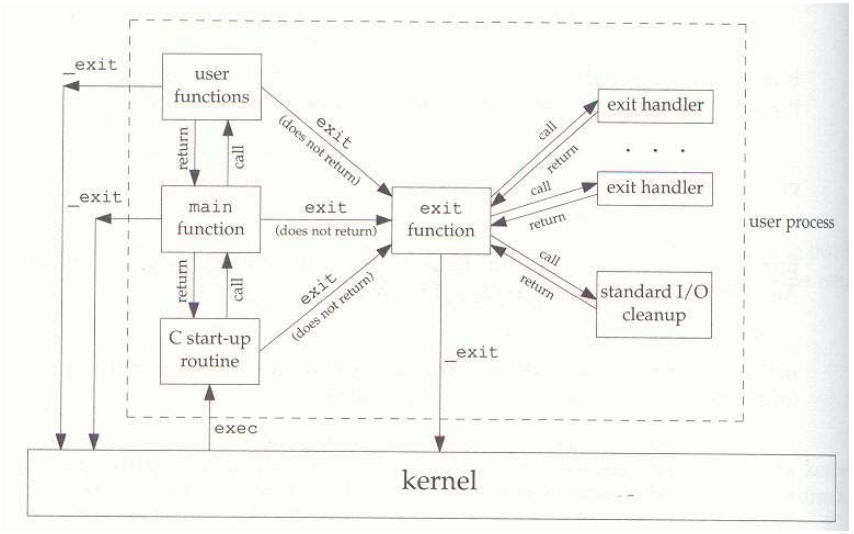
六、行程控制

行程的生成與結束(1/2)

- 行程的啟動
  - C程式是由main()函式開始執行，其原型為
    int main(int argc, char *argv[]);
  - 每個C程式的main()函式是由一特別的start-up程序所帶起，這個程序用來從kernel讀取所需的值(環境變數，參數等等)，並設定給main函式。這個程序在編譯的過程中會由編譯器加入執行檔
- 行程的結束(有5種方式)
  - 正常結束(normal termination)
    - 由main函式中return (相當於 exit(main(argv, argv));)
    - 呼叫 exit
    - 呼叫 _exit
  - 異常結束(abnormal termination)
    - 呼叫abort
    - 被signal 結束

行程的生成與結束(2/2)



exit與_exit的比較

```
#include <stdlib.h>

    void exit(int status);

#include <unistd.h>

    void _exit(int status);
```

- 這兩個函式都是用來正常結束一個行程，exit會執行一些cleanup的動作(例如關閉標準I/O函式庫的檔案)，然後到kernel做後續動作，而_exit是直接到kernel中
  - exit會關閉標準I/O函式庫所開啟的檔案(fclose)，而_exit不會
  - exit會執行atexit所設定的函式，而_exit不會
- 這兩個函式皆有一個參數status，用來代表結束狀態(exit status)

註冊正常結束時所需執行的程序

```
#include <stdlib.h>

    int atexit(void (*function)(void));
    int on_exit(void (*function)(int , void *), void *arg);
```

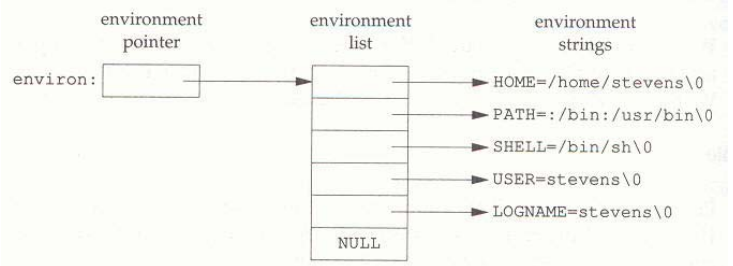
- 這兩個函式用來註冊呼叫exit函式後所需執行的函式。其參數包含一個函式指標，其中atexit只能註冊不需參數的函式，而on_exit可以註冊需要參數的函式
- 可以註冊多個函式，當exit被呼叫時，這些函式會以註冊順序的反向開始執行(也就是越晚註冊的越早被執行)

命令列參數（arguments)

- int main(int argc, char *argv[])
  其中argc為命令列參數的個數，argv為一包含char *的陣列
- 當一個程式被執行時，可以傳遞命令列參數給它，例如./a.out abc 1 2，其中./a.out，abc，1和2皆為命令列參數。程式可以藉由讀取命令列參數取得一些所需的值
- 在ANSI C與POSIX.1的標準中，明訂argv[argc]為null pointer，因此我們可以用下列方式把命令列參數給列出來
  - for(i=0; argv[i]!=NULL; i++) printf("%s\n",argv[i]);
  - 或用for(i=0; i<argc; i++) printf("%s\n",argv[i]);

環境變數表 (environment list)

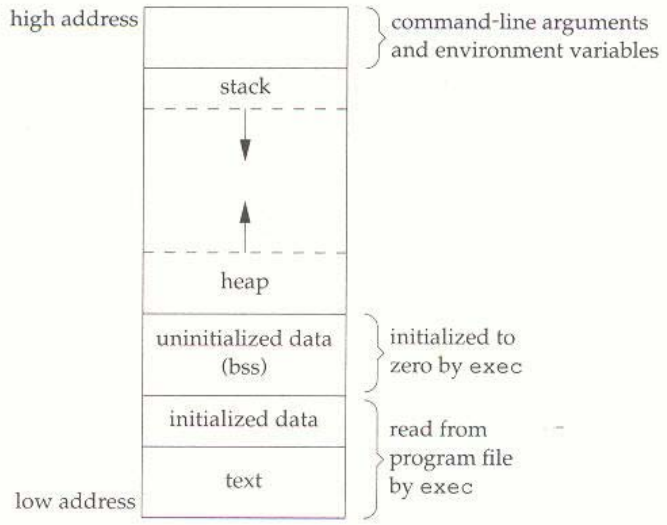
- 要傳遞參數給一個程式，除了使用命令列參數外，也可以使用環境變數。
- 環境變數表為一包含char *(字元指標)的陣列，其結尾的內容為null pointer。程式如要存取，可以宣告如下：
  extern char **environ;
  因此要把所有環境變數列出來，可以使用下列方式
  char **ptr;
  for(ptr=environ; *ptr!=NULL; ptr++) printf("%s\n",*ptr);
- 每個環境變數皆為 name=value 的樣式，例如TERM=vt100



