

Reminders

- n Project 1 due tomorrow at 5:00 pm
- n Homework 2 out, due next Wednesday
- n Start thinking about project groups (3 people) for the rest of the quarter
- n Today: project 1 questions

1

Project 1 – issues

- n C strings
- n Copy_to/from_user and counters
- n Syscalls: macros ; arguments
- n Execvp
- n Other things

2

C strings

- n You only need to use:
 - n **strcmp**(src,dest,256) – compare strings, 0 if equal, not 0 o.w.
 - n **strtok**:
 - n 1st use: tok = strtok(buf, "delimiters");
 - n Subsequent uses: tok = strtok(NULL, "delimiters");
 - n **fgets**(buf, 256, stdin) – read a line (up to 256 chars) from stdin (or getline)
 - n (*maybe*) **strncpy**(dest, src, 256) – copy up to 256 chars from src to dest
 - n (*maybe*) Allocate memory with **malloc**, free with **free**
- n Fine to assume:
 - n A maximum length for a shell command (say, 256)
 - n Maximum number of arguments (say, 256 again)

3

Passing counters

- n Do not printk the statistics in exccounts!!!
- n Exccounts should pass count values to the shell
 - n The shell then prints out statistics
- n Copying counters to userspace:
 - n Shell passes in something to hold data
 - n This could be a pointer to a struct, an array, or four pointers to ints.
 - n exccounts fills the data in

4

Copying data to/from kernel

- n Unsafe to directly access user pointers!


```

long sys_gettimeofday(struct timeval *tv)
{
    if (tv) {
        struct timeval ktv;
        do_gettimeofday(&ktv);
        if (copy_to_user(tv, &ktv, sizeof(ktv)))
            return -EFAULT;
    }
    return 0;
}
      
```
- n **copy_to/from_user** return amount of uncopied data

5

Syscalls

- n Two ways to use one:
 - n Linux style:
 - n in <asm/unistd.h>:


```

#define __NR_foo 292
static inline _syscall2(int, foo, int, arg1,
                        char *, arg2)
              
```
 - n In userspace, just call `foo(4, "test");`
 - n BSD style:
 - n in shell.c:


```

#define __NR_foo 292
ret = syscall(__NR_foo, arg1, arg2);
              
```

6

How syscalls work

In `entry.S`:

```
ENTRY(system_call)
    pushl %eax                # save orig_eax
    SAVE_ALL
    cmpl $(NR_syscalls),%eax
    jae badsys
    call *SYMBOL_NAME(sys_call_table)(,%eax,4)
    movl %eax,EAX(%esp)      # save the return value
restore_all:
    RESTORE_ALL
```

7

Execvp

- n You must build an array of strings to pass to it
- n Make sure the last thing in this array is NULL!
- n Make sure the array includes the program name!

8

Extern

- n How do we access global variables defined in one file from another file?

9

Code quality

- n What's wrong with this:


```
char * buffer;
buffer = malloc(100);
strcpy(buffer, param);
```
- n How do we fix this?

10

Debugging

```
#define MYDEBUG

#ifdef MYDEBUG
    #define DEBUG(x) x
#else
    #define DEBUG(x)
#endif

... DEBUG(sprintf("debug output"));
```

11

Debugging 2

- n Just for printing:


```
#ifndef MYDEBUG
#   ifdef __KERNEL__
/* This one if debugging is on, and kernel space */
#       define PDEBUG(fmt, args...) printk("myprg: " fmt, ## args)
#   else
/* This one for user space */
#       define PDEBUG(fmt, args...) fprintf(stderr, fmt, ## args)
#   endif
#else
#       define PDEBUG(fmt, args...) /* not debugging: nothing */
#endif
```
- n works for both for kernel and userspace
- n To use:


```
PDEBUG("Testing two numbers: %d and %d\n", num, num2);
```

12

Things to check

- n Check that every malloc has a matching free
- n Check for errors
 - n E.g. `execvp` returns -1, `malloc` returns NULL
 - n Frequently, global constant `errno` will be set
 - n Use `perror("error description");` to see what the error was.

13

Fork

- n How does it work?
- n Any problems with it?

14

UNIX startup

- n Kernel only creates one process: *init* (pid 1)
 - n Never creates any other processes
- n *init* spawns other processes using **fork**, **exec**, etc.
- n E.g. *init* creates a bunch of *getty* processes
 - n One for each terminal
 - n Each outputs "login: " prompt, gets input, **execs** *login* process
 - n *Login* prompts for a password, **execs** shell if it's ok
- n Off-topic: why is `cd` a built-in command in your shell?

15

A UNIX Process

The diagram illustrates the memory layout of a UNIX process. On the left, a vertical stack of memory regions is shown: Static Data, Stack (with 'sp' pointing to its top), Heap, and Text (with 'pc' pointing to its start). On the right, the Process Control Block (PCB) is shown as a table with the following fields: PID (short int), User Id, Parent PID, State (Registers), Status (run, wait, ...), Program Filename, Current Working Dir, Open Files, and three vertical ellipses indicating other fields.

16

Same Process after fork

This diagram shows the state of a process after a `fork` system call. It features two identical memory layout diagrams side-by-side, representing the parent and child processes. Each diagram shows the same memory regions: Static Data, Stack (with 'sp' pointing to its top), Heap, and Text (with 'pc' pointing to its start). To the right of the memory diagrams are two identical PCB structures, each containing the same fields as the PCB in slide 16: PID (short int), User Id, Parent PID, State (Registers), Status (run, wait, ...), Program Filename, Current Working Dir, Open Files, and three vertical ellipses.

17