34

行程群組（process group）、作業階段（session）和工作控制（job control）

行程群組（process group）和作業階段（session）在行程之間是兩層的階層關係：行程群組是一組相關行程的集合，而作業階段是一組相關行程群組的集合。讀者透過本章的學習就能釐清「相關」的含義。

行程群組和作業階段是為支援 shell 工作控制而定義的抽象概念，使用者透過 shell 能夠互動式地在前景（foreground）或背景（background）執行指令。「工作（*job*）」一詞經常與「行程群組」做為同義詞使用。

本章將介紹行程群組、作業階段和工作控制。

34.1 概觀

行程群組由一個或多個使用相同行程群組 ID（*process group identifier*，PGID）的行程組成，行程群組 ID 是一個數字，其型別與行程 ID 相同（*pid_t*），一個行程群組會有一個行程群組的組長（*process group leader*），組長行程是群組的第一個行程，其行程 ID 會成為行程群組的 ID，新的行程則會繼承其父行程所屬的行程群組 ID。

行程群組會有生命週期（*lifetime*），其時間週期始於行程組長加入群組時，結束時間為最後一個成員行程離開群組時。一個行程可能會因為終止而離開行程群組，或因為加入了另一個行程群組而結束行程群組，行程群組的組長不必是最後一個離開行程群組的成員。

作業階段（*session*）是一個行程群組的集合，行程的作業階段成員關係是由其作業階段 ID（*session identifier*，SID）決定的，作業階段 ID 與行程群組 ID 的型別都是 *pid_t* 數值，作業階段組長是建立新作業階段的行程，其行程 ID 會成為作業階段 ID，新行程會繼承其父行程的作業階段 ID。

一個作業階段中的每個行程都會共用一個控制終端機（*controlling terminal*），控制終端機會在作業階段組長（*session leader*）初次開啟一個終端設備時建立，一個終端機最多只能成為一個作業階段的控制終端機。

在任何時間點，作業階段中的其中一個行程群組會成為終端機的前景行程群組（*foreground process group*），其他行程群組會成為背景行程群組（*background process group*）。只有前景行程群組中的行程從讀取控制終端機讀取輸入。當使用者在控制終端機輸入其中一個訊號生成的（*signal-generating*）終端機字元之後，該訊號會被發送到前景行程群組中的每個成員。這些字元包括，產生 SIGINT 的中斷字元（通常是 *Control-C*）、產生 SIGQUIT 的結束字元（通常是 *Control-*）、產生 SIGSTP 的暫停字元（通常是 *Control-Z*）。

當與控制終端機建立連接（即開啟）之後，作業階段組長會成為終端機的控制行程（*controlling process*），成為控制行程的目的是，若終端機斷開連接時，核心（kernel）會向該行程發送一個 SIGHUP 訊號。

> 透過檢測 Linux 特有的 /proc/*PID*/stat 檔案，就能確定任意行程的行程群組 ID 和作業階段 ID。此外，還能確定行程的控制終端機裝置 ID（一個十進位整數，包含 major ID 與 minor ID）和控制該終端機的控制行程（行程 ID），更多細節資訊請參考 *proc(5)* 手冊。

作業階段和行程群組的主要用途是用於 shell 工作控制，透過探討一個特定範例有助於釐清這些概念，如對於互動式登入，控制終端機是使用者登入的途徑，登入的 shell 會變成作業階段組長以及終端機的控制行程，也是其自身行程群組唯一成員。從 shell 啟動的每個指令或用管線串接的指令，都會導致建立一個或多個行程，而且 shell 會把這些行程全部放到一個新的行程群組。（這些行程在一開始是其行程群組中的唯一成員，它們建立的每個子行程會成為該群組中的成員）。當指令或以管線連接的一串指令是以 & 符號結束時，則會在背景行程群組執行這些

指令，否則就會在前景行程群組執行這些指令，在登入作業階段中建立的每個行程，都會成為同一個作業階段的一份子。

> 在視窗環境中，控制終端機是一個虛擬終端機（pseudo terminal），每個終端機視窗都有一個獨立的作業階段，隨著視窗的啟動，shell 會成為終端機的作業階段組長與控制行程。
>
> 有時可以在工作控制以外的其他區域使用行程群組，因為行程群組具備兩個有用的屬性：即在特定的行程群組中，父行程能夠等待任何一個子行程（參考 26.1.2 節），以及能將訊號發送給行程群組中的每個成員（參考 20.5 節）。

圖 34.1 說明執行下列指令之後，各個行程之間的行程群組和作業階段關係：

```
$ echo $$                              Display the PID of the shell
400
$ find / 2> /dev/null | wc -l &        Creates 2 processes in background group
[1] 659
$ sort < longlist | uniq -c            Creates 2 processes in foreground group
```

