

操作系统 实验一 进程管理

韩坤洋 16281038

1、打开一个vi进程。通过ps命令以及选择合适的参数，只显示名字为vi的进程。寻找vi进程的父进程，直到init进程为止。记录过程中所有进程的ID和父进程ID。将得到的进程树和由pstree命令的得到的进程树进行比较。

使用vi指令打开上一次实验的mem.c

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

int main(int argc, char *argv[]) {
    int *p = malloc(sizeof(int)); // a1
    assert(p != NULL);
    printf("(d) address pointed to by p: %p\n", getpid(), p); // a2
    *p = 0; // a3
    while (*p < 20) {
        sleep(1);
        *p = *p + 1;
        printf("(d) p: %2d addr_p: %p\n", getpid(), *p, p); // a4
    }
    return 0;
}

~
~
~
~

"mem.c" [Incomplete last line] 17 lines, 413 characters
```

打开另一个终端，使用 ps -c 或者借助 grep 对结果进行筛选名字为 vi 的进程

```
~ ps -ef | grep -w vi
knaveit. 22183 10013 0 08:14 pts/2 00:00:00 vi mem.c
knaveit. 22461 21366 0 08:18 pts/3 00:00:00 grep --color
-exclude-dir=.svn -w vi
```

借助 ps 的 -f 参数，使用 grep 过滤得到目标 pid 对应的列数据。

可以多次执行上述命令得到它所有父节点:vi 命令(22183), zsh控制台(10013), yakuake终端(1508), 然后是图形化界面的gnome-shell(1056), gnome-session-binary(1015), 接着出现gdm有关的进程, pid分别为 1002, 979,513, 最后出现了init(1)。

```

~ ➤ ps -ef | grep -w 10013
knavit 10013 1508 0 Mar12 pts/2 00:00:00 /bin/zsh
knavit 22183 10013 0 08:14 pts/2 00:00:00 vi mem.c
knavit 23366 21366 0 08:25 pts/3 00:00:00 grep --col
-exclude-dir=.svn -w 10013
~ ➤ ps -ef | grep -w 1508
knavit 1508 1056 0 Mar11 tty2 00:00:36 yakuake
knavit 1517 1508 0 Mar11 pts/0 00:00:02 /bin/zsh

```

```

~ ➤ ps -ef | grep -w 1056
knavit 1056 1015 0 Mar11 tty2 00:12:45 /usr/bin/gnome-shell
knavit 1508 1056 0 Mar11 tty2 00:00:36 yakuake
knavit 8904 1056 0 Mar11 tty2 -9 00:15:10 gnome-control-center
knavit 23400 21366 0 08:25 pts/3 00:00:00 grep --color=auto --exclude-dir=.b
-exclude-dir=.svn -w 1056
~ ➤ ps -ef | grep -w 1015
knavit 1015 1002 0 Mar11 tty2 00:00:00 /usr/lib/gnome-session-binary
knavit 1056 1015 0 Mar11 tty2 00:12:45 /usr/bin/gnome-shell

```

```

~ ➤ ps -ef | grep -w 1002
knavit 1002 979 0 Mar11 tty2 00:00:00 /usr/lib/gdm-x-session --run-script /usr/bin/gn
knavit 1004 1002 0 Mar11 tty2 00:06:20 /usr/lib/Xorg vt2 -displayfd 3 -auth /run/user/1
none -noreset -keeppty -verbose 3
knavit 1015 1002 0 Mar11 tty2 00:00:00 /usr/lib/gnome-session-binary
knavit 23423 21366 0 08:26 pts/3 00:00:00 grep --color=auto --exclude-dir=.b
-exclude-dir=.svn -w 1002
~ ➤ ps -ef | grep -w 979
root 979 513 0 Mar11 ? 00:00:00 gdm-session-worker [pam/gdm-password]
knavit 1002 979 0 Mar11 tty2 00:00:00 /usr/lib/gdm-x-session --run-script /usr/bin/gn
knavit 23436 21366 0 08:26 pts/3 00:00:00 grep --color=auto --exclude-dir=.b
-exclude-dir=.svn -w 979
~ ➤ ps -ef | grep -w 513
root 513 1 0 Mar11 ? 00:00:00 /usr/bin/gdm
root 618 513 0 Mar11 ? 00:00:00 gdm-session-worker [pam/gdm-launch-environment]
root 670 513 0 Mar11 ? 00:00:00 gdm-session-worker [pam/gdm-password]

```

接着使用 `pstree` 查看对应的进程树，发现与通过 `ps` 命令查找到的进程号一致。