行程記憶體配置樣式(layout)(1/2)

- 一個行程中包含下列部分
  - Text segment (程式碼區段)
    - 通常相同程式的行程可以共用此一區段，這區段通常是唯讀的
  - Initialized data segment (有初始值的資料，稱為資料區段)
    - 例如程式中有全域變數int a=10;的敘述，其中a就為有初始值的資料，這類區段會佔據程式(program)的空間
  - Uninitialized data segment(未初始的資料，稱為bss)
    - 例如程式中有全域變數long sum[10000]，這些陣列元素就為未初始的資料，這類區段不會佔據程式(program)的空間，而是等執行後由kerenl來初始化，kernel會將這些資料初始值設為0或是null pointer
  - Stack (堆疊)
    - 每一個行程都有自己的堆碟，用來儲存每個呼叫函式的自動變數(automatic variable)與相關資訊。當有函式被呼叫時，其返回位址，caller的相關資訊(暫存器的值)都會儲存在這裡，待函式執行完返回時再由這些值還原
  - Heap
    - 作為動態記憶體配置的空間
- 可以用size(1)命令來觀看一個程式的記憶體使用情況

行程記憶體配置樣式(layout)(2/2)



動態記憶體配置(1/2)

```
#include <stdlib.h>

    void *calloc(size_t nmemb, size_t size);
    void *malloc(size_t size);
    void free(void *ptr);
    void *realloc(void *ptr, size_t size);
```

- 在ANSI C中有三個函式是用來動態配置記憶體
  - malloc函式用來配置大小為size位元組的記憶體，其配置出來的初始值不定
  - calloc函式用來配置單位大小為size位元組，總共nmemb個單位的記憶體區塊，其配置出來的初始值皆為0(每個bit都設為0)
  - realloc函式用來配置或是改變先前配置的記憶體區塊大小，如果是增加的話，其增加部分的初始值也是不定的。如果ptr為null pointer，此函式就如同malloc。另外注意size為新的大小，而不是與舊大小之間的差異
  - 以上函式所配置出來的記憶體保證可以用在任何的data object上，也就是沒有alignment的問題(取所有data objects所需align位置的最小公倍數)
- free 函式用來釋放上述三個函數所配置的記憶體

動態記憶體配置(2/2)

```
#include <alloca.h>

    void *alloca(size_t size);
```

- 此函式與 malloc相似，但它所配置的記憶體空間是在stack中而不是heap，因此它不需要呼叫free來釋放，而是等該函式返回時會自動釋放
- 此函式適合用來配置函式內部所需的動態記憶體，如果使用malloc，在函式離開前必須呼叫free來釋放，使用alloca則無此問題
- alloca函式有下列缺點
  - 不是所有系統都支援這個函式
  - 配置後就不能改變大小

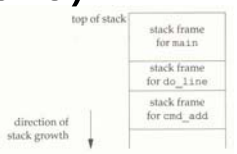
操作環境變數

```
#include <stdlib.h>

    char *getenv(const char *name);
    int putenv(char *string);
    int setenv(const char *name, const char *value, int overwrite);
    void unsetenv(const char *name);
```

- 之前提過，可以透過environ這個變數來存取環境變數，但是如果是要增加環境變數，透過此頁所提的函式較為簡單
- getenv函式用來讀取某個環境變數的值，例如getenv(“HOME”);可以取得使用者的home目錄
- putenv函式用來新增或修改環境變數，其參數的格式必須為name=value，例如putenv(“HOME=/home/native”);。如果name已經存在，舊的值會先被刪除
- setenv函式功能與putenv相似，但它把name=value拆成兩個參數，而最後一個參數用來指定當name已經存在時是否可以覆寫。例如setenv(“HOME”,”/home/native”,1);
- unsetenv函式用來刪除環境變數，例如unsetenv(“HOME”);

非區域跳躍(non-local goto)

```
#include <setjmp.h>

    int setjmp(jmp_buf env);
    void longjmp(jmp_buf env, int val);
```



- 在C語言中，goto只能在同一個函式中跳躍，透過setjmp與longjmp則可以在不同的函式間跳躍
- 適用時機為需要一次返回很多函式時(像是錯誤處理)，例如有A，B，C三個函式，假設其執行流程為A呼叫B，B呼叫C，當C發生錯誤時，希望直接返回A繼續做，而不是先返回B，B發現錯誤後再返回A
- setjmp函式會將當時stack的情況(context)儲存在env中，供日後longjmp函式使用。其返回值有兩種情況
  - 儲存context時的返回值為0
  - 使用longjmp函式時會從對應的setjmp返回，此時返回值為longjmp函式所設定
- longjmp函式用來跳躍到儲存env的setjmp函式，其val參數為非0的值，代表setjmp所傳回的值，因此藉由設定不同的val值，setjmp可以知道是由哪一個longjmp所跳回
- 特別注意，自動變數與register變數不一定會roll back，如果不要讓它roll back可以將變數設定為volatile

