ECF异常控制流

2021年计算机系统导论讨论班

参考投影片（11月30日）

杨宇昕 王畅

1.1 异常控制流的概念

- 逻辑控制流指的是一个进程中执行的指令的PC值的序列
- 对比：
  - 跳转分支和调用返回是**程序状态**导致的控制流突变
  - 需要系统对系统状态作出反应，称为**异常控制流**

- 例：给出几个出现异常控制流的例子？
  - 磁盘数据已经准备好、除以0、用户键入Ctrl-C

1.2 异常的类型

- 有多套语汇，我们遵循异常=中断+陷阱+故障+中止的范式

- 常见考点1：掌握4种异常类型的联系和区别，知道分属各类型的异常的例子若干（选择题）
- 要求：理解并记忆课本图8-4的表格，阅读课本8.1.2的四个要点

- 注意点：同步/异步，返回到哪条指令，是否可能中止
  - 中断是异步的，硬件中断不是执行任何一条指令的结果（即使是DMA方式I/O也需要中断）；一般而言，中断时需要先执行完当前指令。陷阱是有意的，主要是系统调用。故障可能返回也可能不返回（缺页、除法错误、一般保护故障）。中止的例子（机器检查、物理内存错误）。

1.2 异常vs调用

调用	异常
返回地址一定是下一条指令	返回地址可能是当前指令，可能是下一条，也可能不返回
只把参数和返回地址压入用户栈中	会把恢复中断状态所需的额外的处理器状态（如EFLAGS）压入内核栈中
运行在用户模式	运行在内核模式（超级用户模式）

1.2 系统调用

- 由特殊指令触发（x86-64 syscall）
- 将陷入内核，提升为内核态。用户从操作系统得到服务。这样操作系统可以保护自身及资源，同时又能满足用户需求
- 往往遵循一套与普通函数调用不同的参数传递惯例（参P506下部，熟悉之）
- 反汇编中一般看不到，被库函数包装了

