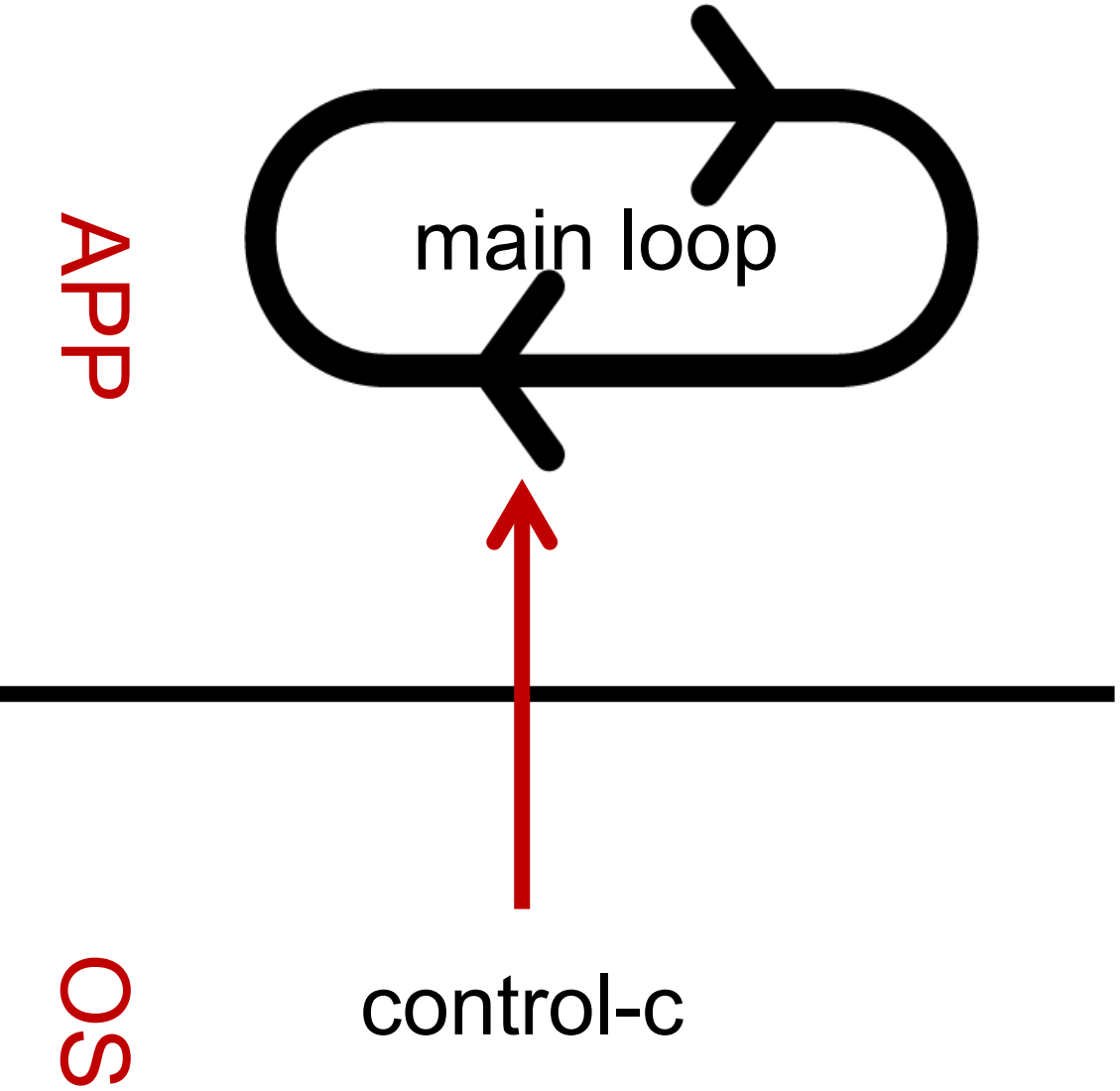
信號(Signals)

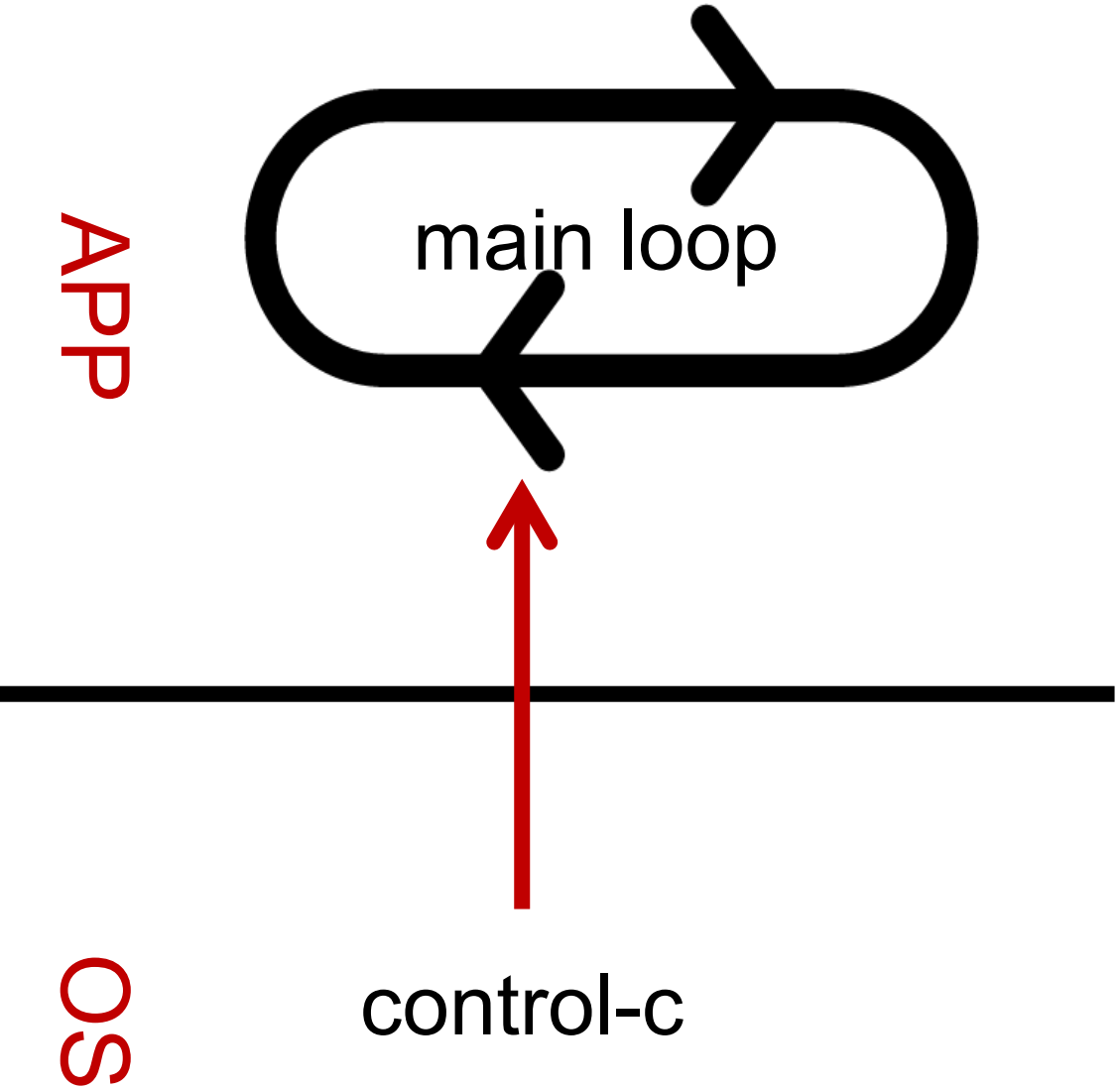
中正大學，作業系統實驗室
羅習五 陽春副教授
shiwulo@gmail.com

什麼是signal



- 主程式通常是由一個巨型的迴圈所構成
- 如果使用者按下ctr-c以後，該主程式如何回應？

- 在主程式偵測ctr-c?
- 由作業系統處理ctr-c?
- 主程式告訴作業系統如何處理ctr-c?



- UNIX的做法（包含Linux）：
- 主程式告知作業系統如何處理ctr-c
- 如果主程式沒有告訴OS如何處理ctr-c，那麼OS會採取預設動作：將這個程式結束掉

