目录

[第一部分 多进程生死抉择 2](#_Toc62066661)

[1.解决生命周期问题(init, systemd) 2](#_Toc62066662)

[2 解决进程程序背景的问题(exec) 3](#_Toc62066663)

[2.1exec不是创建进程，它改变程序执行背景 3](#_Toc62066664)

[2.2 vfork没有写时拷贝 4](#_Toc62066665)

[3main函数的生命周期 5](#_Toc62066666)

[3.1main函数即不是程序的入口，也不是出口 5](#_Toc62066667)

[3.2main函数正常退出方式有： 6](#_Toc62066668)

[第二部分 systemd和守护进程Daemon 8](#_Toc62066669)

[1. 用systemd来实现守护进程 8](#_Toc62066670)

[2自己实现daemon进程 10](#_Toc62066671)

[3.nohup 12](#_Toc62066672)

[4.sys V init和systmd 13](#_Toc62066673)

[第三部分 解决进程之间的通信问题(IPC) 13](#_Toc62066674)

[1pipe： 13](#_Toc62066675)

[2信号的处理方法 15](#_Toc62066676)

[2.1捕获、忽略与缺省，block，pending 15](#_Toc62066677)

[2.2修改信号默认行为： 16](#_Toc62066678)

[2.3信号会引起RACE CONFITION 17](#_Toc62066679)

[3 SYS V IPC接口：信号量、共享内存、消息队列 18](#_Toc62066680)

[4.POSIX接口 19](#_Toc62066681)

[4.1信号量sem 19](#_Toc62066682)

[4.2共享内存 19](#_Toc62066683)

[4. UNIX Domain Socket IPC 22](#_Toc62066684)

[Day5文件描述符fd跨进程共享与共享内存 23](#_Toc62066685)

[5.2 memfd\_create与共享内存 25](#_Toc62066686)

[Day6多进程调试方法 25](#_Toc62066687)

[6.1 gdb 25](#_Toc62066688)

# 第一部分 多进程生死抉择

进程调度，主要解决CPU资源分配问题，多进程解决如下问题：

解决生命周期问题(init,systemd)；

解决进程程序背景的问题(exec)

解决进程之间通信问题(IPC)

设计进程与进程之间的界限；

## 1.解决生命周期问题(init, systemd)

fork创建子进程后，可以监控子进程退出原因；父进程死后，子进程默认挂载到init进程；

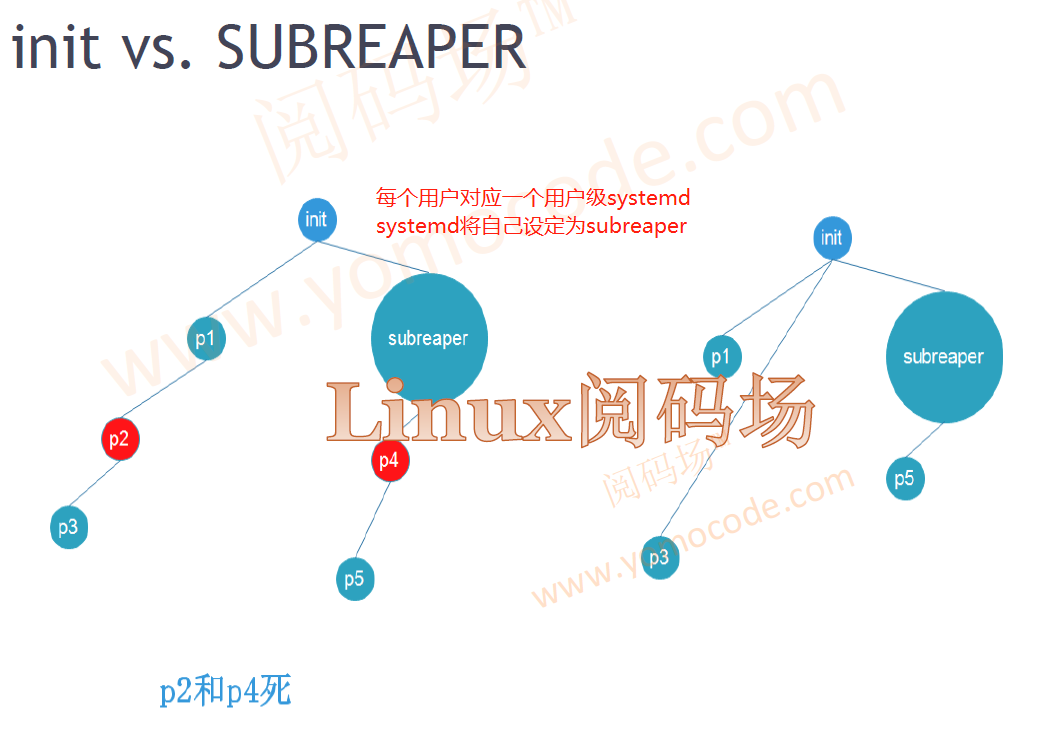
3.4之后的版本，添加了用SUBREPER来挂载孤儿进程；

systemd分两level：

系统级systemd，(init进程对应的软连接),

用户级systemd调用prctl(PR\_SET\_CHILD\_SUBREPER,1)将自己设置为subreper；

Linux为每个用户创建一个systemd。



若父进程先退出，子进程挂靠到对应的systemd根进程。

Systemd源码： vim src/core/main.c

if (!arg\_system)

1984 /\* Become reaper of our children \*/

1985 if (prctl(PR\_SET\_CHILD\_SUBREAPER, 1) < 0)

1986 log\_warning\_errno(errno, "Failed to make us a subreaper: %m");