取得與設定行程資源限制(1/2)

```
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>

    int getrlimit(int resource, struct rlimit *rlim);
    int setrlimit(int resource, const struct rlimit *rlim);
```

- getrlimit函式用來取得行程的某個資源限制，在rlimit結構中包含兩個成員，分別為rlim_cur(soft limit)，與rlim_max(hard limit)
- setrlimit函式用來設定行程的某個資源限制，其更改的規則如下
  - soft limit必須小於等於hard limit
  - 一般使用者設定hard limit時必須介於舊的hard limit與新的soft limit之間
  - 只有root可以調高hard limit
- 常數RLIM_INFINITY代表無限

```
struct rlimit {
        rlim_t rlim_cur;
        rlim_t rlim_max;
};
```

取得與設定行程資源限制(2/2)

可以設定的資源如下：

- RLIMIT_CPU
  - 限制可以使用CPU的時間，單位為秒。當時間用完後(到達soft limit)，會送出SIGXCPU的訊號，此訊號的預設動作為結束該行程。如果到達hard limit，則會送出SIGKILL訊號
- RLIMIT_DATA
  - 限制資料區段的大小(包含已初始，未初始與heap)
- RLIMIT_FSIZE
  - 限制行程能建立的檔案長度，如果超過會送出SIGXFSZ訊號
- RLIMIT_LOCKS
  - 限制行程能建立的lock數量
- RLIMIT_MEMLOCK
  - 限制行程可以lock(mlock函式)多少位元組的記憶體
- RLIMIT_NOFILE
  - 限制行程的最大開檔數
- RLIMIT_NPROC
  - 限制該使用者最多可以建立多少行程
- RLIMIT_RSS
  - 設定行程最多有頁(pages)可以放在記憶體，當記憶體吃緊時，kernel會將超出的部分swap出去
- RLIMIT_STACK
  - 設定行程堆疊最大為多少位元組

取得行程ID

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

    pid_t getpid(void);
    pid_t getppid(void);
```

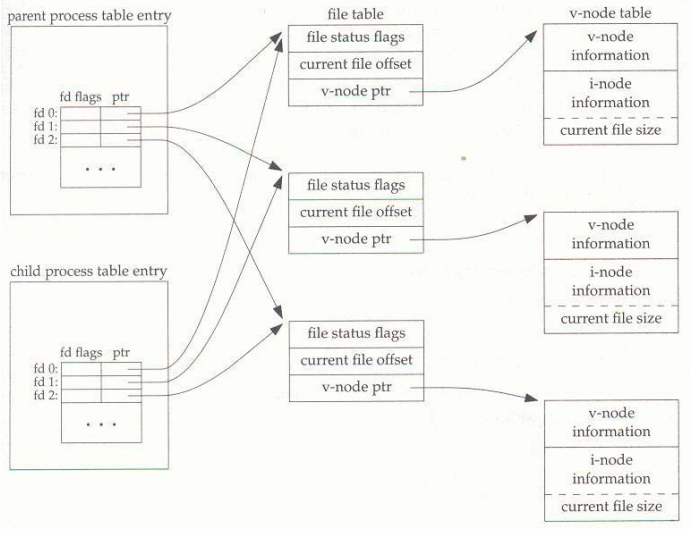
- 每個行程都有一唯一的id，稱為pid。每一個行程都有一個父行程，其父行程的pid稱為ppid。父行程是用來產生這些子行程
- 系統中的第一個行程為init，其pid為1，是由kernel所建立。接下來的行程則由init所產生(init會讀取/etc/rc.d下的設定)
- getpid函式可以取得目前行程的pid
- getppid函式可以取得父行程的pid

建立子行程(1/2)

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

      pid_t fork(void);
```

- 當一行程必須建立另一新行程時，可以使用fork函式，例如A行程建立B行程，則B為A的子行程，而A為B的父行程
- fork函式會拷貝父行程的資料(包掛資料區段，堆疊等等)，因此兩個行程互相獨立，但是子行程會繼承父行程的一些資源，包含如下(只列出一些)
  - file descriptor (已開啟的檔案)
  - resource limit
  - uid, gid, euid, egid
  - process group id, session ID, controlling terminal
  - Current working directory
- fork函式會有兩個返回值，一個返回父行程，另一個返回子行程(換言之從fork之後的程式就分成兩個行程在執行)，父行程的返回值為非零的整數，其值為子行程的pid，而子行程的返回值為0，因此透過返回值就可以知道是在子行程還是父行程中

建立子行程(2/2)



行程結束

- 行程結束的5種方式
  - 正常結束(normal termination)
    - 由main函式中return (相當於 exit(main(argv, argv));)
    - 呼叫 exit ，這個函式由ANSI C所定義
    - 呼叫 _exit，這個函式由POSIX.1所定義
  - 異常結束(abnormal termination)
    - 呼叫abort，這個函式會發出SIGABRT訊號
    - 被signal 結束。這些signal可能是由其它行程傳來或是自己發出(例如呼叫abort)
- 無論是哪一種，最後都會執行到相同的kernel程式。kernel程式會關閉該行程所有開啟的file descriptor，釋放行程所佔據的記憶體等等
- 子行程的exit status會傳遞給父行程
  - 如果父行程比子行程早結束，所有該行程的子行程會將其父行程設為1，也就是init
  - 如果子行程比父行程早結束，kernel會將子行程結束的狀態儲存起來，等父行程來讀取(呼叫wait或waitpid函式)
  - 如果父行程沒有透過wait相關指令讀取子行程的結束狀態，該子行程成為zombie
- 當行程結束時，kernel會發出一個SIGCHLD的訊號給該行程的父行程

等待子行程結束(1/3)