此時，shell（*bash*）、*find*、*wc*、*sort* 與 *uniq* 都正在執行。

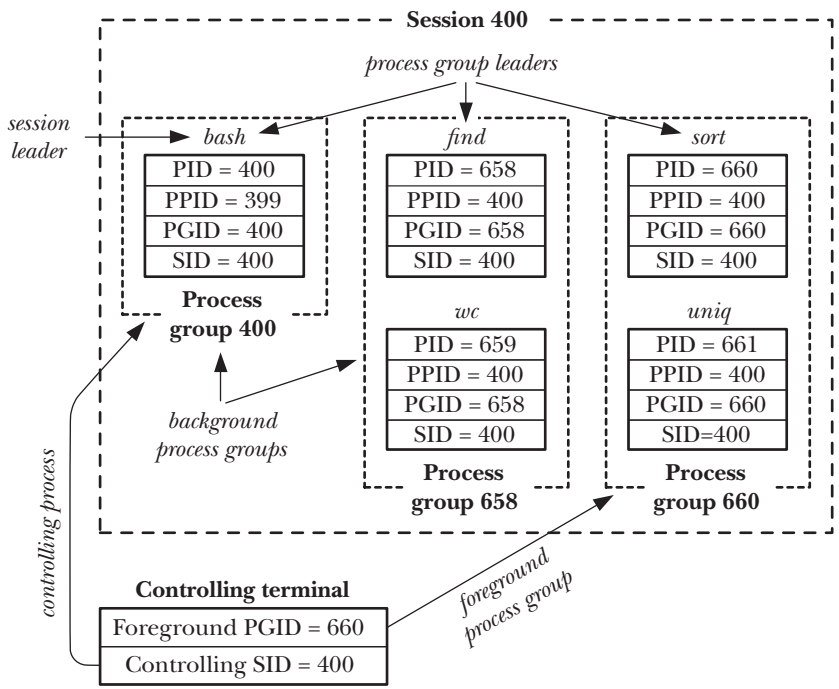

圖 34.1：行程群組、作業階段和控制終端機之間的關係

34.2 行程群組（process group）

每個行程都有一個以數值表示的行程群組 ID，表示該行程所屬的行程群組，新行程會繼承父行程的行程群組 ID，使用 *getpgrp()* 能夠取得一個行程的行程群組 ID。

```
#include <unistd.h>

pid_t getpgrp(void);
                Always successfully returns process group ID of calling process
```

若 *getpgrp()* 的回傳值與呼叫者的行程 ID 匹配，則代表此行程是行程群組組長。

我們執行 *setpgid()* 系統呼叫，可以將 *pid* 行程（行程 ID 為 *pid*）的行程群組 ID 修改為 *pgid*。

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
                                        Returns 0 on success, or –1 on error
```

若將 *pid* 指定為 0，則呼叫行程（calling process）的行程群組 ID 會被改變，若將 *pgid* 的值設定為 0，則行程群組 ID 為 *pid* 的行程，其行程群組 ID 會改成與行程 ID 相同。因此，下列的 *setpgid()* 呼叫是相等的：

```
setpgid(0, 0);
setpgid(getpid(), 0);
setpgid(getpid(), getpid());
```

若 *setpgid()* 將目標行程（target process）的行程群組 ID（由 *pid* 指定的行程）更改為目標行程的行程 ID，則目標行程就會成為新行程群組的組長（此行程群組 ID 與組長的行程 ID 相同）。若 *setpgid()* 將目標行程的行程群組 ID 更改為其他值（與目標行程的行程 ID 不同的值），則目標行程會被移到 *pgid* 指定的現有行程群組。若 *setpgid()* 讓目標行程的行程群組 ID 保持不變，則該呼叫不會對目標行程造成影響。

通常 *setpgid()* 函式（以及 34.3 節介紹的 *setsid()*）的呼叫者是 shell 和 *login(1)*，我們在 37.2 節將會看到一個程式在成為 daemon 的過程中，也會呼叫 *setsid()*。

在呼叫 *setpgid()* 時，有以下限制：

- *pid* 參數只能指定為呼叫的行程或它的其中一個子行程，違反這條規則會導致 ESRCH 錯誤。
- 在群組之間移動行程時，呼叫的行程、*pid* 指定的行程（可能是另外一個行程，也可能就是呼叫行程），以及目標行程群組必須全部屬於同一個作業階段，違反這條規則會導致 EPERM 錯誤。
- *pid* 參數指定的行程不能是作業階段組長，違反這條規則會導致 EPERM 錯誤。
- 一個行程在其子行程已經執行 *exec()* 之後，不能改變子行程的行程群組 ID。違反這條規則會導致 EACCES 錯誤。之所以會有這條約束條件的原因是，若在一個行程開始執行之後，改變其行程群組 ID 會造成程式的混淆。

在工作控制的 shell 使用 *setpgid()*

一個行程在其子行程已經執行 *exec()* 之後，就無法修改該子行程的行程群組 ID，此限制條件會影響有下列需求的工作控制 shell 程式之設計：

- 一個工作（job）中的每個行程（即一個指令或一串以管線連接的指令）必須放置在一個單獨的行程群組。（可以觀察圖 34-1 中 *bash* 建立的兩個行程群組看到所需的結果）。此步驟允許 shell 使用 *killpg()*（或使用負的 *pid* 值來呼叫 *kill()*）同時向行程群組的所有成員發送工作控制訊號。一般而言，此步驟需要在發送任意工作控制訊號前完成。
- 每個子行程在執行（exec）程式之前，必須要先轉移到行程群組，因為程式本身不清楚如何操作行程群組 ID 的。

對於該工作的每個行程而言，父行程和子行程都可以使用 *setpgid()* 來修改子行程的行程群組 ID。然而，由於在父行程執行 *fork()*（參考 24.4 節）之後，父行程與子行程之間的排班順序是無法確定的，因此無法肯定父行程能在子行程執行 *exec()* 之前，改變子行程的行程群組 ID，同樣也無法確定子行程可以在父行程向其發送任何工作控制訊號之前，修改其行程群組 ID。（依賴這任一個行為都會導致競速情況）。因此，在設計工作控制 shell 程式時，需要讓父行程和子行程在 *fork()* 呼叫之後，立即呼叫 *setpgid()*，來將子行程的行程群組 ID 設定為同樣的值，而父行程會忽略在 *setpgid()* 呼叫中出現的任何 EACCES 錯誤。換句話說，在一個工作控制 shell 程式中，可能會出現如列表 34-1 的程式碼。

列表 34-1：工作控制 shell 程式如何設定子行程的行程群組 ID

