<https://blog.csdn.net/beckdon/article/details/48491909>

这里我们主要介绍进程的状态,进程的状态可以通过/proc/PID/status来查看,也可以通过/proc/PID/stat来查看.

如果说到工具大家用的最多的ps也可以看到进程的信息.这里我们通过/proc/PID/status来分析进程的信息.

在2.6.18之后的内核,多了capibilty/cpusets等信息.

查看进程状态信息如下:

more status

Name:   rsyslogd

State:  S (sleeping)

Tgid:   987

Pid:    987

PPid:   1

TracerPid:      0

Uid:    0       0       0       0

Gid:    0       0       0       0

Utrace: 0

FDSize: 32

Groups:

VmPeak:    36528 kB

VmSize:    36528 kB

VmLck:         0 kB

VmHWM:      1432 kB

VmRSS:      1420 kB

VmData:    33980 kB

VmStk:        88 kB

VmExe:       320 kB

VmLib:      2044 kB

VmPTE:        56 kB

VmSwap:        0 kB

Threads:        3

SigQ:   1/7954

SigPnd: 0000000000000000

ShdPnd: 0000000000000000

SigBlk: 0000000000000000

SigIgn: 0000000001001206

SigCgt: 0000000180014c21

CapInh: 0000000000000000

CapPrm: ffffffffffffffff

CapEff: ffffffffffffffff

CapBnd: ffffffffffffffff

Cpus\_allowed:   3

Cpus\_allowed\_list:      0-1

Mems\_allowed:   1

Mems\_allowed\_list:      0

voluntary\_ctxt\_switches:        1

nonvoluntary\_ctxt\_switches:     0

Name:   rsyslogd

解释:进程名

State:  S (sleeping)

解释:进程的状态我们前文已经做了很详细的分析,各进程的状态代表的意义如下:

R (running)", "S (sleeping)", "D (disk sleep)", "T (stopped)", "T(tracing stop)", "Z (zombie)", or "X (dead)"

Tgid:   987

解释:Tgid是线程组的ID,一个线程一定属于一个线程组(进程组).

Pid:    987

解释:这个是进程的ID,更准确的说应该是线程的ID.

例如:

UID        PID  PPID   LWP  C NLWP STIME TTY          TIME CMD

root       987     1   987  0    3 00:18 ?        00:00:00 /sbin/rsyslogd -c 4

root       987     1   989  0    3 00:18 ?        00:00:00 /sbin/rsyslogd -c 4

root       987     1   990  0    3 00:18 ?        00:00:00 /sbin/rsyslogd -c 4

注:

/proc/pid/status中的Pid就是ps命令的LWP列输出,PID一列其实是进程组,而LWP是轻量级进程,也就是线程,因为所有的进程必须一个线程,那就是它自己.

PPid:   1

解释:当前进程的父进程

TracerPid:      0

解释:跟踪当前进程的进程ID,如果是0,表示没有跟踪.

例如:

用strace跟踪top程序

strace top

查看top进程

ps -axjf

PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND

 2491  2500  2500  2491 pts/2     2500 S+       0   0:00          \\_ strace top

 2500  2501  2500  2491 pts/2     2500 S+       0   0:00              \\_ top

查看top进程的TracerPid位

cat /proc/2501/stat

stat    statm   status

test1:/proc/2431# cat /proc/2501/status

Name:   top

State:  S (sleeping)

Tgid:   2501

Pid:    2501

PPid:   2500

TracerPid:      2500

Uid:    0       0       0       0

Gid:    0       0       0       0

解释:

第一列数字(RUID):实际用户ID,指的是进程执行者是谁.

第二列数字(EUID):有效用户ID,指进程执行时对文件的访问权限.

第三列数字(SUID):保存设置用户ID,作为effective user ID的副本,在执行exec调用时后能重新恢复原来的effectiv user ID.

