Chapter 8. Exceptional Control Flow

Lynn Choi
Korea University
Computers 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 system’s physical control flow (or flow of control).

Physical control flow

<startup>
inst_1
inst_2
inst_3
...
inst_n
<shutdown>

PEARSON All rights reserved
Up to Now: two mechanisms for changing control flow:
- Jumps and branches
- Call and return using the stack discipline.
- Both react to changes in program state.

Insufficient for a useful system
- System should react to changes in system state
  - Data arrives from a disk or a network adapter.
  - Instruction divides by zero
  - User hits ctl-alt-delete at the keyboard
  - System timer expires
  - Parent process that creates child processes must be notified when their children terminate

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

- Mechanisms for exceptional control flow exists at all levels of a computer system.

**At the hardware level**
- Exceptions: caused by an internal event created by the currently running process
- Interrupts
  - Caused by an external event asynchronously
- Implemented by combination of hardware and OS software

**At the operating system level**
- Process context switch
- System calls
  - Provide applications with entry points into the operating system

**At the application level**
- Signal
  - A process can send a signal to another process that abruptly transfers control to a signal handler
- Nonlocal jumps (setjmp/longjmp)
  - React to errors, which sidesteps the usual stack discipline and make nonlocal jumps to arbitrary locations (implemented by C language runtime library)
An *exception* is a transfer of control to the OS in response to some *event* (i.e., change in processor state)

- Implemented partly by the hardware and partly by the operating system
- Event might be related to the execution of current instruction (exception)
- Event might be unrelated to the execution of current instruction (interrupt)
- On event, the processor makes an indirect procedure call (to the *exception handler*) through a jump table called *exception table*
(Synchronous) Exceptions

Caused by events that occur as a result of executing an instruction:

- **Traps**
  - *Intentional* exceptions
  - Examples: system calls, breakpoints, special instructions
    - Executing the `syscall` instruction causes a trap to an exception handler that decodes the argument and calls the appropriate kernel routine (run in *kernel mode*)
    - Returns control to “next” instruction

- **Faults**
  - *Unintentional* but possibly recoverable
  - Examples: page faults (recoverable), protection faults (unrecoverable).
  - Either re-executes faulting (“current”) instruction or terminate the process

- **Aborts**
  - Unintentional and *unrecoverable fatal* errors
  - Examples: parity error, machine check.
  - Aborts the current process, and probably the entire system
Interrupt (Asynchronous Exception)

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

**Examples:**
- I/O interrupts
  - Hitting ctl-c at the keyboard, arrival of a packet from a network, arrival of a data sector from a disk
- Hard reset interrupt: hitting the reset button
- Soft reset interrupt: hitting ctl-alt-delete on a PC

(1) Interrupt pin goes high during execution of current instruction
(2) Control passes to handler after current instruction finishes
(3) Interrupt handler runs
(4) Handler returns to next instruction
(External) Interrupt

**Interrupt Classification**

- **Maskable interrupt**
  - Can be disabled/enabled by an instruction
  - Generated by asserting INTR pin
  - External interrupt controllers
    - Intel 8259 PIC (programmable interrupt controller) delivers the interrupt vectors on the system bus during interrupt acknowledge cycle

- **Non-maskable interrupt (NMI)**
  - Cannot be disabled by program
  - Received on the processor’s NMI# input pin
Exception Handling Procedure

**Exception handling procedure**

- Flush all the instructions fetched subsequent to the instruction causing the condition from the pipeline
- Drain the pipeline
  - Complete all outstanding write operations prior to the faulting instruction
- Save the PC of the next instruction to execute
- Also need to save the necessary registers and stack pointers to allow it to restore itself to its state
- Vector the interrupt
- Fetch instructions from the ISR and service the interrupt
- Return from the interrupt
Exception vs. Procedure Call

- **Differences**
  - As with procedure call, CPU pushes return address on the stack before jumping to the handler. But, the return address is *either the current instruction or the next instruction* depending on the event.
  - CPU *pushes some additional processor state* onto the stack such as EEFLAGS register that contains condition codes.
  - All of these states are *pushed onto the kernel’s stack* rather than on the user’s stack.
  - Exception handlers run in *kernel mode*.
  - After the handler processes the event, the handler (optionally) returns to the interrupted program by executing a special “*return from interrupt*” instruction, which pops states back into the registers (restoring user states).
An exception table is a jump table where entry k contains the address of the handler for exception k.

