Exceptional Control Flow: Exceptions and Processes

1 异常控制流

处理器自开机至关机，对各条指令顺序执行。是为控制流。

转移和调用是两种改变控制流（执行顺序）的机制，它们用户程序状态改变作出响应。

但为了响应系统状态改变，需要异常控制流。它是由内部或外部的某种事件造成的控制流转移。

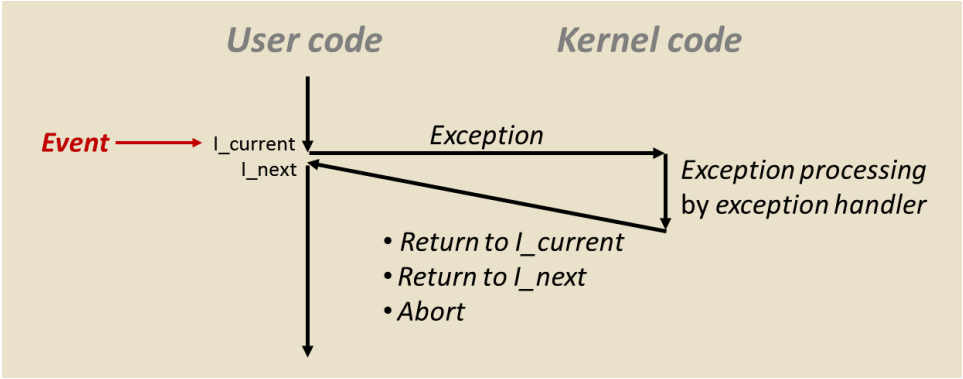
ECF 出现在系统各层次上：

- 低层机制：exception，由硬件和操作系统协同实现；
- 高层机制：process context switch, signal 由操作系统实现，nonlocal jump 由 C 运行时库实现。

2 异常

2.1 基本概念

每个用户程序的虚拟内存的最上端，都有一部分系统内核（OS kernel）的映射，程序通过这一部分进行系统和用户程序的交互。异常即是指程序对某事件作出响应时，将控制转移至系统内核。如图所示：



图：异常发生与处理示意图

©Randy Bryant, CMU

内核保存了一张异常表，每个类型的异常均在其中有一个唯一的索引号，且有一个相应的处理程序。当异常发生时，内核在表中检索与该异常相应的处理程序进行处理。处理后的动作，如上图所示，有返回至当前指令、返回至下一条指令、中止，共三种。

2.2 异常的分类

异常分为异步与同步异常两类，同步异常中又包含 trap、fault、abort 等类别。

2.2.1 异步异常

异步异常即中断（interrupt），是由处理器外部事件引发的，如定时器到时、程序过程中按下 Ctrl-C、网络数据包抵达或磁盘数据抵达等。

异步异常的处理程序通常返回至下一条指令。

2.2.2 同步异常

同步异常包括：

- trap，包括系统调用、断点、特殊指令，是程序有意设置的异常。通常处理方式为返回下一条指令。
- fault，非有意设置，有可能可恢复。如 page fault（即虚拟内存中页面未加载，可恢复）和 protection fault（例如 segmentation fault 保护程序，不可恢复）。可能返回当前指令或者退出。
- abort，非有意、不可恢复。如非法指令、内存或磁盘检测错误等。

2.3 系统调用

系统调用是同步异常中 trap 的一种，指运行在使用者空间的程序向操作系统内核请求需要更高权限运行的服务。

每一个 x86-64 系统调用都包含一个独立的 ID 编号。在汇编语言层面，调用系统调用时，先将相应的编号放入%rax 寄存器，将参数放入%rdi、%rsi 等寄存器（这一点与普通函数调用相同），然后直接使用 syscall 指令调用。调用返回值储存在%rax 中。

如系统调用发生错误，则返回后会将错误编码保存在全局变量 errno 中。这是系统调用与普通函数调用的另一个不同之处。

3 进程

3.1 基本概念

进程是指程序运行时产生的一个实例。它与“程序”的意义不同：通过一个程序可以创建多个进程。

操作系统运行时，内核维护一个进程表，包含所有正在运行的、以及终止后资源未释放的进程。进程表中每个进程都有一个独立的 pid 作为标识。

进程的概念对每个程序都提供两个关键的抽象：每个程序都似乎独占 CPU、独占内存资源。事实上，内存中包括了运行中每个进程的虚拟内存、外加寄存器的保存值。一个 CPU 核心同时只能处理一个进程，但可以在不同进程之间切换。切换时，先中断控制流、将寄存器的值保存至旧进程的相应位置，然后加载新进程中已保存的寄存器值，按新进程的控制流执行。这便提供了不同进程并发执行的假象。

进程切换需要通过操作系统内核完成，这一步骤称作上下文切换（context switch）。须注意的是，内核并非一个独立的进程，而是现存进程中的一部分。

若两个进程的时间起止点之间有重合，则称这两个进程为并发的（concurrent）。特别地，如果两进程同时运行于不同 CPU 核心，则称之为并行的（parallel）。时间无重合的进程称为串行的（sequential）。

进程的可能状态有三种：运行、暂停和终止。运行的进程亦或正在执行、亦或正等待执行（未来将被安排 CPU 执行）；暂停的进程不被执行、且在重新进入运行前不会被安排执行。

