Exceptional Control Flow II
Oct 23, 2001

Topics
- Exceptions
- Process context switches

Role of Exceptions

Error Handling
- Error conditions detected by hardware and/or OS
  - Divide by zero
  - Invalid pointer reference

Getting Help from OS
- Initiate I/O operation
- Fetch memory page from disk

Process Management
- Create illusion that running many programs and services simultaneously

Exceptions

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

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 I/O available or given scheduling priority
- Appears to use (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
- `execv()` and `execve()` replace state of existing process with that of newly started program
  - Called once, never returns

Programming Challenge
- Understanding the non-standard semantics of the functions
- Avoiding improper use of system resources
  - Fewer safeguards provided

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");
}
```

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 */
    }
}
```

| Command | Process ID | User | Time (hh:mm:ss) | Command
---|---|---|---|---
| `ps` | 6639 | user | 00:00:00 | `kill 6639`
| `ps` | 6639 | user | 00:00:03 | `fors <defunct>`
| `ps` | 6640 | user | 00:00:00 | `fors <defunct>`
| `ps` | 6641 | user | 00:00:00 | `fors <defunct>`

- `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);
    }
}
```

**Exec Example**

**Task**
- Sort a set of files
  - E.g., `sortfiles f1.txt f2.txt f3.txt`
- Perform concurrently
  - Using Unix `sort` command
  - Commands of form `sort f1.txt -o f1.txt`

**Steps**
- Invoke a process for each file
- Complete by waiting for all processes to complete

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>

int main(int argc, char *argv[])
{
    int cnt = invoke(argc, argv);
    complete(cnt);
    return 0;
}
```

**Nonterminating Child Example (cont.)**

- Use `fork` and `exec` to spawn set of sorting processes

```c
int invoke(int argc, char *argv[])
{
    int i;
    for (i = 1; i < argc; i++) {
        /* Fork off a new process */
        if (fork() == 0) {
            /* Child: Invoke sort program */
            printf("Process #d sorting file %s\n", getpid(), argv[i]);
            if (exec("/bin/sort", "sort",
                     argv[i], "+o", argv[i], 0) < 0) {
                perror("sort");
                exit(1);
            }
            /* Never reach this point */
        }
    }
    return argc-1;
}
```

**Exec Example (cont.)**

- Use `wait` to wait for and reassociate terminating children

```c
void complete(int cnt)
{
    int i, child_status;
    for (i = 0; i < cnt; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Process #d completed with status #d\n", wpid, WEXITSTATUS(child_status));
        else
            printf("Process #d terminated abnormally\n", wpid);
    }
}
```
Signals

Signals
- Software events generated by OS and processes
  - an OS abstraction for exceptions and interrupts
- Sent from the kernel or a process to other processes.
- Different signals are identified by small integer ID’s.
- Only information in a signal is its ID and the fact that it arrived.

<table>
<thead>
<tr>
<th>Num.</th>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SIGINT</td>
<td>Terminate</td>
<td>Interrupt from keyboard (ctrl-c)</td>
</tr>
<tr>
<td>9</td>
<td>SIGKILL</td>
<td>Terminate</td>
<td>Kill program (cannot override or ignore)</td>
</tr>
<tr>
<td>11</td>
<td>SIGSEGV</td>
<td>Terminate &amp; Dump</td>
<td>Segmentation violation</td>
</tr>
<tr>
<td>14</td>
<td>SIGALRM</td>
<td>Terminate</td>
<td>Timer signal</td>
</tr>
<tr>
<td>17</td>
<td>SIGCHLD</td>
<td>Ignore</td>
<td>Child stopped or terminated</td>
</tr>
</tbody>
</table>

Sending Signals

Unix kill Program
- Sends arbitrary signal to process
  - e.g., /bin/kill –s 9 pid
    - sends SIGKILL to specified process

Function kill
- Send signal to another process
  kill(pid, signal)

Kill Example

```c
void fork12()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
    {
        if ((pid[i] = fork()) == 0) {
            /* Child: Infinite Loop */
            while(1);
        }
    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }
    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 terminated abnormally\n", wpid);
        }
    }
}
```

Handling Signals

Every Signal Type has Default Behavior
- Typically terminate or ignore

Can Override by declares special Signal Handler Function
- `signal(sig, handler)`
  - Indicates that signals of type `sig` should invoke function `handler`
  - Handler returns to point where exception occurred

```c
void int_handler(int sig)
{
    printf("Process %d received signal %d\n", getpid(), sig);
    exit(0);
}
void fork13()
{
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler);
    . . .
}
```
Signal Handler Funkiness

