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

Signal（5/28）

- SIGALARM
  - 每隔一段時間發送「時間訊號」
- 如何將signal的事件，傳回主程序中?
  - 例如：按下『ctr-c』會將目前的結果儲存，然後離開
  - 要怎樣才能正確地儲存呢？？？
  - 必須在主程序中先將工作告一段落，然後將目前的結果儲存
  - 怎樣做？？？
- 在signal handler中，怎樣處理 system call？？

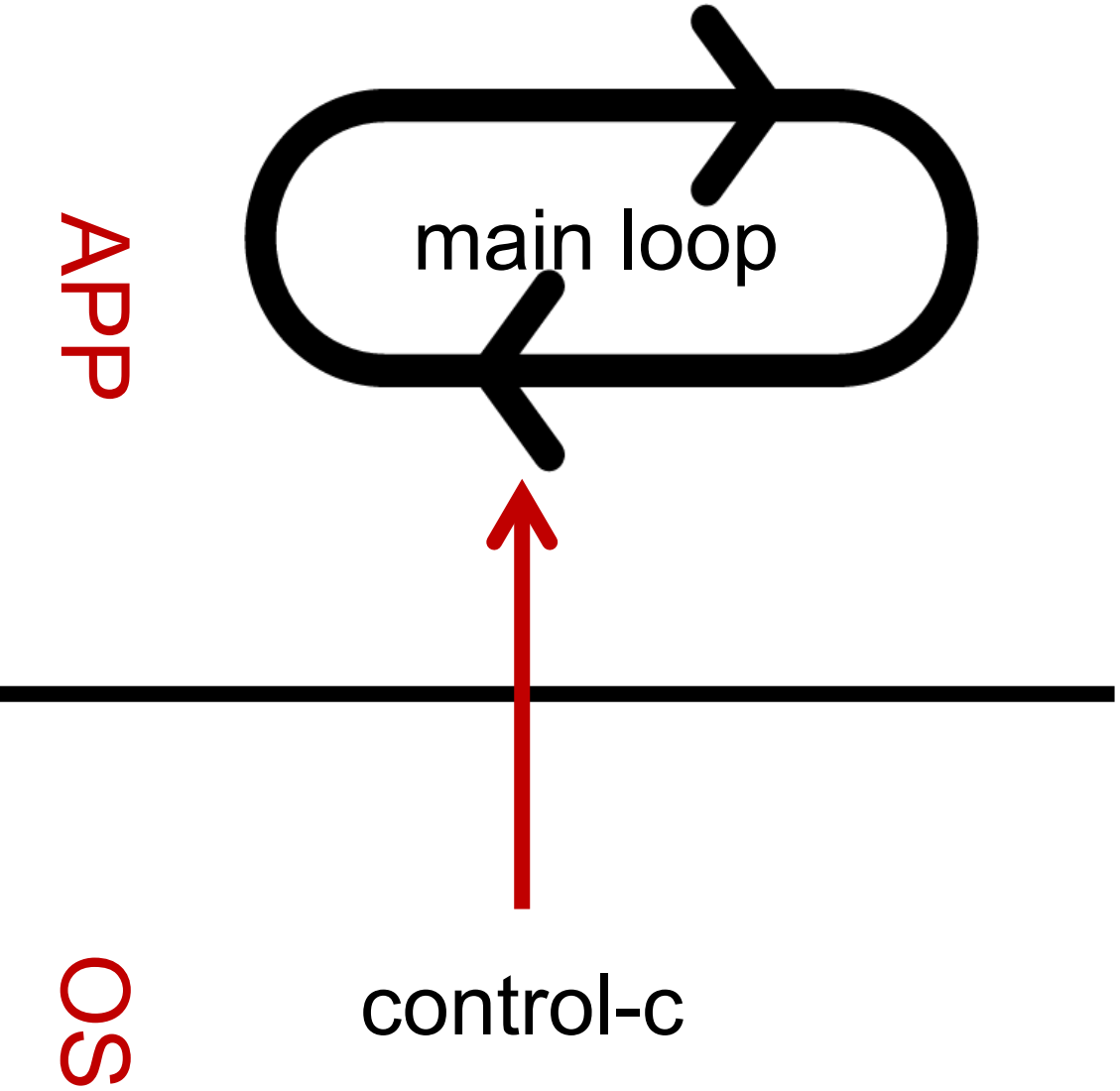
其他（5/28）

- setjmp & longjmp
  - 由『被呼叫者』的『被呼叫者』的...直接跳回去『呼叫者』
- sigaction
  - Signal在不同的作業系統會有不一樣的『語意』
  - Sigaction將所有可能的『語意』變成參數，並且多了一些額外的功能，例如：mask等
- signal fd
  - 將大部分的signal變成檔案一樣

sa_flags (5/28)

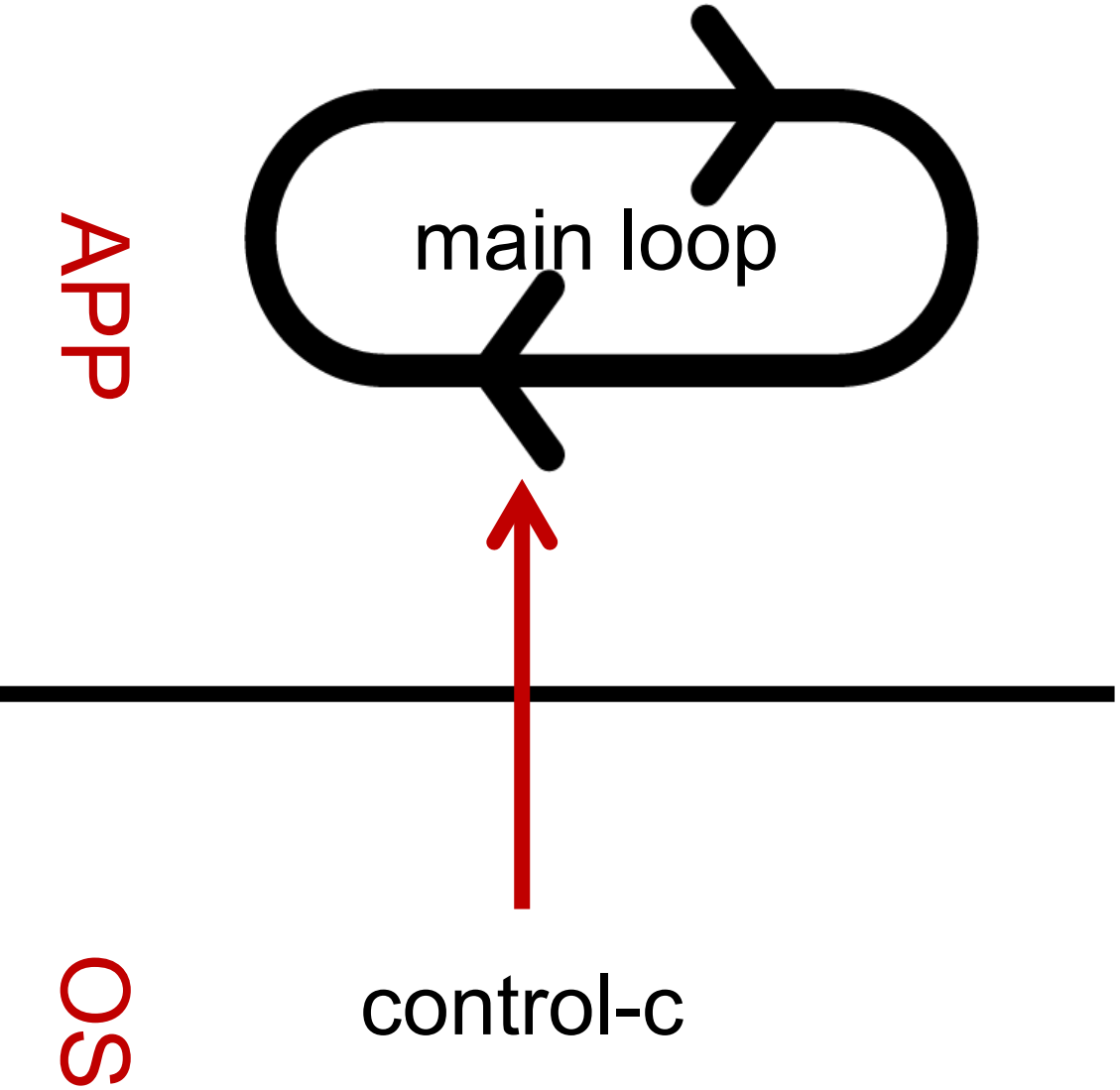
- sa_flags
  - SA_NOCLDSTOP
  - SA_NOCLDWAIT
    - If signum is SIGCHLD, do not transform children into zombies when they terminate.
  - SA_NODEFER
  - SA_ONSTACK
  - SA_RESETHAND
  - **SA_RESTART**
  - **SA_RESTORER**
  - SA_SIGINFO
- 紅色粗體字表示signal預設使用的options