第四列数字(FSUID):目前进程的文件系统的用户识别码.一般情况下,文件系统的用户识别码(fsuid)与有效的用户识别码(euid)是相同的.

这里重点说明RUID和EUID,我们用test用户启动top,如下:

终端1)

su - test

top

查看该进程的EUID和RUID,如下:

终端2)

cat /proc/`pgrep top|grep -v grep`/status

前面略

Uid:    1002    1002    1002    1002

Gid:    1003    1003    1003    1003

后面略

注:这里我们看到进程的RUID和EUID都变成了1002.

我们将程序top加上setuid权限,如下:

chmod +s /usr/bin/top

重新运行top程序,并查看它的RUID和EUID,如下:

cat /proc/`pgrep top|grep -v grep`/status

前面略

Uid:    1002    0       0       0

Gid:    1003    0       0       0

后面略

注:我们看到RUID还是1002,说明程序是由test用户(UID=1002)启动的,而程序设定了setuid,那么在程序运行时是用程序的owner权限来运行程序,而不是启动的用户权限.

由于top的owner是root,那么它的EUID是0.

FDSize: 32

解释:

FDSize是当前分配的文件描述符,这个值不是当前进程使用文件描述符的上限.

我们看到这里是32,但实际并没有分配32个文件,如下:

ls -l /proc/`pgrep rsyslogd|grep -v grep`/fd

total 0

lrwx------ 1 root root 64 2011-04-20 20:03 0 -> socket:[5741]

l-wx------ 1 root root 64 2011-04-20 20:03 1 -> /var/log/auth.log

l-wx------ 1 root root 64 2011-04-20 20:03 10 -> /var/log/mail.err

l-wx------ 1 root root 64 2011-04-20 20:03 11 -> /var/log/news/news.crit

l-wx------ 1 root root 64 2011-04-20 20:03 12 -> /var/log/news/news.err

l-wx------ 1 root root 64 2011-04-20 20:03 13 -> /var/log/news/news.notice

l-wx------ 1 root root 64 2011-04-20 20:03 14 -> /var/log/debug

l-wx------ 1 root root 64 2011-04-20 20:03 15 -> /var/log/messages

lrwx------ 1 root root 64 2011-04-20 20:03 16 -> /dev/xconsole

lr-x------ 1 root root 64 2011-04-20 20:03 17 -> /proc/kmsg

l-wx------ 1 root root 64 2011-04-20 20:03 2 -> /var/log/syslog

l-wx------ 1 root root 64 2011-04-20 20:03 3 -> /var/log/daemon.log

l-wx------ 1 root root 64 2011-04-20 20:03 4 -> /var/log/kern.log

l-wx------ 1 root root 64 2011-04-20 20:03 5 -> /var/log/lpr.log

l-wx------ 1 root root 64 2011-04-20 20:03 6 -> /var/log/mail.log

l-wx------ 1 root root 64 2011-04-20 20:03 7 -> /var/log/user.log

l-wx------ 1 root root 64 2011-04-20 20:03 8 -> /var/log/mail.info

l-wx------ 1 root root 64 2011-04-20 20:03 9 -> /var/log/mail.warn

我们看到这里只用到了18个文件描述符.而如果超过32个文件描述符,将以32进行递增,如果是64位系统,将以64进行递增.

FDSize这个值不会减少,如果我们程序打开了300个文件,并不会因为关闭文件,而减少FDSize这个值.

Groups: 0

解释:

这里的groups表示启动这个进程的用户所在的组.

我们当前的用户test,现在在两个组(1000,2000)里面,如下:

id

uid=1002(test) gid=1002(nagcmd) groups=1000(chenkuo),1002(nagcmd)

用test用户启动top程序,并查看它的groups,如下:

终端1

top

终端2

cat /proc/`pgrep top|grep -v grep`/status

截取信息如下:

Groups: 1000 1002

VmPeak:    36528 kB

解释:这里的VmPeak代表当前进程运行过程中占用内存的峰值.