```
pid_t childPid;
pid_t pipelinePgid;          /* PGID to which processes in a pipeline
                                are to be assigned */
```

```
/* Other code */

childPid = fork();
switch (childPid) {
case -1: /* fork() failed */
    /* Handle error */

case 0: /* Child */
    if (setpgid(0, pipelinePgid) == -1)
        /* Handle error */
    /* Child carries on to exec the required program */

default: /* Parent (shell) */
    if (setpgid(childPid, pipelinePgid) == -1 && errno != EACCES)
        /* Handle error */
    /* Parent carries on to do other things */
}
```

在處理由管線建立的行程時，事情會比列表 34-1 略為複雜，父輩的 shell 需要記錄管線中的第一個行程之行程 ID，並用來做為此行程群組中的每個行程之行程群組 ID（*pipelinePgid*）。

取得和修改行程群組 ID 的其他（過時的）介面

這裡說明為何 *getpgrp()* 和 *setpgid()* 兩個系統呼叫的名稱字尾不同。

起初，4.2BSD 提供一個 *getpgrp(pid)* 系統呼叫，用來傳回行程（行程 ID 為 *pid*）的行程群組 ID。在實務上，*pid* 幾乎總是用來表示執行呼叫的行程。結果，POSIX 委員會認為這個系統呼叫過於複雜，因此改成採用 System V 的 *getpgrp()* 系統呼叫，這個系統呼叫不接收任何參數，並傳回呼叫行程的行程群組 ID。

為了改變行程群組 ID，4.2BSD 提供 *setpgrp(pid, pgid)* 系統呼叫，它的運作與 *setpgid()* 類似。這兩個系統呼叫的主要差別在於，BSD *setpgrp()* 能夠用來將行程群組 ID 設定為任意值，（前面曾經提及，不能使用 *setpgid()* 將一個行程轉移至其他作業階段中的行程群組）。這會引起一些安全問題，但也讓實作工作控制程式較有彈性。結果，POSIX 委員會決定給這個函式增加額外的限制條件，並將其命名為 *setpgid()*。

更複雜的事情是，SUSv3 制定了一個 *getpgid(pid)* 系統呼叫，它的語意與舊有的 BSD *getpgrp()* 相同，而且也定義了一個從 System V 衍生而來的 *setpgrp()*，不接受任何參數，與 *setpgid(0, 0)* 呼叫幾乎是相等的。

雖然在實作 shell 工作控制上，利用前述的 *setpgid()* 和 *getpgrp()* 系統呼叫已經足夠了，Linux 但與其他大多數 UNIX 實作一樣，也提供了 *getpgid(pid)* 和 *setpgrp(void)*。為了向後相容，很多從 BSD 演化而來的實作仍然繼續提供 *setprgp(pid, pgid)*，它與 *setpgid(pid, pgid)* 是一樣的。

若我們在編譯程式時，明確定義 _BSD_SOURCE 功能測試巨集（feature test macro），則 *glibc* 會使用從 BSD 演化而來的 *setpgrp()* 和 *getpgrp()* 來取代預設版本。

34.3 作業階段（session）

一個作業階段是行程群組的一個集合，一個行程的作業階段成員關係是由其作業階段 ID 數值定義的。新的行程會繼承其父行程的作業階段 ID，*getsid()* 系統呼叫會傳回 *pid* 指定行程之作業階段 ID。