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

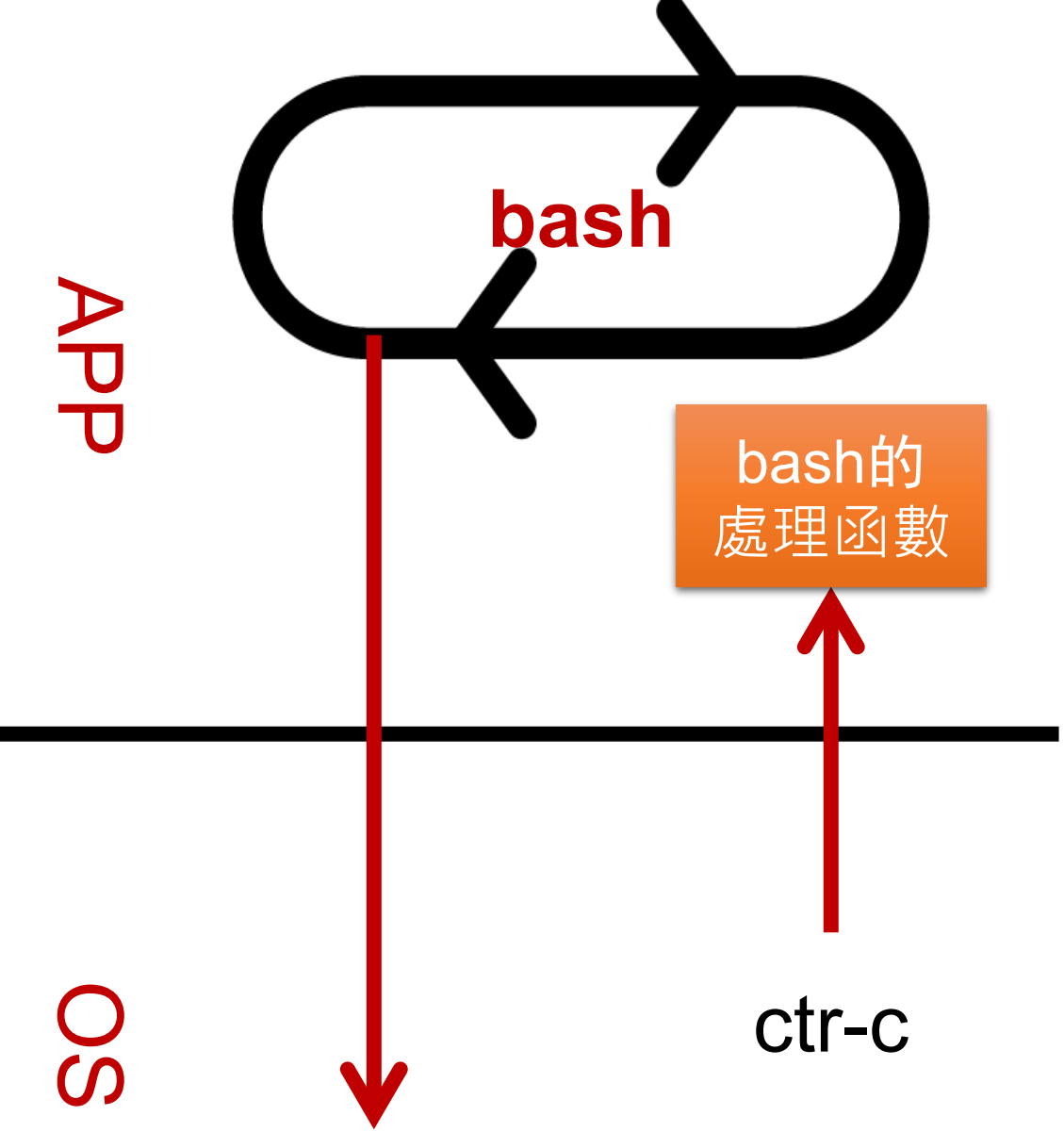
範例

bash

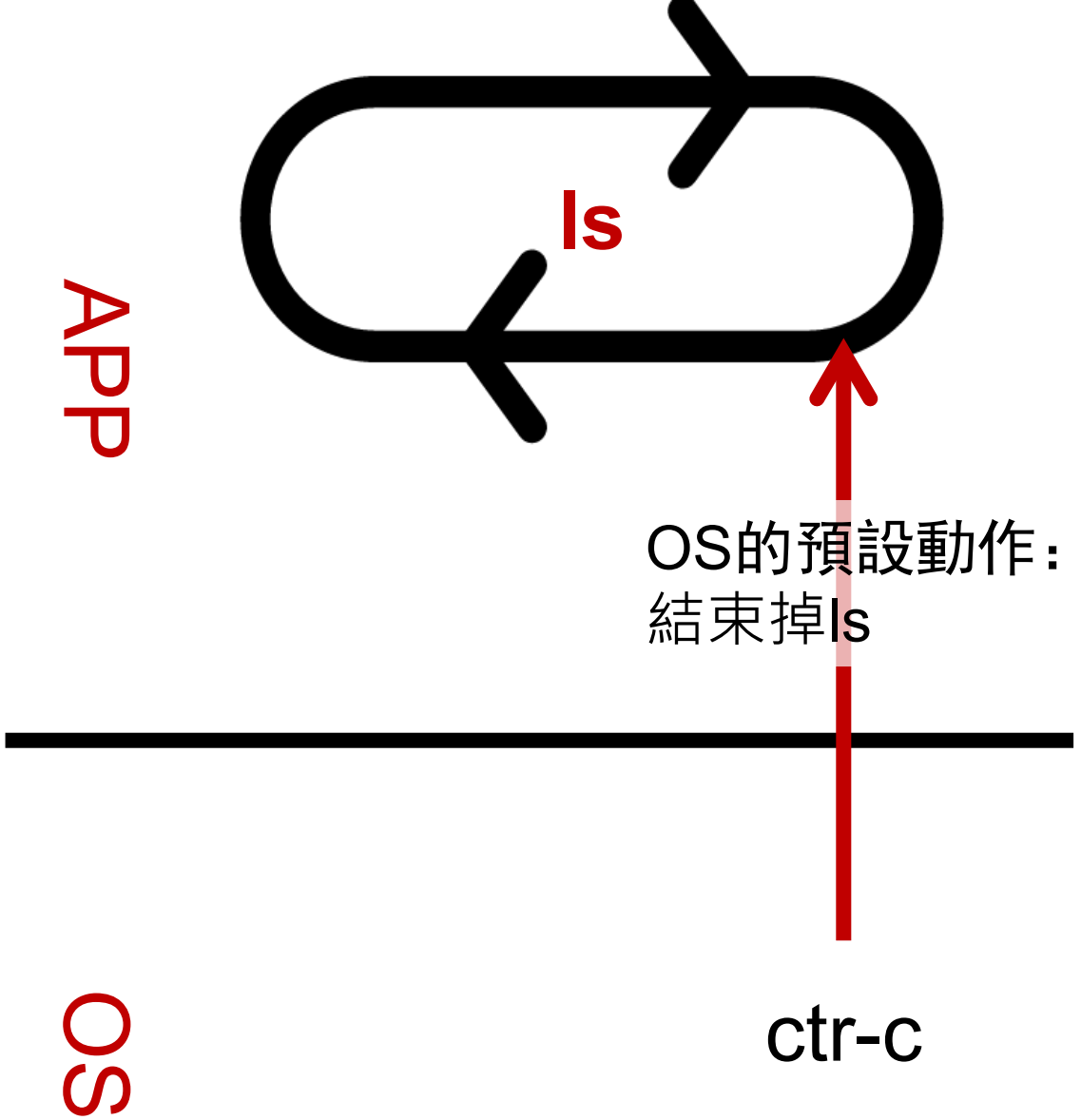
shiwulo@NUC:~$ ^C
shiwulo@NUC:~$ ^C
shiwulo@NUC:~$ ^C
/*按下ctr-c以後沒有反應*/

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
```

5/23上課進度

- 2:45~3:00
  - 回家練習「記憶體存取錯誤」
  - 複習「signal與system call」
- 3:00~3:40
  - 上課練習「signal再signal」
- 3:40~4:00
  - 上課練習「setjmp & longjmp」

練習：記憶體存取錯誤

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'*/
```

signal 與 system call


```
void g(int signum) {
   printf(“『ctr-\』\n”);
}
```

```
void f(int signum) {
   printf(“『ctr-c』\n”);
}
```

```
void main() {
   signal(ctr-c, f);
   signal(ctr-\, g);
   getchar();
   read(fd, buf, 1GB);
   read(fd, buf, 1GB);
}
```

APP

OS scheduler

event	Todo
ctr-c	default
ctr-\	default
kill	default

OS


```
void g(int signum) {
   printf(“『ctr-\』\n”);
}
```

```
void f(int signum) {
   printf(“『ctr-c』\n”);
}
```

```
void main() {
   signal(ctr-c, f);
   signal(ctr-\, g);
   getchar();
   read(fd, buf, 1GB);
   read(fd, buf, 1GB);
}
```

APP

OS scheduler

event	Todo
ctr-c	default
ctr-\	default
kill	default

OS


```
void g(int signum) {
   printf(“『ctr-\』\n”);
}
```

```
void f(int signum) {
   printf(“『ctr-c』\n”);
}
```

```
void main() {
   signal(ctr-c, f);
   signal(ctr-\, g);
   getchar();
   read(fd, buf, 1GB);
   read(fd, buf, 1GB);
}
```

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

OS


```
void g(int signum) {
   printf(“『ctr-\』\n”);
}
```

```
void f(int signum) {
   printf(“『ctr-c』\n”);
}
```

```
void main() {
   signal(ctr-c, f);
   signal(ctr-\, g);
   getchar();
   read(fd, buf, 1GB);
   read(fd, buf, 1GB);
}
```

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
   printf(“『ctr-\』\n”);
}
```

```
void f(int signum) {
   printf(“『ctr-c』\n”);
}
```

```
void main() {
   signal(ctr-c, f);
   signal(ctr-\, g);
   getchar();
   read(fd, buf, 1GB);
   read(fd, buf, 1GB);
}
```

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
   printf(“『ctr-\』\n”);
}
```

```
void f(int signum) {
   printf(“『ctr-c』\n”);
}
```

```
void main() {
   signal(ctr-c, f);
   signal(ctr-\, g);
   getchar();
   read(fd, buf, 1GB);
   read(fd, buf, 1GB);
}
```

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
   printf(“『ctr-\』\n”);
}
```

```
void f(int signum) {
   printf(“『ctr-c』\n”);
}
```

```
void main() {
   signal(ctr-c, f);
   signal(ctr-\, g);
   getchar();
   read(fd, buf, 1GB);
   read(fd, buf, 1GB);
}
```

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
   printf(“『ctr-\』\n”);
}
```

```
void f(int signum) {
   printf(“『ctr-c』\n”);
}
```

```
void main() {
   signal(ctr-c, f);
   signal(ctr-\, g);
   getchar();
   read(fd, buf, 1GB);
   read(fd, buf, 1GB);
}
```

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
   printf(“『ctr-\』\n”);
}
```

