



Process control

中正大學，作業系統實驗室

羅習五 陽春副教授

shiwulo@gmail.com



大綱

🍏 programs & processes

🍏 fork() fork 出來會跟 parent 一模一樣

🍏 wait()

🍏 zombie process

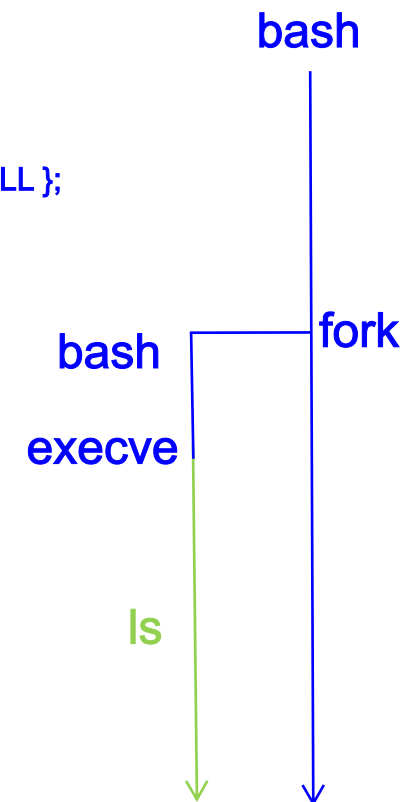
🍏 execve() functions 將新的執行檔載入到這個 process 裡，繼續執行

🍏 vfork() & execv()

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
```

```
int main() {
    pid_t pid;
    char *const parmList[] = { "/usr/bin/ls", "-lash", NULL };
    char *const envParms[] = { "STEPLIB=SASC.V6.LINKLIB", NULL };

    if ( (pid = fork()) == -1 ) perror("fork error");
    else if (pid == 0) {
        execve(parmList[0], parmList, envParms);
    }
    return 0;
}
```



```
int execve(const char *pathname, char *const argv[], char *const envp[]);
```

將新的執行檔載入到這個 process 裡，繼續執行

Task的屬性繼承

🍏 繼承分為『顯性的』與『隱性的』

🍀 顯性

🍀 隱性

🍏 fork下的繼承

🍏 execv下的繼承



Program & process

proc/pid

這些是 pseudo file, 由 Linux 動態產生

```
$ls /proc/
```

```
dr-xr-xr-x  9 root          root          0 May 16 15:09 1
dr-xr-xr-x  9 root          root          0 May 16 15:09 10
dr-xr-xr-x  9 gdm           gdm           0 May 16 15:09 1005
dr-xr-xr-x  9 root          root          0 May 16 15:09 1010
...
```

proc/pid

```
$cd /proc/1
```

```
$sudo ls -alh
```

```
-r--r--r--    1 root root 0 May 16 15:09 cmdline
-rw-r--r--    1 root root 0 May 16 15:09 comm
-rw-r--r--    1 root root 0 May 16 15:10 coredump_filter
-r--r--r--    1 root root 0 May 16 15:10 cpuset
lrwxrwxrwx    1 root root 0 May 16 15:10 cwd -> /
-r-----    1 root root 0 May 16 15:09 environ
lrwxrwxrwx    1 root root 0 May 16 15:09 exe -> /lib/systemd/systemd init OS
```

/proc

```
$less /proc/cpuinfo
```

```
processor      : 0  
vendor_id     : GenuineIntel  
cpu family    : 6  
model         : 94  
model name    : Intel(R) Core(TM) i7-6770HQ CPU @ 2.60GHz  
stepping      : 3  
microcode     : 0x8a  
cpu MHz       : 934.251  
cache size    : 6144 KB
```

/proc/irq

外部中斷

中斷有兩種設定方法：

1. 讓中斷平均分散在 CPU 上
2. 指定 CPU core 處理特定的中斷 (ubuntu default)

2. 的好處：TLB 命中機率大

2. 的壞處：如果中斷真的太頻繁，應該要多分配點 CPU