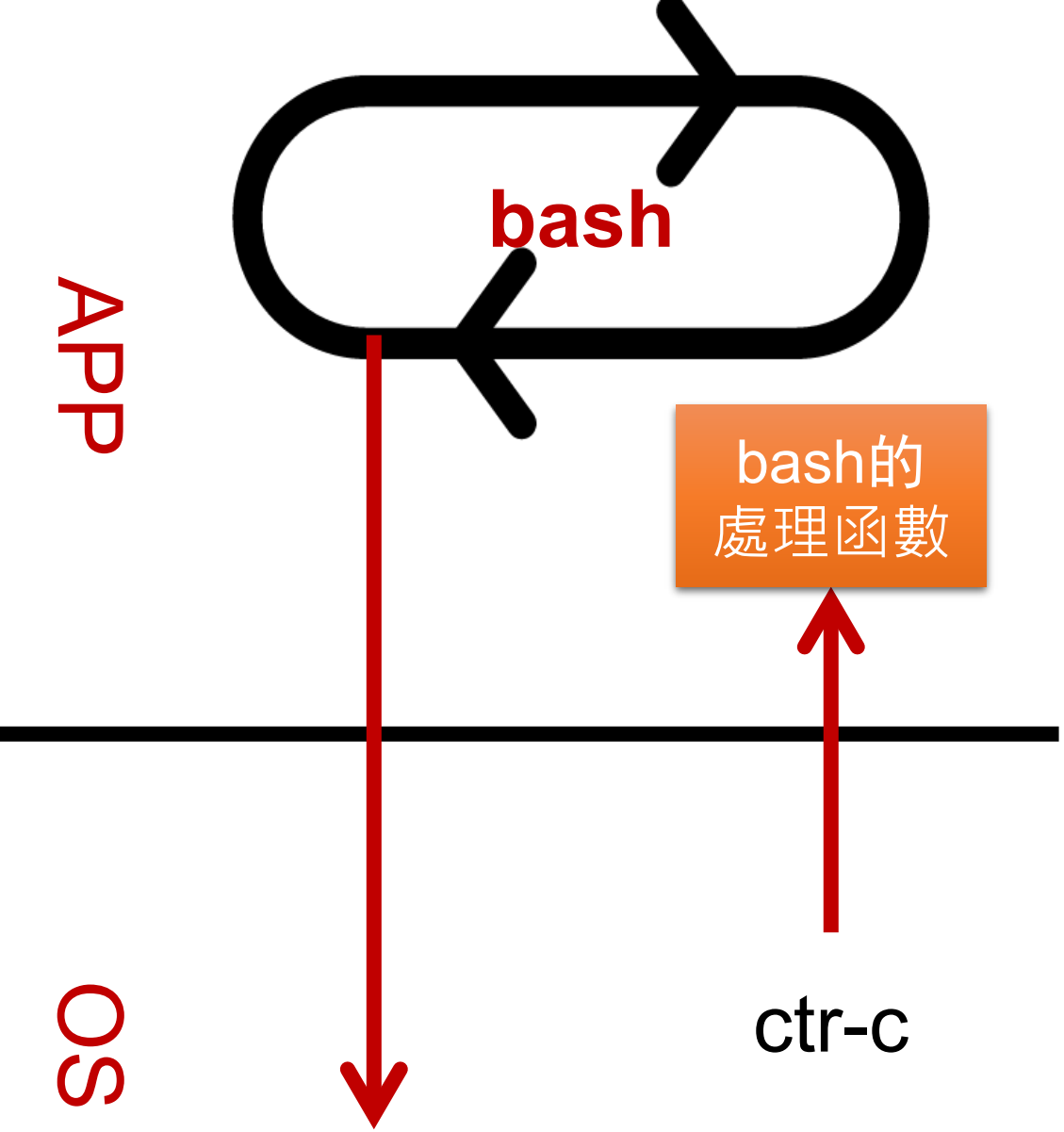
範例

bash

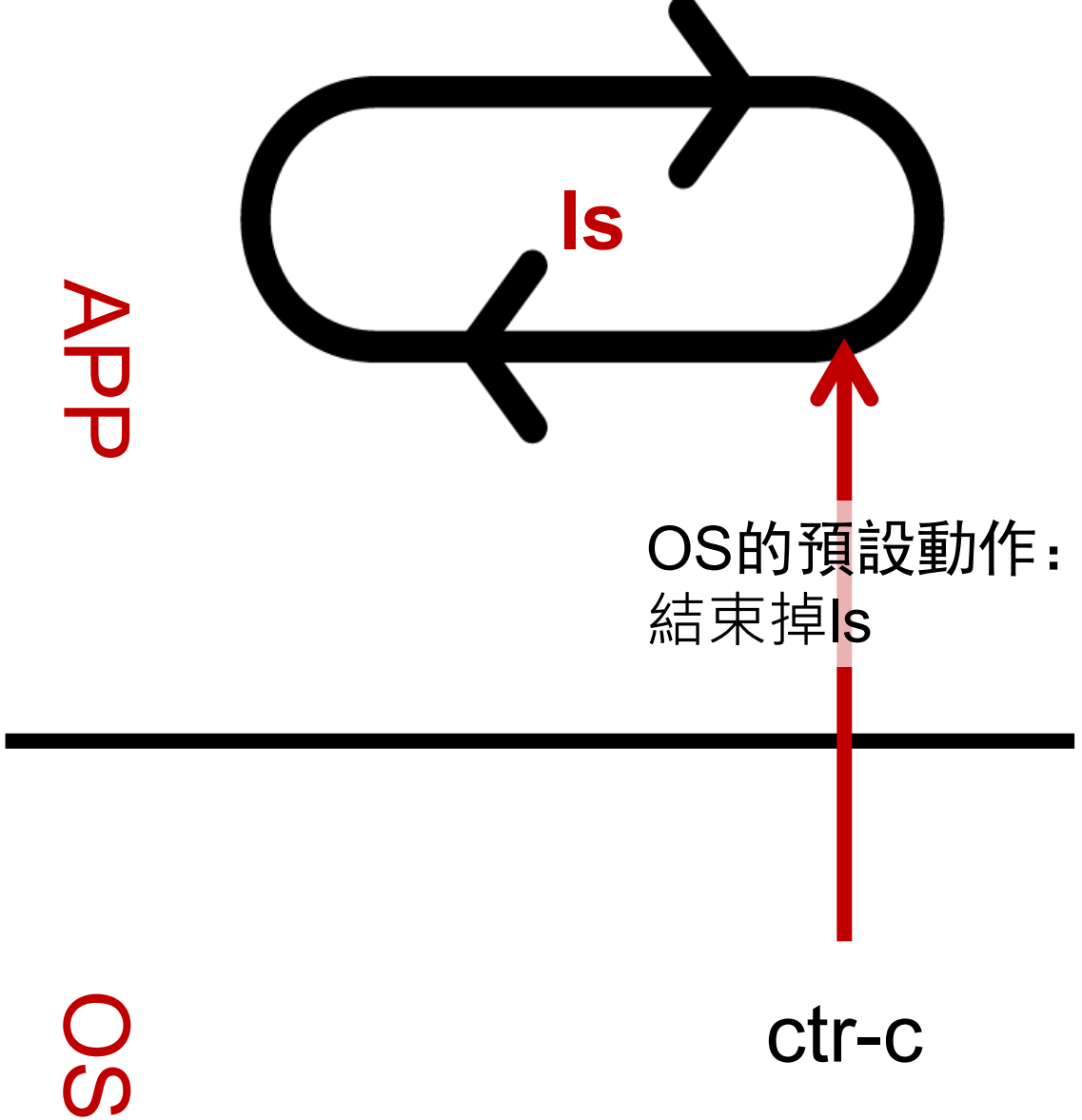
```
shiwulo@NUC:~$ ^C
shiwulo@NUC:~$ ^C
shiwulo@NUC:~$ ^C
/*按下ctr-c以後沒有反應*/
```

ls –R /

```
shiwulo@NUC:~$ ls -R /
/*...*/
/proc/316/task/316/net/stat:
arp_cache  ndisc_cache  rt_cache
ls: cannot open directory
'/proc/316/task/316/ns': Permission denied
/proc/317:
^C
shiwulo@NUC:~$
/*按下ctr-c以後終止執行*/
```



告知OS遇到ctr-c的時候要呼叫「處理函數」



沒有告知OS遇到ctr-c的時候要怎樣處理


建立signal的最簡單方法

Linux上的signal的函數宣告

- #include <signal.h>
- typedef void (*sighandler_t)(int);
- sighandler_t signal(int signum, sighandler_t handler);

signal的用法

- 第一個參數接一個signal的編號，例如：SIGKILL
- 第二個參數接一個函數指標，該函數的參數是signal的編號，回傳值是void『或，第二個參數是SIG_IGN、SIG_DFL』
  - 如果是SIG_IGN，則忽略該signal
  - 如果是SIG_DEF，則採用Linux內建的處理方式

- 但這個函數在不同作業系統上行為不太一樣，『**跨平台時**最好用sigaction』代替signal
  - 這門課假設是在Linux下撰寫程式，因此在大部分情況下singal是足夠的
  - 後面會介紹sigaction相較於signal，明確定義的地方

送出一個signal

- Linux指令： kill

- kill: To send a signal to a process or a process group

```
#include <signal.h>
int kill(pid_t pid, int signo);
```

Both return: 0 if OK, -1 on error

kill(pid_t *pid*, int *signo*)

- pid可以是下列值

< -1	所有group id為|pid|的child結束
-1	送signal給所有的task（前提是，要有權限送）
0	任意一個跟自己的group id一樣的child結束
> 0	等process ID為pid的child結束

- signo=0：判斷該行程是否存在，是否有權限送signal給該行程

kill function

- Permission to send signals:
  - Superuser: to any process
  - Others: real/effective ID of sender must be equal to real/effective ID of receiver

list_sig.c：列印所有可註冊的signal

```
1.  void sighandler(int signumber) {
2.      printf("get a signal named '%d', '%s'\n",
3.          signumber, sys_siglist[signumber]);
4.  }

5.  int main(int argc, char **argv) {
6.      int sig_exist[100];
7.      int idx = 0;
8.      for (idx = 0; idx < 100; idx++) {
9.          if (signal(idx, sighandler) == SIG_ERR) {
```

list_sig.c：列印所有可註冊的signal

```
1.              sig_exist[idx] = 0;
2.          } else {
3.              sig_exist[idx] = 1;
4.          }
5.      }
6.      for (idx = 0; idx < 100; idx++) {
7.          if (sig_exist[idx] == 1)
8.              printf("%2d %s\n", idx, sys_siglist[idx]);
9.      }
10.     printf("my pid is %d\n", getpid());
11.     printf("press any key to resume\n");
12.     getchar();
13. }
```

results (MAC OS X)

```
 1 Hangup
 2 Interrupt
 3 Quit
 4 Illegal instruction
 5 Trace/BPT trap
 6 Abort trap
 7 EMT trap
 8 Floating point exception
10 Bus error
11 Segmentation fault
12 Bad system call
13 Broken pipe
14 Alarm clock
15 Terminated
```

```
16 Urgent I/O condition
18 Suspended
19 Continued
20 Child exited
21 Stopped (tty input)
22 Stopped (tty output)
23 I/O possible
24 Cputime limit exceeded
25 Filesize limit exceeded
26 Virtual timer expired
27 Profiling timer expired
28 Window size changes
29 Information request
30 User defined signal 1
31 User defined signal 2
```

results (Linux，不可靠信號)

```
 1 Hangup
 2 Interrupt
 3 Quit
 4 Illegal instruction
 5 Trace/breakpoint trap
 6 Aborted
 7 Bus error
 8 Floating point exception
10 User defined signal 1
11 Segmentation fault
12 User defined signal 2
13 Broken pipe
14 Alarm clock
15 Terminated
```

```
17 Child exited
18 Continued
20 Stopped
21 Stopped (tty input)
22 Stopped (tty output)
23 Urgent I/O condition
24 CPU time limit exceeded
25 File size limit exceeded
26 Virtual timer expired
27 Profiling timer expired
28 Window changed
29 I/O possible
30 Power failure
31 Bad system call
```

results (Linux，可靠信號)

```
34  (null)
35  (null)
36  (null)
37  (null)
38  (null)
39  (null)
40  (null)
41  (null)
42  (null)
43  (null)
44  (null)
45  (null)
46  (null)
47  (null)
48  (null)
49  (null)
```

```
50  (null)
51  (null)
52  (null)
53  (null)
54  (null)
55  (null)
56  (null)
57  (null)
58  (null)
59  (null)
60  (null)
61  (null)
62  (null)
63  (null)
64  (null)
```

signal

hardware	Terminal	Sortware
SIGBUS（通常是沒有對齊word）	SIGINT（ctr+C）	SIGCHILD（子行程結束）
SIGFPE（浮點運算或『/0』）	SIGQUIT（ctr+\）	SIGURG
SIGILL（錯誤的指令）	SIGTSTP（ctr+Z）	SIGWINCH（窗口大小改變）
SIGPWR	SIGHUP	SIGUSR1、SIGUSR2
SIGIO	SIGKILL	SIGPIPE
SIGTRAP（除錯）	SIGTERM	SIGALARM
	SIGSTOP	SIGVALARM
	SIGTSTP	SIGPROF
	SIGTTIN	SIGABRT
	SIGTTOU	SIGXCPU
	SIGCONT	SIGXFSZ
		SIGSYS

Linux‘s signal

First the signals described in the original POSIX.1-1990 standard.

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating-point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers; see pipe(7)
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at terminal
SIGTTIN	21,21,26	Stop	Terminal input for background process
SIGTTOU	22,22,27	Stop	Terminal output for background process

The signals **SIGKILL** and **SIGSTOP** cannot be caught, blocked, or ignored.

http://man7.org/linux/man-pages/man7/signal.7.html

Linux‘s signal

Next the signals not in the POSIX.1-1990 standard but described in SUSv2 and POSIX.1-2001.

Signal	Value	Action	Comment
SIGBUS	10,7,10	Core	Bus error (bad memory access)
SIGPOLL		Term	Pollable event (Sys V). Synonym for **SIGIO**
SIGPROF	27,27,29	Term	Profiling timer expired
SIGSYS	12,31,12	Core	Bad system call (SVr4); see also seccomp(2)
SIGTRAP	5	Core	Trace/breakpoint trap
SIGURG	16,23,21	Ign	Urgent condition on socket (4.2BSD)
SIGVTALRM	26,26,28	Term	Virtual alarm clock (4.2BSD)
SIGXCPU	24,24,30	Core	CPU time limit exceeded (4.2BSD); see setrlimit(2)
SIGXFSZ	25,25,31	Core	File size limit exceeded (4.2BSD); see setrlimit(2)

http://man7.org/linux/man-pages/man7/signal.7.html

Linux‘s signal

Next various other signals.

Signal	Value	Action	Comment
SIGIOT	6	Core	IOT trap. A synonym for **SIGABRT**
SIGEMT	7,-,7	Term	Emulator trap
SIGSTKFLT	-,16,-	Term	Stack fault on coprocessor (unused)
SIGIO	23,29,22	Term	I/O now possible (4.2BSD)
SIGCLD	-,-,18	Ign	A synonym for **SIGCHLD**
SIGPWR	29,30,19	Term	Power failure (System V)
SIGINFO	29,-,-		A synonym for **SIGPWR**
SIGLOST	-,-,-	Term	File lock lost (unused)
SIGWINCH	28,28,20	Ign	Window resize signal (4.3BSD, Sun)
SIGUNUSED	-,31,-	Core	Synonymous with **SIGSYS**

(Signal 29 is **SIGINFO** / **SIGPWR** on an alpha but **SIGLOST** on a sparc.)

http://man7.org/linux/man-pages/man7/signal.7.html

課堂小作業

試試看

“執行`kill -l`”

結果

```
 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
```

課堂小作業 – list_sig

試試看

1. kill -4 pid
2. 調整terminal window的大小

list_sig

```
$./list_sig
...
63 (null)
64 (null)
my pid is 2271
press any key to resume
get a signal named '4', 'Illegal instruction'
```

```
$ kill -4 2271
```

課堂小作業

試試看
“故意存取錯誤的記憶體”

seg_fault.c

```
1.  int *c;
2.  void sighandler(int signumber) {
3.          printf("get a signal named '%d', '%s'\n", signumber, sys_siglist[signumber]);
4.  }
5.  int main(int argc, char **argv) {
6.          assert(signal(SIGSEGV, sighandler) != SIG_ERR);
7.          *c = 0xCOFE;/*c沒有初始化就使用*/
8.          printf("press any key to resume\n");
9.          getchar();
10. }
```

執行結果

```
get a signal named '11', 'Segmentation fault'
get a signal named '11', 'Segmentation fault'
get a signal named '11', 'Segmentation fault'
get a signal named '11', 'Segmentation fault'
get a signal named '11', 'Segmentation fault'
get a signal named '11', 'Segmentation fault'
get a signal named '11', 'Segmentation fault'
get a signal named '11', 'Segmentation fault'
get a signal named '11', 'Segmentation fault'
get a signal named '11', 'Segmentation fault'
^C
/*因為變數c依然是無意義的指標，sighandler執行完以後，會重新執行第13行，所以不斷的造成'Segmentation fault'*/
```

seg_fault_recover.c

```
1.  int *c;
2.  void sighandler(int signumber) {
3.      printf("get a signal named '%d', '%s'\n", signumber, sys_siglist[signumber]);
4.      c=(int*)malloc(sizeof(int)); /*替c分配記憶體*/
5.  }
6.  int main(int argc, char **argv) {
7.      assert(signal(SIGSEGV, sighandler) != SIG_ERR);
8.      *c = 0xC0FE;/*c沒有初始化就使用*/
9.      printf("press Enter to continue\n");
10.     getchar();
11. }
```

執行結果

```
get a signal named '11', 'Segmentation fault'
get a signal named '11', 'Segmentation fault'
get a signal named '11', 'Segmentation fault'
get a signal named '11', 'Segmentation fault'
get a signal named '11', 'Segmentation fault'
get a signal named '11', 'Segmentation fault'
get a signal named '11', 'Segmentation fault'^C
/*在sighandler中，替變數c分配記憶體似乎是無用的*/
```

使用gdb除錯

```
1. (gdb) b main
2. Breakpoint 1 at 0x400745: file seg_fault_recover.c, line 13.
3. (gdb) b sighandler
4. Breakpoint 2 at 0x400701: file seg_fault_recover.c, line 9.
5. (gdb) r
6. Starting program: /home/shiwulo/Dropbox/course/2017-
   sp/ch10/seg_fault_recover


7. Breakpoint 1, main (argc=1, argv=0x7fffffffdd88) at
   seg_fault_recover.c:13
8. 13     assert(signal(SIGSEGV, sighandler) != SIG_ERR);
9. (gdb) n
```

