Further adventures with GeekOS

David Hovemeyer

http://geekos.sourceforge.net/
Outline

• Where we left off

• New stuff
  ▶ Context switch
  ▶ Scheduling and timeslicing
  ▶ User mode
  ▶ Thread synchronization

• Work in progress

• The future

GeekOS — http://geekos.sourceforge.net/
Where we left off

- Old version of the code, April 2001
- Boots from floppy, executes 32 bit C code
- Drivers for keyboard and VGA text mode
- Cooperatively scheduled threads
  - Kernel-mode only
  - Round robin scheduling (no priorities)
  - No scheduling in interrupt return code; interrupted thread will continue executing until it voluntarily gives up the CPU
  - `Wait()` and `Wake_Up()` functions for events

GeekOS — http://geekos.sourceforge.net/
Outline

• Where we left off

• New stuff
  ▶ Context switch
  ▶ Scheduling and timeslicing
  ▶ User mode
  ▶ Thread synchronization

• Work in progress

• The future
Background

- This talk describes the code as of October 12th, 2001
- ‘User mode thread’ == Process
- Some threads stay in kernel mode all the time
Context switch

• To have real preemptive multitasking, we need to be able to choose a new thread when returning from an interrupt (for example, the timer interrupt)

• Obviously, the context of the original thread must be saved first

• Originally the context was saved in the Registers struct in the Kernel_Thread object

• However, the Interrupt_State struct already contains enough context to fully suspend and resume the thread — why not use it instead?

• Solution: use iret for both thread activation and resumption from interrupt

GeekOS — http://geekos.sourceforge.net/
iret based context switch

- Change `Switch_To_Thread` to make the stack look like an interrupt has occurred (lowlevel.asm, line 267)
  - Push values for `eflags` and `cs` before return address
  - Then use the `Save_Registers` macro to save context, just as though we were handling an interrupt
  - Save stack pointer in thread object

- Change `Restore_Thread` to simply switch to the stack of the new thread, `Restore_Registers`, and execute `iret`
  - There are some other details having to do with scheduling and user mode, but that’s the basic idea

- Now the only context information stored explicitly in the `Kernel_Thread` structure is the stack pointer (the `esp` field)
Scheduling in the interrupt return code

• With the (voluntary) context switch code using the `{Save,Restore}_Registers` macros and the `iret` instruction, it’s easy to suspend/resume threads from the interrupt return code

• Handle INTERRUPT, lowlevel.asm, line 216
  ▶ After calling the C handler function, check the `g_needReschedule` flag
  ▶ If set, save stack pointer, choose a new thread to run, and switch to its stack
Scheduling and timeslicing

• Now that interrupt handlers can force a new thread to be chosen, we can implement timeslicing by installing a handler for the timer interrupt (in timer.c)

▷ Threads are allowed to execute for a given number of ticks, then they are put on the end of the run queue and a new thread is chosen

▷ When choosing a new thread to run, selection is based on static priority (Find_Best(), kthread.c, line 350)
  ◦ Yes, this is cheesy
  ◦ Good project: a implement real scheduler with dynamically adjusted priorities
User mode

- To be a proper OS, we obviously want user programs to be isolated from the kernel and other user programs

- The x86 gives us a really easy way to implement this using segmentation

  ▶ Create_User_Context(), user.c, line 52
  ▶ Allocate a flat chunk of memory for the user program (code and data, like .com files in DOS)
  ▶ Create user-level (ring 3) code and data segments describing the memory chunk
  ▶ Put the segments in a local descriptor table (LDT)
  ▶ By having a separate LDT for each user context, user programs cannot access each other’s code or data
User mode programs

• What is a user-mode program?
  ▶ We prepare the images for user mode programs in much the same way as the kernel image (ld, objcopy), except that user programs are linked at base address 0
  ▶ These images can be copied directly into user memory
  ▶ Because we have no filesystem, user programs are linked directly into the program as data structures (see the rule for uprogs.c in the Makefile)
  ▶ Start_User_Program(), user.c, line 218: takes a User_Program data structure and starts a user-mode thread (process)
Stack switch, privilege transition

- When an interrupt occurs in user mode (causing a transition to kernel mode), we would like to handle it on the kernel thread’s stack.

- The x86 TSS data structure allows us to specify a particular stack for each privilege level.

- `user.c`, line 207: setting the kernel stack pointer before entering user mode.

- When an interrupt causes a privilege transition, the processor pushes the original stack segment selector and stack pointer onto the receiving (kernel) stack before the usual `eflags`, `cs`, and `eip` sequence.
Note about the TSS

- A TSS (task state segment) is an x86-specific data structure that represents the state of a task

- They can be used to take advantage of the x86’s automatic task switch mechanism (using one TSS for each thread or process)

- GeekOS currently uses a single TSS, only for the purpose of specifying the kernel stack of the current thread for interrupts which occur while executing in user mode

- In fact, just think of the TSS’s esp0 and ss0 fields as special processor registers for specifying the location of the current kernel stack
Entering user mode

