Exceptional Control Flow

Part I

Dr. Steve Goddard
goddard@cse.unl.edu

http://cse.unl.edu/~goddard/Courses/JDEP284

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).

Physical control flow

<startup>
inst1
inst2
inst3
...
instn
<shutdown>

Time

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)
- Combination of hardware and OS software

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.

Altering 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 ct+c at the keyboard
  - System timer expires

System needs mechanisms for “exceptional control flow”
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
Each type of event has a unique exception number k
- Index into jump table (a.k.a., interrupt vector)
- Jump table entry k points to a function (exception handler).
- Handler k is called each time exception k occurs.

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)
- 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 Open file
Fault Example #1

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

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

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

Fault Example #2

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

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

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

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.
- Private address space

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

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.

**Private Address Spaces**

Each process has its own private address space.

- Kernel virtual memory (code, data, heap, stack):
  - Memory invisible to user code
  - Heap (stack pointer)
- Memory-mapped region for shared libraries:
  - Run-time heap (managed by malloc)
  - Read-only segment (e.g., init)
  - Read-only segment (e.g., text, runtime)
- User stack (created at runtime)
- Unmapped

**fork: Creating new processes**

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

**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 `printf` statements undefined

**Fork Example #1**

```c
void fork1()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

**Key Points**

- Both parent and child can continue forking

**Fork Example #2**

```c
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```

**Key Points**

- Both parent and child can continue forking

**Fork Example #3**

```c
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```
**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
- Various tables maintained by OS
- 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

**Zombie Example**

```
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 */
}
```

**Nonterminating Child Example**

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

**Exit: Destroying Process**

```c
void exit(int status)
{
    exit a process
    Normally return with status 0
    atexit() registers functions to be executed upon exit
}
```

```
void cleanup(void)
{
    print("Cleaning up\n");
}
```

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

**Linux Exit Example**

```
$ ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
$ ps
PID TTY          TIME CMD
6595 ttyp9    00:00:00 tcsh
6639 ttyp9    00:00:03 forks
6639 ttyp9    00:00:03 forks
6640 ttyp9    00:00:00 forks <defunct>
6641 ttyp9    00:00:00 ps
$ kill 6639
[1]    Terminated
$ ps
PID TTY          TIME CMD
6595 ttyp9    00:00:00 tcsh
6678 ttyp9    00:00:00 ps
$ ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
$ ps
PID TTY          TIME CMD
6595 ttyp9    00:00:00 tcsh
6676 ttyp9    00:00:06 forks
6677 ttyp9    00:00:00 ps
$ kill 6676
$ ps
PID TTY          TIME CMD
6595 ttyp9    00:00:00 tcsh
6678 ttyp9    00:00:00 ps
$```
**wait: Synchronizing with children**

```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` is NULL, then the object points to will be set to a status indicating why the child process terminated.

```c
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.

```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);
  }
}
```

**Wait/Waitpid Example Outputs**

**Using wait (fork10)**

- Child 3568 terminated with exit status 100
- Child 3569 terminated with exit status 101
- Child 3564 terminated with exit status 104

**Using waitpid (fork11)**

- Child 3568 terminated with exit status 100
- Child 3569 terminated with exit status 101
- Child 3570 terminated with exit status 102
- Child 3571 terminated with exit status 103
- Child 3572 terminated with exit status 104

**exec: Running new programs**

```c
int exec(char *path, char **args, char *argl, .., 0)
```

- Loads and runs 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 argl, etc.
- List of args is terminated by a (char *)0 argument.
- Returns -1 if error, otherwise doesn’t return.

```c
main() {
  if (fork() == 0) {
    exec("/usr/bin/cp", "/etc/passwd", "cp", "/home/lbn", "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