Micro Assignment - 01

Process and Environment Information

Process is a representation of an executing program. Program is a file that contains the code and data (initialized and uninitialized) that describes how the process should be created.

In UNIX/Linux, the executable file formats could be one of the following.

- a.out (Assembler output)
- COFF (Common Object File Format)
- ELF (Executable Linking Format)

The kernel parses the program file according to the file format. The following information is included in a typical program file.

- *Machine-language instructions*: These encode the algorithm of the program.
- *Program entry-point address*: This identifies the location of the instruction at which execution of the program should commence.
- *Data*: The program file contains values used to initialize variables and also literal constants used by the program (e.g., strings).
- *Symbol and relocation tables*: These describe the locations and names of functions and variables within the program. These tables are used for a variety of purposes, including debugging and run-time symbol resolution (dynamic linking).
- *Shared-library and dynamic-linking information*: The program file includes fields listing the shared libraries that the program needs to use at run time and the pathname of the dynamic linker that should be used to load these libraries.
- *Other information*: The program file contains various other information that describes how to construct a process.

A program can create one or more processes.

From a kernel's point-of-view, a process can be considered as a range of user-space memory containing code and data, kernel data structures maintaining the state of the process,

Each process is represented in the systems by a process ID (a positive number). Process ID is useful for sending messages to the process or associating other data such as files with the process.

The *getpid()* system call returns the process ID of the calling process.

```
#include <unistd.h>

pid_t getpid(void);
```

Always successfully returns process ID of caller

Linux kernel limits process IDs to 32767. Once it reaches this limit, it resets the counter and uses free IDs from 300 upwards. This is done because a running OS has pretty much used all process IDs below 300.

Each process has a parent - the process that created it! The parent can be found out by:

```
#include <unistd.h>

pid_t getppid(void);
```

Always successfully returns process ID of parent of caller

Each process has its own parent going back to the init process (process ID = 1). The hierarchy of processes constitute the family tree for a process.

If a process is orphaned (immediate parent dies), it is adopted by the init process.

More Information About a Process

In older UNIX systems, there was no easy way of getting information from the kernel to answer questions like the following:

- How many processes are running on the system and who owns them?
- What files does a process have open?
- What files are currently locked, and which processes hold the locks?
- What sockets are being used on the system?

To solve this problem, modern UNIX systems provide /proc interface. The /proc interface is a virtual file system (i.e., it is a file system like interface for reading and sometimes writing kernel maintained parameters.

More information about a process can be obtained by examining the directory **/proc/pid**

In particular, the file *status* has lot of information about a process. For the init (the first system program launched by a UNIX/Linux), the following is an example status file.

```
$ cat /proc/1/status
Name:   init                        Name of command run by this process
State:  S (sleeping)                State of this process
Tgid:   1                           Thread group ID (traditional PID, getpid())
Pid:    1                           Actually, thread ID (gettid())
PPid:   0                           Parent process ID
TracerPid:      0                   PID of tracing process (0 if not traced)
Uid:    0       0       0       0   Real, effective, saved set, and FS UIDs
Gid:    0       0       0       0   Real, effective, saved set, and FS GIDs
FDSize: 256                         # of file descriptor slots currently allocated
Groups:                             Supplementary group IDs
VmPeak:      852 kB                 Peak virtual memory size
VmSize:      724 kB                 Current virtual memory size
VmLck:         0 kB                 Locked memory
VmHWM:       288 kB                 Peak resident set size
VmRSS:       288 kB                 Current resident set size
VmData:      148 kB                 Data segment size
VmStk:        88 kB                 Stack size
VmExe:       484 kB                 Text (executable code) size
VmLib:         0 kB                 Shared library code size
VmPTE:        12 kB                 Size of page table (since 2.6.10)
Threads:       1                    # of threads in this thread's thread group
SigQ:   0/3067                      Current/max. queued signals (since 2.6.12)
SigPnd: 0000000000000000            Signals pending for thread
ShdPnd: 0000000000000000            Signals pending for process (since 2.6)
SigBlk: 0000000000000000            Blocked signals
SigIgn: fffffffe5770d8fc            Ignored signals
SigCgt: 00000000280b2603            Caught signals
CapInh: 0000000000000000            Inheritable capabilities
CapPrm: 00000000ffffffff            Permitted capabilities
```

Figure 1: Example /proc/pid/status file.

