七、訊號(signal)

訊號概述

- 每一個訊號都有一個名稱，開頭前三個字母是SIG，可以用kill -l 來列出所有的訊號。訊號名稱定義在singal.h中，每個訊號都有一個對應的非零正整數(可以man 7 signal)
- 當某件事情發生時，會產生相對應的訊號
  - 硬體錯誤：例如除以0會產生SIGFPE訊號
  - 使用kill(2)函式：必須是該行程的擁有者或是為superuser
  - 使用kill(1)命令
  - 軟體狀態改變時：例如鬧鈴時間到了時會產生SIGALRM訊號
- 行程收到訊號時有下列三種處理方法
  - 忽略該訊號
  - 交由事先註冊好的signal handler來處理
  - 使用預設的處理動作

常用訊號(1/2)

- 1) SIGHUP(TERM)
  當偵測到controlling terminal掛掉或是controling process結束
- 2) SIGINT(TERM)
  在terminal中按ctrl+C
- 3) SIGQUIT(CORE)
  在terminal中按ctrl+\
- 4) SIGILL(CORE)
  執行到錯誤的指令
- 5) SIGTRAP(CORE)
  執行到硬體中斷點
- 6) SIGABRT(CORE)
  行程呼叫abort函式
- 7) SIGBUS(CORE)
  錯誤的記憶體存取
- 8) SIGFPE(CORE)
  錯誤的浮點運算
- 9) SIGKILL(TERM)
  砍除行程的訊號
- 10) SIGUSR1(TERM)
  由行程自行決定用途
- 11) SIGSEGV(CORE)
  錯誤的記憶體參考
- 12) SIGUSR2(TERM)
  由行程自行決定用途
- 13) SIGPIPE(TERM)
  寫入一個沒有行程讀取的pipe
- 14) SIGALRM(TERM)
  由alarm函式設定鬧鈴時間，時間到了就會發出這個訊號
- 15) SIGTERM(TERM)
  結束行程的訊號
- 17) SIGCHLD(IGN)
  子行程結束或暫停執行

紅色字的訊號不能被攔截

常用訊號(2/2)

- 18) SIGCONT
  送這個訊號給行程可以讓該行程繼續往下執行
- 19) SIGSTOP(STOP)
  送這個訊號給行程可以讓該行程暫停執行
- 20) SIGTSTP(STOP)
  在termainl中按ctrl+Z，這個訊號會送給所有在foreground process group的行程
- 21) SIGTTIN(STOP)
  在background process group中的行程嘗試讀取controlling terminal
- 22) SIGTTOU(STOP)
  在background process group中的行程嘗試寫入controlling terminal
- 23) SIGURG(IGN)
  從網路上收到Out of bound的資料
- 24) SIGXCPU(CORE)
  超過CPU使用限制
- 25) SIGXFSZ(CORE)
  超過最大檔案限制
- 26) SIGVTALRM(TERM)
  由setitimer函式所設定的時間
- 27) SIGPROF(TERM)
  由setitimer函式所設定的時間
- 28) SIGWINCH(IGN)
  當terminal改變大小時
- 29) SIGIO(TERM)
  當asynchronous I/O可以存取時
- 30) SIGPWR(TERM)
  當停電時
- 31) SIGSYS(CORE)
  使用錯誤的參數呼叫系統呼叫時

紅色字的訊號不能被攔截

signal Function

CODE. 7-1

```
#include <signal.h>

    typedef void (*sighandler_t)(int);
    sighandler_t signal(int signum, sighandler_t handler);
```

- 相當於
  void (*signal(int signum,void (*handler)(int)))(int);
- 三個常數
  - SIG_ERR:用在傳回值，代表錯誤
  - SIG_DFL:當handler使用這個值時，signum訊號將會使用預設的處理方式
  - SIG_IGN:當handler使用這個值時，signum訊號將會被忽略

signal函式的問題(1/3)

