第8章 异常控制流

> 通俗的理解：

- ECF（Exceptional Control Flow）就是在正常指令进行（控制流control flow)的过程中应对突发变化做出的反应。

> Exceptions：

- Instruction curr --Exception--> Exception processing --> I curr / I next / Abort
- Type:

类别	原因	异步同步	返回行为
Interrupt	来自IO设备的信号	Async	返回到下一条指令
Trap	有意的异常	Sync	返回到下一条指令
Fault	潜在可恢复的错误	Sync	可能返回到当前指令
Abort	不可恢复的错误	Sync	不返回

  - Interrupt: 网络适配器、磁盘控制器、定时器芯片等，中断引脚触发中断
  - Trap：syscall(read,fork,execve,exit,etc.)
    %rax包含系统调用号
    参数寄存器传递，而非栈传递，最多6个
    返回时%rcx,%r11被破坏，%rax包含返回值，负数返回值对应errno

    > System Call Error Handling error-handling wrappers:

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

pid_t Fork(void)
{
  pid_t pid;

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

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

  - Fault: 缺页异常（page fault exception）
  - Abort: DRAM或SRAM位损坏时发生奇偶校验错误

- 常见异常：

Exception number	Description	Exception Class
0	除法错误（通常选择终止程序）	Fault
13	一般保护故障（Segmentation Fault）	Fault
14	缺页	Fault
18	Machine check（机器故障）	Abort
32-255	操作系统定义的异常	Interrupt or Trap

Processes

- 两个关键抽象：
  1. 一个独立的逻辑控制流
  2. 一个私有的地址空间
- 区分两个概念：
  1. 并发(concurrency)：time slice有交叉
  2. 并行(parallel)：多核处理和多机协同
- 理解：
  1. User Mode & Kernel Mode
  2. Context Switches
- Process Control:
  1. 获取进程ID

```
#include <sys/types.h> //定义了pid_t为int
#include <unistd.h>

pid_t getpid(void);//返回当前进程PID
pid_t getppid(void);//返回父进程PID
```

  2. 创建和终止进程

```
#include <stdlib.h>
void exit(int status);//不返回，有退出状态
```

```
#include <sys/types.h>
#include <unistd.h>
//调用一次，返回两次；子进程返回0，父进程返回子进程的PID，出错返回-1
//并发执行，相同但独立的地址空间，共享文件
pid_t fork(void);
```

3. 回收子进程

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

//pid>0 => 等待这一个子进程
//pid=-1 => 等待所有子进程

//默认options=0,等待一个子进程终止就返回，返回已终止子进程的PID
//options=WNOHANG => 没有可回收子进程就立即返回
//options=WUNTRACED => 等待一个terminated or stopped子进程
//options=WCONTINUED => 等待一个terminated子进程 or SIGCONT继续一个stopped子进程

//WIFEXITED(status) => exit或者return正常终止，则返回true
  //WEXITSTATUS(status) => 返回退出状态
//WIFSIGNALED(status) => 因未捕获的信号终止，则返回true
  //WTERMSIG(status) => 返回导致终止的信号编号
//WIFSTOPPED(status) => 子进程当前是STOPPED，则返回true
  //WSTOPSIG(status) => 返回引起停止的信号编号
//WIFCONTINUED(status) => 子进程收到SIGCONT重新启动，则返回真

//没有子进程，则返回-1，设置errno为ECHILD
//被信号中断，则返回-1，设置errno为EINTR
pid_t waitpid(pid_t pid, int *statusp, int options)

//wait(&status)等价于waitpid(-1, &status, 0)
pid_t wait(int *statusp);
```

4. 让进程休眠

```
#include <unistd.h>
//时间到返回0，否则返回剩下的秒数
unsigned int sleep(unsigned int secs);
//休眠直到收到信号，总是返回-1
int pause(void);
```

5. 加载并运行程序

```
#include <unistd.h>
//成功不返回，错误返回-1
//argv和envp都是以null结尾的指针数组，其中每个指针指向栈中的一个命令行/环境变量字符串
//全局变量environ指向envp[0]
int execve(const char *filename, const char *argv[], const char *envp[]);
```

```
#include <stdlib.h>

//在环境变量数组中搜索“name=value”，若存在则返回指向value的指针，否则返回NULL
char *getenv(const char *name);
//成功返回0，失败返回-1.
//若存在且overwrite非零则覆盖重写，不存在则添加
int setenv(const char *name, const char *newvalue, int overwrite);
//不返回，删除对应字符串
void unsetenv(const char *name);
```

Signals

- Sending Signals
  - 一个进程可以发送信号给自己
  - process group:

```
#include <unistd.h>
//返回调用进程的进程组ID
pid_t getpgrp(void);
//将进程pid的进程组改为pgid
//pid=0 => 使用当前进程pid
//pgid=0 => 使用pid作为进程组ID
//成功返回0，失败返回-1
int setpgid(pid_t pid, pid_t pgid);
```

  - 使用程序

    `linux> /bin/kill -9 15213 //发给进程 linux> /bin/kill -9 -15213 //发给进程组的每个进程`
  - 键盘发送
    - Ctrl+C 发送SIGINT到前台进程组的每个进程，终止作业
    - Ctrl+Z 发送SIGTSTP到前台进程组的每个进程，停止（挂起suspend）作业
  - kill函数

```
#include <sys/types.h>
#include <signal.h>
//pid=0 => 发送给调用进程所在进程组的每个进程，包括自己
//pid<0 => 发送给进程组中每个进程
//pid>0 => 发送给某个进程
//成功返回0，错误返回-1
int kill(pid_t pid, int sig);
```

  - alarm函数

```
#include <unistd.h>
//向自己发送SIGALRM信号
```

```
//返回前一次闹钟剩余的秒数，若以前没有设定闹钟就返回0
//secs=0 => 不安排新的闹钟
//任何调用都将取消待处理（pending）的闹钟
unsigned int alarm(unsigned int secs);
```

- Receiving Signals

```
#include <signal.h>
typedef void (*sighandler_t)(int);
//可以修改信号相关联的默认行为，SIGSTOP和SIGKILL不可修改
//handler=SIG_IGN => 忽略signum的信号
//handler=SIG_DFL => 恢复signum的默认行为

sighandler_t signal(int signum, sighandler_t handler);
```

- Blocking and Unblocking Signals

```
#include <signal.h>
//how=SIG_BLOCK => blocked=blocked | set
//how=SIG_UNBLOCK => blocked=blocked & ~set
//how=SIG_SETMASK => block=set
//oldset非空 => blocked位向量之前的值保存在oldset中
//成功返回0，出错返回-1
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);

//signum是set的成员返回1，不是返回0，出错返回-1
int sigismember(const sigset_t *set, int signum);
```

- Signal Handlers
  - Safe：

    1. G0.处理程序要尽可能简单
    2. G1.在处理程序中只调用异步信号安全（async-signal-safe）的函数

异步信号不安全的	异步信号安全的
printf sprintf	write sio_putl sio_puts sio_error
malloc	--
exit	_exit
--	fork execve waitpid signal

    3. G2.保存和恢复errno（处理程序要返回时才有必要，_exit终止就不需要了）

4. G3.阻塞所有的信号，保护对共享全局数据结构的访问（也即保证事务的原子性）
5. G4.用volatile声明全局变量（也即保证数据的一致性，不要缓存，必须每次从内存读取）
6. G5.用sig_atomic_t声明flags `volatile sig_atomic_t flag;`

- Correct:

1. 信号不会排队等待，信号block后，后来的pending信号被简单丢弃

- Portable：

1. signal的函数语义各有不同
2. 系统调用可以被中断
   解决方案：Posix标准

```
#include <signal.h>

//成功返回0，出错返回-1
int sigaction(int signum, struct sigaction *act, struct sigaction
*oldact);
```

- Concurrency Bugs：
  - race: 并发交错使得有时先delete后add，有时先add后delete
  - 运用显式block达到Synchronizing Flows来避免这种问题
- Explicitly Waiting:
  1. Wasteful `while(!pid);`
  2. Race

```
while(!pid)
  pause();
```

3. Too slow

```
while(!pid)
  sleep(1);
```

4. Proper

```
#include <signal.h>
//返回-1
//暂时用mask替换当前阻塞集合，然后挂起该进程，直到收到一个信号。
//若信号行为终止，不返回就直接终止
//若信号行为运行处理程序，从处理程序返回，恢复调用原有的阻塞集合
int sigsuspend(const sigset_t *mask);
```

Nonlocal Jumps

```
#include <setjmp.h>

//自己调用返回0，longjump调用返回非零
int setjmp(jmp_buf env);
//sig-是可以被信号处理程序使用的版本，savesigs表明是否保存信号到上下文
int sigsetjmp(sigjmp_buf env, int savesigs);

//retval设置了到setjmp的返回值
void longjmp(jmp_buf env, int retval);
void siglongjmp(sigjmp_buf env, int retval);
```

Tools:

- STRACE: 打印一个正在运行的程序和它的子进程调用的每个系统调用的轨迹。
- PS： 列出系统中的进程
- /proc： 一个虚拟文件系统