```
/proc$ less interrupts
```

	CPU0	CPU1	CPU2	CPU3	CPU4	CPU5	CPU6	CPU7			
0:	39	0	0	0	0	0	0	0	IR-IO-APIC	2-edge	timer
3:	0	0	0	0	0	0	0	0	IR-IO-APIC	3-edge	nuvoton-cir
8:	0	0	0	1	0	0	0	0	IR-IO-APIC	8-edge	rtc0
9:	0	5	0	0	0	0	0	0	IR-IO-APIC	9-fasteoi	acpi
17:	0	0	0	0	0	0	0	0	IR-IO-APIC	17-fasteoi	mmc0
120:	0	0	0	0	0	0	0	0	DMAR-MSI	0-edge	dmars0
121:	0	0	0	0	0	0	0	0	DMAR-MSI	1-edge	dmars1
122:	0	0	653	0	0	2698	0	0	IR-PCI-MSI	327680-edge	xhci_hcd
123:	0	0	0	0	184113	0	0	0	IR-PCI-MSI	376832-edge	ahci[0000:00:17.0]
124:	0	0	0	0	0	43161	427	0	IR-PCI-MSI	520192-edge	eno1

fork



fork()

🍏 #include <unistd.h>

🍏 pid_t fork(void);

🍏 POSIX 作業系統的標準中，新的程序均需藉由呼叫fork()函數而產生

♣ 父程序(parent process): 呼叫fork()函數的程序。

♣ 子程序(child process): fork()回傳後，新產生的程序。

🍏 回傳值:

♣ Parent: child's pid

♣ Child: 0

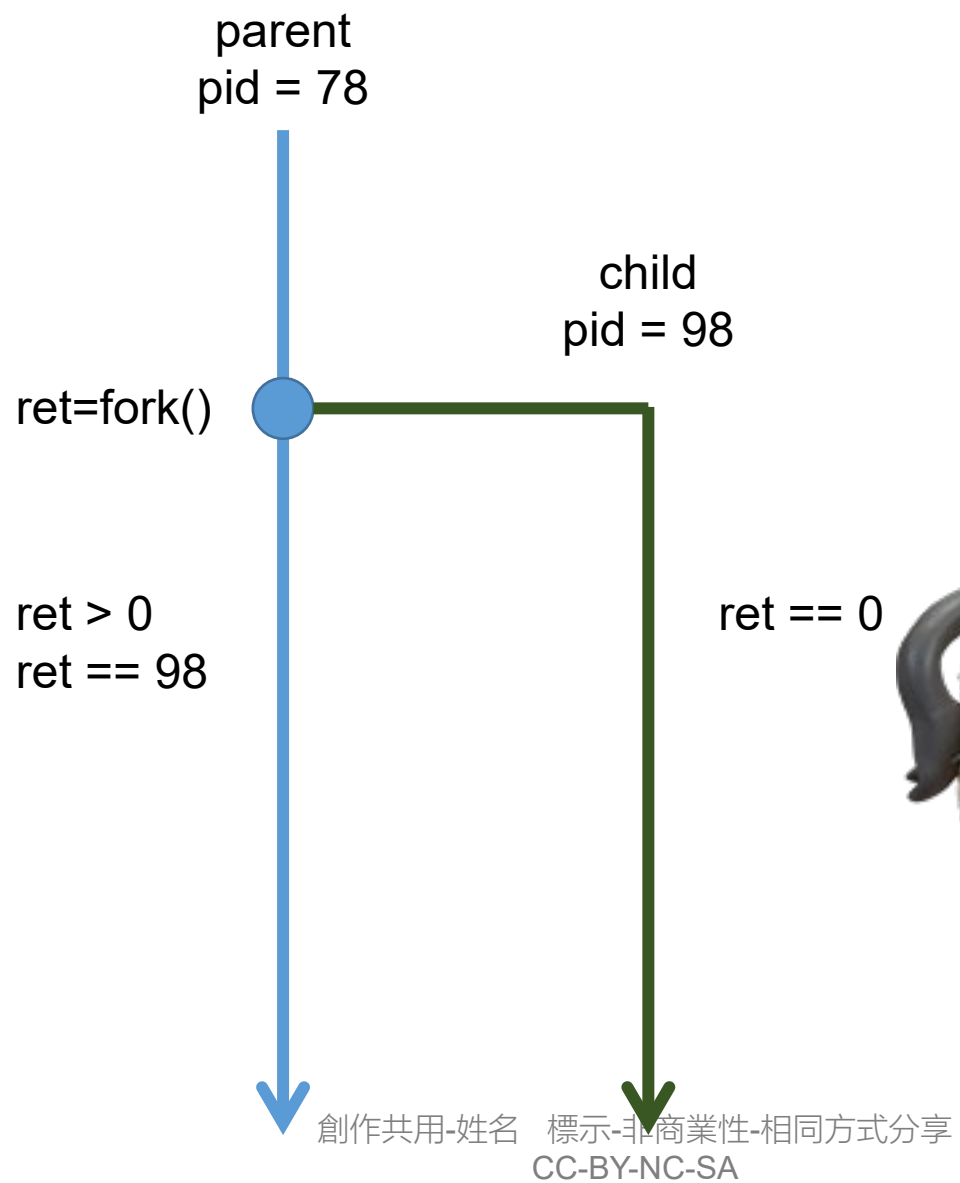
♣ 失敗: -1

🍏 fork()功用

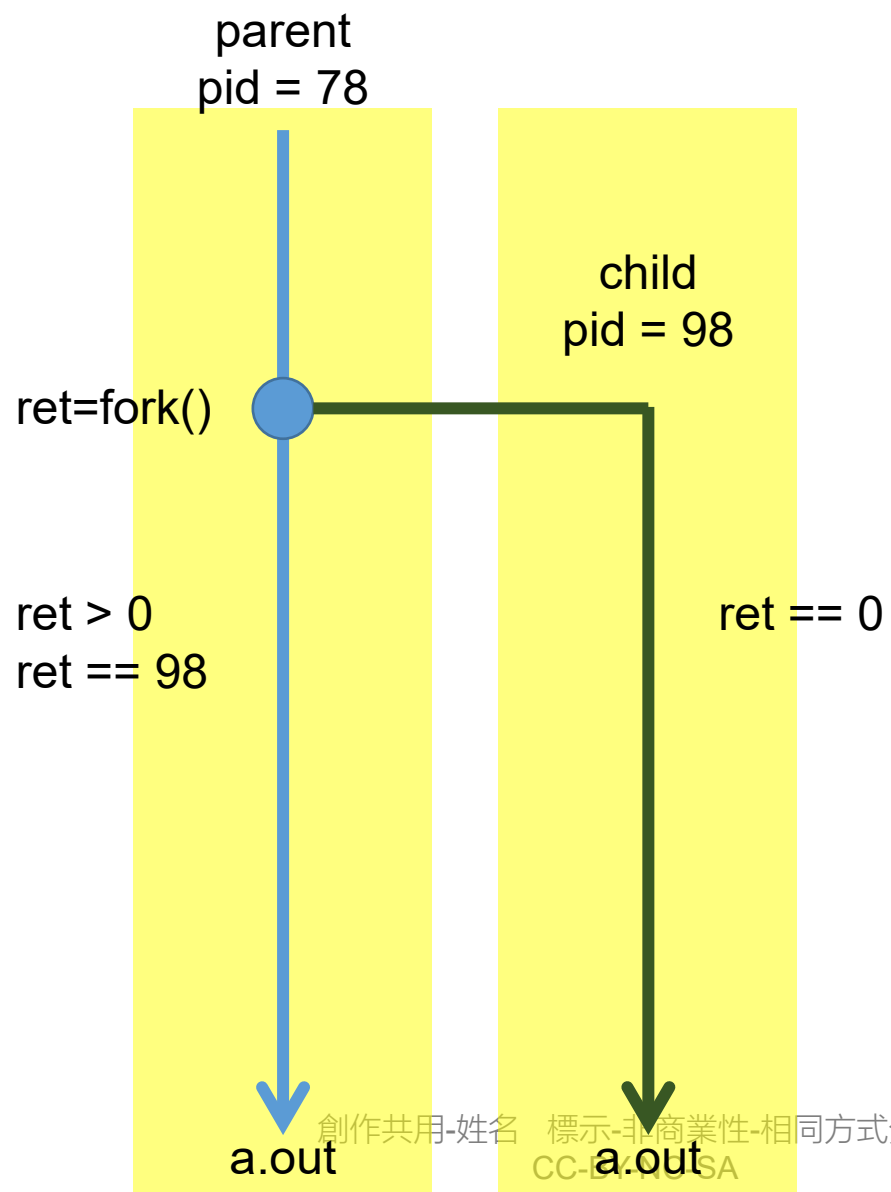
♣ 分工合作: 父程序與一個以上的子程序，共同分工合作完成任務。

♣ 執行新程式: 由子程序執行新的程式，父程序可選擇等待或不等子程序完成執行程式。

fork()



fork()



fork1.c