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

    pid_t wait(int *status);
    pid_t waitpid(pid_t pid, int *status, int options);
```

- 此兩個函式是用來等待子行程的結束(無論是正常結束或異常結束)
- wait函式會block住，直到有任一子行程結束，status為子行程的exit status
- waitpid函式比wait函式要強大，可以指定要等待哪一個(些)行程，而且可以指定是否要block或是否支援job control
  - pid可以有下列值
    - < -1 等待|pid|所代表的process group中的行程
    - -1 等待任何子行程，與wait相同
    - 0 等待與該行程相同process group中的行程
    - >0 等待該單一行程
  - options可以有下列值(可以or起來)
    - WNOHANG 代表不block
    - WUNTRACED 當有子行程stop時也返回

等待子行程結束(2/3)

- exit status為exit或_exit函式所指定，會被kernel轉換成termination status(exit status=termination status & 0377)，父行程可以透過wait或waitpid函式來取得這些資訊
- 父行程可以用下列幾種巨集來檢視termination status
  - WIFEXITED(status) 如果非零(true)代表正常結束
  - WEXITSTATUS(status) 取得exit status，也就是最小的8bits
  - WIFSIGNALED(status) 如果非零代表因某個signal沒有catch而導致結束
  - WTERMSIG(status) 取得造成行程結束的signal編號
  - WIFSTOPPED(status) 如果非零代表子行程已被stop
  - WSTOPSIG(status) 取得造成行程stop的signal編號
  - WCOREDUMP(status) 如果非零代表該行程有產生core dump

等待行程結束(3/3)

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>

    pid_t wait3(int *status, int options, struct rusage *rusage);
    pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage);
```

- 這兩個函式與waitpid相似，其中wait3等待任一子行程結束，而wait4可以指定等待那個(些)子行程結束。其中pid與options參數設定方式皆與waitpid相同
- waitpid函式只能取得子行程的termination status，而wait3與wait4函式除了這點外，還可以取得子行程所消耗的資源統計(參考getrusage函式)

競爭情況（race condition)

- 當多個行程同時處理某個共享的資源時，其結果會因執行的順序而產生錯誤稱為競爭情況
- 當fork一個行程時，我們沒有辦法預知父行程與子行程的執行順序，因此我們必須在程式中加入同步機制(可以透過IPC來實作)

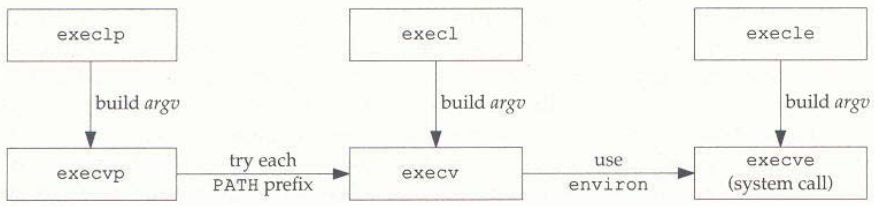
執行程式(1/2)

```
#include <unistd.h>

    int execl(const char *path, const char *arg, ...);
    int execlp(const char *file, const char *arg, ...);
    int execle(const  char  *path,  const  char  *arg  , ..., char * const envp[]);
    int execve(const  char  *filename,  char  *const  argv [], char *const envp[]);
    int execv(const char *path, char *const argv[]);
    int execvp(const char *file, char *const argv[]);
```

- 這六個函式都用來執行程式，其中字母l代表list，字母v代表vector，字母e代表環境變數，字母p代表會搜尋PATH路徑
- 這些函式執行後會取代原有的行程(包含text segment，data segment等等)，但是其pid不會改變，此外它會繼承原有行程的下列資源(只列出部分)
  - pid與ppid
  - uid與gid
  - file locks
  - pending signals
  - resource limits
  - tms_utime, tms_stime, tms_cutime和tms_ustime

執行程式(2/2)

Function	*pathname*	*filename*	Arg list	*argv[]*	environ	*envp[]*
execl	•		•		•	
execlp		•	•		•	
execle	•		•			•
execv	•			•	•	
execvp		•		•	•	
execve	•			•		•
(letter in name)		p	l	v		e



vfork與fork的比較

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

    pid_t vfork(void);
```

- vfork與fork相似，但是子行程與父行程在execve前共用相同的記憶體空間，此外父行程在vfork後會被block住，直到子行程呼叫execve等函示
- 適用於fork之後馬上execve的流程，如果用fork函式，父行程必須拷貝一份給子行程，然後馬上被execve給置換，因此拷貝的動作是多餘的，這時可以用vfork，增加效率。
  - 目前linux採用copy-on-write，也就是有改變才會拷貝新的，如果都沒有改變則共用相同的部分，可以降低fork的負擔
- 如果使用vfork但沒有使用execve，則子行程離開時必須使用_exit，如果使用exit會將父行程的標準I/O一起關閉

更改user id與group id

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

    int setuid(uid_t uid);
    int setgid(gid_t gid);
    int seteuid(uid_t euid);
    int setegid(gid_t egid);
    int setreuid(uid_t ruid, uid_t euid);
    int setregid(gid_t rgid, gid_t egid);
```

- setuid(setgid)函式會改變uid(gid)與euid(egid)兩個值
  - 只有root可以改變成任意的uid(gid)
  - 一般使用者只能設定成real user id(real group id)
- seteuid(setegid)函式只會改變effective uid (effective gid)
- setreuid(setregid)函式可以分別設定uid(gid)與euid(egid)兩個值

執行shell命令

```
#include <stdlib.h>

    int system(const char *string);
```

- 這個函式相當於fork，execve，waitpid三個函式的合成。簡單來說，這個函式會呼叫sh –c來執行string中所描述的命令，例如：system(“/bin/cat /etc/passwd > /tmp/test”);
- 由於有waitpid，因此system會執行完才返回，其返回值就為該命令的termination status
- 由於是sh的命令，所以可以<，>，|等redirect的符號

