Exceptional Control Flow: Exceptions and Processes

15-213: Introduction to Computer Systems
14th Lecture, October 12th, 2017

Instructor:
Randy Bryant
Today

- Exceptional Control Flow
- Exceptions
- Processes
- Process Control
**Control Flow**

- **Processors do only one thing:**
  - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time
  - This sequence is the CPU’s *control flow* (or *flow of control*)

*Physical control flow*

```
<startup>
inst_1
inst_2
inst_3
...
inst_n
<shutdown>
```
Altering the Control Flow

- Up to now: two mechanisms for changing control flow:
  - Jumps and branches
  - Call and return
React to changes in program state

- Insufficient for a useful system:
  Difficult to react to changes in system state
  - Data arrives from a disk or a network adapter
  - Instruction divides by zero
  - User hits Ctrl-C at the keyboard
  - System timer expires

- System needs mechanisms for “exceptional control flow”
Exceptional Control Flow

- Exists at all levels of a computer system

- Low level mechanisms
  - 1. Exceptions
    - Change in control flow in response to a system event (i.e., change in system state)
    - Implemented using combination of hardware and OS software

- Higher level mechanisms
  - 2. Process context switch
    - Implemented by OS software and hardware timer
  - 3. Signals
    - Implemented by OS software
  - 4. Nonlocal jumps: `setjmp()` and `longjmp()`
    - Implemented by C runtime library
Today

- Exceptional Control Flow
- Exceptions
- Processes
- Process Control
Exceptions

- An exception is a transfer of control to the OS kernel in response to some event (i.e., change in processor state)
  - Kernel is the memory-resident part of the OS
  - Examples of events: Divide by 0, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C
Exception Tables

- Each type of event has a unique exception number $k$
- $k = \text{index into exception table (a.k.a. interrupt vector)}$
- Handler $k$ is called each time exception $k$ occurs
(partial) Taxonomy

- ECF
  - Asynchronous
    - Interrupts
  - Synchronous
    - Traps
    - Faults
    - Aborts
Asynchronous Exceptions (Interrupts)

- **Caused by events external to the processor**
  - Indicated by setting the processor’s *interrupt pin*
  - Handler returns to “next” instruction

- **Examples:**
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the kernel to take back control from user programs
  - I/O interrupt from external device
    - Hitting Ctrl-C at the keyboard
    - Arrival of a packet from a network
    - Arrival of data from a disk
Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - **Traps**
    - Intentional
    - Examples: *system calls*, breakpoint traps, special instructions
    - Returns control to “next” instruction
  - **Faults**
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - Either re-executes faulting (“current”) instruction or aborts
  - **Aborts**
    - Unintentional and unrecoverable
    - Examples: illegal instruction, parity error, machine check
    - Aborts current program
System Calls

- Each x86-64 system call has a unique ID number
- Examples:

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>read</td>
<td>Read file</td>
</tr>
<tr>
<td>1</td>
<td>write</td>
<td>Write file</td>
</tr>
<tr>
<td>2</td>
<td>open</td>
<td>Open file</td>
</tr>
<tr>
<td>3</td>
<td>close</td>
<td>Close file</td>
</tr>
<tr>
<td>4</td>
<td>stat</td>
<td>Get info about file</td>
</tr>
<tr>
<td>57</td>
<td>fork</td>
<td>Create process</td>
</tr>
<tr>
<td>59</td>
<td>execve</td>
<td>Execute a program</td>
</tr>
<tr>
<td>60</td>
<td>_exit</td>
<td>Terminate process</td>
</tr>
<tr>
<td>62</td>
<td>kill</td>
<td>Send signal to process</td>
</tr>
</tbody>
</table>
System Call Example: Opening File

- User calls: `open(filename, options)`
- Calls `__open` function, which invokes system call instruction `syscall`

```
000000000000e5d70 __open>
...
e5d79:  b8 02 00 00 00       mov  $0x2,%eax  # open is syscall #2
e5d7e:  0f 05               syscall  # Return value in %rax
e5d80:  48 3d 01 f0 ff ff    cmp   $0xfffffffffffff001,%rax
...
e5df9:  c3                   retq
```

- `%rax` contains syscall number
- Other arguments in `%rdi, %rsi, %rdx, %r10, %r8, %r9`
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`
System Call Example: Opening File

- User calls: `open(filename, options)`
- Calls `__open` function, which invokes system call instruction `syscall`

```
000000000000e5d70 <__open>:
...
e5d79: b8 02 00 00 00
     mov $0x2, %eax  # open is syscall #2
     ...
e5d80: 48 3d 01 f0 ff
     cmp $0xfffffffffffff001, %rax
     ...
e5dfa: c3
     retq
```

Almost like a function call
- Transfer of control
- On return, executes next instruction
- Passes arguments using calling convention
- Gets result in `%rax`

One Important exception!
- Executed by Kernel
- Different set of privileges
- And other differences:
  - E.g., “address” of “function” is in `%rax`
  - Uses `errno`
  - Etc.

- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`
Fault Example: Page Fault

- User writes to memory location
- That portion (page) of user’s memory is currently on disk

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

80483b7: c7 05 10 9d 04 08 0d movl $0xd,0x8049d10

User code

Kernel code

movl

Exception: page fault

Copy page from disk to memory

Return and reexecute movl
Fault Example: Invalid Memory Reference

```c
int a[1000];
main ()
{
    a[5000] = 13;
}
```

80483b7: `c7 05 60 e3 04 08 0d movl $0xd,0x804e360`

- Sends **SIGSEGV** signal to user process
- User process exits with “segmentation fault”
Today

- Exceptional Control Flow
- Exceptions
- Processes
- Process Control
Processes

- Definition: A process is an instance of a running program.
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”

- Process provides each program with two key abstractions:
  - Logical control flow
    - Each program seems to have exclusive use of the CPU
    - Provided by kernel mechanism called context switching
  - Private address space
    - Each program seems to have exclusive use of main memory.
    - Provided by kernel mechanism called virtual memory
Multiprocessing: The Illusion

- **Computer runs many processes simultaneously**
  - Applications for one or more users
    - Web browsers, email clients, editors, ...
  - Background tasks
    - Monitoring network & I/O devices
Multiprocessing Example

- Running program “top” on Mac
  - System has 123 processes, 5 of which are active
  - Identified by Process ID (PID)
Multiprocessing: The (Traditional) Reality

- Single processor executes multiple processes concurrently
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system (later in course)
  - Register values for nonexecuting processes saved in memory
Multiprocessing: The (Traditional) Reality

- Save current registers in memory
Multiprocessing: The (Traditional) Reality

- Schedule next process for execution
Multiprocessing: The (Traditional) Reality

- Load saved registers and switch address space (context switch)
Multiprocessing: The (Modern) Reality

- **Multicore processors**
  - Multiple CPUs on single chip
  - Share main memory (and some caches)
  - Each can execute a separate process
    - Scheduling of processors onto cores done by kernel
Concurrent Processes

- Each process is a logical control flow.
- Two processes run concurrently (are concurrent) if their flows overlap in time
- Otherwise, they are sequential
- Examples (running on single core):
  - Concurrent: A & B, A & C
  - Sequential: B & C
User View of Concurrent Processes

- Control flows for concurrent processes are physically disjoint in time

- However, we can think of concurrent processes as running in parallel with each other
Context Switching

- Processes are managed by a shared chunk of memory-resident OS code called the **kernel**
  - Important: the kernel is not a separate process, but rather runs as part of some existing process.

- Control flow passes from one process to another via a *context switch*
Today

- Exceptional Control Flow
- Exceptions
- Processes
- Process Control
System Call Error Handling

- On error, Linux system-level functions typically return -1 and set global variable `errno` to indicate cause.

- Hard and fast rule:
  - You must check the return status of every system-level function
  - Only exception is the handful of functions that return `void`

- Example:

```c
if ((pid = fork()) < 0) {
    fprintf(stderr, "fork error: %s\n", strerror(errno));
    exit(-1);
}
```
Error-reporting functions

- Can simplify somewhat using an *error-reporting function*:

```c
void unix_error(char *msg) /* Unix-style error */
{
    fprintf(stderr, "%s: %s\n", msg, strerror(errno));
    exit(-1);
}
```

```c
if ((pid = fork()) < 0)
    unix_error("fork error");
```

- But, must think about application. Not always appropriate to exit when something goes wrong.

Note: csapp.c exits with 0.
Error-handling Wrappers

- We simplify the code we present to you even further by using Stevens-style error-handling wrappers:

```c
pid_t Fork(void)
{
    pid_t pid;

    if ((pid = fork()) < 0)
        unix_error("Fork error");
    return pid;
}
```

- NOT what you generally want to do in a real application

```
pid = Fork();
```
Obtaining Process IDs

- **pid_t getpid(void)**
  - Returns PID of current process

- **pid_t getppid(void)**
  - Returns PID of parent process
Creating and Terminating Processes

From a programmer’s perspective, we can think of a process as being in one of three states

- **Running**
  - Process is either executing, or waiting to be executed and will eventually be *scheduled* (i.e., chosen to execute) by the kernel

- **Stopped**
  - Process execution is *suspended* and will not be scheduled until further notice (next lecture when we study signals)

- **Terminated**
  - Process is stopped permanently
Terminating Processes

- Process becomes terminated for one of three reasons:
  - Receiving a signal whose default action is to terminate (next lecture)
  - Returning from the `main` routine
  - Calling the `exit` function