- Start address of the exception table is contained in a special CPU register called the exception table base register.
- Each type of event has a unique exception number k.
- Using the exception number as an index, you can fetch the address of the corresponding handler.
- The index of the jump table (or the corresponding entry in the jump table) is called interrupt vector.
- Handler k is called each time exception k occurs.
Opening a File

- User calls `open(filename, options)`

```
0804d070 <__libc_open>:
  . . .
  804d082:   cd 80    int    $0x80
  804d084:   5b       pop     %ebx
  . . .
```

- OS must find or create file, get it ready for reading or writing
- Returns integer file descriptor
Fault Example #1

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

- Page handler must load page into physical memory
- Returns to faulting instruction
- Successful on second try

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

User Process  OS

![Diagram of memory reference and page fault process]
Fault Example #2

- **Memory Reference**
  - User writes to memory location
  - Address is not valid

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

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

- Page handler detects invalid address
- Sends SIGSEGV signal to user process
- User process exits with “segmentation fault”
Exceptions in Intel Processors

- **Pentium system can have up to 256 different exception types**
  - The first 32 exceptions (exception 0-31) are defined by the Pentium architecture and identical to any Pentium-class system.
  - Exceptions 32-255 correspond to traps and interrupts defined by OS

- **Examples**
  - Exception 0  Divide by 0  Fault
    - Unix does not attempt to recover from divide errors, opting instead to *abort program*. Unix shell report divide errors as “floating point exceptions”.
  - Exception 13  General protection fault  Fault
    - Program references an undefined area of virtual memory, or write into read-only segment. Unix shell report as “*segmentation fault*”.
  - Exception 14  Page fault  Fault
  - Exception 18  Machine check  Abort
  - 32-127  OS-defined exceptions  Interrupt or trap
  - 128 (0x80)  System call  Trap
    - System calls are provided via a trapping instruction called **INT n**, where n can be the index of any of the 256 entries in the exception table (software interrupt).
  - 129-255  OS-defined exceptions  Interrupt or trap
Def: A process is an instance of a program in execution.

- 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.
- Private address space
  - Each program seems to have exclusive use of main memory.

How are these Illusions maintained?

- Multitasking: process executions are interleaved
  - In reality, many other programs are running on the system.
  - Processes take turns in using the processor
  - Each time period that a process executes a portion of its flow is called a time slice
- Virtual memory: memory system provides a private space for each process
  - The private space is also called the virtual address space, which is a linear array of bytes, addressed by n bit virtual address (0, 1, 2, 3, … 2^n-1)
Each process has its own private address space.

- **Kernel virtual memory** (code, data, heap, stack)
- **Memory mapped region for shared libraries**
- **Run-time heap** (created at runtime)
- **User stack** (created at runtime)
- **Unused memory**
- **Read/write segment** (.data, .bss)
- **Read-only segment** (.init, .text, .rodata)
- **BRK**: 0xc0000000
- **%esp (stack pointer)**
- **Memory invisible to user code**

Pearson All rights reserved
Each process has its own logical control flow
Concurrent Processes

Concurrent processes

- Two processes run concurrently (are concurrent) if their flows overlap in time.
- Otherwise, they are sequential.

Examples:
- Concurrent: A & B, A & C
- Sequential: B & C

Control flows for concurrent processes are *physically disjoint* in time.
- However, we can think of concurrent processes as *logically running in parallel* with each other.
Processes are managed by a shared chunk of OS code called the **kernel**

- *Important*: the kernel is not a separate process, but rather runs as part of some user process
  - Processors typically provide this capability with a mode bit in some control register

**User mode and kernel mode**

- If the mode bit is set, the process is running in *kernel mode* (*supervisor mode*), and can execute any instruction and can access any memory location
- If the mode bit is not set, the process is running in *user mode* and is not allowed to execute *privileged instructions*
  - A process running application code is initially in user mode
  - The only way to change from user mode to kernel mode is via an exception and exception handler runs in kernel mode
**Context Switching**

**Context**
- The kernel maintains a *context* for each process
  - The context is the state of a process that the kernel needs to restart a preempted process
  - Consist of general purpose registers, FP registers, PC, user’s stack, status registers, kernel’s stack, and various kernel data structures such as page table and file table