```
1. #include <stdio.h>
2. #include <unistd.h>
3. int main()
4. {
5.     int var = 0;
6.     pid_t pid;
7.     pid = fork();
8.     printf("%d", var); ← child, parent 都會執行
9.     if(pid == 0) { /* child 執行 */
10.         var = 1;
11.     }
12.     else if (pid > 0) { /* parent 執行 */
13.         var = 2;
14.     }
15.     printf("%d", var) ← child, parent 都會執行
16.     return 0;
17. }
```


fork1.c

- 🍏 利用fork()的回傳值 (亦即pid)進行父程序與子程序各自需要完成任務的程式碼區分
- 🍏 各程序的執行結果：
 - ♣️ 父程序: 02
 - ♣️ 子程序: 01
- 🍏 然而，系統並沒有限制兩個程序的執行順序
 - ♣️ 若fork()成功，執行結果可能為: 0012, 0021, 0102, 或 0201
 - ♣️ 若fork()失敗，執行結果為: 00
- 🍏 你的實際結果是？
 - ♣️ Linux的執行順序其實是固定的 如果只有一顆 CPU，預設 child 先執行
 - ♣️ `/proc/sys/kernel/sched_child_runs_first`

調換fork與printf(frok2.c)，其輸出結果為？

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  int main()
4.  {
5.      int var = 0;
6.      pid_t pid;
7.      printf("%d", var);
8.      pid = fork();
9.      if(pid == 0) { /* child 執行 */
10.         var = 1;
11.     } else if (pid > 0) { /* parent 執行 */
12.         var = 2;
13.     }
14.     printf("%d", var)
15.     return 0;
16. }
```

他會印兩次，因為 `_IOLBF` 暫存在 buffer 裡
fork 後，連著 buffer 內容也 copy 過去

printf是line buffer

- 🍏 『printf(“%d”, var);』 會被queue起來，放在buffer中
- 🍏 fork時，會將所有記憶體，包含buffer複製出一份



對fork
除錯

debug fork()

- 🍏 When a program forks, gdb will continue to debug the parent process and the child process will run unimpeded.
 - ~~🍀 If you have set a breakpoint in any code which the child then executes, the child will get a SIGTRAP signal which (unless it catches the signal) will cause it to terminate. ???~~
 - 🍀 testDebug.c
- 🍏 debugging the child process
 - 🍀 using touch commend (hint: put sleep after fork)
 - 🍀 debug child process instead of parent process (using "set follow-fork-mode child")

<http://sourceware.org/gdb/onlinedocs/gdb/Forks.html>

中正大學 - 羅翊珉 http://www.cnblogs.com/zhenjing/archive/2011/06/01/gdb_fork.html

debug fork() – debugFork1.c

```
$gdb debugFork1
```

```
(gdb)b main /*set a breakpoint*/
```

```
(gdb)r /*run*/
```

```
(gdb)n /*next*/
```

```
(gdb) show follow-fork-mode
```

Debugger response to a program call of fork or vfork is “parent”

```
(gdb)c /*continue*/
```

debug fork() – debugFork1.c

...

```
(gdb) set follow-fork-mode child
```

```
(gdb) n
```

...

```
(gdb) p pid /*child OR parent?*/
```

debug fork() – debugFork2.c

```
$gdb debugFork2  
(gdb) b 9  
(gdb) r  
(gdb) n  
(gdb) p pid 29979
```

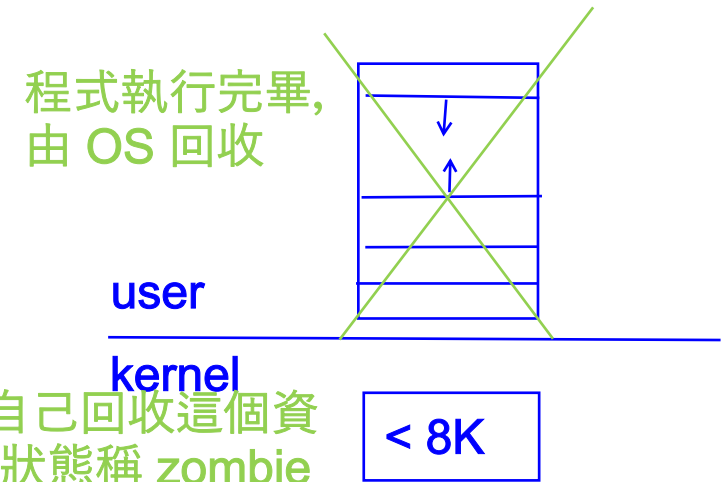
```
/*  
開第二個terminal  
*/      debug 別人的程式碼  
$sudo gdb debugFrok2  
(gdb) attach 29979  
(gdb) set waiting=0  
/*continue*/
```

debugFork2.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.
4.  int main(void) {
5.      pid_t  pid;
6.      int waiting = 1;
7.
8.      if ( (pid = fork()) == 0 ) {
9.          while(waiting)
10.             ;
11.         printf("child");
12.     } else {
13.         printf("child's pid = %d\n",
14.             pid);
15.         printf("parent");
16.     }
17.     return 0;
18. }
```

A collage of various Linux mascot characters, including several penguins in different outfits (some wearing hats, some holding tools, some reading books) and a bull. A large, stylized orange 'wait()' text is overlaid on the right side of the image. In the center, there is a green penguin wearing a fire helmet with the number '18' and 'F.D.N.Y.' on it, and a large orange arrow pointing towards it. The background is white with some faint, colorful, abstract shapes.

等待子程序狀態轉換：wait()



🍏 當子程序狀態轉換時，父程序負責處理其狀態。

🍀 子程序結束：父程序等待後，可使子程序所佔用的資源釋放還給系統。

🍇 若父程序沒有等帶子程序，子程序會變成殭屍程序(zombie process)。return 什麼值

🍀 子程序收到號誌後暫停運行 (stopped)。

🍀 子程序收到號誌後恢復運行 (resumed)。

parent 執行 wait(), 將這個資料結構的內容收回，此時 OS 確定 parent 已經拿到他想要的東西，這個 child 才會真正結束，記憶體徹底釋放

🍀 號誌 (signal) 將在下個章節介紹

wait & waitpid

🍏 `#include <sys/types.h>`

🍏 `#include <sys/wait.h>`

🍏 `pid_t wait(int *wstatus);` 任何一個 child 結束後都會回傳

🍏 `pid_t waitpid(pid_t pid, int *wstatus, int options);`

🍏 回傳值：

🍀 「大致上」成功傳回該child的pid，失敗的話回傳-1

🍀 詳細的部分請看man 2 wait

wait(int *wstatus);

檢查子程序狀態可使用表中的macro。號誌 (signal) 下一節介紹。

MACRO	描述
WIFEXITED(status)	TRUE, 若子程序正常結束
WEXITSTATUS(status)	子程序結束狀態 (exit status)
WIFSIGNALED(status)	TRUE, 若子程序被號誌終止
WTERMSIG(status)	子程序被號誌終止的號誌編號 (signal number)
WCOREDUMP(status)	TRUE, 若子程序有產生CORE檔
WIFSTOPPED(status)	TRUE, 若子程序被號誌暫停
WSTOPSIG(status)	子程序被號誌暫停的號誌編號 (signal number)
WIFCONTINUED(status)	TRUE, 若子程序收到SIGCONT號誌而恢復運行

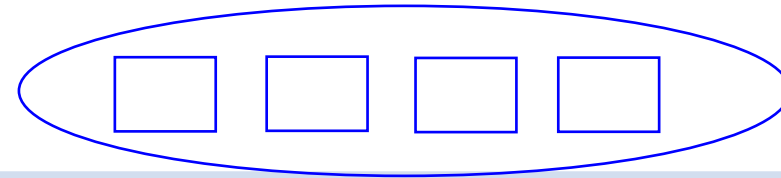
不正確結束, coredump, OS 會將這個死掉的程式的記憶體內容存到硬碟裡, 即為 core 檔

waitpid(pid, *wstatus, opt)

這個程式是 40 年前的程式了，現在如果要設計類似的程式的話，最好多加一個 **type** 清楚表明是要哪一種

The value of pid can be:

Group: 一群彼此相關的 process



< -1	任意一個group id為 pid 的child結束	在 Linux 裡，group 裡 pid 最小的程式就是這個 group 的 id
-1	任意的一個child	
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	

Process group