1.2 轮流回答问题

- 以下说法错误的是哪个？

A. 发生异常和异常处理意味着控制流的突变。

B. 与异常相关的处理是由硬件和操作系统共同完成的。

C. 异常是由于计算机系统发生了不可恢复的错误导致的。

D. 异常的发生可能是异步的，也可能是同步的。

C

1.2 轮流回答问题

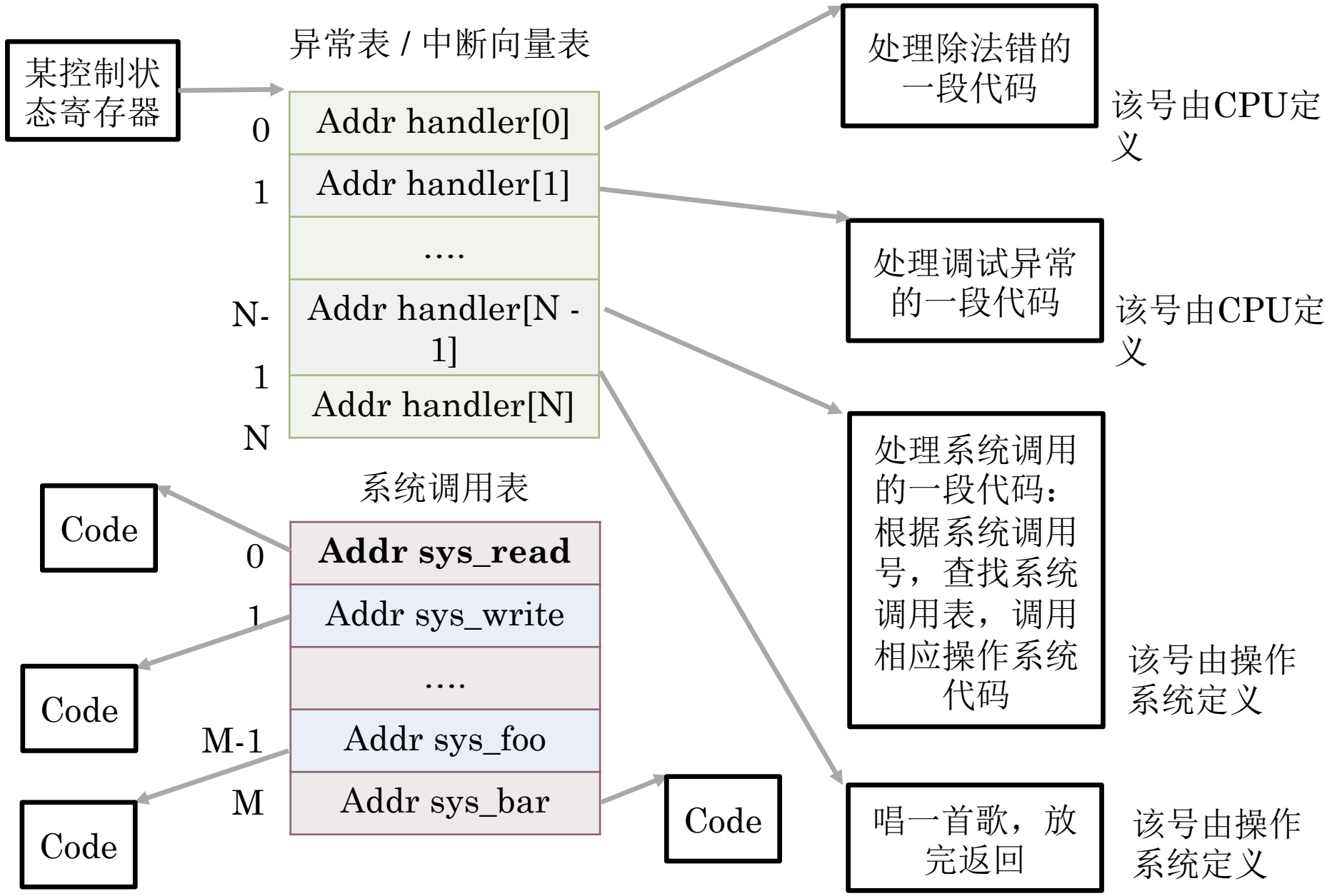
- 下列行为分别触发什么类型的异常？

1.	执行指令mov $57, %eax; syscall	1. （陷阱）
2.	程序执行过程中，发现它所使用的物理内存损坏了	2. （中止）
3.	程序执行过程中，试图往main函数的内存中写入数据	3. （故障）
4.	按下键盘	4. （中断）
5.	磁盘读出了一块数据	5. （中断）
6.	用read函数发起磁盘读	6. （陷阱）
7.	用户程序执行了指令lgdt，但是这个指令只能在内核模式下执行	7. （故障）

1.3 异常的处理过程

- 异常是硬件和软件共同实现的
  - 一般来说，异常表、跳转到异常表由硬件实现，异常处理程序由软件（操作系统）实现
  - 启动时，异常表由操作系统填写（例如一开始由BIOS负责，进入保护模式后另外填写）

- 复述异常处理的过程
  - 检测事件，确定异常号，切换内核模式，保存上下文
  - 用异常表基址寄存器和异常号得到处理例程的入口地址
  - 触发异常，间接跳转
  - 进行异常处理
  - （如果需要返回）恢复上下文，回到用户模式


某控制状态寄存器
异常表 / 中断向量表
0
Addr handler[0]
1
Addr handler[1]
....
N-1
Addr handler[N - 1]
N
Addr handler[N]
处理除法错的一段代码
该号由CPU定义
处理调试异常的一段代码
该号由CPU定义
处理系统调用的一段代码：根据系统调用号，查找系统调用表，调用相应操作系统代码
该号由操作系统定义
唱一首歌，放完返回
该号由操作系统定义
系统调用表
Code
0
Addr sys_read
1
Addr sys_write
....
M-1
Addr sys_foo
M
Addr sys_bar
Code
Code
Code

2.1 进程的基本概念

- 进程是执行中程序的实例，几乎可以和程序本身无差别混用
- 上下文：该实例的状态集合，包括代码、数据、栈、通用寄存器、程序计数器、环境变量、描述符表，均为独立占有
- 独立性：逻辑控制流独立（c.f. 1.1）、地址空间私有
- 仔细读图8-13

2.2 调度、抢占、上下文切换

- 操作系统通过调度程序，决定什么进程上CPU执行
- 操作系统通过给每个进程一个可执行的时间片，并在时间片用完或者其他系统事件发生时切换它们的上下文，来使不同的进程得以安全地并发执行，同时向用户制造出它们在同时执行的假象
  - 切换过程：保存失去CPU/被抢占的进程的上下文，读出要上CPU进程的上下文，控制转移到上CPU进程暂停的地方
- 例：发起I/O，主动放弃CPU，阻塞，系统去做别的事；I/O数据准备好，中断（也发生上下文切换），系统重新调度排队，若调度之，则控制继续。参图8-14例子熟悉之

