CSE 451: Operating Systems

Section 3

System calls, Memory allocation, Makefiles

Library calls versus system calls

- Which of the following map to system calls and which execute purely in userspace?
  - `strlen()`, `execvp()`, `fork()`, `printf()`, `clone()`, `open()`, `atoi()`, `exit()`
- `unistd.h` (generally found under `/usr/include`) contains the declarations of many system calls
  - Other library functions rely directly or indirectly on system calls defined in this header

Adding a system call

- The good part: how do we actually add a system call to the kernel in the version (3.8.3) that we are using?
  - Let's look at a semi-recent patch to the kernel as an example
- Files to modify/add:
  - `arch/x86/syscalls/syscall_64.tbl`
  - `include/linux/syscalls.h`
  - `kernel/sys_ni.c`
  - `kernel/Makefile`
  - `write: kernel/[your_file].c`

Adding a system call

- Add a simple system call that uses `printk()` to print a value and returns the value as its exit code
- `printk()`s are written to `/var/log/messages` and can be printed to the console with the `dmesg` command
- Useful for debugging!

Invoking a system call

- Use the syscall() function from userspace to invoke system calls “directly”

```
#include <stdio.h>   // for printf()
#include <stdlib.h>  // for atoi()
#include <unistd.h>  // for syscall()

int main(int argc, char* argv[]) {
  if (argc != 2) {
    fprintf(stderr, "Usage: %s value\n", argv[0]);
    return 1;
  }

  int syscall_number = 314;  // number of the newly-added syscall
  int value = atoi(argv[1]);
  int ret = syscall(syscall_number, value);
  printf("Return value is %d\n", ret);

  return 0;
}
```

Userspace memory allocation

- In userspace C programs, `malloc()` and `calloc()` allocate memory on the heap and `free()` frees it
  - libc maintains a free list in the data segment to facilitate memory allocation
  - When a userspace process attempts to allocate memory and libc has none to give it, libc increases the size of the data segment via `sbrk()` (see man 2 sbrk)

Kernel memory allocation

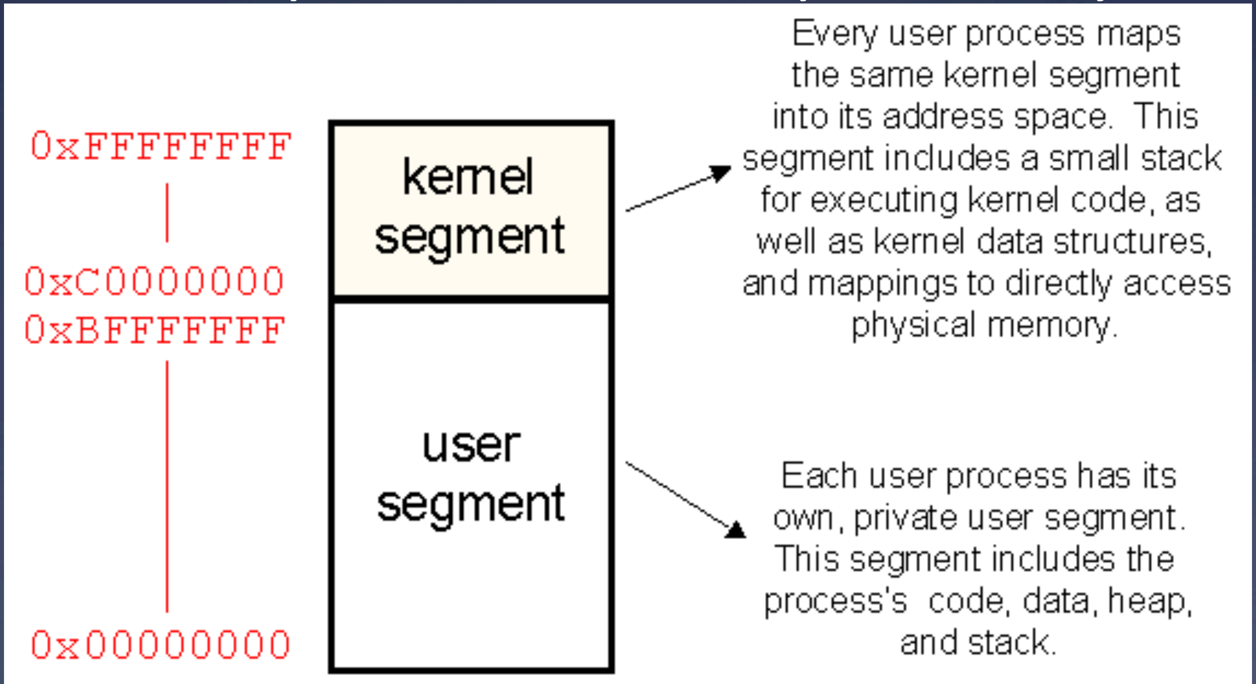
- In the kernel, there are some different use cases and considerations:
  - Some modules allocate and free memory frequently, whereas others hold memory for long periods of time
  - If the kernel blocks or sleeps when allocating memory, the performance of other processes will be impacted
- What happens if the kernel attempts to read uninitialized memory? Unallocated memory?