我们用下面的程序申请内存,然后释放内存,最后通pause()函数中止程序的运行,程序源码如下:

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <unistd.h>

int

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

{

        if (argc != 2)

                exit (0);

        size\_t mb = strtoul(argv[1],NULL,0);

        size\_t nbytes = mb \* 0x100000;

        char \*ptr = (char \*) malloc(nbytes);

        if (ptr == NULL){

                perror("malloc");

                exit (EXIT\_FAILURE);

        }

        printf("allocated %d mb\n", mb);

        free(ptr);

        pause();

        return 0;

}

gcc callmem.c -o callmem

./callmem 10

allocated 10 mb

终端2

我们打开status文件,查看VmPeak值,如下:

cat /proc/`pgrep callmem|grep -v grep`/status

Name:   callmem

State:  S (sleeping)

Tgid:   2930

Pid:    2930

PPid:   2831

TracerPid:      0

Uid:    1002    1002    1002    1002

Gid:    1002    1002    1002    1002

FDSize: 256

Groups: 1000 1002

VmPeak:    11852 kB

VmSize:     1608 kB

VmLck:         0 kB

VmHWM:       396 kB

VmRSS:       396 kB

VmData:       28 kB

VmStk:        84 kB

VmExe:         4 kB

VmLib:      1468 kB

VmPTE:        12 kB

下面略

注:我们看到程序申请了10240kb(10MB)的内存,VmPeak的值为11852kb,为什么不是10MB呢,因为除了我们申请的内存外,程序还会为加载动态链接库而占用内存.

VmSize:    36528 kB

解释:VmSize代表进程现在正在占用的内存

这个值与pmap pid的值基本一致,如果略有不同,可能是内存裂缝所造成的.

VmLck:         0 kB

解释:VmLck代表进程已经锁住的物理内存的大小.锁住的物理内存不能交换到硬盘.

我们用下面的程序进行测试,如下:

#include <stdio.h>

#include <sys/mman.h>

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

{

        char array[2048];

        if (mlock((const void \*)array, sizeof(array)) == -1) {

                perror("mlock: ");

                return -1;

        }

        printf("success to lock stack mem at: %p, len=%zd\n",

                        array, sizeof(array));

        sleep(60);

        if (munlock((const void \*)array, sizeof(array)) == -1) {

                perror("munlock: ");

                return -1;

        }

        printf("success to unlock stack mem at: %p, len=%zd\n",

                        array, sizeof(array));

        return 0;

}

编译后运行:

gcc memlock.c -o memlock

我们这里将2048个字节的数组地址空间锁定到了物理内存中.

接下来我们看下Vmlck值的变化,如下:

cat /proc/`pgrep memlock|grep -v grep`/status

Name:   memlock

State:  S (sleeping)

Tgid:   3249

Pid:    3249

PPid:   3139

TracerPid:      0

Uid:    0       0       0       0

Gid:    0       0       0       0

FDSize: 256

Groups: 0

VmPeak:     1624 kB

VmSize:     1608 kB

VmLck:         4 kB

VmHWM:       356 kB

VmRSS:       356 kB

VmData:       28 kB

VmStk:        84 kB

VmExe:         4 kB

VmLib:      1468 kB

VmPTE:        16 kB

我们看到Vmlck的值为4Kb,这是因为分配的最少单位是4KB,以后每次递增都是4KB的整数倍.

VmHWM:      1432 kB

VmRSS:      1420 kB

解释:

VmHWM是程序得到分配到物理内存的峰值.

VmRSS是程序现在使用的物理内存.