2.3 内核模式和用户模式

- 由硬件中控制寄存器的位决定
- 为了保护系统和硬件资源，以及限制用户行为
- 用户代码总是要运行在用户态下。处理异常的操作系统代码一般需要运行在内核态下。如果需要使用到用户态不能提供的服务，那么必须在内核态下进行

2.4 并发流和并行流

- 逻辑控制流相互重叠就称为并发

- 常见考点2：并发和并行的区别
- 并行一定是并发，假并行（即通过上下文切换形成同时执行的假象）是并发但不是并行

2.4 轮流回答问题

- 考虑如下进程：

进程	开始时间	结束时间	运行处理器
A	5	7	P0
B	2	4	P1
C	3	6	P0
D	1	8	P1

- 判断以下进程对是否并行、并发：AB、AC、AD、BD
- AB都不是、AC并发不并行、AD并发且并行、BD并发不并行

2.5 进程管理函数

- 一系列系统函数或者系统调用接口。后者例如fork(), waitpid(), execve()，它们封装了更底层的操作系统提供给用户的系统调用
- 常见考点3：常见API的使用注意事项（参数、返回值）
- 函数非常多，建议亲自写程序体会，也可以用man命令查手册记住

- getpid()返回值是有符号整数
- fork()函数是重难点。第一是它返回两次，父进程返回PID，子进程返回0。第二个是子进程是父进程的一份独立的副本（结合系统I/O）。
- waitpid()是另一个重难点。僵尸子进程原则上要回收；如果子进程尚未结束而父进程结束了，则它们都变成孤儿，由init代管。注意学习掩码使用的范式
- execve()执行成功时不返回

2.5 轮流回答问题

- 关于进程，以下说法正确的是：

A. 没有设置模式位时，进程运行在用户模式中，允许执行特权指令，例如发起I/O操作。

B. 调用waitpid(-1, NULL, WNOHANG & WUNTRACED)会立即返回：如果调用进程的所有子进程都没有被停止或终止，则返回0；如果有停止或终止的子进程，则返回其中一个的PID。

C. execve函数的第三个参数envp指向一个以null结尾的指针数组，其中每一个指针指向一个形如name=value的环境变量字符串。

D. 进程可以通过使用signal函数修改和信号相关联的默认行为，唯一的例外是SIGKILL，它的默认行为是不能修改的。

C

2.5 进程管理函数

- 常见考点4：结合fork的程序分析（系统I/O中还有更多）

- 方法：理解父进程和子进程的联系；会画进程图，并能快速进行拓扑排序（回忆数据结构的知识）
- 不要用书上的那种图，建议使用更加简明的画法
- 当需要wait时，要在关系图中，子进程结束前的最后一个输出向父进程wait后的第一个输出连一条有向边，然后再拓扑排序

2.5 关于printf的问题

- 缓冲：内存空间的一个区域
- 设想一个用户open一个磁盘文件然后每次从文件读取1个字符，一直读到换行符。缓冲把突发的大数量较小规模的I/O整理成平稳的小数量较大规模的I/O（可提高性能）
- 输入缓冲区/输出缓冲区
- 全缓冲/行缓冲/不带缓冲
  - 例：行缓冲：键盘输入，先放在缓冲区，直到按回车或者满就刷新。
- 缓冲区刷新：缓冲区满时、关闭文件、fflush函数
- printf一般认为是缓冲区满、fflush、换行和结束时刷新，输出缓冲区中内容

2.5 轮流回答问题

- 下面的程序中，父进程的输出是什么，子进程的输出是什么？

```
int main()
{
    int a = 9;
    if (Fork() == 0)
        printf("p1: a=%d\n", a--);
    printf("p2: a=%d\n", a++);
    exit(0);
}
```

