Kthreads, Mutexes, and Debugging

Sarah Diesburg
CS 3430
Operating Systems
Story of Kernel Development

Some context…
In the old days…

- There were no modules or virtual machines
- The kernel is a program
  - Has code, can compile, re-compile, make executable
  - When changes needed to be made, developers make changes in source and re-compile
How is the kernel different from a regular program?

- Mostly in how it is executed
  - Boot loader loads the kernel image/executable during boot time
  - Sets kernel mode
  - Jumps to the entry point in the image/executable

- Remember the generic booting sequence?
Quick Question

How would you make changes to the kernel and run those changes?
1. Make changes to the source
2. Re-complie the kernel source
3. Re-install the kernel source
4. Make sure the bootloader sees the new kernel image (grub)
5. Reboot and profit!
Getting more modern..

- Modules were created as bits of code that can be loaded and unloaded by the kernel in kernel mode.
- Made development easier.
  - Instead of re-compiling, re-installing, and rebooting into the new kernel, one could just re-compile and load a module
Quick Question

How would you make changes to a module and run those changes?

1. Make changes to module source code
2. Re-compile the module
3. Load the new module
Present Day

- Reboots into new kernels and loading new modules often freezes machines
- Enter virtual machine software
  - Process that emulates the hardware necessary to run an OS in user-space
  - Guest OS is executed inside the virtual machine process!
Kthreads

Run the main logic of your module in a kthread!
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)
{
    printk(KERN_ALERT "Hello, world!\n");
    return 0;
}

static void hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, sleepy world.\n");
}

module_init(hello_init);
module_exit(hello_exit);
Kernel Modules

- Remember, kernel modules are very **event-based**
- We need a way to start an independent thread of execution in response to an event
  - e.g. `start_printer()` for project 2…
**kthread_run**

```
kthread_run(threadfn, data, namefmt, ...)
```

- Creates a new thread and tells it to run
  - `threadfn` – the name of the function the thread should run
  - `data` – data pointer for `threadfn` (can be NULL if the function does not take any args)
  - `namefmt` – name of the thread (displayed during “ps” command)
- Returns a task_struct
struct task_struct *t;

t = kthread_run(run, NULL, "penguin");
    if (IS_ERR(t)){
        ret=PTR_ERR(t);
    }

**kthread_stop**

```c
int kthread_stop(struct task_struct *k);
```

- **Sets** `kthread_should_stop` for `k` to return true, wakes the thread, and waits for the thread to exit.
- Returns the result of the thread function.
ret=kthread_stop(t);
if(ret != -EINTR)
    printk("Main logic thread stopped.\n");
static int run(void *arg)
{
    /* Lock here */

    while(!kthread_should_stop()) {

        /* Do stuff */
    }

    /* Unlock here */

    printk("%s: kernel thread exits.\n", __FUNCTION__);
    return 0;
}
Sample kthread code

- Take a look at the sample module that spawns a kthread on load
  - You will want to move this kthread to start when user writes a “0” to /proc/penguin
  - You will want to stop the kthread when user writes a “-1” to /proc/penguin
Concurrent Aspects of Project 2

- Synchronizing access to request queue(s)
  - Multiple producers may access request queue(s) at the same time
  - Multiple consumers may access request queue(s) at the same time

- Synchronizing access to other global data
Kitchen Queue Concurrency

- Jobs may appear on the queue at the same time the printer module checks the queue.
- The status may be read at the same time that you're updating:
  - Number of jobs that you've serviced
  - Which job is currently being processed
  - Which slot in the queue printer is looking at
- How do you guarantee correctness?
Global Data vs. Local Data

- **Global data** is declared at global scope, e.g. outside of any function body
  - Often necessary for kernel programming
- Particularly sensitive to concurrency issues
  - Be **extra** careful when handling globals
Global Data vs. Local Data

- *Local data* is declared within a function
- Local data is sensitive to concurrency issues when it depends on global data or when parallel access is possible
  - Think carefully about whether it needs to be synchronized
Synchronization Primitives

- Semaphores
  - User space
  - Kernel space
- Mutexes
  - User space
  - Kernel space
- Spin locks
  - Kernel space
- Atomic Functions
Synchronization Primitives (We'll Only Cover These)

- Mutexes
  - User space
  - Kernel space

- Does anyone remember the differences between a mutex and semaphore?
The Mutex

Caught up in the mutex?
Mutexes

- Mutex – A construct to provide MUTual EXclusion
- Based on the concept of a semaphore
- Found at `<source_dir>/include/linux/mutex.h`
- Can be locked or unlocked
  - Only one thread of execution may hold the lock at a time
Kernel-Space Mutex - Initialization