行程資源使用統計(1/2)

```
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>
      int getrusage(int who, struct rusage *usage);
```

```
struct rusage {
        struct timeval ru_utime; /* user time used */
        struct timeval ru_stime; /* system time used */
        long   ru_maxrss;        /* maximum resident set size */
        long   ru_ixrss;        /* integral shared memory size */
        long   ru_idrss;        /* integral unshared data size */
        long   ru_isrss;        /* integral unshared stack size */
        long   ru_minflt;       /* page reclaims */
        long   ru_majflt;       /* page faults */
        long   ru_nswap;        /* swaps */
        long   ru_inblock;      /* block input operations */
        long   ru_oublock;      /* block output operations */
        long   ru_msgsnd;       /* messages sent */
        long   ru_msgrcv;       /* messages received */
        long   ru_nsignals;     /* signals received */
        long   ru_nvcsw;        /* voluntary context switches */
        long   ru_nivcsw;       /* involuntary context switches */
};
```

Linux目前只支援紅字部分

行程資源使用統計(2/2)

- 可以使用getrusage函式取的行程的資源使用情況，此函數有兩個參數
  - who
    - RUSAGE_SELF 代表自己
    - RUSAGE_CHILDREN 代表已經結束的子行程
  - usage
    - 指向用來存放結果的空間

取得使用者身份

```
#include <unistd.h>

      char *getlogin(void);

#include <stdio.h>

      char *cuserid(char *string);
```

- getlogin函式用來取得login的名稱，特別注意傳回值為指向該函式內部static變數，因此下次呼叫就會把上次呼的結果給覆蓋掉
  - 也可以用getpwuid(getuid())取得passwd entry，但是如果相同的uid有不同的login名稱時，就只能用getlogin
  - 也可以用環境變數LOGNAME得到login名稱，但這個值可能被使用者更改
- cuserid函式用來取得effective user id的名稱，string為caller所配置空間的指標

行程所花的時間

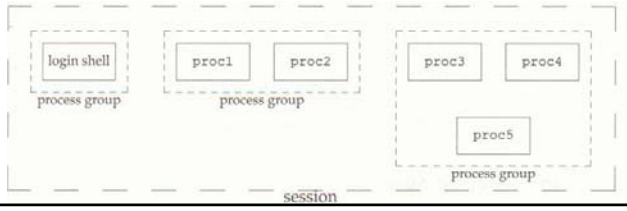
```
#include <sys/times.h>

      clock_t times(struct tms *buf);
```

```
struct tms {
        clock_t tms_utime;  /* user time */
        clock_t tms_stime;  /* system time */
        clock_t tms_cutime; /* user time of children */
        clock_t tms_cstime; /* system time of children */
};
```

- 可以用times函式取得行程所花的時間，其傳回值為某個時間點到現在所經過的clock ticks數。一秒有多少個clock ticks記錄在_SC_CLK_TCK中，通常其值為100，可以用sysconf(_SC_CLK_TCK)來取得該值
- tms結構中的成員，其單位也為clock ticks，其中tms_utime代表該行程在user space所花的時間，tms_stime代表該行程在kernel space所花的時間，tms_cutime代表該行程之已結束子行程在user space所花的時間，tms_cstime代表已結束子行程在kernel space所花的時間
- 因此要知道行程執行所經過的時間可以在行程的開頭與結尾處放執行兩個times，然後將兩個時間相減即為經過的時間。要知道行程所耗費的CPU時間則可以看結尾處times所設定的tms結構

行程間的關係

- 行程除了父子關係之外，還有其它的關係，像是行程屬與哪一個session，行程屬於哪一個process group等等
  - 例如在shell下打 a | b | c | d 執行，會產生4個行程，其父行程皆為shell。如果執行到一半按ctrl+C(結束執行)，則應該送signal給這四個行程，因此如果我們可以將此4個行程設定成同一個process group，則我們只需送signal給這個process group即可
  - 當我們登入一台主機後，我們可以執行許多的命令，有些是背景執行，但是我們一登出這些命列都會全部結束，這是因為這些指令都在同一個session中，登出會將該session中的所有process group給關閉
- 一個session中可以包含許多的process group
- 可以用指令ps -j 來觀看



設定與取得屬於那個process group

```
#include <unistd.h>

    int setpgid(pid_t pid, pid_t pgid);        //可以指定設定那個行程
    pid_t getpgid(pid_t pid);                  //可以指定取得那個行程
    int setpgrp(void);                         //設定目前行程
    pid_t getpgrp(void);                       //取得目前行程
```