Files such as the above are "dynamic". It is not appropriate to look for a particular line number. Instead these files should be parsed for a particular matching pattern and the required data should be read off.

The following table shows other important files found in the /proc/pid directory.

File	Description (process attribute)
`cmdline`	Command-line arguments delimited by `\0`
`cwd`	Symbolic link to current working directory
`environ`	Environment list *NAME=value* pairs, delimited by `\0`
`exe`	Symbolic link to file being executed
`fd`	Directory containing symbolic links to files opened by this process
`maps`	Memory mappings
`mem`	Process virtual memory (must *lseek()* to valid offset before I/O)
`mounts`	Mount points for this process
`root`	Symbolic link to root directory
`status`	Various information (e.g., process IDs, credentials, memory usage, signals)
`task`	Contains one subdirectory for each thread in process (Linux 2.6)

Figure 2: Some files found in /proc/pid directories.

Environment of a Process

Each process has an NAME=VALUE array that is called environment. This is a mechanism for passing arbitrary information to a process. The names in the list are known as the environment variables.

At creation, a process inherits its parent's environment variables. It can make local changes and pass it to its children.

Shells use this mechanism to pass configuration parameters to the programs running within them.

One way of accessing the environment variable is to use the **environ global variable created by the C runtime.

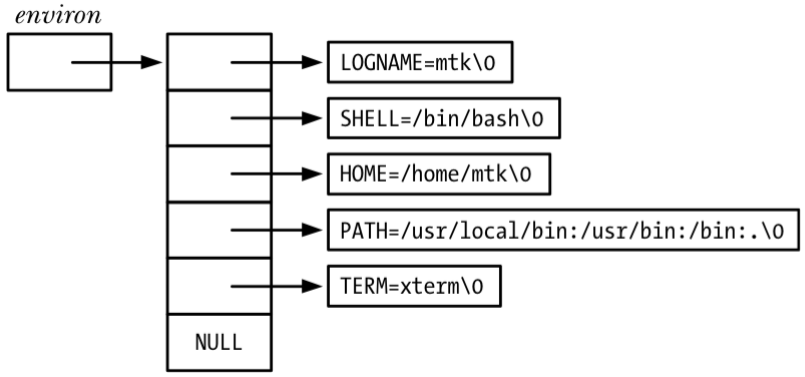

Figure 3: Example composition of the **environ variable.

Another way to access the environment variables is to read the contents of the environ file in the /proc/pid directory. Each NAME=VALUE pair will appear as NULL terminated strings in that file.

UNIX/Linux systems also provide library routines for getting and setting environment variables.

```
#include <stdlib.h>

char *getenv(const char *name);
```

Returns pointer to (value) string, or NULL if no such variable

```
#include <stdlib.h>

int setenv(const char *name, const char *value, int overwrite);
```

Returns 0 on success, or −1 on error

```
#include <stdlib.h>

int unsetenv(const char *name);
```

Returns 0 on success, or −1 on error

What Do You Need To Do?

Read the FULL handout and reflect on the concepts. If you have questions on the concepts post them on the discussion board and find answers for them.

Write a program that does the following:

Prints the process ID and parent process ID.

Reads the /proc/pid/status file for the process. Use fopen() and fscanf() to read the content of the /proc/pid/status file. Parse the file to find out the values for the following parameters

1. Name:
2. State:
3. Stack usage:
4. voluntary_ctxt_switches:

5. nonvoluntary_ctxt_switches:

Prints all the environment variables and their values.