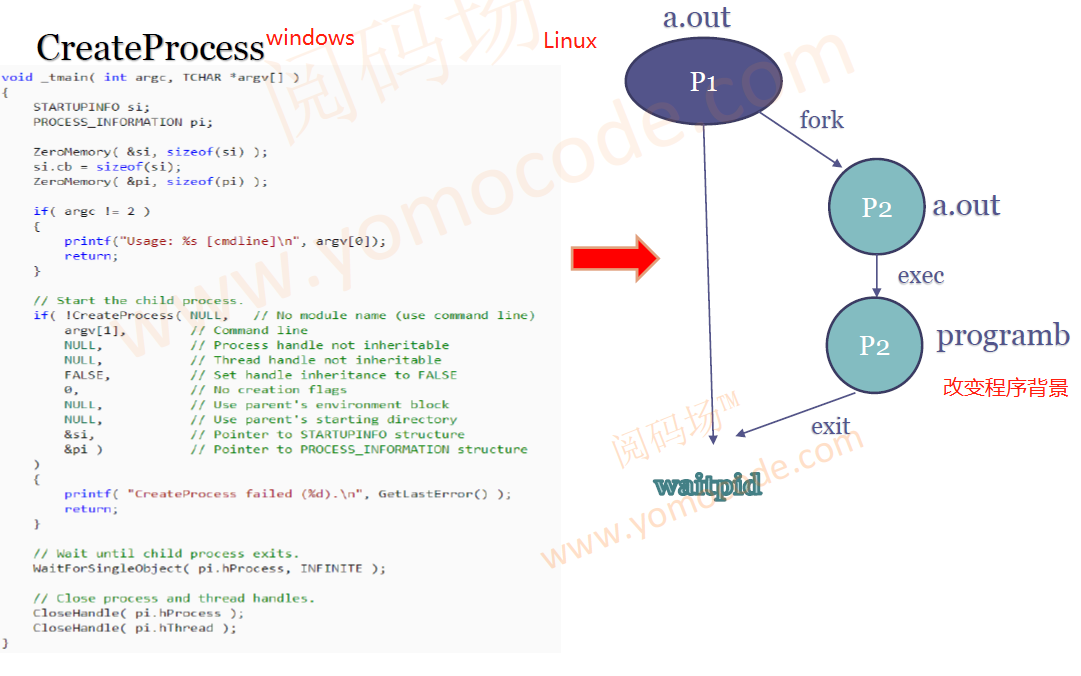
1987

## 2 解决进程程序背景的问题(exec)

### 2.1exec不创建进程，它改变程序执行背景

Windows CreateProcess创建子进程，Linux exec不创建进程，原进程PID不变，只是改变程序**执行背景**(fork原子进程与父进程是COW机制共享资源比如MM等，exec是全新的)；

同一个进程，执行程序一样了；



父子进程都可以改变程序环境。

### 2.2 vfork没有写时拷贝

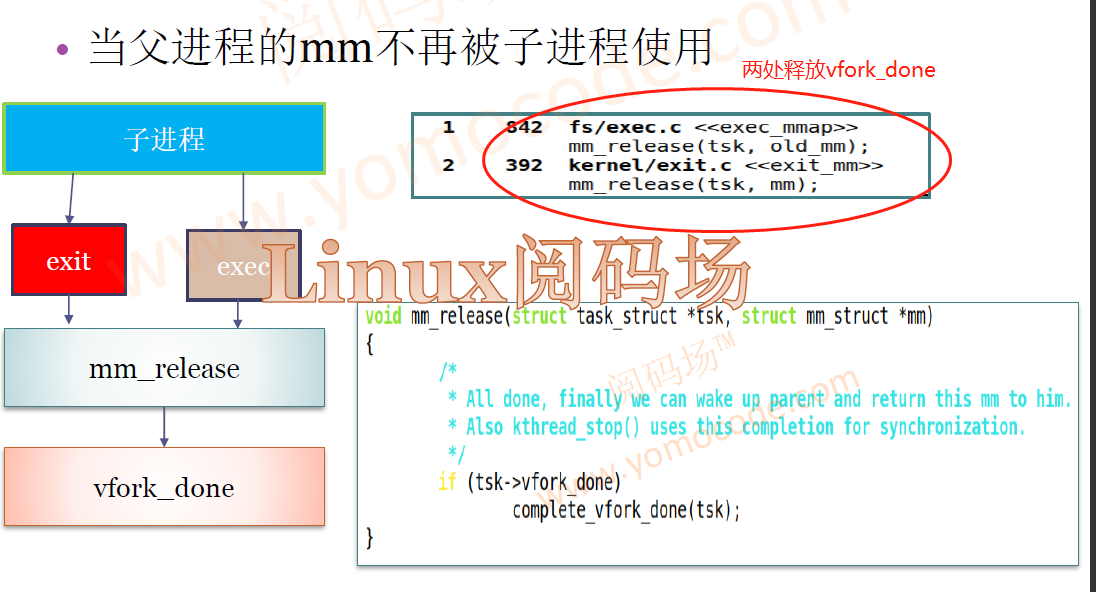
fork()依赖于硬件MMU实现，对于没有MMU的系统，用vfork代替,vfork没有写时拷贝；



vfork后，父进程什么时候继续执行？

当父进程MM不再被子进程共享时；

vfork创建子进程后，父进程阻塞等待vfork\_done，再继续往下执行；



Linux唤醒vfork\_done有两个地方，**进程退出**和**exec**

所以，在子进程创建后立马调用exec的场景，用vfork更合适（可以省掉fork的COW）；

调用exec之前指向同一个mm\_struct，**调用exec之后申请新的mm\_struct**。

## 3main函数的生命周期

### 3.1main函数即不是程序的入口，也不是出口

main函数进去之前，执行了动态库的构造函数；

main函数退出之后；

执行动态库的析构函数

flush stdio

atexit()注册的钩子函数；注意，后注册的钩子函数先执行；

\_\_attribute\_\_ ((constructor))

Xxx1()

{

…

}

\_\_attribute\_\_ ((destructor))

Xxx2()

{

…

}

### 3.2main函数正常退出方式有：

(1)正常/默认退出return 0; 会调用库函数exit()

