

异常控制流 01

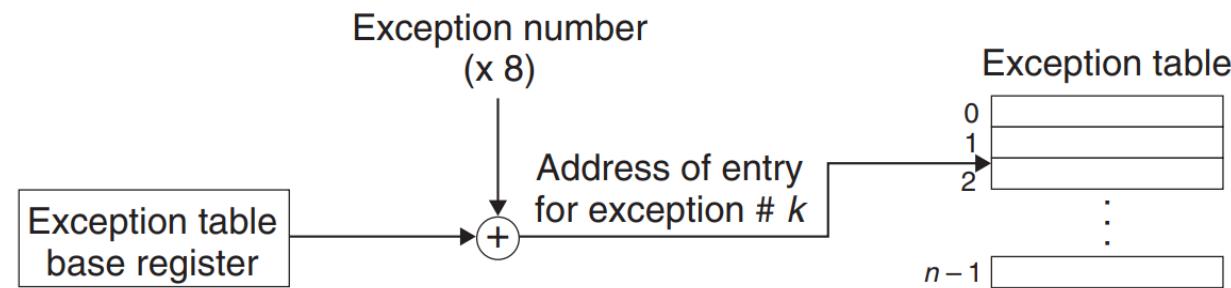
Nov 22, 2023

异常(Exception)

- 异常(Exception): 控制流中的突变
 - 注意: 异常 \neq 出错. 异常也可能是为了实现某些功能(中断/陷阱)
- 异常号: 非负整数, 每个
- 异常表: 系统启动时分配的跳转表. 表目k包含异常k的处理程序地址.
 - 异常表基址寄存器: 特殊的CPU寄存器, 存放异常表的起始地址

Figure 8.3

Generating the address
of an exception handler.
The exception number is
an index into the exception
table.



异常(Exception)

- 异常的分类: 中断 / 陷阱 / 故障 / 终止
 - 中断: 外部I/O设备
 - 陷阱(同步): 故意引发的异常, 目的是进行系统调用
 - 故障(同步): 出错了, 有可能修复. 例: 缺页, 除法错误, 一般保护故障(段错误)
 - 终止(同步): 致命错误, 无法恢复. 例: DRAM/SRAM损坏
- 注意区分: 同步/异步? 返回行为?

Class	Cause	Async/sync	Return behavior
Interrupt	Signal from I/O device	Async	Always returns to next instruction
Trap	Intentional exception	Sync	Always returns to next instruction
Fault	Potentially recoverable error	Sync	Might return to current instruction
Abort	Nonrecoverable error	Sync	Never returns

Figure 8.4 Classes of exceptions. Asynchronous exceptions occur as a result of events in I/O devices that are external to the processor. Synchronous exceptions occur as a direct result of executing an instruction.

系统调用

- 每个系统调用有一个编号, 对应内核中的一张跳转表中的条目 (不是异常表)
- 触发指令: `syscall` -> 陷入内核
 - `%rax`存放系统调用号/返回值, `%rdi %rsi %rdx %r10 %r8 %r9`存放参数

Number	Name	Description	Number	Name	Description
0	<code>read</code>	Read file	33	<code>pause</code>	Suspend process until signal arrives
1	<code>write</code>	Write file	37	<code>alarm</code>	Schedule delivery of alarm signal
2	<code>open</code>	Open file	39	<code>getpid</code>	Get process ID
3	<code>close</code>	Close file	57	<code>fork</code>	Create process
4	<code>stat</code>	Get info about file	59	<code>execve</code>	Execute a program
9	<code>mmap</code>	Map memory page to file	60	<code>_exit</code>	Terminate process
12	<code>brk</code>	Reset the top of the heap	61	<code>wait4</code>	Wait for a process to terminate
32	<code>dup2</code>	Copy file descriptor	62	<code>kill</code>	Send signal to a process

Figure 8.10 Examples of popular system calls in Linux x86-64 systems.