- 當訊號產生後，signal函式會將該訊號的signal handler設為SIG_DFL，這造成由signal函式註冊的signal handler，在執行後(訊號產生後)必須重新註冊(因為之前註冊的值會被重設為SIG_DFL)，而註冊之前為空窗期，此時如果有該訊號產生，會發生遺失的現象
- signal函式能做的控制有限，只能做擷取或是忽略的動作，不能暫時將訊號block住

signal函式的問題(2/3)

```
1: int sig_int();                       //訊號處理函式
2: ….
3: signal(SIGINT, sig_int);             //註冊當發生SIGINT時執行sig_int函式
4:
5: int sig_int(){
6:          signal(SIGINT, sig_int);    //重新註冊
7:          …….
8: }
```

- 當產生SIGINT訊號時，程式會跳到sig_int函式執行，此時(執行sig_int的過程中)如果又有另一個SIGINT產生，則有兩種可能情況：
  - 如果第二個SIGINT訊號發生在第5行執行之前，此SIGINT會以預設方式(SIG_DFL)來處理，也就是結束程式執行
  - 如果第二個SIGINT訊號發生在第5行之後，sig_int函式會有重入的現象

signal函式的問題(3/3)

```
 1: int sig_int_flag;
 2: int sig_int();
 3:
 4: int main(void){
 5:          signal(SIGINT, sig_int);
 6:          …
 7:          while(sig_int_flag==0)
 8:                    pause();
 9:}
10:
11: int sig_int(){
12:          signal(SIGINT, sig_int);
13:          sig_int_flag=1;
14 }
```

- pause函式會將程式暫停，直到有訊號發生為止(用來避免busy waiting)。上面程式的正常運作流程為程式會停在第8行，直到有SIGINT訊號產生，然後執行sig_int函式把sig_int_flag設為1，如此就可以離開第7行的迴圈
- 如果訊號發生在sig_int_flag==0的判斷之後，pause函式之前會發生什麼事？程式會永遠被block住(除非在產生另一個SIGINT訊號)。造成這個原因是因為第7和第8行不是一個atomic operation

可被訊號中斷的系統呼叫

- 一般來說，訊號產生時會跳到該訊號的處理函式執行，然後返回原來的地方繼續執行，但是如果訊號在系統呼叫的執行期間發生，就有下列兩種情況
  - 訊號處理函式執行完畢後跳回系統呼叫繼續執行
  - 訊號處理函式執行完畢後，該系統呼叫也會被中斷，然後回傳錯誤，而errno會設為EINTR。如果是這種情況，就必須自行處理中斷發生時該怎麼做，如下面程式
- 因此系統呼叫可以分成兩大類，一類是不會被中斷執行(訊號處理完後會繼續執行)，一類是會被中斷執行(訊號處理完後會結束執行)

```
again:
        if((n=read(fd,buff,BUFFSIZE<0){
                if(errno==EINTR)
                        goto again
        }
```


會被訊號中斷的系統呼叫(1/2)

- 會被訊號中斷的系統呼叫我們稱為"slow" system calls，包含下列(這邊永久的意思代表某個狀態不發生，就會永久被block住)：
  - 讀取會造成永久block住的檔案(像是pipe，terminal device，socket等等)
  - 寫入會造成永久block住的檔案(像是pipe，terminal device，socket等等)
  - 開啟會造成永久block住的檔案(例如開啟terminal device必須等到電話有反應才能繼續執行下去)
  - pause與wait函式
  - 某些ioctl操作
  - 某些行程通訊
- 特別注意，所有跟disk I/O相關的slow system calls皆不會被中斷，雖然讀取disk上的檔案也需要一段時間，但是這時間不是永久

會被訊號中斷的系統呼叫(2/2)

- 之前提到過，程式設計師必須特別考慮系統呼叫被訊號中斷的情況，必須在此類系統呼叫後加一個判斷，如果是被訊號中斷，則必須重做
- 這對於程式設計師來說是很大的負擔，因此ioctl，read，readv，write，writev，wait和waitpid函式在某些作業系統中會自動重做。由於這可能不符合程式設計師的需要，因此最好使用sigaction函式來指定是否要重做(設定SA_RESTART)