**Context switching**
- The OS kernel implements multitasking using an exceptional control flow
- At certain points during the execution of a process, the kernel decide to preempt the current process and restart a previously preempted process
  - This is called scheduling and handled by code in the kernel called scheduler
- Context switching
  - The kernel first saves the context of the current process
  - The kernel restores the context of some previously preempted process
  - Then, the kernel passes control to this newly restored process
Context Switching

Time
- read
- disk interrupt
- return from read

Process A
- code
- user code
- kernel code

Process B
- code
- user code
- kernel code

context switch

PEARSON All rights reserved
**Process control**

- Unix provides a number of system calls for manipulating processes from C program
  - Obtain Process ID, Create/Terminate Process, etc.

**int fork(void)**

- Creates a new process (child process) that is identical to the calling process (parent process)
- Returns 0 to the child process
- Returns child’s `pid` to the parent process

```c
if (fork() == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

Fork is interesting (and often confusing) because it is called *once* but returns *twice*
Fork Example #1

- **Parent and child both run the same code**
  - Distinguish parent from child by return value from `fork`

- **Duplicate but separate address space**
  - Start with same state, but each has private copy
  - Relative ordering of their print statements undefined

- **Shared files**
  - Both parent and child print their output on the same screen

```c
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```
Fork Example #2

- **Both parent and child can continue forking**
- **Process graph**
  - Each horizontal arrow corresponds to a process
  - Each vertical arrow corresponds to the execution of a `fork` function

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

PEARSON All rights reserved
Fork Example #3

**Key Points**
- Both parent and child can continue forking

```c
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```

PEARSON All rights reserved
**Key Points**

- Both parent and child can continue forking

```c
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```
Fork Example #5

**Key Points**
- Both parent and child can continue forking

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

PEARSON All rights reserved
exit: Destroying Process

void exit(int status)

- Terminate a process with an exit status
  - Normally with status 0
- atexit() registers functions to be executed upon exit

```c
void cleanup(void) {
    printf("cleaning up\n");
}

void fork6() {
    atexit(cleanup);
    fork();
    exit(0);
}
```

A process can be terminated for one of three reasons

- Call the exit function
- Return from the main function
- Receive a signal whose default action is to terminate the process
Reaping Child Process

- **Zombie**
  - When a process terminates, the kernel does not remove it from the system immediately.
  - The process is kept around in a terminated state until it is reaped by its parent.
  - A terminated process that has not yet been reaped is called a zombie.
    - Living corpse, half alive and half dead.
    - Zombie still consumes system resources such as various tables maintained by OS.

- **Reaping**
  - Performed by the parent on a terminated child.
  - Parent is given an exit status information.
  - Kernel discards process.

- **What if Parent Doesn’t Reap?**
  - If any parent terminates without reaping a child, then the child will be reaped by the **init** process.
    - The init process has a PID of 1 and is created by the kernel during system initialization.
  - Long-running programs such as shells or servers should always reap their zombie children.
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"
- Killing parent allows child to be reaped
Nonterminating Child Example

Child process still active even though parent has terminated
Must kill explicitly, or else will keep running indefinitely

```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);
    }
}
```
**wait: Synchronizing with Children**

```c
int wait(int *child_status)
{
    // Suspends the current process until one of its children terminates
    // The terminated child is removed from the system.
    // Return value is the pid of the child process that terminated
    // if child_status != NULL, then it will be set to a status indicating why the child process terminated

    void fork9() {
        int child_status;

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

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status
  - WIFEEXITED returns true if the child terminated normally

```c
void fork10()
{
    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 = 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);
    }
}
```
`waitpid(pid, &status, options)`

- Can wait for specific process
- Various options

```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 = 0; i < N; 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 terminated abnormally\n", wpid);
    }
}```
## Wait/Waitpid Example Outputs

### Using `wait` (fork10)

<table>
<thead>
<tr>
<th>Child ID</th>
<th>Exit Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>3565</td>
<td>103</td>
</tr>
<tr>
<td>3564</td>
<td>102</td>
</tr>
<tr>
<td>3563</td>
<td>101</td>
</tr>
<tr>
<td>3562</td>
<td>100</td>
</tr>
<tr>
<td>3566</td>
<td>104</td>
</tr>
</tbody>
</table>

