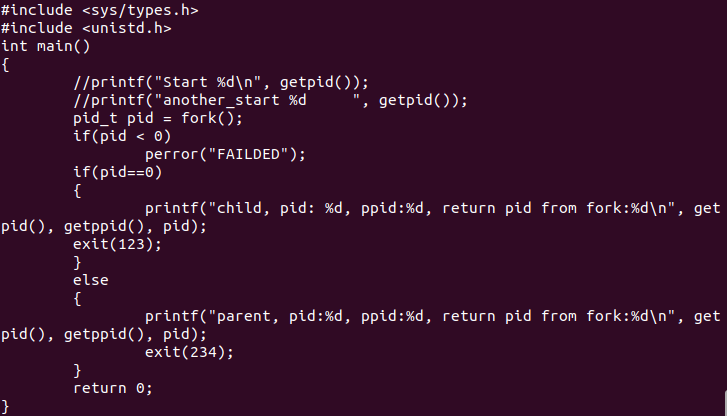
Wait 等待进程结束，因此是和main函数的返回值有关的

**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 | #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); |
| DESCRIPTION | 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.  If a child has already changed state, then these calls return immediately.  WIFEXITED(wstatus)  Returns true if the child terminated normally;  WEXITSTATUS(wstatus)  Returns the exit status of the child. |

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

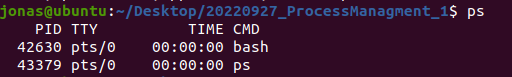




What’s 42630??

Just ps it

And confusion solved.



## 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

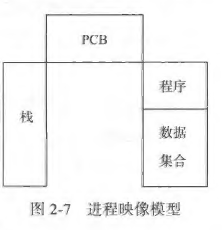
它是一个内存级的实体并由：

[进程控制块](https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B%E6%8E%A7%E5%88%B6%E5%9D%97?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B%E6%98%A0%E5%83%8F/_blank)（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。

[](https://baike.baidu.com/pic/è¿ç¨æ å/4792725/0/faedab64034f78f0231fbc8d76310a55b3191cb3?fr=lemma%26fromModule=lemma_content-image%26ct=single)

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, execle, execv, execvp, execvpe - execute a file |
| SYNOPSIS | #include<unistd.h>  int execl (const char \***pathname**, const char \*arg, ... /\*(char \*) NULL\*/)；  int execlp (const char \***file**, const char \*arg, ... /\*(char \*) NULL\*/)  int execle (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[]); |
| DESCRIPTION | 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 execve(2).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 **金蝉脱壳**    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.  或许只有execve才是真正的系统调用，上述六者都是经过封装的。  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，exec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。  Int execvpe(const char \*file, char const\*argv[], char \*const envp[])  用于获取环境变量 |
| 六种系统功能调用的解释 | Execle & execvpe用到了char \*envp[]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。  Execlp & execvp & execvpe 可以直接将file作为参数，例如ls，会自动到PATH里去找。  l - ***execl(), execlp(), execle()*** list of one or more pointers, terminated by null.  v -***execv(), execvp(), execvpe()*** specify the command-line args as a vector.  e - ***execle(), execvpe()*** envp arg.  p - ***execlp(), execvp(), execvpe()*** duplicate them by searching executable file. |

### 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 | #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); |
| DESCRIPTION | 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.  If a child has already changed state, then these calls return immediately.  WIFEXITED(wstatus)  Returns true if the child terminated normally;  WEXITSTATUS(wstatus)  Returns the exit status of the child. |