- `mutex_init(&mutex)`

- Declare and initialize a mutex
  - Only initialize it once
Kernel-Space Mutex - Locking

- `void mutex_lock(struct mutex *)`;
- `int mutex_lock_interruptible(struct mutex *)`;

- `mutex_lock()` can wait indefinitely
- `mutex_lock_interruptible()` locks a mutex as long as it is not interrupted
  - returns 0 if locking succeeded, < 0 if interrupted
- Use of the interruptible version is typically preferred
Kernel-Space Mutex – Unlocking

- void mutex_unlock(struct mutex *);

- Guarantees that the mutex is unlocked
  - Why is there no interruptible version of this function?
/* Declare your mutex */
struct mutex my_mutex;

/* Initialize your mutex */
mutex_init(&my_mutex);

/* Lock */
if(mutex_lock_interruptible(&my_mutex))
    return -ERESTARTSYS;

    /* Do stuff to protected global variables */

/* Unlock */
mutex_unlock(&my_mutex);
User-Space Mutex

- Also used with pthreads in regular user applications
  - pthreads operate very similar to kthreads
  - Might be useful if you are prototyping your elevator in user-space before porting to kernel
User-Space Mutex - Initialization

- `int pthread_mutex_init(pthread_mutex_t *, NULL);`
- `int pthread_mutex_destroy(pthread_mutex_t *);`

- Pthread_mutex_init() dynamically allocates a mutex
- Pthread_mutex_destroy() frees a mutex
User-Space Mutex - Locking

- int pthread_mutex_lock(pthread_mutex_t *);

- Returns 0 on locking, < 0 otherwise
User-Space Mutex - Unlocking

- `int pthread_mutex_unlock(pthread_mutex_t *);`

- Returns 0 on unlocking, < 0 otherwise
Printer Scheduling Advice
General Advice

- Just make printer work first
  - Use a very simple algorithm
  - My printer searches the queue in round-robin fashion and processes every maintenance request

- Optimize if there is time
Round Robin

- **Method:**
  - Service requests in round-robin fashion (e.g. queue slot 0, 1, 2, 3, etc.)

- **Pros/Cons?**
Shortest Job First

- Method:
  - Service fastest orders first

- Pros/Cons?
Hybrid

- Combine methods, come up with something new
- Up to your creativity
Getting Help

- Class time to answer questions
- Regular office hours
- Email
Other Hints

- This is not a simple project
  - Setup is different
  - You need to use different files and methods of compilation/running
  - Do NOT wait until 2 days before it is due to start
    - Too late
  - The Internet will likely NOT be very helpful
Other Hints

- Set milestones
- Ask questions early
  - If it’s a good question, I’ll share it with the class
Debugging
Kernel Debugging Configurations

- Timing info on printk
- Detection of hung tasks
- Kernel memory leak detector
- Mutex/lock debugging
- Kmemcheck
- Check for stack overflow
Select Kernel Hacking

Arrow keys navigate the menu.  <Enter> selects submenus --->.
Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
<M> modularizes features.  Press <Esc><Esc> to exit, ? for Help, </ for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >

---

Networking support --->
Device Drivers --->
Firmware Drivers --->
File systems --->
Kernel hacking --->
Security options --->
Cryptographic API --->
Virtualization --->
Library routines --->

<Select>  < Exit >  < Help >
Enable Debugging Options

.show timing information on printks
[*] Enable __deprecated logic
[*] Enable __must_check logic
(1024) Warn for stack frames larger than (needs gcc 4.4)
[*] Magic SysRq key
[ ] Strip assembler-generated symbols during link
[*] Enable unused/obsolete exported symbols
**- Debug Filesystem
[ ] Run 'make headers_check' when building vmlinux
[*] Kernel debugging
[ ] Debug shared IRQ handlers
Debugging through reads to /proc/printer

- Necessary to help you “see” what’s going on!
- General process
  - Identify data to monitor in your module
  - Run your module
  - Query /proc/kitchen for that information at any time
Kernel Oops and Other Errors

- Kernel errors often only appear on first tty (terminal interface)
  - Why?
Oops!