### Using `waitpid` (fork11)

<table>
<thead>
<tr>
<th>Child ID</th>
<th>Exit Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>3568</td>
<td>100</td>
</tr>
<tr>
<td>3569</td>
<td>101</td>
</tr>
<tr>
<td>3570</td>
<td>102</td>
</tr>
<tr>
<td>3571</td>
<td>103</td>
</tr>
<tr>
<td>3572</td>
<td>104</td>
</tr>
</tbody>
</table>
exec: Running new programs

**int execl(char *path, char *arg0, char *arg1, ..., 0)**

- Load and run executable at path with args arg0, arg1, ...
  - path is the complete path of an executable
  - arg0 becomes the name of the process
    - typically arg0 is either identical to path, or else it contains only the executable filename from path
  - “real” arguments to the executable start with arg1, etc.
  - list of args is terminated by a (char *)0 argument
- returns -1 if error, otherwise doesn’t return!
  - Calls once but never returns

```c
main() {
    if (fork() == 0) {
        execl("/usr/bin/cp", "cp", "foo", "bar", 0);
    }
    wait(NULL);
    printf("copy completed\n");
    exit();
}
```

PEARSON All rights reserved
int execve(char *filename, char *argv[], char *envp[]);

- Load and run the executable with argument list argv and the environment variable list envp
- After execve loads the filename, it calls the startup code, which sets up the stack and passes control to the main routine
  - int main(int argc, char *argv[], char *envp[]);
- When main begins executing, the user stack has the organization shown on the right side

**Dynamic linker variables**
- argv[argc] = NULL
- argv[argc-1]
- ...
- argv[0]

- envp[n-1]
- ...
- envp[0]

**Null-terminated command-line arg strings**
- envp[n] == NULL

**Null-terminated environment variable strings**

**Stack frame for main**
- argc
- argv
- envp
Environment Variables

**Environment variables**

- A set of dynamic named values that can affect the way running processes will behave on a computer.
  - Create the *operating environment* in which a process runs.
- All Unix operating system flavors, MS-DOS, and Microsoft Windows have environment variables; however, they do not all use the same variable names.

**Examples of environment variables**

- PATH - lists directories the shell searches for the commands
- HOME - indicate where a user's home directory is located in the file system
- TERM - specifies the type of computer terminal (e.g., vt100).
- PS1 - specifies how the prompt is displayed
- MAIL - used to indicate where a user's mail is to be found.
- TEMP - location where processes can store temporary files
The World of Multitasking

- **System Runs Many Processes Concurrently**
  - Process: executing program
    - State consists of memory image + register values + program counter
  - Continually switches from one process to another
    - Suspend process when it needs I/O resource or timer event occurs
    - Resume process when it is given scheduling priority
  - Appears to user(s) as if all processes executing simultaneously
    - Even though most systems can only execute one process at a time
    - Except possibly with lower performance than if running alone
Programmer’s Model of Multitasking

**Basic Functions**
- `fork()` spawns new process
  - Called once, returns twice
- `exit()` terminates own process
  - Called once, never returns
  - Puts it into “zombie” status
- `wait()` and `waitpid()` wait for and reap terminated children
- `execl()` and `execve()` run a new program in an existing process
  - Called once, (normally) never returns

**Programming Challenge**
- Understanding the nonstandard semantics of the functions
- Avoiding improper use of system resources
  - E.g. “Fork bombs” can disable a system.
Unix Process Hierarchy

[0]

init [1]

Daemon 
 e.g. httpd

Login shell

Child

Child

Child

Grandchild

Grandchild
1. Pushing reset button loads the PC with the address of a small bootstrap program.
2. Bootstrap program loads the boot block (disk block 0).
3. Boot block program loads kernel binary (e.g., /boot/vmlinux)
4. Boot block program passes control to kernel.
5. Kernel handcrafts the data structures for process 0.

**Diagram:**

- **[0]**: Process 0: handcrafted kernel process
- **init [1]**: Process 0 forks child process 1
- **Child process 1 execs /sbin/init**
Unix Startup: Step 2

init [1]

/etc/inittab

Daemons
  e.g. ftpd, httpd

[0]

init forks and execs daemons per /etc/inittab, and forks and execs a getty program for the console
Unix Startup: Step 3

The `getty` process execs a login program.
Unix Startup: Step 4

login reads login and passwd. if OK, it execs a *shell*. if not OK, it execs another *getty*.