Linux进程信号（上）

本节重点：

1. 掌握Linux信号的基本概念。
2. 掌握信号产生的一般方式。
3. 理解信号递达和阻塞的概念，原理。
4. 掌握信号捕捉的一般方式。
5. 了解中断过程，理解中断的意义
6. 掌握操作系统运行，系统调用原理，理解缺页异常或其他软件异常的基本原理
7. 重新了解可重入函数的概念。
8. 了解竞态条件的情景和处理方式。
9. 了解SIGCHLD信号， 重新编写信号处理函数的一般处理机制。

1. 信号快速认识

1.1 生活角度的信号

· 你在网上买了很多件商品，再等待不同商品快递的到来。但即便快递没有到来，你也知道快递来临时，你该怎么处理快递。也就是你能“识别快递”

· 当快递员到了你楼下，你也收到快递到来的通知，但是你正在打游戏，需5min之后才能去取快递。那么在在这5min之内，你并没有下去去取快递，但是你是知道有快递到来了。也就是取快递的行为并不是一定要立即执行，可以理解成“在合适的时候去取”。

· 在收到通知，再到你拿到快递期间，是有一个时间窗口的，在这段时间，你并没有拿到快递，但是你知道有一个快递已经来了。本质上是你“记住了有一个快递要去取”

· 当你时间合适，顺利拿到快递之后，就要开始处理快递了。而处理快递一般方式有三种：

1. 执行默认动作（幸福的打开快递，使用商品）
2. 执行自定义动作（快递是零食，你要送给你你的女朋友）
3. 忽略快递（快递拿上来之后，扔掉床头，继续开一把游戏）

· 快递到来的整个过程，对你来讲是异步的，你不能准确断定快递员什么时候给你打电话

🎉 📌 基本结论：

· 你怎么能识别信号呢？识别信号是内置的，进程识别信号，是内核程序员写的内置特性。

· 信号产生之后，你知道怎么处理吗？知道。如果信号没有产生，你知道怎么处理信号吗？知道。所以，信号的处理方法，在信号产生之前，已经准备好了。

· 处理信号，立即处理吗？我可能正在做优先级更高的事情，不会立即处理？什么时候？合适的时候。

· 信号到来 | 信号保存 | 信号处理

· 怎么进行信号处理啊？a.默认 b.忽略 c.自定义， 后续都叫做信号捕捉。

1.2 技术应用角度的信号

1.2.1 一个样例

代码块

```
// sig.cc
#include <iostream>
#include <unistd.h>
int main()
{
    while(true)
    {
        std::cout << "I am a process, I am waiting signal!" << std::endl;
        sleep(1);
    }
}

$ g++ sig.cc -o sig
$ ./sig
I am a process, I am waiting signal!
I am a process, I am waiting signal!
^C
```

· 用户输入命令,在Shell下启动一个前台进程

· 用户按下 Ctrl+C ,这个键盘输入产生一个硬件中断，被OS获取，解释成信号，发送给目标前台进程

· 前台进程因为收到信号，进而引起进程退出

1.2.2 一个系统函数

```
NAME
    signal - ANSI C signal handling
SYNOPSIS
    #include <signal.h>
    typedef void (*sighandler_t)(int);
    sighandler_t signal(int signum, sighandler_t handler);

参数说明:
signum: 信号编号[后面解释，只需要知道是数字即可]
handler: 函数指针，表示更改信号的处理动作，当收到对应的信号，就回调执行handler方法
```

而其实， Ctrl+C 的本质是向前台进程发送 SIGINT 即 2 号信号，我们证明一下，这里需要引入一个系统调用函数

开始测试

代码块

```
#include <iostream>
#include <unistd.h>
#include <signal.h>

void handler(int signumber)
{
    std::cout << "我是: " << getpid() << ", 我获得了一个信号: " << signumber <<
    std::endl;
}

int main()
{
    std::cout << "我是进程: " << getpid() << std::endl;
    signal(SIGINT/*2*/, handler);
    while(true)
    {
        std::cout << "I am a process, I am waiting signal!" << std::endl;
        sleep(1);
    }
}

$ g++ sig.cc -o sig
$ ./sig
我是进程: 212569
I am a process, I am waiting signal!
I am a process, I am waiting signal!
^C我是: 212569, 我获得了一个信号: 2
I am a process, I am waiting signal!
```

```
I am a process, I am waiting signal!
^C我是: 212569, 我获得了一个信号: 2
I am a process, I am waiting signal!
I am a process, I am waiting signal!
```

🏝️ 📌 思考:

· 这里进程为什么不退出?