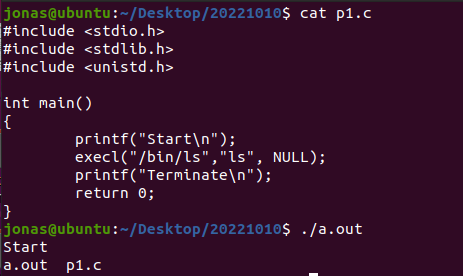
**To make relations clear:**

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

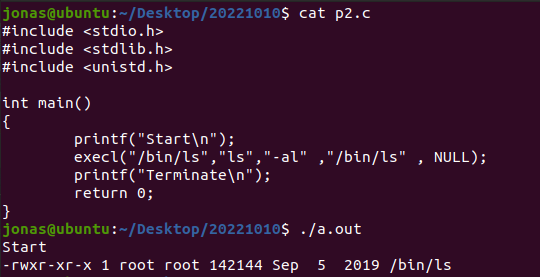
### 动手做

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**

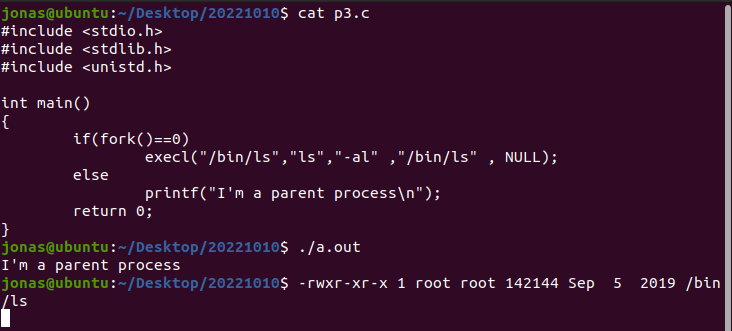


添加更多参数

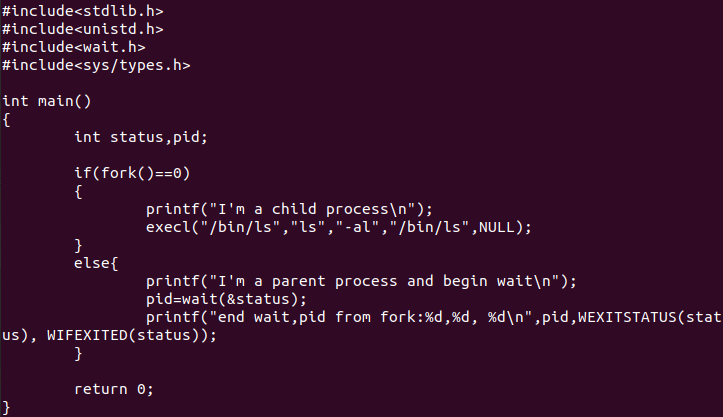


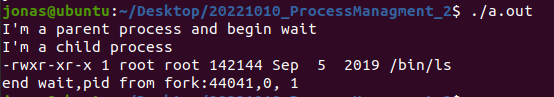
**创建子进程**

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



*Wait* 等待子进程状态结束（变化）





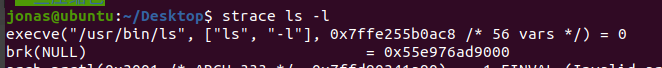
## 20221018 C语言程序执行shell程序&管道实现进程通信

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

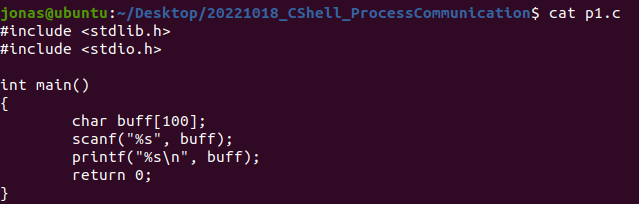
最快生成文本文件：Cat>x.c；

所有程序都是文本文件；交给/bin/bash

Shell的执行过程



### 简单读写



Ls 和 ls | sort 是两个命令；

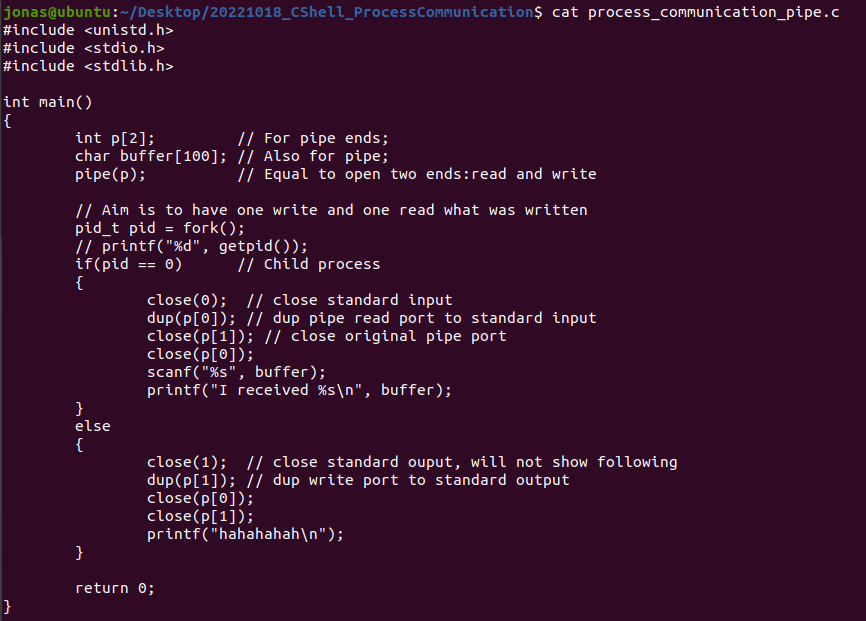
如何在两个进程之间假设一个管道线，使前一个程序的输出作为下一个程序的输入；

### 实现ls|sort

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

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

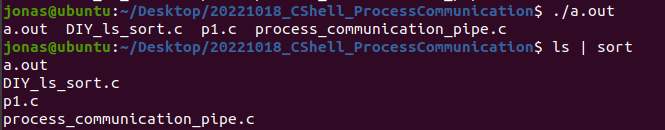
使用管道实现两个进程通信，父进程的输出作为子进程的输入，并输出。





类似地，实现ls | sort，将ls的结果送给sort，得到最终结果