进程

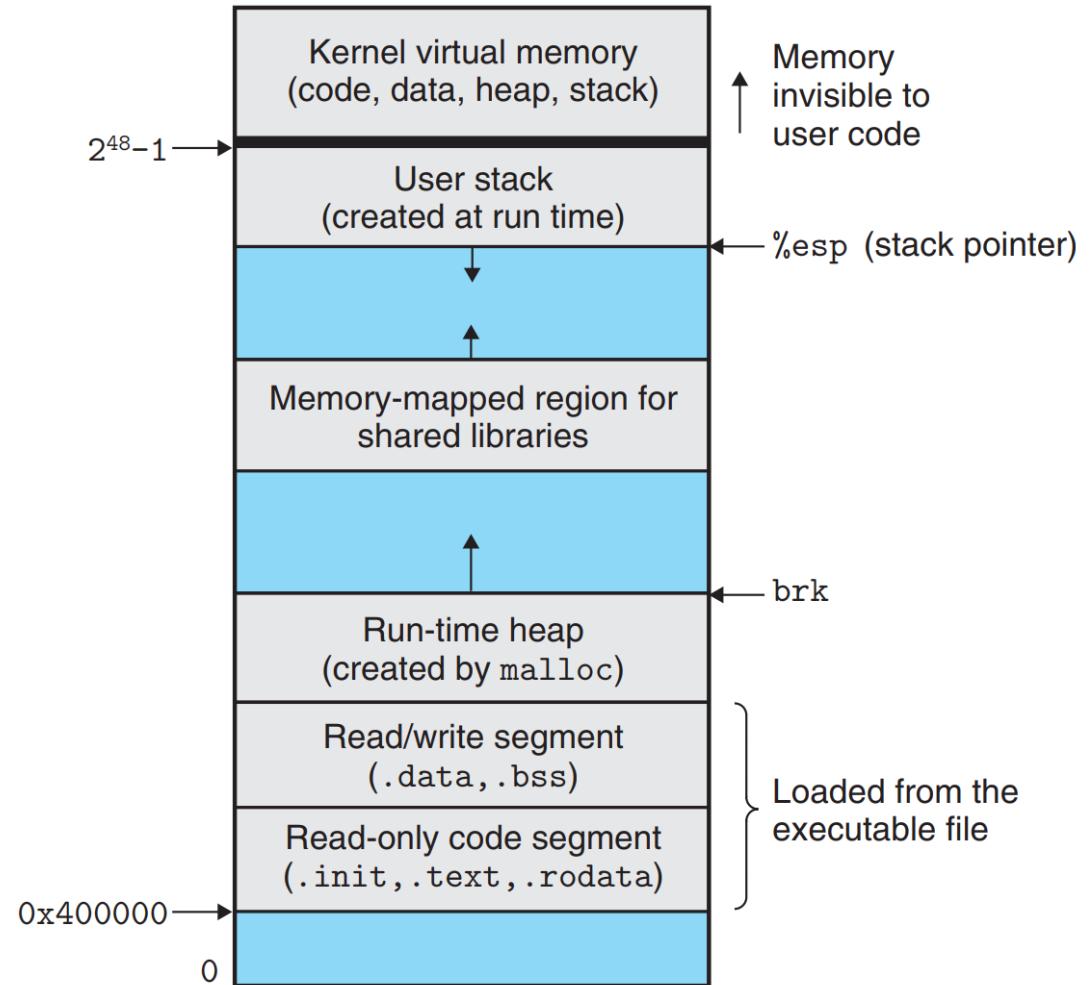
- 定义: 执行中的程序实例
 - 作用: 隔离不同程序, **让每个程序都认为系统中只有自己在运行**
 - 运行一个新的exe = 创建一个新的进程
- 进程间相互独立:
 - 独立的逻辑控制流
 - 私有的地址空间
- 每个进程都有一个上下文
 - 上下文: 程序运行所需的相关状态
 - 栈 / 寄存器 / 程序计数器 / 环境变量 / 打开的文件描述符

进程

- 逻辑控制流: 不同进程轮流使用CPU
 - 进程A占用CPU一段时间之后, 被进程B抢占(preempt)
 - 接下来运行进程B
- 上下文切换
 - 保存当前进程A的上下文(寄存器/PC等)
 - 恢复已保存的之前被抢占的进程B的状态
 - 将控制传递给进程B
- 发生上下文切换的例子
 - 系统调用 (例: read读盘 / sleep主动休眠)
 - 中断 (例: 周期性定时器中断 / 磁盘读取完成)

