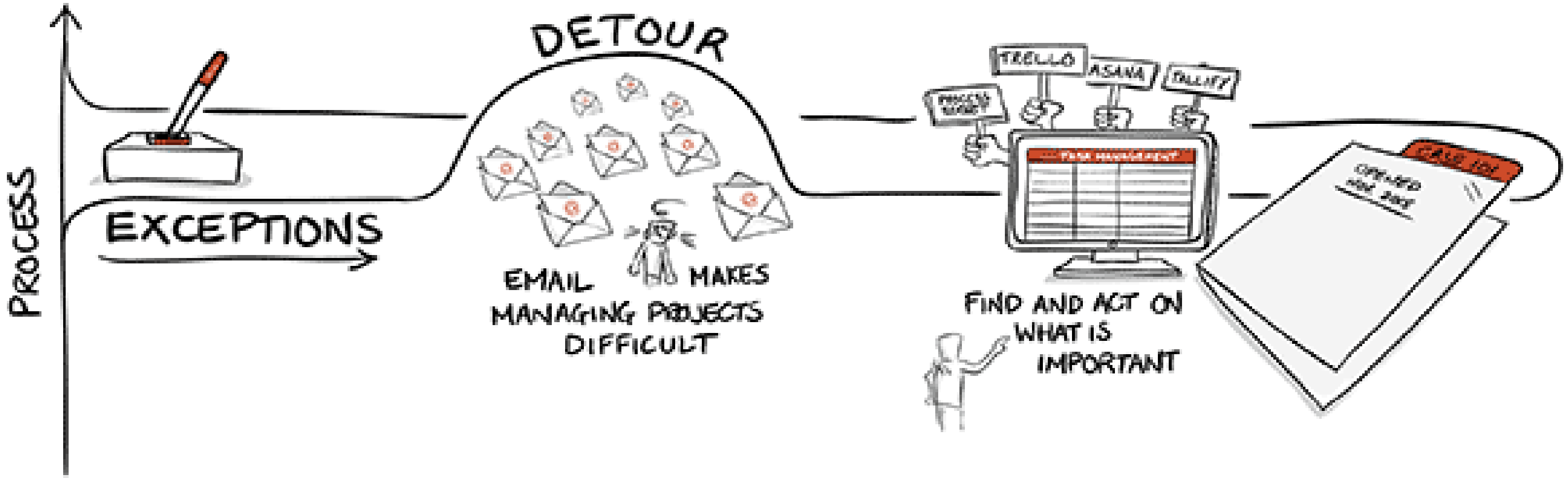
Exceptional Control Flow

Computer Systems Chapter 8



Normal Control Flow

- %rip set to the initial instruction when program loaded
- %rip updated
  - Points to next sequential instruction after decode
  - May be modified by jump/call/ret instructions
- Allows program to respond to internal state
- But what happens when things occur EXTERNAL to the program?

Reasons for Exceptional Control Flow

- Abnormal condition – e.g. segmentation violation or memory full
- I/O interrupt – e.g. requested READ completes
- Timer interrupt
- External interrupt – e.g. Ctrl-C (Kill signal)
- Operating System interrupt – e.g. swap-out
- Memory interrupt – e.g. page fault
- Network traffic – e.g. new packet arrives
- et cetera

Exceptions vs. Signals

Exceptions

- Low level X86 concept
- Implemented in hardware and software

Signals

- Higher level UNIX concept
- Implemented only in software
- Built on top of exceptions

Exception

- "an abrupt change of control flow in response to some change in the processor state"
- Change in state is called an "event"
  - e.g. segmentation violation or IO signal
- Processor responds by transferring control to "exception handler"
  - Different handlers for different exceptions : exception table
- When finished, the exception handler may:
  - Return to the instruction that was executing when the event occurred
  - Return to the next instruction after the one that was executing
  - Abort the program

“Kernel” vs. “User” Execution

- Our code runs in “User” mode
  - Runs normal x86 instructions
  - To protect the system, certain functions are disabled
    - Such as resource manipulation, cross-memory communication, etc.
  - User must INVOKE “kernel” routines with a special “syscall” instruction to invoke these functions
- Kernel Mode code
  - Trusted functions – operating system code designed and proven to prevent malicious actions
  - May only invoke other kernel functions or return to User mode
  - Uses it’s own “kernel” stack instead of the regular stack

Handling an Exception event

- When exception event occurs, it is assigned a numeric event type
- Depending on event type, return address is pushed onto the (kernel) stack
  - Either currently executing instruction, %rip, or abort routine
- Some state info is also pushed on stack (e.g. condition code flags)
- Event type is an index into exception table. Value in the exception table is the "kernel" routine to handle that exception

Classes of Exceptions

Class	Cause	Return Behavior
Interrupt	I/O event e.g. read complete	Next Instruction
Trap	Intentional Exception event e.g. “syscall” to enter kernel	Next Instruction
Fault	Potentially recoverable error event e.g. page fault	Current Instruction or abort
Abort	Unrecoverable error event e.g. RAM parity check	Abort

Exception Examples

Exc. Num	Class	Description
0	Fault	Divide by zero (Floating point exceptions)
13	Fault	Memory Protection Fault (Segmentation Fault)
14	Fault	Page Fault (4K page not in real memory)
18	Abort	Fatal hardware error
32-255	Interrupt or Trap	OS-Defined exceptions

Syscall (Trap) Examples

Num	Name	Descr
0	read	Read file
1	write	Write file
2	open	Open file
3	close	Close file
4	stat	File info
9	mmap	Map file to memory
12	brk	Reset heap
32	dup2	Copy file descriptor