可重入(reentrant)的函式

- 在signal handler中，只能使用可重入的函式，否則可能產生問題，不可重入函式可能是下列原因造成
  - 使用static data structures
  - 使用malloc與free函式
  - 標準I/O函式庫(通常使用全域資料結構)
- 每一個行程只有一個errno變數，因此有些可重入的函式還是會改變errno的值，例如read，因此最好的方式是在signal handler的開頭處把errno存起來，然後結束時在將原errno的值回存回去

_exit	fork	pipe	stat
abort*	fstat	read	sysconf
access	getegid	rename	tcdrain
alarm	geteuid	rmdir	tcflow
cfgetispeed	getgid	setgid	tcflush
cfgetospeed	getgroups	setpgid	tcgetattr
cfsetispeed	getpgrp	setsid	tcgetpgrp
cfsetospeed	getpid	setuid	tcsendbreak
chdir	getppid	sigaction	tcsetattr
chmod	getuid	sigaddset	tcsetpgrp
chown	kill	sigdelset	time
close	link	sigemptyset	times
creat	longjmp*	sigfillset	umask
dup	lseek	sigismember	uname
dup2	mkdir	signal*	unlink
execle	mkfifo	sigpending	utime
execve	open	sigprocmask	wait
exit*	pathconf	sigsuspend	waitpid
fcntl	pause	sleep	write

發送訊號給行程

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

    int kill(pid_t pid, int sig);
    int raise(int sig);
```

- kill函式用來發送sig訊號給某個或某些行程(pid)，其中pid的值可以如下：
  - pid>0 送訊號給pid行程
  - pid==0 送給同一個process group的所有行程
  - pid==-1 送給除了init與自己之外的所有行程
  - pid<-1 送給process group id=|pid|的行程
- 如果sig為0(沒有訊號為零的訊號)，kill函式用來測試該行程是否存在，而不是送訊號0給行程
- superuser可以用kill函式送訊號給任何行程，其它使用者只能送訊號給相同real user或effective user的行程
- raise用來送訊號給自己，相當於kill(getpid(),sig);

暫停行程的執行

```
#include <unistd.h>

    unsigned int alarm(unsigned int seconds);
    int pause(void);
```

- alarm函式用來設定鬧鈴，單位為秒，當時間到達後會發出SIGALRM訊號，特別注意，該時間是訊號發出的時間，不代表被處理的時間
- 每一個行程只有一個鬧鈴，因此如果先前已經呼叫過alarm函式，接下來呼叫的alarm函式就會傳回前一次所剩下的時間，而鬧鈴時間會重新設定與計算
- 如果seconds參數為0，之前設定的鬧鈴會被取消
- SIGALRM的預設處理方式為結束(terminate)行程，不過大部分的程式會擷取這個訊號自行處理
- pause函式用來暫停程式執行，直到有訊號產生，pause函式會在signal handler執行完畢後返回。其返回值為-1，errno會設為EINTR

使用alarm函式來實作sleep(1/2)

- 可以用alarm函式來實作sleep，下列程式，如果SIGALRM發生在第一個alarm函式後，pause函式前，則pause函式永遠都被block住

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

static void
sig_alrm(int signo)
{
      return; /* nothing to do, just return to wake up the pause */
}

unsigned int
sleep1(unsigned int nsecs)
{
      if (signal(SIGALRM, sig_alrm) == SIG_ERR)
            return(nsecs);
      alarm(nsecs);                  /* start the timer */
      pause();                       /* next caught signal wakes us up */
      return( alarm(0) );            /* turn off timer, return unslept time */
}
```


使用alarm函式來實作sleep(2/2)

- 可以用下列程式解決上面程式的問題(race condition)，但還是有另一個問題，當SIGALRM發生在其它signal handler時，longjmp函式會讓原本的signal handler執行到一半而已