```
void f(int signum) {
   printf(“『ctr-c』\n”);
}
```

```
void main() {
   signal(ctr-c, f);
   signal(ctr-\, g);
   getchar();
   read(fd, buf, 1GB);
   read(fd, buf, 1GB);
}
```

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
   printf(“『ctr-\』\n”);
}
```

```
void f(int signum) {
   printf(“『ctr-c』\n”);
}
```

```
void main() {
   signal(ctr-c, f);
   signal(ctr-\, g);
   getchar();
   read(fd, buf, 1GB);
   read(fd, buf, 1GB);
}
```

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
   printf(“『ctr-\』\n”);
}
```

```
void f(int signum) {
   printf(“『ctr-c』\n”);
}
```

```
void main() {
   signal(ctr-c, f);
   signal(ctr-\, g);
   getchar();
   read(fd, buf, 1GB);
   read(fd, buf, 1GB);
}
```

從被ctr-c打斷的那一行繼續

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
   printf(“『ctr-\』\n”);
}

void f(int signum) {
   printf(“『ctr-c』\n”);
}

void main() {
   signal(ctr-c, f);
   signal(ctr-\, g);
   getchar();
   read(fd, buf, 1GB);
   read(fd, buf, 1GB);
}
```

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
   printf(“『ctr-\』\n”);
}
```

```
void f(int signum) {
   printf(“『ctr-c』\n”);
}
```

```
void main() {
   signal(ctr-c, f);
   signal(ctr-\, g);
   getchar();
   read(fd, buf, 1GB);
   read(fd, buf, 1GB);
}
```

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
   printf(“『ctr-\』\n”);
}
```

```
void f(int signum) {
   printf(“『ctr-c』\n”);
}
```

```
void main() {
   signal(ctr-c, f);
   signal(ctr-\, g);
   getchar();
   read(fd, buf, 1GB);
   read(fd, buf, 1GB);
}
```

read()執行到一半，取消

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
   printf(“『ctr-\』\n”);
}
```

```
void f(int signum) {
   printf(“『ctr-c』\n”);
}
```

```
void main() {
   signal(ctr-c, f);
   signal(ctr-\, g);
   getchar();
   read(fd, buf, 1GB);
   read(fd, buf, 1GB);
}
```

已取消

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
   printf(“『ctr-\』\n”);
}
```

```
void f(int signum) {
   printf(“『ctr-c』\n”);
}
```

```
void main() {
   signal(ctr-c, f);
   signal(ctr-\, g);
   getchar();
   read(fd, buf, 1GB);
   read(fd, buf, 1GB);
}
```

已取消

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
   printf(“『ctr-\』\n”);
}
```

```
void f(int signum) {
   printf(“『ctr-c』\n”);
}
```

```
void main() {
   signal(ctr-c, f);
   signal(ctr-\, g);
   getchar();
   read(fd, buf, 1GB);
   read(fd, buf, 1GB);
}
```

已取消

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
   printf(“『ctr-\』\n”);
}
```

```
void f(int signum) {
   printf(“『ctr-c』\n”);
}
```

```
void main() {
   signal(ctr-c, f);
   signal(ctr-\, g);
   getchar();
   read(fd, buf, 1GB);
   read(fd, buf, 1GB);
}
```

已取消

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
   printf(“『ctr-\』\n”);
}
```

```
void f(int signum) {
   printf(“『ctr-c』\n”);
}
```

```
void main() {
   signal(ctr-c, f);
   signal(ctr-\, g);
   getchar();
   read(fd, buf, 1GB);
   read(fd, buf, 1GB);
}
```

已取消

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
   printf(“『ctr-\』\n”);
}
```

```
void f(int signum) {
   printf(“『ctr-c』\n”);
}
```

```
void main() {
   signal(ctr-c, f);
   signal(ctr-\, g);
   getchar();
   read(fd, buf, 1GB);
   read(fd, buf, 1GB);
}
```

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
   printf(“『ctr-\』\n”);
}
```

```
void f(int signum) {
   printf(“『ctr-c』\n”);
}
```

```
void main() {
   signal(ctr-c, f);
   signal(ctr-\, g);
   getchar();
   read(fd, buf, 1GB);
   read(fd, buf, 1GB);
}
```

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
   printf(“『ctr-\』\n”);
}
```

```
void f(int signum) {
   printf(“『ctr-c』\n”);
}
```

```
void main() {
   signal(ctr-c, f);
   signal(ctr-\, g);
   getchar();
   read(fd, buf, 1GB);
   read(fd, buf, 1GB);
}
```

深入探討：
getchar()使用write實現，讀取stdin，因此getchar()也被重新啟動

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS

signal再signal

問題描述

- 如果按下ctr-c以後馬上再按一次ctr-c會怎樣?
- 如果按下ctr-c以後馬上再按一次ctr-\會怎樣?
- 理解signal mask

問題描述

- 如果按下ctr-c以後馬上再按一次ctr-c會怎樣?
- **如果按下ctr-c以後馬上再按一次ctr-\會怎樣?**


```
void g(int signum) {
    printf(“『ctr-\』\n”);
    sleep(10);
}
```

```
void f(int signum) {
    printf(“『ctr-c』\n”);
    sleep(10);
}
```

```
void main() {
    signal(ctr-c, f);
    signal(ctr-\, g);
    getchar();
}
```

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
    printf(“『ctr-\』\n”);
    sleep(10);
}
```

```
void f(int signum) {
    printf(“『ctr-c』\n”);
    sleep(10);
}
```

```
void main() {
    signal(ctr-c, f);
    signal(ctr-\, g);
    getchar();
}
```

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
    printf(“『ctr-\』\n”);
    sleep(10);
}
```

```
void f(int signum) {
    printf(“『ctr-c』\n”);
    sleep(10);
}
```

```
void main() {
    signal(ctr-c, f);
    signal(ctr-\, g);
    getchar();
}
```

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
    printf(“『ctr-\』\n”);
    sleep(10);
}
```

```
void f(int signum) {
    printf(“『ctr-c』\n”);
    sleep(10);
}
```

```
void main() {
    signal(ctr-c, f);
    signal(ctr-\, g);
    getchar();
}
```

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
    printf(“『ctr-\』\n”);
    sleep(10);
}
```

```
void f(int signum) {
    printf(“『ctr-c』\n”);
    sleep(10);
}
```

```
void main() {
    signal(ctr-c, f);
    signal(ctr-\, g);
    getchar();
}
```

在函數f()中，時間已經過了5秒鐘

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
    printf(“『ctr-\』\n”);
    sleep(10);
}
```

```
void f(int signum) {
    printf(“『ctr-c』\n”);
    sleep(10);
}
```

```
void main() {
    signal(ctr-c, f);
    signal(ctr-\, g);
    getchar();
}
```

在函數f()中，時間已經過了5秒鐘

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
    printf(“『ctr-\』\n”);
    sleep(10);
}
```

```
void f(int signum) {
    printf(“『ctr-c』\n”);
    sleep(10);
}
```

```
void main() {
    signal(ctr-c, f);
    signal(ctr-\, g);
    getchar();
}
```

在函數f()中，時間已經過了5秒鐘

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
    printf(“『ctr-\』\n”);
    sleep(10);
}
```

```
void f(int signum) {
    printf(“『ctr-c』\n”);
    sleep(10);
}
```

```
void main() {
    signal(ctr-c, f);
    signal(ctr-\, g);
    getchar();
}
```

在函數f()中，時間已經過了5秒鐘

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
    printf(“『ctr-\』\n”);
    sleep(10);
}
```

```
void f(int signum) {
    printf(“『ctr-c』\n”);
    sleep(10);
}
```

```
void main() {
    signal(ctr-c, f);
    signal(ctr-\, g);
    getchar();
}
```

在函數f()中，時間已經過了5秒鐘

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
    printf(“『ctr-\』\n”);
    sleep(10);
}
```

```
void f(int signum) {
    printf(“『ctr-c』\n”);
    sleep(10);
}
```

```
void main() {
    signal(ctr-c, f);
    signal(ctr-\, g);
    getchar();
}
```

在函數f()中，時間已經過了5秒鐘

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
    printf(“『ctr-\』\n”);
    sleep(10);
}
```

```
void f(int signum) {
    printf(“『ctr-c』\n”);
    sleep(10);
}
```

```
void main() {
    signal(ctr-c, f);
    signal(ctr-\, g);
    getchar();
}
```

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
    printf(“『ctr-\』\n”);
    sleep(10);
}
void f(int signum) {
    printf(“『ctr-c』\n”);
    sleep(10);
}
void main() {
    signal(ctr-c, f);
    signal(ctr-\, g);
    getchar();
}
```

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
    printf(“『ctr-\』\n”);
    sleep(10);
}
```

```
void f(int signum) {
    printf(“『ctr-c』\n”);
    sleep(10);
}
```

```
void main() {
    signal(ctr-c, f);
    signal(ctr-\, g);
    getchar();
}
```

之前
在函數f()中，時間
已經過了5秒鐘

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
    printf(“『ctr-\』\n”);
    sleep(10);
}
```

```
void f(int signum) {
    printf(“『ctr-c』\n”);
    sleep(10);
}
```

```
void main() {
    signal(ctr-c, f);
    signal(ctr-\, g);
    getchar();
}
```

之前
在函數f()中，時間
已經過了5秒鐘

sleep是system call
但他『不會』
『auto restart』

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
    printf(“『ctr-\』\n”);
    sleep(10);
}
```

```
void f(int signum) {
    printf(“『ctr-c』\n”);
    sleep(10);
}
```

```
void main() {
    signal(ctr-c, f);
    signal(ctr-\, g);
    getchar();
}
```

之前
在函數f()中，時間
已經過了5秒鐘

sleep是system call
但他『不會』
『auto restart』
直接回傳剩餘秒數

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS


```
void g(int signum) {
    printf(“『ctr-\』\n”);
    sleep(10);
}
```

```
void f(int signum) {
    printf(“『ctr-c』\n”);
    sleep(10);
}
```

```
void main() {
    signal(ctr-c, f);
    signal(ctr-\, g);
    getchar();
}
```

之前
在函數f()中，時間
已經過了5秒鐘

sleep是system call
但他『不會』
『auto restart』
直接回傳剩餘秒數
回傳『5』

APP

OS scheduler

event	Todo
ctr-c	f()
ctr-\	g()
kill	default

OS

Signal的行為與UNIX的版本

- 請在Linux上詳細的閱讀man page
  - man 7 signal
- 有些會自動restart
  - 例如：open、read、write、…
- 有些不會
  - "Input" socket interfaces, "Output" socket interfaces, Interfaces used to wait for signals, multiplexing interfaces, System V IPC interfaces, Sleep interfaces, io_getevents
  - 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.