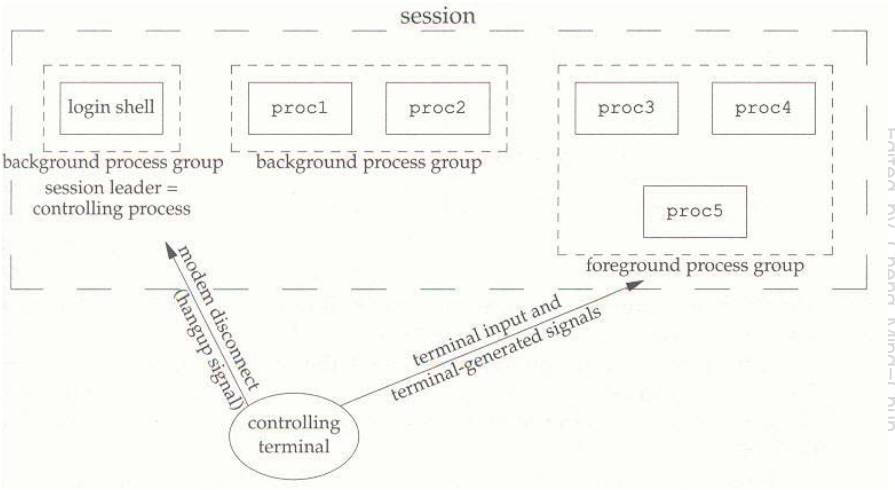
- 每一個process group擁有一個唯一的id，如同pid一般，稱為pgid
- 每一個process group包含一個以上的行程，這些行程中可以有一個是process group leader，其pid與pgid相同
- process group的生命週期與process group leader是否已結束沒有關係，當其沒有包含任何行程時，會自動被關閉
- 可以用setpgid函式來加入或建立一個新的process group，這些process group必須在同一個session裡
  - 只能設定自己或子行程的pgid
  - 當子行程執行execve後就不能設定pgid
  - 當要建立新的process group可以用setpgid(0,0); 或是setpgrp();
- 可以用getpgrp函式取得目前所在的process group

建立一個新的session

```
#include <unistd.h>

    pid_t setsid(void);       //開啓新的session
    pid_t getsid(pid_t pid); //取得session
```

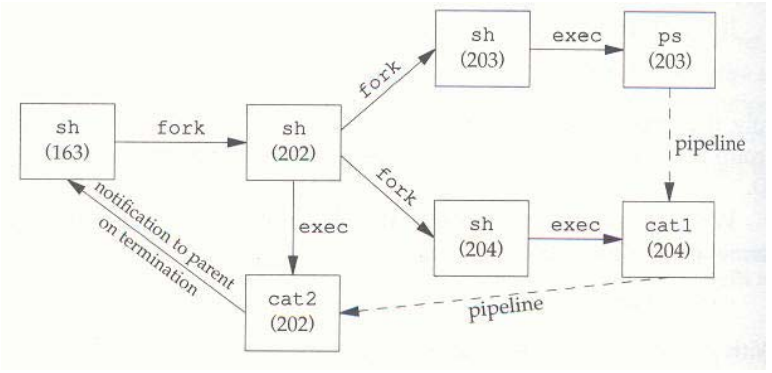
- 一個session可以包含多個process group
- 如果呼叫setsid函式的行程不是process group leader，則會產生新的session
  - 該行程成為該session leader
  - 會自動產生一個process group，而該process為process group leader
  - 該行程沒有controlling terminal
- getsid函式用來取得session id

Controlling terminal



一個shell命令執行範例

- ps –xj | cat1 | cat2



執行緒(Thread)

- 執行緒又稱為lightweight process，它與process不同處在於同一個process中的thread共用相同的記憶體空間。在thread間切換也比在process間切換來的快速
- Thread可以分成三類
  - User space thread
    這類thread在user space執行，通常以library的方式存在，kernel本身不知道user space thread的存在，因此kernel還是以process為排程單位
  - Kernel space thread
    這類thread在kernel space執行，kernel可以直接以它作為排程單位，由於切換需要進入kernel，所以其速度比user therad稍慢
  - Hybrid
    前兩者的混合體

哪時候需要使用執行緒

- I/O動作會block住直到完成，此時可以用thread來做其他事
- 使用者介面，當使用者點選某個功能後，可以用thread來處理該動作
- 同一個行程中需要同時跑許多函式時，例如網路程式
- 等等...

clone Function

```
#include <sched.h>

    int clone(int (*fn)(void *), void *child_stack, int flags, void *arg);
```

- Linux特有的函式，clone函式與fork函式相同，都可以用來建立新的行程，但它主要是用來建立執行緒
  - fn為函式指標，新的行程會執行該函式
  - child_stack為新的行程所使用的stack位址
  - flag用來指定建立行程的方式(可以or起來使用)
    - CLONE_VM 如果設定，子行程與父行程共用相同的記憶體空間
    - CLOME_FS 如果設定，子行程與父行程共用相同的檔案系統資訊，例如子行程呼叫chroot，chdir，umask就會影響父行程
    - CLONE_FILES 如果設定，子行程與父行程共用相同的file descriptor，如果不設定，子行程還是會繼承父行程的file descriptor，但是read，write，close這些file descriptor將不會影響父行程
    - CLONE_PID 如果設定，子行程與父行程的PID將相同
  - arg為傳給fn的參數
- 不要直接用clone函式來建立執行序，而是用pthread_create來建立

Pthread Interface

- Pthread為POSIX所定義的thread標準界面，它目前已經被移植到許多平台上，因此使用pthread會有較好的可移植性。接下來的內容將主要介紹如何使用pthread
- 在linux中pthread是透過clone函式來產生thread，因此為kernel space thread

執行緒的建立與結束

```
#include <pthread.h>

    int  pthread_create(pthread_t  *  thread, pthread_attr_t * attr, void *
                                (*start_routine)(void *), void * arg);
    void pthread_exit(void *retval);
```

- pthread_create函式用來建立新的執行緒，該執行緒的識別會儲存在thread所指的空間上，attr用來設定建立的屬性為何(可以是null pointer代表預設值)，可以透過pthread_attr_*函式來設定，start_routine為該執行產生後所會執行函式的函式指標，最後的arg為傳遞給start_routine的參數
- 執行緒要結束有兩種情況
  - 呼叫pthread_exit，此函式會執行先前由pthread_cleanup_push所註冊的cleanup handler
  - 從start_routine返回

等待執行緒結束

```
#include <pthread.h>

    int pthread_join(pthread_t th, void **thread_return);
    int pthread_detach(pthread_t th);
```