**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;

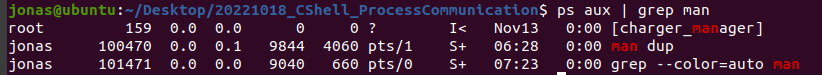
}

## 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. |

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。



对于 man dup ，想杀死它，可以直接 kill 100470



### 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 |

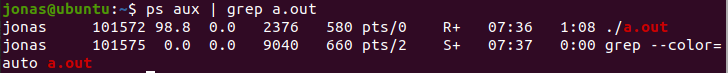
### 断点？

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~~，。

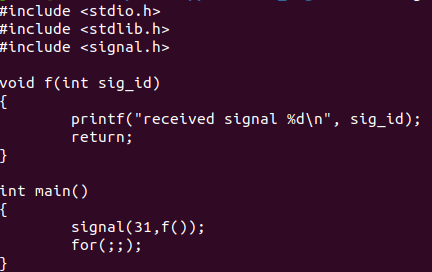
### 动手做

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：

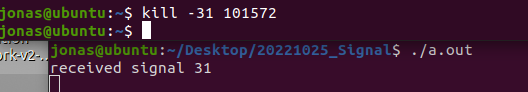


Pid是101572，查看会被捕获的信号编号，当前进程处于运行状态：



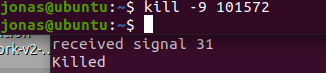


通过kill，将信号发送给该进程查看结果，



信号被成功捕获，

现在想要杀死改进程：kill -9 101572



#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

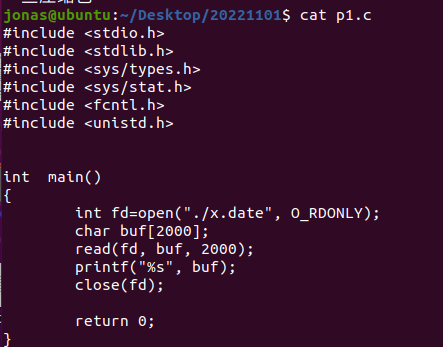
### 文件操作：

|  |  |
| --- | --- |
| NAME | Open, openat, creat - open and possibly create a file |
| SYNOPSIS | #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) |
| DESCRIPTION | The open() system call opens the file specified by pathname. Not existed, create.  文件不存在创建它，并指定模式 |