```

~ ➤ pstree -p -s 22183
systemd(1)─gdm(513)─gdm-session-wor(979)─gdm-x-session(1002)─gnome-session-b(1015)─gnome-shell(1056)─yakuake(1508)─zsh(10013)─+
++
~ ➤ pstree -p -s 22183
systemd(1)─gdm(513)─gdm-session-wor(979)─gdm-x-session(1002)─gnome-session-b(1015)─gnome-shell(1056)─yakuake(1508)─zsh(10013)─vi(22183)

```

2、编写程序，首先使用 **fork** 系统调用，创建子进程。在父进程中继续执行空循环操作；在子进程中调用 **exec** 打开 **vi** 编辑器。然后在另外一个终端中，通过 **ps -Al** 命令、**ps aux** 或者 **top** 等命令，查看 **vi** 进程及其父进程的运行状态，理解每个参数所表达的意义。选择合适的命令参数，对所有进程按照 **cpu** 占用率排序。

对应程序代码：

```

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

int main (){
    pid_t fpid; //fpid表示fork函数返回的值
    fpid = fork();

    if (fpid < 0)

```

```

        printf("error in fork!");
    else if (fpid == 0) {
        int ret;
        ret = execl("/bin/vi", "vi",
"/home/knavit/Documents/_University/Junoir/OS/bjtu_OS_16281038/2_exp/2.c");
        if (ret == -1) {
            perror("execl");
        }
    }
    else {
        for (int i = 0; ; i < 1) {

        }
    }

    return 0;
}

```

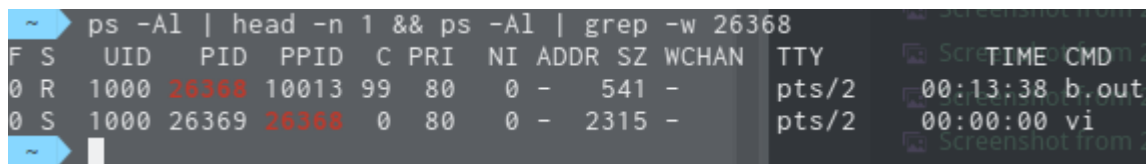
ps -Al 命令

执行 man 指令得到 ps 的详细信息:

-A Select all processes. Identical to -e

-l Long format. The -y option is often useful with this.

那么 -Al 则是选取所有进程以详细格式表示。利用 grep 筛选所需信息。得到了 b.out 及其子进程的相关信息。



```

~ ➤ ps -Al | head -n 1 && ps -Al | grep -w 26368
F S    UID    PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY      TIME CMD
0 R    1000  26368 10013  99  80   0 -  541 -          pts/2    00:13:38 b.out
0 S    1000  26369 26368   0  80   0 - 2315 -          pts/2    00:00:00 vi

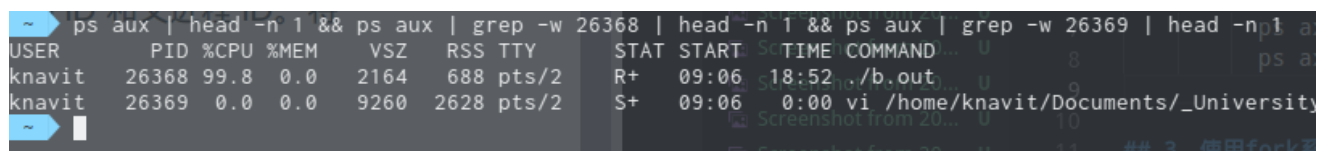
```

ps aux 命令

同上得到 aux 指令的信息:

To see every process on the system using BSD syntax:

查看所有使用 BSD 参数的进程。同样使用 grep 进行筛选得到 b.out 相关的进程信息。