```
#define _XOPEN_SOURCE 500
#include <unistd.h>

pid_t getsid(pid_t pid);
                    Returns session ID of specified process, or (pid_t) –1 on error
```

若 *pid* 參數的值為 0，則 *getsid()* 會傳回呼叫行程的作業階段 ID。

> 在有些 UNIX 實作中（如 HP-UX 11），只有當呼叫行程與 pid 指定的行程屬於同一個作業階段時，才能使用 *getsid()* 來取得行程的作業階段 ID。（SUSv3 無此限制）。換句話說，只能透過此呼叫提供的結果為成功或失敗（EPERM）來判定指定的行程是否與呼叫行程屬於同一個作業階段，此限制在 Linux 與大多數其他系統並不存在。

若呼叫行程不是行程群組的組長，則 *setsid()* 會建立一個新的作業階段。

```
#include <unistd.h>

pid_t setsid(void);
                    Returns session ID of new session, or –1 on error
```

系統呼叫 *setsid()* 會依照下列步驟建立一個新的作業階段：

- 呼叫行程成為新作業階段的組長，以及作業階段中新行程群組的組長。呼叫行程的行程群組 ID 和作業階段 ID 會設定為該行程的行程 ID。
- 呼叫行程沒有控制終端機，之前與控制終端機的全部連接都會斷開。

若呼叫行程是一個行程群組的組長，則 *setsid()* 呼叫會回報 EPERM 錯誤，確保避免此錯誤發生的最簡易方式是，執行一個 *fork()*、讓父行程結束，以及讓子行程呼叫 *setsid()*。由於子行程會繼承父行程的行程群組 ID，並接收屬於自己的唯一行程 ID，因此它無法成為行程群組的組長。

限制行程群組的組長呼叫 *setsid()* 是有必要的，因為若沒有這個約束，行程群組的組長就能將自己轉移到另一個（新的）作業階段，而該行程群組的其他成員則仍然處於原來的作業階段。（不會建立一個新行程群組，因為根據定義，行程群組組長的行程群組 ID 已經與其行程 ID 相同）。這會破壞作業階段和行程群組之間的嚴格兩階層級，因此一個行程群組的每個成員必須屬於同一個作業階段。

> 當使用 *fork()* 建立一個新行程時，核心會確保它擁有唯一的行程 ID，而該行程 ID 不會與任何現有行程的行程群組 ID 和作業階段 ID 重複。這樣，即使行程群組或作業階段組長結束之後，新行程也無法再次使用組長的行程 ID，因而無法成為既有作業階段和行程群組的組長。

列表 34-2 示範如何使用 *setsid()* 建立一個新的作業階段，為了檢查行程已經不再擁有控制終端機，這個程式試著開啟 /dev/tty 特殊檔（下一節將介紹），當執行這個程式時會看到下列的結果：

```
$ ps -p $$ -o 'pid pgid sid command'              $$ is PID of shell
  PID PGID SID COMMAND
12243 12243 12243 bash                            PID, PGID, and SID of shell
$ ./t_setsid
$ PID=12352, PGID=12352, SID=12352
ERROR [ENXIO Device not configured] open /dev/tty
```

如輸出所示，行程成功地將其自身轉移到新的作業階段的新行程群組，由於這個作業階段沒有控制終端機，因此 *open()* 呼叫會失敗。（從上面程式輸出的倒數第二行中可以看出，shell 提示字元與程式輸出混在一起了，因為 shell 注意到，父行程在 *fork()* 呼叫之後就結束了，因此在子行程結束之前就印出下一個提示字元）。

列表 34-2：建立一個新的作業階段

pgsjc/t_setsid.c

```
#define _XOPEN_SOURCE 500
#include <unistd.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    if (fork() != 0)            /* Exit if parent, or on error */
```

```
            _exit(EXIT_SUCCESS);

        if (setsid() == -1)
            errExit("setsid");

        printf("PID=%ld, PGID=%ld, SID=%ld\n", (long) getpid(),
                (long) getpgrp(), (long) getsid(0));