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

static jmp_buf  env_alrm;
static void sig_alrm(int signo){
      longjmp(env_alrm, 1);
}

unsigned int sleep2(unsigned int nsecs){
      if (signal(SIGALRM, sig_alrm) == SIG_ERR)
            return(nsecs);
      if (setjmp(env_alrm) == 0) {
            alarm(nsecs);          /* start the timer */
            pause();                   /* next caught signal wakes us up */
      }
 return( alarm(0) );           /* turn off timer, return unslept time */
}
```

訊號集合(signal sets)

CODE. 7-2

```
#include <signal.h>

    int sigemptyset(sigset_t *set);
    int sigfillset(sigset_t *set);
    int sigaddset(sigset_t *set, int signum);
    int sigdelset(sigset_t *set, int signum);
    int sigismember(const sigset_t *set, int signum);
```

- 訊號集合用來描述一群訊號，我們可以使用上述函式來處理sigset_t所代表的集合
  - sigemptyset用來將sigset_t集合中的訊號清空
  - sigfillset用來將所有訊號加入sigset_t集合中
  - sigaddset用來將signum訊號加入sigset_t集合中
  - sigdelset用來將signum訊號從sigset_t集合中移除
  - sigismember用來判斷sigset_t集合中是否有signum訊號
- 一般來說，我們可以用sigemptyset把集合清空，然後把要的訊號加入。或是用sigfillset把集合清空，然後把不要的訊號移除

訊號遮罩(signal mask)

CODE. 7-3

```
#include <signal.h>

    int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

- 可以透過設定訊號遮罩(signal mask)，阻止(block)某些訊號送到行程中，例如某些時刻我們不希望被某些訊號給中斷，此時就可以先用訊號遮罩將這些不想處理的訊號給遮罩掉。sigprocmask函式就是用來設定前述的訊號遮罩
- 當訊號產生時，如果該訊號在訊號遮罩中，則該訊號會被暫時block住，等到unblock後才會送給行程處理。如果在block的其間產生同個訊號多次，在unblock之後還是只有一個訊號會被處理，也就是只記錄該訊號是否在block其間發生過，而不會記錄發生的次數
- sigprocmask有三個參數，其中如果oldset不為NULL時，原本的訊號遮罩就會記錄到oldset中。另外set函式為要設定的訊號遮罩，how有下列三種
  - SIG_BLOCK: 將set集合中的訊號加入訊號遮罩中
  - SIG_UNBLOCK: 將set集合中的訊號從訊號遮罩移除
  - SIG_SETMASK: 將set集合直接設定成訊號遮罩

取得尚未處理(被block)的訊號

```
#include <signal.h>

    int sigpending(sigset_t *set);
```

- sigpending函式用來取得被訊號遮罩block住的訊號，被block的訊號會被設定在set集合中，然後再使用sigismember來測試某個訊號是否被block住

CODE. 7-4

```
sigset_t newmask, oldmask, pendmask;

signal(SIGQUIT, sig_quit);                          //註冊SIGQUIT訊號的signal handler

sigemptyset(&newmask);
sigaddset(&newmask, SIGQUIT);
sigprocmask(SIG_BLOCK,&newmask,&oldmask);           //將SIGQUIT遮罩掉

sleep(10);  //如果在這邊產生SIGQUIT，該訊號會被block，不會由sig_quit處理

sigpending(&pendmask); //取得尚未處理的訊號
if(sigismember(&pendmask,SIGQUIT)) printf("SIGQUIT pending\n";

sigprocmask(SIG_SETMASK, &oldmask, NULL);           //回原成原來的訊號遮罩

sleep(10); //如果在這邊產生SIGQUIT，該訊號會直接由sig_quit處理
```


註冊訊號處理函式

CODE. 7-5

```
#include <signal.h>

    int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