(2)调用库函数exit()退出；

库函数exit()会执行flush io、调用atexit()注册的钩子函数，再调用系统调用\_exit(n) 退出, (n为return函数返回的退出码)；

strace ./a.out

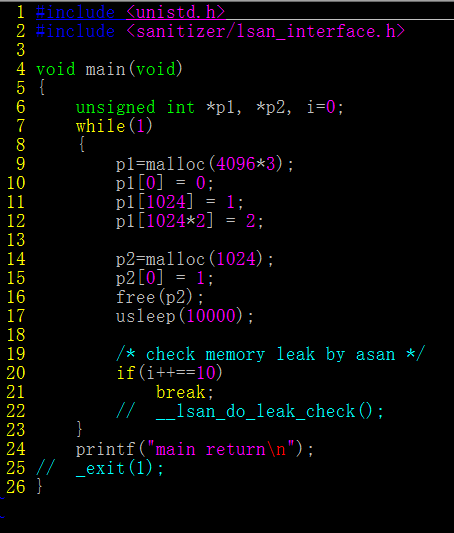
exit\_group(1):同TGID全部退出

进程被信号杀掉，会调用\_exit()，不会调用钩子函数，也不会执行flush，直接退出；；

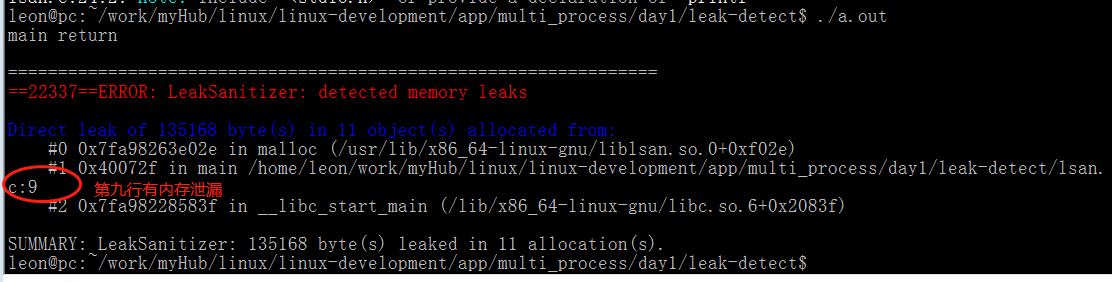
调用\_exit(n)，非正常返回；

**4.Lsan检测内存泄漏：**

实际上是调用钩子函数\_\_lsan\_do\_leak\_check()，它依赖返回，如果不返回则不调用；



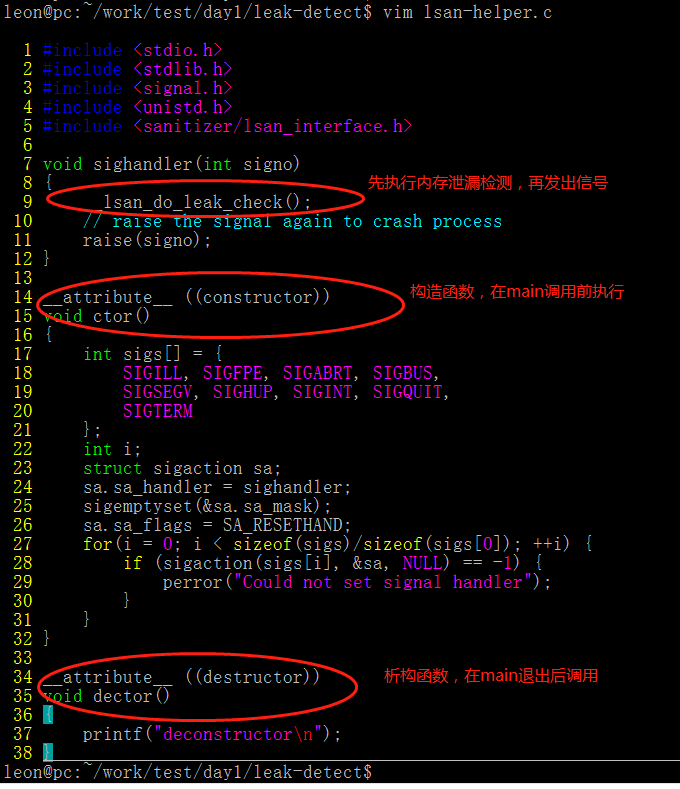
gcc lsan.c -g -fsanitize=leak



但是如果程序不退出，或者用信号杀死，都不会执行\_\_lsan\_do\_leak\_check()；

解决办法，**实际工程上，可以把内存泄漏检测放在捕捉信号函数上；**

gcc -shared -fPIC lsan-helper.c -o lsan-helper.so -lpthread -ldl



#include <sanitizer/lsan\_interface.h>

\_\_lsan\_do\_leak\_check()

LD\_PRELOAD强制加载动态库

LD\_PRELOAD=./lsan-helper.so ./a.out

# 第二部分 systemd和守护进程Daemon

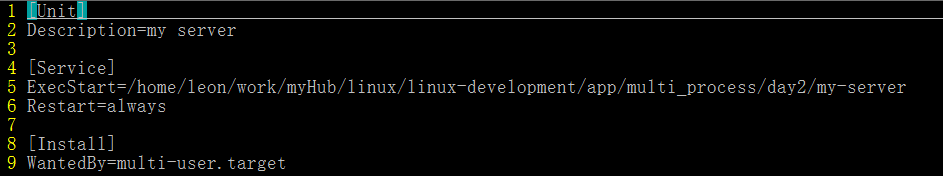
* 2.1 sys v init和systemd
* 2.2 daemon进程是如何形成的?
* 2.3 daemon进程的stdout, stderr到哪里去了？
* 2.4 daemon进程如何写log?
* 2.5 daemon与终端terminal
* 2.6 nohup命令
* 2.7 systemd和daemon之间是什么关系？

后台服务的启动主流方法：

自daemon这种方法逐步淘汰，违背了Do One Thing and Do It Well的设计哲学；

通过systemd来启动，是主流和建议的方法；

## 用systemd来实现守护进程



sudo cp my-server.service /lib/systemd/system

sudo systemctl enable my-server

sudo systemctl start my-server

sudo systemctl status my-server

cat /var/log/syslog

默认打印输出到系统日志文件

配置守护进程的重启，可以根据子进程死亡原因，配置是否重启

修改service文件后需要reload

sudo systemctl daemon-reload

sudo systemctl start my-server

leon@pc:~/work/myHub/linux/linux-development/app/multi\_process/day2$ sudo systemctl status my-server

● my-server.service - my server

Loaded: loaded (/lib/systemd/system/my-server.service; enabled; vendor preset: enabled)

Active: active (running) since 四 2020-10-29 17:30:26 CST; 9s ago

Main PID: 9885 (my-server)

Tasks: 1

Memory: 196.0K

CPU: 983us

CGroup: /system.slice/my-server.service

└─9885 /home/leon/work/myHub/linux/linux-development/app/multi\_process/day2/my-server

10月 29 17:30:26 pc systemd[1]: Started my server.

[Unit]

Description=my server

[Service]

ExecStart=/home/leon/work/myHub/linux/linux-development/app/multi\_process/day2/my-server

Restart=always

[Install]

WantedBy=multi-user.target

Sudo killall my-service

sudo systemctl status my-server

查看，发现my-service已经重启，进程号已经改变；

Server.service配置项

Type=oneshot 开机执行一次，比如扫描硬盘

查看syslog日志

journalctl -u my-server

参考：

https://wiki.archlinux.org/index.php/systemd\_(%E7%AE%80%E4%BD%93%E4%B8%AD%E6%96%87)

## 2自己实现daemon进程

#include <stdio.h>

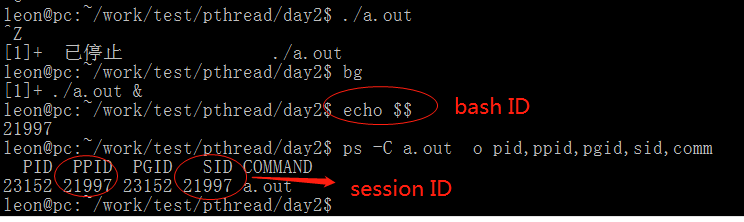
int main(int argc, const char \*argv[])

{

while(1);

return 0;

}



#include <stdio.h>

int main(int argc, const char \*argv[])

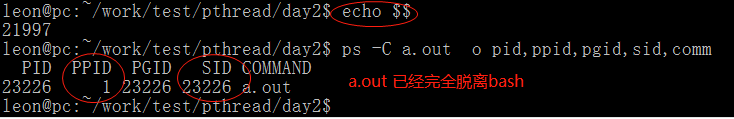
{

daemon(0,0);

while(1);

return 0;

}



#if defined(LIBC\_SCCS) && !defined(lint)

static char sccsid[] = "@(#)daemon.c 8.1 (Berkeley) 6/4/93";