```
$ ls -R | egrep "\\.c$" | wc -l  
335
```

/*共有ls, egrep, wc三個程式一起完成工作, bash將這三個程式設成同一個process group*/

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

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

```
int setpgid(pid_t pid, pid_t pgid);
```

```
pid_t getpgid(pid_t pid);
```

```
/*設定process group的function, 請自行查閱*/
```


waitpid(pid, *wstatus, opt)

blocking: 我叫他做他就會做，但是他之前可能還有其他事情要做，我叫他做的事可能要等很久
asynchronous: 我叫他做他就會做，而我可以往下繼續執行

The value of options is an OR of zero or more of the following constants:

簡單來說，就是當child還未結束時，parent想要知道child的狀態
有些 sys. call 或 func. call 是 non-blocking (不阻塞) 的 (可以做就立刻做，不能做就立刻回傳不能做)，

WNOHANG	return immediately if no child has exited. non-blocking
WUNTRACED	also return if a child has stopped (but not traced via ptrace(2), 簡單來說被ptrace就是被debug). ptrace 是一個 sys. call, 用途為 debug
WCONTINUED	also return if a stopped child has been resumed by delivery of SIGCONT. (SIGCONT是signal, 下個章節介紹)

wait.c

```
1.  int main(int argc, char *argv[]) {  
2.      pid_t cpid, w;  
3.      int wstatus;  
4.      cpid = fork();  
5.      if (cpid == 0) {          /* Code executed by child */  
6.          printf("Child PID is %ld\n", (long) getpid());  
7.          pause();              /* Wait for signals */  
8.      }
```

```
9.     else {           /* Code executed by parent */
10.     do {
11.         w = waitpid(cpid, &wstatus, WUNTRACED | WCONTINUED);
12.         if (WIFEXITED(wstatus))
13.             printf("Parent: child is exited, status=%d\n", WEXITSTATUS(wstatus));
14.         if (WIFSIGNALED(wstatus))
15.             printf("Parent: child is killed by signal %d\n", WTERMSIG(wstatus));
16.         if (WIFSTOPPED(wstatus))
17.             printf("Parent: child is stopped by signal %d\n", WSTOPSIG(wstatus));
18.         if (WIFCONTINUED(wstatus))
19.             printf("Parent: child is continued\n");
20.     } while (!WIFEXITED(wstatus) && !WIFSIGNALED(wstatus));
21.     /*當子行程沒有結束並且沒有被signal終止*/
22.     printf("Parent: bye bye\n");
23.     exit(EXIT_SUCCESS);
24. }
25. }
```

pause()

1. `#include <unistd.h>`

2. `int pause(void);`

用途：讓行程一直睡覺，直到遇到終止號誌（signal），或者遇到可以處理的號誌

執行結果

使用 2 個 terminal 操作

```
$ ./wait
```

```
Child PID is 9006
```

```
Parent: child is stopped by  
signal 19
```

```
Parent: child is continued
```

```
Parent: child is killed by signal  
15
```

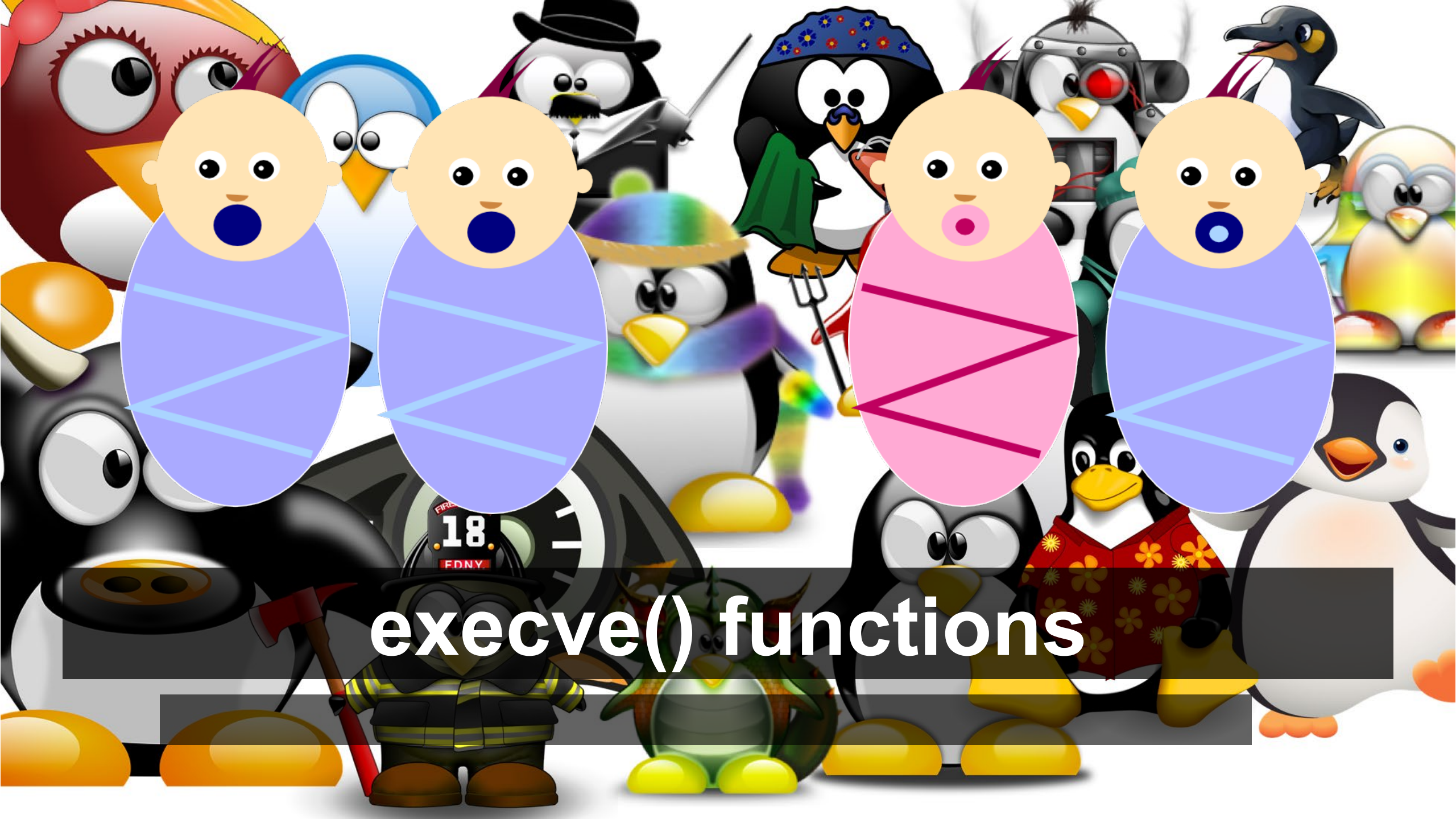
```
Parent: bye bye
```

