System Calls & Signals

CS449 Fall 2016
Operating system

- OS – a layer of software interposed between the application program and the hardware

- Two primary goal
  - Manage hardware resources
  - Abstract details about hardware
    - I/O devices abstracted by files
    - Main memory abstracted by virtual memory
    - Processors abstracted by processes
What is an OS

• Pedantically it is just a Kernel
  – A program that implements the core OS functions
    (process scheduling, memory/HW management)
• More commonly it refers to the basic environment
  – Libraries, user interface, foundational programs
• A Kernel is a program, but it is not an application
  – Never executes on its own (except at boot time)
  – Only reacts to events and requests
    • System calls, exceptions, and interrupts
OS Interactions w/ Rest of System

• With application
  – **System Calls**: Application requests the OS to do something
    (Read a file, print to console, etc...)
  – **Signals**: OS notifies an application of something
    (SEGFAULT, Killed, etc...)

• With hardware
  – **Exceptions**: Application does something that needs OS handling
    • Access invalid memory address: Send a SEGFAULT signal to application
    • Access unmapped address (**page fault**): Map physical page to address
    • Divide by 0: Send a divide-by-zero signal to application
  – **Interrupts**: Processor receives a hardware signal on an interrupt pin
    (Mouse move, keyboard type, etc...)
x86 Privilege Levels

- **Ring 3**
- **Ring 2**
- **Ring 1**
- **Ring 0**
  - Kernel
  - Device drivers
  - Applications

Legend:
- Least privileged
- Most privileged
Signals

• Notifications sent to a program by OS
  – Indicate special events that program might want to handle
  – Usually exceptions or interrupts forwarded to program by OS

• Allows **asynchronous** notification rather than **polling**

• Polling – to explicitly ask if something occurred, usually repeatedly in a loop

• Asynchronous – property of happening “out of sync” with program code, and not at specific points in code
  – Signals can happen anywhere in the middle of execution
  – Stop program -> Handle signal -> Resume
  – More efficient compared to polling periodically
### kill -l

<table>
<thead>
<tr>
<th>Signal</th>
<th>Signal</th>
<th>Signal</th>
<th>Signal</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGHUP</td>
<td>SIGINT</td>
<td>SIGQUIT</td>
<td>SIGILL</td>
<td>SIGTRAP</td>
</tr>
<tr>
<td>SIGABRT</td>
<td>SIGBUS</td>
<td>SIGFPE</td>
<td>SIGKILL</td>
<td>SIGUSR1</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>SIGUSR2</td>
<td>SIGPIPE</td>
<td>SIGALRM</td>
<td>SIGTERM</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>SIGCONT</td>
<td>SIGSTOP</td>
<td>SIGTSTP</td>
<td>SIGTTIN</td>
</tr>
<tr>
<td>SIGTTOU</td>
<td>SIGURG</td>
<td>SIGXCPU</td>
<td>SIGXFSZ</td>
<td>SIGVTALRM</td>
</tr>
<tr>
<td>SIGPROF</td>
<td>SIGWINCH</td>
<td>SIGIO</td>
<td>SIGPWR</td>
<td>SIGSYS</td>
</tr>
<tr>
<td>SIGRTMIN</td>
<td>SIGRTMIN+1</td>
<td>SIGRTMIN+2</td>
<td>SIGRTMIN+3</td>
<td>SIGRTMIN+4</td>
</tr>
<tr>
<td>SIGRTMIN+5</td>
<td>SIGRTMIN+6</td>
<td>SIGRTMIN+7</td>
<td>SIGRTMIN+8</td>
<td>SIGRTMIN+9</td>
</tr>
<tr>
<td>SIGRTMIN+10</td>
<td>SIGRTMIN+11</td>
<td>SIGRTMIN+12</td>
<td>SIGRTMIN+13</td>
<td>SIGRTMIN+14</td>
</tr>
<tr>
<td>SIGRTMIN+15</td>
<td>SIGRTMAX-14</td>
<td>SIGRTMAX-13</td>
<td>SIGRTMAX-12</td>
<td>SIGRTMAX-11</td>
</tr>
<tr>
<td>SIGRTMAX-10</td>
<td>SIGRTMAX-9</td>
<td>SIGRTMAX-8</td>
<td>SIGRTMAX-7</td>
<td>SIGRTMAX-6</td>
</tr>
<tr>
<td>SIGRTMAX-5</td>
<td>SIGRTMAX-4</td>
<td>SIGRTMAX-3</td>
<td>SIGRTMAX-2</td>
<td>SIGRTMAX-1</td>
</tr>
<tr>
<td>SIGRTMAX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Common Error Signals

• **SIGILL** – Illegal Instruction

• **SIGBUS** – Bus Error, usually caused by bad data alignment or a bad address

• **SIGFPE** – Floating Point Exception

• **SIGSEGV** – Segmentation violation, i.e., a bad address
Termination Signals

• **SIGINT** – Interrupt, or what happens when you hit CTRL + C
• **SIGTERM** – Ask nicely for a program to end (can be caught and handled)
• **SIGKILL** – Ask meanly for a program to end (cannot be caught and handled)
• **SIGABRT, SIGQUIT** – End a program with a core dump
kill

• From the shell in UNIX you can send signals to a program.

