Exceptional Control Flow
Part I

Dr. Steve Goddard
goddard@cse.unl.edu

http://cse.unl.edu/~goddard/Courses/CSCE230J

Giving credit where credit is due

- Most of slides for this lecture are based on slides created by Drs. Bryant and O’Hallaron, Carnegie Mellon University.
- I have modified them and added new slides.

Topics

- Exceptions
- Process context switches
- Creating and destroying processes

Control Flow

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).

Exchanging the Control Flow

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

- Difficult for the CPU 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

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

Low level Mechanism

- exceptions
  - change in control flow in response to a system event (i.e., change in system state)

Higher Level Mechanisms

- Process context switch
- Signals
- Nonlocal jumps (setjmp/longjmp)
- Implemented by either:
  - OS software (context switch and signals).
  - C language runtime library: nonlocal jumps.
System context for exceptions

Exceptions

An exception is a transfer of control to the OS in response to some event (i.e., change in processor state)

Interrupt Vectors

Asynchronous Exceptions (Interrupts)

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 ctrl-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 ctrl-alt-delete on a PC

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)
  - Either re-executes faulting (“current”) instruction or aborts.
- Aborts
  - unintentional and unrecoverable
  - Examples: parity error, machine check.
  - Aborts current program

Trap Example

Opening a File
- User calls open(filename, options)
  &0040570: __libc_open>

- Function open executes system call instruction int
- OS must find or create file, get it ready for reading or writing
- Returns integer file descriptor

User Process

OS

Int pop return
Fault Example #1

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

User Process

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

OS

Fault Example #2

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

User Process

- Page handler detects invalid address
- Sends SIGSEGV signal to user process
- User process exits with "segmentation fault"

OS

Processes
Def: 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.
  - Each program seems to have exclusive use of main memory.

How are these illusions maintained?
- Process executions interleaved (multitasking)
- Address spaces managed by virtual memory system

Logical Control Flows
Each process has its own logical control flow

User View of 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

User View of Concurrent Processes

Control flows for concurrent processes are physically disjoint in time.
However, we can think of concurrent processes are running in parallel with each other.
Context Switching

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.

Control flow passes from one process to another via a context switch.

```
+-----------------+              +-----------------+
| Process A code  |              | Process B code  |
+-----------------+              +-----------------+
    user code     |              | user code       |
    kernel code   | context switch| kernel code     |
    user code     |              | user code       |
```

Private Address Spaces

Each process has its own private address space.

```
+-----------------+              +-----------------+
| kernel virtual memory |              | memory mapped to user code |
| (code, data, heap, stack) |              |                           |
+-----------------+              +-----------------+
    user reads (managed by kernel) |
    read/write segment (data, text) |
    read-only segment (init, text, rodata) |
    unshared |
```

fork: Creating new processes

```
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

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

Fork Example #1

```
Key Points
- Parent and child both run same code
- Distinguish parent from child by return value from fork
- Start with same state, but each has private copy
- Including shared output file descriptor
- Relative ordering of their print statements undefined

void fork1()

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

Fork Example #2

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

void fork2()

- printf("L0\n");
- fork();
- printf("L1\n");
- fork();
- printf("L2\n");
```

Fork Example #3

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

void fork3()

- printf("L0\n");
- fork();
- printf("L1\n");
- fork();
- printf("L2\n");
- fork();
- printf("Bye\n");
```

Page 4
Fork Example #4

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");
}
```

Zombies

Idea
- When process terminates, still consumes system resources
- Called a “zombie”
- Living corpse, half alive and half dead

Reaping
- Performed by parent on terminated child
- Parent is given exit status information
- Kernel discards process

What if Parent Doesn’t Reap?
- If any parent terminates without reaping a child, then child will be reaped by init process
- Only need explicit reaping for long-running processes
  - E.g., shells and servers

Nonterminating Child Example

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

```
Zombie Example

<table>
<thead>
<tr>
<th>Process</th>
<th>PID</th>
<th>TTY</th>
<th>TIME CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>6585</td>
<td>tty9</td>
<td>00:00:00 tcs</td>
<td></td>
</tr>
<tr>
<td>6439</td>
<td>tty9</td>
<td>00:00:00 forks</td>
<td></td>
</tr>
<tr>
<td>6440</td>
<td>tty9</td>
<td>00:00:00 forks &lt;defunct&gt;</td>
<td></td>
</tr>
<tr>
<td>6441</td>
<td>tty9</td>
<td>00:00:00 ps</td>
<td></td>
</tr>
</tbody>
</table>

- ps shows child process as “defunct”
- Killing parent allows child to be reaped
void forkl0() {
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */

    pid_t wpid = wait(child_status);
    if (WEXITED(child_status))
        printf("Child %d terminated with exit status \n", wpid, WEXITSTATUS(child_status));
    else
        printf("Child %d terminate abnormally\n", wpid);
}

void forkl1() {
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */

    pid_t wpid = wait(pid[i], &child_status, 0);
    if (WEXITED(child_status))
        printf("Child %d terminated with exit status \n", wpid, WEXITSTATUS(child_status));
    else
        printf("Child %d terminate abnormally\n", wpid);
}

int exec(char *path, char *argv, char *arg1, … , 0)
    loads and runs executable at path with args argv, arg1, …
    path is the complete path of an executable
    argv becomes the name of the process
    typically argv is either identical to path, or else it contains only
    the executable filename from path
    "\n" arguments to the executable start with argv, etc.
    list of args is terminated by a (\n) argument
    returns -1 if error, otherwise doesn’t return!

int main() {
    if (fork() == 0) {
        exec("/usr/bin/cp", "cp", "foo", "bar", 0);
        exit(NULL);
        printf("copy completed\n");
    }
}
Summarizing

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, though
- Each process appears to have total control of processor + private memory space

Summarizing (cont.)

Spawning Processes
- Call to fork
  - One call, two returns

Terminating Processes
- Call exit
  - One call, no return

Reaping Processes
- Call wait or waitpid

Replacing Program Executed by Process
- Call exec1 (or variant)
  - One call, (normally) no return