#endif /\* LIBC\_SCCS and not lint \*/

#include <errno.h>

#include <fcntl.h>

#include <paths.h>

#include <unistd.h>

#include <sys/stat.h>

#include <device-nrs.h>

#include <not-cancel.h>

int

daemon(nochdir, noclose)

int nochdir, noclose;

{

int fd;

switch (\_\_fork()) {

case -1:

return (-1);

case 0:

break;

default:

\_exit(0);

}

if (\_\_setsid() == -1) //新的seesion leader，脱离terminal组

return (-1);

if (!nochdir)

(void)\_\_chdir("/"); //避免当前路径为busy

if (!noclose) {

struct stat64 st;

//IO重定向到/dev/null

if ((fd = open\_not\_cancel(\_PATH\_DEVNULL, O\_RDWR, 0)) != -1

&& (\_\_builtin\_expect (\_\_fxstat64 (\_STAT\_VER, fd, &st), 0)

== 0)) {

if (\_\_builtin\_expect (S\_ISCHR (st.st\_mode), 1) != 0

#if defined DEV\_NULL\_MAJOR && defined DEV\_NULL\_MINOR

&& (st.st\_rdev

== makedev (DEV\_NULL\_MAJOR, DEV\_NULL\_MINOR))

#endif

) {

(void)\_\_dup2(fd, STDIN\_FILENO);

(void)\_\_dup2(fd, STDOUT\_FILENO);

(void)\_\_dup2(fd, STDERR\_FILENO);

if (fd > 2)

(void)\_\_close (fd);

} else {

/\* We must set an errno value since no

function call actually failed. \*/

close\_not\_cancel\_no\_status (fd);

\_\_set\_errno (ENODEV);

return -1;

}

} else {

close\_not\_cancel\_no\_status (fd);

return -1;

}

}

return (0);

}

Daemon进程符合以下原则

(1)父进程通常是reaper

(2)不依附于terminal

(3)当前路径通常是/

(4)printf,perror…重定向

Daemon默认IO重定向到/dev/null

Systemd默认定向到cat /var/log/syslog

systemd很好解决了以上问题，并且更方便配置；推荐用systemd实现daemon程序。

pidof a.out

lasof –p xxxid

当前terminal bash

Echo $$

ps -C a.out o pid,ppid,pgid,sid,comm

bash退出时，会广播给所有组进程，bash下所有进程都推出；

ssh登录，伪bash，ssh退出之后，所有ssh开启的进程都退出；

kill –l

1) SIGHUP

## 3.nohup

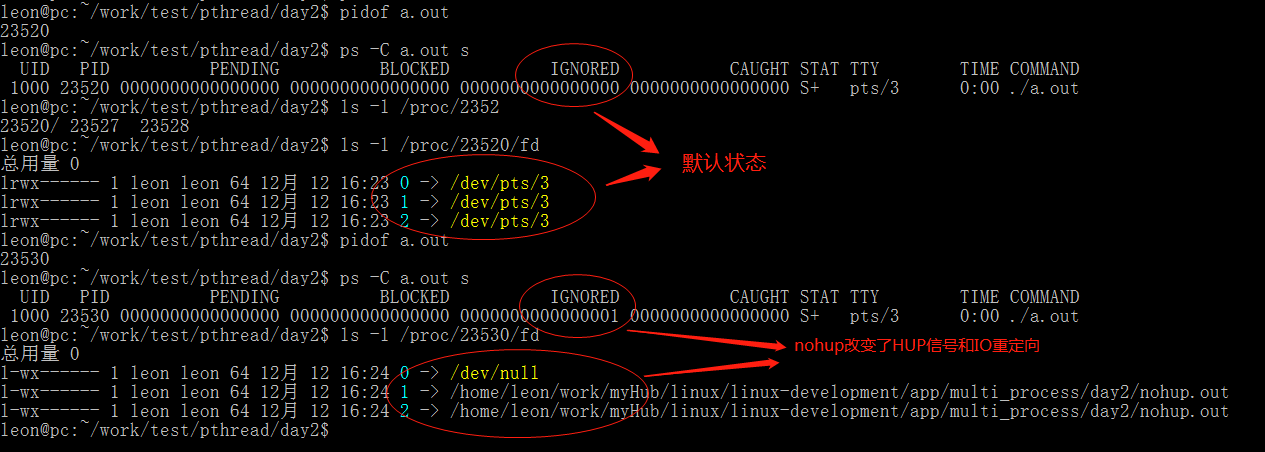
nohup ./a.out

signal(SIGHUP, SIG\_IGN);

./a.out 与nohup ./a.out对比

ps –C a.out s //查看相关信号

ls –l /proc/id/fd/ //IO重定向



忽略SIGHUP信号；

重定向input-> /dev/null

重定向outptu->nohup.out文件

Nohup效果与守护进程有类似，但不是守护进程；

## 4.sys V init和systmd

sys V init的服务是脚本式启动的，systmd是编写service文件并行启动的，程序实现

sys v init服务的脚本，实在/etc/rc.d/rc\*.d建立软连接；

cd /etc/init.d/

shell脚本实现

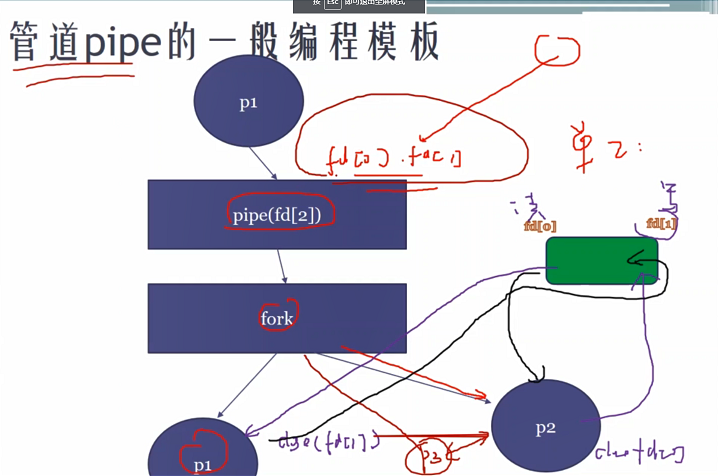
Android的init与systmd类似，C程序式实现的，非脚本启动；

# 第三部分 解决进程之间的通信问题(IPC)

1pipe：用于有亲缘关系的进程间通信

不使用大量数据传输；

父子进程，兄弟进程都可以继承fd描述符



Shell pipe原理

#include <stdio.h>

#include <unistd.h>

int main(int argc,char\* argv[])

{

int f\_des[2];

int pid;

if(argc!=3){

printf("Usage: %s comand1 comand2\n",argv[0]);

return 1;

}

if(pipe(f\_des)==-1){

perror("cannot create the IPC pipe");

return 1;

