System Calls

COMS W4118

References: Operating Systems Concepts (9e), Linux Kernel Development, previous W4118s
Copyright notice: care has been taken to use only those web images deemed by the instructor to be in the public domain. If you see a copyrighted image on any slide and are the copyright owner, please contact the instructor. It will be removed.
• Processes can’t access anything outside address space
• How do they communicate with outside world?
System calls

• User processes cannot perform privileged operations themselves

• Must request OS to do so on their behalf by issuing system calls

• System calls elevate privilege of user process
  – Must ensure kernel is not tricked into doing something a user process should not be doing
  – Must verify every single parameter!
Library vs. System Calls

• C program invoking printf() libc library call, which calls write() system call
### Examples of Windows and Unix System Calls

<table>
<thead>
<tr>
<th>Windows</th>
<th>Unix</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Process Control</strong></td>
<td></td>
</tr>
<tr>
<td>CreateProcess()</td>
<td>fork()</td>
</tr>
<tr>
<td>ExitProcess()</td>
<td>exit()</td>
</tr>
<tr>
<td>WaitForSingleObject()</td>
<td>wait()</td>
</tr>
<tr>
<td><strong>File Manipulation</strong></td>
<td></td>
</tr>
<tr>
<td>CreateFile()</td>
<td>open()</td>
</tr>
<tr>
<td>ReadFile()</td>
<td>read()</td>
</tr>
<tr>
<td>WriteFile()</td>
<td>write()</td>
</tr>
<tr>
<td>CloseHandle()</td>
<td>close()</td>
</tr>
<tr>
<td><strong>Device Manipulation</strong></td>
<td></td>
</tr>
<tr>
<td>SetConsoleMode()</td>
<td>ioctl()</td>
</tr>
<tr>
<td>ReadConsole()</td>
<td>read()</td>
</tr>
<tr>
<td>WriteConsole()</td>
<td>write()</td>
</tr>
<tr>
<td><strong>Information Maintenance</strong></td>
<td></td>
</tr>
<tr>
<td>GetCurrentProcessID()</td>
<td>getpid()</td>
</tr>
<tr>
<td>SetTimer()</td>
<td>alarm()</td>
</tr>
<tr>
<td>Sleep()</td>
<td>sleep()</td>
</tr>
<tr>
<td><strong>Communication</strong></td>
<td></td>
</tr>
<tr>
<td>CreatePipe()</td>
<td>pipe()</td>
</tr>
<tr>
<td>CreateFileMapping()</td>
<td>shmget()</td>
</tr>
<tr>
<td>MapViewOfFile()</td>
<td>mmap()</td>
</tr>
<tr>
<td><strong>Protection</strong></td>
<td></td>
</tr>
<tr>
<td>SetFileSecurity()</td>
<td>chmod()</td>
</tr>
<tr>
<td>InitializeSecurityDescriptor()</td>
<td>umask()</td>
</tr>
<tr>
<td>SetSecurityDescriptorGroup()</td>
<td>chown()</td>
</tr>
</tbody>
</table>
System Call Dispatch

- How should actual system call be invoked?
  - Program can’t see kernel namespace

- Need hardware support to change privilege level

- Traps
  - Type of interrupt
  - Software interrupts and exceptions
  - Software interrupts initiated by programmer
  - Exceptions occur automatically
Traps, Interrupts,Exceptions

• On x86, int n (n=0:255) calls interrupts n
• Some interrupts are privileged
• Can’t be called by user mode
• Others aren’t, e.g., syscalls
• Processor transitions to privileged mode when handling interrupt

for(;;) {
  if (interrupt) {
    n = get interrupt number
    call interrupt handler n
  }
  fetch next instruction
  run next instruction
}

for(;;) {
  fetch next instruction
  run next instruction {
    if (instr == "int n")
      call interrupt handler n
  }
  if (error or interrupt) {
    n = get error or interrupt type
    call interrupt handler n
  }
}

for(;;) {
  if (interrupt) {
    n = get interrupt number
    call interrupt handler n
  }
  fetch next instruction
  run next instruction
}
Three kinds of interrupts

