CSE 451: Operating Systems

Section 3:
Project 0 recap, Project 1

Project 0

Project 0: queue problems

* Must check for empty queues before reversing or sorting

Project 0: common hash table problems

* Linear probing misunderstandings
  * Must mark cells as *vacated* (different than free)
* Consider hash table size of 10
  * Two inserts:
    * key1 -> hash = 5
    * key2 -> hash = 15
  * Linear probing: will occupy slots 5 and 6
  * Delete key1
  * Lookup key2: slot 5 is empty, but need to check 6!

Project 0: other hash table problems

* Not improving the test code
* Memory leaks

Coding style

* Please indent consistently
  * **man 1 indent**
  * Let your text editor do it for you!
    * vi: press **1 G**, then press **= G**
* Add comments for complex code

Memory management

* The problem of *ownership*:

```
void do_stuff(char *buf, int len) {
   ...
   free(buf);
}

int main() {
   char *mybuf =
       (char *)malloc(LEN*sizeof(char));
   do_stuff(mybuf, LEN);
   ...
   free(mybuf);     // Double free: undefined
                    // behavior!
}
```


Memory management

* Always be *explicit* about who owns memory
  * If a function allocates some memory that the caller must free, say so!
  * If a function frees some memory that the caller should no longer use, say so!
* Define pairs of allocate and free functions
  * Ideally, whoever calls the allocate function also calls the free function; if not, carefully consider usage

Advanced memory management

* What if multiple processes or threads are accessing the same structure in memory?
  * When can we *free*?
  * What technique can we use to track who is using the memory?
    * Reference counting

Kernel memory management

* How does memory management within the kernel differ?
  * Different pools of memory
    * e.g. for devices that use DMA
  * Different priority / reliability requirements
    * Can the kernel block/sleep when allocating memory?
  * Different performance requirements
    * Frequent allocations use "slab allocator" [Bonwick '94]
* Kernel memory allocation functions: `kmalloc()`, `vmalloc()`

Project 1

* Due Wednesday at 11:59pm!
* Follow turnin instructions carefully
  * Only one group member needs to run **turnin**
* The writeup is a critical component
  * Don't forget to include your group number and everybody's name in the writeup

Project 1: turnin

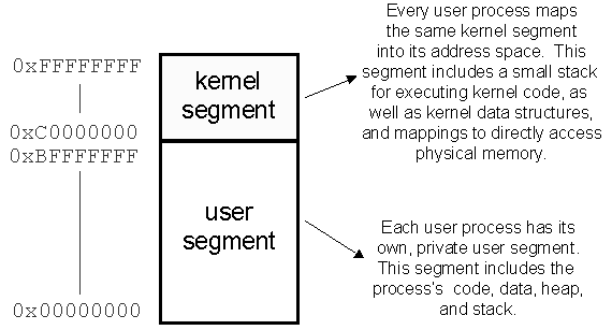
* Preserve directories when submitting changed files
  * When we extract your changed files, they should go to the right directory, so it is unambiguous which file you changed
  * This is easy to do with the **tar** command
    * Recommendation: double-check your turnin on a clean copy of the kernel
* Writeup requires a list of modified files (#7): please use full path name

Developing on `forkbomb.cs.washington.edu`

* What's a forkbomb?
* How do you stop a forkbomb?
  * If you're still logged in, try **killall**
  * If you can't log in, e-mail support@cs!
    * The TAs can't do anything about it

Project 1: system calls

* Special functions must be used to copy data between user space and kernel. Why?



Linux kernel memory safety

* `copy_from_user()`, `copy_to_user()`, `access_ok()`: look for example usage in kernel
  * Definition, gory details: *arch/x86/lib/usercopy_32.c*

Libraries

* Linux has two types of executable programs:
  * Statically linked
  * Dynamically linked
* What are the benefits of each?
* Are these library calls or system calls?
  * `strlen()`
  * `exec()`
  * `execvp()`
  * `execve()`
  * `fork()`

What kind of executable are you creating for project 1?

Project 1: example Makefile

```
all: standalone linked

# Produces getexeccounts.o:
getexeccounts: getexeccounts.c getexeccounts.h
    gcc -c getexeccounts.c

# Produces getcounts.a:
library: getexeccounts
    ar r getcounts.a getexeccounts.o

standalone: getexeccounts
    gcc -o getdriver_standalone getdriver.c \
        getexeccounts.o

linked: getexeccounts library
    gcc -o getdriver_linked getdriver.c getcounts.a

clean:
    rm -f *.o *.a getdriver_standalone \
        getdriver_linked
```


Useful commands for libraries & syscalls

- **strace**: trace system calls
- **ltrace**: trace library calls
- **ldd**: list shared libraries program depends on
- **objdump**: display info from object files
- **readelf**: display info from executable files
- **strings**: print strings found in a binary file

Project 1: debugging

- How will you debug project 1?
- Recommendation: implement and test one basic step at a time
- Debug messages: `printk()`
  - Where does `printk()` output go?
    - Possibly to console (*include/linux/kernel.h*: defines `KERN_XYZ` log levels)
    - **dmesg** command
    - */var/log/messages*

Project 1: testing

- How will you test project 1?
- Check `execounts` correctness by comparing its output to output from other tools
- Test bad input: to shell, to system call
- Shell must be able to read commands from a file: use this for testing!
- What else?

Project 1 tips

* Re-read the project description for hints
* Read the man pages!
* Navigating Linux kernel code: **ctags**, **cscope**
* Use the discussion board