}

pid=fork();

if(pid==-1){

perror("cannot create new process");

return 1;

}else if(pid==0){

dup2(f\_des[1],STDOUT\_FILENO);//绑定标准输出到管道写端

close(f\_des[0]);

close(f\_des[1]);

if(execlp(argv[1],argv[1],NULL)==-1){

perror("in child process,cannot execute the command");

return 1;

}

return 1;

}else {

dup2(f\_des[0],STDIN\_FILENO);//绑定管道读端为标准输入

close(f\_des[0]);

close(f\_des[1]);

if(execlp(argv[2],argv[2],NULL)==-1){

perror("in parent process,cannot execute the command");

return 1;

}

return 1;

}

return 0;

}

## 2信号的处理方法

### 2.1捕获、忽略与缺省，block，pending

#include <stdio.h>

#include <unistd.h>

#include <signal.h>

void sighandler(int sig)

{

printf("got the signal...\n");

}

int main(void)

{

if(signal(SIGINT, /\*sighandler\*/SIG\_IGN)==SIG\_ERR){

perror("cannot reset the SIGINT signal handler");

return 1;

}

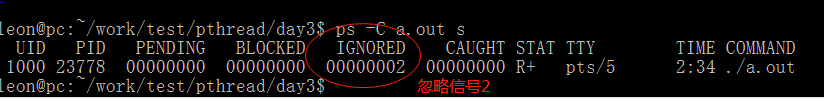
while(1);

return 0;

}

### 2.2修改信号默认行为：

Kill -l



除了9/19信号，其他都可以被捕获，修改默认行为；

#include <signal.h>

#include <setjmp.h>

#include <stdarg.h>

#include <stdlib.h>

#include <stdio.h>

static jmp\_buf env;

void segvSignal(int sig)

{

longjmp(env, 1); 返回大于0

}

int main(int argc, char \*\*argv)

{

int r = setjmp(env);//保存现场，返回0

if (r == 0) {

signal(SIGSEGV, segvSignal);

printf("making segment fault\n");

int \*p = NULL;

\*p = 0;

} else {

printf("after segment fault\n");

}

return 0;

}

### 2.3信号会引起RACE CONFITION

信号是异步的；

信号访问其他线程正在访问的资源…

可以考虑屏蔽信号

#include <signal.h>

#include <stdio.h>

struct two\_long { long a, b; } data;

void signal\_handler(int signum){

printf ("%ld, %ld\n", data.a, data.b);

alarm (1);

}

int main (void){

static struct two\_long zeros = { 0, 0 }, ones = { 1, 1 };

signal (SIGALRM, signal\_handler);

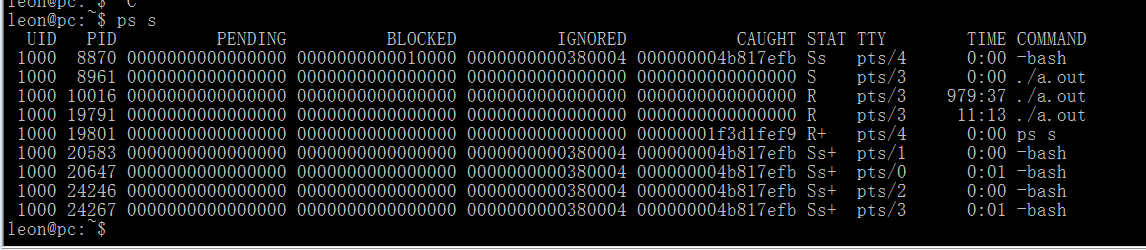
data = zeros;

alarm (1);

while (1)

{data = zeros; data = ones;}

}



在可能竟态的数据上屏蔽信号；

#include <signal.h>

#include <stdio.h>

struct two\_int { long a, b; } data;

void signal\_handler(int signum){

printf ("%ld, %ld\n", data.a, data.b);

alarm (1);

}

int main (void){

sigset\_t bset, oset;

static struct two\_int zeros = { 0, 0 }, ones = { 1, 1 };

signal (SIGALRM, signal\_handler);

data = zeros;

alarm (1);

sigemptyset(&bset);

sigaddset(&bset, SIGALRM);

while (1) {

if (sigprocmask(SIG\_BLOCK, &bset, &oset) != 0)//屏蔽信号

printf("!! Set mask failed\n");

data = zeros;

if (sigprocmask(SIG\_SETMASK, &oset, NULL) != 0)//解屏蔽

printf("!! Set mask failed\n");

if (sigprocmask(SIG\_BLOCK, &bset, &oset) != 0)

printf("!! Set mask failed\n");

data = ones;

if (sigprocmask(SIG\_SETMASK, &oset, NULL) != 0)

printf("!! Set mask failed\n");