Kernel memory allocation

- `kmalloc()`: Standard method of allocating memory within the kernel
  - Flags parameter allows caller to specify who will be using the memory (userspace or kernel) and whether the call should be allowed to sleep
- `vmalloc()`: Allocates large blocks of virtually contiguous memory
  - Not many use cases require it and furthermore Linus (a.k.a. the kernel god) disapproves
  - Slower than `kmalloc()`

Address space mapping

- Parts of the kernel are mapped into the address space of userspace processes for faster access
- There are special functions for copying memory between userspace and kernel space—why is this?



Kernel memory safety

* `copy_from_user()`
  * Copy memory from userspace to kernel space
  * Why is there a special function for this?
* `copy_to_user()`
  * Copy memory from kernel space to userspace
* `access_ok()`
  * Check if access to a particular userspace memory address of a given size is okay
  * How would you implement this?

Beyond fsh.c

* What is bash doing when you run a process in the background? How does that differ from fsh?
* How does bash kill its children when it quits?
* How does it “disown” its children so they aren’t killed when it quits? (see `nohup` and `disown`)

Uses of `fork`

- When can you imagine using `fork` that's not as a shell?
- Long ago the internet super-service daemon (inetd) sat there waiting for connections on all ports, and started up the appropriate server on demand (this saved on precious memory)
- Android runs a Linux kernel. It keeps a "warm" Dalvik VM image that `fork`s to start your app, avoiding the startup cost of a full Java VM

Signals and `ps`

- You can send arbitrary signals to your processes with `kill`, not just SIGKILL.
- Add signal handlers with signal() to respond to them.
- ps tricks:
  - `ps -faux` – show all processes as a tree, see who spawned whom
  - `ps -melf` – show all the threads that belong to a process
  - Hopefully this order of options is easy to remember… faux and melf.

Makefiles

- Makefiles can simplify the development process for the userspace parts of project 1—be sure to use them effectively!
- Some advanced functionality: `patsubst` and suffix-based rules

Makefiles

- `patsubst(a, b, c)`: replace occurrences of `a` in `c` with `b`
- Special macros:
  - `$@`: Name of Makefile target
  - `$<`: Name of left-most dependency of Makefile target
  - `$^`: Names of all Makefile target dependencies
- `.d` files: GCC is capable of scanning source files and identifying their dependencies. This means automatic recompilation when dependent files change without even naming them in rules :)

Sample Makefile

```
NODEPS=clean
CC=gcc
CFLAGS=-std=gnu99 -g -Wall -O0
SRCS=$(shell find . -maxdepth 1 -name "*.c")
DEPFILES=$(patsubst %.c, %.d, $(SRCS))
OBJS=$(patsubst %.c, %.o, $(SRCS))

example: $(OBJS)
	$(CC) $(CFLAGS) -o $@ $(OBJS)

%.o: %.c %.d
	$(CC) $(CFLAGS) -o $@ -c $<

%.d: %.c
	$(CC) -MM -MT '$(patsubst %.c, %.o, $<)' $< -MF $@

clean:
	rm -f $(OBJS) $(PROGRAMS) $(DEPFILES)

# Don't generate dependencies for all rules
ifeq (0, $(words $(findstring $(MAKECMDGOALS), $(NODEPS))))
	-include $(DEPFILES)
endif
```

Sample Makefile

- Any .c files in the current directory will be built automatically and linked into the `example` executable
- If one of the .c files depends on a .h file that changes, the rules in its .d file will cause it to be rebuilt when `make` is next invoked
- Project 1 has fairly simple requirements, but becoming more familiar with Makefiles will prove a boon to you in the future

More project 1 advice

- Be wary of race conditions in the kernel code that you write
  - What happens if two processes update the count stored in a task struct at the same time?
  - Use atomics in `include/asm-generic/atomic.h` or `cmpxchg` in `include/asm-generic/cmpxchg.h`
    - If you use `cmpxchg`, you'll need to call it from a loop (why?)
- Don't forget to check that access to a userspace buffer is okay before attempting to read from it or write to it
  - As a test, try passing a variety of valid and invalid userspace and kernel addresses to your system call

More project 1 advice

* Implement the "." command for the shell early on so you can have some automated test cases
* Make sure to test a variety of bad inputs to the shell and verify that none of them cause it to crash or behave unexpectedly

More project 1 advice

* Use the `strace` command to see if your system call counts are reasonable
* For example, we can check how many times the `echo` command calls `open()`:

```
$ strace echo "hi" 2>&1 | grep open
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
open("/lib64/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
open("/usr/lib/locale/locale-archive", O_RDONLY|
O_CLOEXEC) = 3
```