CSE 451: Operating Systems

Section 9

Debugging kernel modules, project 3

Preliminary project 2b feedback

- Many groups disabled interrupts unnecessarily or too early/for too long
- Do `sthread_user_mutex_free` and `sthread_user_cond_free` need to disable interrupts/acquire a lock?
  - No: They are only invoked after all function calls using them have finished
- Be consistent in whether you disable interrupts or whether you acquire a lock to protect a certain data structure: mixing the two is dangerous and can lead to deadlocks

Debugging kernel modules

- Debugging kernel modules with GDB is tricky—GDB needs to know both what the symbols are (from the .ko file) and where in the kernel they are located
- We have the kernel object (.ko) file, but how can we figure out where in the running kernel the symbols are located?
  - Answer: the kernel tells us!

Debugging kernel modules

- After loading a kernel module in Qemu, look under `/sys/modules/[module-name]/sections/` to see a file for each of its sections:

```
> cd /sys/module/ext2undelete/sections/
> ls -A
.bss .init.text .smp_locks .text
.exit.text .note.gnu.build-id .strtab
__mcount_loc .gnu.linkonce.this_module
.rodata .symtab
```

Debugging kernel modules

- The contents of each file is the address within the kernel of the corresponding section:

```
> cat .text .rodata .bss
0xffffffffa0000000
0xffffffffa0001030
0xffffffffa0002260
```

Debugging kernel modules

* Next, connect GDB to your running Qemu instance using the directions on the VM Info course page, then load the module file's symbols:

```
(gdb) add-symbol-file 451repo/project3/ext2undelete.ko \
      0xffffffffa0000000 -s .rodata 0xffffffffa0001030 \
      -s .bss 0xffffffffa0002260
add symbol table from file
"451repo/project3/ext2undelete.ko" at
.text_addr = 0xffffffffa0000000
.rodata_addr = 0xffffffffa0001030
.bss_addr = 0xffffffffa0002260
(y or n) y
Reading symbols from
451repo/project3/ext2undelete.ko...done.
```

Debugging kernel modules

- Now we're set! Can examine symbols, set breakpoints, etc. from the comfort of GDB
- (Show demo here)
- This material is also available as a tutorial on the course website

Project 3 tips

- How can we figure out which inodes have been deleted?
- First step: Check the inode bitmap
  - The bits of the inode bitmap describe which inodes are currently in use
  - If the address of the inode bitmap is `ib_ptr`, how can we test if the `n`th inode is not in use?
- Second step: Check whether the inode was actually deleted
  - What tells us that an inode was deleted as opposed to simply never having been used?

Project 3 tips

- As an aside, `arch/arm/include/asm/bitops.h` defines a number of efficient bitwise operators
- When `ext2_new_inode` in `fs/ext2/ialloc.c` looks for the next available inode number, it (indirectly) invokes the `find_first_zero_bit_le` function, which finds the index of the first zero bit for a little endian integer of a given size

Project 3 tips

- There are many scenarios to test to make sure your undelete module is working…check as many as you can!
  - Calls to `undelete_read` with a small buffer size (for example, a single byte)
    - Should advance `buffer_read_offset` without reading the next block
  - File systems spanning multiple block groups
  - File systems with a variety of block sizes