}

}

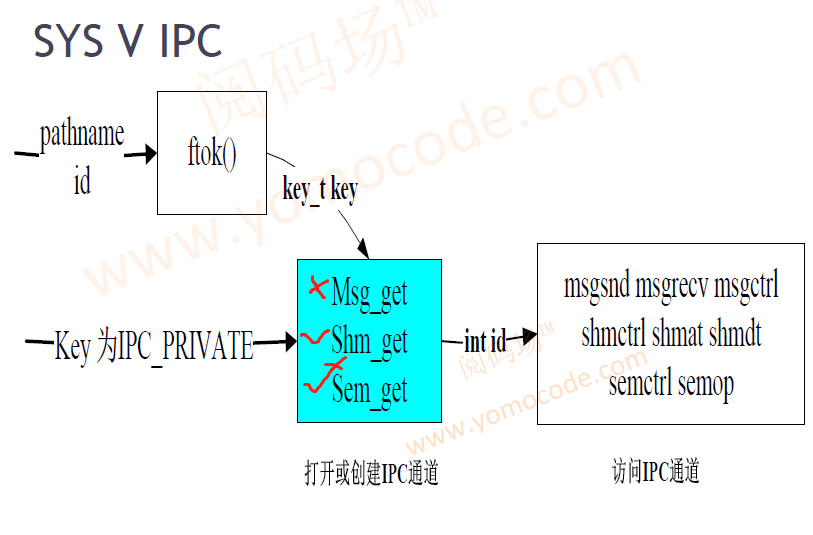
### 3 SYS V IPC接口：信号量、共享内存、消息队列

比较老套，基本都不用了

类似于PIPE，但是可用于处理无亲缘关系的进程间通信；

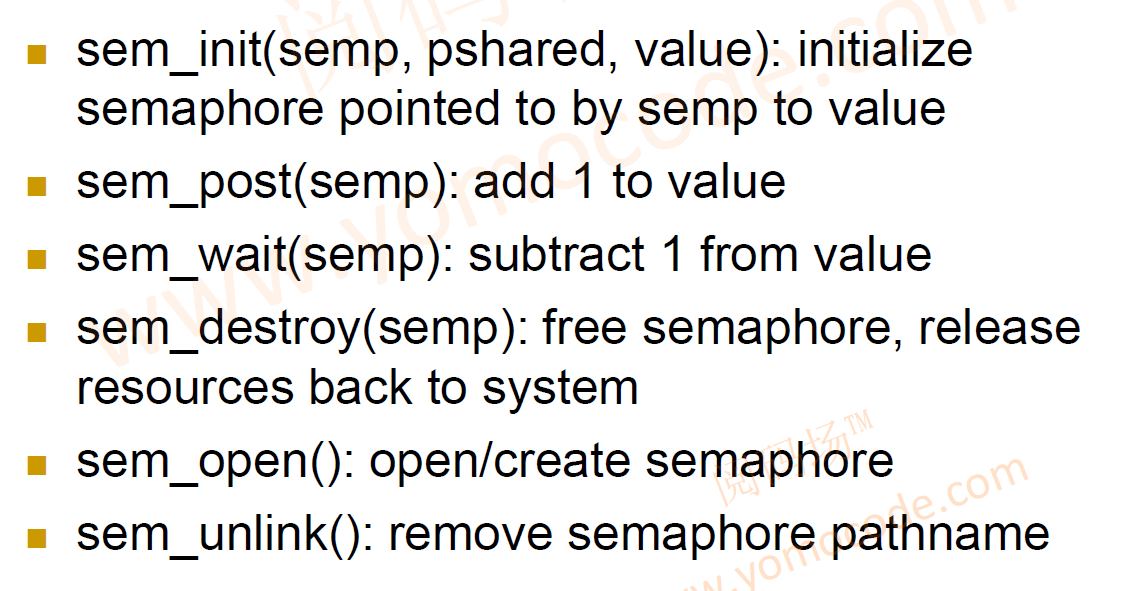
Ubuntu查看sys V ipc

$ipcs



## 4.POSIX接口

### 4.1信号量sem



### 4.2共享内存

Ipcs

Sys V

Posix，接口更简洁，更符合主流

有亲缘关系，直接继承mmap地址，映射为匿名页，共享；

无亲缘关系，用相同路径文件，映射内存；

内核可以配置SYSVIPC=y,兼容老程序

Posix-shm-w.c

#include <sys/mman.h>

#include <sys/stat.h> /\* For mode constants \*/

#include <fcntl.h> /\* For O\_\* constants \*/

#include <stdlib.h> /\* For O\_\* constants \*/

#include <string.h>

#define MAX\_LEN 10000

struct region { /\* Defines "structure" of shared memory \*/

int len;

char buf[MAX\_LEN];

};

struct region \*rptr;

int fd;

int main(int argc, char \*\*argv)

{

/\* Create shared memory object and set its size \*/

fd = shm\_open(argv[1], O\_CREAT | O\_RDWR, S\_IRUSR | S\_IWUSR);

if (fd == -1) {

perror("shm open failed\n"); /\* Handle error \*/;

return -1;

}

if (ftruncate(fd, sizeof(struct region)) == -1) {

perror("ftruncate failed\n") /\* Handle error \*/;

return -1;

}

/\* Map shared memory object \*/

rptr = mmap(NULL, sizeof(struct region),

PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0);

if (rptr == MAP\_FAILED) {

perror("mmap failed\n") /\* Handle error \*/;

return -1;

}

/\* Now we can refer to mapped region using fields of rptr;

for example, rptr->len \*/

memset(rptr->buf, 0, MAX\_LEN);

rptr->len=1000;

strcpy(rptr->buf, "hello world");

while(1);

}

Posix-shm-r.c

#include <sys/mman.h>

#include <sys/stat.h> /\* For mode constants \*/

#include <fcntl.h> /\* For O\_\* constants \*/

#include <stdlib.h> /\* For O\_\* constants \*/

#include <string.h>

#include <stdio.h>

#define MAX\_LEN 10000

struct region { /\* Defines "structure" of shared memory \*/

int len;

char buf[MAX\_LEN];

};

struct region \*rptr;

int fd;

int main(int argc, char \*\*argv)

{

/\* Create shared memory object and set its size \*/

fd = shm\_open(argv[1], O\_RDWR, S\_IRUSR | S\_IWUSR);

if (fd == -1) {

perror("shm open failed\n"); /\* Handle error \*/;

return -1;

}

if (ftruncate(fd, sizeof(struct region)) == -1) {

perror("ftruncate failed\n") /\* Handle error \*/;

return -1;

}

/\* Map shared memory object \*/

rptr = mmap(NULL, sizeof(struct region),

PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0);

if (rptr == MAP\_FAILED) {

perror("mmap failed\n") /\* Handle error \*/;

return -1;

}

/\* Now we can refer to mapped region using fields of rptr;

for example, rptr->len \*/

printf("%d %s\n", rptr->len, rptr->buf);

}

有亲缘关系的共享内存：

#include <sys/mman.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

int main(int argc, char \*\*argv)

{

pid\_t pid;

char \*p;

p = (char \*)mmap(NULL, 4096, PROT\_READ | PROT\_WRITE,

MAP\_SHARED | MAP\_ANONYMOUS, -1, 0);

//相同vma,shared,物理内存相同；若改为private，虚拟地址相同，物理地址不同

strcpy(p, "XXXXXXXXXXXXXXXXX");

pid = fork();

if (pid == -1) {

exit(-1);

} else if (pid == 0) { //shared vma

sprintf(p, "%s", "Hello World");

munmap(p, 4096);

\_exit(0);

} else {

sleep(2);

printf("%s\n", p);

munmap(p, 4096);

}

return 0;

}

## 4. UNIX Domain Socket IPC

现代Linux用的普遍的是UNIX domain socket

socketAPI原本为网络通讯设计的，但后来在socket的框架上发展出一种IPC机制UNIX domain socket，socketAPI也可以用于同一台主机的进程通讯，但是UNIX domain socket用于IPC更有效率：它不需要经过网络协议栈，不需要打包拆包、计算校验、维护序号和应答等，只是将应用层数据从一个进程拷贝到另一个进程。

UNIX domain socket也提供了面向流和数据包两种接口，类似于TCP/UDP，但是面向消息的UNIX domain socket也是可靠的，消息既不会丢失也不会乱序；

UNIX domain socket是全双工的，相比其他IPC机制有明显的优越性，目前已成为使用最广泛的IPC机制，比如X Windows和GUI之间通讯；

查看UNIX.domain

netstat –an |grep UNIX

pidof -p id

5.Socketpair:类似Pipe，但支持双工

适合有亲缘关系进程间通信，

Lsof

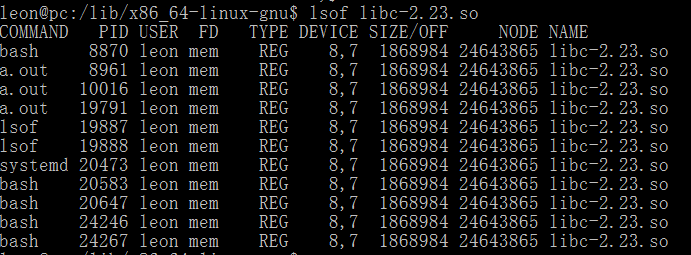
看进程打开的文件；

看文件被什么进程打开；

看IPC;