进程

- 并发 / 并行
 - 并发: 多个进程轮流在同一个核上运行
 - 并行: 同一时刻, 多个进程在不同核上运行
- 私有地址空间
 - 用户: $0x400000 \sim 2^{48}-1$
 - 内核: $\geq 2^{48}$ (注: 此处课本不准确, 可参考:
<https://zh.m.wikipedia.org/wiki/X86-64>)
 - 用户栈从高向低增长, 堆从低向高增长
- 用户模式 / 内核模式(超级用户模式)
 - 由模式位控制
 - 用户模式通过系统调用间接访问内核
 - /proc文件系统



进程控制

- 同一时刻, 操作系统中有若干个进程
- 进程的三种状态
 - 运行
 - 同一时间可以有若干个进程同时运行
 - 运行≠正在CPU上执行
 - 也可能是在等待调度的队列中
 - 停止: 进程被挂起, 且不会进入等待调度的队列
 - 收到以下4种信号会导致进程停止: SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU
 - 收到SIGCONT后被转为运行状态
 - 终止: 进程永不运行, 不能被转为运行状态
 - 可能原因: ①收到相关信号 ②从主程序返回 ③调用exit函数
 - 一个进程终止后必须被回收

进程控制

- PID (process ID): 每个进程有唯一的pid, 正数
 - getpid(): 返回当前进程的pid
 - getppid(): 返回父进程的pid
- ★ fork函数
 - 作用: 将原来的进程复制一份 (系统中多出了一个独立的进程)
 - 原进程称为父进程, 新的进程称为子进程
 - 此时两个进程完全相同: 地址空间, 堆栈, 变量值, 代码, 打开的文件
 - 在父进程中, fork()的返回值为子进程的pid
 - 在子进程中, fork()的返回值为0 (从而可以区分父进程和子进程)