Signals are not Queued

- For each signal type, just have single bit indicating whether or not signal has occurred
- Even if multiple processes have sent this signal

```c
void child_handler(int sig)
{
    int child_status;
    pid_t pid = wait(child_status);
    count--;
    printf("Received signal %d from process %d\n", sig, pid);
}

void fork14()
{
    pid_t pid[N];
    int i, child_status;
    count = N;
    signal(SIGHUP, child_handler);
    for (i = 0; i < N; i++)
    if ((pid[i] = fork()) == 0) {
        /* Child: Exit */
        exit(0);
    }
    while (count > 0)
        pause();//* Suspend until signal occurs */
}
```

A program that reacts to externally generated events (ctrl-c)

```c
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

static void handler(int sig) {
    printf("You think hitting ctrl-c will stop the bomb?\n");
    sleep(2);
    printf("Well...\n");
    fflush(stdout);
    sleep(1);
    printf("OK!\n");
    exit(0);
}

main() {
    signal(SIGINT, handler);
    /* installs ctrl-c handler */
    while (1) {
        /* handler returns here */
    }
}
```

A program that reacts to internally generated events

```c
#include <stdio.h>
#include <signal.h>

int beeps = 0;

/* SIGALRM handler */
void handler(int sig) {
    printf("BEEP\n");
    fflush(stdout);
    if (++beeps < 5)
        alarm(1);
    else {
        printf("BOOM! \n");
        exit(0);
    }
}

main() {
    signal(SIGALRM, handler);
    alarm(1); /* send SIGALRM in 1 second */
    while (1) {
        /* handler returns here */
    }
}
```
Nonlocal jumps: setjmp() / longjmp()

Powerful (but dangerous) user-level mechanism for transferring control to an arbitrary location.
- controlled to way to break the procedure call/return discipline
- useful for error recovery

```c
int setjmp(jmp_buf j)
```
- must be called before longjmp
- identifies a return site for a subsequent longjmp.
- Called once, returns one or more times

**Implementation:**
- remember where you are by storing the current register context, stack pointer, and PC value in jmp_buf.
- return 0

```c
void longjmp(jmp_buf j, int i)
```
- meaning:
  - return from the setjmp remembered by jump buffer j again...
  - ...this time returning i
- called after setjmp
- Called once, but never returns

**longjmp Implementation:**
- restore register context from jump buffer j
- set %eax (the return value) to i
- jump to the location indicated by the PC stored in jump buf j.

---

**setjmp/longjmp example**

```c
#include <setjmp.h>
jmp_buf buf;

main()
{
    if (setjmp(buf) != 0)
        printf("back in main due to an error\n");
    else
        printf("first time through\n");
    p1(); /* p1 calls p2, which calls p3 */
    ...
    p3()
    <error checking code>
    if (error)
        longjmp(buf, 1)
}
```

---

**Putting it all together: A program that restarts itself when ctrl-c’d**

```c
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>

sigjmp_buf buf;

void handler(int sig)
{
    siglongjmp(buf, 1);
}

main()
{
    signal(SIGINT, handler);
    if (!sigsetjmp(buf, 1))
        printf("starting\n");
    else
        printf("restarting\n");
}
```

```c
while(1)
{
    sleep(1);
    printf("processing...\n");
}
```

```bash
bass> a.out
starting
processing...←Ctrl-c
restarting
processing...←Ctrl-c
restarting
processing...←Ctrl-c
restarting
processing...
```
Limitations of Long Jumps

Works Within Stack Discipline
- Can only long jump to environment of function that has been called but not yet completed

```c
jmp_buf env;
P1()
{
    if (setjmp(env)) {
        /* Long Jump to here */
    } else {
        P2();
    }
}
P2()
{
    . . . P2(); . . . P3();
}
P3()
{
    longjmp(env, 1);
}
```

Before longjmp

After longjmp

Summary

Signals Provide Process-Level Exception Handling
- Can generate with kill
- Can define effect by declaring signal handler

Some Caveats
- Very high overhead
  - >10,000 clock cycles
  - Only use for exceptional conditions
- Don’t have queues
  - Just one bit of status for each signal type

Long Jumps Provide Exceptional Control Flow Within Process
- Within constraints of stack discipline