```
$ kill -STOP 9006
```

```
$ kill -STOP 9006
```

```
$ kill -CONT 9006
```

```
$ kill -TERM 9006
```



execve() functions

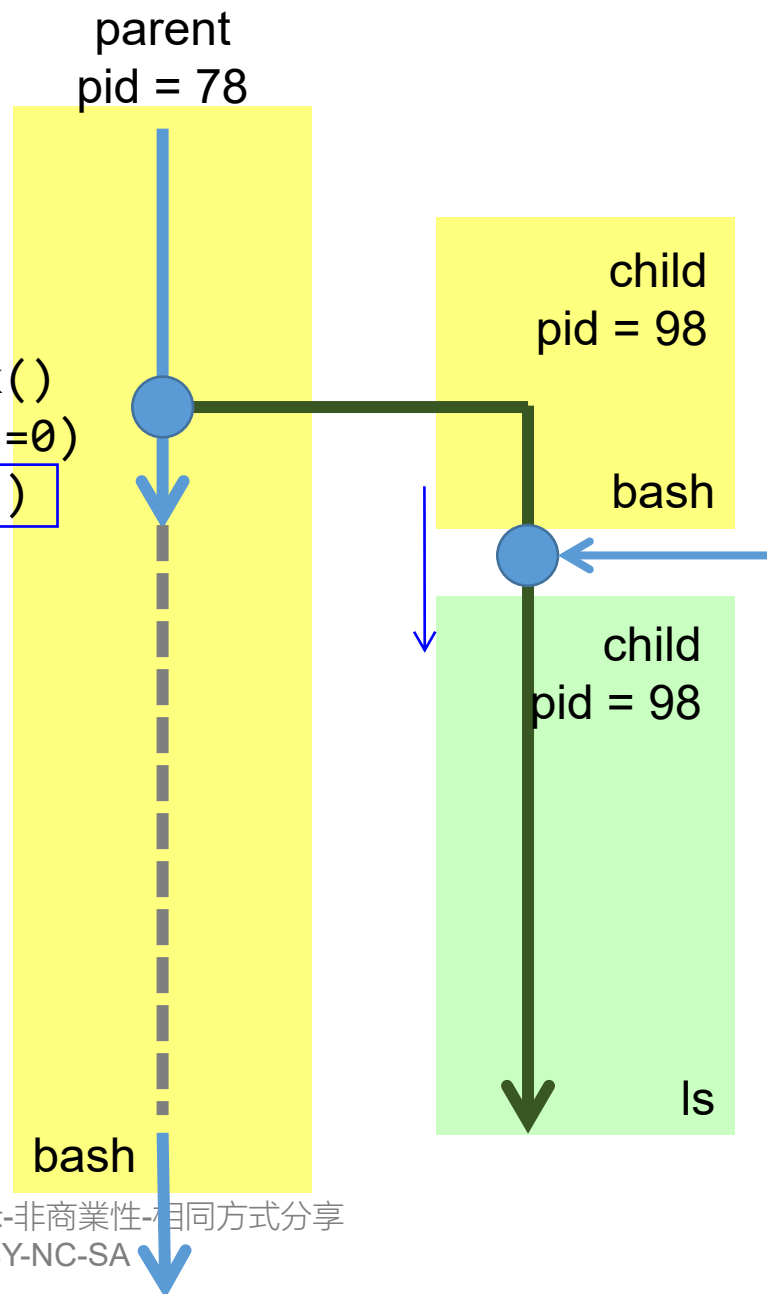
fork() + execve

執行 `execve` 後，有些東西會繼承下去

pid
開檔
環境變數
部分 signal 的狀態

我要等待並知道
child 的執行結果，
我才繼續執行

```
ret=fork()  
if (ret!=0)  
    wait()
```



```
if(ret == 0)  
    execv("ls");
```

execve-family

```
#include <unistd.h>

extern char **environ;

int execl(const char *path, const char *arg, ...
/* (char *) NULL */);

int execlp(const char *file, const char *arg, ...
/* (char *) NULL */);

int execlx(const char *path, const char *arg, ...
/*, (char *) NULL, char * const envp[] */);

int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
            char *const envp[]);
```

execve-family

- 🍏 `execl`, `execvp`, and `execle`的第二個以後的參數是一群字串，
「習慣上」第一個字串是執行檔本身，最後一個參數必須是
`(char*)NULL`
- 🍏 `execv`, `execvp`, `execvpe`的第二個參數是：字串陣列
- 🍏 `execle`, `execvpe`環境變數放在 `envp`
- 🍏 失敗回傳-1，錯誤訊息放在 `errno`

execve-family

🍏 The `execvp()`, `execvp()`, and `execvpe()` 如果執行檔的path-name沒有"/"那麼這幾個函數會依照PATH搜尋執行檔。

🍏 例如：

```
$ echo $PATH  
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
```

execve-family

表格整理

	list			vector		
參考PATH?	NO	YES ✓	NO	NO	YES	YES
參考environ?	YES	YES	NO	YES	YES	NO
函數	execl()	execvp()	execle()	execv()	execvp()	execvpe()

myShell.c (完整程式碼請直接看.c檔)