使用gdb除錯

```
10.14     *c = 0xF0FE;/*c沒有初始化就使用*/
11.(gdb) n
12.Program received signal SIGSEGV, Segmentation fault.
13.0x000000000040077a in main (argc=1,
   argv=0x7fffffffdd88)
14.     at seg_fault_recover.c:14
15.14     *c = 0xF0FE;/*c沒有初始化就使用*/
16.(gdb) p c
17.$1 = (int *) 0x0 /*驗證一下，c是null pointer*/
18.(gdb) p *c   /*驗證一下，null pointer不可以讀寫*/
19.Cannot access memory at address 0x0
```

使用gdb除錯

```
20. (gdb) n
21. sighandler (signumber=4) at seg_fault_recover.c:8
22. 8  void sighandler(int signumber) {
23. (gdb) n
24. Breakpoint 2, sighandler (signumber=11) at seg_fault_recover.c:9
25. 9       printf("get a signal named '%d', '%s'\n", signumber,
    sys_siglist[signumber]);
26. (gdb) n
27. get a signal named '11', 'Segmentation fault'
28. 10      c=(int*)malloc(sizeof(int)); /*替c分配了記憶體*/
29. (gdb) n
30. 11 }
```

使用gdb除錯

```
31.(gdb) p c      /*分配完以後，c是否從null pointer變成有意義的*/
32.$2 = (int *) 0x602420
33.(gdb) p *c     /*列印c變數的值，發現真的可以讀取*/
34.$3 = 0
35.(gdb) c        /*繼續執行，理論上回到main後，*c可以讀寫*/
36.Continuing.
37.Program received signal SIGSEGV, Segmentation fault.
38.0x000000000040077a in main (argc=1, argv=0x7fffffffdd88)
39.     at seg_fault_recover.c:14
40.14     *c = 0xF0FE;/*c沒有初始化就使用*/
41./*現在c已經在sighandler初始化了，為何還是SIGSEGV*/
```

使用gdb除錯

```
42.  (gdb) disassemble main
Dump of assembler code for function main:
   0x0000000000400736 <+0>:     push   %rbp
   0x0000000000400737 <+1>:     mov    %rsp,%rbp
   0x000000000040073a <+4>:     sub    $0x10,%rsp
   0x000000000040073e <+8>:     mov    %edi,-0x4(%rbp)
   0x0000000000400741 <+11>:    mov    %rsi,-0x10(%rbp)
   0x0000000000400745 <+15>:    mov    $0x4006f6,%esi
   0x000000000040074a <+20>:    mov    $0xb,%edi
   0x000000000040074f <+25>:    callq  0x4005d0 <signal@plt>
   0x0000000000400754 <+30>:    cmp    $0xffffffffffffffff,%rax
   0x0000000000400758 <+34>:    jne    0x400773 <main+61>
   0x000000000040075a <+36>:    mov    $0x40089f,%ecx
   0x000000000040075f <+41>:    mov    $0xd,%edx
   0x0000000000400764 <+46>:    mov    $0x400847,%esi
   0x0000000000400769 <+51>:    mov    $0x400860,%edi
   0x000000000040076e <+56>:    callq  0x4005a0 <__assert_fail@plt>
   0x0000000000400773 <+61>:    mov    0x200af6(%rip),%rax        # 0x601270 <c>
⇒  0x000000000040077a <+68>:    movl   $0xc0fe,(%rax)
   /*因c的位址早被載入到rax暫存器，因此重新執行0x40077a行程式碼，也無效（不會重新載入）*/
   0x0000000000400780 <+74>:    mov    $0x400887,%edi
   0x0000000000400785 <+79>:    callq  0x400580 <puts@plt>
   0x000000000040078a <+84>:    callq  0x4005c0 <getchar@plt>
   0x000000000040078f <+89>:    mov    $0x0,%eax
```

使用kdbg（seg_fault_recover.c）

```
12 int main(int argc, char **argv) {
13     assert(signal(SIGSEGV, sighandler) != SIG_ERR);
14     *c = 0xC0FE;/*c沒有初始化就使用*/
  0x8ac mov    0x200765(%rip),%rax        # 0x201018 <c>
  0x8b3 movl   $0xc0fe,(%rax)
15     printf("press Enter to continue\n");
16     getchar();
17 }
```

seg_fault2.c：修復記憶體錯誤（高手寫的程式碼，僅供參考）

```
1.  /*作者 https://www.facebook.com/scottt.tw*/
2.  /*修改 羅習五*/
3.  #define _GNU_SOURCE
4.  #include <unistd.h>
5.  #include <assert.h>
6.  #include <signal.h>
7.  #include <stdio.h>
8.  #include <stdlib.h>
9.  #include <sys/ucontext.h>
```

seg_fault2.c：修復記憶體錯誤（高手寫的程式碼，僅供參考）

```
10.   /* Define 'c' as a global register variable
11.      https://gcc.gnu.org/onlinedocs/gcc/Global-Register-Variables.html */
12.   register unsigned long *c __asm__("r12");
13.
14.   void sighandler(int signumber, siginfo_t *sinfo, void *ucontext) {
15.      ucontext_t *context = ucontext;
16.
17.      printf("got a signal %d(%s)\n", signumber,
18.      sys_siglist[signumber]);
19.
20.      /* NOTE: assigning to 'c' instead of 'REG_R12' likely won't work on most systems
21.         due to register content restoration after a signal handler returns */
22.      context->uc_mcontext.gregs[REG_R12] = (unsigned long) malloc(sizeof(char));
23.   }
```

seg_fault2.c：修復記憶體錯誤（高手寫的程式碼，僅供參考）

```
24.  __attribute__((optimize("Os")))
25.  int main(int argc, char **argv) {
26.      int r;
27.      struct sigaction sa = { {0} };
28.      sa.sa_flags = SA_SIGINFO;
29.      sa.sa_sigaction = sighandler;
30.      r = sigaction(SIGSEGV, &sa, NULL);
31.      assert(r == 0);
32.      *c = 0xC0FE; /*segmentation fault*/
33.      //printf("my pid is %d\n", getpid());
34.      printf("press any key to resume\n");
35.      getchar();
36.      return 0;
37.  }
```

使用kdbg（seg_fault2.c)

```
25
26 __attribute__((optimize("Os")))
27 int main(int argc, char **argv) {
28     int r;
29     struct sigaction sa = { {0} };
30     sa.sa_flags = SA_SIGINFO;
31     sa.sa_sigaction = sighandler;
32     r = sigaction(SIGSEGV, &sa, NULL);
33     assert(r == 0);
34     *c = 0xC0FE; /*segmentation fault*/
   0x4006df movq   $0xc0fe,(%r12)
35     //printf("my pid is %d\n", getpid());
36     printf("press any key to resume\n");
37     getchar();
38     return 0;
39 }
40
```

執行結果

```
$ ./seg_fault2
got a signal 11(Segmentation fault) /*在sighandler中修復了錯誤*/
/*因此只出現一次11(Segmentation fault) */
press any key to resume
```

應用： myShell.c

- 要增加的功能
  - 當使用者按下ctr-c不會中斷myShell
  - 如果使用者正在執行外部指令，按下ctr-c，終止該外部指令

預備知識：atomic data type

- 這個型態（sig_atomic_t）保證設值時可以一次設定完成
- 其他資料型態就不一定，以long long為例

如果沒用 sig_atomic_t

Source code

```
long long test=0;
int main() {
    test = 0xc0fec0fec0fec0fe;
```

X86-64

```
mov rax, 0xc0fec0fec0fec0fe
mov QWORD PTR test[rip], rax
```

設定值的部分只有一行即：
`mov QWORD PTR test[rip], rax`
因此可能的情況只有二種：

- 設定前
- 設定後

如果沒用 sig_atomic_t

Source code

```
long long test=0;
int main() {
```

設定值的部分有二行即：

7. sw $3,4($4)
8. sw $2,0($4)

因此可能的情況只有三種：

- 設定前
- 設定後
- 設定到一半

mips

```
1.  lw   $4,%got(test)($28)
2.  li   $3,-1057095680
#-1057095680 := 0xffffffffc0fe0000
3.  movz   $31,$31,$0
4.  ori   $3,$3,0xc0fe
5.  li   $2,-1057095680
#-1057095680 := 0xffffffffc0fe0000
6.  ori   $2,$2,0xc0fe
7.  sw   $3,4($4)
8.  sw   $2,0($4)
9.  move   $2,$0
```

以MIPS指令集為例

- test = 0xc0fec0fec0fec0fe;
- 編譯成八道組合語言，其中最後二道組合語言真正對test設定值
- 先設定後32個位元，再設定前32個位元

```
7.  sw   $3,4($4)
8.  sw   $2,0($4)
```

- 如果在第七行和第八行之間使用者按下ctr-c，那麼signal handler讀到的值是

test = 0x00000000c0fec0fe;

請注意，按照我們程式碼的邏輯test只有二種可能

```
1.  test = 0x0000000000000000      //未設定前
2.  test = 0xc0fec0fec0fec0fe  //設定後
```

再回過頭來看sig_atomic_t

- 資料的更新可能不是一道指令就可以完成
  - long long為例，x86可以一道指令完成設定
  - 但，MIPS卻需要二道指令
- 使用sig_atomic_t可以保證在所有的CPU、compiler上，都是一道指令完成
- 因此不會有設定一半的情況

- 關於atomic的更詳細討論會在pthread章節介紹

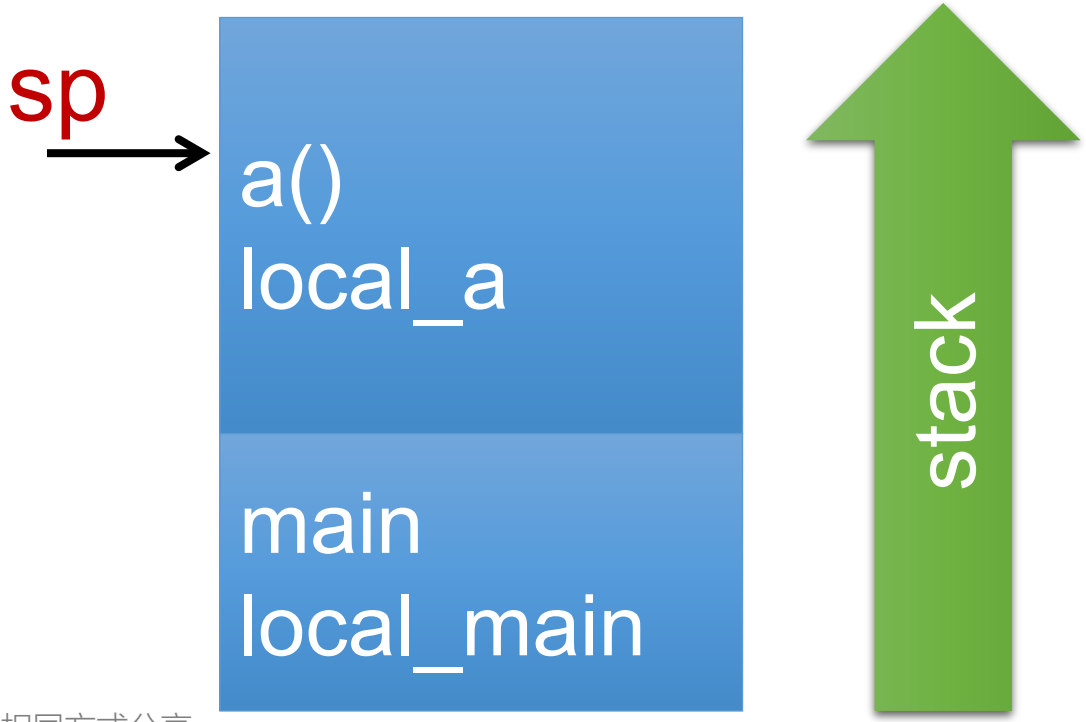
預備知識： setjmp

```
jmp_buf bookmark;
main() {
    int local_main;
    setjmp(bookmark);
    a();
}
void a() {
    int local_a;
    b();
}
void b() {
    int local_c;
    c();
    longjmp(bookmark);
}
```

setjmp

```
jmp_buf bookmark;
main() {
    int local_main;
    setjmp(bookmark);
    a();
}
void a() {
    int local_a;
    b();
}
void b() {
    int local_c;
    c();
    longjmp(bookmark);
}
```