· 这个例子能说明哪些问题? 信号处理，是自己处理

· 请将生活例子和 Ctrl-C 信号处理过程相结合，解释一下信号处理过程? 进程就是你，操作系统就是快递员，信号就是快递，发信号的过程就类似给你打电话

🎹 📌 注意

· 要注意的是，signal函数仅仅是设置了特定信号的捕捉行为处理方式，并不是直接调用处理动作。如果后续特定信号没有产生，设置的捕捉函数永远也不会被调用！！

· Ctrl-C 产生的信号只能发给前台进程。一个命令后面加个&可以放到后台运行,这样Shell不必等待进程结束就可以接受新的命令,启动新的进程。

· Shell可以同时运行一个前台进程和任意多个后台进程,只有前台进程才能接到像 Ctrl-C 这种控制键产生的信号。

· 前台进程在运行过程中用户随时可能按下 Ctrl-C 而产生一个信号,也就是说该进程的用户空间代码执行到任何地方都有可能收到 SIGINT 信号而终止,所以信号相对于进程的控制流程来说是异步(Asynchronous)的。

· 关于进程间关系，我们在网络部分会专门来讲，现在就了解即可。

· 可以渗透 & 和 nohup

1.3 信号概念

> 信号是进程之间事件异步通知的一种方式，属于软中断。

```
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

> 每个信号都有一个编号和一个宏定义名称,这些宏定义可以在signal.h中找到,例如其中有定义
>
> #define SIGINT 2

```
/* ISO C99 signals.  */
#define SIGINT      2  /* Interactive attention signal.  */
#define SIGILL      4  /* Illegal instruction.  */
#define SIGABRT     6  /* Abnormal termination.  */
#define SIGFPE      8  /* Erroneous arithmetic operation.  */
#define SIGSEGV     11 /* Invalid access to storage.  */
#define SIGTERM     15 /* Termination request.  */

/* Historical signals specified by POSIX. */
#define SIGHUP      1  /* Hangup.  */
#define SIGQUIT     3  /* Quit.  */
#define SIGTRAP     5  /* Trace/breakpoint trap.  */
#define SIGKILL     9  /* Killed.  */
#define SIGBUS    10 /* Bus error.  */
#define SIGSYS      12 /* Bad system call.  */
#define SIGPIPE     13 /* Broken pipe.  */
#define SIGALRM     14 /* Alarm clock.  */

/* New(er) POSIX signals (1003.1-2008, 1003.1-2013).  */
#define SIGURG      16 /* Urgent data is available at a socket.  */
#define SIGSTOP     17 /* Stop, unblockable.  */
#define SIGTSTP     18 /* Keyboard stop.  */
#define SIGCONT     19 /* Continue.  */
#define SIGCHLD     20 /* Child terminated or stopped.  */
#define SIGTTIN     21 /* Background read from control terminal.  */
#define SIGTTOU     22 /* Background write to control terminal.  */
#define SIGPOLL     23 /* Pollable event occurred (System V).  */
#define SIGXCPU     24 /* CPU time limit exceeded.  */
#define SIGXFSZ     25 /* File size limit exceeded.  */
#define SIGVTALRM   26 /* Virtual timer expired.  */
#define SIGPROF     27 /* Profiling timer expired.  */
#define SIGUSR1     30 /* User-defined signal 1.  */
#define SIGUSR2     31 /* User-defined signal 2.  */
```

编号34以上的是实时信号,本章只讨论编号34以下的信号,不讨论实时信号。这些信号各自在什么条件下产生,默认的处理动作是什么,在signal(7)中都有详细说明: `man 7 signal`

```
Signal dispositions
   Each signal has a current disposition, which determines how the process behaves when it is delivered the signal.

   The  entries  in  the "Action" column of the table below specify the default disposition for each signal, as fol-
   lows:

   Term   Default action is to terminate the process.

   Ign    Default action is to ignore the signal.

   Core   Default action is to terminate the process and dump core (see core(5)).

   Stop   Default action is to stop the process.

   Cont   Default action is to continue the process if it is currently stopped.

   A process can change the disposition of a signal using sigaction(2) or signal(2).  (The latter is  less  portable
   when  establishing  a  signal handler; see signal(2) for details.)  Using these system calls, a process can elect
   one of the following behaviors to occur on delivery of the signal: perform the default action; ignore the signal;
   or  catch  the signal with a signal handler, a programmer-defined function that is automatically invoked when the
   signal is delivered.

   By default, a signal handler is invoked on the normal process stack.  It is possible to arrange that  the  signal
   handler  uses an alternate stack; see sigaltstack(2) for a discussion of how to do this and when it might be use-
   ful.

   The signal disposition is a per-process attribute: in a multithreaded application, the disposition of a  particu-
   lar signal is the same for all threads.

   A child created via fork(2) inherits a copy of its parent's signal dispositions.  During an execve(2), the dispo-
   sitions of handled signals are reset to the default; the dispositions of ignored signals are left unchanged.
```

1.3.1 信号处理

`sigaction` 函数稍后详细介绍，可选的处理动作有以下三种:

· 忽略此信号

代码块

```cpp
#include <iostream>
1#include <unistd.h>
#include <signal.h>
void handler(int signumber)
{
    std::cout << "我是: " << getpid() << ", 我获得了一个信号: " << signumber
    << std::endl;
}
int main()
{
    std::cout << "我是进程: " << getpid() << std::endl;
    signal(SIGINT/*2*/, SIG_IGN); // 设置忽略信号的宏
    while(true)
    {
        std::cout << "I am a process, I am waiting signal!" << std::endl;
        sleep(1);
    }
}

```

```
$ g++ sig.cc -o sig
$ ./sig
我是进程: 212681
I am a process, I am waiting signal!
I am a process, I am waiting signal!
^C^C^C^C^C^CI am a process, I am waiting signal! // 输入ctrl+c毫无反应
I am a process, I am waiting signal!
```

· 执行该信号的默认处理动作。

代码块