• Use ps to get a process ID

  (1) thot $ ps -af

<table>
<thead>
<tr>
<th>UID</th>
<th>PID</th>
<th>PPID</th>
<th>C</th>
<th>STIME</th>
<th>TTY</th>
<th>TIME</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>wahn</td>
<td>27500</td>
<td>27470</td>
<td>0</td>
<td>07:13</td>
<td>??</td>
<td>00:00:00</td>
<td>crashed_program</td>
</tr>
<tr>
<td>wahn</td>
<td>27507</td>
<td>27474</td>
<td>0</td>
<td>07:13</td>
<td>pts/5</td>
<td>00:00:00</td>
<td>ps -af</td>
</tr>
</tbody>
</table>

• kill it!

  kill 27500 – Sends SIGTERM

  kill -9 27500 – Sends SIGKILL
kill

- `kill()` is the system call that can send a process a signal (any signal, not just SIGKILL)

```c
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>

int main()
{
    pid_t my_pid = getpid();
    kill(my_pid, SIGSTOP);
    return 0;
}
```
Catching Signals

• Most signals can be caught and handled
• Generally bad to try to continue, signals often indicate an error or termination state
  – Do some cleanup, then exit
• Some signals can be caught safely though
  – User signals, alarm signals etc…
#include <unistd.h>
#include <signal.h>

int timer = 10;

void catch_alarm(int sig_num) {
    printf("%d\n", timer--);
    alarm(1);
}

int main() {
    signal(SIGALRM, catch_alarm);
    alarm(1);
    while (timer > 0) {
        alarm(0);
    }
    return 0;
}
Exceptions

• You’ve seen these in Java
  – Generated by errors detected by the Java Virtual Machine

• Hardware exceptions exist for the same purpose
  – HW tells OS that a program tried to do something illegal
    • Executed an illegal instruction, violated memory protection
  – OS then has to respond to the violation
    • Handle the exception and let program continue to run. E.g.
      – Page Fault: map physical page and read page in from swap space
    • Notify the program that something happened (via signal) E.g.
      – Divide-by-0: send SIGFPE to program and let it handle it
Exception Handling

• Each exception has an entry in a **dispatch table**
  – An array of function pointers to specific handlers
    • Indexed by exception number
  – Privilege level raised by CPU when raising exception
  – Registers must be saved to/restored from kernel stack

• Hardware interrupts handled by same dispatch table
  – Dispatch table also called **Interrupt Vector Table**
  – Handlers are also called **Interrupt Service Routines (ISR)**
System Call

• System calls often implemented using exceptions
  – x86 versions uses a **trap** instruction: int 0x80
    • Trap: an intentional exception to invoke a specific handler
    • Jumps to entry 0x80 in dispatch table
      (ISR that does system call handling in Linux)

• Why not a simple function call?
  – Cannot service system calls in user mode
  – Exceptions: a safe way to transition to kernel mode
    • Allows execution of higher privileged instructions to control HW
    • Allows access to protected pages containing kernel data structures
System Call

User space

1. Push arguments
2. Call printf
3. 3 Syscall no. in register
4. Return to caller

Kernel space (OS)

Dispatch

5. 6. Sys call handler

Library (printf call)

User code
strace ./hello

fstat(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(136, 7), ...}) = 0

mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x2a95557000

write(1, "Hello world!\n", 13Hello world!) = 13

exit_group(0)
Why make System Calls so Complex?

• In order to do a system call in Linux/x86:
  – Application stores system call number in $EAX
  – Application emits int 0x80
  – CPU raises interrupt 0x80 which invokes OS after raising privilege level to kernel mode
  – OS saves user registers to kernel stack
  – OS jumps to ISR 0x80 after reading interrupt status register
  – OS ISR 0x80 (syscall routine) reads $EAX to jump to correct syscall handler in system call table
  – OS restores user registers and lowers privilege level before returning

• This makes system calls more expensive than function calls
Why make System Calls so Complex?

• What is the alternative?
  – Application does a normal function call to OS
  – OS executes an instruction to raise privilege level to kernel mode
    (Hypothetically a Changemode instruction)
  – OS lowers privilege level before returning

• But is this system secure?
  – Nothing prevents application from emitting Changemode instruction
  – Applications can gain kernel privilege whenever they want!

• Exception mechanism is the single gateway into kernel mode
• …and that single gateway always leads to an OS routine
Linux Syscalls

- 325 syscall slots reserved in system call table (2.6.23.1 kernel) -- Not all are used

<table>
<thead>
<tr>
<th>Syscall</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>exit</td>
<td>Causes a process to terminate</td>
</tr>
<tr>
<td>fork</td>
<td>Creates a new process, identical to the current one</td>
</tr>
<tr>
<td>read</td>
<td>Reads data from a file or device</td>
</tr>
<tr>
<td>write</td>
<td>Writes data to a file or device</td>
</tr>
<tr>
<td>open</td>
<td>Opens a file</td>
</tr>
<tr>
<td>close</td>
<td>Closes a file</td>
</tr>
<tr>
<td>creat</td>
<td>Creates a file</td>
</tr>
</tbody>
</table>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    int fd;
    char buffer[100];
    strcpy(buffer, "Hello, World!\n");

    fd = open("hello.txt", O_WRONLY | O_CREAT);
    write(fd, buffer, strlen(buffer));
    close(fd);
    exit(0);

    return 0;
}
OR-ing Flags

- Define constants as powers of 2
- Bitwise OR to combine
- Bitwise AND to test

```c
#define O_RDONLY      0
#define O_WRONLY      1
#define O_RDWR        2
#define O_CREAT       16
```