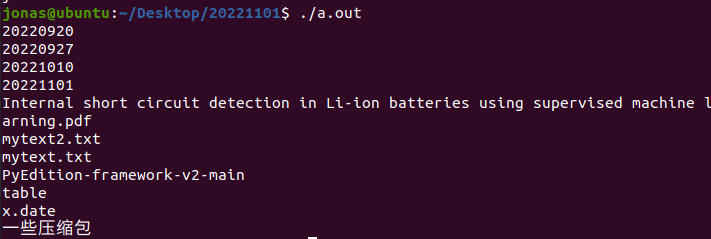
### 动手做

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



结果：



### 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。

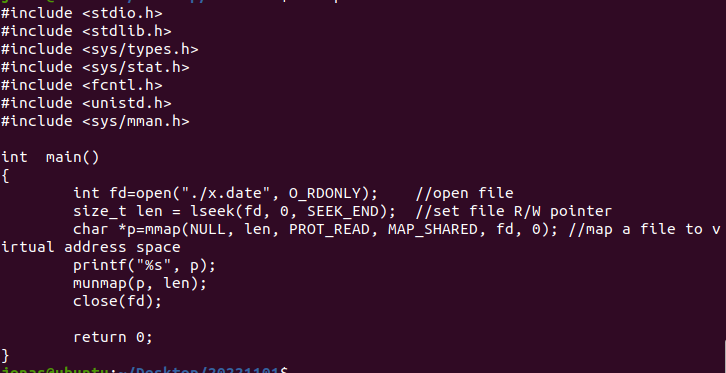
|  |  |
| --- | --- |
| NAME | Mmap, munmap - map or unmap files or devices into memory |
| SYNOPSIS | #include <sys/mman.h>  void \*mmap(void \*addr, size\_t length, int prot, int flags, int fd, off\_t offset);  int munmap (void \*addr, size\_t length); |
| DESCRIPTION | mmap() creates a new mapping 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