```
#include <iostream>
#include <unistd.h>
#include <signal.h>
void handler(int signumber)
{
    std::cout << "我是: " << getpid() << ", 我获得了一个信号: " << signumber
    << std::endl;
}

int main()
{
    std::cout << "我是进程: " << getpid() << std::endl;
    signal(SIGINT/*2*/, SIG_DFL);
    while(true)
    {
        std::cout << "I am a process, I am waiting signal!" << std::endl;
        sleep(1);
    }
}

$ g++ sig.cc -o sig
$ ./sig
我是进程: 212791
I am a process, I am waiting signal!
I am a process, I am waiting signal!
^C          // 输入ctrl+c,进程退出，就是默认动作
```

· 提供一个信号处理函数,要求内核在处理该信号时切换到用户态执行这个处理函数,这种方式称为自定义捕捉(Catch)一个信号。

代码块

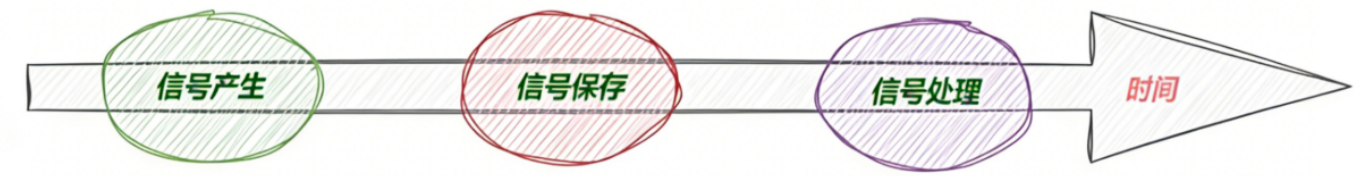
```
// 就是开始的样例
```

注意看源码：

代码块

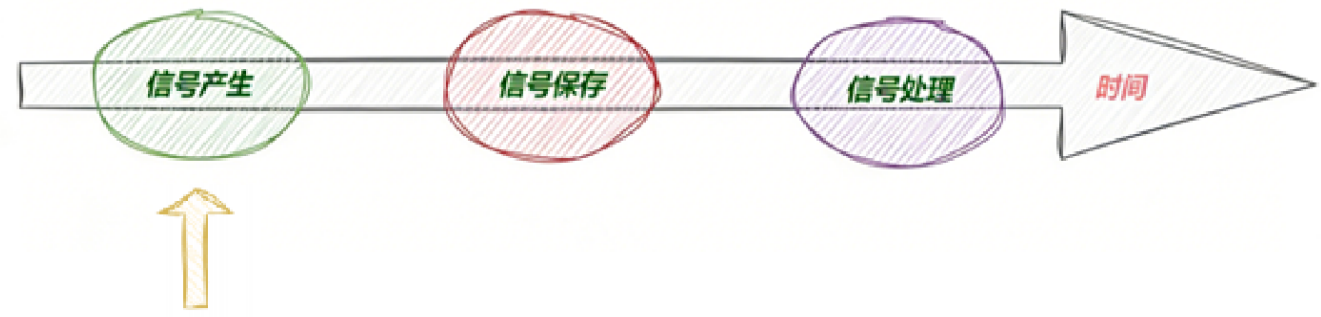
```
#define SIG_DFL ((__sighandler_t) 0) /* Default action. */
#define SIG_IGN ((__sighandler_t) 1) /* Ignore signal. */

/* Type of a signal handler. */
typedef void (*__sighandler_t) (int);

// 其实SIG_DFL和SIG_IGN就是把0,1强转为函数指针类型
```

上面的所有内容，我们都没有做非常多的解释，主要是先用起来，然后渗透部分概念和共识，下面我们从理论和实操两个层面，来进行对信号的详细学习、论证和理解。为了保证条理，我们采用如下思路来进行阐述：



2. 产生信号

当前阶段：



2.1 通过终端按键产生信号

2.1.1 基本操作

· Ctrl+C (SIGINT) 已经验证过，这里不再重复

· Ctrl+\（SIGQUIT）可以发送终止信号并生成core dump文件，用于事后调试（后面详谈）

代码块

```
#include <iostream>
#include <unistd.h>
#include <signal.h>

void handler(int signumber)
{
    std::cout << "我是: " << getpid() << ", 我获得了一个信号: " << signumber <<
    std::endl;
}

int main()
{
    std::cout << "我是进程: " << getpid() << std::endl;
    signal(SIGQUIT/*3*/, handler);
    while(true)
    {
        std::cout << "I am a process, I am waiting signal!" << std::endl;
        sleep(1);
    }
}

$ g++ sig.cc -o sig
$ ./sig
我是进程: 213056
I am a process, I am waiting signal!
I am a process, I am waiting signal!
I am a process, I am waiting signal!
^\我是: 213056, 我获得了一个信号: 3