我们用下面的程序进行测试,如下:

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <unistd.h>

int

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

{

        if (argc != 2)

                exit (0);

        size\_t mb = strtoul(argv[1],NULL,0);

        size\_t nbytes = mb \* 0x100000;

        char \*ptr = (char \*) malloc(nbytes);

        if (ptr == NULL){

                perror("malloc");

                exit (EXIT\_FAILURE);

        }

        size\_t i;

        const size\_t stride = sysconf(\_SC\_PAGE\_SIZE);

        for (i = 0;i < nbytes; i+= stride) {

                ptr[i] = 0;

        }

        printf("allocated %d mb\n", mb);

        pause();

        return 0;

}

编译:

gcc callmem.c -o test

注意这个程序在每页都修改一个字节的数据,导致系统必须为它分配占用物理内存.

首先我们查看当前的内存,如下:

free -m

             total       used       free     shared    buffers     cached

Mem:           503         18        484          0          0          5

-/+ buffers/cache:         12        490

Swap:         7632          7       7624

我们看到当前有490MB的空闲物理内存.

运行callmem分配450MB的物理内存,如下:

./test 450&

[1] 2402

allocated 450 mb

我们查看进程的VmHWM和VmRSS,如下:

cat /proc/`pgrep test`/status

略

VmHWM:    461208 kB

VmRSS:    461208 kB

略

我们看到此时VmHWM和VmRSS是一样的,表示占用了460MB左右的物理内存(因为它会用到动态链接库等).

下面我们查看当前的内存使用情况,如下:

free -m

             total       used       free     shared    buffers     cached

Mem:           503        470         33          0          0          6

-/+ buffers/cache:        463         40

Swap:         7632          7       7625

我们看到还有40MB空闲物理内存.

我们下面再申请100MB的内存,此时系统会通过物理内存和SWAP的置换操作,把第1次运行的test进程所占用的物理内存置换到SWAP,把空出来的物理内存分配给第2次运行的程序,如下:

mv test test1

./test1 100&

[1] 2419

allocated 100 mb

再次查看test进程所占用的物理内存,如下:

cat /proc/`pgrep test`/status

略

VmHWM:    461208 kB

VmRSS:    386704 kB

略

最后我们看到VmHWM没有变化,因为它表示的是该进程所占用物理内存的峰值,不会因为把内存置换到SWAP,而做改变.

而VmRSS则由461208KB变成了386704KB,说明它占用的物理内存因为置换所以减少.

VmData:    33980 kB

VmStk:        88 kB

VmExe:       320 kB

VmLib:      2044 kB

解释:

VmData:表示进程数据段的大小.

VmStk:表示进程堆栈段的大小.

VmExe:表示进程代码的大小.

VmLib:表示进程所使用LIB库的大小.

关于代码段,堆栈段,数据段:

代码段可以为机器中运行同一程序的数个进程共享

堆栈段存放的是子程序（函数）的返回地址、子程序的参数及程序的局部变量

数据段则存放程序的全局变量、常数以及动态数据分配的数据空间（比如用malloc函数申请的内存）

与代码段不同，如果系统中同时运行多个相同的程序，它们不能使用同一堆栈段和数据段.

注意:

堆栈段代表的是程序中的堆区(stack),堆区一般是编译器自动分配释放的.

我们用malloc申请的内存,它占用的其实是栈区(heap),栈区一般是程序员自已分配释放的,而栈区在这里属于数据段,所以我们看到上面测试程序通过调用malloc函数后,VmData一值有了很大的变化.

VmPTE:        56 kB

VmSwap:        0 kB

VmPTE:        56 kB

解释:

占用的页表的大小.

VmSwap:0 kB

解释:

进程占用Swap的大小.

Threads:        3

解释:

表示当前进程组有3个线程.