小回顧

- 在Linux中，大部分的system call遇到signal，大部分會自動「重新執行」，少部分不會，不會的大部分回傳回「EINTR」
- 發生第 # 號signal，那麼作業系統會在第 # 號signal結束前，會自動封鎖 #
- 可以使用sigprocmask封鎖暫時不想處理的signal
  - 這些signal會變成「懸而未決」，等到解除封鎖後，會立即進入到應用程式
  - 更好的方式是用sigaction告訴OS，當處理第 # signal時，封鎖哪些signal
- sleep() 遇到signal是回傳「剩餘時間」

上課練習

```
void g(int signum) {
   start=now();
   sleep(10);
   printf(“time = %f”, now()-start);
}
```

```
void f(int signum) {
   start=now();
   sleep(10);
   printf(“time = %f”, now()-start);
}
```

```
void main() {
   signal(ctr-c, f);
   signal(ctr-\, g);
   start=now();
   sleep(10);
   printf(“time = %f”, now()-start);
   getchar();
}
```

工具箱

```
1.  #include <unistd.h>
2.  #include <assert.h>
3.  #include <signal.h>
4.  #include <stdio.h>
5.  #include <stdlib.h>
6.  #include <signal.h>
7.  #include <time.h>
8.      void printSigMask() {
9.          sigset_t oldsigset;
10.         sigprocmask(SIG_SETMASK, NULL, &oldsigset);
11.         for (int i=0; i<SIGRTMAX; i++)
12.             if (sigismember(&oldsigset, i) == 1)
13.                 printf("Signal ""%s"" is blocked\n", sys_siglist[i]);
14.     }
15.     double getCurTime() {
16.         struct timespec now;
17.         clock_gettime(CLOCK_MONOTONIC, &now);
18.         double sec = now.tv_sec;
19.         double nano_sec = now.tv_nsec;
20.         return sec+nano_sec*10E-9;
21.     }
```


setjmp & longjmp

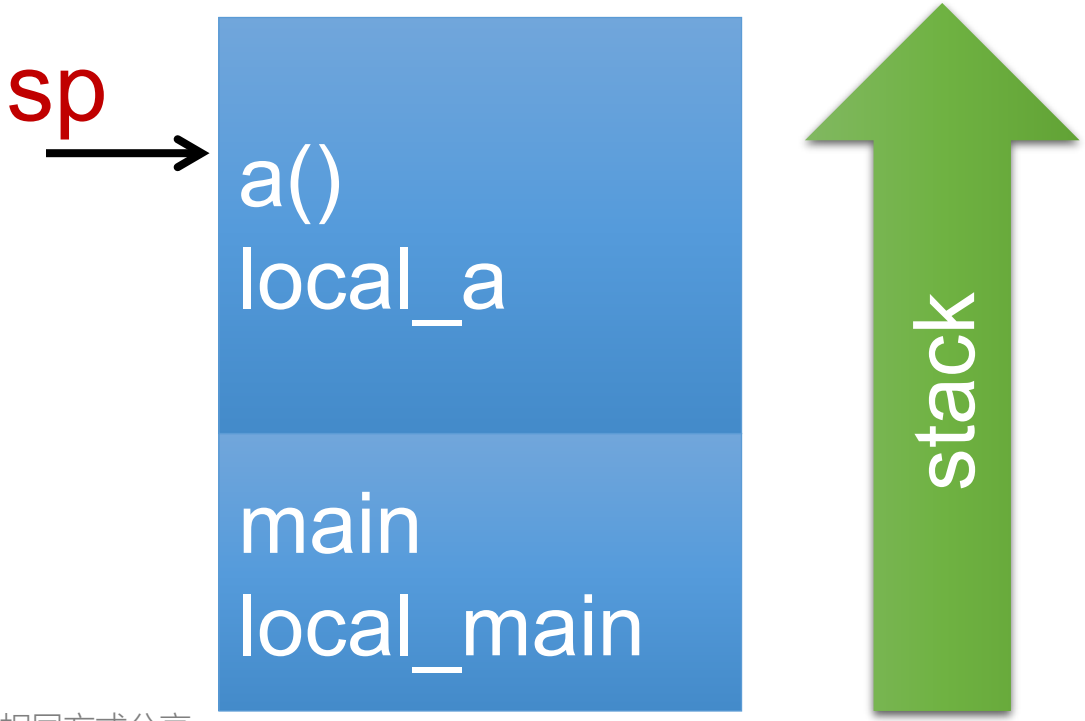
預備知識： setjmp

```
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);
}
```