// 注释掉13行代码
$ ./sig
我是进程: 213146
I am a process, I am waiting signal!
I am a process, I am waiting signal!
^\Quit

```

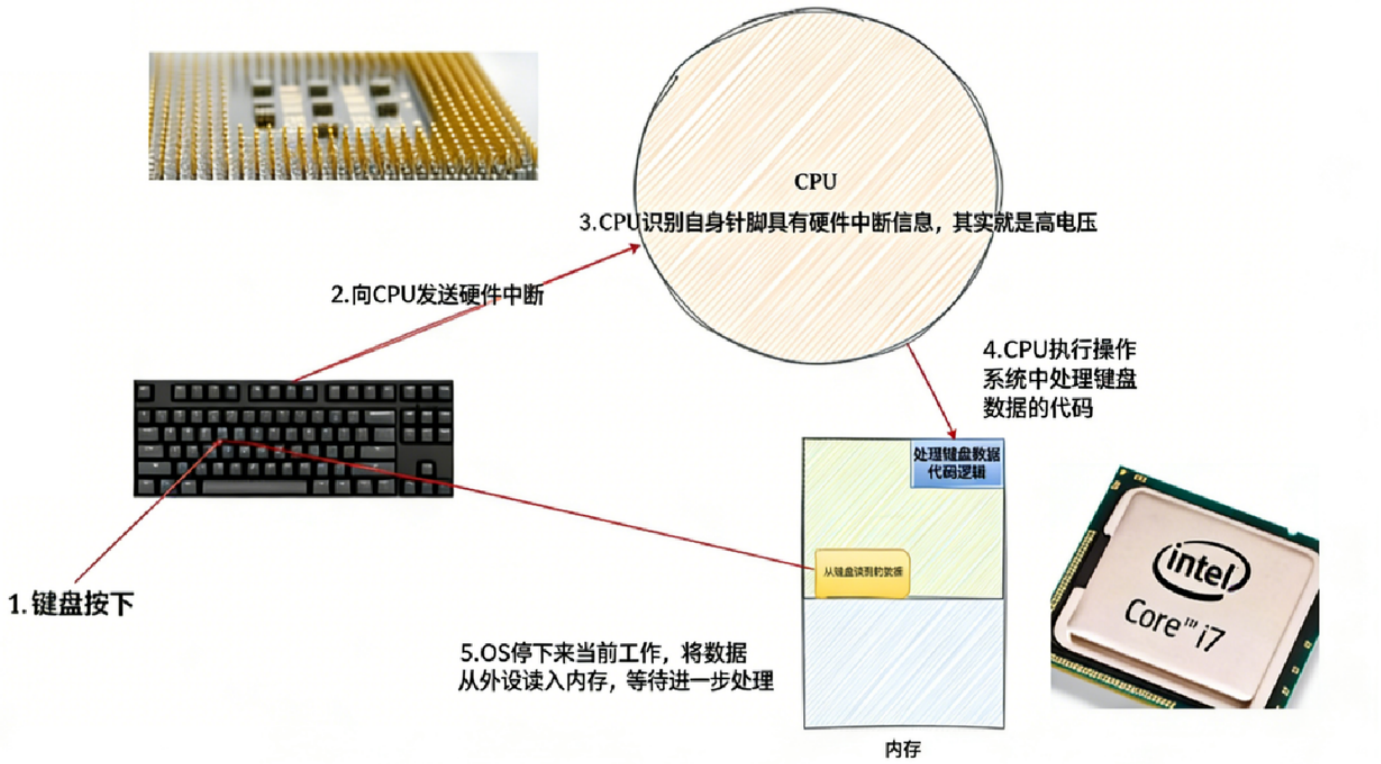
· Ctrl+Z（SIGTSTP）可以发送停止信号，将当前前台进程挂起到后台等。

代码块

```
#include <iostream>
#include <unistd.h>
```

```
#include <signal.h>
void handler(int signumber)
{
    std::cout << "我是: " << getpid() << ", 我获得了一个信号: " << signumber <<
    std::endl;
}

int main()
{
    std::cout << "我是进程: " << getpid() << std::endl;
    signal(SIGTSTP/*20*/, handler);
    while(true)
    {
        std::cout << "I am a process, I am waiting signal!" << std::endl;
        sleep(1);
    }
}

$ ./sig
我是进程: 213552
I am a process, I am waiting signal!
I am a process, I am waiting signal!
^Z我是: 213552, 我获得了一个信号: 20

// 注释掉13行代码
$ ./sig
我是进程: 213627
I am a process, I am waiting signal!
I am a process, I am waiting signal!
I am a process, I am waiting signal!
^Z
[1]+ Stopped ./sig
whb@bite:~/code/test$ jobs
[1]+ Stopped ./sig
```

2.1.2 理解OS如何得知键盘有数据



2.1.3 初步理解信号起源

> 👍 📌 注意：
>
> · 信号其实是从纯软件角度，模拟硬件中断的行为
>
> · 只不过硬件中断是发给CPU，而信号是发给进程
>
> · 两者有相似性，但是层级不同，这点我们后面的感觉会更加明显

2.2 调用系统命令向进程发信号

示例代码

代码块

```
#include <iostream>
#include <unistd.h>
#include <signal.h>
int main()
{
    while(true)
    {
        sleep(1);
    }
}

$ g++ sig.cc -o sig // step 1
```

```
$ ./sig & // step 2
$ ps ajx |head -1 && ps ajx | grep sig // step 3
   PPID     PID   PGID   SID  TTY       TPGID STAT   UID   TIME COMMAND
 211805 213784 213784 211805 pts/0    213792 S     1002   0:00 ./sig
```

> 首先在后台执行死循环程序,然后用kill命令给它发SIGSEGV信号。

代码块