- pthread_join這個函式跟wait相似，只是它是用來等待某個執行序結束，呼叫它的執行緒將會被block住，直到等待的執行緒結束後才能繼續執行
  - 等待執行緒th的返回值會儲存在thread_return所指的空間上
  - 等待執行緒th的狀態必須為joinable，而且其狀態不能為detached
  - 執行緒所使用的記憶體資源並不會釋放，直到被某個執行緒所join
  - 一個執行緒只能被一個執行緒所等待(join)
- pthread_detach函式用來將執行中的執行緒狀態改成detached，狀態為detached的執行緒在執行完後會自動釋放記憶體
  - 在pthread_create時可以透過attr的設定，直接讓產生的執行緒其狀態為detached

註冊新執行緒產生時所執行的handler

```
#include <pthread.h>

    int  pthread_atfork(void  (*prepare)(void),  void (*parent)(void), void
                                    (*child)(void));
```

- 這個函式用來註冊三種handler，執行時機如下：
  - prepare函式在執行緒產生前會被呼叫
  - parent函式會在舊的執行緒呼叫
  - child函式會在新的執行緒呼叫
  - 這些參數可以為NULL
- 可以執行許多次pthread_atfork來註冊一個以上的handler
- 這個函式可以用來清除新執行緒中所複製的mutexes

取消執行緒的執行

```
#include <pthread.h>
    int pthread_cancel(pthread_t thread);
    int pthread_setcancelstate(int state, int *oldstate);
    int pthread_setcanceltype(int type, int *oldtype);
    void pthread_testcancel(void);
```

- pthread_cancel函式用來取消一個執行緒的執行
- phtead_setcancelstate函式用來設定該執行緒是否可以cancel，其中state為新的設定值，oldstate會儲存舊的值。設定值有兩種
  - PTHREAD_CANCEL_ENABLE 代表可以CANCEL
  - PTHREAD_CANCEL_DISABLE 代表不可以CANCEL
- pthread_setcanceltype函式來用設定當遇到cancel時要如何處理，其中type為新的設定值，有下列兩種
  - PTHREAD_CANCEL_ASYNCHRONOUS 代表遇到cancel馬上取消
  - PTHREAD_CANCEL_DEFERRED 代表遇到cancel不馬上取消，等到cancel point才取消，cancel point有下列幾個
    - pthread_join(3)
    - pthread_cond_wait(3)
    - pthread_cond_timedwait(3)
    - pthread_testcancel(3)
    - sem_wait(3)
    - sigwait(3)
- 執行緒的預設值為PTHREAD_CANCEL_ENABLE 與PTHREAD_CANCEL_DEFERRED
- pthread_testcancel函式用來測試是否有cancel

註冊cleanup handler

```
#include <pthread.h>

    void pthread_cleanup_push(void (*routine) (void *), void *arg);
    void pthread_cleanup_pop(int execute);
```

- 這些註冊的handler在執行緒呼叫pthread_exit結束或是發生cancel時會執行，這些handler是以stack的方式來擺放
- pthread_cleanup_push函式用來註冊新的handler(push到堆疊頂端)，而pthread_cleanup_pop函式是用來取消最後一個註冊的handler(pop出堆疊頂端)。這兩個巨集的配對必須在同一個函式中，而且必須在同一個block level
- 這些handler用來釋放資源，例如mutexes

Pthread Conditions

```
#include <pthread.h>
      pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
      int pthread_cond_init(pthread_cond_t   *cond,   pthread_condattr_t
                                  *cond_attr);
      int pthread_cond_signal(pthread_cond_t *cond);
      int pthread_cond_broadcast(pthread_cond_t *cond);
      int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
      int pthread_cond_timedwait(pthread_cond_t   *cond,   pthread_mutex_t
                                  *mutex, const struct timespec *abstime);
      int pthread_cond_destroy(pthread_cond_t *cond);
```

- 這些函式的用途為暫停某個執行緒的執行，直到某個condition variable符合為止
- 可以用PTHREAD_COND_INITIALIZER來產生一個condition variable或是用pthread_cond_init函式來產生，在linux中，cond_attr被忽略
- pthread_cond_destroy函式用來消滅condition variable
- pthread_cond_signal用來叫醒等待在該condition variable執行緒的其中之一
- pthread_cond_broadcast用來叫醒等待在該condition variable的所有執行緒
- pthread_cond_wait用來等待某個condition variable，這個函式會自動unlock mutex，等到該執行緒可以執行時，會自動重新取得mutex的lock。pthread_cond_timewait與pthread_cond_wait相似，只是它只等待某一段時間

判斷是否為相同的thread

```
#include <pthread.h>

      int pthread_equal(pthread_t thread1, pthread_t thread2);
```

- 由於POSIX並沒有規定如何實作pthread_t，因此它的實作方式可能有許多種，例如可以用一個structure來實作他，此時就不能用 if(thread1==thread2) 這種寫法
- 這個函式的目的是用來比較兩個執行緒是否相同
- 傳回非零值(true)代表thread1與thread2相同

設定執行緒的屬性(1/2)

Attribute	*Value*	*Meaning*
detachstate	PTHREAD_CREATE_JOINABLE*	Joinable state
	PTHREAD_CREATE_DETACHED	Detached state
schedpolicy	SCHED_OTHER*	Normal, not real-time
	SCHED_RR	Real-time, round-robin
	SCHED_FIFO	Real-time, first in, first out
schedparam	policy specific	
inheritsched	PTHREAD_EXPLICIT_SCHED*	Set by schedpolicy and schedparam; inherited from parent process
	PTHREAD_INHERIT_SCHED	
scope	PTHREAD_SCOPE_SYSTEM*	One system timeslice for each thread; threads share the same system timeslice (not supported under Linux)
	PTHREAD_SCOPE_PROCESS	