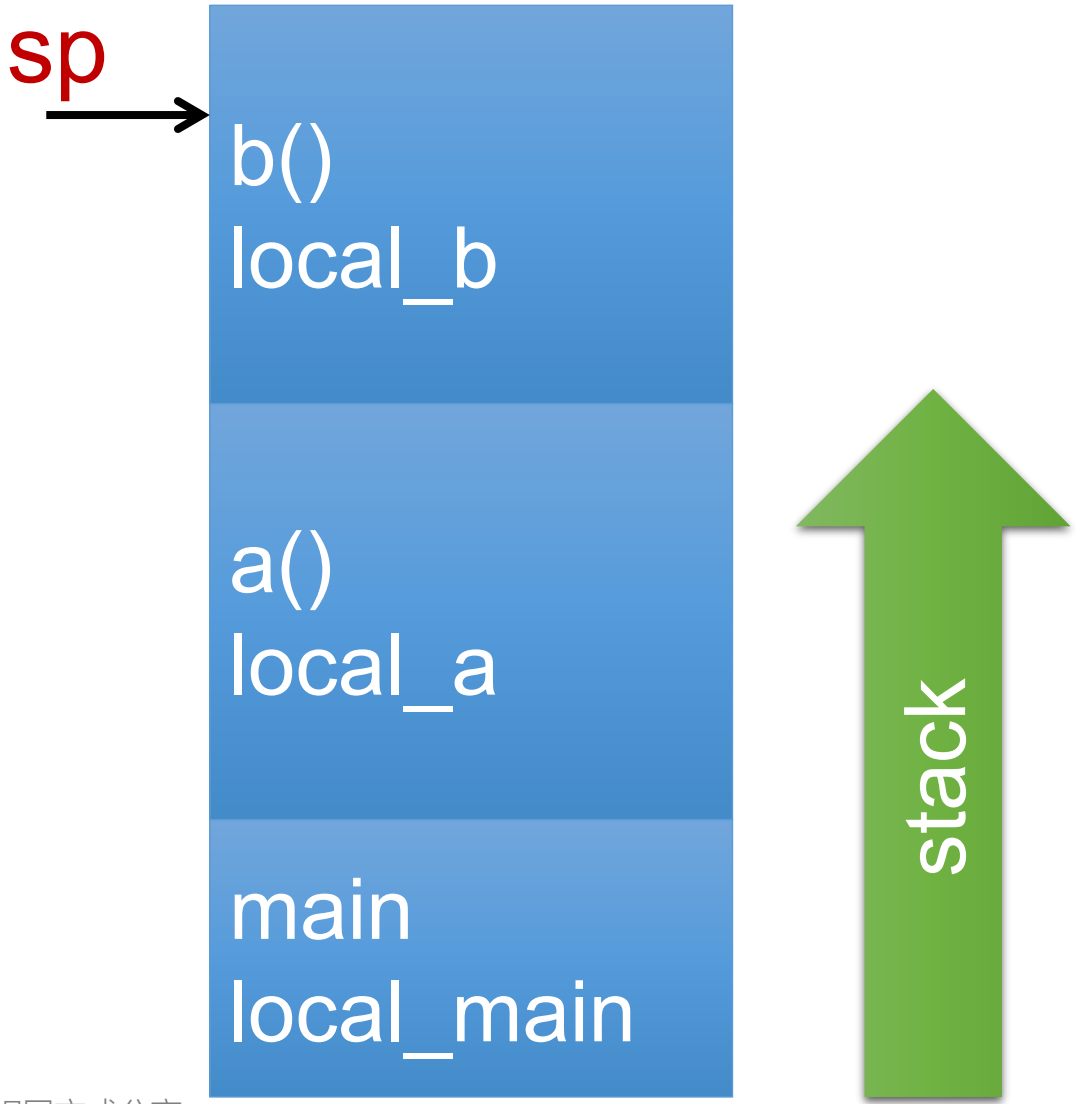
pc



setjmp

```
jmp_buf bookmark;
main() {
    int local_main;
    setjmp(bookmark);
    a();
}
void a() {
    int local_a;
    b();
}
void b() {
    int local_c;
    c();
    longjmp(bookmark);
}
```

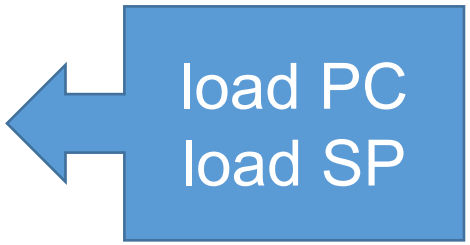
pc

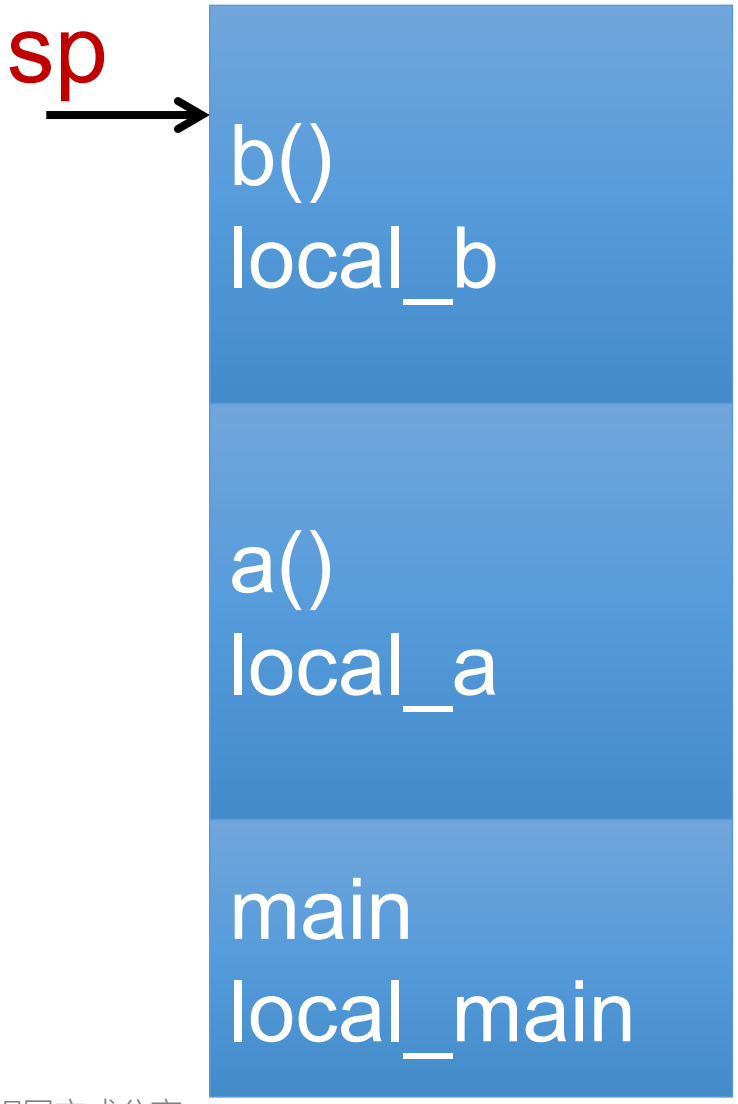


setjmp

```
jmp_buf bookmark;
main() {
    int local_main;
    setjmp(bookmark);
    a();
}
void a() {
    int local_a;
    b();
}
void b() {
    int local_c;
    c();
    longjmp(bookmark);
}
```

pc

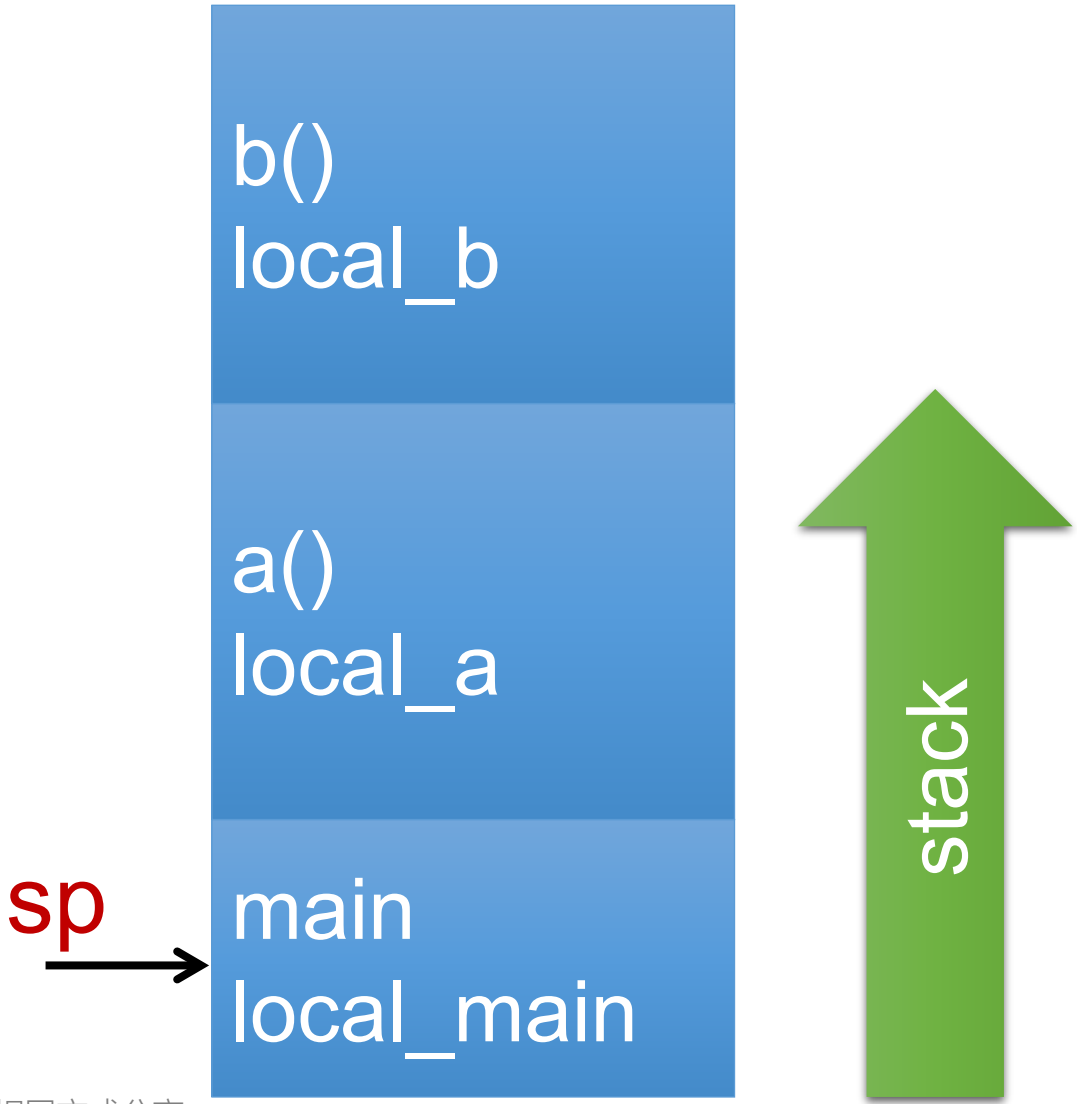




longjmp

```
jmp_buf bookmark;
main() {
    int local_main;
    setjmp(bookmark);
    a();
}
void a() {
    int local_a;
    b();
}
void b() {
    int local_c;
    c();
    longjmp(bookmark);
}
```

pc →



setjmp_longjmp.c

```
1.  jmp_buf buf;
2.  int b() {
3.      puts("stat of b");
4.      //回傳值可以是任意數字，
5.      //例如5，但請不要回傳0以免造成混淆
6.      longjmp(buf, 5);
7.      puts("end of b");
8.  }
9.  int a() {
10.     puts("stat of a");
11.     b();
12.     puts("end of a");
13. }
```

```
14. int main(int argc, char** argv) {
15.     int ret;
16.     register int p1=11;
17.     volatile int p2=22;
18.     int p3=33;
19.     p1=1;
20.     p2=2;
21.     p3=3;
22.     //回傳值0有特別用途，代表setjmp成功
23.     if ((ret=setjmp(buf)) == 0)
24.         a();
25.     else {
26.         printf("return form longjmp."
27.                " the return value is %d\n", ret);
28.         printf("p1 = %d, p2 = %d, p3 = %d\n",
29.                                 p1, p2, p3);
30.     }
31. }
```

結果

```
$ ./setjmp_longjmp
stat of a
stat of b
return form longjmp. the return value is 5
p1 = 1, p2 = 2, p3 = 3
```

setjmp_longjmp.c

```
1.  jmp_buf buf;
2.  int b() {
3.      puts("stat of b");
4.      //回傳值可以是任意數字，
5.      //例如5，但請不要回傳0以免造成混淆
6.      longjmp(buf, 5);
7.      puts("end of b");
8.  }
9.  int a() {
10.     puts("stat of a");
11.     b();
12.     puts("end of a");
13. }
```

```
14. int main(int argc, char** argv) {
15.     int ret;
16.     register int p1=11;
17.     volatile int p2=22;
18.     int p3=33;
19.     p1=1;
20.     p2=2;
21.     p3=3;
22.     //回傳值0有特別用途，代表setjmp成功
23.     if ((ret=setjmp(buf)) == 0)
24.         a();
25.     else {
26.         printf("return form longjmp."
27.                " the return value is %d\n", ret);
28.         printf("p1 = %d, p2 = %d, p3 = %d\n",
29.                                 p1, p2, p3);
30.     }
31. }
```

結果

```
$ ./setjmp_longjmp
stat of a
stat of b
return form longjmp. the return value is 5
p1 = 1, p2 = 2, p3 = 3
/*也有可能跑出底下的結果*/
p1 = 11, p2 = 2, p3 = 33
/*唯一可以確定的是p2，因為p2宣告為volatile*/
```