        if (open("/dev/tty", O_RDWR) == -1)
            errExit("open /dev/tty");
        exit(EXIT_SUCCESS);
    }
```

pgsjc/t_setsid.c

34.4 控制終端機與控制行程

一個作業階段中的每個行程可能會有一個（單獨的）控制終端機，作業階段在剛建立時是沒有控制終端機的，除非在呼叫 *open()* 時指定 O_NOCTTY 旗標，否則在作業階段組長初次開啟一個終端機（尚未成為某個作業階段的控制終端機）時，就會建立控制終端機，一個終端機最多只能成為一個作業階段的控制終端機。

> SUSv3 定義了函式 *tcgetsid(int fd)*（定義在 <*termios*.h> 標頭檔），它傳回一個作業階段 ID（與 *fd* 指定的控制終端機相關聯），*glibc* 也有提供此函式（使用 *ioctl()* 搭配 TIOCGSID 操作實作）。

由 *fork()* 建立的子行程會繼承控制終端機，控制終端機也可以跨 *exec()* 呼叫而保留。

當作業階段組長開啟一個控制終端機之後，它同時成為終端機的控制行程。若終端機斷開了，則核心會向控制行程發送一個 SIGHUP 訊號，以通知此事件的發生，我們在 34.6.2 節中將會介紹更多有關這一方面的細節資訊。

若一個行程有一個控制終端機，則開啟 /dev/tty 特殊檔案就能取得終端機的檔案描述符。若標準輸入與標準輸出已經重新導向，而程式想要確定自己正在與控制終端機通信，則可以利用這個方式。例如，在 8.5 節介紹的 *getpass()* 函式會為此開啟 /dev/tty。若行程沒有控制終端機，則在開啟 /dev/tty 時會回報 ENXIO 的錯誤。

移除行程與控制終端機之間的關聯

使用 *ioctl(fd,* TIOCNOTTY*)* 操作能夠移除行程與控制終端機（檔案描述符 *fd* 指定的控制終端機）之間的關聯關係。在呼叫這個函式之後，再試圖開啟 /dev/tty 檔案就

會失敗。（雖然 SUSv3 沒有指定這個操作，但大多數 UNIX 實作都支援 TIOCNOTTY 操作）。

若呼叫行程是終端機的控制行程，則在控制行程終止時（參考 34.6.2），會發生下列步驟：

1. 作業階段中的所有行程將會失去與控制終端機之間的關聯關係。
2. 控制終端機失去了與該作業階段之間的關聯關係，因此另一個作業階段組長（session leader）就能夠取得該終端機，以成為控制終端機。
3. 核心會向前景行程群組的所有成員發送一個 SIGHUP 訊號（和一個 SIGCONT 訊號），來通知它們控制終端機的分離。

在 BSD 上建立一個控制終端機

SUSv3 對於作業階段取得控制終端機的方式保留不予規範，在開啟終端機時僅指定 O_NOCTTY 旗標，只能確保終端機不會成為作業階段的控制終端機。上述的 Linux 語意源自 System V 系統。

在 BSD 系統上，不管是否指定 O_NOCTTY 旗標，以作業階段組長開啟一個終端機，絕不會導致該終端機成為控制終端機。作業階段組長反而會需要使用 *ioctl()* TIOCSCTTY 操作，明確地將 *fd* 檔案描述符指定的終端機建立為控制終端機。

```
if (ioctl(fd, TIOCSCTTY, 0) == -1)
    errExit("ioctl");
```

若作業階段還沒有控制終端機，則可以執行這個操作。

Linux 系統上也有 TIOCSCTTY 操作，但在其他（非 BSD）系統上並未獲得廣泛使用。

取得表示控制終端機的路徑名稱：*ctermid()*

函式 *ctermid()* 傳回表示控制終端機的路徑名稱。

```
#include <stdio.h>              /* Defines L_ctermid constant */

char *ctermid(char *ttyname);
        Returns pointer to string containing pathname of controlling terminal,
                                    or NULL if pathname could not be determined
```

函式 *ctermid()* 以兩種不同的方式傳回控制終端機的路徑名稱：透過函式的傳回值與透過 ttyname 指向的緩衝區。

若 *ttyname* 不為 NULL，則它應該是一個大小至少為 L_ctermid 位元組的緩衝區，而路徑名稱則會被複製到這個陣列。在此，函式的回傳值也是一個指向此緩衝區的指標，若 *ttyname* 為 NULL，則 *ctermid()* 會傳回一個指標（指向靜態配置的緩衝區），緩衝區中包含了路徑名稱。當 *ttyname* 為 NULL 時，*ctermid()* 是不可重入的。

在 Linux 和其他 UNIX 實作中，*ctermid()* 通常會產生 /dev/tty 字串，此函式的目的是便於移植程式到非 UNIX 系統。

34.5 前景和背景行程群組

控制終端機保留前景行程群組的概念，在一個作業階段中，同時只有一個行程群組能在前景行程，作業階段中的其他行程都是在背景行程群組。前景行程群組是唯一能夠自由地讀取和寫入控制終端機的行程群組。當在控制終端機中輸入其中一個會產生訊號的終端機字元之後，終端機驅動程式會將相對應的訊號發送給前景行程群組的成員，34.7 節將會對此進行深入介紹。

> 理論上，可能會出現一個作業階段沒有前景行程群組的情況，例如，若前景行程群組中的每個行程都結束了，而且沒有其他行程注意到這件事而將自己轉移到前景行程群組時，就會出現這種情況。但在實務中這種情況是比較少見的，通常 shell 行程會監控前景行程群組的狀態，當它注意到前景行程群組結束之後（透過 *wait()*），則會將自己移動到前景。

函式 *tcgetpgrp()* 和 *tcsetpgrp()* 分別取得與修改一個終端機的行程群組，這些函式主要供工作控制 shell 使用。