Unmounting local filesystems... umount: tmpfs busy - remounted read-only done.
Unable to 1 paging request at virtual address f8fb37dc

Oops:

Modules linked in: bnep rfcmm hidp l2cap irda crc_ccitt binfmt_misc ipv6 finerq hci_hcd uhci_hcd tg3 ohci1394 yenta_socket rsrcc_nonstatic pcmcia_core nls_isozone wbsd mmc_block mmc_core tun msr cpuid cpufreq_stats container video hotplug battery ac speedstep_centrino freq_table processor sr_mod sbp2 scsi_mod

CPU: 0
EIP: 0060: [c02bbf60] Tainted: P VLI
EFLAGS: 00010282 (2.6.13-rc5-x300)
EIP is at suspend_device+0x8a/0x17b
eax: f8fb3640 ebx: f71b8be4 ecx: 00000000 edx: 00000000
esi: f71b8be4 edi: 00000000 ebp: 00000003 esp: f688de44
ds: 007b es: 007b ss: 0068

Process halt (pid: 7171, threadinfo=f680c000 task=f6d08020)
Stack: c038c83f 00000066 f680de6c c011d134 c1a06aa0 00000246 f71b8ca4 f71b8ce3
 f71b8d3c f71b8be4 00000000 00000003 c02bc102 f71b8be4 00000000 00000003
4321fedc bf8b9f29 b7f8bb80 f680c000 c0139e94 00000003 00000003 00000000

Call Trace:

activate_task+0x61/0x70
device_suspend+0xcf/0x1d9
kernel_power_off+0x35/0x4e
Unable to handle kernel paging request at virtual address f8fb37dc
printing eip:
c02bbf60
*pde = 3774f067
Oops: 0000 [#1]
PREEMPT

Reason for failure
done.
Unmounting local filesystems...umount: tmpfs busy - remounted read-only
done.
Unable to handle kernel paging request at va: 0000000000000000
printing eip:
    c02bbf60
*pde = 3774f067
Oops: 0000 [#1]
PREEMPT
Modules linked in: bnep rfcomm hidp l2cap irda crc_ccitt binfmt_misc ipv6 firewall
    snd_pcm OSS snd_mixer_OSS snd_pcm snd_timer snd_soundcore snd_page_alloc
    hci_hcd uhci_hcd tg3 ohci1394 yenta_socket rsara_nonstatic pcmcia_core nls_iso
dooh wbsd mmc_block mmc_core tun msr cpuid cpufreq_stats container video hotk
ormal battery ac speedstep_centrino freq_table processor sr_mod sbp2 scsi_mod
CPU: 0
EIP: 0060:[<c02bbf60>] Tainted: P VLI
EFLAGS: 00010282 (2.6.13-rc5-x300)
EIP is at suspend_device+0xa8/0x17b
eax: f8f3b3640 ebx: f71b8be4 ecx: 00000000 edx: 00000000
esi: f71b8be4 edi: 00000000 ebp: 00000003 esp: f680de44
dsx: 007b es: 007b ss: 0068
Process halt (pid: 7171, threadinfo=f680c000 task=f6d08020)
Stack: c038c83f 00000066 f680de6c c011d134 c1a6aaa0 00000246 f71b8ca4 f71b8cc
    f71b8d3c f71b8be4 00000000 00000003 c02bc102 f71b8e4 00000003 00000000
4321fedc bf88b9f29 b7f88e80 f680c000 c0139e94 00000003 00000000 00000000
Call Trace:
    [<c011d134>] activate_task+0x61/0x70
    [<c02bc102>] device_suspend+0xc0/0x1d9
    [<c0139e94>] kernel_power_off+0x35/0x4e
Call Trace

activate_task+0x61/0x70
device_suspend+0xcf/0x1d9
kernel_power_off+0x35/0x4e
sys_reboot+0x181/0x1af
__group_send_sig_info+0xcb/0xe9
preempt_schedule+0x4a/0x56
kill_proc_info+0x69/0x6b
sys_kill+0x5b/0x62
do_ioctl+0x2d/0x81
vfs_ioctl+0x61/0x1fb
sys_ioctl+0x3c/0x5a
syscall_call+0x7/0xbb

halt (pid: 7171, threadinfo=f680c000 task=f6d08020)

c038c83f 00000000 f680de6c c011d134 c1a06aa0 00000246 f71b8ca4 f71b8cecc
f71b8ce4 c71b8be4 00000000 00000003 c02bc102 f71b8be4 00000003 00000003
4321fedc bf8b9f29 b7f88e80 f680c000 c0139e94 00000003 00000003 00000000
Failed command