結果（可能受到編譯器、函數庫的影響）

gcc setjmp_longjmp.c

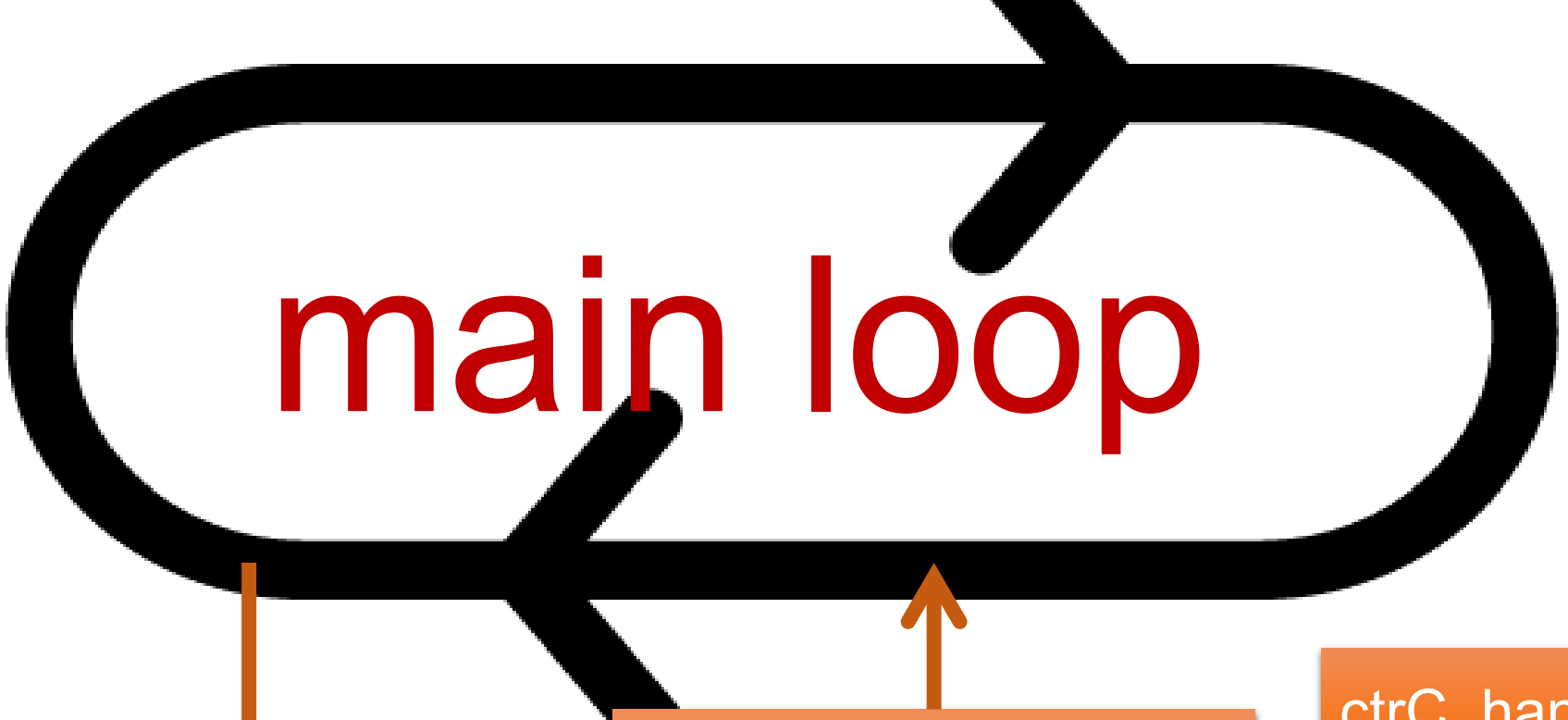
```
stat of a
stat of b
return form longjmp. the
return value is 5
p1 = 1, p2 = 2, p3 = 3
```

gcc -O3 setjmp_longjmp.c

```
stat of a
stat of b
return form longjmp. the
return value is 5
p1 = 1, p2 = 2, p3 = 33
//有些編譯器 p1會等於 11
//只有宣告為nonvolatile的
變數的值是確定更新的
```

sig_setjmp & sig_longjmp

除了儲存PC、SP以外
還儲存signal的狀態（是否被mask）



告知OS遇到ctr-c的時候要呼叫「ctrC_handler」

```
while(1) {
   setjmp(buf)
   cmd = gets();
   if(cmd==“^C”)
      continue;
   else
      execve(“cmd”)
}
```

ctrC_handler()

```
kill child?
unget(“^C”)
longjmp(buf)
```

myShell.c

```
1.  sigjmp_buf jumpBuf;
2.  volatile sig_atomic_t hasChild = 0;
3.  pid_t childPid;

4.  void ctrC_handler(int sigNumber) {
5.      if (hasChild) {
6.          kill(childPid, sigNumber);
7.          hasChild = 0;
8.      } else if (argVect[0] == NULL) {
9.          /*底下程式碼將signal轉成字串^c丟回給主迴圈*/
10.         ungetc('\n', stdin);ungetc('c', stdin);ungetc('^', stdin);
11.         siglongjmp(jumpBuf, 1);
12.     } else fprintf(stderr, "info, 處理字串時使用者按下ctr-c\n");
13. }
```

```
1.  int main (int argc, char** argv) {
2.      signal(SIGINT, ctrC_handler);          /*程式碼註冊ctr-c signal的處理方式*/
3.      signal(SIGQUIT, SIG_IGN);  /*程式碼註冊ctr-\ signal的處理方式*/
4.      signal(SIGTSTP, SIG_IGN);  /*程式碼註冊ctr-z signal的處理方式*/
5.      while(1) {
6.          hasChild = 0;//設定化hasChild, argVect[0]，避免發生race condtion
7.          argVect[0]=NULL;
8.          sigsetjmp(jumpBuf, 1);//設定從signal返回位置
9.          fgets(cmdLine, 4096, stdin);   //讀取指令
10.         if (strcmp(exeName, "^c") == 0)  //使用者按下control-c，^c是由signal handler放入
11.             continue;
12.         if (pid == fork()) execvp(exeName, argVect); else { //除了exit, cd，其餘為外部指令
13.             childPid = pid; /*通知singal handler，如果使用者按下ctr-c時，要處理這個child*/
14.             hasChild = 1;  /*通知singal handler，正在處理child*/
15.             wait(&wstatus); //等待cild執行結束
16.             if (WIFSIGNALED(wstatus))
17.                 printf("terminated by a signal %d.\n", WTERMSIG(wstatus));
18. }   }   }
```

執行結果

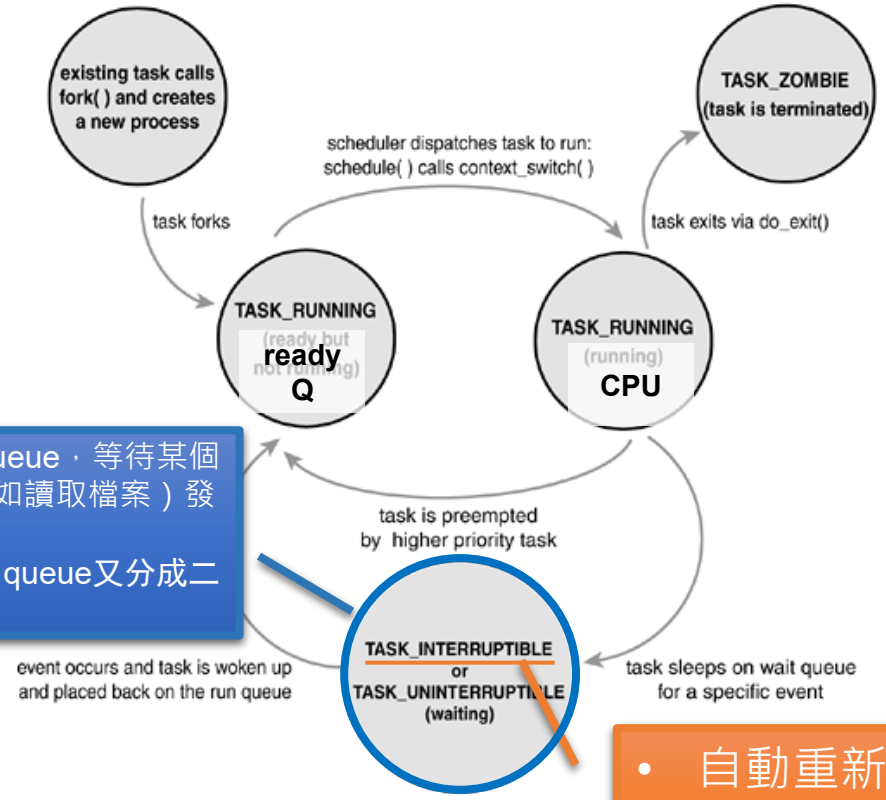
```
shiwulo@NUC ~/sp/ch10 $ ./myShell
shiwulo@NUC:~/Dropbox/course/2018-sp/ch10 >> ls -R / --color
/snap/gnome-3-26-1604/59/usr/share/locale/mr:
total 0
drwxr-xr-x 2 root root 294 Mar 29 21:49 LC_MESSAGES
/snap/gnome-3-26-1604/59/usr/share/locale/mr/LC_MESSAGES:
^Creturn value of ls is 0
the child process was terminated by a signal 2, named Interrupt.
shiwulo@NUC:~/Dropbox/course/2018-sp/ch10>> ^C
shiwulo@NUC:~/Dropbox/course/2018-sp/ch10>>
```

Linux (glibc)的signal特性

特性（Linux）

- 非oneshot
  - 設定好一次signal的行為以後，往後照舊
- 屏蔽當前signal
  - 如果正在處理第X個signal，在處理X的過程中，X會被屏蔽（pending）
- 自動重新啟動被signal中斷的system call
  - 大部分的system call都會被自動「重新啟動」
  - 少部分不會，請參考 man 7 signal，此時失敗的system call會回傳錯誤，且錯誤代碼為EINTR（errno）

何謂「自動啟動」從process的生命流程來看



waiting queue，等待某個事件（例如讀取檔案）發生
在waiting queue又分成二種情況

- 自動重新啟動；或
- 交由使用者決定是否重新啟動

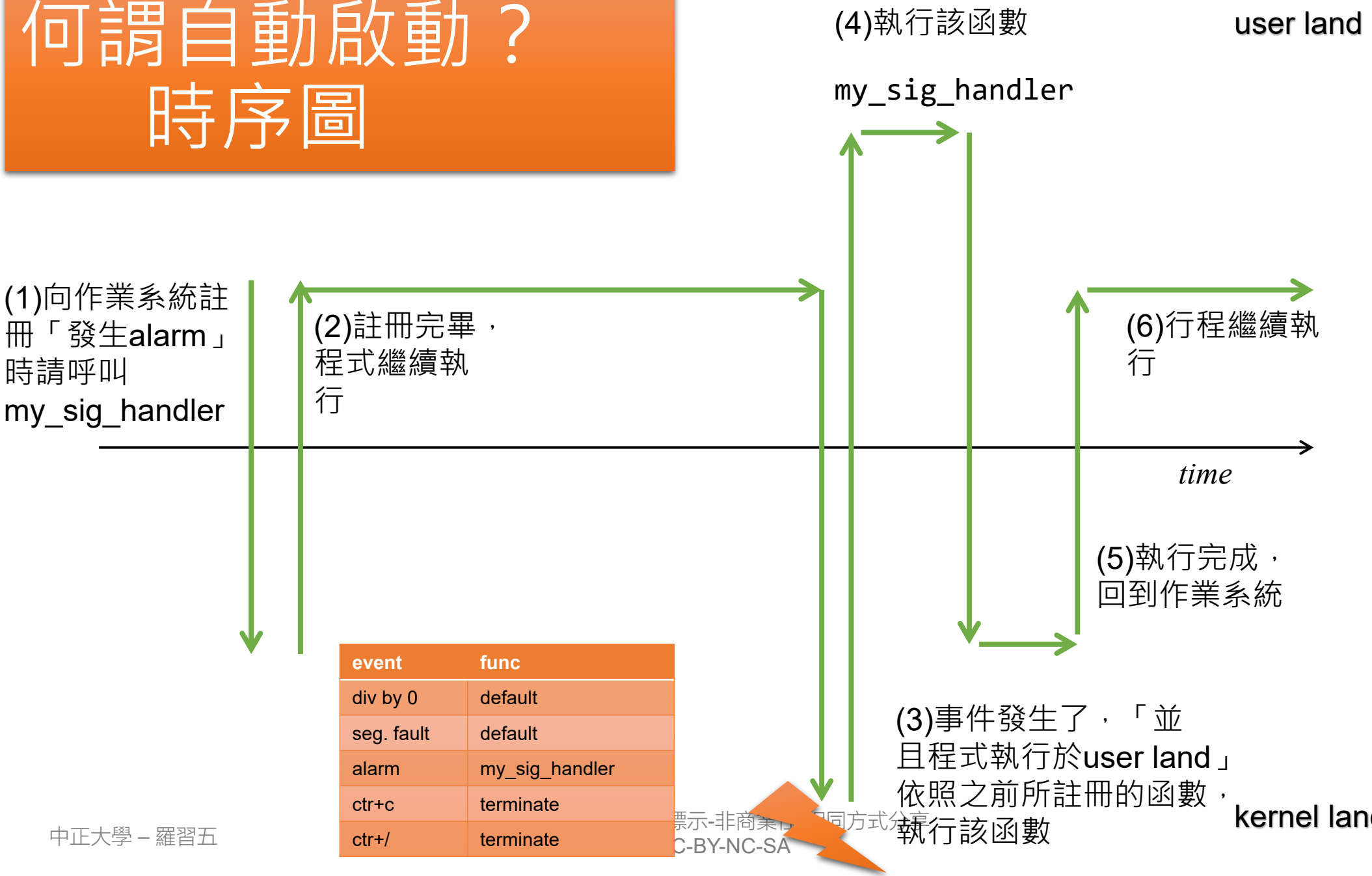
- 在Linux中，task_running分成二種
  - 在ready queue中等待被CPU執行
  - 正在CPU執行
- 如果process需要執行一些特別的處理（例如：從硬碟上讀取檔案）
  - process會進入waiting queue，在waiting queue，如果在waiting queue時，剛好有人送signal給這個process，這時候分為二種情況

1. 這個process不理會signal，繼續讀檔案（task_uninterruptable）
2. 先處理signal，磁碟動作先取消，取消的檔案動作該如何做後續處理呢？（task_interruptable）
   1) OS自動重新啟動該檔案動作
   2) 由使用者決定是否重新啟動