```
$ kill -SIGSEGV 213784
$ // 多按一次回车
[1]+ Segmentation fault ./sig
```

> · 213784 是 sig 进程的pid。之所以要再次回车才显示 Segmentation fault ,是因为在213784 进程终止掉之前已经回到了Shell提示符等待用户输入下一条命令, Shell 不希望Segmentation fault 信息和用户的输入交错在一起,所以等用户输入命令之后才显示。
>
> · 指定发送某种信号的 kill 命令可以有多种写法,上面的命令还可以写成 kill -11213784 , 11 是信号 SIGSEGV 的编号。以往遇到的段错误都是由非法内存访问产生的,而这个程序本身没错,给它发 SIGSEGV也能产生段错误。

2.3 使用函数产生信号

2.3.1 kill

> kill 命令是调用 kill 函数实现的。 kill 函数可以给一个指定的进程发送指定的信号。

代码块

```
NAME
        kill - send signal to a process
SYNOPSIS
        #include <sys/types.h>
        #include <signal.h>

        int kill(pid_t pid, int sig);

RETURN VALUE
        On success (at least one signal was sent), zero is returned. On error,
        -1 is returned, and errno is set appropriately.
```

样例：实现自己的 kill 命令

代码块

```cpp
#include <iostream>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>

// mykill -signumber pid
int main(int argc, char *argv[])
{
    if(argc != 3)
    {
        std::cerr << "Usage: " << argv[0] << " -signumber pid" << std::endl;
        return 1;
    }

    int number = std::stoi(argv[1]+1); // 去掉-
    pid_t pid = std::stoi(argv[2]);

    int n = kill(pid, number);
    return n;
}
```

2.3.2 raise

> raise 函数可以给当前进程发送指定的信号(自己给自己发信号)。

代码块

```
NAME
        raise - send a signal to the caller

SYNOPSIS
        #include <signal.h>
        int raise(int sig);

RETURN VALUE
        raise() returns 0 on success, and nonzero for failure.
```

样例：

代码块

```cpp
#include <iostream>
#include <unistd.h>
#include <signal.h>

void handler(int signumber)
```

```
{
    // 整个代码就只有这一处打印
    std::cout << "获取了一个信号: " << signumber << std::endl;
}

// mykill -signumber pid
int main()
{
    signal(2, handler); // 先对2号信号进行捕捉
    // 每隔1S，自己给自己发送2号信号
    while(true)
    {
        sleep(1);
        raise(2);
    }
}

$ g++ raise.cc -o raise
$ ./raise
获取了一个信号: 2
获取了一个信号: 2
获取了一个信号: 2
```

2.3.3 abort

> abort 函数使当前进程接收到信号而异常终止。

代码块

```
NAME
        abort - cause abnormal process termination

SYNOPSIS
        #include <stdlib.h>

        void abort(void);
RETURN VALUE
        The abort() function never returns.
        // 就像exit函数一样,abort函数总是会成功的,所以没有返回值。
```

代码块

```
#include <iostream>
1#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
```

```
5
6   void handler(int signumber)
7   {
8       // 整个代码就只有这一处打印
9       std::cout << "获取了一个信号: " << signumber << std::endl;
10  }
11
12  // mykill -signumber pid
13  int main()
14  {
15      signal(SIGABRT, handler);
16      while(true)
17      {
18          sleep(1);
19          abort();
20      }
21  }
22
23  $ g++ Abort.cc -o Abort
24  $ ./Abort
25  获取了一个信号: 6 // 实验可以得知，abort给自己发送的是固定6号信号，虽然捕捉了，但是
26  还是要退出
27  Aborted
28
29  // 注释掉15行代码
30  $ ./Abort
31  Aborted
```

2.4 由软件条件产生信号

`SIGPIPE` 是一种由软件条件产生的信号,在“管道”中已经介绍过了。本节主要介绍 `alarm` 函数

和 `SIGALRM` 信号。

代码块

```
1   NAME
2           alarm - set an alarm clock for delivery of a signal
3
4   SYNOPSIS
5           #include <unistd.h>
6
7           unsigned int alarm(unsigned int seconds);
8
9   RETURN VALUE
10          alarm() returns the number of seconds remaining until any previously
```

```
11          scheduled alarm was due to be delivered, or zero if there was no previ-
12          ously scheduled alarm.
13
```

· 调用 alarm 函数可以设定一个闹钟,也就是告诉内核在 seconds 秒之后给当前进程发 `SIGALRM` 信号，该信号的默认处理动作是终止当前进程。

· 这个函数的返回值是0或者是以前设定的闹钟时间还余下的秒数。打个比方,某人要小睡一觉,设定闹钟为30分钟之后响,20分钟后被人吵醒了,还想多睡一会儿,于是重新设定闹钟为15分钟之后响,“以前设定的闹钟时间还余下的时间”就是10分钟。如果seconds值为0,表示取消以前设定的闹钟,函数的返回值仍然是以前设定的闹钟时间还余下的秒数。

2.4.1 基本alarm验证-体会IO效率问题

> 程序的作用是1秒钟之内不停地数数,1秒钟到了就被SIGALRM信号终止。
>
> 必要的时候，对SIGALRM信号进行捕捉

代码块

```
1   // IO 多
2   #include <iostream>
3   #include <unistd.h>
4   #include <signal.h>
5   int main()
6   {
7       int count = 0;
8       alarm(1);
9       while(true)
10      {
11          std::cout << "count : "
12          << count << std::endl;
13          count++;
14      }
15
16      return 0;
17  }
```

代码块