3.2 关于进程的系统调用

3.2.1 fork

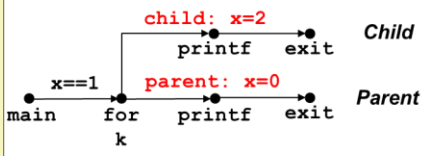
该系统调用无参数，作用是创建一个与当前进程完全相同的拷贝，并将后者作为当前进程的子进程。整个虚拟内存空间的内容都相同。一经执行，在两个进程中都会返回，自此并行执行。在父进程中返回子进程 pid（大于 0），子进程返回 0，代码凭此区分哪一进程正进行调用。

并行的进程部分存在竞争（race）的问题，即两者执行的顺序不确定。当对不同进程的执行顺序有所要求时，这一点需要特别注意。虽然运行多达 10000 次执行顺序的策略都可能是一样的，但处理器并不对此提供保证。可能的执行顺序，可以用进程图（有向无环图，节点表示指令，偏序关系表示限制的执行顺序）的所有可能拓扑排序来表示。

```
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

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

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



图：进程图示例

©Randy Bryant, CMU

如果需要确保父子进程的执行顺序，可以在二者中加入一个时长安全的延时（通过 sleep 系统调用暂停进程执行）。

父进程和子进程对已打开的文件是共享的。

3.2.2 wait 和 waitpid

进程终止之后，分配给该进程的内存资源需要被释放，同时该进程将从进程表中移除。此时该进程即被收割（reap）。进程终止后与被收割前，为僵尸进程（zombie）。僵尸进程被收割后，其 pid 与在进程表中的表项都可以被系统重用；如果未被收割，则僵尸进程将保留进程表中的表项，导致资源泄漏。

进程收割仅能由其（直接的）父进程调用 wait 或 waitpid 系统调用实现。调用这两种方法时，父进程将被暂停，直至某子进程终止为止，此时将其收割。

wait 函数包括一个 int 指针参数，用于保存子进程的退出状。该函数等待第一个被终止的子进程。

waitpid 可指定所等待子进程的 pid，且可设置同时等待进程暂停、可设置若当前无终止（或暂停）子进程即不予等待。

若等待成功，两个函数将返回收割子进程的 pid、未收割则返回 0。若调用过程出错，则返回负值、且将错误信息保存在 errno 中。

当父进程终止却没有收割子进程，则它的子进程由 init 进程（pid = 1）接管。

父进程的父进程不能收割的原因是，父进程的父进程没有对子进程 pid 的引用。

init 进程是系统进程，一旦被终止则系统崩溃。

3.2.3 execve 系列函数

这些函数用于将当前进程变为另一个程序，即将进程所属虚拟内存中的代码段、数据段完全替换为一个所指定的新程序初始化后的状态；仅保留进程的 pid。

这些函数若执行正常，则不会返回；若有返回则表示执行出错。

execve 函数接受 3 个参数：欲执行的程序文件路径、程序接受的命令行参数以及环境变量。

环境变量是一个键值对形式的字符串数组，在程序初始化时位于栈的顶端，用于保存程序执行所需的环境信息。

execve 通常配合 fork 使用。若不使用 execve，则每次 fork 后、父子进程各自所需执行的代码，都需要一个判断 fork 返回值的条件分支语句。在子进程中直接调用 execve，则有助于代码整洁及模块化管理。

操作系统壳（shell）的工作是打开用户所需执行的程序，它的原理便是：作为一个运行的进程，使用 fork + execve，在子进程中打开用户所要求执行的程序。

附：waitpid 用法

表头文件

```
#include<sys/types.h>
#include<sys/wait.h>
```

函数定义

```
pid_t waitpid(pid_t pid,int * status,int options);
```

函数说明

waitpid()会暂时停止目前进程的执行,直到有信号来到或子进程结束。

如果在调用 wait()时子进程已经结束,则 wait()会立即返回子进程结束状态值。

子进程的结束状态值会由参数 status 返回,而子进程的进程识别码也会一快返回。

如果不在意结束状态值,则参数 status 可以设成 NULL。

参数 pid

参数 pid 为欲等待的子进程识别码,其他数值意义如下:

pid<-1 等待进程组识别码为 pid 绝对值的任何子进程。

pid=-1 等待任何子进程,相当于 wait()。

pid=0 等待进程组识别码与目前进程相同的任何子进程。

pid>0 等待任何子进程识别码为 pid 的子进程。

参数 option

参数 option 可以为 0 或下面的 OR 组合:

WNOHANG 如果没有任何已经结束的子进程则马上返回, 不予以等待。

WUNTRACED 如果子进程进入暂停执行情况则马上返回,但结束状态不予以理会。

退出状态

子进程的结束状态返回后存于 status,底下有几个宏可判别结束情况:

WIFEXITED(status)如果子进程正常结束则为非 0 值。

WEXITSTATUS(status)取得子进程 exit()返回的结束代码,一般会先用 WIFEXITED 来判断是否正常结束才能使用此宏。

WIFSIGNALED(status)如果子进程是因为信号而结束则此宏值为真

WTERMSIG(status) 取得子进程因信号而中止的信号代码,一般会先用 WIFSIGNALED 来判断后才使用此宏。

WIFSTOPPED(status) 如果子进程处于暂停执行情况则此宏值为真。一般只有使用 WUNTRACED 时才会有此情况。

WSTOPSIG(status) 取得引发子进程暂停的信号代码,一般会先用 WIFSTOPPED 来判断后才使用此宏。

返回值

如果执行成功则返回子进程识别码(PID) ,如果有错误发生则返回返回值-1。失败原因存于 errno 中。

来自 <http://www.2cto.com/os/201203/124851.html>