UNM SCHOOL of ENGINEERING
Department of Computer Science

Lecture #24 : Chap. 8 Exceptional Control Flow (ECF) (2)

Prof. Soraya Abad-Mota, PhD

Previously

- *Exceptions*:
  - when do they happen?
  - Types of exceptions: *asynchronous (interrupts)* and *synchronous* (classified as: *traps, faults*, and *aborts*)
  - How to handle them? Who handles each kind? What's next once they are handled? (HW, OS software, user?)
- The *process* abstraction

From the index cards

- Why do we need exception handlers?
- Page fault: what is it? (more when we cover virtual memory)
- Concurrency vs parallelism (more today)
- What is the *kernel*? (more today)

Concurrent Processes

- Each process is a logical control flow.
- Two processes *run concurrently* (*are concurrent)* if their flows overlap in time
- Otherwise, they are *sequential (one after the other)*
- Examples (running on single core):
  - 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 (an illusion) of concurrent processes as running in parallel with each other

Context Switching (sec. 8.2.5)

- Processes are managed by a shared chunk of memory-resident OS code called the *kernel*
  - Important: the kernel is not a separate process, but rather runs as part of some existing process.
- Control flow passes from one process to another via a *context switch*

The mechanism of context switching

1. save context of current process
2. restore saved context of previously preempted process
3. pass control to this newly restored process

Who does this?

What triggers context switching?

Topics

Exceptional Control Flow

- Exceptions
- The Process abstraction
- Process Control
  - error handling
  - system calls for manipulating processes:
    - creation and termination
    - destruction and reaping
    - loading and running programs
    - process commands and shells

System Call Error Handling (sec. 8.3)

- On error, Linux system-level functions typically return -1 and set global variable `errno` to indicate cause.
- Hard and fast rule:
  - You must check the return status of every system-level function
  - Only exception is the handful of functions that return `void`
- Example: (with the fork system function)

```
if ((pid = fork()) < 0) {
    fprintf(stderr, "fork error: %s\n", strerror(errno));
    exit(0);
}
```

Error-reporting functions

- Since some skip it because it makes the code hard to read
- Can simplify somewhat using an *error-reporting function*:

```
void unix_error(char *msg) /* Unix-style error */
{
    fprintf(stderr, "%s: %s\n", msg, strerror(errno));
    exit(0);
}
```

```
 if ((pid = fork()) < 0)
    unix_error("fork error");
```

Error-handling Wrappers

- Simplify the code presented to you even further by using Stevens-style error-handling wrappers:

```
pid_t Fork(void)
{
    pid_t pid;

    if ((pid = fork()) < 0)
        unix_error("Fork error");
    return pid;
}
```

Notice that the name of the wrapper is the same as the function but with capital F

```
  pid = Fork();
```

Topics

Exceptional Control Flow

- Exceptions
- The Process abstraction
- Process Control
  - error handling
  - system calls for manipulating processes:
    (provided by Unix to C programs)
    - creation and termination (Sec. 8.4.2)
    - destruction and reaping
    - loading and running programs
    - process commands and shells

Obtaining Process IDs (sec. 8.4.1)

- `pid_t getpid(void)`
  - Returns PID of current process

- `pid_t getppid(void)`
  - Returns PID of parent process (i.e. the process that created the calling process)

Creating and Terminating Processes

(section 8.4.2)

From a programmer's perspective, we can think of a process as being in one of three states

- Running
  - Process is either executing, or waiting to be executed and will eventually be *scheduled* (i.e. chosen to execute) by the kernel.
- Stopped
  - Process execution is *suspended* and will not be scheduled until further notice (we study signals to understand this better).
- Terminated
  - Process is stopped permanently . Signal to terminate, returning from main, or call exit(). (Next slide)

Dynamic diagram of a *process*

Terminating Processes

- Process becomes terminated for one of three reasons:
  - Receiving a signal whose default action is to terminate (signals next lecture)
  - Returning from the `main` routine
  - Calling the `exit` function

- `void exit(int status)`
  - Terminates with an *exit status* of `status`
  - Convention: normal return status is 0 and nonzero on error
  - Another way to explicitly set the exit status is to return an integer value from the main routine

- `exit` function is called once but never returns.

Creating Processes

- *Parent process* creates a new running *child process* by calling `fork`

- `int fork(void)`
  - Returns 0 to the child process, child's PID to parent process
  - Child is *almost* identical to parent:
    - Child gets an identical (but separate) copy of the parent's virtual address space.
    - Child gets identical copies of the parent's open file descriptors (we cover this in Ch. 10: I/O)
    - Child has a different PID than the parent

- `fork` is interesting (and often confusing) because it is called *once* but returns *twice*

fork Example

```c
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
        exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

fork.c

```
linux> ./fork
parent: x=0
child : x=2
```

- **Call once, return twice**
- **Concurrent execution**
  - Can't predict execution order of parent and child
- **Duplicate but separate address space**
  - `x` has a value of 1 when fork returns in parent and child
  - Subsequent changes to `x` are independent
- **Shared open files**
  - `stdout` is the same in both parent and child

Practice Problem 8.2 (p. 743)