```
#include <unistd.h>

pid_t tcgetpgrp(int fd);
            Returns process group ID of terminal's foreground process group,
                                                        or –1 on error
int tcsetpgrp(int fd, pid_t pgid);
                                        Returns 0 on success, or –1 on error
```

函式 *tcgetpgrp()* 會傳回前景行程群組的行程群組 ID（檔案描述符 *fd* 指定的終端機之前景行程群組，此終端機必須是呼叫行程的控制終端機）。

> 若此終端機沒有前景行程群組，則 *tcgetpgrp()* 會傳回大於 1 的值（此值不會與全部的既有行程群組 ID 重複）（在 SUSv3 有規範此行為）。

函式 *tcsetpgrp()* 可以改變終端機的前景行程群組，若呼叫行程擁有一個控制終端機，則檔案描述符 *fd* 參考的就是那個終端機，接著 *tcsetpgrp()* 會將終端機的前景行程群組設定為 *pgid* 參數指定的行程群組，此參數必須能與呼叫行程所屬的作業階段之某個行程的行程群組 ID 匹配。

函式 *tcgetpgrp()* 和 *tcsetpgrp()* 都已經納入 SUSv3 的標準，Linux 與很多其他 UNIX 實作一樣，利用兩個非標準的 *ioctl()* 操作（即 TIOCGPGRP 和 TIOCSPGRP）來實作此函式。

34.6 SIGHUP 訊號

當一個控制行程失去與終端機的連接之後，核心會發送一個 SIGHUP 訊號來通知它。（也會發送一個 SIGCONT 訊號，以確保在行程之前已因訊號而停止時，可以重新啟動行程）。通常，可能會發生在兩種情況：

- 當終端機驅動程式偵測到連接斷開後，表示數據機或終端機行（terminal line）收不到訊號。
- 當工作站的終端機視窗關閉時，因為最近開啟的檔案描述符（與終端機視窗關聯的虛擬終端機 master 端）已經關閉。

SIGHUP 訊號的預設處理方式是終止行程，若控制行程處理了或忽略了這個訊號，則再繼續讀取終端機就會收到檔案結尾（end-of-file）。

> SUSv3 提及，如果終端機斷開同時發生呼叫 *read()* 的 EIO 錯誤，則無法預期 *read()* 到底會傳回檔案結尾或發生 EIO 錯誤。可攜的程式必須處理這兩種情況。在 34.7.2 節和 34.7.4 節中將介紹在哪些情況下呼叫 *read()* 會發生 EIO 錯誤。

對控制行程發送 SIGHUP 訊號會引起一種連鎖反應，因而導致將 SIGHUP 訊號發送給很多其他行程，這個過程可能會以下列兩種方式發生：

- 控制行程通常是 shell，shell 建立一個 SIGHUP 訊號的處理常式（handler），以便在行程終止之前，shell 能夠將 SIGHUP 訊號發送給它所建立的各個任務。在預設情況下，這個訊號會終止那些任務，但若它們捕獲了這個訊號，就能知道 shell 行程已經終止了。

- 在終止終端機的控制行程時，核心會解除作業階段中所有行程與該控制終端機之間的關聯關係，以及控制終端機與該作業階段的關聯關係（因此另一個作業階段組長可以請求該終端機成為控制終端機），並且透過向該終端機的前景行程群組的成員發送 SIGHUP 訊號，以通知它們控制終端機的結束。

下一節將深入探討這兩種方式。

> SIGHUP 訊號也有其他用途，在 34.7.4 節，我們可以看到當一個行程組成為孤兒行程群組時，會產生 SIGHUP 訊號。此外，手動發送 SIGHUP 訊號通常用來觸發 daemon 行程重新初始化自身或重新讀取其設定檔。（根據定義，daemon 行程沒有控制終端機，因此無法從核心接收 SIGHUP 訊號）。37.4 節將會介紹如何配合使用 SIGHUP 訊號和 daemon 行程。

34.6.1 利用 shell 處理 SIGHUP 訊號

在登入作業階段（login session），shell 通常是終端機的控制行程。大多數 shell 程式在以互動式執行時，會為 SIGHUP 訊號建立一個處理常式，這個處理常式會終止 shell，但在終止之前會向 shell 建立的各個行程群組（包括前景和背景行程群組）發送一個 SIGHUP 訊號。（在 SIGHUP 訊號之後可能會發送一個 SIGCONT 訊號，這取決於 shell 本身以及任務目前是否處於停止狀態）。至於這些群組中的行程如何回應 SIGHUP 訊號，則需要根據應用程式的具體需求，若不採取特殊的動作，則預設情況下將會終止行程。

> 有些工作控制的 shell 在正常結束（如登出或在 shell 視窗中按下 *Control-D*）時，也會發送 SIGHUP 訊號來停止背景工作，*bash* 和 Korn shell 都採取了這種處理方式（初次試著登出時、輸出一筆訊息之後）。
>
> 指令 *nohup(1)* 可以用來使一個指令對 SIGHUP 訊號免疫，即執行指令時將 SIGHUP 的訊號處置設定為 SIG_IGN，*bash* 內建的 *disown* 指令提供類似功能，它從 shell 的任務清單中移除一個任務，這樣在 shell 終止時，就不會向該任務發送 SIGHUP 訊號了。

我們可以使用列表 34-3 的程式進行示範，在 shell 接收 SIGHUP 訊號時，它會依序送出 SIGHUP 訊號給所建立的任務。這個程式的主要任務是建立一個子行程，然後讓父行程和子行程暫停執行，以捕獲 SIGHUP 訊號，並在收到該訊號時輸出一筆訊息。若在執行程式時使用一個選配的命令列參數（它可以是任意字串），則子行程會將其自身放置在一個不同的行程群組中（在同一個作業階段中）。可用來示範說明，shell 不會將 SIGHUP 訊號送給一個不是自己建立的行程群組，即使此行程群組處在與 shell 相同的作業階段。（由於程式中最後一個 for 迴圈是一個無窮迴圈，

因此這個程式使用 *alarm()* 設定一個計時器來發送 SIGALRM 訊號。若行程沒有終止，則當它接收到 SIGALRM 訊號而不做處理時會導致行程終止）。

列表 34-3：捕獲 SIGHUP 訊號

pgsjc/catch_SIGHUP.c