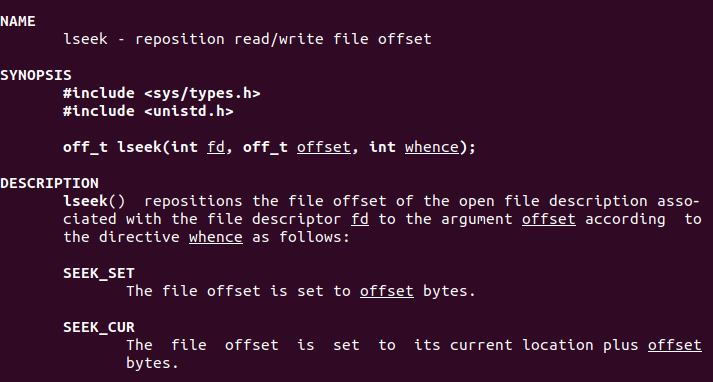
进程里一段地址对应外存中某个文件，一旦产生缺页中断，知道的是丢失的地址信息。

根据虚地址，知道落在哪个区域内；

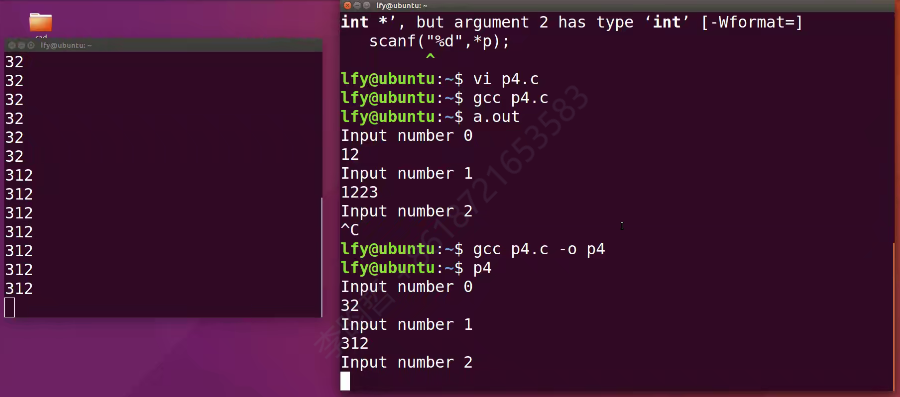
进程中所有的信息都来自外存，能放内存就尽量放



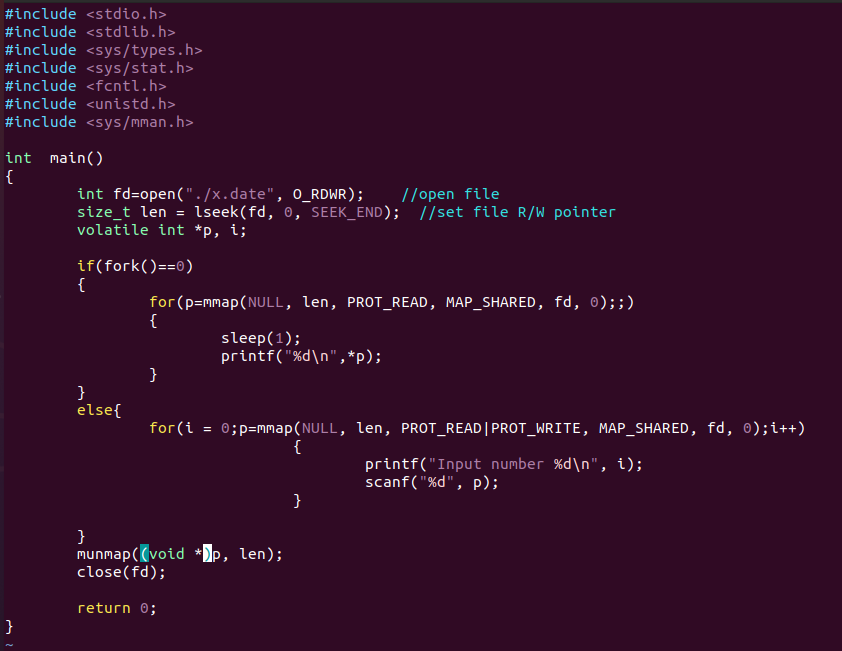
取文件长度 lseek



### 文件共享存储

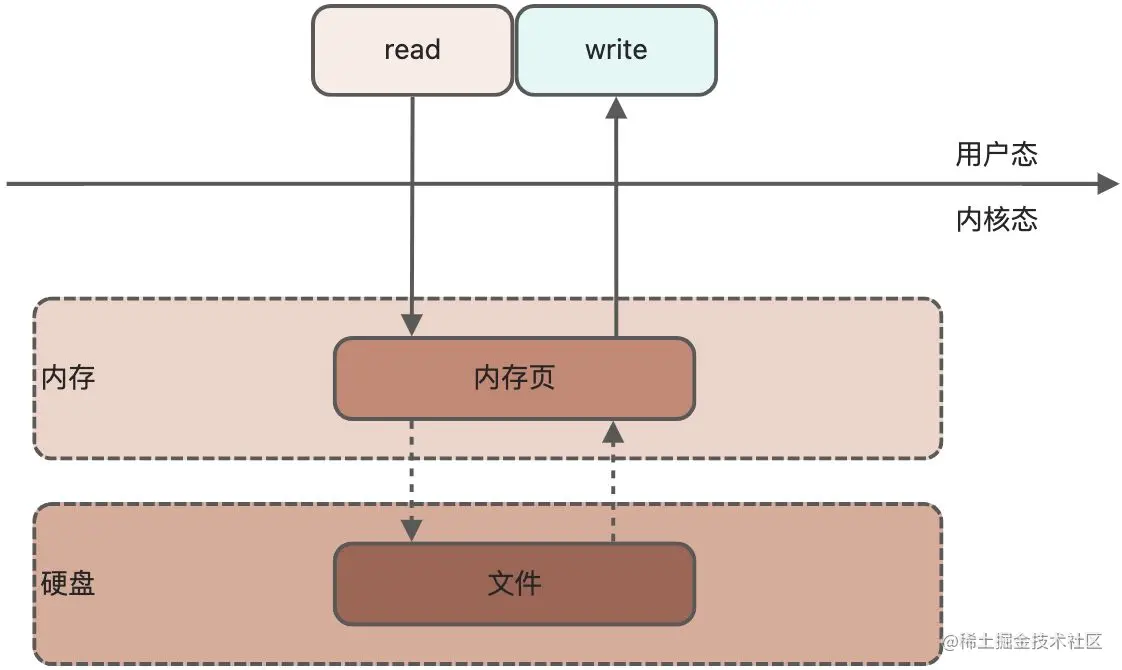


Scanf不仅做了写，还做了读，因此权限上出问题了



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，为线程分配内存是也是以页为单位。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 多万个 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](https://so.csdn.net/so/search?q=%E8%99%9A%E6%8B%9F%E5%86%85%E5%AD%98&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/m0_64420071/article/details/_blank)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。

简单地说，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