**Interrupt Process (from three potential sources):**

- **Hardware:**
  - Interrupt Request (IRQ) sent from device to processor

- **Processor:**
  - Exception / Trap sent from processor to processor
  - Processor halts thread execution
  - Processor saves thread state
  - Processor executes interrupt handler
  - Processor resumes thread execution

- **Software:**
  - Software Interrupt instruction loaded by processor
1. Kernel assigns system call type a system call number
2. Kernel initializes system call table, mapping system call number to functions implementing the system call
   – Also called system call vector
3. User process sets up system call number and arguments
4. User process runs int X (on Linux, X=80h)
5. Hardware switches to kernel mode and invokes kernel’s interrupt handler for X (interrupt dispatch)
6. Kernel looks up syscall table using system call number
7. Kernel invokes the corresponding function
8. Kernel returns by running iret (interrupt return)
Linux System Call Dispatch

To find code for a Linux syscall: http://syscalls.kernelgrok.com
System call parameter passing

• Typical methods
  – Pass via registers (e.g., Linux)
    • More parameters than registers?
  – Pass via user-mode stack
    • Complex: user mode and kernel mode stacks
  – Pass via designated memory region
    • Address passed in register
Linux System Call Parameter Passing

- Syscalls with fewer than 6 parameters passed in registers
  - %eax (syscall number), %ebx, %ecx, %esi, %edi, %ebp
- If 6 or more arguments
  - Pass pointer to block structure containing argument list
- Maximum size of argument is register size
  - Larger arguments passed as pointers
  - Stub code copies parameters onto kernel stack before calling syscall code (kernel stack, will study later)
- Use special routines to fetch pointer arguments
  - get_user(), put_user(), copy_to_user(), copy_from_user
    - Include/asm/uaccess.S
    - These functions can block. Why?
    - Why use these functions?
- OS must validate system call parameters
Linux system call naming convention

- Usually the user-mode wrapper `foo()` traps into kernel, which calls `sys_foo()`
  - `sys_foo` is implemented by `DEFINEX(foo, ...)`
  - Expands to “asmlinkage long sys_foo(void)”
  - Where x specifies the number of parameters to syscall
  - Often wrappers to `foo()` in kernel

- System call number for `foo()` is `__NR_foo`
  - `arch/x86/include/asm/unistd_32.h`
  - Architecture specific

- All system calls begin with `sys_`
• Generic syscall stub provided in libc
  – _syscalln
  – Where n is the number of parameters

• Example
  – To implement: `ssize_t write(int fd, const void *buf, size_t count);`
  – Declare:
    ```c
    #define __NR_write 4 /* Syscall number */
    _syscall3(ssize_t, write, int, fd, const void*, buf, size_t count)
    ```

• Usually done in libc for standard syscalls
Tracing system calls in Linux

• Use the “strace” command (man strace for info)

• Linux has a powerful mechanism for tracing system call execution for a compiled application

• Output is printed for each system call as it is executed, including parameters and return codes

• ptrace() system call is used to implement strace
  – Also used by debuggers (breakpoint, singlestep, etc)

• Use the “ltrace” command to trace dynamically loaded library calls
System Call Tracing Demo

- `pwd`

- `ltrace pwd`
  - Library calls
  - `setlocale`, `getcwd`, `puts`: makes sense

- `strace pwd`
  - System calls
  - `execve`, `open`, `fstat`, `mmap`, `brk`: what are these?
  - `getcwd`, `write`
Interesting System Calls

• brk, sbrk: increase size of program data
  – void* sbrk(int bytes)
  – Accessed through malloc

• mmap
  – Another way to allocate memory
  – Maps a file into a process’s address space
  – Or just grab memory with MAP_ANONYMOUS
  – MAP_PRIVATE or MAP_SHARED