CEng 536 Advanced Unix
Fall 2018
Kernel Project 1
Due: 24/12/2018

1 Description

In this project you are going to implement two system calls for getting children and threads of a process. In Linux/Unix kernel there is not system call to get such statistics. Only way is to scan process list from proc filesystem and findout parent relations. Linux threads can be traversed through `/proc/task/*` but still indirect way to access this information.

In this project you are going to introduce to system calls:

```
int getchildids(pid_t proc, pid_t *buf, int max);
int getthreadids(pid_t proc, pid_t *buf, int max);
```

For process with given id, this calls will return the number of children and number of threads respectively. Also user supplies a buf buffer and the size of the buffer so that calls will fill in the first max element of the buffer with process ids of the children and task ids of the threads respectively (in Linux both children and threads have task_struct structures and pid of a thread task will give task id).

When number of children is less than or equal to max the number is returned and the buffer is filled with that many elements. If it is greater than max, max elements are put in the buffer but still the actual number is returned. If there is no element, 0 is returned.

All calls above are defined as macros mapped to system calls in get536.h:

```
#ifdef __x86_64__
#define NR_GETCHLDIDS 332
#define NR_GETTHRDIDS 333
#else
/* 32 bit */
#define NR_GETCHLDIDS 383
#define NR_GETTHRDIDS 384
#endif

#define getchildids( p, b, m) syscall(NR_GETCHLDIDS, p, b, m)
#define getthreadids( p, b, m) syscall(NR_GETTHRDIDS, p, b, m)
```

383 and 384 are to be replaced by the first available system call slot in your system.

2 Usage:

The following is a typical program:

```
#include<stdio.h>
#include<errno.h>
#include<unistd.h>
#include"get536.h"

void * test(void *p)
{
    printf("my thread id is %d\n", syscall(SYS_gettid));
    sleep(10);
}

int main() {
    int i, ret;
```

```
    pid_t pids[10];
    pthread_t threads[10];

    ret = getchildids(1, pids, 10);
    if (ret < 0)
        perror("getchildids");
    else {
        for (i = 0; i < ((r<10)?r:10); i++) {
            printf("%d ", pids[i]);
        }
        printf(" are child of 1\n");
    }

    if (fork() && fork() && fork())  { // parent of 3
        ret = getchildids(getpid(), pids, 10);
        if (ret < 0)
            perror("getchildids");
        else {
            for (i = 0; i < ((r<10)?r:10); i++) {
                printf("%d ", pids[i]);
            }
            printf(" are child of %d\n", getpid());
        }

        for (i = 0; i < 10; i++)
            pthread_create(threads+i, NULL, test, NULL);

        ret = getthreadids(getpid(), pids, 10);
        if (ret < 0)
            perror("getthreadids");
        else {
            for (i = 0; i < ((r<10)?r:10); i++) {
                printf("%d ", pids[i]);
            }
            printf(" are threads of %d\n", getpid());
        }
        for (i = 0; i < 10; i++)
            pthread_join(threads[i], NULL);

    } else {
        sleep(10);
    }
    return 0;
}
```

3 Implementation

You need the recompile the kernel in order to add system call. You need the following steps:

1. Get kernel source 4.9.0 (Any version ¿ 4.7 should work as long as following steps are valid)
2. Before compilation edit arch/x86/entry/syscalls/syscall_32.tbl. Add following lines:
   ```
   383     i386    getchildids     sys_getchildids
   384     i386    getthreadids    sys_getthreadids
   ```

   Edit arch/x86/entry/syscalls/syscall_64.tbl. Add following in common section:
   ```
   332     common  getchildids     sys_getchildids
   ```

```
333     common  getthreadids   sys_getthreadids
```

3. Under kernel directory (chosen as a standard place, actually it can be anywhere with a proper Makefile), create get536.c file with following empty declarations:

```
SYSCALL_DEFINE3(getchildids, pid_t, pid, pid_t __user *, buf, int, max) {
        printk("in getchildids(): %d %d\n", pid, max);
        return 0;
}
SYSCALL_DEFINE3(getthreadids, pid_t, pid, pid_t __user *, buf, int, max) {
        printk("in getthreadids(): %d %d\n", pid, max);
        return 0;
}
```

4. Add following line in kernel/Makefile
   obj-y += fd536.o

5. Copy your systems config as kernel configuration under source top directory:
   `cp /boot/config-`uname -r` .config`
6. For compatibility with the test setup I provide change these in the .config:

   VERSION = 4
   PATCHLEVEL = 9
   SUBLEVEL = 0
   EXTRAVERSION = -4-amd64

7. run `make -j 2 bzImage`
8. Now you can install this kernel and boot Linux system. However installation of modules to initial ram disk is not trivial. Scripts and more explanation will be given in newsgroup for booting a virtual machine with this new kernel.

