Machine Problem 3: UNIX Processes (Due: 02/24/17)

Introduction:

The /proc directory is a pseudo filesystem that allows access to kernel data structures while in user space. It allows you to view some of the information the kernel keeps on running processes. To view information about a specific process you just need to view files inside of the directory: /proc/[pid]. For more information simply view the manpage with man proc.

The Assignment:

Using the files stored at /proc write a program/script to find information about a specific process using a user provided pid. In the following, you will find a list of the task_struct members for which you are required to find their value. In the task_struct a lot of the data you are finding is not represented as member values but instead pointers to other linux data structures that contain these members. All of the information you will be retrieving can be found in a processs proc directory (/proc/[pid]). Your program must be able to retrieve the following data about any given process if the given process Id is existing under direcotry /proc/:

Table #1: Process Attributes

Category	Required Variables/Items	Description
Identifiers	PID, PPID EUID, EGID RUID, RGID FSUID, FSGID	Process ID of the current process and its parent Effective user and group ID Real user and group ID File system user and group ID
State	R, S, D, T, Z, X	Running, Sleeping, Disk sleeping, Stopped, Zombie, and Dead
Thread Information	Thread_Info	Thread IDs of a process
Priority	Priority Number Niceness Value	Integer value from 1 to 99 for real time processes Integer value from -20 to 19
Time Information	stime & utime cstime & cutime	Time that a process has been scheduled in kernel/user mode Time that a process has waited on children being run in kernel/user mode
Address Space	Startcode & Endcode ESP & EIP	The start and end of a process in memory
Resources	File Handles & Context Switches	Number of fds used, and number of voluntary/involuntary context switches
Processors	Allowed processors and Last used one	Which cores the process is allowed to run on, and which one was last used
Memory Map	Address range, permissions offset, dev, inode, and path name	Output a file containing the process's currently mappped memory regions

Refer to Machine Problem 2 for exec function family and fork

Deliverables:

- **Code**:
  - You can do this project in three different languages: C++, Bash and Python. If you do it in C++, then you are to turn in the following files: proctest.h, proctest.cpp, main.cpp and Makefile. If you do it in Bash, submit one file: proctest.sh. If you do it in Python, submit one file: proctest.py. Make sure you also submit the Makefile if you do it in C++.
  - Be sure to start with the skeleton code provided with this assignment. You can implement those functions in the files in your way. However, do not change the declarations of the functions or the grading script that will test your assignment will not be able to give you the points you deserve.
  - Website for learning bash: [https://www.ibm.com/developerworks/library/l-bash/]
- **Report**:
  - Answer the following additional questions
    1. For a process run by a user other than yourself, find the following items from Table #1: [Identifiers, State, Thread Information, Priority, Time Information, Resources, and Memory Map]
    2. For a process that you have created, retrieve all items enumerated in Table.
    3. What are the differences between the real user IDs and effective user IDs, and what is a situation where these will be different?
    4. Why are most of the files in /proc read only?
    5. Why is the task_struct so important to the kernel and what is it used for?

Advanced Concepts:

Task_Struct Members

Linux and UNIX distributions organize all internal process data into structures named task_struct. The kernel maintains one task_struct member for each process running on the system. Each task_struct member contains two pointers specifically designated to point to other task_struct members. As such, the kernel organizes all task_struct members into linked lists. On startup, the kernel also initializes a pid hash table whose elements are linked lists. This is done to save some time searching for process data structures. Instead of having to search one large list (for perspective, pid_max on the author's Debian 8 system is 2^{15}), the kernel quickly hashes the process pid into one of the hash table elements and then searches a much smaller list to find the correct task_struct member (Bovet, 81 & Glass, 555).

A visual description of the organization of the task_struct structure is provided in Figure 1. Also provided, for the curious reader, is the entire listing of all task_struct members. Inside, you should be able to see all of the fields that you are required to find for the assignment (or at least pointers to other kernel data structures that actually contain those fields). For even further reading see *Bovet* and *Glass* in the bibliography section

Figure #1: Task Struct Members & Fields (Glass, 554)