- `void exit(int status)`
  - Terminates with an `exit status` of `status`
  - Convention: normal return status is 0, nonzero on error
  - Another way to explicitly set the exit status is to return an integer value from the main routine

- `exit` is called once but never returns.
Creating Processes

- *Parent process* creates a new running *child process* by calling `fork`

- `int fork(void)`
  - Returns 0 to the child process, child’s PID to parent process
  - Child is *almost* identical to parent:
    - Child get an identical (but separate) copy of the parent’s virtual address space.
    - Child gets identical copies of the parent’s open file descriptors
    - Child has a different PID than the parent

- `fork` is interesting (and often confusing) because it is called *once* but returns *twice*
Conceptual View of fork

- Make complete copy of execution state
  - Designate one as parent and one as child
  - Resume execution of parent or child
fork Example

```c
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {
        /* Child */
        printf("child: x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```

- Call once, return twice
- Concurrent execution
  - Can’t predict execution order of parent and child

```
linux> ./fork
parent: x=0
child: x=2

linux> ./fork
parent: x=0
child: x=2

linux> ./fork
parent: x=0
child: x=2
```

Making `fork` More Nondeterministic

- **Problem**
  - Linux scheduler does not create much run-to-run variance
  - Hides potential race conditions in nondeterministic programs
    - E.g., does `fork` return to child first, or to parent?

- **Solution**
  - Create custom version of library routine that inserts random delays along different branches
    - E.g., for parent and child in `fork`
  - Use runtime interpositioning to have program use special version of library code
/* fork wrapper function */

pid_t fork(void) {
    initialize();
    int parent_delay = choose_delay();
    int child_delay = choose_delay();
    pid_t parent_pid = getpid();
    pid_t child_pid_or_zero = real_fork();
    if (child_pid_or_zero > 0) {
        /* Parent */
        if (verbose) {
            printf("Fork. Child pid=%d, delay = %dms. Parent pid=%d, delay = %dms\n", 
                    child_pid_or_zero, child_delay, 
                    parent_pid, parent_delay);
            fflush(stdout);
        }
        ms_sleep(parent_delay);
    } else {
        /* Child */
        ms_sleep(child_delay);
    }
    return child_pid_or_zero;
}
forkx2 Example

```c
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child: x=%d\n", ++x);
        printf("child: x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    printf("parent: x=%d\n", --x);
    return 0;
}
```

- Call once, return twice
- Concurrent execution
  - Can’t predict execution order of parent and child
- Duplicate but separate address space
  - `x` has a value of 1 when fork returns in parent and child
  - Subsequent changes to `x` are independent
- Shared open files
  - `stdout` is the same in both parent and child

```
linux> ./fork2
parent: x=0
parent: x=-1
child : x=2
child : x=3
```
Modeling `fork` with Process Graphs

- A *process graph* is a useful tool for capturing the partial ordering of statements in a concurrent program:
  - Each vertex is the execution of a statement
  - \(a \rightarrow b\) means \(a\) happens before \(b\)
  - Edges can be labeled with current value of variables
  - `printf` vertices can be labeled with output
  - Each graph begins with a vertex with no inedges

- Any *topological sort* of the graph corresponds to a feasible total ordering.
  - Total ordering of vertices where all edges point from left to right
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}

fork.c
Interpreting Process Graphs

- **Original graph:**

  - Main:
    - x==1
    - for k

  - Child:
    - printf
    - exit
    - x=2

- **Relabeled graph:**

  - Nodes: a, b, c, d, e, f

- **Feasible total ordering:**

  - a → b → c → e → f → d

- **Infeasible total ordering:**

  - a → b → f → c → e → d
fork Example: Two consecutive forks

```c
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

Feasible output:
- L0
- L1
- Bye
- Bye
Feasible output:
- L0
- Bye
- L1
- Bye
- Bye

Infeasible output:
fork Example: Nested forks in parent

```c
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

Feasible output:
```
L0
L1
Bye
Bye
L2
Bye
```

Infeasible output:
```
L0
Bye
L1
Bye
L2
Bye
```
fork Example: Nested forks in children

```c
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

Feasible output:  
```
L0  Bye  L1  Bye  L2  Bye  Bye
```

Infeasible output:  
```
L0  Bye  L1  Bye  L2  Bye  Bye  L2
```
Quiz Time!