```

~ ➤ ps aux | head -n 1 && ps aux | grep -w 26368 | head -n 1 && ps aux | grep -w 26369 | head -n 1
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   STOP COMMAND
knavit    26368 99.8   0.0    2164    688 pts/2    R+   09:06   18:52 ./b.out
knavit    26369  0.0   0.0    9260   2628 pts/2    S+   09:06   00:00 vi /home/knavit/Documents/_University

```

top 命令

使用 top 命令之后得到如下状态, 加入 -p 指令筛选对应线程之后只显示了一个空循环的父进程。猜测是由于目前 vi 没有占用cpu的操作所以没有被列出来。

```
~ # ps -c 26369 ash -m x
PID CLS PRI TTY STAT TIME COMMAND
26369 TS 19 pts/2 S+ 0:00 vi /home/knavit/Documents/_University/Junoir/OS/

~ # top -p26369 -p26368
top - 09:32:27 up 2 days, 22:27, 5 users, load average: 1.23, 1.72, 1.57
Tasks: 2 total, 1 running, 1 sleeping, 0 stopped, 0 zombie
%Cpu0 : 0.7 us, 0.3 sy, 0.0 ni, 92.7 id, 0.0 wa, 6.0 hi, 0.3 si, 0.0 st
%Cpu1 : 1.0 us, 0.0 sy, 0.0 ni, 99.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu2 : 100.0 us, 0.0 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu3 : 1.0 us, 0.0 sy, 0.0 ni, 98.7 id, 0.0 wa, 0.3 hi, 0.0 si, 0.0 st
%Cpu4 : 2.7 us, 0.3 sy, 0.0 ni, 97.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu5 : 2.0 us, 0.3 sy, 0.0 ni, 97.3 id, 0.0 wa, 0.3 hi, 0.0 si, 0.0 st
%Cpu6 : 2.3 us, 0.0 sy, 0.0 ni, 97.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu7 : 1.0 us, 0.3 sy, 0.0 ni, 98.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 16141328 total, 7315764 free, 6071924 used, 2753640 buff/cache
KiB Swap: 0 total, 0 free, 0 used. 9214688 avail Mem

  PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
26368 knavit 20 2164 688 620 R 100.0 0.0 26:11.77 /b.out
```

在 直接使用 top 命令之后，键入 P 会按照CPU的使用率进行降序排列。并且进程的数量不断浮动。

```
scroll coordinates: y = 1/325 (tasks), x = 1/12 (fields)
  PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
26368 knavit 20 2164 688 620 R 100.0 0.0 27:18.36 /b.out
1056 knavit 20 3530544 334912 146220 S 6.0 2.1 16:15.43 /usr/bin/gnome-shell
8904 knavit 20 727896 100840 70024 S 1.7 0.6 18:18.90 gnome-control-center
1004 knavit 20 556196 116304 67020 S 0.7 0.1 0:16.55 /usr/lib/xdg/vt2 -di
2443 knavit 20 1588068 386568 151120 S 1.0 2.4 15:56.04 /opt/google/chrome/ch
1 root 20 191608 10232 7688 S 0.7 0.1 0:29.07 /sbin/init
495 dbus 20 13364 6172 3576 S 0.3 0.0 5:08.20 /usr/bin/dbus-daemon
496 root 20 632448 23964 15820 S 0.3 0.1 3:51.62 /usr/bin/NetworkManag
514 polkitd 20 2276716 24900 15220 S 0.3 0.2 0:10.59 /usr/lib/polkit-1/pol
557 root -51 S 0.3 0:29.42 [irq/51-DLL082A:]
1508 knavit 20 428564 72544 54760 S 0.3 0.4 0:58.71 yakuake
13879 knavit 20 688488 165156 82844 S 0.3 1.0 1:07.83 /opt/google/chrome/ch
21310 knavit 20 2106576 288564 177212 S 0.3 1.8 2:07.58 /usr/lib/office6/wps
24950 knavit 20 1178376 168248 78964 S 0.3 1.0 0:21.84 /usr/lib/electron/ele
28760 knavit 20 16704 4132 3148 R 0.3 0.0 0:00.23 top
```

3、使用fork系统调用，创建如下进程树，并使每个进程输出自己的ID和父进程的ID。观察进程的执行顺序和运行状态的变化。

编写程序

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

int myFork(int, int);

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

    printf("I'm father, my pid : %d\r\n", getpid());
    myFork(1, 2);

    return 0;
}
```

```

int myFork(int counter, int step) {
    pid_t fpid = fork();

    if (fpid < 0)
        printf("error in fork!");
    else if (fpid == 0) {
        printf("Father's pid : %d my pid : %d\r\n", getppid(), getpid());
        if (counter > 0) {
            myFork(counter - 1, 1);
        }
        for (int i = 0; ; i < 1) {
            sleep(5);
            printf("Father's pid : %d my pid : %d\r\n", getppid(), getpid());
        }
    }
    else {
        if (counter >= 0) {
            myFork(counter - step, step + 1);
        }
        for (int i = 0; ; i < 1) {
            sleep(5);
            if (step == 3) {
                printf("I'm father, my pid : %d\r\n", getpid());
            }
            else {
                printf("Father's pid : %d my pid : %d\r\n", getppid(), getpid());
            }
        }
    }
    return 0;
}