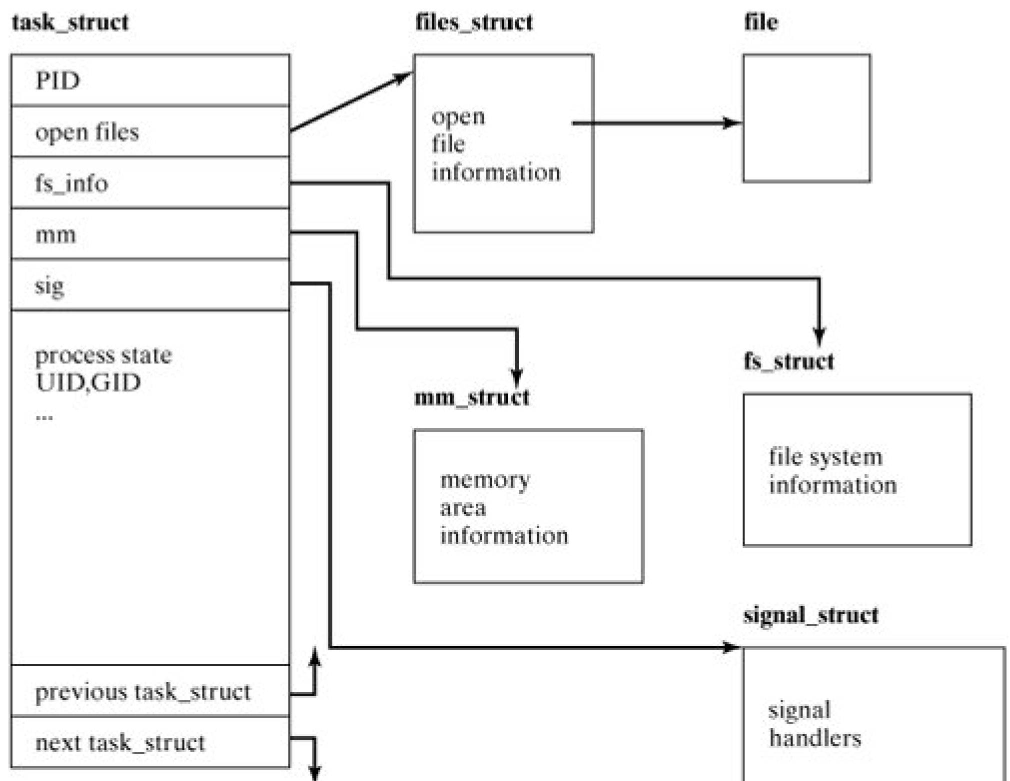
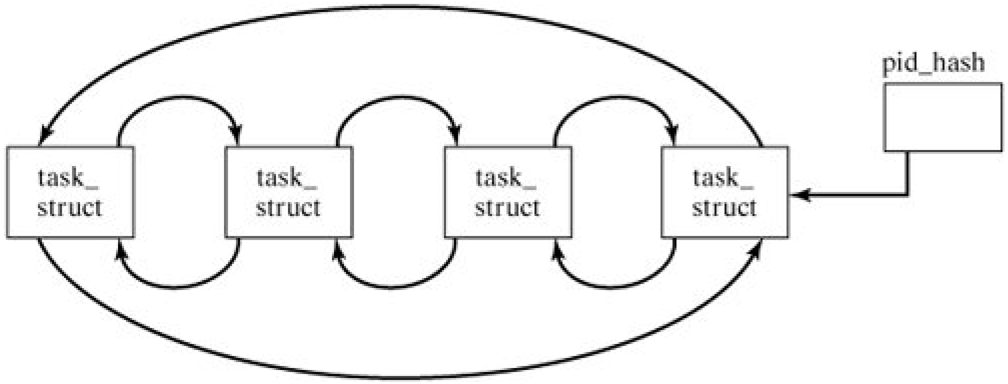


Figure #2: Pid Hash Table (Glass, 555)