2.5 轮流回答问题

- 下列程序可能的输出是：
  - A. 1 2 -1 0
  - B. 0 0 -1 1
  - C. 1 -1 0 0
  - D. 0 -1 1 2

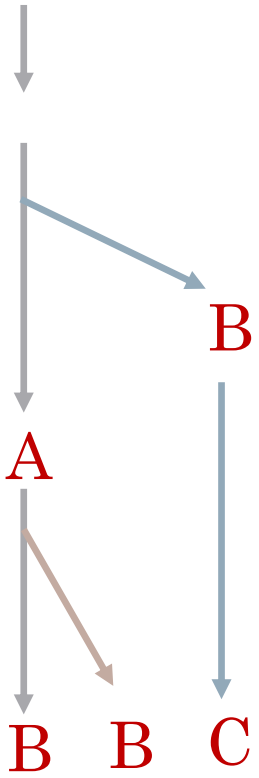
- A

```
int count = 0;
int pid = fork();
if (pid == 0) {
    printf("count = %d\n", --count);
} else {
    printf("count = %d\n", ++count);
}
printf("count = %d\n", ++count);
```

2.5 轮流回答问题

- 阅读右边的程序，下列哪些输出是可能的？

- AABBBC **Y**
- ABCABB **Y**
- ABBABC **N**
- AACBBC **N**
- ABABCB **Y**
- ABCBAB **N**

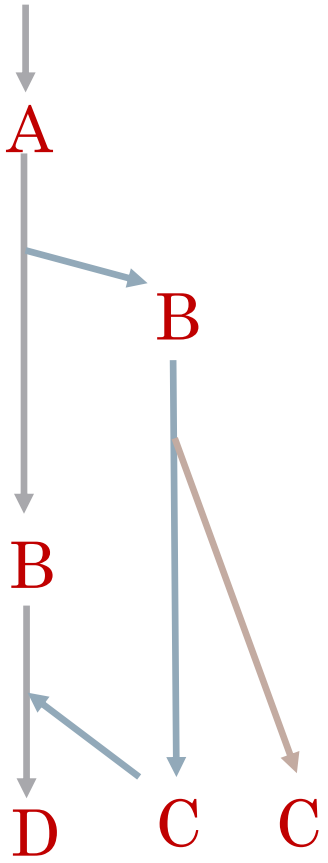



```c
int main() {
    char c = 'A';
    printf("%c", c);
    fflush(stdout);
    if (fork() == 0) {
        c++;
        printf("%c", c);
        fflush(stdout);
    }
    else {
        printf("%c", c);
        fflush(stdout);
        fork();
    }
    c++;
    printf("%c", c);
    fflush(stdout);
    return 0;
}
```

2.5 轮流回答问题

- 阅读左边的程序，下列哪些输出是可能的？

- ABBCCD **Y**
- ABBCDC **Y**
- ABBDCC **N**
- ABDBCC **N**
- ABCDBC **N**
- ABCDCB **N**




```c
int main() {
    int child_status;
    char c = 'A';
    printf("%c", c);
    fflush(stdout);
    c++;
    if (fork() == 0) {
        printf("%c", c);
        fflush(stdout);
        c++;
        fork();
    }
    else {
        printf("%c", c);
        fflush(stdout);
        c += 2;
        wait(&child_status);
    }
    printf("%c", c);
    fflush(stdout);
    exit(0);
}
```

2.5 轮流回答问题

- 以下程序的运行结果输出几个1？
- （5个）

```
int main() {
    if (fork() || fork())
        fork();
    printf("1 ");
    return 0;
}
```

2.5 轮流回答问题

- 以下程序的运行结果输出几个2？
- （7个）

```
int main() {
    if (fork() && (!fork())) {
        if (fork() || fork()) {
            fork();
        }
    }
    printf("2 ");
    return 0;
}
```

3.1 信号

- 信号是软件形式的异常

- 读图8-26，了解常见的信号，默认行为以及触发原因，包括
  - SIGHUP
  - SIGINT
  - SIGKILL
  - SIGSEGV
  - SEGALRAM
  - SIGTSTP
  - SIGCHLD
  - SGFPE
  - SIGSTOP
  - SIGUSR1
  - SIGUSR2

3.2 信号的发送与接收

- 发送：可以发送给自己（kill函数通用，alarm特别）
- 接收：可以忽略，终止或者执行一个信号处理程序（signal函数注册handler）
  - 惟SIGKILL、SIGSTOP（不要和SIGTSTP弄混了）不能修改默认行为
- 待处理：等待接收。一种类型至多有一个，不排队，后来的被丢弃；在pending位向量中维护，接收时被清除
- 阻塞：选择性地不接收，仍可以发送；在blocked位向量中维护
  - 隐式阻塞：处理某个信号时，相同类型信号会被阻塞
  - 显式阻塞：sigprocmask函数（熟悉用法）