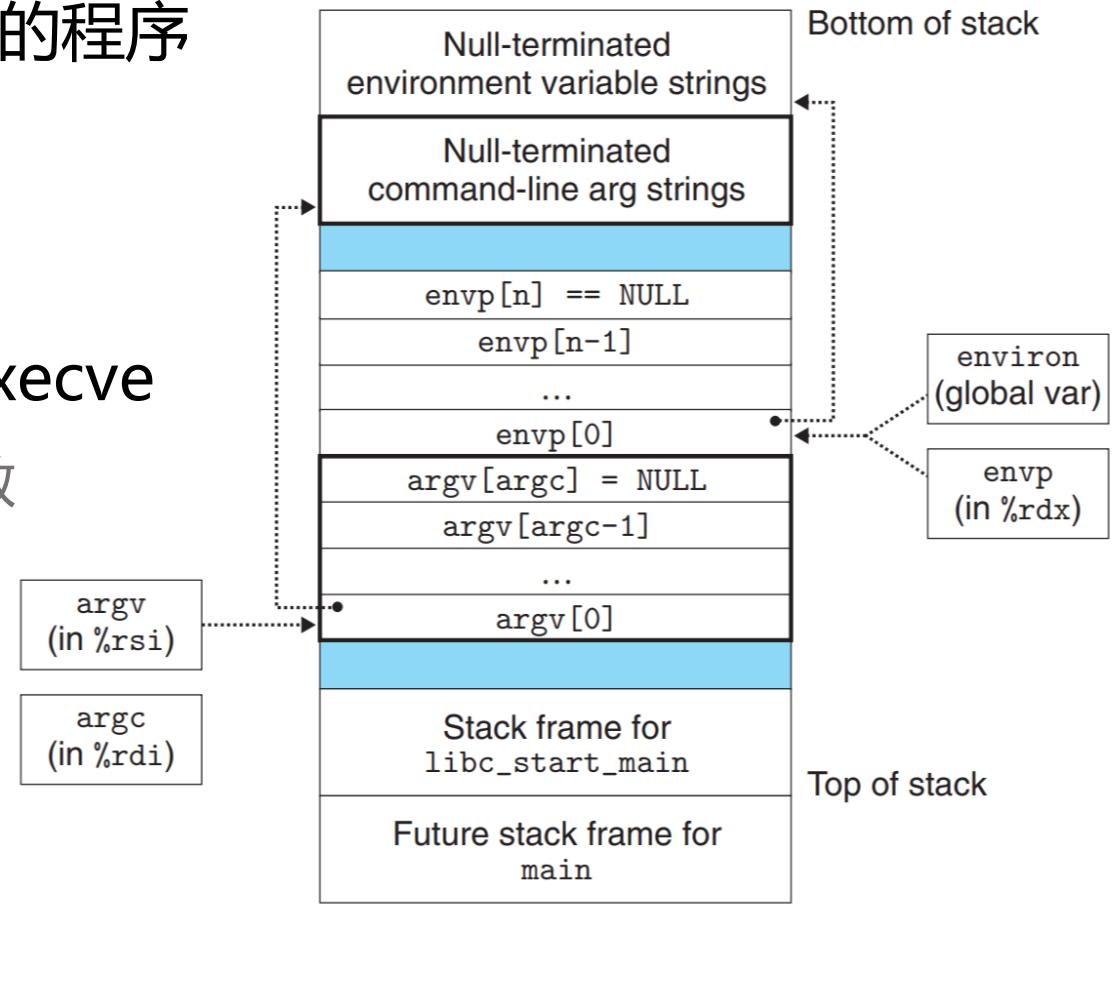
进程控制

- 一个进程终止后必须被其父进程回收
 - 如果父进程已经终止, 则安排init进程作为养父
 - init进程: pid=1, 系统启动由内核创建的第一个进程
 - 所有的进程都是它衍生出来的
- waitpid(): 父进程调用waitpid()等待其子进程终止
 - 注意3个参数的具体含义和使用
 - pid: 表明等待集合包含哪个/哪些子进程
 - statusp: 若放入一个指针, 则waitpid返回后会把子进程的相关状态信息存储在该指针指向的位置
 - options: 修改等待的具体行为

进程控制

- execve: 在当前进程的上下文中加载并运行一个新程序
 - execve运行后, 整个进程彻底变成新的程序
 - 注意参数和环境变量的放置
- 常见操作:
fork出一个子进程, 然后在子进程中调用execve

```
if(Fork() == 0) { //大写字母开头表示包装函数  
    // 子进程  
}  
  
else {  
    //父进程  
}
```



例

7. 关于 x86-64 系统中的异常，下面那个判断是正确的：
- A. 除法错误是异步异常，Unix 会终止程序；
 - B. 键盘输入中断是异步异常，异常服务后会返回当前指令执行；
 - C. 缺页是同步异常，异常服务后会返回当前指令执行；
 - D. 时间片到时中断是同步异常，异常服务后会返回下一条指令执行；

例

7. 关于 x86-64 系统中的异常，下面那个判断是正确的：
- A. 除法错误是异步异常，Unix 会终止程序；
 - B. 键盘输入中断是异步异常，异常服务后会返回当前指令执行；
 - C. 缺页是同步异常，异常服务后会返回当前指令执行；
 - D. 时间片到时中断是同步异常，异常服务后会返回下一条指令执行；
7. C。除法错误是不可恢复故障，是同步的；I/O 中断是异步的，一般会执行完当前指令，再去处理，返回就不需要再处理了；缺页异常当然需要重新执行遇到问题的访存指令；时间片到中断属于时钟中断，属于异步异常。

例

13. 对于 Linux 系统，下列说法错误的是：
- A. 在单核 CPU 上，所有看似并行的逻辑流实际上是并发的。
 - B. 用户模式不可访问/proc 文件系统中包含的内核数据结构的内容。
 - C. 在 execve 函数参数的 argv 数组加入”> 1.txt”，不能自动实现 IO 重定向。
 - D. 即便在信号处理程序中调用 printf 前阻塞所有信号，也不一定安全。