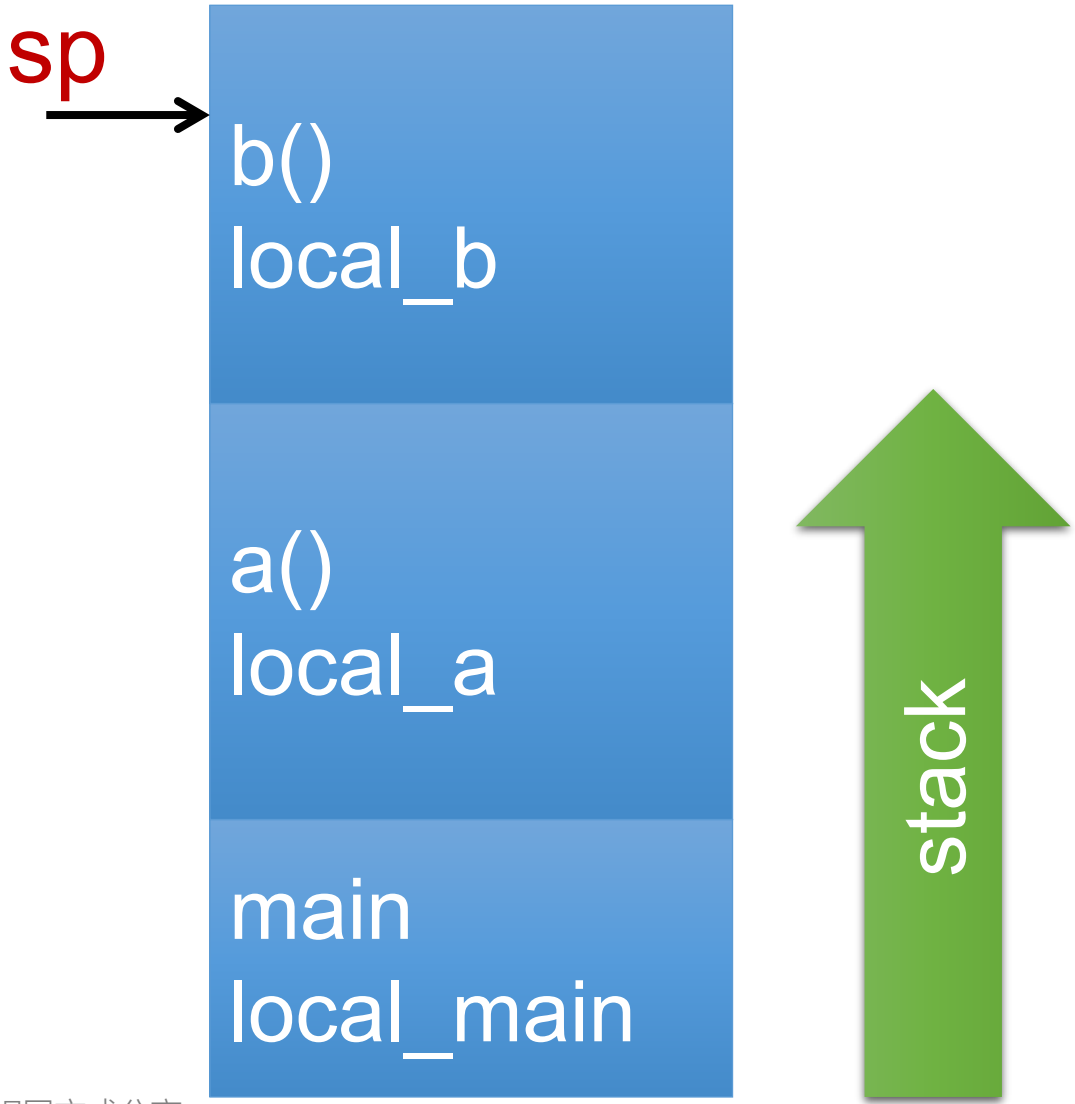
pc



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);
}
```

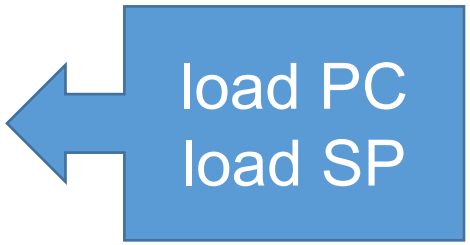
pc

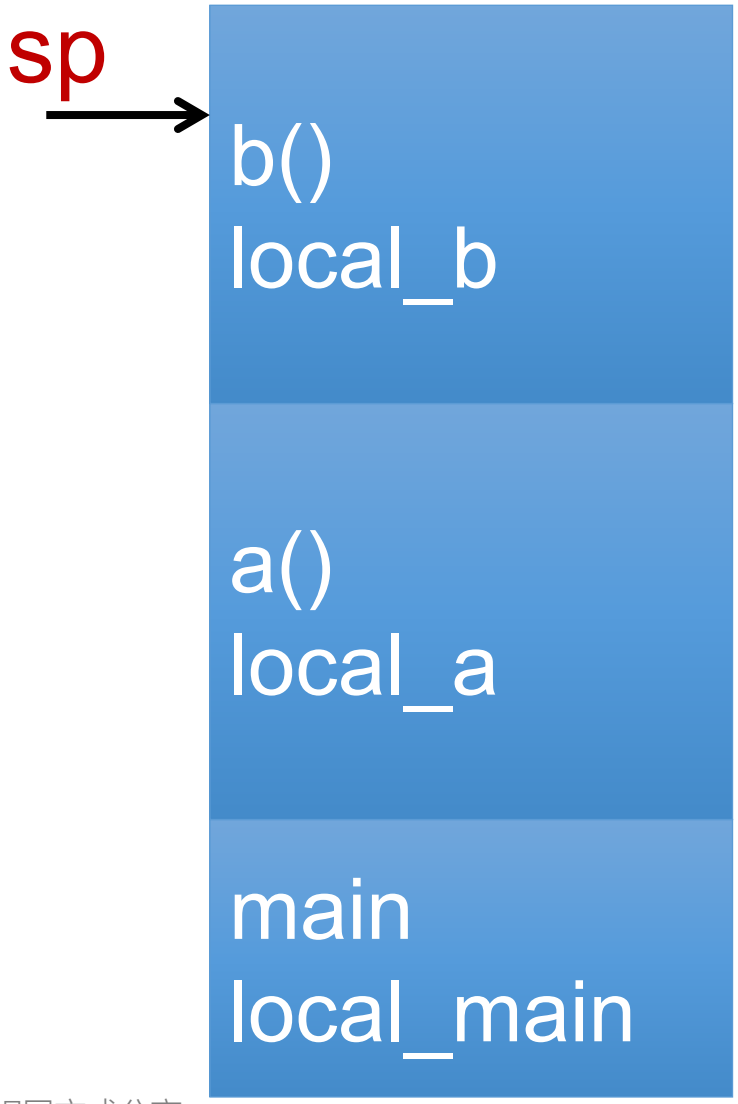


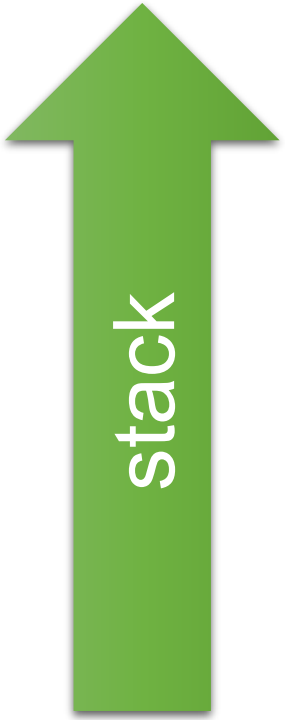
setjmp

```
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



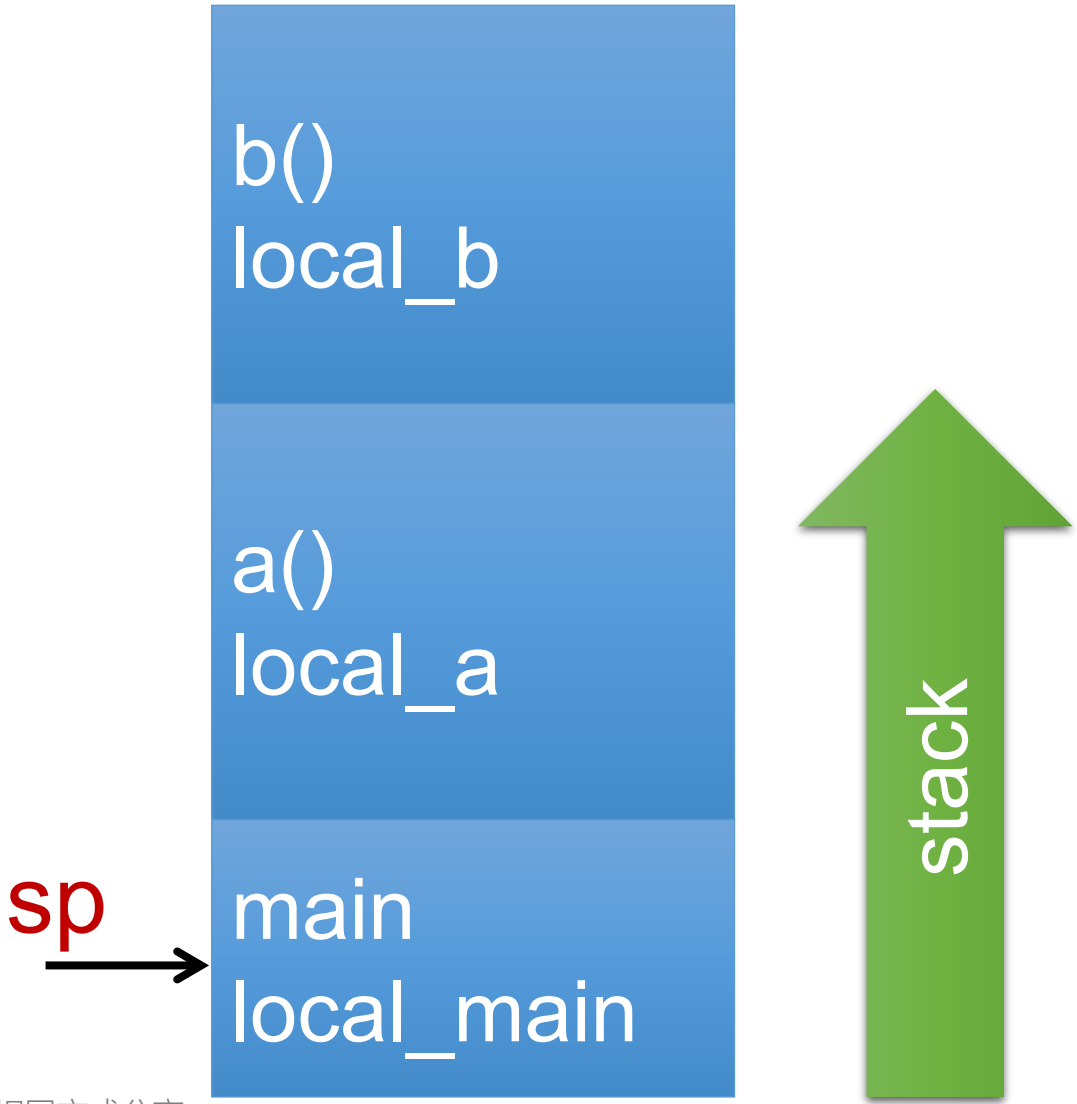




longjmp

```
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

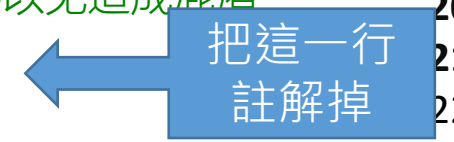
```
stat of a
stat of b
return form longjmp. the
return value is 5
p1 = 1, p2 = 2, p3 = 33
//有些編譯器 p1會等於 11
//只有宣告為nonvolatile的
變數的值是確定更新的
```

sig_setjmp & sig_longjmp

除了儲存PC、SP以外
還儲存signal的狀態（是否被mask）

課堂練習

```
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. }
```

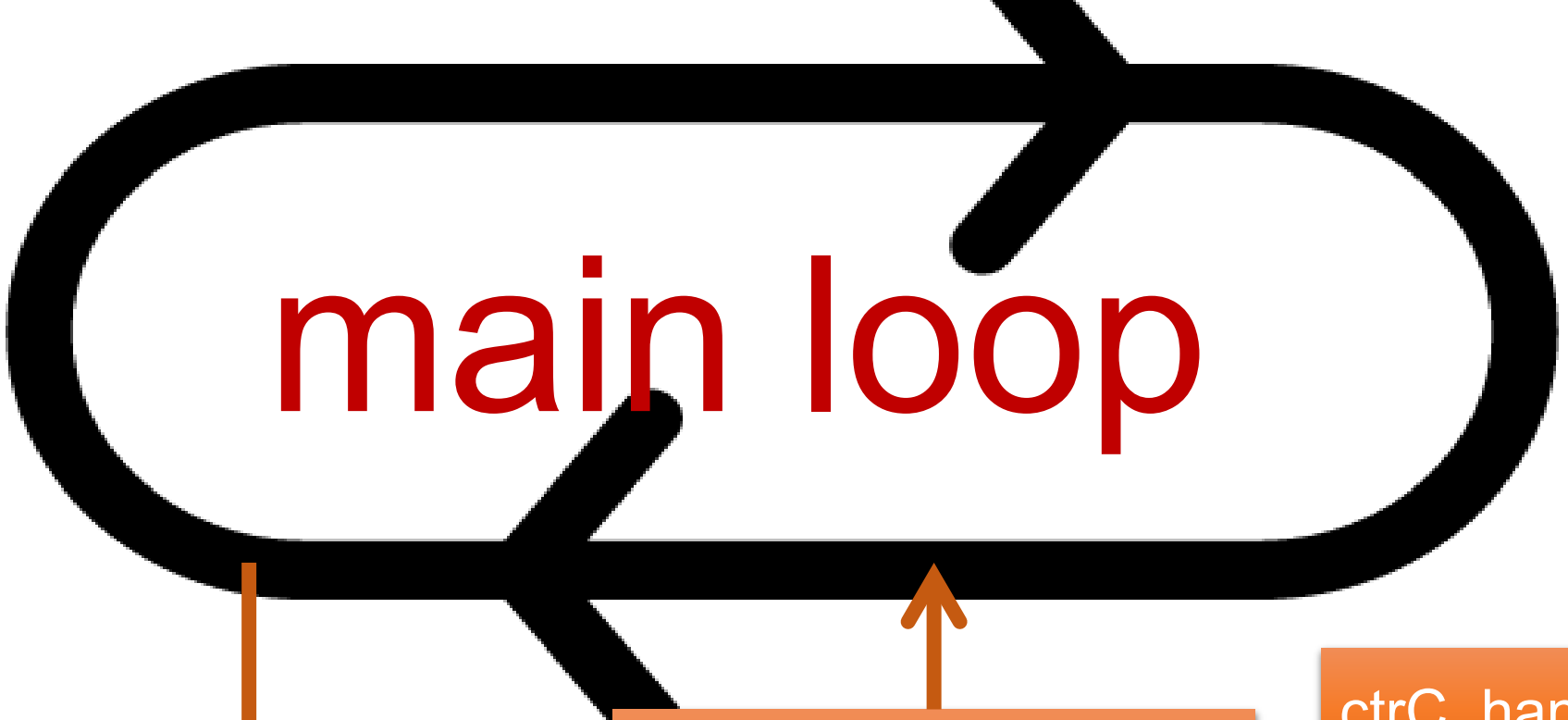