```
1   ... ...
2   count : 107148
3   count : 107149
4   Alarm clock
```

代码块

```
// IO 少
#include <iostream>
#include <unistd.h>
#include <signal.h>

int count = 0;

void handler(int signumber)
{
    std::cout << "count : " <<
    count << std::endl;
    exit(0);
}

int main()
{
    signal(SIGALRM, handler);
    alarm(1);
    while (true)
    {
        count++;
    }
    return 0;
}
```

代码块

```
$ g++ alarm.cc -o alarm
whb@bite:~/code/test$ ./alarm
count : 492333713
```

🥖 📌 结论：

- 闹钟会响一次，默认终止进程
- 有IO效率低

2.4.2 设置重复闹钟

代码样例

代码块

```
#include <iostream>
#include <unistd.h>
```

```cpp
#include <signal.h>
#include <vector>
#include <functional>

using func_t = std::function<void()>;

int gcount = 0;
std::vector<func_t> gfuncs;

// 把信号 更换 成为 硬件中断
void hanlder(int signo)
{
    for(auto &f : gfuncs)
    {
        f();
    }

    std::cout << "gcount : " << gcount << std::endl;
    int n = alarm(1); // 重设闹钟，会返回上一次闹钟的剩余时间
    std::cout << "剩余时间 : " << n << std::endl;
}

int main()
{
    //gfuncs.push_back([](){ std::cout << "我是一个内核刷新操作" << std::endl;
    });
    //gfuncs.push_back([](){ std::cout << "我是一个检测进程时间片的操作，如果时间片到了，我会切换进程" << std::endl; });
    //gfuncs.push_back([](){ std::cout << "我是一个内存管理操作，定期清理操作系统内部的内存碎片" << std::endl; });

    alarm(1); // 一次性的闹钟，超时alarm会自动被取消
    signal(SIGALRM, hanlder);
    while (true)
    {
        pause();
        std::cout << "我醒来了..." << std::endl;
        gcount++;
    }
}
NAME
    pause - wait for signal

SYNOPSIS
    #include <unistd.h>

    int pause(void);
```

```
48
49  DESCRIPTION
50          pause() causes the calling process (or thread) to sleep until a signal
51          is delivered that either terminates the process or causes the invoca-
52          tion of a signal-catching function.
53
54  RETURN VALUE
55          pause() returns only when a signal was caught and the signal-catching
56          function returned. In this case, pause() returns -1, and errno is set
57          to EINTR.
```

代码块

```
1  // 窗口 1
2  $ ./alarm
3  我的进程pid是: 216982
4  剩余时间 : 13 // 提前唤醒它，剩余时间
5  剩余时间 : 0
6  剩余时间 : 0
7  剩余时间 : 0
```

代码块

```
1  // 窗口 2
2  $ kill -14 216982
```

📌 结论：

- 闹钟设置一次，起效一次 · 重复设置的方法
- 如果时间允许，可以测试一下 alarm(0)

2.4.3 如何理解软件条件

在操作系统中，信号的软件条件指的是由软件内部状态或特定软件操作触发的信号产生机制。这些条件包括但不限于定时器超时（如alarm函数设定的时间到达）、软件异常（如向已关闭的管道写数据产生的SIGPIPE信号）等。当这些软件条件满足时，操作系统会向相关进程发送相应的信号，以通知进程进行相应的处理。简而言之，软件条件是因操作系统内部或外部软件操作而触发的信号产生。

2.4.4 如何简单快速理解系统闹钟

系统闹钟，其实本质是OS必须自身具有定时功能，并能让用户设置这种定时功能，才可能实现闹钟这

样的技术。

现代Linux是提供了定时功能的，定时器也要被管理：先描述，在组织。内核中的定时器数据结构是：

代码块

```
struct timer_list
{
    struct list_head entry;
    unsigned long expires;

    void (*function)(unsigned long);
    unsigned long data;

    struct tvec_t_base_s *base;
};
```

我们不在这部分进行深究，为了理解它，我们可以看到：定时器超时时间expires和处理方法function。

操作系统管理定时器，采用的是时间轮的做法，但是我们为了简单理解，可以把它在组织成为"堆结构"

2.5 硬件异常产生信号

硬件异常被硬件以某种方式被硬件检测到并通知内核,然后内核向当前进程发送适当的信号。例如当前进程执行了除以0的指令, CPU的运算单元会产生异常, 内核将这个异常解释为SIGFPE信号发送给进程。再比如当前进程访问了非法内存地址, MMU会产生异常,内核将这个异常解释为SIGSEGV信号发送给进程。

2.5.1 模拟除0

代码块