看socket;

可以查看动态库被那些进程打开



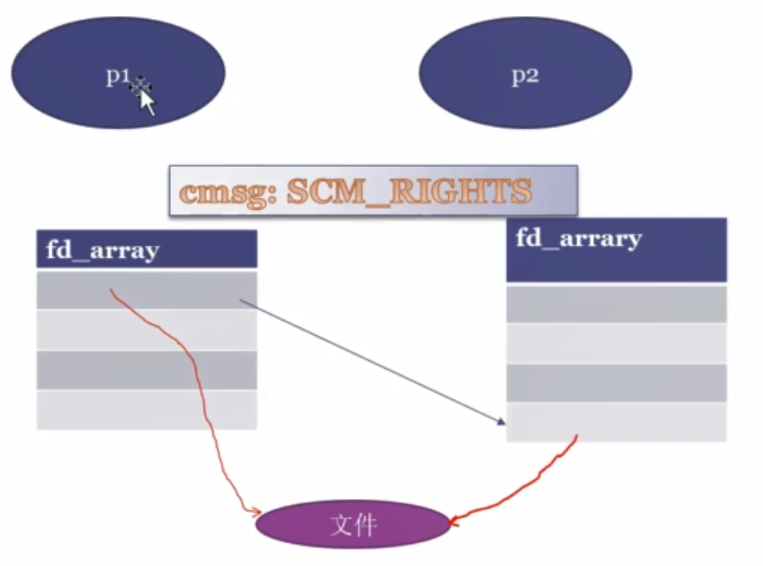
### Day5文件描述符fd跨进程共享与共享内存

* 5.1 跨进程共享文件描述符fd
* 5.2 memfd\_create与共享内存
* 5.3 dma-buffer:Multimedia/Graphis数据跨进程共享
* 5.4 dma-buffer跨进程共享
* 5.5 dma-buffer跨设备共享
* 5.1 跨进程共享文件描述符fd

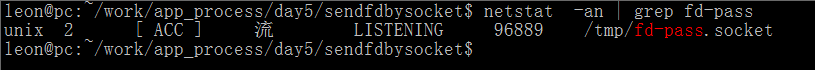
可以通过socket将打开的fd(包括文件，共享内存)发送给另外一个进程，两个进程fd可能可能不等，但指向同一个文件(内存)。