```
14. int main(int argc, char** argv) {
15.     int ret;
16.     register int p1=11;
17.     volatile int p2=22;
18.     int p3=33;
19.     p1=1;
20.     p2=2;
21.     p3=3;
22.     //回傳值0有特別用途，代表setjmp成功
23.     if ((ret=setjmp(buf)) == 0)
24.         a();
25.     else {
26.         printf("return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

完整的練習： shell

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



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)
```

```
e)     } else {
f)       ungets_ctr_c();
g)       siglongjmp(jmpbuf 1);
h) }
```

```
1. void main() {
2.     signal(ctr-c, ctr_c);
3.     while(1) {
4.             sigsetjmp(jmp_buf,1);
5.             fgets(cmdline, 256, stdin);
6.       parseString(cmdLine, &exeName);
7.       if (cmdline == “^c”) continue;
8.       if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.             childPid = pid;
10.            hasChild = 1;
11.            wait();//等待child執行結束
12.      } else
13.            execvp(exeName, argVect);     //child，執行命令
14.    }
15.}
```

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

```
e)     } else {
f)       ungets_ctr_c();
g)       siglongjmp(jmpbuf 1);
h) }
```


```
1. void main() {
2.     signal(ctr-c, ctr_c);
3.     while(1) {
4.               sigsetjmp(jmp_buf,1);
5.               fgets(cmdline, 256, stdin);
6.       parseString(cmdLine, &exeName);
7.       if (cmdline == “^c”) continue;
8.       if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.               childPid = pid;
10.              hasChild = 1;
11.              wait();//等待child執行結束
12.      } else
13.              execvp(exeName, argVect);     //child，執行命令
14.    }
15.}
```

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

```
e)     } else {
f)       ungets_ctr_c();
g)       siglongjmp(jmpbuf 1);
h) }
```


```
1. void main() {
2.     signal(ctr-c, ctr_c);
3.     while(1) {
4.               sigsetjmp(jmp_buf,1);
5.               fgets(cmdline, 256, stdin);
6.       parseString(cmdLine, &exeName);
7.       if (cmdline == “^c”) continue;
8.       if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.               childPid = pid;
10.              hasChild = 1;
11.              wait();//等待child執行結束
12.      } else
13.              execvp(exeName, argVect);     //child，執行命令
14.    }
15.}
```

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

```
e)     } else {
f)       ungets_ctr_c();
g)       siglongjmp(jmpbuf 1);
h) }
```


```
1. void main() {
2.     signal(ctr-c, ctr_c);
3.     while(1) {
4.             sigsetjmp(jmp_buf,1);
5.             fgets(cmdline, 256, stdin);
6.       parseString(cmdLine, &exeName);
7.       if (cmdline == “^c”) continue;
8.       if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.             childPid = pid;
10.            hasChild = 1;
11.            wait();//等待child執行結束
12.      } else
13.            execvp(exeName, argVect);     //child，執行命令
14.    }
15.}
```

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

```
e)     } else {
f)       ungets_ctr_c();
g)       siglongjmp(jmpbuf 1);
h) }
```

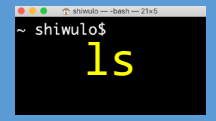

```
1. void main() {
2.     signal(ctr-c, ctr_c);
3.     while(1) {
4.             sigsetjmp(jmp_buf,1);
5.             fgets(cmdline, 256, stdin);
6.       parseString(cmdLine, &exeName);
7.       if (cmdline == “^c”) continue;
8.       if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.             childPid = pid;
10.            hasChild = 1;
11.            wait();//等待child執行結束
12.      } else
13.            execvp(exeName, argVect);     //child，執行命令
14.    }
15.}
```

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

```
e)     } else {
f)       ungets_ctr_c();
g)       siglongjmp(jmpbuf 1);
h) }
```

```
1. void main() {
2.     signal(ctr-c, ctr_c);
3.     while(1) {
4.             sigsetjmp(jmp_buf,1);
5.             fgets(cmdline, 256, stdin);
6.       parseString(cmdLine, &exeName);
7.       if (cmdline == “^c”) continue;
8.       if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.             childPid = pid;
10.            hasChild = 1;
11.            wait();//等待child執行結束
12.      } else
13.            execvp(exeName, argVect);     //child，執行命令
14.    }
15.}
```

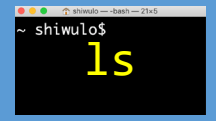


OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

```
e)     } else {
f)       ungets_ctr_c();
g)       siglongjmp(jmpbuf 1);
h) }
```

```
1. void main() {
2.     signal(ctr-c, ctr_c);
3.     while(1) {
4.             sigsetjmp(jmp_buf,1);
5.             fgets(cmdline, 256, stdin);
6.       parseString(cmdLine, &exeName);
7.       if (cmdline == “^c”) continue;
8.       if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.             childPid = pid;
10.            hasChild = 1;
11.            wait();//等待child執行結束
12.      } else
13.            execvp(exeName, argVect);     //child，執行命令
14.    }
15.}
```

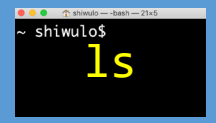


OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

```
e)     } else {
f)       ungets_ctr_c();
g)       siglongjmp(jmpbuf 1);
h) }
```

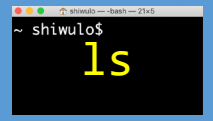

```
1. void main() {
2.     signal(ctr-c, ctr_c);
3.     while(1) {
4.             sigsetjmp(jmp_buf,1);
5.             fgets(cmdline, 256, stdin);
6.       parseString(cmdLine, &exeName);
7.       if (cmdline == “^c”) continue;
8.       if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.             childPid = pid;
10.            hasChild = 1;
11.            wait();//等待child執行結束
12.      } else
13.            execvp(exeName, argVect);     //child，執行命令
14.    }
15.}
```



OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

```
e)     } else {
f)       ungets_ctr_c();
g)       siglongjmp(jmpbuf 1);
h) }
```

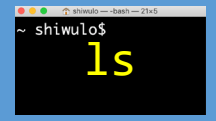



```
1. void main() {
2.     signal(ctr-c, ctr_c);
3.     while(1) {
4.             sigsetjmp(jmp_buf,1);
5.             fgets(cmdline, 256, stdin);
6.       parseString(cmdLine, &exeName);
7.       if (cmdline == “^c”) continue;
8.       if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.             childPid = pid;
10.            hasChild = 1;
11.            wait();//等待child執行結束
12.      } else
13.            execvp(exeName, argVect);     //child，執行命令
14.    }
15.}
```

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

```
e)     } else {
f)       ungets_ctr_c();
g)       siglongjmp(jmpbuf 1);
h) }
```



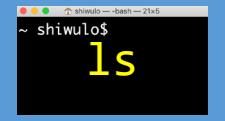

```
1. void main() {
2.     signal(ctr-c, ctr_c);
3.     while(1) {
4.             sigsetjmp(jmp_buf,1);
5.             fgets(cmdline, 256, stdin);
6.       parseString(cmdLine, &exeName);
7.       if (cmdline == “^c”) continue;
8.       if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.             childPid = pid;
10.            hasChild = 1;
11.            wait();//等待child執行結束
12.      } else
13.            execvp(exeName, argVect);     //child，執行命令
14.    }
15.}
```

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

```
e)     } else {
f)       ungets_ctr_c();
g)       siglongjmp(jmpbuf 1);
h) }
```

```
1. void main() {
2.     signal(ctr-c, ctr_c);
3.     while(1) {
4.             sigsetjmp(jmp_buf,1);
5.             fgets(cmdline, 256, stdin);
6.       parseString(cmdLine, &exeName);
7.       if (cmdline == “^c”) continue;
8.       if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.             childPid = pid;
10.            hasChild = 1;
11.            wait();//等待child執行結束
12.      } else
13.            execvp(exeName, argVect);     //child，執行命令
14.    }
15.}
```

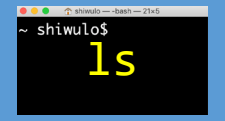


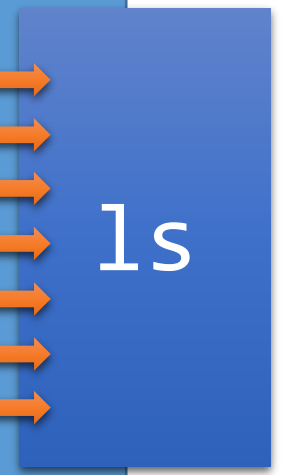
OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

```
e)     } else {
f)       ungets_ctr_c();
g)       siglongjmp(jmpbuf 1);
h) }
```

```
1. void main() {
2.     signal(ctr-c, ctr_c);
3.     while(1) {
4.             sigsetjmp(jmp_buf,1);
5.             fgets(cmdline, 256, stdin);
6.       parseString(cmdLine, &exeName);
7.       if (cmdline == “^c”) continue;
8.       if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.             childPid = pid;
10.            hasChild = 1;
11.            wait();//等待child執行結束
12.      } else
13.            execvp(exeName, argVect);     //child，執行命令
14.    }
15.}
```



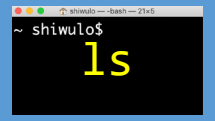


OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

```
e)     } else {
f)       ungets_ctr_c();
g)       siglongjmp(jmpbuf 1);
h) }
```

```
1. void main() {
2.     signal(ctr-c, ctr_c);
3.     while(1) {
4.             sigsetjmp(jmp_buf,1);
5.             fgets(cmdline, 256, stdin);
6.       parseString(cmdLine, &exeName);
7.       if (cmdline == “^c”) continue;
8.       if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.             childPid = pid;
10.            hasChild = 1;
11.            wait();//等待child執行結束
12.      } else
13.            execvp(exeName, argVect);     //child，執行命令
14.    }
15.}
```

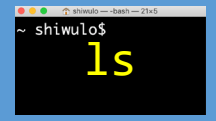


ls

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