```
#include <stdio.h>
#include <signal.h>

void handler(int sig)
{
    printf("catch a sig : %d\n", sig);
}

// v1
int main()
{
```

```
    //signal(SIGFPE, handler); // 8) SIGFPE
    sleep(1);
    int a = 10;
    a/=0;

    while(1);
    return 0;
}
```

2.5.2 模拟野指针

代码块

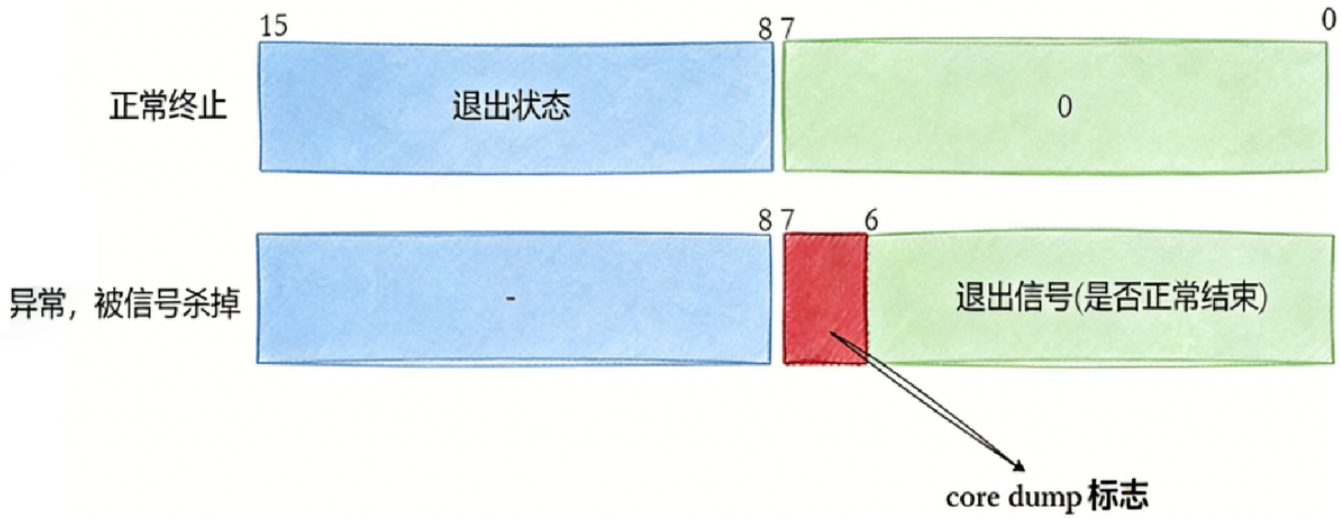
```
//默认行为
[hb@localhost code_test]$ cat sig.c
#include <stdio.h>
#include <signal.h>

void handler(int sig)
{
    printf("catch a sig : %d\n", sig);
}

int main()
{
    //signal(SIGSEGV, handler);
    sleep(1);
    int *p = NULL;
    *p = 100;

    while(1);
    return 0;
}

[hb@localhost code_test]$ ./sig
Segmentation fault (core dumped)
[hb@localhost code_test]$

//捕捉行为
[hb@localhost code_test]$ cat sig.c
#include <stdio.h>
#include <signal.h>

void handler(int sig)
{
    printf("catch a sig : %d\n", sig);
```

```
}

int main()
{
    //signal(SIGSEGV, handler);
    sleep(1);
    int *p = NULL;
    *p = 100;

    while(1);
    return 0;
}

[hb@localhost code_test]$ ./sig
[hb@localhost code_test]$ ./sig
catch a sig : 11
catch a sig : 11
catch a sig : 11
```

由此可以确认，我们在C/C++当中除零，内存越界等异常，在系统层面上，是被当成信号处理的。

> 🐵 📌 注意：
>
> 通过上面的实验，我们可能发现：
>
> 发现一直有8号信号产生被我们捕获，这是为什么呢？上面我们只提到CPU运算异常后，如何
>
> 处理后续的流程，实际上 OS 会检查应用程序的异常情况，其实在CPU中有一些控制和状态寄存器，主要用于控制处理器的操作，通常由操作系统代码使用。状态寄存器可以简单理解为一个位图，对应着一些状态标记位、溢出标记位。OS 会检测是否存在异常状态，有异常存
>
> 在就会调用对应的异常处理方法。
>
> 除零异常后，我们并没有清理内存，关闭进程打开的文件，切换进程等操作，所以CPU中还保留上下文数据以及寄存器内容，除零异常会一直存在，就有了我们看到的一直发出异常信号的现象。访问非法内存其实也是如此，大家可以自行实验。

2.5.3 子进程退出core dump



代码块

```
#include <iostream>
#include <string>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/wait.h>

int main()
{
    if (fork() == 0)
    {
        sleep(1);
        int a = 10;
        a /= 0;

        exit(0);
    }

    int status = 0;
    waitpid(-1, &status, 0);

    printf("exit signal: %d, core dump: %d\n", status&0x7F, (status>>7)&1);
    return 0;
}

$ man 7 signal
SIGABRT         P1990         Core     Abort signal from abort(3)
SIGALRM         P1990         Term     Timer signal from alarm(2)
SIGBUS          P2001         Core     Bus error (bad memory access)
SIGCHLD         P1990         Ign      Child stopped or terminated
SIGCLD            -           Ign      A synonym for SIGCHLD
```

```
SIGCONT         P1990         Cont     Continue if stopped
SIGEMT            -           Term     Emulator trap
SIGFPE          P1990         Core     Floating-point exception
SIGHUP          P1990         Term     Hangup detected on controlling terminal
                                       or death of controlling process
SIGILL          P1990         Core     Illegal Instruction
SIGINFO           -                    A synonym for SIGPWR
SIGINT          P1990         Term     Interrupt from keyboard
SIGIO             -           Term     I/O now possible (4.2BSD)
SIGIOT            -           Core     IOT trap. A synonym for SIGABRT
SIGKILL         P1990         Term     Kill signal
SIGLOST           -           Term     File lock lost (unused)
SIGPIPE         P1990         Term     Broken pipe: write to pipe with no
readers; see pipe(7)
SIGPOLL         P2001         Term     Pollable event (Sys V).