Check out:

https://canvas.cmu.edu/courses/1221
Reaping Child Processes

**Idea**

- When process terminates, it still consumes system resources
  - Examples: Exit status, various OS tables
  - Called a “zombie”
    - Living corpse, half alive and half dead

**Reaping**

- Performed by parent on terminated child (using `wait` or `waitpid`)
- Parent is given exit status information
- Kernel then deletes zombie child process

**What if parent doesn’t reap?**

- If any parent terminates without reaping a child, then the orphaned child will be reaped by `init` process (pid == 1)
- So, only need explicit reaping in long-running processes
  - e.g., shells and servers
Zombie Example

```c
void fork7() {
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

- `ps` shows child process as “defunct” (i.e., a zombie)
- Killing parent allows child to be reaped by `init`

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
   PID TTY          TIME CMD
 6585 tttyp9    00:00:00 tcsh
 6639 tttyp9    00:00:03 forks
 6640 tttyp9    00:00:00 forks <defunct>
 6641 tttyp9    00:00:00 ps
linux> kill 6639
[1] Terminated
linux> ps
   PID TTY          TIME CMD
 6585 tttyp9    00:00:00 tcsh
 6642 tttyp9    00:00:00 ps
```
Non-terminating Child Example

```c
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n", getpid());
        exit(0);
    }
}
```

```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
   PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
   PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```
**wait: Synchronizing with Children**

- Parent reaps a child by calling the `wait` function

```
int wait(int *child_status)
```

- **Suspends current process until one of its children terminates**

---

**Parent Process**

```
syscall ...
```

**Kernel code**

```
Exception
```

And, potentially other user processes, including a child of parent
**wait: Synchronizing with Children**

- Parent reaps a child by calling the `wait` function

```c
int wait(int *child_status)

- Suspends current process until one of its children terminates
- Return value is the `pid` of the child process that terminated
- If `child_status != NULL`, then the integer it points to will be set to a value that indicates reason the child terminated and the exit status:
  - Checked using macros defined in `wait.h`
    - `WIFEXITED`, `WEXITSTATUS`, `WIFSIGNALED`, `WTERMSIG`, `WIFSTOPPED`, `WSTOPSIG`, `WIFCONTINUED`
  - See textbook for details
```
**wait: Synchronizing with Children**

```c
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
        exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
```

**Feasible output(s):**
- HC
- HP
- HP
- HC
- CT
- CT
- Bye
- Bye

**Infeasible output:**
- HP
- CT
- Bye
- HC
Another wait Example

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```c
void fork10() {
    pid_t pid[N];
    int i, child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            exit(100+i); /* Child */
        }
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
forks.c
```
waitpid: Waiting for a Specific Process

- **pid_t waitpid(pid_t pid, int *status, int options)**
  - Suspends current process until specific process terminates
  - Various options (see textbook)

```c
void fork11() {
    pid_t pid[N];
    int i;
    int child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = N-1; i >= 0; i--)
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
}
```

forks.c
execve: Loading and Running Programs

- int execve(char *filename, char *argv[], char *envp[])

- Loads and runs in the current process:
  - Executable file `filename`
    - Can be object file or script file beginning with `#!interpreter` (e.g., `#!/bin/bash`)
  - ...with argument list `argv`
    - By convention `argv[0] == filename`
  - ...and environment variable list `envp`
    - “name=value” strings (e.g., `USER=droh`)
    - `getenv`, `putenv`, `printenv`

- Overwrites code, data, and stack
  - Retains PID, open files and signal context

- Called **once and never returns**
  - ...except if there is an error
Structure of the stack when a new program starts

Null-terminated environment variable strings

Null-terminated command-line arg strings

envp[n] == NULL
envp[n-1]
...
envp[0]
argv[argc] = NULL
argv[argc-1]
...
argv[0]

Stack frame for libc_start_main

Future stack frame for main

argv (in %rsi)

argc (in %rdi)

environ (global var)

envp (in %rdx)

Bottom of stack

Top of stack
execve Example

- **Execute** "/bin/ls -lt /usr/include" in child process using current environment:

  - `envp[n] = NULL`
  - `envp[n-1]`
  - `...`
  - `envp[0]`

  - `"PWD=/usr/droh"
  - `"USER=droh"

- `environ`

- `myargv[argc] = NULL`
  - `myargv[2]`
  - `myargv[1]`
  - `myargv[0]`

  - `(argc == 3)`
  - `/usr/include`
  - `"-lt"
  - `/bin/ls`

```c
if ((pid = Fork()) == 0) {  /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: Command not found.\n", myargv[0]);
        exit(1);
    }
}
```
Summary

- **Exceptions**
  - Events that require nonstandard control flow
  - Generated externally (interrupts) or internally (traps and faults)

- **Processes**
  - At any given time, system has multiple active processes
  - Only one can execute at a time on any single core
  - Each process appears to have total control of processor + private memory space
Summary (cont.)

- **Spawning processes**
  - Call `fork`
  - One call, two returns

- **Process completion**
  - Call `exit`
  - One call, no return

- **Reaping and waiting for processes**
  - Call `wait` or `waitpid`

- **Loading and running programs**
  - Call `execve` (or variant)
  - One call, (normally) no return