SigQ:   1/7954

解释:

表示当前待处理信号的个数,我们用下面和程序进行测试,如下:

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <signal.h>

#include <unistd.h>

volatile int done = 0;

void handler (int sig)

{

  const char \*str = "handled...\n";

  write (1, str, strlen(str));

  done = 1;

}

void child(void)

{

  int i;

  for (i = 0; i < 3; i++){

    kill(getppid(), SIGRTMIN);

    printf("child - BANG!\n");

  }

  exit (0);

}

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

{

  signal (SIGRTMIN, handler);

  sigset\_t newset, oldset;

  sigfillset(&newset);

  sigprocmask(SIG\_BLOCK, &newset, &oldset);

  pid\_t pid = fork();

  if (pid == 0)

  child();

  printf("parent sleeping \n");

  int r = sleep(30);

  printf("woke up! r=%d\n", r);

  sigprocmask(SIG\_SETMASK, &oldset, NULL);

  while (!done){

  };

  printf("exiting\n");

  exit(0);

}

编译:

gcc sig.c -o sig

本程序会发达三次信号,此后进入sleep,我们可以在这期间来查看待处理信号的个数,如下:

./sig

parent sleeping

child - BANG!

child - BANG!

child - BANG!

woke up! r=0

handled...

handled...

handled...

exiting

cat /proc/`pgrep sig`/status

略

SigQ:   4/4294967295

略

我们发送了三次信号,这里为什么是4呢,因为我们用了fork派生了子进程,子进程结束后会发送SIGCHLD信号.所以这里有4个信号待处理.

SigPnd: 0000000000000000

ShdPnd: 0000000000000000

SigBlk: 0000000000000000

SigIgn: 0000000001001206

SigCgt: 0000000180014c21

解释:

SigPnd:屏蔽位,存储了该线程的待处理信号,等同于线程的PENDING信号.

ShnPnd:屏蔽位,存储了该线程组的待处理信号.等同于进程组的PENDING信号.

SigBlk:存放被阻塞的信号,等同于BLOCKED信号.

SigIgn:存放被忽略的信号,等同于IGNORED信号.

SigCgt:存放捕获的信号,等同于CAUGHT信号.

CapInh: 0000000000000000

CapPrm: ffffffffffffffff

CapEff: ffffffffffffffff

CapBnd: ffffffffffffffff

解释:

CapEff:当一个进程要进行某个特权操作时,操作系统会检查cap\_effective的对应位是否有效,而不再是检查进程的有效UID是否为0.

CapPrm:表示进程能够使用的能力,在cap\_permitted中可以包含cap\_effective中没有的能力，这些能力是被进程自己临时放弃的,也可以说cap\_effective是cap\_permitted的一个子集.

CapInh:表示能够被当前进程执行的程序继承的能力.

CapBnd:是系统的边界能力,我们无法改变它.

Cpus\_allowed:   3

Cpus\_allowed\_list:      0-1

解释:

Cpus\_allowed:3指出该进程可以使用CPU的亲和性掩码,因为我们指定为两块CPU,所以这里就是3,如果该进程指定为4个CPU(如果有话),这里就是F(1111).

Cpus\_allowed\_list:0-1指出该进程可以使用CPU的列表,这里是0-1.

Mems\_allowed:   1

Mems\_allowed\_list:      0

内存同CPU一样,进程rsyslogd只是使用了结点0的内存资源.

我们这里调整该进程到CPU0,如下:

taskset -p 1 987

pid 987's current affinity mask: 3

pid 987's new affinity mask: 1

cat /proc/987/status

略

Cpus\_allowed:   1

Cpus\_allowed\_list:      0

Mems\_allowed:   1

Mems\_allowed\_list:      0

注:我们看到Cpus\_allowed/Cpus\_allowed\_list较之前有了变化.Cpus\_allowed由3变成了1.表明我们只会用CPU0.

voluntary\_ctxt\_switches:        1

nonvoluntary\_ctxt\_switches:     0

voluntary\_ctxt\_switches表示进程主动切换的次数.

nonvoluntary\_ctxt\_switches表示进程被动切换的次数.

首先查看一下当前进程,如下:

echo $$

1544

执行如下命令:

while ((1)); do echo 1; sleep 1; done

查看该进程的主动切换与被动切换,如下:

cat status

略

voluntary\_ctxt\_switches:        949

nonvoluntary\_ctxt\_switches:     55

我们看到主动切换和被动切换有了明显的变化.