$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 7643
max locked memory       (kbytes, -l) 65536
max memory size         (kbytes, -m) unlimited
open files                      (-n) 65535
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 7643
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

2.5.4 Core Dump

· SIGINT的默认处理动作是终止进程,SIGQUIT的默认处理动作是终止进程并且Core Dump,现在我们来验证一下。

· 首先解释什么是Core Dump。当一个进程要异常终止时,可以选择把进程的用户空间内存数据全部保存到磁盘上,文件名通常是core,这叫做Core Dump。

· 进程异常终止通常是因为有Bug,比如非法内存访问导致段错误,事后可以用调试器检查core文件以查清错误原因,这叫做 `Post-mortem Debug` （事后调试）。

· 一个进程允许 产生多大的 `core` 文件取决于进程的 Resource Limit (这个信息保存 在PCB

中)。默认是不允许产生 `core` 文件的, 因为 core 文件中可能包含用户密码等敏感信息,不安全。

· 在开发调试阶段可以用 `ulimit` 命令改变这个限制,允许产生 `core` 文件。 首先用 `ulimit` 命令

改变 `Shell` 进程的 `Resource Limit` ,如允许 `core` 文件最大为 `1024K: $ ulimit -c 1024`

代码块

```
$ ulimit -c 1024
$ ulimit -a
core file size              (blocks, -c) 1024
data seg size               (kbytes, -d) unlimited
scheduling priority                 (-e) 0
file size                   (blocks, -f) unlimited
pending signals                     (-i) 7643
max locked memory           (kbytes, -l) 65536
max memory size             (kbytes, -m) unlimited
open files                          (-n) 65535
pipe size                (512 bytes, -p) 8
POSIX message queues         (bytes, -q) 819200
real-time priority                  (-r) 0
stack size                  (kbytes, -s) 8192
cpu time                   (seconds, -t) unlimited
max user processes                  (-u) 7643
virtual memory              (kbytes, -v) unlimited
file locks                          (-x) unlimited
```

> 然后写一个死循环程序:
>
> ```
> [root@localhost test11]# cat test.c
> #include <stdio.h>
> int main()
> {
>         printf("pid is : %d\n",getpid());
>         while(1);
>         return 0;
> }
> ```
>
> 前台运行这个程序,然后在终端键入Ctrl-C（貌似不行）或Ctrl-\（介个可以）：

```
[root@localhost test11]# ./test
pid is : 4506
^\Quit (core dumped)
[root@localhost test11]# ll
total 92
-rw-------. 1 root root 159744 Apr 21 18:04 core.4506
-rw-r--r--. 1 root root     61 Apr 21 18:00 Makefile
-rwxr-xr-x. 1 root root   4766 Apr 21 18:03 test
-rw-r--r--. 1 hb   hb       92 Apr 21 18:03 test.c
-rw-r--r--. 1 root root    627 Apr 21 17:36 test.cpp
[root@localhost test11]#
```

ulimit命令改变了Shell进程的Resource Limit,test进程的PCB由Shell进程复制而来,所以也具 有和Shell进程相同的Resource Limit值,这样就可以产生Core Dump了。 使用core文件：

```
total 16
-rw-r--r--. 1 root root   62 Feb 28 17:32 Makefile
-rwxr-xr-x. 1 root root 5870 Feb 28 18:03 test
-rw-r--r--. 1 root root  139 Feb 28 17:30 test.c
[root@bit_tech test78]# ./test
Segmentation fault (core dumped)
[root@bit_tech test78]# ls
core.2884  Makefile  test  test.c
[root@bit_tech test78]# gdb test
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-75.el6)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /BIT/project/test78/test...done.
(gdb) core-file core.2884
[New Thread 2884]
Missing separate debuginfo for
Try: yum --enablerepo='*-debug*' install /usr/lib/debug/.build-id/d5/942f21cf3002919e55180a33793d3c25a060f3
Reading symbols from /lib/libc-2.12.so...Reading symbols from /usr/lib/debug/lib/libc-2.12.so.debug...done.
done.
Loaded symbols for /lib/libc-2.12.so
Reading symbols from /lib/ld-2.12.so...Reading symbols from /usr/lib/debug/lib/ld-2.12.so.debug...done.
done.
Loaded symbols for /lib/ld-2.12.so
Core was generated by `./test'.
Program terminated with signal 11, Segmentation fault.
#0  0x080483ad in fun () at test.c:8
8                       *p = 100;
(gdb)
```

2.6 总结思考一下

- 上面所说的所有信号产生，最终都要有OS来进行执行，为什么？OS是进程的管理者
- 信号的处理是否是立即处理的？在合适的时候
- 信号如果不是被立即处理，那么信号是否需要暂时被进程记录下来？记录在哪里最合适呢？
- 一个进程在没有收到信号的时候，能否能知道，自己应该对合法信号作何处理呢？
- 如何理解OS向进程发送信号？能否描述一下完整的发送处理过程？