Using this template, develop your project. Note that you need to submit only get536.c. You can use any kernel library and synchronization primitives you like. Note that you do not change any value in kernel and all accesses are readonly. Using rculist.h primitives for list traversals will be mostly sufficient for synchronization.
As a reminder child of a process is given in list_head children field of a task structure and from this list you need to make a sibling member traversal. For threads list_head thread_group contains the list of threads for the process. For getting task structure of a given pid, you can use the implementation from getpid() system call. The thread implementation in kernel code is more sophisficated than we need (tasks sharing mm_struct value considered thread, signal structures are traversed etc.). Make it simple, only traverse the list given above.
Use following header files when required:
`syscalls.h, semaphore.h, slab.h, kernel.h, printk.h, rculist.h, errno.h`
Usually you won't need but if you need system wide initializations at startup, use a prototype as:

```
int __init myinitialization (void);
core_initcall ( myinitialization )
```

That will register your myinitialization() function and automatically call when kernel is loaded. You only need this when you have global variables/environment to initialize.

4 Submission and External libraries

You need to submit a C source code. C++ is not allowed in kernel. Submit get536.c kernel code only. Submission details will be announced later. Please ask all questions to:
`news://news.ceng.metu.edu.tr:2050/metu.ceng.course.536/`
`https://cow.ceng.metu.edu.tr/News/thread.php?group=metu.ceng.course.536`

5 Kernel Compilation Guidelines

Qemu and virtiualbox image: 700MB, (1.7G uncompressed):
In campus: `http://sehitoglu.web.tr/ceng536-17.vmdk.gz`
Out of campus: `https://drive.google.com/file/d/1HZns88cW976BG4gAlkY2pEFUnVcO8e7P/view?usp=sharing`

- gunzip ceng536.vmdk.gz
- Then you can define it as a virtualbox image or use qemu as:
  `qemu-system-x86_64 -enable-kvm -smp 2 -hda ceng536-17.vmdk -m 1G`
  `-m` is for memory `-smp` for processor. If you don't compile your kernel in the vm, you can use smaller values like 512M.
- root password is 'ceng536'. If you execute dhclient you can use host network and install packages, and ssh to 10.0.2.2 for host access. (scp, sftp works same way)
- I recommend compiling kernel in the host. guest is compiled in 4.9.0-4-amd64 of debian:
  `https://drive.google.com/open?id=1wQRor59xrK1eXrd6Q4uTym45nDlN8BhA`
- extract it under a directory. copy following config and initrd.img files:
  `https://drive.google.com/open?id=1G1ykfLIIQR_UMXRzO2lDruCf8cuD7pYS`
  `https://drive.google.com/open?id=1ju5OINZo4Zsf0TR5Dt8p8IF_3UjiJctQ`
  `cp config-4.9.0-4-amd64 .config`
- Then edit first lines of Makefile as given above:

  ```
  VERSION = 4
  PATCHLEVEL = 9
  SUBLEVEL = 0
  EXTRAVERSION =-4-amd64
  ```

- Then compile kernel
  `make -j 4 bzImage`
- should compile the kernel and put it in arch/x86/boot. Than make necessary changes to compile your system call.
- ```
  qemu-system-x86_64 -enable-kvm -smp 2 -hda ceng536-17.vmdk \
      -kernel arch/x86/boot/bzImage -initrd initrd.img-4.9.0-4-amd64 \
    -append \
  'root=UUID="9f3f63c6-2e33-43ce-bf94-7ab3240b087c" ro init=/lib/sysvinit/init'\
      -m 512M
  ```

  should boot your new kernel.

If you do not have a kernel close to 4.9, you can repeat the same process within the guest machine. /usr/src contains the tar.xz file for linux source. It misses couple of pakcages like libc6-dev. Then you can compile and copy new image as /boot/vmlinuz-4.9.0-new-4-amd64 . Then call `update-grub2` and reboot.
  ```