```
1.  char* argVect[256];
2.  int main (int argc, char** argv) {
3.      while(1) {
4.          printf("myShell$ ");
5.          fgets(cmdLine, 4096, stdin);
6.          parseString(cmdLine, &exeName);
7.          if (strcmp(exeName, "exit") == 0)
8.              break;
9.          pid = fork();
10.         if (pid == 0)
11.             execvp(exeName, argVect);
12.         else
13.             wait(&wstatus);
14.     }
15. }
```



```
16. void parseString(char* str, char** cmd) {
17.     int idx=0;
18.     char* retPtr;
19.     retPtr=strtok(str, "\n");
20.     while(retPtr != NULL) {
21.         argVect[idx++] = retPtr;
22.         if (idx==1)
23.             *cmd = retPtr;
24.         retPtr=strtok(NULL, "\n");
25.     }
26.     argVect[idx]=NULL;
27. }
```

執行結果

```
$ ./myShell
shiwulo@NUC:~/Dropbox/course/2018-sp/ch09>> ls *.c
ls: cannot access '*.c': No such file or directory
return value of ls is 2
shiwulo@NUC:~/Dropbox/course/2018-sp/ch09>> ls myShell myShell.c -lh
-rwxr-xr-x 1 shiwulo shiwulo 17K May 17 13:08 myShell
-rw-rw-r-- 1 shiwulo shiwulo 4.1K May 17 13:08 myShell.c
return value of ls is 0
shiwulo@NUC:~/Dropbox/course/2018-sp/ch09>> ls myShell myShell.c -lh --color
-rwxr-xr-x 1 shiwulo shiwulo 17K May 17 13:08 myShell
-rw-rw-r-- 1 shiwulo shiwulo 4.1K May 17 13:08 myShell.c
return value of ls is 0
shiwulo@NUC:~/Dropbox/course/2018-sp/ch09>> cd ~
shiwulo@NUC:~>> ggg
myShell: No such file or directory
return value of ggg is 254
```

System call - execve

不建議使用，因為可能換了 OS，就不叫 `execve` 了
但是前面介紹的 6 個 C API func. call 是一定有的

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

```
int execve(const char *filename, char *const argv[], char *const envp[]);
```



install kdbg

使用校內網路

```
$ sudo wget -O /etc/apt/source.list  
http://lonux.cs.ccu.edu.tw/source.list.17.10
```

```
$sudo apt install kdbg
```




vfork & execve

Windows建立行程

```
BOOL WINAPI CreateProcess(  
_In_opt_ LPCTSTR lpApplicationName,  
_Inout_opt_ LPTSTR lpCommandLine,  
_In_opt_ LPSECURITY_ATTRIBUTES lpProcessAttributes,  
_In_opt_ LPSECURITY_ATTRIBUTES lpThreadAttributes,  
_In_ BOOL bInheritHandles,  
_In_ DWORD dwCreationFlags,  
_In_opt_ LPVOID lpEnvironment,  
_In_opt_ LPCTSTR lpCurrentDirectory,  
_In_ LPSTARTUPINFO lpStartupInfo,  
_Out_ LPPROCESS_INFORMATION lpProcessInformation );
```

Linux vs. Windows: 於執行新的program

- 🍏 Linux要二個system call
 - ♣️ fork產生行程
 - ♣️ execve將新的執行檔放到這個新的行程的記憶體空間
- 🍏 Windows只需要一個system call
 - ♣️ 即CreateProcess(..., filename,...)
- 🍏 Linux的設計較適合用來設計server
 - ♣️ http, ftp, BBS等
 - ♣️ 上述server在每一個使用者連線以後，都執行相同的執行檔
 - ♣️ 換言之，只要fork就可以服務一個client

fork vs. vfork

```
1.  #include <sys/types.h>
2.  #include <unistd.h>
3.  #include <stdlib.h>
4.  #include <stdio.h>
5.  int main(int argc, char** argv) {
6.      int i, num, pid, wstatus;
7.      sscanf(argv[1], "%d", &num);
8.      for (int i=0; i<num; i++) {
9.          pid = fork();
10.         if (pid==0) exit(0);
11.         if (pid != 0) continue;
12.     }
13. }
```

fork vs. vfork

```
1.  #include <sys/types.h>
2.  #include <unistd.h>
3.  #include <stdlib.h>
4.  #include <stdio.h>
5.  int main(int argc, char** argv) {
6.      int i, num, pid, wstatus;
7.      sscanf(argv[1], "%d", &num);
8.      for (int i=0; i<num; i++) {
9.          pid = vfork();
10.         if (pid==0) exit(0);
11.         if (pid != 0) continue;
12.     }
13. }
```

比較

manyFork

```
$ time ./manyFork 100000  
real 0m1.810s  
user 0m0.073s  
sys 0m1.612s
```

manyVFork

```
$time ./manyVFork 100000  
real 0m0.534s  
user 0m0.005s  
sys 0m0.404s
```

使用vfork替代fork

“只有在fork後馬上跟著
execve時才用vfork”

否則child會改到parent的資料，這通常是我們不樂見的

使用vfork替代fork

vfork並不能完全取代fork

因為parent會暫停執行，直到child執行execve或結束

使用vfork替代fork

🍏 在上述前提下，可以直接將fork改成vfork

🍏 課堂作業：

將myShell改用vfork

殭屍問題 (zombie process)



zombie.c

```
1.  int main(int argc, char** argv) {
2.      int i, num, pid, wstatus;
3.      sscanf(argv[1], "%d", &num);
4.      for (int i=0; i<num; i++) {
5.          pid = vfork();
6.          if (pid==0) exit(0);
7.          if (pid != 0) continue;
8.      }
9.      if (pid != 0)
10.         getchar();
11. }
```

結果

產生zombie

```
./zombie 10000
```

觀察

```
$ps -a
32753 pts/0      00:00:00 zombie <defunct>
32754 pts/0      00:00:00 zombie <defunct>
32755 pts/0      00:00:00 zombie <defunct>
32756 pts/0      00:00:00 zombie <defunct>
32757 pts/0      00:00:00 zombie <defunct>
32758 pts/0      00:00:00 zombie <defunct>
32759 pts/0      00:00:00 zombie <defunct>
32760 pts/0      00:00:00 zombie <defunct>
```

結果 – 造成的影響

```
$ sudo less /proc/slabinfo | grep task_struct
# name          <active_objs>    <num_objs>       <objsize>
task_struct     10257            10280            6016
```

/*總共浪費掉 $10257 * 6016 = 58.84\text{MB}$ 核心記憶體*/