```
int main()
{
    int x = 1;
    if (Fork() == 0)
        printf("p1: x=%d\n",++x);
    printf("p2: x=%d\n",--x);
    exit(0);
}
```

Output of child process?

Output of parent process?

Process graph?

starts in 2 slides (47)

Topics

Exceptional Control Flow

- Exceptions
- The Process abstraction
- Process Control
  - error handling
  - system calls for manipulating processes:
    - creation and termination
    - modeling fork with Process Graphs
    - destruction and reaping
    - loading and running programs
    - process commands and shells

Modeling `fork` with Process Graphs

- A *process graph* is a useful tool for capturing the partial ordering of statements in a concurrent program:
  - Each vertex is the execution of a statement
  - a -> b means `a` happens before b
  - Edges can be labeled with current value of variables
  - `printf` vertices are labeled with output
  - Each graph begins with a vertex with no in-edges
- Any *topological sort* of the graph corresponds to a feasible total ordering.
  - Total ordering of vertices where all edges point from left to right

Process Graph Example

```c
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
    exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

fork.c

Interpreting Process Graphs

- Original graph:

- Relabeled graph:

Feasible total ordering:

Infeasible total ordering:

f cannot be executed before e

fork Example: Two consecutive forks

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

Feasible output:
L0
L1
Bye
Bye
L1
Bye
Bye

Infeasible output:
L0
Bye
L1
Bye
L1
Bye
Bye

Topics

Exceptional Control Flow

- Process Control
  - system calls for manipulating processes:
    - creation and termination
    - two more examples of modeling fork with Process Graphs
    - destruction and reaping
    - loading and running programs
    - process commands and shells

Two more examples

- of programs using fork
  - fork4
  - fork5
- let's do them before we continue
- what is done by the fork function? (stop for a moment and write it down)

example fork4

```
void fork4()
 {
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

fork Example: Nested forks in children

Practice

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

forks.c

Lifetime of a process (recall)

What happens in Terminated state?

- Terminated
  - process still consumes memory resources
  - it cannot be scheduled again but has memory allocated
- To free all its resources, the process must be reaped

Reaping Child Processes (sec. 8.4.3)

- Idea
  - Process terminates, it still consumes system (memory) resources
    - Examples: Exit status, various OS tables
  - Called a "zombie"
    - "Living corpse, half alive and half dead"
- Reaping
  - Performed by parent on terminated child (using `wait` or `waitpid`)
  - Parent is given exit status information
  - Kernel then discards zombie child process, which ceases to exist
- What if parent doesn't reap?
  - If any parent terminates without reaping a child, then the orphaned child will be reaped by **`init`** process (pid == 1)
  - So, only need explicit reaping in 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 */
    }
}
```

forks.c

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

- `ps` shows child process as "defunct" (i.e., a zombie)
- Killing parent allows child to be reaped by `init`

Non-terminating 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);
    }
}
```

forks.c

```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

- Child process still active even though parent has terminated
- Must kill child explicitly, or else will keep running indefinitely

`wait`: Synchronizing with Children

- Parent reaps a child by calling the `wait` function
- `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 != NULL`**, then the integer it points to will be set to a value that indicates reason the child terminated and the exit status:
    - Checked using macros defined in `wait.h`
      - `WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG, WIFSTOPPED, WSTOPSIG, WIFCONTINUED`
      - Used to interpret the status argument, see textbook for details (p.745)

wait: Synchronizing with Children

```c
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
        exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
```

forks.c

Feasible output: (why?)

HC: hello from child
HP: hello from parent
CT: child has terminated
Bye

Infeasible output:

HP: hello from parent
CT: child has terminated
Bye
HC: hello from child

Another wait Example

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status (detailed explanation in textbook pp. 746-749)

```
void fork10() {
    pid_t pid[N];
    int i, child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) { /* creating N children*/
            exit(100+i); /* Child */
        }
    for (i = 0; i < N; i++) { /* Parent */
        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);
    }
}
```

forks.c

waitpid: Waiting for a Specific Process

- `pid_t waitpid(pid_t pid, int &status, int options)`
  - Suspends current process until specific process terminates
  - Various options (see textbook)

```
void fork11() {
    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 = N-1; i >= 0; 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 terminate abnormally\n", wpid);
    }
}
```

forks.c

Topics

- Process Control:
  - system calls for manipulating processes:
    - two more examples on modeling for with process graphs
    - destruction and reaping
    - loading and running programs (execve in sec. 8.4.5)
    - process commands and shells

students read 8.4.4 on their own
(Putting Processes to sleep: sleep and pause functions)

`execve`: Loading and Running Programs

- `int execve(char *filename, char *argv[], char *envp[])`
- Loads and runs in the context of current process:
  - Executable file **`filename`**
    - Can be object file or script file beginning with `#!interpreter` (e.g., `#!/bin/bash`)
  - …with argument list **`argv`**
    - By convention **`argv[0]==filename`** (of the file to be executed)
  - …and environment variable list **`envp`**
    - "name=value" strings (e.g., `USER=droh`)
    - `getenv, putenv, printenv`
- Overwrites code, data, and stack
  - Retains PID, open files and signal context
- Called once and never returns
  - …except if there is an error

Structure of the stack when a new program starts

execve Example

- **Executes** "/bin/ls -lt /usr/include" **in child process using current environment:**

```
if ((pid = Fork()) == 0) {   /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: Command not found.\n", myargv[0]);
        exit(1);
    }
}
```

Summary

- Exceptions (in general)
  - 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 on a single core, though
  - Each process appears to have total control of processor + private memory space

Summary (cont.)

- Spawning processes
  - Call `fork`
  - One call, two returns
- Process completion
  - Call `exit`
  - One call, no return
- Reaping and waiting for processes
  - Call `wait` or `waitpid`
- Loading and running programs
  - Call `execve` (or variant)
  - One call, (normally) no return

If enough time

- Continue with the final 7 slides

and

Next time: *Signals* (to finish Chapter 8)