何謂自動啟動？ -- 生活的例子

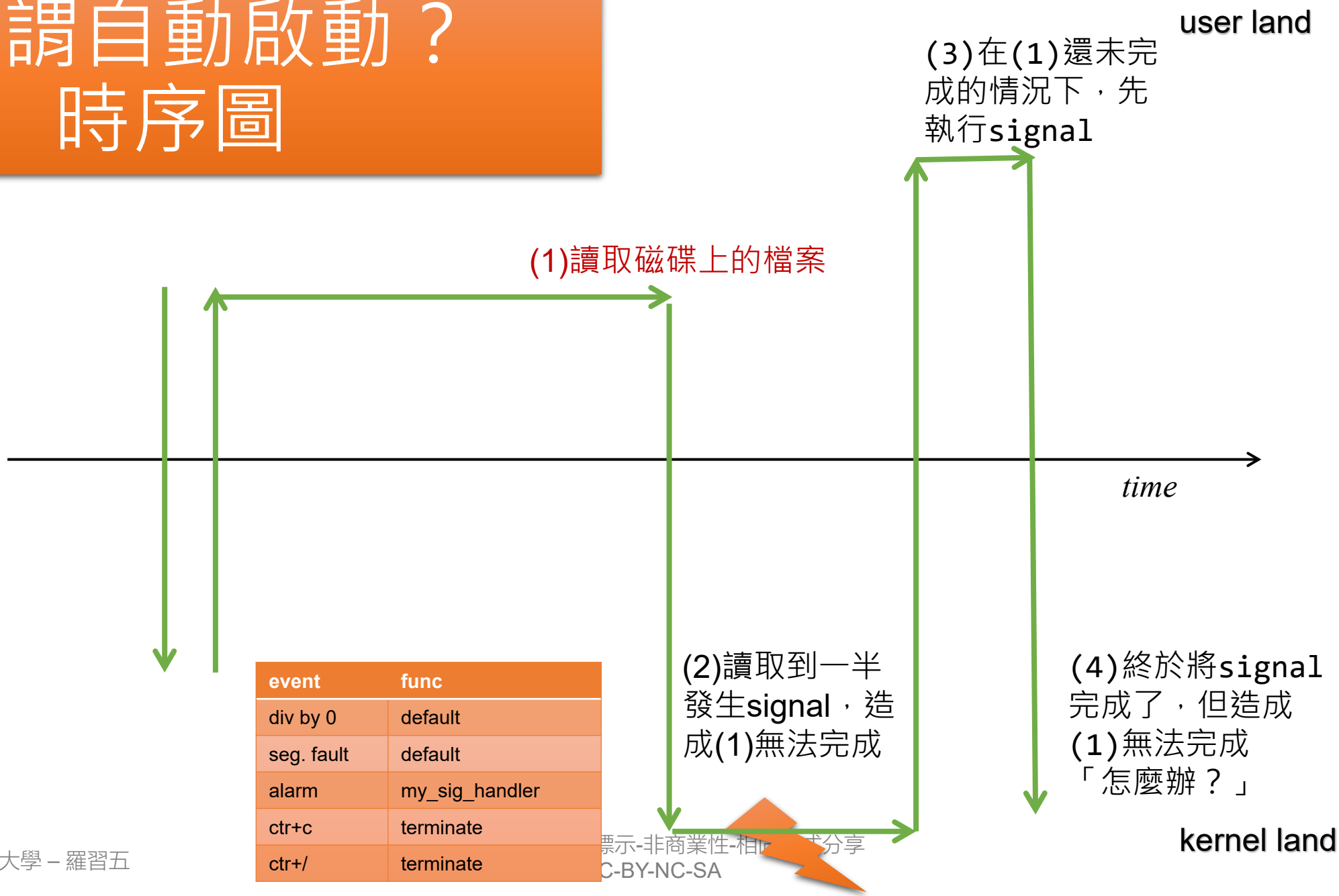
- shiwulo（應用程式）到大吃市買麵；老闆看到Ron博士（該應用程式的插斷，即signal）來了，Ron博士想要立刻買飯，請問老闆要怎樣處理
  - 老闆（作業系統）堅持先做shiwulo的麵（不可中斷）
  - 老闆放棄做到一半的麵（shiwulo的麵），先做Ron博士的麵（可中斷）
    - 麵做好以後有二種情況
    - 老闆自己重新開始煮麵（shiwulo的麵，自動重做）
    - 老闆問Shiwulo要不要重新煮麵（因為shiwulo可能放棄了，不買麵了）（由使用者決定是否重做）

何謂自動啟動？ 時序圖



event	func
div by 0	default
seg. fault	default
alarm	my_sig_handler
ctr+c	terminate
ctr+/	terminate

何謂自動啟動？ 時序圖



event	func
div by 0	default
seg. fault	default
alarm	my_sig_handler
ctr+c	terminate
ctr+/	terminate

如果system call被signal中斷該如何處理（Linux）？

- 大部分的system call都會被自動「重新啟動」
- 少部分不會，請參考 man 7 signal，此時失敗的system call會回傳錯誤，且錯誤代碼為EINTR（errno）

The following interfaces are never restarted after being interrupted by a signal handler, regardless of the use of **SA_RESTART**; they always fail with the error **EINTR** when interrupted by a signal handler:

* "Input" socket interfaces, when a timeout (**SO_RCVTIMEO**) has been set on the socket using **setsockopt**(2): **accept**(2), **recv**(2), **recvfrom**(2), **recvmmsg**(2) (also with a non-NULL timeout argument), and **recvmsg**(2).

* "Output" socket interfaces, when a timeout (**SO_RCVTIMEO**) has been set on the socket using **setsockopt**(2): **connect**(2), **send**(2), **sendto**(2), and **sendmsg**(2).

* Interfaces used to wait for signals: **pause**(2), **sigsuspend**(2), **sigtimedwait**(2), and **sigwaitinfo**(2).

* File descriptor multiplexing interfaces: **epoll_wait**(2), **epoll_pwait**(2), **poll**(2), **ppoll**(2), **select**(2), and **pselect**(2).

* System V IPC interfaces: **msgrcv**(2), **msgsnd**(2), **semop**(2), and **semtimedop**(2).

* Sleep interfaces: **clock_nanosleep**(2), **nanosleep**(2), and **usleep**(3).

* **read**(2) from an **inotify**(7) file descriptor.

* **io_getevents**(2).

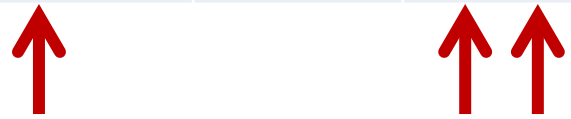
The **sleep**(3) function is also never restarted if interrupted by a handler, but gives a success return: the number of seconds remaining to sleep.

「可靠」與「不可靠」信號

- 在Linux中1~31為不可靠信號
  - 發送的次數與接收的次數會有明顯的不同
- 34-64為「可靠信號」
  - 發送的次數與接收的次數會相同
  - 如果Linux kernel無法負擔「超快速」的signal，**那麼也不是那麼可靠**

不可靠信號在Linux kernel中的實作

Singal#	1	2	3	4	5	6	7	8	9
Set?		1		1					



每一個signal只有一個bit，發生該signal就將該bit設定為1。

如果行程來不及收取該signal。在這樣的情況下（例如4），發生了2次signal，但只會記載1（因此有一個signal不見了）

可靠信號在Linux kernel中的實作

Singal#	1	2	3	4	5	6	7	8	9
queue									

每一個signal有一個queue，當發生signal event的時候，就將該signal event放入queue，例如：4發生5次signal event，就在queue中插入5次。
注意：系統裡面的signal event的個數是有限個，因此如果發生非常多signal event還是會造成部分signal event沒收到

實驗

send_sig不斷的送signal給rec_sig
觀察rt-signal和普通signal的差別

rec_sig.c

```
1.  int nSig[100];

2.  void sighandler(int signumber) {
3.      nSig[signumber]++;
4.  }
5.  int main(int argc, char **argv) {
6.      int sig_exist[100];
7.      int idx = 0;
8.      for (idx = 0; idx < 100; idx++) {
9.          if (signal(idx, sighandler) ==
10.                              SIG_ERR) {
11.             sig_exist[idx] = 0;
12.         } else {
13.             sig_exist[idx] = 1;
14.         }
15.     }
16.     printf("my pid is %d\n", getpid());
17.     getchar();
18.     for (idx=0; idx<100; idx++) {
19.         if (nSig[idx] != 0)
20.             printf("signal #%d, %d
21.                                     times\n", idx,
22.                 nSig[idx]);
23.     }
24. }
```

send_sig.c

```
1.  int main() {
2.      int pid, signum, times;
3.      int i;
4.      printf("process ID\n");
5.      scanf("%d", &pid);
6.      printf("signal number\n");
7.      scanf("%d", &signum);
8.      printf("times\n");
9.      scanf("%d", &times);
10.     for (i=0; i<times; i++)
11.         assert(kill(pid, signum)==0);
12. }
```

結果（好像...）

1. process ID
2. 15745
3. signal number
4. 8
5. times
6. 10000
7. shiwulo@vm:~/sp/ch10$ ./send_sig
8. process ID
9. 15745
10. signal number
11. 50
12. times
13. 10000

1. my pid is 15745
2. press any key to exit

3. signal #8, 757 times
4. signal #50, 10000 times

結果（都不太可靠!!!）

```
1.  shiwulo@vm:~/sp/ch10$ ./send_sig
2.  process ID
3.  17006
4.  signal number
5.  8
6.  times
7.  1000000
8.  shiwulo@vm:~/sp/ch10$ ./send_sig
9.  process ID
10. 17006
11. signal number
12. 50
13. times
14. 1000000
```

```
1.  shiwulo@vm:~/sp/ch10$ ./rec_sig
2.  my pid is 17006
3.  press any key to count the signal number

4.  signal #8, 76430 times
5.  signal #50, 18832 times
```

According to POSIX, an implementation should permit at least **_POSIX_SIGQUEUE_MAX** (32) real-time signals to be queued to a process. However, Linux does things differently. In kernels up to and including 2.6.7, Linux imposes a system-wide limit on the number of queued real-time signals for all processes. This limit can be viewed and (with privilege) changed via the /proc/sys/kernel/rtsig-max file. A related file, /proc/sys/kernel/rtsig-nr, can be used to find out how many real-time signals are currently queued. In Linux 2.6.8, these /proc interfaces were replaced by the **RLIMIT_SIGPENDING** resource limit, which specifies a per-user limit for queued signals; see **setrlimit**(2) for further details.

Real-time signal (SIGRTMIN (34) ~ SIGRTMAX(64))

ulimit -a

```
1.  shiwulo@vm:~$ ulimit -a
2.  core file size          (blocks, -c) 0
3.  data seg size           (kbytes, -d) unlimited
4.  scheduling priority             (-e) 0
5.  file size               (blocks, -f) unlimited
6.  pending signals                 (-i) 15633
7.  max locked memory       (kbytes, -l) 64
8.  max memory size         (kbytes, -m) unlimited
9.  open files                      (-n) 1024
10. pipe size            (512 bytes, -p) 8
11. POSIX message queues     (bytes, -q) 819200
12. real-time priority              (-r) 0
13. stack size              (kbytes, -s) 8192
14. cpu time               (seconds, -t) unlimited
15. max user processes              (-u) 15633
16. virtual memory          (kbytes, -v) unlimited
17. file locks                      (-x) unlimited
```

cat /proc/pid/limits

```
1.  shiwulo@vm:/proc/2199$ cat limits
2.  Limit                     Soft Limit           Hard Limit           Units
3.  Max cpu time              unlimited            unlimited            seconds
4.  Max file size             unlimited            unlimited            bytes
5.  Max data size             unlimited            unlimited            bytes
6.  Max stack size            8388608              unlimited            bytes
7.  Max core file size        0                    unlimited            bytes
8.  Max resident set          unlimited            unlimited            bytes
9.  Max processes             15633                15633                processes
10. Max open files            1024                 4096                 files
11. Max locked memory         65536                65536                bytes
12. Max address space         unlimited            unlimited            bytes
13. Max file locks            unlimited            unlimited            locks
14. Max pending signals       15633                15633                signals
15. Max msgqueue size         819200               819200               bytes
16. Max nice priority         0                    0
17. Max realtime priority     0                    0
18. Max realtime timeout      unlimited            unlimited            us
```

可同時多人呼叫的函數
reentrant function

Reentrant Functions

- 可以在同一個時間點，讓一個行程多次呼叫，而不會產生錯誤的函數
- 這類函數通常
  - 不會存取全域『變數、物件』，或者存取全域物件時，使用鎖定（lock）
  - 只存取堆疊（stack）內的變數、物件