```

编译并运行，利用 `ps tree` 命令得到进程树，可以看到符合题目要求的结构。

The terminal output shows the execution of the program. The parent process (PID 9555) prints "I'm father, my pid : 9555". It then forks two child processes (PIDs 9556 and 9557). Each child process forks two more child processes (PIDs 9558 and 9559). The output shows the sequence of prints from the parent and then the children.

```

~$ gcc -o c.out 3.c
~$ ./c.out
I'm father, my pid : 9555
Father's pid : 9555 my pid : 9556
Father's pid : 9555 my pid : 9557
Father's pid : 9556 my pid : 9558
Father's pid : 9556 my pid : 9559

```

The process tree (ps tree -p 9555) shows the following structure:

```

~$ ps tree -p 9555
c.out(9555)
├── c.out(9556)
│   ├── c.out(9558)
│   └── c.out(9559)
└── c.out(9557)

```

从第一张图片中输出的顺序中可以看出是先执行了父进程的代码，创建了 9556、9557，又执行了子进程的代码，创建了 9558、9559。

4、修改上述进程树中的进程，使得所有进程都循环输出自己的ID和父进程的ID。然后终止p2进程(分别采用kill -9、自己正常退出exit()、段错误退出)，观察p1、p3、p4、p5进程的运行状态和其他相关参数有何改变。

代码是在第三题基础上进行了更改，增加了从命令行读取参数的过程，然后编写了根据参数执行 exit、段错误、等待kill等操作的函数，并放在了合适的位置。

```
...
void suicide();
char type;

int main (int argc, char *argv[]){
    type = argv[1][0];

    ...

    return 0;
}

int myFork(int counter, int step) {
    ...

    if (...)
        ...
    else if (...) {
        ...
    }
    else {
        ...
        for (...) {
            ...
            else {
                // p2 位置
                suicide();
            }
        }
    }
    ...
}

void suicide() {
    int num[2] = {0, 0};
    int *p = 0x0;
    switch (type) {
        case '1':
            printf("self exit\r\n");
            exit(0);
            break;
        case '2':
```



```

    printf("illegal address accessing\r\n");
    num[1000] = *p;
    printf("didn't stop\r\n");
    break;
default:
    break;
}
}

```

kill 终止程序

首先运行程序，从输出中得到父程序 pid 为 2959，使用 `pstree` 查看完整的结构，然后执行 `kill -9`，发现被终止的进程还存在于树中，但是已经没有了子进程。

再次查看程序输出，发现 p2 的两个子进程直接将 pid 1 作为了父进程，使用 `ps -c` 查看 p2 详细信息，发现其 STAT 标识为僵尸进程。

```

I'm father, my pid : 2959
Father's pid : 2959 my pid : 2961
Father's pid : 2960 my pid : 2962
Father's pid : 2959 my pid : 2960
I'm father, my pid : 2959
Father's pid : 2960 my pid : 2963
Father's pid : 1 my pid : 2963
Father's pid : 2959 my pid : 2961
I'm father, my pid : 2959
Father's pid : 1 my pid : 2962
Father's pid : 1 my pid : 2963
Father's pid : 2959 my pid : 2961
Father's pid : 1 my pid : 2962

```

```

~> pstree -p 2959
d.out(2959)─d.out(2960)─d.out(2962)
              │         └─d.out(2963)
              └─d.out(2961)
~> kill -9 2960
~> pstree -p 2959
d.out(2959)─d.out(2960)
              └─d.out(2961)
~> ps -c 2960
  PID CLS PRI TTY      STAT      TIME COMMAND
 2960 TS   19 pts/0    Z+       0:00 [d.out] <defunct>