```
#define _XOPEN_SOURCE 500
#include <unistd.h>
#include <signal.h>
#include "tlpi_hdr.h"

static void
handler(int sig)
{
}

int
main(int argc, char *argv[])
{
    pid_t childPid;
    struct sigaction sa;

    setbuf(stdout, NULL);       /* Make stdout unbuffered */

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = handler;
    if (sigaction(SIGHUP, &sa, NULL) == -1)
        errExit("sigaction");

    childPid = fork();
    if (childPid == -1)
        errExit("fork");

    if (childPid == 0 && argc > 1)
        if (setpgid(0, 0) == -1)        /* Move to new process group */
            errExit("setpgid");

    printf("PID=%ld; PPID=%ld; PGID=%ld; SID=%ld\n", (long) getpid(),
            (long) getppid(), (long) getpgrp(), (long) getsid(0));

    alarm(60);                  /* An unhandled SIGALRM ensures this process
                                   will die if nothing else terminates it */
    for(;;) {                   /* Wait for signals */
        pause();
        printf("%ld: caught SIGHUP\n", (long) getpid());
    }
}
```

pgsjc/catch_SIGHUP.c

假設我們在一個終端機視窗輸入下列的指令，以執行列表 34-3 程式的兩個實體（instance），接著關閉終端機視窗：

```
$ echo $$                                   PID of shell is ID of session
5533
$ ./catch_SIGHUP > samegroup.log 2>&1 &
$ ./catch_SIGHUP x > diffgroup.log 2>&1
```

第一個指令會建立兩個行程，這兩個行程屬於由 shell 建立的行程群組，第二個指令建立一個子行程，子行程將自身放置在一個不同的行程群組。

在我們查看 samegroup.log 時，會發現其中包含了下列的輸出，表示兩個行程群組的成員都收到了 shell 發送的訊號：

```
$ cat samegroup.log
PID=5612; PPID=5611; PGID=5611; SID=5533      Child
PID=5611; PPID=5533; PGID=5611; SID=5533      Parent
5611: caught SIGHUP
5612: caught SIGHUP
```

當我們檢查 diffgroup.log 內容時，會發現下列的輸出，表示 shell 在收到 SIGHUP 時，不會向非由它建立的行程群組發送訊號：

