15-213
"The course that gives
Exceptional Control Flow
Part I
October 14, 2003

Topics
- Exceptions
- Process context switches
- Creating and destroying processes

control flow

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 ctrl-c at the keyboard
  - System timer expires

System needs mechanisms for “exceptional control flow”

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>
inst_1
inst_2
inst_3
...
inst_n
<shutdown>

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.
System context for exceptions

USB Ports
- Keyboard
- Mouse
- Modem
- Printer
- Interrupt controller
- Timer
- Serial port controllers
- Parallel port controller
- Local I/O Bus
- IDE disk controller
- SCSI controller
- Video adapter
- Network adapter
- Disk
- Disk
- CDROM
- Display
- Network

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

Precise vs. Imprecise Faults

- Precise Faults: the exception handler knows exactly which instruction caused the fault. All prior instructions have completed and no subsequent instructions had any effect.
- Imprecise Faults: the CPU was working on multiple instructions concurrently and an ambiguity may exist as to which instruction caused the fault. For example, multiple FP instructions were in the pipe and one caused an exception (Alpha Microprocessors).

Trap Example

Opening a File

- User calls open(filename, options)
- User Process
  - int pop
  - exception
  - Open file
- OS
  - 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

User Process

- User Process
  - int a[500];
  - main ()
  - {
    a[500] = 13;
  }
  - movl $0edx, $esi
  - stosd
  - Page fault
  - Create page and load into memory
  - OS
  - OS
  - exception
  - Return
Fault Example #2

Memory Reference with TLB miss
- User writes to memory location
- That portion (page) of user's memory is currently in physical memory, but the processor has forgotten how to translate the virtual address to the physical address
- TLB must be reloaded with current translation
- Returns to faulting instruction
- Successful on second try

User Process OS or Hardware

Fault Example #3

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

User Process OS

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.
- Private address space
  - 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
**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

![Diagram of concurrent processes]

**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.

![Diagram of user view of concurrent processes]

---

**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**.

![Diagram of context switching]

---

**Private Address Spaces**

Each process has its own private address space.

![Diagram of private address spaces]

---

---

---

---
Virtual Machines

All current general purpose computers support multiple, concurrent user-level processes. Is it possible to run multiple kernels on the same machine?

- Yes: Virtual Machines (VM) were supported by IBM mainframes for over 30 years
- Intel’s IA32 instruction set architecture is not virtualizable (neither are the Sparc, Mips, and PPC ISAs)
- With a lot of clever hacking, Vmware™ managed to virtualize the IA32 ISA in software
- User Mode Linux

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 p.id to the parent process

    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

**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

```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

**Key Points**

- Both parent and child can continue forking

```c
void fork2()
{
    printf("\n\n");
    fork();
    printf("\n\n");
    fork();
    printf("\n\n");
}
```
**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");
}
```

**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");
    }
}
```

**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");
}
```

**exit: Destroying Process**

**void exit(int status)**
- exits a process
- Normally return with status 0
- `atexit()` registers functions to be executed upon exit

```c
void exit(int status)
{
    printf("closing up\n");
}
```

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

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

```c
void fork() {
    if (fork() == 0) {
        printf("running child, pid = %d\n", getpid());
        sleep(200000);
        exit(0);
    } else {
        printf("running parent, pid = %d\n", getppid());
        while (1) {
            sleep(10000000);
        }
    }
}
```

```sh
class=Linux
command=ps
args=TIME CMD 6585 ttys0 00:00:00 ssh
6639 ttys0 00:00:03 forks
6640 ttys0 00:00:00 forks <defunct>
6641 ttys0 00:00:00 ps
```

- ps shows child process as "defunct"
- Killing parent allows child to be reaped

Nonterminating Child Example

```sh
class=Linux
command=ps
args=TIME CMD 6576
```

- Child process still active even though parent has terminated

```sh
class=Linux
command=kill
args=6576
```

- Must kill explicitly, or else will keep running indefinitely

wait: Synchronizing with children

```c
int wait(int *child_status) {
    suspend current process until one of its children terminates
    return value is the pid of the child process that terminated
    if child_status != NULL, then object it points to will be set to a status indicating why the child process terminated
}
```

- Child process still active even though parent has terminated

- Must kill explicitly, or else will keep running indefinitely
### wait: Synchronizing with children

```c
void fork2() {
    int child_status;
    if (fork() == 0) {
        printf("WC: hello from child\n");
    } else {
        printf("BP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
        exit();
    }
}
```

### Wait() Example

- If multiple children completed, will take in arbitrary order
- Can use macros WEXITED 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 = waitpid(pid[i], &child_status, 0);
                if (WEXITED(child_status)) {
                    printf("Child %d terminated with exit status %d\n", wpid, WEXITStatus(child_status));
                } else {
                    printf("Child %d terminated abnormally\n", wpid);
                }
            }
        }
    }
}
```

### Waitpid() Example Outputs

**Using wait (fork10)**
- Child 3565 terminated with exit status 103
- Child 3564 terminated with exit status 102
- Child 3563 terminated with exit status 101
- Child 3562 terminated with exit status 100
- Child 3566 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

---

**Page 9**
exec: Running new programs

```c
int exec1(char *path, char *arg0, char *arg1, ..., 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 arg1, etc.
- list of args is terminated by a (char *) 0 argument
- returns -1 if error, otherwise doesn't return!
```

main()
{
  if (fork() == 0) {
    exec1("/usr/bin/er", "cp", "foo", "bar", 0);
    wait(NULL);
    printf("copy completed\n");
    exit();
  }
}

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 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