Reentrant Functions

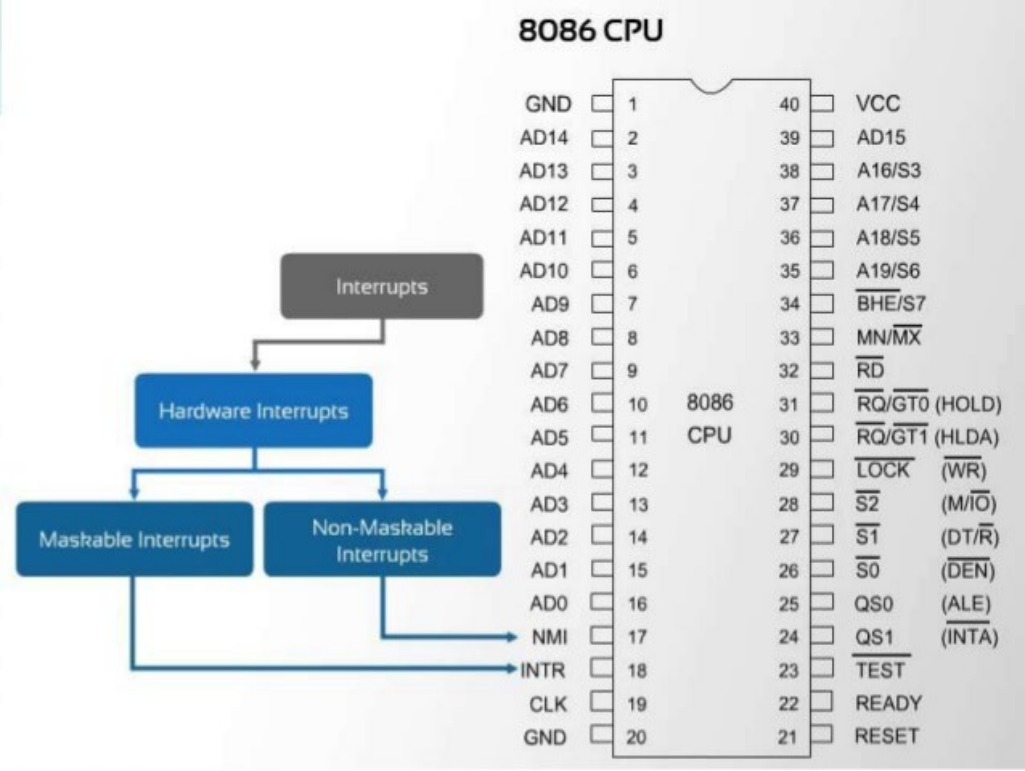
- signal handler本身必須是reentrant function
- signal handler的程式碼只能呼叫reentrant function
- **特別注意**，errno是全域變數，因此在執行signal handler的程式碼之前必須儲存errno的值，執行完signal handler之後必須回存errno

signal-safe functions

- 請參考： man 7 signal

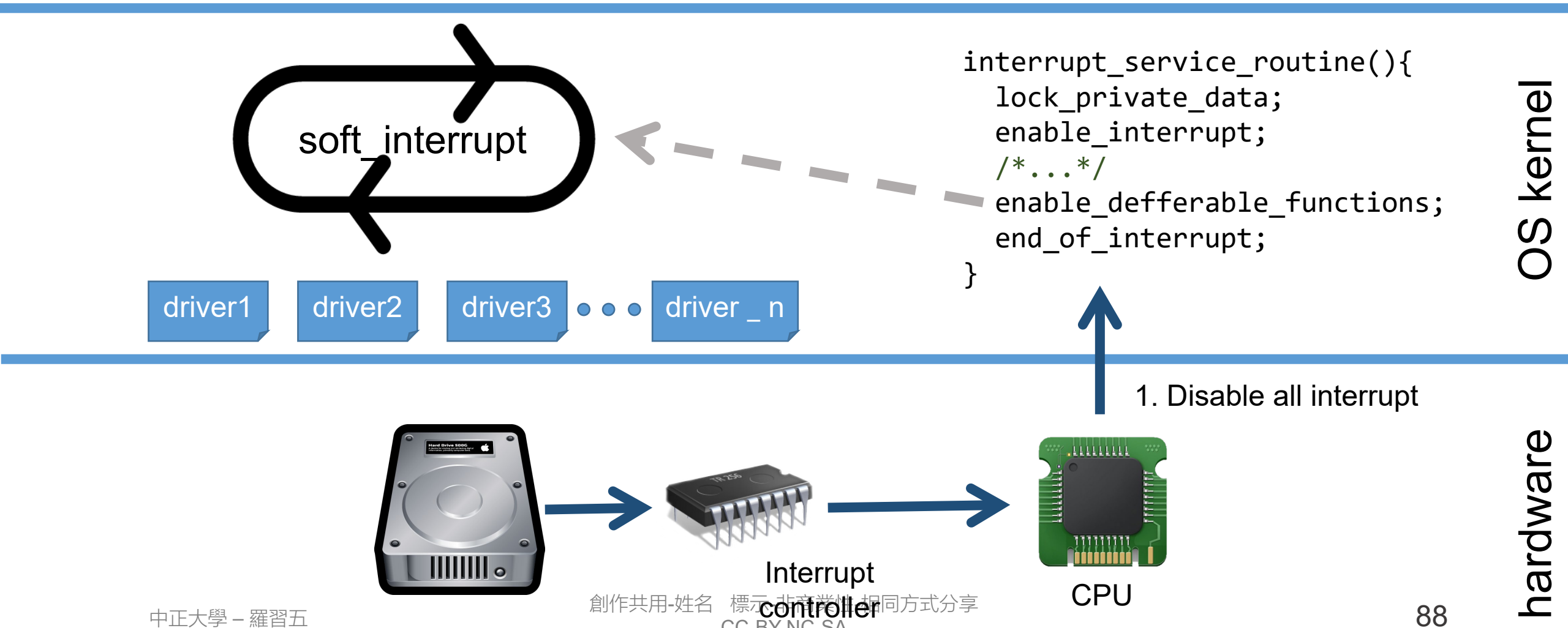
擋掉 signal

singal可以類比「中斷處理器」



- x86的中斷共有15個
  - Signal共有31個
- 中斷發生時x86預設會將其他中斷遮罩住
  - Signal可以用sigprocmask遮罩住其他signal

Interrupt & device driver



sigprocmask()

```
1.  #include <signal.h>
2.  int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

the value of how, as follows

SIG_BLOCK

The set of blocked signals is the union of the current set and the set argument.

SIG_UNBLOCK

The signals in set are removed from the current set of blocked signals. It is permissible to attempt to unblock a signal which is not blocked.

SIG_SETMASK

The set of blocked signals is set to the argument set.

sigprocmask.c

```
1.    void sighandler(int signumber) {
2.        printf("get a signal named '%d', '%s'\n", signumber,
3.                sys_siglist[signumber]);
4.    }
5.
6.    int main(int argc, char **argv) {
7.        sigset_t sigset;
8.        assert(signal(SIGQUIT, sighandler) != SIG_ERR);
9.        /*終止所有的signal*/
10.       sigfillset(&sigset);
11.       sigprocmask(SIG_SETMASK, &sigset, NULL);
```

sigprocmask.c

```
15.     /*睡10秒鐘*/
16.     printf("sleep 10sec\n");
17.     for(int i=0; i<10; i++) {
18.     sleep(1); write(1, "*", 1);
19.     }
20.     printf("\n");
21.     /*重新啓動所有的signal*/
22.     sigemptyset(&sigset);
23.     sigprocmask(SIG_SETMASK, &sigset, NULL);
24.     while (1) {
25.         pause();
26.     }
27. }
```

執行結果

```
shiwulo@NUC:~/sp/ch10$ ./sigprocmask
sleep 10sec
**^C*^C^C^C***^\****

shiwulo@NUC:~/sp/ch10$ ./sigprocmask
sleep 10sec
**^\*^\^\^\^\^\*******
get a signal named '3', 'Quit'
^\get a signal named '3', 'Quit'
^C
```

等待signal

- #include <unistd.h>
- int pause(void);
  - ☘等待任何signal發生
- #include <signal.h>
- int sigsuspend(const sigset_t *mask);
  - ☘等待特定的signal發生
  - ☘用法：先將其他signal用mask遮蓋掉

將signal同步化

同步化的signal處理

```
1. #include <signal.h>
2. int sigwait(const sigset_t *set, int *sig);
3. int sigwaitinfo(const sigset_t *set, siginfo_t *info);
```

- 用set指定要等哪些signal，等到的signal的編號寫入到sig中
- 使用sigwait就不需要signal handler

sigwait.c

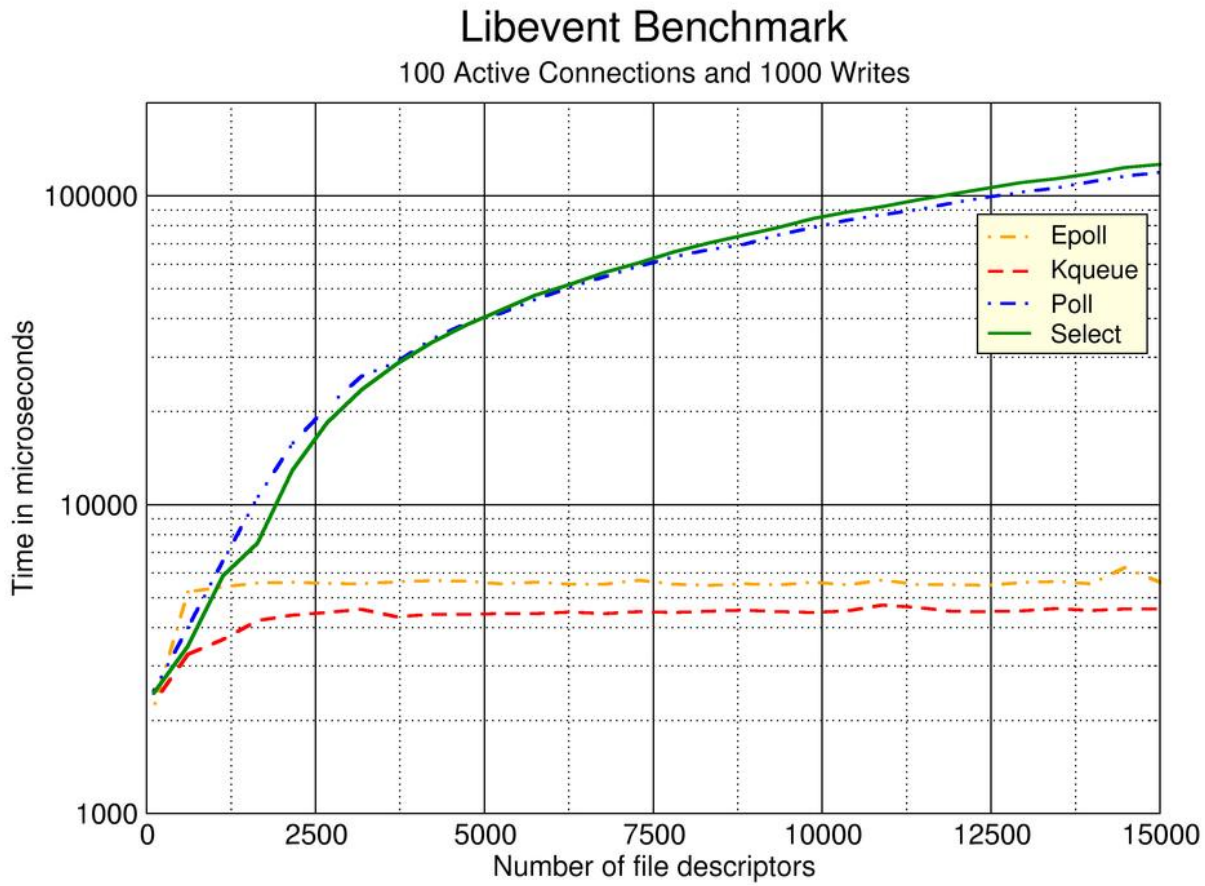
```
1.  int main() {
2.      sigset_t sigset;
3.      int signo;
4.      sigfillset(&sigset);
5.      sigprocmask(SIG_SETMASK, &sigset, NULL);
6.      printf("pid = %d\n", getpid());
7.      while(1) {
8.          assert(sigwait(&sigset, &signo) == 0);
9.          printf("recv sig#%d\n", signo);
10.     }
11. }
```

執行結果

```
shiwulo@vm:~/sp/ch10$
sudo kill -s 31 4188
shiwulo@vm:~/sp/ch10$
sudo kill -s 50 4188
shiwulo@vm:~/sp/ch10$
sudo kill -s 60 4188
```

```
shiwulo@vm:~/sp/ch10$ ./s
igwait
pid = 4188
recv sig#31
recv sig#50
recv sig#60
```


signalfd & I/O
multiplexing

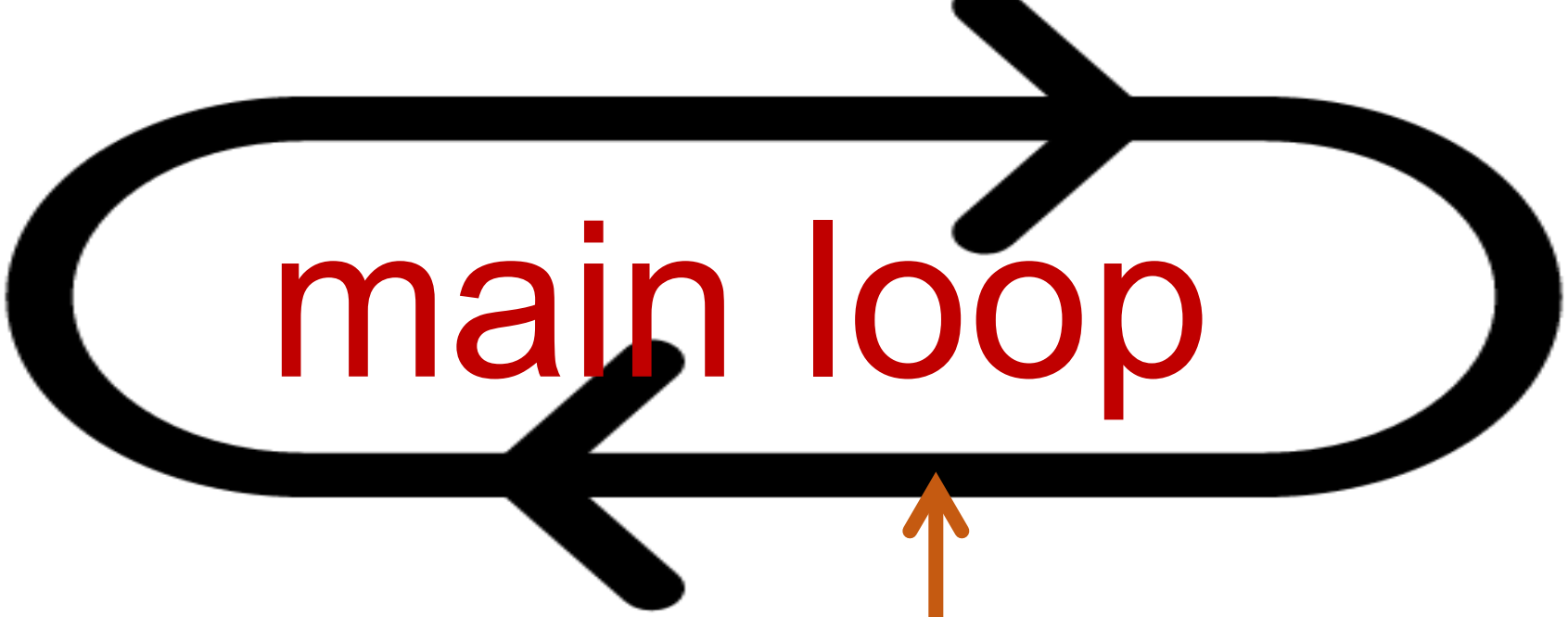


https://monkey.org/~provos/libevent/libevent-benchmark2.jpg

使用signalfd

- int signalfd(int fd, const sigset_t *mask, int flags);

- fd如果為-1則建立一個新的。非-1代表要對該fd進行修改
- mask，希望觀察的signal，先用sigprocmask遮罩起來
- flags可以設定SFD_NONBLOCK、 SFD_CLOEXEC，分別代表nonblocking I/O，呼叫execv等函數時關閉該fd




