15-213
“The course that gives CMU its Zip!”

Exceptional Control Flow I
Oct 18, 2001

Topics
• Exceptions
• Process context switches
• Creating and destroying processes
Control flow

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).
Altering the Control Flow

We’ve discussed two mechanisms for changing the 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)
• Implemented as a combination of both 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

- Processor
- Interrupt controller
- Memory
- IDE disk controller
- IDE disk
- Keyboard controller
- Keyboard
- Serial port controller
- Serial port
- SCSI controller
- SCSI bus
- SCSI disk
- CDROM
- Video adapter
- Display
- Network adapter
- Network
- Modem
- Modem
- Printer
- Parallel port controller
- Parallel port

Diagram showing the interconnection between different components of a computer system, including processor, memory, keyboard, mouse, modem, printer, interrupt controllers, IDE disk controller, SCSI controller, video adapter, network adapter, and various peripherals.
Exceptions

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

1. Each type of event has a unique exception number $k$

2. Jump table (interrupt vector) entry $k$ points to a function (exception handler).

3. Handler $k$ is called each time exception $k$ occurs.
Asynchronous exceptions (interrupts)

Caused by events (changes in state) 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
Synchronous exceptions

Caused by events (changes in state) that occur as a result of executing an instruction:

• Traps
  – intentional
  – returns control to “next” instruction
  – Examples: system calls, breakpoint traps

• Faults
  – unintentional but possibly recoverable
  – either re-executes faulting (“current”) instruction or aborts.
  – Examples: page faults (recoverable), protection faults (unrecoverable).

• Aborts
  – unintentional and unrecoverable
  – aborts current program
  – Examples: parity error, machine check.
Trap Example

Opening a File

- User calls `open(filename, options)`

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

- 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     exception    Open file
pop

return
```

---

class16.ppt
Fault Example #1

Memory Reference

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

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

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

User Process

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

OS

Create page and load into memory

```
Create page and load into memory
```
Fault Example #2

Memory Reference

- User writes to memory location
- Address is not valid

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

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

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

User Process

```
event
```

OS

```
page fault
Detect invalid address
Signal process
```

```
class16.ppt
```
Processes

Def: A process is an instance of a running program.
  • One of the most profound ideas in computer science.

Process provides each program with two key abstractions:
  • Logical control flow
    – gives each program the illusion that it has exclusive use of the CPU.
  • Private address space
    – gives each program the illusion that has exclusive use of main memory.

How is this illusion 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 as 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.
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("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("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\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 cleanup(void) {
    printf("cleaning up\n");
}

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

Zombies

Idea

• What 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 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

```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);
    }
}
```

```bash
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
    PID TTY    TIME   CMD
  6585 tty9    00:00:00  tcsh
  6676 tty9    00:00:06  forks
  6677 tty9    00:00:00  ps
linux> kill 6676
linux> ps
    PID TTY    TIME   CMD
  6585 tty9    00:00:00  tcsh
  6678 tty9    00:00:00  ps
```

- `ps` shows child process as “defunct”
- Killing parent allows child to be reaped
wait: Synchronizing with children

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

void fork8() {
    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();
    }
}

class16.ppt
Wait Example

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

```c
void fork9()
{
    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

- `waitpid(pid, &status, options)`
  - Can wait for specific process
  - Various options

```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 = 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` (fork9)**

<table>
<thead>
<tr>
<th>Child PID</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` (fork10)**

<table>
<thead>
<tr>
<th>Child PID</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**

```c
int execl(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!
```

```c
main() {
    if (fork() == 0) {
        execl("/usr/bin/cp", "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

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

Terminating Processes
- Call exit

Reaping Processes
- Call wait or waitpid