設定執行緒的屬性(2/2)

```
#include <pthread.h>

    int pthread_attr_init(pthread_attr_t *attr);
    int pthread_attr_destroy(pthread_attr_t *attr);
    int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
    int  pthread_attr_getdetachstate(const   pthread_attr_t   *attr,   int
                                *detachstate);
    int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
    int pthread_attr_getschedpolicy(const pthread_attr_t *attr,  int  *policy);
    int  pthread_attr_setschedparam(pthread_attr_t   *attr,  const  struct
                                      sched_param *param);
    int pthread_attr_getschedparam(const  pthread_attr_t   *attr,
                                  struct sched_param *param);
    int pthread_attr_setinheritsched(pthread_attr_t *attr, int inherit);
    int pthread_attr_getinheritsched(const   pthread_attr_t   *attr,  int *inherit);
    int pthread_attr_setscope(pthread_attr_t *attr, int scope);
    int pthread_attr_getscope(const pthread_attr_t *attr, int *scope);
```

請 man 3 pthread_attr_init

mutex：mutual exclusion (1/2)

- Mutex為lock的一種，也可以說是binary semaphore。用來保護資料同時被多個執行緒修改
- Mutex只有兩種狀態，locked與unlocked
- 每個mutex只能被一個執行緒所擁有(也就是取得該mutex的lock)，其它執行緒如果也嘗試取得該mutex的lock，則該執行緒會被block住，直到原擁有mutex的執行緒將mutex釋放出來

mutex：mutual exclusion (2/2)

```
#include <pthread.h>
    pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;
    pthread_mutex_t recmutex = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
    pthread_mutex_t errchkmutex = PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
    int pthread_mutex_init(pthread_mutex_t  *mutex,  const  pthread_mutexattr_t *mutexattr);
    int pthread_mutex_lock(pthread_mutex_t *mutex);
    int pthread_mutex_trylock(pthread_mutex_t *mutex);
    int pthread_mutex_unlock(pthread_mutex_t *mutex);
    int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- 可以用前三個常數，或pthread_mutex_init函式來產生mutex。其中的差異如下：
  - PTHREAD_MUTEXT_INITIALIZER，當執行緒無法取得該mutex的lock時(因為已經被其它執行緒給lock)，該執行緒會block住
  - PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP，無論是否被lock住，pthread_mutex_lock都會成功，然後將mutex的lock數加1，必須有等數量的pthread_mutex_unlock才能將其狀態unlock
  - PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP，當執行緒無法取得該mutex的lock時(因為已經被其它執行緒給lock)，會傳回錯誤值
- pthread_mutex_lock用來取得mutex的lock，其執行結果會因為mutex種類而有不同
- pthread_mutex_trylock與pthread_mutex_lock相似，只是不會block住，用來測試該mutex是否已經被lock
- pthread_mutex_unlock用來釋放lock，只有擁有該lock的執行緒可以執行此函式
- pthread_mutex_destroy用來消滅mutex所佔據的資源

Semaphore(1/2)

- 為什麼需要semaphore
  - 因為mutex只能為處理互斥(binary semaphore)的情況，不能處理共享的情況
  - Mutex在signal時，可能產生不公平的情況(可能造成飢餓)
- Semaphore為一資源計數器，當要求資源時，該計數器會減1，當計數器為零時，如果有另外的執行緒要求該資源，則該執行緒會被block住。當使用完該資源後，釋放該資源會讓計數器加1
  - 一個Semaphore有兩種屬性
    - 計數器：用來代表有多少個執行緒可以同時共用該資源
    - 等待執行緒列表：用來紀錄有哪些執行緒被block住，其用來保證FIFO的順序

Semaphore(2/2)

```
#include <semaphore.h>
    int sem_init(sem_t *sem, int pshared, unsigned int value);
    int sem_wait(sem_t * sem);
    int sem_trywait(sem_t * sem);
    int sem_post(sem_t * sem);
    int sem_getvalue(sem_t * sem, int * sval);
    int sem_destroy(sem_t * sem);
```

- sem_init函式用來建立semaphore variable，目前linux中只支援local的seamphore(也就是pshared為0)，value代表seamphore的初始計數器
- sem_wait函式用來取得semaphore，如果該semaphore的計數器>0，則可以繼續往下執行，其計數器會減一，如果<0，則該執行緒會被加入等待執行緒列表中。sem_trywait函式與sem_wait相似，不過它只是用來測試計數器是否>0
- sem_post函式用來釋放資源，如果等待執行緒列表有執行緒存在，則叫醒它。此函式會將計數器加一
- sem_getvalue函式用來取得計數器目前的值
- sem_destroy函式用來消滅semaphore

每個執行緒的獨立空間(TSD)

```
#include <pthread.h>
    int pthread_key_create(pthread_key_t *key, void (*destr_function) (void *));
    int pthread_key_delete(pthread_key_t key);
    int pthread_setspecific(pthread_key_t key, const void *pointer);
    void * pthread_getspecific(pthread_key_t key);
```

- 同一個行程中的執行緒使用相同的空間，如果希望每個執行緒擁有自己特定的區塊(也就是不共用)，可以使用這頁的函式。我們將這個thread specific data簡稱為TSD
- pthread_key_create函式用來建立TSD的key，destr_function為一函式指標，當執行緒結束時，該函式會被呼叫，用來釋放佔據的資源
- pthread_key_delete函式用來釋放該TSD所佔據的記憶體
- pthread_key_setspecific函式用來將pointer所指的資料寫入key所對應的區塊內
- pthread_key_getspecific函式用來將key所對應區塊的資料讀出來