```
$ cat diffgroup.log
PID=5614; PPID=5613; PGID=5614; SID=5533      Child
PID=5613; PPID=5533; PGID=5613; SID=5533      Parent
5613: caught SIGHUP                           Parent was signaled, but not child
```

34.6.2 SIGHUP 訊號與控制行程的終止

若因為終端機斷線，而發送 SIGHUP 訊號給控制行程並導致控制行程終止，則會將 SIGHUP 訊號發送給終端機前景行程群組的所有成員（見 25.2 節）。這個行為是控制行程終止的結果，而非與 SIGHUP 訊號關聯的行為。若控制行程出於任何原因終止，則前景行程群組就會收到 SIGHUP 訊號。

> 在 Linux 系統上，SIGHUP 訊號後面會跟著一個 SIGCONT 訊號，以確保行程群組之前若受到一個訊號停止時，可以讓行程群組恢復執行。但 SUSv3 並沒有規範此行為，並且多數其他 UNIX 實作在這種狀況並不會發送 SIGCONT 訊號。

我們可以使用列表 34-4 程式，此程式示範控制行程的終止會導致對終端機前景行程群組的每個成員發送 SIGHUP 訊號。此程式為每個命令列參數建立一個子行程②。若相對應的命令列參數是 *d*，則子行程會將自身放置在自己的（不同的）行程群組③；否則子行程會加入父行程所在的行程群組中。（這裡使用了字母 *s* 來指定後面這種處理方式，雖然只要使用 *d* 以外的任何字母即可）。接著各個子行程設定

了 SIGHUP 訊號處理常式④。為確保它們在無行程終止事件的情況也能終止，父行程和子行程都呼叫了 *alarm()* 設定一個計時器，以在 60 秒之後發送一個 SIGALRM 訊號⑤。最後所有行程（包括父行程）輸出它們的行程 ID 和行程群組 ID ⑥，接著以迴圈等待訊號的到達⑦。當送出訊號之後，處理常式會輸出行程的行程 ID 和訊號數值①。

列表 34-4：在終端機斷開發生時捕獲 SIGHUP 訊號

pgsjc/disc_SIGHUP.c

```
    #define _GNU_SOURCE     /* Get strsignal() declaration from <string.h> */
    #include <string.h>
    #include <signal.h>
    #include "tlpi_hdr.h"

    static void             /* Handler for SIGHUP */
    handler(int sig)
    {
①       printf("PID %ld: caught signal %2d (%s)\n", (long) getpid(),
                sig, strsignal(sig));
                            /* UNSAFE (see Section 21.1.2) */
    }

    int
    main(int argc, char *argv[])
    {
        pid_t parentPid, childPid;
        int j;
        struct sigaction sa;

        if (argc < 2 || strcmp(argv[1], "--help") == 0)
            usageErr("%s {d|s}... [ > sig.log 2>&1 ]\n", argv[0]);

        setbuf(stdout, NULL);               /* Make stdout unbuffered */

        parentPid = getpid();
        printf("PID of parent process is:       %ld\n", (long) parentPid);
        printf("Foreground process group ID is: %ld\n",
                (long) tcgetpgrp(STDIN_FILENO));

②       for (j = 1; j < argc; j++) {        /* Create child processes */
            childPid = fork();
            if (childPid == -1)
                errExit("fork");

            if (childPid == 0) {            /* If child... */
③               if (argv[j][0] == 'd')      /* 'd' --> to different pgrp */
                    if (setpgid(0, 0) == -1)
```

```
                errExit("setpgid");

            sigemptyset(&sa.sa_mask);
            sa.sa_flags = 0;
            sa.sa_handler = handler;
④           if (sigaction(SIGHUP, &sa, NULL) == -1)
                errExit("sigaction");
            break;                      /* Child exits loop */
        }
    }

    /* All processes fall through to here */

⑤   alarm(60);      /* Ensure each process eventually terminates */

⑥   printf("PID=%ld PGID=%ld\n", (long) getpid(), (long) getpgrp());
    for (;;)
⑦       pause();        /* Wait for signals */
}
```

pgsjc/disc_SIGHUP.c

假設使用下列指令在一個終端機視窗執行列表 34-4 的程式：

```
$ exec ./disc_SIGHUP d s s > sig.log 2>&1
```

指定 *exec* 是一個 shell 內建的指令，它會讓 shell 執行一個 *exec()*，並使用指定的程式取代自己。由於 shell 是終端機的控制行程，因此現在這個程式已經成為控制行程，並且在終端機視窗關閉時會收到 SIGHUP 訊號，在關閉終端機視窗之後，可以在 sig.log 檔案看到下列輸出：

```
PID of parent process is:       12733
Foreground process group ID is: 12733
PID=12755 PGID=12755                First child is in a different process group
PID=12756 PGID=12733                Remaining children are in same PG as parent
PID=12757 PGID=12733
PID=12733 PGID=12733                This is the parent process
PID 12756: caught signal  1 (Hangup)
PID 12757: caught signal  1 (Hangup)
```

關閉終端機視窗會導致 SIGHUP 訊號被發送給控制行程（父行程），進而導致終止該行程。從上面可以看出，兩個子行程與父行程位於同一個行程群組中（終端機的前景行程群組），它們都收到了 SIGHUP 訊號，但位於另一個行程群組（背景）中的子行程並沒有收到這個訊號。