```
e)     } else {
f)       ungets_ctr_c();
g)       siglongjmp(jmpbuf 1);
h) }
```




```
1. void main() {
2.     signal(ctr-c, ctr_c);
3.     while(1) {
4.             sigsetjmp(jmp_buf,1);
5.             fgets(cmdline, 256, stdin);
6.       parseString(cmdLine, &exeName);
7.       if (cmdline == “^c”) continue;
8.       if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.             childPid = pid;
10.            hasChild = 1;
11.            wait();//等待child執行結束
12.      } else
13.            execvp(exeName, argVect);     //child，執行命令
14.    }
15.}
```

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default


```
a) void f(int signum) {
b)    if (hasChild) {
c)      kill(childPid, SIGINT);
d)      hasChild = 0;
e)    } else {
f)      ungets_ctr_c();
g)      siglongjmp(jmpbuf 1);
h) }
```

```
8.      if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.              childPid = pid;
10.             hasChild = 1;
11.             wait();//等待child執行結束
12.     } else
13.             execvp(exeName, argVect);    //child，執行命令
14.   }
15.}
```

ls

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default


```
a) void f(int signum) {
b)    if (hasChild) {
c)      kill(childPid, SIGINT);
d)      hasChild = 0;
e)    } else {
f)      ungets_ctr_c();
g)      siglongjmp(jmpbuf 1);
h) }
```

```
8.      if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.              childPid = pid;
10.             hasChild = 1;
11.             wait();//等待child執行結束
12.     } else
13.             execvp(exeName, argVect);    //child，執行命令
14.   }
15.}
```

ls

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default


```
a) void f(int signum) {
b)    if (hasChild) {
c)      kill(childPid, SIGINT);
d)      hasChild = 0;
e)    } else {
f)      ungets_ctr_c();
g)      siglongjmp(jmpbuf 1);
h) }
```

```
8.      if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.              childPid = pid;
10.             hasChild = 1;
11.             wait();//等待child執行結束
12.     } else
13.             execvp(exeName, argVect);    //child，執行命令
14.   }
15.}
```

ls

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default


```
a) void f(int signum) {
b)    if (hasChild) {
c)      kill(childPid, SIGINT);
d)      hasChild = 0;
e)    } else {
f)      ungets_ctr_c();
g)      siglongjmp(jmpbuf 1);
h) }
```

```
8.      if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.              childPid = pid;
10.             hasChild = 1;
11.             wait();//等待child執行結束
12.     } else
13.             execvp(exeName, argVect);    //child，執行命令
14.   }
15.}
```

ls

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default


```
a) void f(int signum) {
b)    if (hasChild) {
c)      kill(childPid, SIGINT);
d)      hasChild = 0;
e)    } else {
f)      ungets_ctr_c();
g)      siglongjmp(jmpbuf 1);
h) }
```

```
8.      if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.              childPid = pid;
10.             hasChild = 1;
11.             wait();//等待child執行結束
12.     } else
13.             execvp(exeName, argVect);    //child，執行命令
14.   }
15.}
```

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default


```
a) void f(int signum) {
b)    if (hasChild) {
c)      kill(childPid, SIGINT);
d)      hasChild = 0;
e)    } else {
f)      ungets_ctr_c();
g)      siglongjmp(jmpbuf 1);
h) }
```

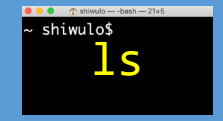
```
8.      if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.              childPid = pid;
10.             hasChild = 1;
11.             wait();//等待child執行結束
12.     } else
13.             execvp(exeName, argVect);    //child，執行命令
14.   }
15.}
```

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

```
e)     } else {
f)       ungets_ctr_c();
g)       siglongjmp(jmpbuf 1);
h) }
```

```
1. void main() {
2.     signal(ctr-c, ctr_c);
3.     while(1) {
4.             sigsetjmp(jmp_buf,1);
5.             fgets(cmdline, 256, stdin);
6.       parseString(cmdLine, &exeName);
7.       if (cmdline == “^c”) continue;
8.       if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.             childPid = pid;
10.            hasChild = 1;
11.            wait();//等待child執行結束
12.      } else
13.            execvp(exeName, argVect);     //child，執行命令
14.    }
15.}
```



OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

```
e)     } else {
f)       ungets_ctr_c();
g)       siglongjmp(jmpbuf 1);
h) }
```

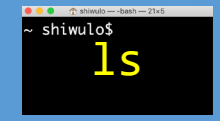
```
1. void main() {
2.     signal(ctr-c, ctr_c);
3.     while(1) {
4.             sigsetjmp(jmp_buf,1);
5.             fgets(cmdline, 256, stdin);
6.       parseString(cmdLine, &exeName);
7.       if (cmdline == “^c”) continue;
8.       if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.             childPid = pid;
10.            hasChild = 1;
11.            wait();//等待child執行結束
12.      } else
13.            execvp(exeName, argVect);     //child，執行命令
14.    }
15.}
```

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

```
e)     } else {
f)       ungets_ctr_c();
g)       siglongjmp(jmpbuf 1);
h) }
```

```
1. void main() {
2.     signal(ctr-c, ctr_c);
3.     while(1) {
4.             sigsetjmp(jmp_buf,1);
5.             fgets(cmdline, 256, stdin);
6.       parseString(cmdLine, &exeName);
7.       if (cmdline == “^c”) continue;
8.       if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.             childPid = pid;
10.            hasChild = 1;
11.            wait();//等待child執行結束
12.      } else
13.            execvp(exeName, argVect);     //child，執行命令
14.    }
15.}
```



OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

```
e)     } else {
f)       ungets_ctr_c();
g)       siglongjmp(jmpbuf 1);
h) }
```

```
1. void main() {
2.     signal(ctr-c, ctr_c);
3.     while(1) {
4.             sigsetjmp(jmp_buf,1);
5.             fgets(cmdline, 256, stdin);
6.       parseString(cmdLine, &exeName);
7.       if (cmdline == “^c”) continue;
8.       if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.             childPid = pid;
10.            hasChild = 1;
11.            wait();//等待child執行結束
12.      } else
13.            execvp(exeName, argVect);     //child，執行命令
14.    }
15.}
```

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

```
e)     } else {
f)       ungets_ctr_c();
g)       siglongjmp(jmpbuf 1);
h) }
```


```
1. void main() {
2.     signal(ctr-c, ctr_c);
3.     while(1) {
4.             sigsetjmp(jmp_buf,1);
5.             fgets(cmdline, 256, stdin);
6.       parseString(cmdLine, &exeName);
7.       if (cmdline == “^c”) continue;
8.       if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.             childPid = pid;
10.            hasChild = 1;
11.            wait();//等待child執行結束
12.      } else
13.            execvp(exeName, argVect);     //child，執行命令
14.    }
15.}
```

event	Todo
ctr-c	f()
ctr-\	default
kill	default


```
a) void f(int signum) {
b)    if (hasChild) {
c)      kill(childPid, SIGINT);
d)      hasChild = 0;
e)    } else {
f)      ungets_ctr_c();
g)      siglongjmp(jmpbuf 1);
h) }
```

```
8.      if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.              childPid = pid;
10.             hasChild = 1;
11.             wait();//等待child執行結束
12.     } else
13.             execvp(exeName, argVect);    //child，執行命令
14.   }
15.}
```

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default


```
a) void f(int signum) {
b)    if (hasChild) {
c)      kill(childPid, SIGINT);
d)      hasChild = 0;
e)    } else {
f)      ungets_ctr_c();
g)      siglongjmp(jmpbuf 1);
h) }
```

```
8.      if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.              childPid = pid;
10.             hasChild = 1;
11.             wait();//等待child執行結束
12.     } else
13.             execvp(exeName, argVect);    //child，執行命令
14.   }
15.}
```

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default


```
a) void f(int signum) {
b)    if (hasChild) {
c)      kill(childPid, SIGINT);
d)      hasChild = 0;
e)    } else {
f)      ungets_ctr_c();
g)      siglongjmp(jmpbuf 1);
h) }
```

```
8.      if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.              childPid = pid;
10.             hasChild = 1;
11.             wait();//等待child執行結束
12.     } else
13.             execvp(exeName, argVect);    //child，執行命令
14.   }
15.}
```

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default


```
a) void f(int signum) {
b)    if (hasChild) {
c)      kill(childPid, SIGINT);
d)      hasChild = 0;
e)    } else {
f)      ungets_ctr_c();
g)      siglongjmp(jmpbuf 1);
h) }
```

```
8.      if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.              childPid = pid;
10.             hasChild = 1;
11.             wait();//等待child執行結束
12.     } else
13.             execvp(exeName, argVect);    //child，執行命令
14.   }
15.}
```

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default


```
a) void f(int signum) {
b)    if (hasChild) {
c)      kill(childPid, SIGINT);
d)      hasChild = 0;
e)    } else {
f)      ungets_ctr_c();
g)      siglongjmp(jmpbuf 1);
h) }
```

```
8.      if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.              childPid = pid;
10.             hasChild = 1;
11.             wait();//等待child執行結束
12.     } else
13.             execvp(exeName, argVect);    //child，執行命令
14.   }
15.}
```

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

```
e)     } else {
f)       ungets_ctr_c();
g)       siglongjmp(jmpbuf 1);
h) }
```


```
1. void main() {
2.     signal(ctr-c, ctr_c);
3.     while(1) {
4.             sigsetjmp(jmp_buf,1);
5.             fgets(cmdline, 256, stdin);
6.       parseString(cmdLine, &exeName);
7.       if (cmdline == “^c”) continue;
8.       if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.             childPid = pid;
10.            hasChild = 1;
11.            wait();//等待child執行結束
12.      } else
13.            execvp(exeName, argVect);     //child，執行命令
14.    }
15.}
```

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

```
e)     } else {
f)       ungets_ctr_c();
g)       siglongjmp(jmpbuf 1);
h) }
```