Num	Name	Descr
33	pause	Wait for signal
37	alarm	Schedule alarm
39	getpid	Get process ID
57	fork	Create new process
59	execve	Load/Execute a program
60	_exit	Terminate process
61	wait4	Wait for child process
62	kill	Send signal to process

Invoking kernel functions (syscall)

- From C code:
  - Usually we use C library wrappers around functions which invoke syscall e.g. printf, sscanf, fork, execve, open, etc.
  - There is a syscall library function : `long syscall(long number, ...);`

- In X86_64:
  - Put the syscall number in %rax (see /usr/include/asm/unistd_64.h)
  - Put parameters registers: %rdi, %rsi, %rdx, %r10, %r8, %r9
  - Invoke "syscall" instruction
  - return value in %rax

Hello World Examples

```
#include <stdio.h>
int main() {
        printf("Hello world\n");
        return 0;
}
```

```
#include <unistd.h>
int main() {
        write(1,"Hello World\n",12);
         _exit(0);
}
```

```
.data
msg: .ascii "Hello World\n"
.text
        movq $1, %rax ; use the write syscall
        movq $1, %rdi ; write to stdout
        movq $msg, %rsi ; use "Hello World"
        movq $12, %rdx ; write 12 characters
        syscall
        movq $60, %rax ; use the _exit syscall
        movq $0, %rdi ; return code of 0
        syscall
```

Syscall Error Handling

- If there is a syscall error (e.g. “file not found” on open)
- Return value (%rax) set to -1
- Global variable errno (declared in errno.h) set to unique error number
- use perror(“myfunc encounterred: ”); to print an error message

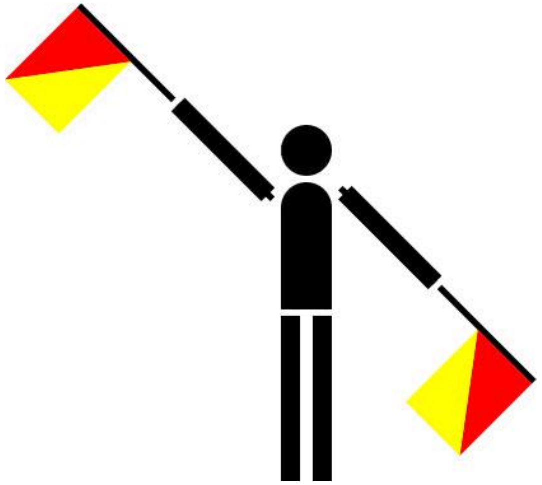
UNIX Signals

- "A signal is a small message that notifies a program that an event of some type has occurred"
- UNIX defines the list of valid signals – identified by an index
- Each signal corresponds to some type of system event (exception)
  - Not all exceptions map to signals… only those that the programmer can do something about
- Signals enable programs to respond to events (in user mode)
- If program does not handle a signal, default actions are supplied

Some Example Signals

Num	Name	Descr	Default Action
1	SIGHUP	Halt User Process	Terminate
2	SIGINT	Interrupt User Process (Ctrl-C)	Terminate
3	SIGQUIT	Quit User Process (Ctrl-/)	Terminate
9	SIGKILL	Kill User Process	Terminate
6	SIGABRT	Abort signal	Terminate/Dump
10	SIGUSR1	User Signal 1	Terminate
12	SIGUSR2	User Signal 2	Terminate
11	SIGSEGV	Segmentation Violation	Terminate/Dump
19	SIGSTOP	Stop processing (Ctrl-Z)	Stop until SIGCONT
18	SIGCONT	Continue Processing	Ignore/Continue
…	…	…	…

Sending Signals



- Exception events may cause signals to get sent
  - e.g. Segmentation violation causes SIGSEGV to get sent
- Signals can get sent by keyboard actions
  - e.g. Ctrl-C sends SIGINT to current executing processes
- Signals can get sent by programs via system call:

```
int kill(pid_t pid, int sig);
```

- The UNIX "kill" command is a wrapper around kill system call
  - e.g. `>kill –CONT 31023`

Receiving Signals

- Each signal has a default signal handler
- SIGSTOP and SIGKILL cannot be overridden
- All others: Specify a new signal handler to override default
  - You may specify “SIG_IGN” to ignore this signal
  - You may specify “SIG_DFL” to revert to the default signal handler
  - You may specify the name of your own signal handler routine
- Use the C library “signal“ function defined in signal.h to override
  - First argument is the signal number
  - Second argument is the signal handling function
    - Signal handling function takes a single int argument
    - Signal handling function returns “void”
    - May be SIG_IGN or SIG_DFL
  - Returns “SIG_ERR” (-1) if it fails

Coding a signal handler

- Function that takes one argument
  - The signal number of the signal sent to this process
  - Useful only when a the signal handler function handles multiple signals
- Handler may run *concurrently* with the original function
  - And can use the same global variables
  - This can cause problems!
- Returns void
- May exit (to abort)
- If signal handler returns, returns to instruction that was executing when the signal occurred
  - Possibly a signal handler for a different signal!

Blocking / Unblocking Signals

- To prevent endless loops, while a signal handler is processing a signal, that signal is automatically blocked
  - When signal handler returns, signal is unblocked
- It is also possible to explicitly block a signal using the C library `sigprocmask` function (and its helpers)
- A blocked signal is still sent, but cannot be received (handled)
  - Signal handler not invoked for that signal
- When the signal is unblocked it can be received
  - Signal handler can now be invoked