

CSIS0230A Principles of Operating Systems (Class A)

Tutorial 5

Process Structure

Add two system calls to the kernel, with the following specifications.

int *getchild*(*pid_t* *pid*, *pid_t** *child*, **int** *size*)

Given a specific process *pid*, fill the process IDs of all its children in the *child* array of size *size*. At most that many entries in *child* will be used by the system call. It returns the number of children of *pid*. Thus a caller can check whether the buffer is large enough by inspecting the return value.

Input:

pid—the specified process ID

child—an array given by the user process for storing the children process IDs

size—the size of *child* array

Return value: the number of children of *pid*

Possible errors:

ESRCH—process cannot be found

EFAULT—memory specified in *child* is not writable

int *getname*(*pid_t* *pid*, **char*** *comm*)

Given a process *pid*, get the command name of the program executed by the process (i.e., the *comm* member in the PCB) to *comm*.

Input:

pid—the specified process ID

comm—a buffer given by the user process for storing *comm* of the process

Return value: 0 if successful.

Possible errors:

ESRCH—process cannot be found

EFAULT—memory specified in name is not writable

After you finish writing these system calls, write a user program to test whether the system calls work as expected. A *pid* should be read from the command line, and the program should print all the *pids* and names of all its children.

For example, one can execute the program as

```
./a.out 541
```

Output:

```
The children of pid 541 are:
```

```
9305: in.telnetd
```

```
9146: in.telnetd
```

One possible structure of the user program:

```
#include <syscall.h>
#include <errno.h>
#include <sys/types.h>
#include <stdlib.h>
/* Define __NR_getchild and __NR_getname here */
_syscall3(int, getchild, pid_t, pid, pid_t*, child, int, size);
_syscall2(int, getname, pid_t, pid, char*, name);
int main(int argc, char** argv) {
    pid_t pid;
    if (argc != 2) {
        printf("Usage: %s <pid>\n", argv[0]);
        exit(1);
    }
    pid = atol(argv[1]);
    /* find number of children of pid (calling getchild with size=0) */
    /* allocate memory for children array */
    /* find all pid's children process IDs */

    /* for each child process ID
       find the name of process
       print the process ID and name */
}
```

To ease debugging, remember to check for errors returned by the system calls, and print out the error message in such cases.

Hints:

1. Don't forget to modify `entry.S`, `sys.h` and `Makefile`.
2. Using the user address space without checking will leave a really big security hole in your kernel. So remember to check them by using `copy_to_user` or `put_user`.
3. Use `printk()` in kernel code for debugging. For example, you can use `printk("<0>pid=%d\n", pid)` in order to see the value of the variable `pid`. The "`<0>`" says that it is an "emergent" message, and should be communicated to the user through all possible means (show in text-mode console, show in an X-console window, in log files, etc).