```
$ ulimit -a | grep process
```

```
max user processes          (-u) 127320
```

/*每個使用者可以開啟的process數量有限，zombie process也算在內*/

/*zombie process沒辦法kill掉*/



- 🍏 因此parent一定要執行wait(), 將child回收
- 🍏 但如果忘記, 怎麼辦! ?

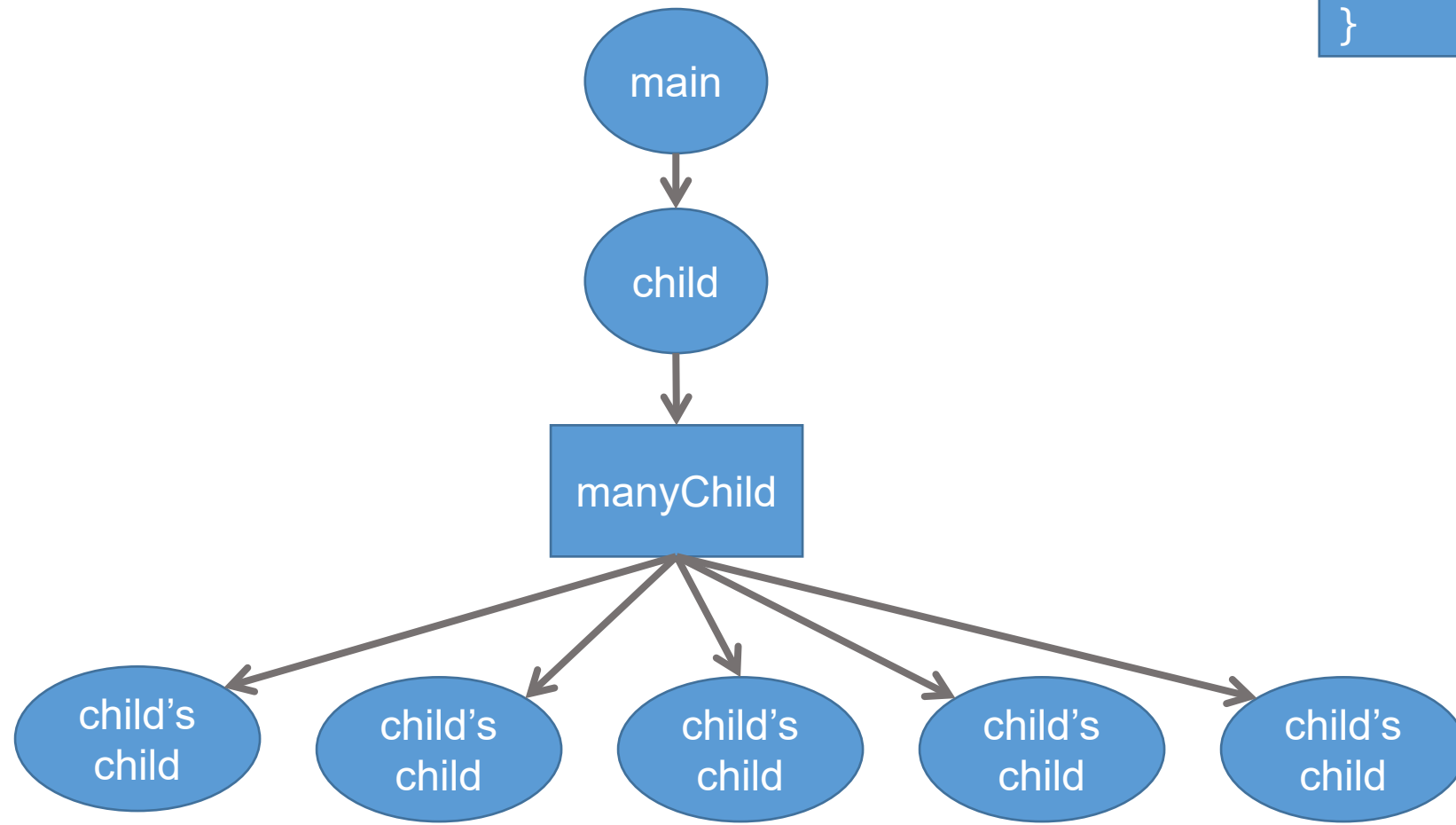
解決方法 NoZombie.c

```
1. void manyChild(int num) {  
2.     int i, pid;  
3.     for (int i=0; i<num; i++) {  
4.         pid = vfork();  
5.         if (pid == 0) exit(0);  
6.         if (pid != 0) continue;  
7.     }  
8. }
```

```
9. int main(int argc, char** argv) {  
10.     int pid, num;  
11.     sscanf(argv[1], "%d", &num);  
12.     pid = fork();  
13.     if (pid == 0) {  
14.         manyChild(num);  
15.         exit(0);  
16.     }  
17.     getchar();/*main處理其他工作*/  
18. }
```

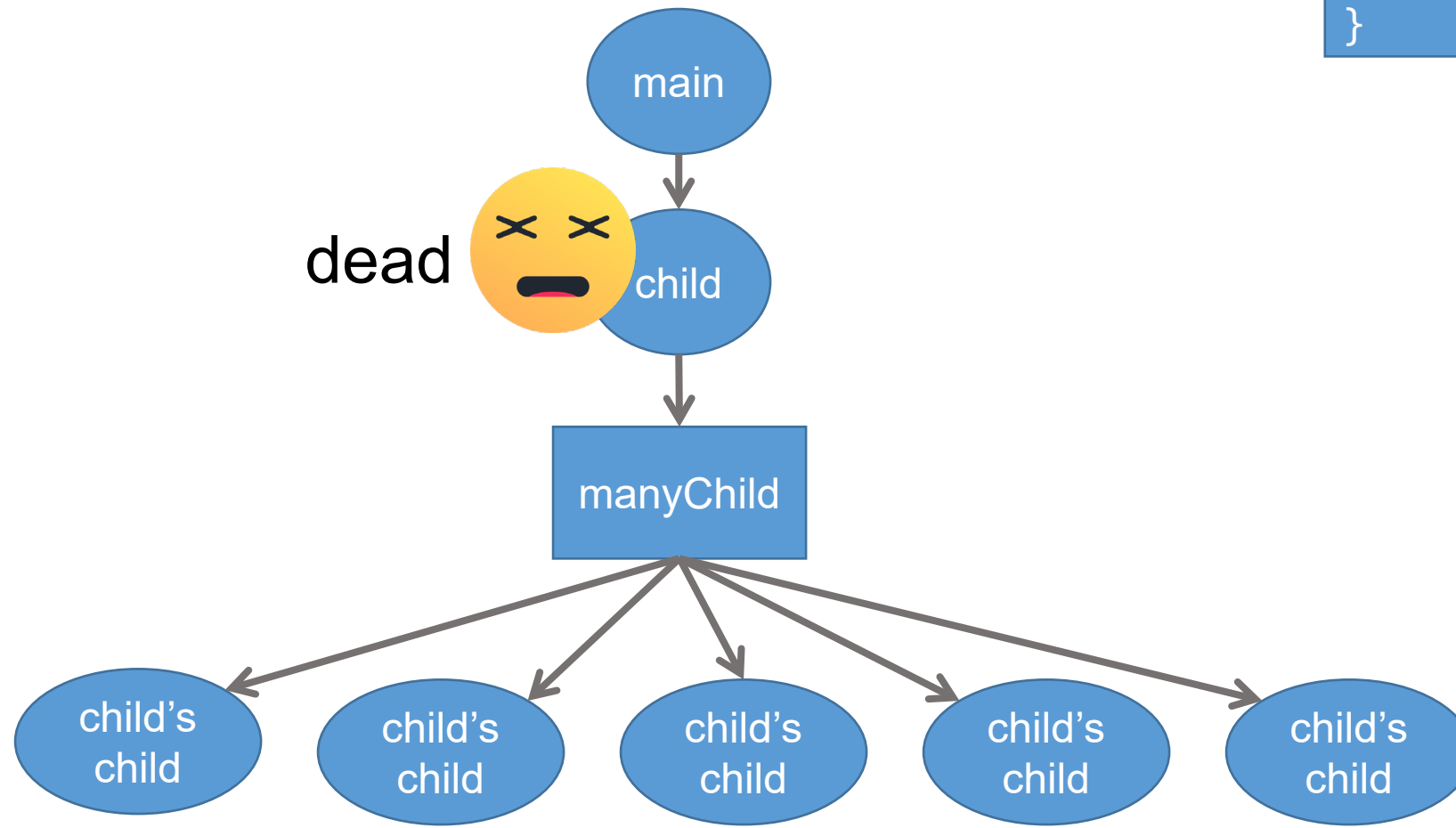
NoZombie概念圖