例

13. 对于 Linux 系统，下列说法错误的是：
- A. 在单核 CPU 上，所有看似并行的逻辑流实际上是并发的。
 - B. 用户模式不可访问/proc 文件系统中包含的内核数据结构的内容。
 - C. 在 `execve` 函数参数的 `argv` 数组加入”> 1.txt”，不能自动实现 IO 重定向。
 - D. 即便在信号处理程序中调用 `printf` 前阻塞所有信号，也不一定安全。
13. B。A 显然是正确的。B 我们实验过，一般这种读取对于用户来说是允许的，参看书 P511。C 一般同学不会实验过，需要推理一下，因为参数列表和环境变量实际上是程序自己处理的，而且 shell lab 中重定向是 shell 帮忙完成的，故推测 C 正确。D 是陆老师著名的“灵魂出窍”例子，即 `printf` 在信号处理程序中可能导致死锁。

例

下列行为分别触发什么样的异常? 对应的后续行为是什么?

1. 执行指令`mov $57, %eax; syscall`
2. 程序执行过程中, 发现它所使用的物理内存损坏了
3. 程序执行过程中, 试图往`main`函数的内存中写入数据
4. 按下键盘`Ctrl+Z`
5. 按下键盘`Ctrl+C`
6. 磁盘读完成
7. 用`read`函数发起磁盘读
8. 用户程序执行了一个只能在内核模式下执行的指令

例

下列行为分别触发什么样的异常? 对应的后续行为是什么?

1. 执行指令`mov $57, %eax; syscall`
2. 程序执行过程中, 发现它所使用的物理内存损坏了
3. 程序执行过程中, 试图往`main`函数的内存中写入数据
4. 按下键盘`Ctrl+Z`
5. 按下键盘`Ctrl+C`
6. 磁盘读完成
7. 用`read`函数发起磁盘读
8. 用户程序执行了一个只能在内核模式下执行的指令

1-4 陷阱 终止 故障 中断 5-8 中断 中断 陷阱 故障

例

下面代码一共输出几个A?

```
fork() || fork();  
printf( "A\n" );  
exit(0);
```

例

下面代码一共输出几个A?

```
fork() || fork();
printf( "A\n" );
exit(0);
```

3个.

如果第一个fork()确定返回值为1, 则第二个fork()不会被运行

例

9. C 语言中的代码如下：

```
fork() && fork();
printf("-");
fork() || fork();
printf("-");
```

这段代码一共输出（ ）个“-”字符。

- A. 12
- B. 18
- C. 20
- D. 32

例

9. C 语言中的代码如下：

```
fork() && fork();
printf("-");
fork() || fork();
printf("-");
```

这段代码一共输出（ ）个“-”字符。

- A. 12
- B. 18
- C. 20
- D. 32

9. B。注意 `printf` 是行缓冲，缓冲区会被复制，最后都一次性输出。另外 `fork() && fork()` 是父进程再 `fork()`，子进程不 `fork()`，而 `fork() || fork()` 则是子进程再 `fork()`，父进程不 `fork()`。故实质上一次产生 3 个进程，所以答案为 $3^2 \times 2 = 18$ 。（如果不考虑缓冲区则要去掉第二次产生的 6 个净拷贝，答案应为 12。）

例

阅读右边的程序，下列哪些输出是可能的？

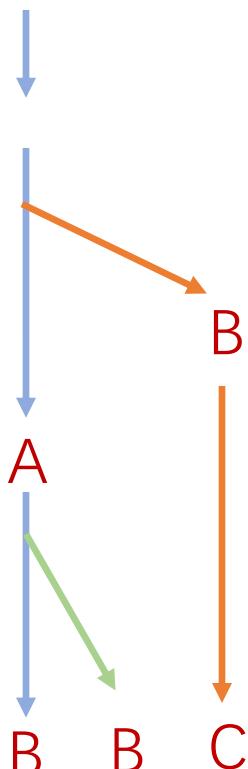
1. ABBCCD
2. ABBCDC
3. ABBDCC
4. ABDBCC
5. ABCDBC
6. ABCDCB

```
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);
}
```

例

阅读右边的程序，下列哪些输出是可能的？

1. ABBCCD
 2. ABBCDC
 3. ABBDCC
 4. ABDBCC
 5. ABCDBC
 6. ABCDCB
- YYNNNNN



```
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);  
}
```

Thank you!