```
while(1) {
   epoll(keyboard & signal);//wait
   if (fd == keyboard) parseCommand();
   if (fd == signal)
       switch (#signal) {
             case SIGCHLD:/*...*/
             case SIGINT:/*...*/
}     }
```

epoll

- epoll_create：跟系統要一個epoll物件
- epoll_ctl：將感興趣的事件（fd）註冊到epoll裡面去
- epoll_wait：會一直等待（block）直到感興趣的某一個事件（fd）發生

epoll

- `int epoll_create(int size);`

作用：產生一個epoll物件（file descriptor）

參數：在新版的Linux，size用不到

回傳值：epoll物件的fd

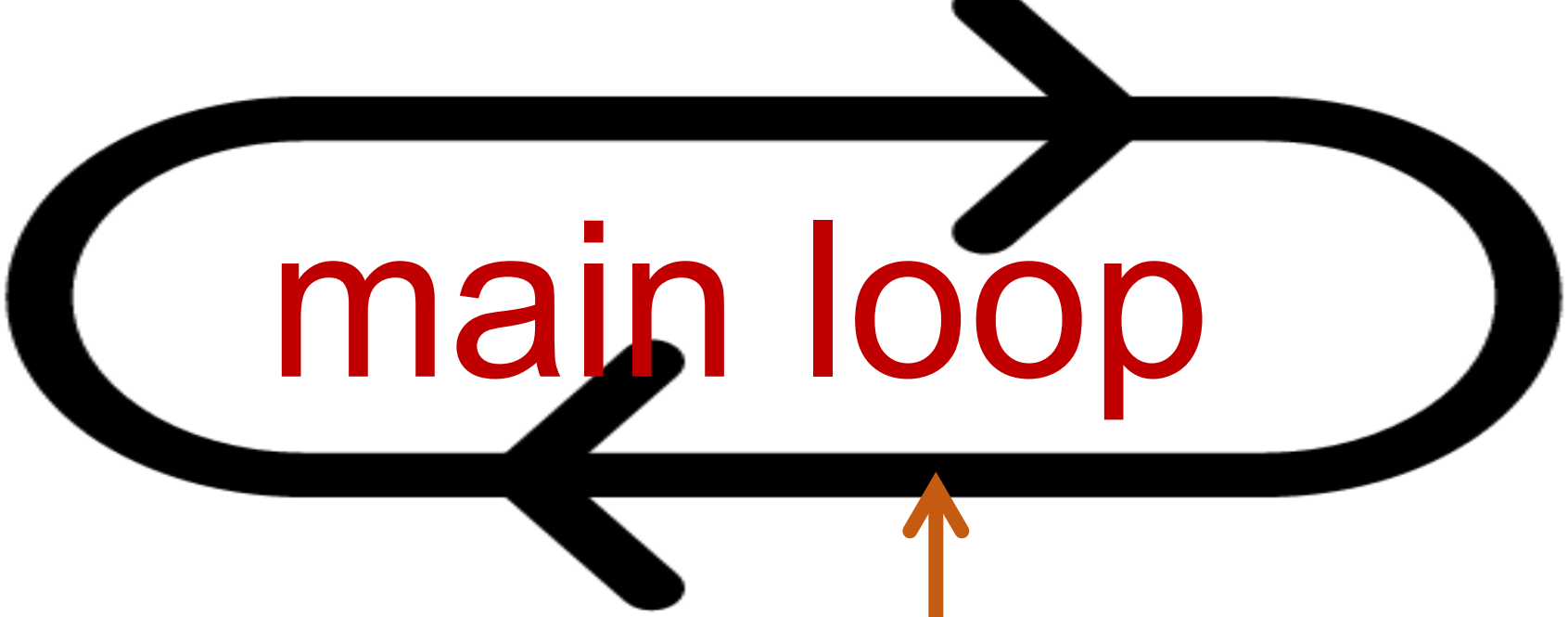
epoll

- int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
- 用法：將fd的event事件op（例如：新增、刪除）到epfd（epoll_create的回傳值）

- struct epoll_event {
- uint32_t events; /* Epoll events */
- epoll_data_t data; /* User data variable */
- };
- events可以填入下面這些值的ORing
- EPOLLIN 可以讀了
- EPOLLOUT 可以寫了
- /*還有很多，請參考man epoll_ctl*/

I/O Multiplexing - epoll

- epollfd = epoll_create1(0);
- //設定要聽的事件
- ev.events = EPOLLIN;　　//聽是否可以read
- ev.data.fd = STDIN_FILENO; //聽鍵盤
- epoll_ctl(epollfd, EPOLL_CTL_ADD, stdin, &ev);
- //一次聽一個事件，沒有timeout
- epoll_wait(epollfd, &event, 1, -1);




```
while(1) {
   epoll(keyboard & signal);//wait
   if (fd == keyboard) parseCommand();
   if (fd == signal)
       switch (#signal) {
             case SIGCHLD:/*...*/
             case SIGINT:/*...*/
}     }
```

shell_sigfd.c

```
1.  int main (int argc, char** argv) {
2.      //設定要監聽的signal
3.      sigset_t sigset;
4.      sigemptyset(&sigset);
5.      sigaddset(&sigset, SIGINT);
6.      sigaddset(&sigset, SIGCHLD);
7.      sigprocmask(SIG_BLOCK, &sigset, NULL);
8.      sig_fd=signalfd(-1, &sigset, 0);
9.      //使用epoll系列函數同時監聽鍵盤和signal
10.     epollfd = epoll_create1(0);
11.     ev.events = EPOLLIN;
12.     ev.data.fd = STDIN_FILENO;  //聽鍵盤
13.     assert(epoll_ctl(epollfd, EPOLL_CTL_ADD, 1, &ev)!=-1);
14.     ev.data.fd = sig_fd;    //聽signal
15.     assert(epoll_ctl(epollfd, EPOLL_CTL_ADD, sig_fd, &ev)!=-1);
```

```
16.     /*無窮迴圈直到使用者輸入exit*/
17.     while(1) {
18.       printPrompt();
19.     wait_event:
20.       //如果有signal或者敲鍵盤epoll會繼續往下執行
21.       epoll_wait(epollfd, &event, 1, -1);
22.       if(event.data.fd == sig_fd) {    //事件是signal
23.         read(sig_fd, &fdsi, sizeof(struct signalfd_siginfo);    //讀取事件相關訊息
24.           switch(fdsi.ssi_signo) {    //判斷signal的編號
25.             case SIGINT:    //按下ctr-c
26.               if (child_pid > 0) {
27.                 int ret=kill(child_pid, fdsi.ssi_signo);
28.                 child_pid = -1;  goto wait_event;
29.               }
30.               break;
31.             case SIGCHLD:   //child執行結束
32.               child_pid = -1;
33.               break;
34.           }
35.           continue;  //如果是signal事件，處理到此就好，繼續下一個迴圈
36.       }
```

```
37. if (event.data.fd == STDIN_FILENO) { //來自鍵盤
38.     int ret = read(STDIN_FILENO, cmdLine, 4096);
39. //因為read不會在字串後面加上'\0'，因此將'\n'換成'\0'，成為標準的C字串
40.     cmdLine[ret - 1] ='\0';
41.     if (child_pid > 0)
42.         goto wait_event; //如果child正在執行，就暫時不處理使用者新的命令
43. }
44. parseString(cmdLine, &exeName);
45. child_pid = vfork(); //除了exit, cd，其餘為外部指令
46. if (child_pid == 0) {
47.   //因為使用vfork，因此child更新startTime會寫到parent的記憶體空間
48.   clock_gettime) CLOCK_REALTIME, &startTime);
49.   //要記得打開signal的遮罩，如果沒打開遮罩，chlild可能會有些signal收不到
50.   sigset_t sigset; sigfillset(&sigset); sigprocmask(SIG_UNBLOCK, &sigset, NULL);
51.     execvp(exeName, argVect);          //產生一個child執行使用者的指令
52. }   }   }
```

功能強大的『sigaction』

sigaction

```
1. #include <signal.h>
2. int sigaction(int signo, const struct sigaction *act, struct sigaction *oact);
3. int sigqueue(pid_t pid, int sig, const union sigval value);
```

1. 如果act不是null表示要修改signal handler，oact不是null的話，表示要將舊有的儲存起來。
2. 跟signal比較起來，因為它可以設定sa_flags，因此他的行為更準確，更適合跨平台

sigaction(UNIX版本)

```
1.  struct sigaction {
2.      /*addr of signal handler or SIG_IGN or SIG_DFL */
3.      void (*sa_handler)(int);
4.      /* additional signals to block */
5.      sigset_t sa_mask;
6.      /* signal options*/
7.      int sa_flags;
8.      /* alternate handler */
9.      void (*sa_sigaction)(int, siginfo_t *, void *);
10. };
```

sigaction（Linux版本）

```
1.  struct sigaction {
2.      /*同signal的第二個參數，處理該signal的函數*/
3.      void    (*sa_handler)(int);
4.      /*加強版的sa_handler*/
5.      void    (*sa_sigaction)(int, siginfo_t *, void *);
6.      /*處理此signal的時候，要暫停處理哪一些signal*/
7.      sigset_t   sa_mask;
8.      /*要如何處理這個signal(後面介紹)*/
9.      int        sa_flags;
10.     /*未定義，不要使用*/
11.     void    (*sa_restorer)(void);
12. };
```

sigaction專屬的signal handler

```
/*
 ucontext_t：signal context information that was saved on the user-    space stack by the kernel
 ucontext_t：與硬體相關，不具有可移植性，例如：AX, BX, CX...暫存器
 siginfo_t：下一張投影片介紹
 */
void handler(int sig, siginfo_t *info, void *ucontext)
{
   /*...*/
}
```

siginfo_t

```
1.      siginfo_t {
2.          int     si_signo;    /* Signal number */
3.          int     si_errno;    /* An errno value */
4.          int     si_code;     /* Signal code */
5.          int     si_trapno;   /* Trap number that caused
6.                                  hardware-generated signal
7.                                  (unused on most architectures) */
8.          pid_t   si_pid;      /* Sending process ID */
9.          uid_t   si_uid;      /* Real user ID of sending process */
10.         int     si_status;   /* Exit value or signal */
11.         clock_t  si_utime;    /* User time consumed */
12.         clock_t  si_stime;    /* System time consumed */
13.         sigval_t si_value;    /* Signal value */
14.         int     si_int;      /* POSIX.1b signal */
15.         void    *si_ptr;     /* POSIX.1b signal */
16.         int     si_overrun;  /* Timer overrun count;
17.                                 POSIX.1b timers */
18.         int     si_timerid;  /* Timer ID; POSIX.1b timers */
19.         void    *si_addr;    /* Memory location which caused fault */
```

siginfo_t

```
20.             long    si_band;    /* Band event (was int in
21.                             glibc 2.3.2 and earlier) */
22.             int     si_fd;      /* File descriptor */
23.             short   si_addr_lsb;  /* Least significant bit of address
24.                             (since Linux 2.6.32) */
25.             void    *si_lower;    /* Lower bound when address violation
26.                             occurred (since Linux 3.19) */
27.             void    *si_upper;    /* Upper bound when address violation
28.                             occurred (since Linux 3.19) */
29.             int     si_pkey;    /* Protection key on PTE that caused
30.                             fault (since Linux 4.6) */
31.             void    *si_call_addr; /* Address of system call instruction
32.                             (since Linux 3.5) */
33.             int     si_syscall;   /* Number of attempted system call
34.                             (since Linux 3.5) */
35.             unsigned int si_arch;  /* Architecture of attempted system call
36.                             (since Linux 3.5) */
37.         }
```

sa_flags

- sa_flags
  - SA_NOCLDSTOP
  - SA_NOCLDWAIT
    - If signum is SIGCHLD, do not transform children into zombies when they terminate.
  - SA_NODEFER
  - SA_ONSTACK
  - SA_RESETHAND
  - **SA_RESTART**
  - **SA_RESTORER**
  - SA_SIGINFO
- 紅色粗體字表示signal預設使用的options

小結

- 如果確定程式碼只會在Linux上執行，那麼signal是一個比較簡單的方法
- singal_handler中能夠呼叫的函數有限，因此可以將主要的處理丟回給主迴圈
- 編號1~31的signal不是「可靠的」signal，請注意「可靠」的含義
- 可以使用signalfd配合epoll同步化signal和其他I/O的處理，signalfd是Linux獨有的
- sigaction除了具有跨平台的優勢以外，siginfo_t也有較多的訊息

作業

- 修改shell_sigfd.c，完成相同的功能，但是使用sigaction()實作
- 執行檔名稱必須是shell_sigaction