- To summarize, when entering user mode (via an iret instruction):
  - restore user mode registers (RestoreRegisters macro)
  - call Switch_User_Context(), user.c, line 162, which
  - checks to see if we’re actually returning to user mode
  - if so, checks to see if the context we’re returning to is different than the current one (lazy switching)
  - if so, switches to the new user context’s LDT
  - and switches to the new thread’s kernel stack

- Once all of the above has taken place, iret will transfer control back into user mode
System calls

- System calls are software interrupts generated from user mode
- GeekOS uses interrupt 0x90
- This interrupt descriptor of this interrupt gate has dpl set to 3, to allow access from user mode (see idt.c, line 64)
- System call number is passed in the eax register
- Arguments are passed in other registers
- Return value is passed in the eax register
Example system call (user mode side)

- test/libuser.c, line 52:

```c
int Print_String( const char* message )
{
    int num = SYS_PRINTSTRING, rc;
    size_t len = strlen( message );

    __asm__ __volatile__ ( 
        "int $0x90" 
        : "=a" (rc) 
        : "a" (num), "b" (message), "c" (len) 
    );

    return rc;
}
```

GeekOS — http://geekos.sourceforge.net/
Example system call (kernel side)

- trap.c, line 32 (system call trap handler)

```c
static void Syscall_Handler( struct Interrupt_State* state )
{
    unsigned int syscallNum = state->eax;
    int rc;

    if ( syscallNum >= g_numSyscalls ) {
        // ... illegal system call, kill current thread ...
        return;
    }
    Enable_Interrupts(); // system calls run with ints enabled
    rc = g_syscallTable[ syscallNum ]( state ); // call handler
    state->eax = rc; // return value in eax
    if ( Interrupts_enabled() )
        Disable_Interrupts(); // disable ints for int return
}
```

GeekOS — http://geekos.sourceforge.net/
Example system call (kernel side)

- syscall.c, line 48

```c
static int Sys_Printf( struct Interrupt_State* state ) {
    const void* userPtr = (const void*) state->ebx;
    unsigned int length = state->ecx;
    unsigned char* buf;

    if ( length > 1024 ) return -1;
    buf = Malloc_Atomic( length + 1 );
    if ( buf == 0 ) return -1;
    if ( !Copy_From_User( buf, userPtr, length ) ) return -1;
    buf[ length ] = '\0';
    Mutex_Lock( &s_screenLock );
    Print( "%s", buf );
    Mutex_Unlock( &s_screenLock );
    Free_Atomic( buf );
    return 0;
}
```
Thread synchronization

• Threads are expected, as much as possible, to run with interrupts enabled
  ▶ Interrupts are always enabled in user mode
  ▶ They may be disabled in kernel mode

• Mutual exclusion and waiting on a condition are needed when accessing shared kernel data structures

• \{Disable,Enable\}_Interrupts(), Wait(), and Wake_Up() can be used, but are somewhat clumsy

• Better solution: mutexes and condition variables
Mutexes and condition variables

- Implemented as header file synch.h, structs Mutex and Condition

- These work very much like pthread_mutex_t and pthread_cond_t from POSIX threads
  - Mutex has lock and unlock operations
  - Condition has wait, signal, and broadcast operations

- Implemented internally using {Disable, Enable}_Interrupts(), Wait(), and Wake_UP()
Other new thread operations

- **Exit()**: explicitly exit a thread (rather than returning from its start function)

- **Join()**: wait for a thread to exit
  - Currently, must be thread’s parent to be allowed to `Join()`
  - Lots of potential race conditions here
Outline

• Where we left off

• New stuff
  ▶ Context switch
  ▶ Scheduling and timeslicing
  ▶ User mode
  ▶ Thread synchronization

• Work in progress

• The future
Work in progress

• I’m working on message passing with the following operations:

  struct Mailbox* Create_Mailbox( void* buf, unsigned long bufSize, Boolean isBlocking );
  int Send( struct Mailbox* mb, const void* data, unsigned long sz );
  void Receive( struct Mailbox* mailbox, unsigned long* sz );
  void Acknowledge( struct Mailbox* mailbox );
  void Receive_Any( struct Mailbox** pMailbox, unsigned long* sz );

  ▶ A mailbox has a single fixed size buffer
  ▶ Sender calls Send() (blocks until buffer is available)
  ▶ Recipient must Receive() (blocks until a message is sent), then
  Acknowledge() (unblocks sender if mailbox is blocking, makes
  mailbox available again)
  ▶ Receive_Any() is like Receive(), but blocks until any mailbox
  has a message available
Work in progress

- Eventually, I would like to use message passing for all I/O and IPC (i.e., microkernel)

- More thought needed here
Outline

- Where we left off

- New stuff
  - Context switch
  - Scheduling and timeslicing
  - User mode
  - Thread synchronization

- Work in progress

- The future
The future

- Stuff that might get done at some point:
  - Proper scheduler with dynamically adjusted priorities
  - Real C library (maybe newlib from RedHat)
  - Mass storage drivers (floppy, IDE disk)
  - Filesystem layer
  - Framebuffer, window system
  - Misc. devices (mouse, serial port, parallel port)
  - Network support (maybe)
  - Paging, virtual memory (maybe)
  - Ports to other architectures?
  - Nethack

- Volunteers?

GeekOS — http://geekos.sourceforge.net/