Example #1: task_struct Declaration

```
struct task_struct {
/* these are hardcoded - don't touch */
  volatile long         state;          /* -1 unrunnable, 0 runnable, >0 stopped */
  long                  counter;
  long                  priority;
  unsigned              long signal;
  unsigned              long blocked;   /* bitmap of masked signals */
  unsigned              long flags;     /* per process flags, defined below */
  int errno;
  long                  debugreg[8];    /* Hardware debugging registers */
  struct exec_domain    *exec_domain;
/* various fields */
  struct linux_binfmt   *binfmt;
  struct task_struct    *next_task, *prev_task;
  struct task_struct    *next_run, *prev_run;
  unsigned long         saved_kernel_stack;
  unsigned long         kernel_stack_page;
  int                   exit_code, exit_signal;
  /*Other features of a process. personality is for setting process execution domain;
      dumpable:1 means process can be dumpable and so on.*/
  unsigned long         personality;
  int                   dumpable:1;
  int                   did_exec:1;
  int                   pid;
  int                   pgrp;
  int                   tty_old_pgrp;
  int                   session;
  /* boolean value for session group leader */
  int                   leader;
  int                   groups[NGROUPS];
  /*
```

```
 * pointers to (original) parent process, youngest child, younger sibling,
 * older sibling, respectively. (p->father can be replaced with
 * p->p_pptr->pid)
 */
struct task_struct   *p_opptr, *p_pptr, *p_cptr,
                     *p_ysptr, *p_osptr;
struct wait_queue    *wait_chldexit;
unsigned short       uid,euid,suid,fsuid;
unsigned short       gid,egid,sgid,fsgid;
unsigned long        timeout, policy, rt_priority;
unsigned long        it_real_value, it_prof_value, it_virt_value;
unsigned long        it_real_incr, it_prof_incr, it_virt_incr;
struct timer_list    real_timer;
long                 utime, stime, cutime, cstime, start_time;
/* mm fault and swap info: this can arguably be seen as either
   mm-specific or thread-specific */
unsigned long        min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
int swappable:1;
unsigned long        swap_address;
unsigned long        old_maj_flt;   /* old value of maj_flt */
unsigned long        dec_flt;       /* page fault count of the last time */
unsigned long        swap_cnt;      /* number of pages to swap on next pass */
/* limits */
struct rlimit        rlim[RLIM_NLIMITS];
unsigned short       used_math;
char                 comm[16];
/* file system info */
int                  link_count;
struct tty_struct    *tty;          /* NULL if no tty */
/* ipc stuff */
struct sem_undo      *semundo;
struct sem_queue     *semsleeping;
/* ldt for this task - used by Wine. If NULL, default_ldt is used */
struct desc_struct *ldt;
/* tss for this task */
struct thread_struct tss;
/* filesystem information */
struct fs_struct     *fs;
/* open file information */
struct files_struct  *files;
/* memory management info */
struct mm_struct     *mm;
/* signal handlers */
struct signal_struct *sig;
#ifdef __SMP__
```

```
int            processor;
int            last_processor;
int            lock_depth;     /* Lock depth.
                                  We can context switch in and out
                                  of holding a syscall kernel lock ... */
#endif
};
```

Check out the following website which talks linux in much details:
[http://www.makelinux.net/books/lkd2/?u=ch03lev1sec1]

UNIX man Pages:

One incredibly useful feature of UNIX operating systems that many new developers do not know about is the built in manual system. Using the command 'man', you can access information about most aspects of the operating system from general commands all the way to system call APIs.

The structure of man pages in UNIX are organized into sections by number follows (Wikimedia Foundation):

1. General Commands
2. System Calls
3. Library Functions (Specifically, the C standard library)
4. Special Files
5. File Formats
6. Games
7. Miscellanea
8. System Administration

You may (or may not, that's fine too) find the following manual pages useful when creating this assignment. Each one of these lines can be executed as a valid shell command to open a particular manual page. Note, the number indicates the manual section that that function resides.

- `man 3 exec`
- `man 2 fork`
- `man 2 chdir`
- `man 2 pipe`
- `man 2 dup`
- `man 2 wait`

Grading Rubric:

Points will be assigned according to the following rubric. The code portion of the assignment is worth 160 points, divided as shown below. The report is worth an additional 40 points. Consequently, the point total for the assignment is 200 points.

- getpid(): 5 points
- getppid(): 5 points
- geteuid(): 5 points
- getegid(): 5 points
- getruid(): 5 points
- getrgid(): 5 points
- getfsuid(): 5 points
- getfsgid(): 5 points
- getstate(): 5 points
- getthread_count(): 5 points
- getpriority(): 5 points
- getniceness(): 5 points
- getstime(): 5 points
- getutime(): 5 points
- getcstime(): 5 points
- getcutime(): 5 points
- getstartcode(): 5 points
- getendcode(): 5 points
- getesp(): 5 points
- geteip(): 5 points
- getfiles(): 20 points
- getvoluntary(): 5 points
- getnonvol(): 5 points
- getallowed_cpus(): 5 points
- getlast_cpu(): 5 points
- getmemory_map(): 20 points

Bibliography:

[1] Bovet, Daniel Pierce. *Understanding the Linux Kernel: From I/O Ports to Process Management*. U.S.A: O'Reilly, 2003. Print.

[2] Glass, Graham. *Linux for Programmers and Users*. Upper Saddle River, NJ: Pearson Prentice Hall, 2006. Print.