- 常见考点5：信号发送、接收、处理、阻塞的注意事项

3.2 轮流回答问题

- 设一段程序中阻塞了SIGCHLD、SIGUSR1信号。接下来，向它按顺序发宋SIGCHLD、SIGUSR1、SIGCHLD信号，当程序取消阻塞继续执行时，将处理这三个信号中的哪几个？

- 处理一次SIGCHLD，一次SIGUSR1

3.2 轮流回答问题

- 判断下列说法正确性

1. SIGTSTP信号既不能被捕获，也不能被忽略
2. 存在信号的默认处理行为是进程停止直到被SIGCONT信号重启
3. 系统调用不能被中断，因为那是操作系统的工作
4. 在任何时刻，一种类型至多只会有一个待处理信号
5. 信号既可以发送给一个进程，也可以发送给一个进程组
6. SIGTERM和SIGKILL信号既不能被捕获，也不能被忽略
7. 当进程在前台运行时键入Ctrl-C，内核就会发一个SIGINT信号给这个前台进程
8. 子进程能给父进程发送信号，但不能发送给兄弟进程

1. （错）
2. （对）
3. （错）
4. （对）
5. （对）
6. （错）
7. （对）
8. （错）

3.2 轮流回答问题

- 阅读右边的程序，下列哪些输出是可能的？

- ACBC **Y**
- ABCCD **Y**
- ACBDC **N**
- ABDCC **N**
- BCDAC **Y**
- ABCC **Y**

```c
void handler() {
    printf("D\n");
    return;
}
int main() {
    signal(SIGCHLD, handler);
    if (fork() > 0) {
        printf("A\n");
    } else {
        printf("B\n");
    }
    printf("C\n");
    exit(0);
}
```

3.2 轮流回答问题

- 阅读右边的程序，给出全部可能的结果。

```c
int c = 1;
void handler1(int sig) {
    c++;
    printf("%d", c);
}

int main() {
    signal(SIGUSR1, handler1);
    sigset_t s;
    sigemptyset(&s);
    sigaddset(&s, SIGUSR1);
    sigprocmask(SIG_BLOCK, &s, 0);

    int pid = fork() ? fork() : fork();
    if (pid == 0) {
        kill(getppid(), SIGUSR1);
        printf("S");
        sigprocmask(SIG_UNBLOCK, &s, 0);
        exit(0);
    } else {
        while (waitpid(-1, NULL, 0) != -1);
        sigprocmask(SIG_UNBLOCK, &s, 0);
        printf("P");
    }
    return 0;
}
```

3.3 信号处理的安全问题

- 请仔细阅读课本的shell框架，学习优良的代码范式

- 阅读课本P533-546，通过shell lab学习

- 内容较多，但主要原理就两个
- 一、信号处理程序是一种“随时打断”式的程序，其内部最好不要被中断，而其他普通代码很多不能保证时刻正确响应信号处理程序的存在。（有关阻塞信号、全局共享数据的处理、volatile关键字、可重入函数或原子的的概念）
- 二、进程调度顺序是不确定的，不能有任何假设，否则可能有问题。因此必须用阻塞、同步等方式防止出现racing（冒险/竞争）。（有关sigsuspend、阻塞等）

3.4 非本地跳转

- setjmp和longjmp函数的用法
  - longjmp从env中恢复调用环境，然后从最近的一次setjmp中返回，它的第二个参数会成为setjmp此次的返回值
  - setjmp的返回值不能赋值给变量，但是可以用于switch

```
#define TRY do{ jmp_buf ex_buf__; if( !setjmp(ex_buf__) ){
#define CATCH } else {
#define ETRY } }while(0)
#define THROW longjmp(ex_buf__, 1)

int main(int argc, char** argv) {
    TRY {
        printf("In Try Statement\n");
        THROW;
        printf("I do not appear\n");
    }
    CATCH
    {
      printf("Got Exception!\n");
    }
    ETRY;
    return 0;
}
```

3.4 轮流回答问题

- 下面关于非局部跳转的描述，正确的是

A. setjmp可以和siglongjmp使用同一个jmp_buf变量

B. setjmp必须放在main()函数中调用

C. 虽然 longjmp 通常不会出错，但仍然需要对其返回值进行出错判断

D. 在同一个函数中既可以出现setjmp，也可以出现longjmp

D

any questions?

Thanks & 感谢观看