```

exit(0) 终止

同样运行程序，使用 `pstree` 得到原始进程树。等待 p2 自动执行 `exit`（输出“self exit”）之后再次查看进程树，发现 p4 p5 同上一操作一样，把 pid 1 作为了父进程。执行了 `exit` 的 p2 已经没有子进程。

这次利用 `ps -f` 查看进程情况，发现 p2 又是处于僵尸进程（defunct）状态。其余进程正常执行。

```

master ~> ./d.out 1
I'm father, my pid : 2105
Father's pid : 2105 my pid : 2106
Father's pid : 2105 my pid : 2107
Father's pid : 2106 my pid : 2108
Father's pid : 2106 my pid : 2109
Father's pid : 2105 my pid : 2107
I'm father, my pid : 2105
self exit
Father's pid : 2106 my pid : 2108
Father's pid : 2106 my pid : 2109
Father's pid : 2105 my pid : 2107
Father's pid : 1 my pid : 2109
I'm father, my pid : 2105
Father's pid : 1 my pid : 2108
Father's pid : 1 my pid : 2109

```

```

~> pstree -p 2105
d.out(2105)─d.out(2106)─d.out(2108)
              │         └─d.out(2109)
              └─d.out(2107)
~> pstree -p 2105
d.out(2105)─d.out(2106)
              └─d.out(2107)
~> ps -ef | grep 210
knavit 2105 1547 0 15:41 pts/0 00:00:00 ./d.out 1
knavit 2106 2105 0 15:41 pts/0 00:00:00 [d.out] <defunct>
knavit 2107 2105 0 15:41 pts/0 00:00:00 ./d.out 1
knavit 2108 1 0 15:41 pts/0 00:00:00 ./d.out 1
knavit 2109 1 0 15:41 pts/0 00:00:00 ./d.out 1
knavit 2129 1887 0 15:42 pts/1 00:00:00 grep --color=auto
e-dir=.hg --exclude-dir=.svn 210

```

段错误退出

这里为了触发段错误结束程序，直接使用了指针访问了 0 地址的内存并存入了数组访问越界的位置。如果没有发生 segmentation fault 则会在控制台上输出“didn't stop”。

```
int num[2] = {0, 0};
int *p = 0x0;
num[1000] = *p;
```

直接运行程序，同样是 `pstree` 得到进程树。发现同前两次一样，`p4 p5` 把 `pid 1` 作为了父进程，`p2` 处于僵尸进程（`defunct`）状态。

```
I'm father, my pid : 5566
Father's pid : 5566 my pid : 5567
Father's pid : 5566 my pid : 5568
Father's pid : 5567 my pid : 5569
Father's pid : 5567 my pid : 5570
I'm father, my pid : 5566
Father's pid : 5566 my pid : 5568
illegal address accessing
Father's pid : 5567 my pid : 5569
Father's pid : 5567 my pid : 5570
I'm father, my pid : 5566
Father's pid : 5566 my pid : 5568

~> pstree -p 5566
d.out(5566)─d.out(5567)─d.out(5569)
              │          │          │
              │          └─d.out(5570)
              └─d.out(5568)

~> pstree -p 5566
d.out(5566)─d.out(5567)
              │
              └─d.out(5568)

~> ps -c 5567
  PID CLS PRI TTY      STAT   TIME COMMAND
  5567 TS   19 pts/0    Z+      0:00 [d.out] <defunct>
```

遇到的问题

第三问进程的创建

第三问要额外建立 5 个线程，如果堆叠在 `main` 函数中会显得过于臃肿也不太适合阅读，所以就决定写一个函数在正确的位置执行 5 次来节省代码量。

但是发现逻辑上比较难处理，第一次创建的时候直接递归创建了“无数个”进程，真是刺激。然后修改了递归执行的逻辑，数量立马下来了，创建了 6 个子进程。最后又增加了一个变量控制递归逻辑才完成。

段地址错误

本来以为数组越界就可以触发段地址错误，一直访问到了 `a[1000000]`（读），都没有出现结束的迹象，最后访问了 `0x0` 才引发了段地址错误。