SCM: socket control massage

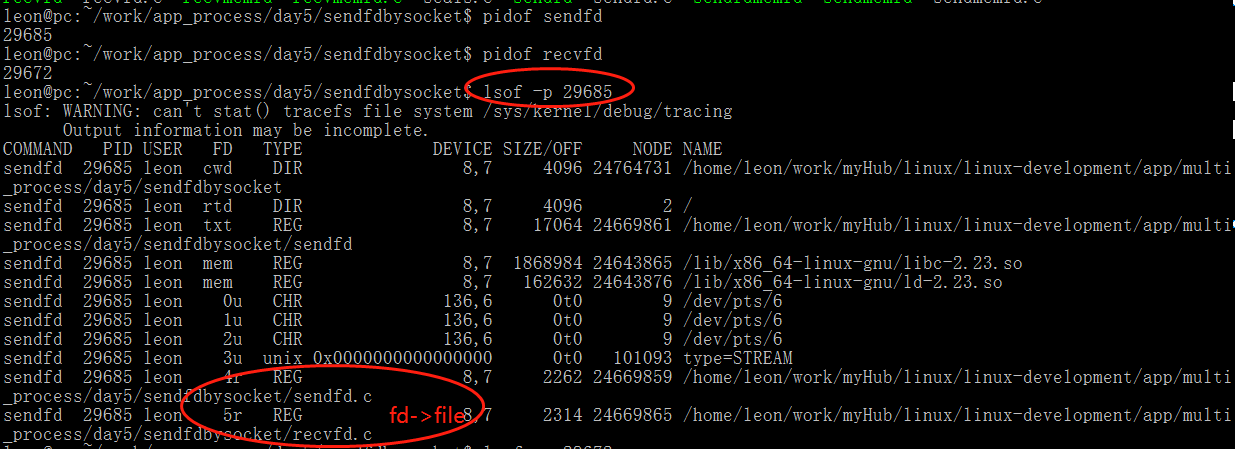
将Fd封装成一个特殊的cmsg，发送

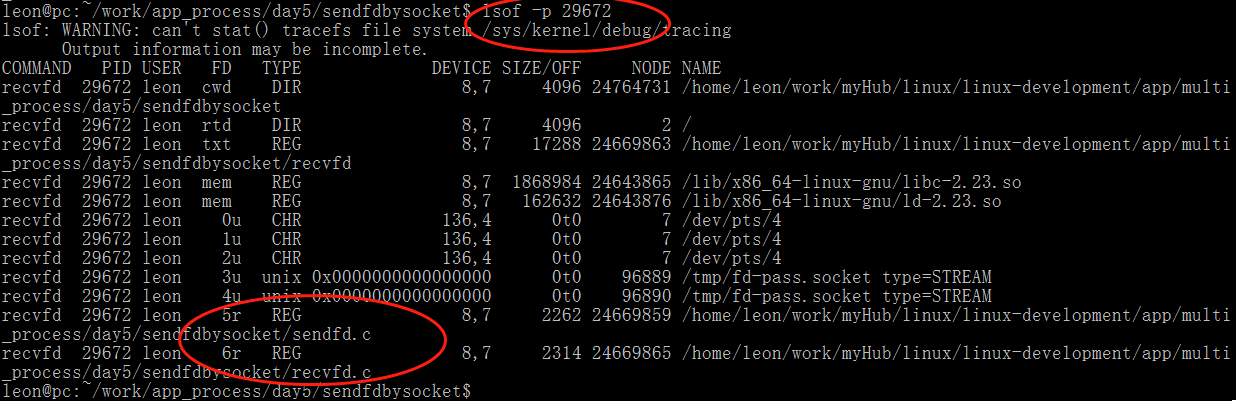
|

netstat -an |grep fd-pass



查询进程打开的文件:lsof –o pid





由上知，发送端用4/5，接收端5/6对应实际的文件sendfd.c/recvfd.c

## 5.2 memfd\_create与共享内存

创建个匿名文件，通过socket将fd发送给其他进程，实现共享内存；

Seals

对共享内存，添加封印；

Man memfd\_create

5.3 dma-buffer:Multimedia/Graphis数据跨进程共享

5.4 dma-buffer跨进程共享

5.5 dma-buffer跨设备共享

# Day6多进程调试方法

## 6.1 gdb基础

### (1).基础命令

gcc test1.c -o test1 –g

gdb test1

start

quit



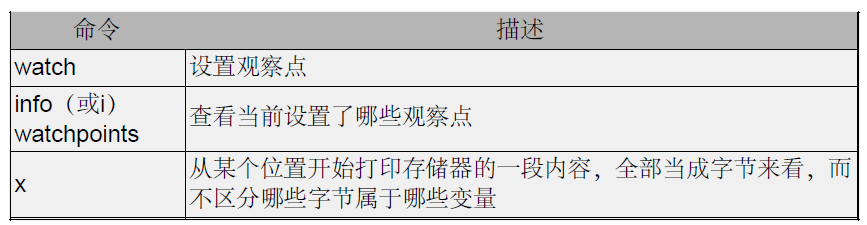
(2)断点breakpoint，

执行到某一行，停下



### (3)观察点

某个变量改变，就停下



### (4)段错误

Bt查看栈信息

这可以算是一条规律，如果某个函数中发生访问越界，很可能并不立即产生段错误，而在函数返回时却产生段错误。

## 6.2多进程的GDB

默认情况下，

(gdb) show detach-on-fork

Whether gdb will detach the child of a fork is on.

(gdb) show follow-fork-mode

Debugger response to a program call of fork or vfork is "parent".

gdb跟踪一个进程，且跟踪父进程；

(1)设置同时跟踪父子进程，

set detach-on-fork off

在不同进程切换

i inferiors

inferiors xx

finish：完成当函数帧；

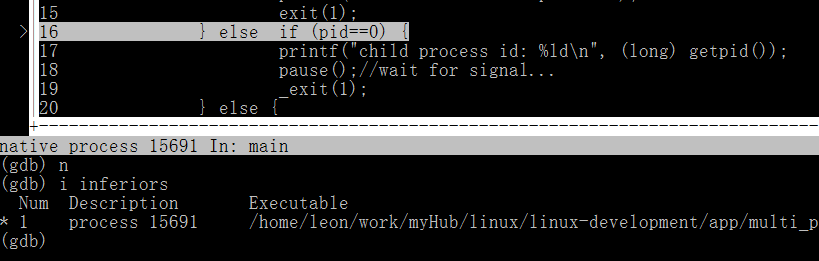
### (2)设置默认跟踪进程

set follow-fork-mode parent/child

查看栈帧信息

bt

ctrl+x a调出gdbtui。



可以启用另外一个gdb跟踪某进程

sudo gdb attach 15695

## 6.3进程的coredump

cat /proc/sys/kernel/core\_pattern

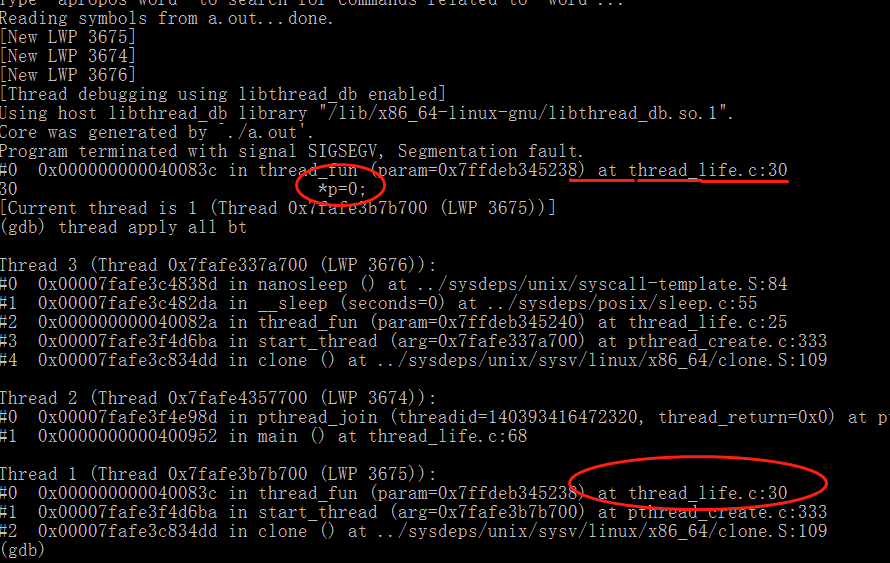
/usr/share/apport/apport %p %s %c %d %P %E

启动core文件

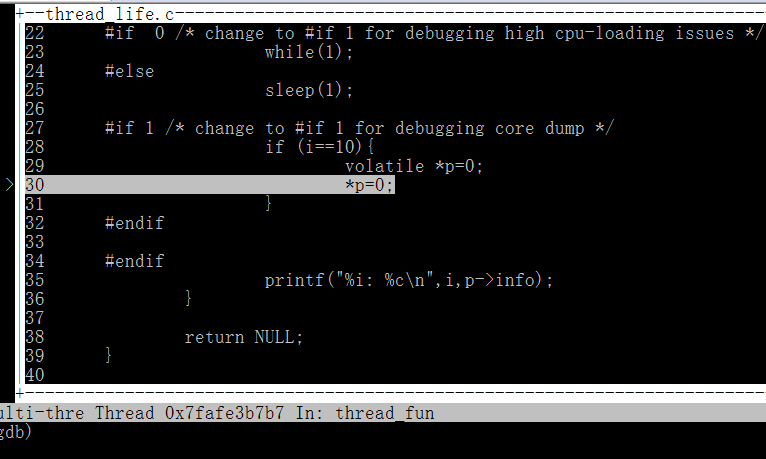
gdb ./a.out –c xxx-coredump

查看所有线程栈帧

Thread apply all bt



Ctrl+x+a



coredump工程实施：

jira coredump存放服务器，保存thread apply all bt信息；

可以制作工具，当coredump产生时，自动产生jira BUG，自动填写comment信息

minicoredumper,压缩的core文件：

sudo apt install minicoredumper

**6.4 strace,ltrace与多进程**

跟踪多进程，shell\_pipe.c

strace -f ./a.out ls cat （-f 强制显示多进程，默认跟踪父进程）；

-f follow forks

-b execve detach on execvf syscall

$strace -b execve ./a.out ls cat //只跟踪a.out

**6.5 socat侦听unix domain socket**

mv /tmp/UNIX.domain /tmp/UNIX.domain.original

socat -t100 -x -v UNIXLISTEN:/tmp/UNIX.domain,mode=777,reuseaddr,fork UNIXCONNECT:/tmp/UNIX.domain.original

时序

step1, 启动server 绑定/tmp/Unix.domain

step2, 改变/tmp/Unix.domain名字；

step3, 启动socat，重新创建/tmp/Unix.domain

step4,启动client;

6.6进程间通信的调试

软件自带monitor,dbus-monitor

$sudo dbus-monitor –system

WAYLAND\_DEBUG

$WAYLAND\_DEBUG=1 transmission-gtk wayland

6.7查看日志

Direct write

rsyslogd

systemd-journald