```
/*pid == 1*/  
init {  
    while(1)  
        wait();  
}
```

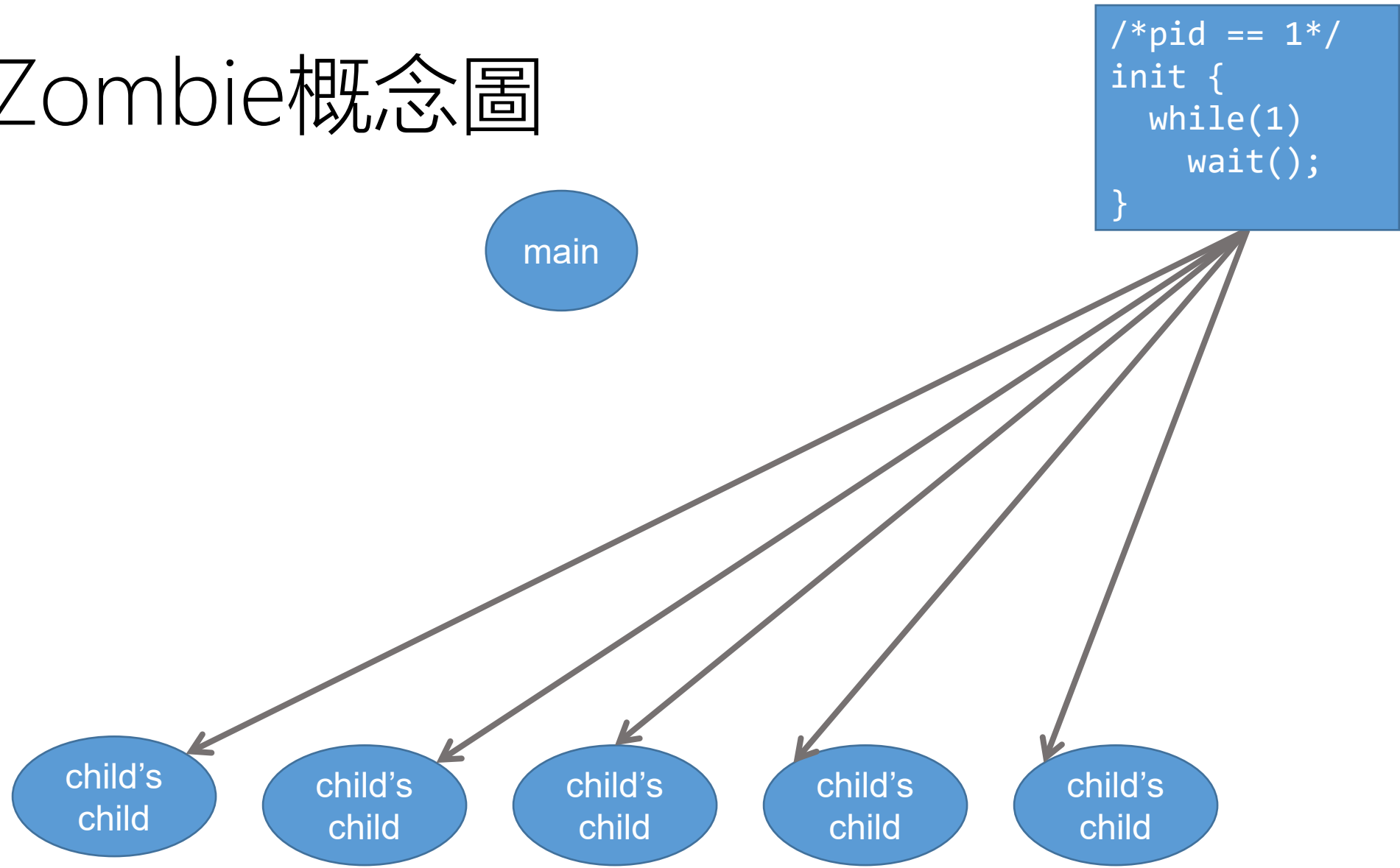


NoZombie概念圖

```
/*pid == 1*/  
init {  
    while(1)  
        wait();  
}
```



NoZombie概念圖



小結

- 🍏 `fork()`: 用以產生新的process, 新的、舊的都來自於同一個執行檔
- 🍏 `wait()`: 通常用於parent等child完成
- 🍏 Zombie process: child結束後, 若parent沒有執行wait, 會造成zombie, zombie使用kill殺不掉
- 🍏 `execve`: 載入新的執行檔到process的記憶體空間, 通常會和`vfork()`一起使用

作業一

🍏 請寫底下程式，驗證執行結束後總共會有多少個process

1. `#include <stdio.h>`

2. `int main() {`

3. `fork();`

4. `fork();`

5. `fork();`

6. `fork();`

7. `}`

作業二

🍏 修改myShell，改用execle實作，要能夠用PATH搜尋執行檔