```
1. void main() {
2.     signal(ctr-c, ctr_c);
3.     while(1) {
4.             sigsetjmp(jmp_buf,1);
5.             fgets(cmdline, 256, stdin);
6.       parseString(cmdLine, &exeName);
7.       if (cmdline == “^c”) continue;
8.       if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.             childPid = pid;
10.            hasChild = 1;
11.            wait();//等待child執行結束
12.      } else
13.            execvp(exeName, argVect);     //child，執行命令
14.    }
15.}
```

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

```
e)     } else {
f)       ungets_ctr_c();
g)       siglongjmp(jmpbuf 1);
h) }
```


```
1. void main() {
2.     signal(ctr-c, ctr_c);
3.     while(1) {
4.             sigsetjmp(jmp_buf,1);
5.             fgets(cmdline, 256, stdin);
6.       parseString(cmdLine, &exeName);
7.       if (cmdline == “^c”) continue;
8.       if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.             childPid = pid;
10.            hasChild = 1;
11.            wait();//等待child執行結束
12.      } else
13.            execvp(exeName, argVect);     //child，執行命令
14.    }
15.}
```

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

```
e)     } else {
f)       ungets_ctr_c();
g)       siglongjmp(jmpbuf 1);
h) }
```


```
1. void main() {
2.     signal(ctr-c, ctr_c);
3.     while(1) {
4.             sigsetjmp(jmp_buf,1);
5.             fgets(cmdline, 256, stdin);
6.       parseString(cmdLine, &exeName);
7.       if (cmdline == “^c”) continue;
8.       if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.             childPid = pid;
10.            hasChild = 1;
11.            wait();//等待child執行結束
12.      } else
13.            execvp(exeName, argVect);     //child，執行命令
14.    }
15.}
```

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

```
e)     } else {
f)       ungets_ctr_c();
g)       siglongjmp(jmpbuf 1);
h) }
```


```
1. void main() {
2.     signal(ctr-c, ctr_c);
3.     while(1) {
4.             sigsetjmp(jmp_buf,1);
5.             fgets(cmdline, 256, stdin);
6.       parseString(cmdLine, &exeName);
7.       if (cmdline == “^c”) continue;
8.       if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.             childPid = pid;
10.            hasChild = 1;
11.            wait();//等待child執行結束
12.      } else
13.            execvp(exeName, argVect);     //child，執行命令
14.    }
15.}
```

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

```
e)     } else {
f)       ungets_ctr_c();
g)       siglongjmp(jmpbuf 1);
h) }
```


```
1. void main() {
2.     signal(ctr-c, ctr_c);
3.     while(1) {
4.             sigsetjmp(jmp_buf,1);
5.             fgets(cmdline, 256, stdin);
6.       parseString(cmdLine, &exeName);
7.       if (cmdline == “^c”) continue;
8.       if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.             childPid = pid;
10.            hasChild = 1;
11.            wait();//等待child執行結束
12.      } else
13.            execvp(exeName, argVect);     //child，執行命令
14.    }
15.}
```

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

```
e)     } else {
f)       ungets_ctr_c();
g)       siglongjmp(jmpbuf 1);
h) }
```

```
1. void main() {
2.     signal(ctr-c, ctr_c);
3.     while(1) {
4.             sigsetjmp(jmp_buf,1);
5.             fgets(cmdline, 256, stdin);
6.       parseString(cmdLine, &exeName);
7.       if (cmdline == “^c”) continue;
8.       if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.             childPid = pid;
10.            hasChild = 1;
11.            wait();//等待child執行結束
12.      } else
13.            execvp(exeName, argVect);     //child，執行命令
14.    }
15.}
```

OS scheduler

event	Todo
ctr-c	f()
ctr-\	default
kill	default

```
a) void f(int signum) {
b)    if (hasChild) {
c)      kill(childPid, SIGINT);
d)      hasChild = 0;
e)    } else {
f)      ungets_ctr_c();
g)      siglongjmp(jmpbuf 1);
h) }
```

```
1. void main() {
2.    signal(ctr-c, ctr_c);
3.    while(1) {
4.            sigsetjmp(jmp_buf,1);
5.            fgets(cmdline, 256, stdin);
6.     parseString(cmdLine, &exeName);
7.     if (cmdline == “^c”) continue;
8.     if ((pid=fork()) > 0) {  //parent，也就是shell的部分
9.            childPid = pid;
10.           hasChild = 1;
11.           wait();//等待child執行結束
12.    } else
13.           execvp(exeName, argVect);    //child，執行命令
14.   }
15.}
```

工具箱

```
1.  #include <unistd.h>
2.  #include <sys/types.h>
3.  #include <sys/wait.h>
4.  #include <string.h>
5.  #include <stdio.h>
6.  #include <stdlib.h>
7.  #include <errno.h>
8.  #include <signal.h>
9.  #include <time.h>
10. #include <setjmp.h>
11. #include <sys/resource.h>
12. void parseString(char* str, char** cmd) {
13.     int idx=0;
14.     char* retPtr;
15.     retPtr=strtok(str, " \n");
16.     while(retPtr != NULL) {
17.         argVect[idx++] = retPtr;
18.         if (idx==1)
19.             *cmd = retPtr;
20.         retPtr=strtok(NULL, " \n");
21.     }
22.     argVect[idx]=NULL;
23. }
```

工具箱

```
1.  void ungets_ctr_c() {
2.      ungetc('\n', stdin);
3.      ungetc('c', stdin);
4.      ungetc('^', stdin);
5.  }
```

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. }
```

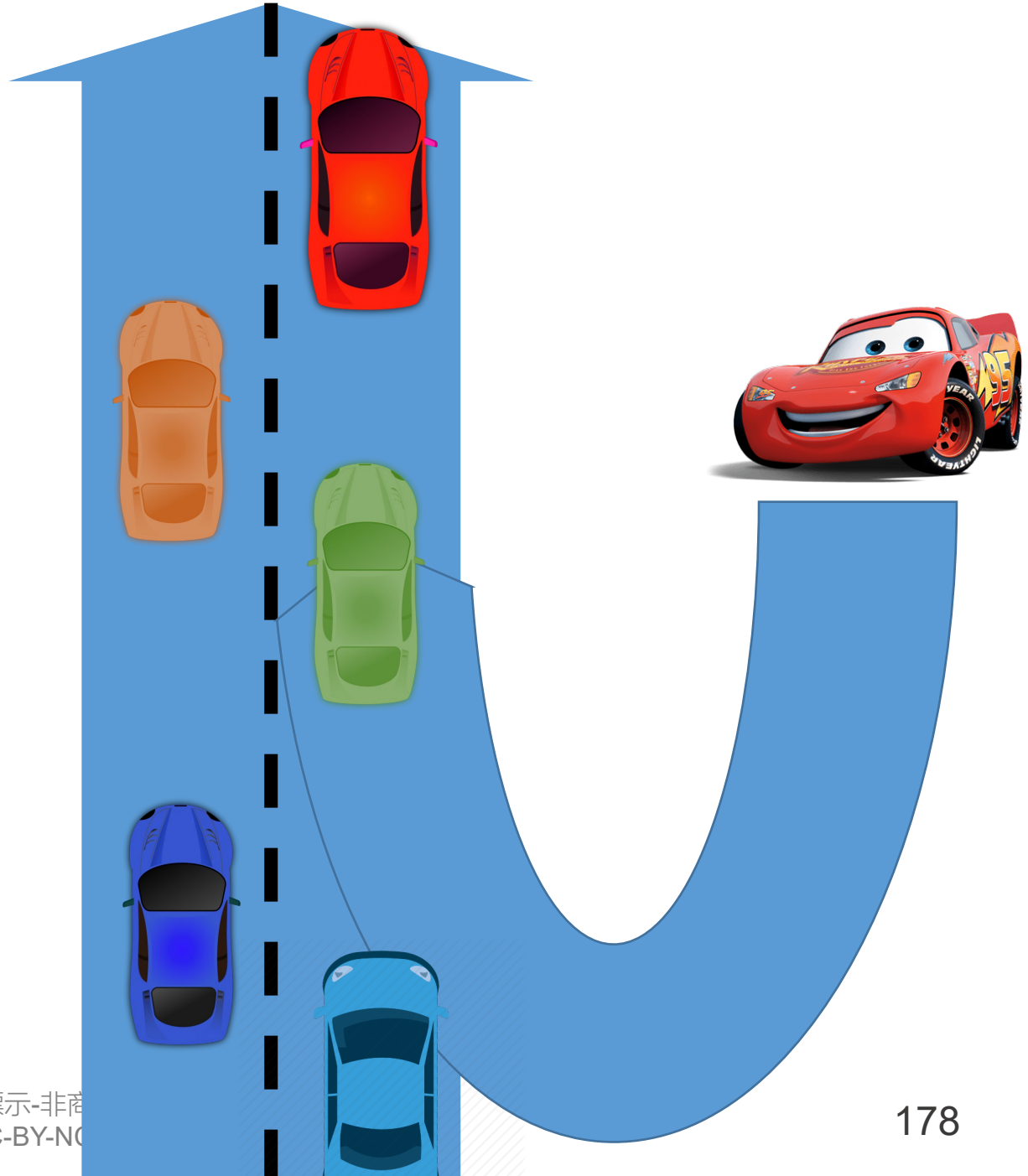
```
1.  int main (int argc, char** argv) {
2.      signal(SIGINT, ctrC_handler);          /*程式碼註冊ctr-c signal的處理方式*/
3.      signal(SIGQUIT, SIG_IGN);  /*程式碼註冊ctr-\ signal的處理方式*/
4.      signal(SIGTSTP, SIG_IGN);  /*程式碼註冊ctr-z signal的處理方式*/
5.      while(1) {
6.          hasChild = 0;//設定化hasChild, argVect[0]，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>>
```

將signal同步化

併入主迴圈處理



併入主回圈的常見方法

- 在主回圈中，定期等待signal（通常用於「遇到signal才處理」）
- 設定flag，然後在主回圈中檢查
- 使用setjmp + longjmp跳回主回圈（注意同步性）
- 使用pipe + select + read，於主回圈每次提取新的「輸入」時同時檢查「signal」
- 使用signalfd（較新的方法，Linux特有）

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
```

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myshell.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