```
struct sigaction {
        void (*sa_handler)(int);
        sigset_t sa_mask;
        int sa_flags;
}
```

- sigaction函式與signal函式相同，都是用來註冊signal handler，但是功能比signal函式更強
- sigaction函式有一個參數act，用來設定新的signal handler，而舊的設定會放在oldact中。在結構sigaction中，sa_handler為函式指標，代表signum訊號發生時會執行那個函式。sa_mask用來設定執行sa_handler時，要遮罩掉哪些訊號，換言之在執行sa_handler前，sa_mask的值會自動加入訊號遮罩中，當sa_handler執行完後，會自動還原成原本的訊號遮罩。sa_flags用來設定signal handler的處理方式，有下列可能的值(可以OR起來)
  - SA_NOCLDSTOP 當signum為SIGCHLD時，當子行程暫停時不發出此訊號
  - SA_ONESHOT 執行此signal handler一次後就將signal handler設為預設值(SIG_DFL)
  - SA_RESTART 當系統呼叫被此訊號中斷時會自動重新啟動
  - SA_NOMASK 不會把自己(signum)加入訊號遮罩中
- 由sigaction註冊的訊號發生時，該訊號會自動加入訊號遮罩中，換言之同一個訊號不會重入
- 由sigaction註冊的signal handler其效力為永久(除非重新設定)，不像signal函式每次都要重新設定

sigsetjmp與siglongjmp

```
#include <setjmp.h>

    int sigsetjmp(sigjmp_buf env, int savesigs);
    void siglongjmp(sigjmp_buf env, int val);
```

- 一個signal handler可以用下列三種方式返回(ANSI C標準)
  - 呼叫abort函式
  - 呼叫exit函式
  - 呼叫longjmp函式
- 如果我們在signal handler中使用longjmp函式返回，可能會發生問題。因為使用sigaction方式註冊的訊號在執行signal handler之前會把自己加入訊號遮罩中，因此如果直接用longjmp跳回而不修改訊號遮罩的話，該訊號就會一直被訊號遮罩給block住
- 在signal handler中必須使用siglongjmp和與之搭配的sigsetjmp函式，sigsetjmp可以記錄之前訊號遮罩的狀態，在siglongjmp時可以將訊號遮罩的狀態還原回來

sigsuspend Function

```
#include <signal.h>

    int sigsuspend(const sigset_t *mask);
```

- 假設有一段程式碼如下:
  sigprocmask(SIG_SETMASK,&empty,NULL);
  pause();
  其中第一行程式將訊號遮罩清除，此時被block住的訊號就會送到行程去處理，而第二行程式會暫停程式直到有任何訊號產生。但有可能訊號是發生在第一行與第二行之間，此時pause函式永遠都不會返回，其原因就是這兩個函式不是一個atomic operation
- sigsuspend函式就是用來解決上述問題，它會做兩件事情，依序為
  - 改變訊號遮罩
  - 暫停程式執行直到有訊號產生
- 上述兩件事情為一個atomic operation，所以不會有訊號遺失
- sigsuspend永遠都是傳回-1，而errno會被設為EINTR

abort Function

```
#include <stdlib.h>

    void abort(void);
```

- abort函式用來送SIGABRT訊號給自己，我們可以註冊一個signal handler用來做clean up的動作。除非signal handler不返回，否則該行程會異常結束(產生core dump)
- SIGABRT訊號不會被遮罩掉

sleep Function

```
#include <unistd.h>

    unsigned int sleep(unsigned int seconds);
```

- sleep函式用來讓行程暫停執行seconds秒，它返回的條件必須為下列兩個其中之一
  - 等待的時間到了，此時sleep的回傳值為0
  - 某個訊號被行程擷取，該訊號的signal handler處理後返回時，此時sleep的回傳值為剩下的時間

執行緒中的訊號處理

CODE. 7-6

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

    int pthread_sigmask(int how, const sigset_t  *newmask,  sigset_t  *oldmask);
    int pthread_kill(pthread_t thread, int signo);
    int sigwait(const sigset_t *set, int *sig);
```

- 每一個執行緒可以有自己的訊號遮罩，可以用pthread_sigmask來設定，其用法與sigporcmask函式相同。而在同一個行程中的執行緒則是共用相同的signal handler，因此當某個訊號產生時，預設所有的執行緒都會收到
- signal_kill函式用來送訊號給特定的執行緒
- sigwait會暫停執行緒的執行直到set中所描述的訊號產生為止(只要某一個訊號產生就行)，產生訊號的個數會記錄在sig中