第六章
旗號與事件

6-1 前言

旗號(semaphore)的基本概念在第二章裡簡述過，本章將對其程式原理作詳細介紹。另外，與旗號性質時分類似的事件(event)，在τ - OS 中亦有處理。簡而言之，事件的功能在提供同步的動作，不似旗號著重於共享資源的管理。這兩項技巧，在某一程度上與核心是可以脫離的。也就是說核心部份並不使用旗號或事件，而由使用者在需要的情形下來啟用。

6-2 旗號與事件之基本原理

旗號與事件雖是兩種特性與使用時機均不相同的物件，但兩者的動作卻頗有相似之處，因此程式設計時，可以一起考慮。下面先看兩者所用的演算法，在找出其異同處。

旗號(Semaphore)

旗號的演算法有許多種，其中較為可靠的，是稱為計數旗號(counting semaphore)的演算法。這個方法是運用一個計數器，初值設為1，當要使用旗號時，必須先將計數器減一，再檢查計數器的值是否小於零，若是，則代表旗號有人正在使用，程式必須進入等待串列(τ - OS 使用pend())；相反的若計數器大於等於零，則可立刻取用。釋放旗號時，是採相反的動作，先把計數器加一，若計數器仍小於等於零，表示旗號仍有人在等待，必須叫醒等待串列中第一個等待的程式(τ - OS 使用post())；但若計數器大於零，表示無人等候。

這個演算法可以下列虛碼(pseudo code)表示:

---

(1)取用旗號

```
        s=1
        Begin:      s--;
                    if s<0 then
                            pend();      /*進入等候串列*/
                    endif
                    return
        End:
```

(2)釋放旗號

```
        Begin:      s++;
                    if s<=0 then
                            post();      /*叫醒等候串列中第一個程式*/
                    endif
                    return
        End:
```

事件**(event)**

事件的等候與發生, 和旗號並不相同:當要等候事件時, 不論是否已有人等候, 程式都必須進入等候串列;而事件發生則必須把所有在等候串列中的程式叫醒。因此事件的發生與否, 用一個位元的旗標即可表示:0表示事件尚未發生, 1表示事件已發生過了。事件的演算法可以這麼寫:

(1)等候事件

```
        Begin:      if Event Flag !=1 then
                            pend();      /*進入等候串列*/
                    endif
                    Event Flag =0;    /*事件發生後, reset Event Flag */
        End: return
```

(2)事件發生

```
Begin:      Event Flag =1;
                while someone is still in the pending list
                    then
                    post();       /*告知事件發生*/
End:        return
```

兩個演算法的pend()命令都是使程式永遠等下去，直到旗號被釋放或事件發生，這樣很容易產生忙碌等待(busy waiting)或死結，所以pend()命令最好是有time-out能力，使程式僅等候一段時間，若無法取得資源，即立刻返回一個錯誤碼，以防止程式無限制地等待。

6-3 程式設計

旗號與事件最大的相同處，在於兩者都須要讓程式進入等待串列或從串列叫醒程式的執行，因此程式寫作時，可以考量將這部份的動作獨立成一個模組，減少旗號與事件中程式碼重覆的部份。SIG即是因應這個需求而產生的物件。

SIG物件須要維持一個等待串列及使用pend()和post()兩個成員函式來操作SIG物件；等待串列中要記錄進入等待的執行緒ID，記錄ID的物件為sigItem(遺傳自_note)。SIG的基本規格定好之後，接下來先看看pend()和post()要如何動作。

pend()的演算法：

```
int pend(int time_out)
{
配置一個sigItem物件，並填入執行緒ID
將sigItem物件加入等待串列
if(time_out > 0)
        將執行緒移入延遲佇列
```

```
        else
                將執行緒移入懸置佇列，並設為等待狀態。
        內文切換。
        if(執行緒仍在等待串列中){    /*表示等候時間time out*/
                將sigItem物件移出等待串列
                釋放sigItem物件
                傳回time out錯誤碼
                }
        傳回無誤。
        }
```

post的演算法：

```
        int post()
        {
        if(等候串列不是空的){
                do    {
                        取出第一個sigItem
                        將sigItem所代表的執行緒回復執行
                        釋放sigItem
                        } while(執行緒回復失敗及串列不是空的);
                }       /* end if */
        else 傳回串列已空
        }
```

在這兩個演算法中有兩個地方必須特別討論。第一個是關於sigItem的配置。sigItem本身並不是很大的物件，由於串列動態的特性，使sigItem不合適用靜態宣告，若每次使用pend()就配置一個sigItem的記憶體空間，則必定使記憶破碎嚴重(memory fragmentation，指可用記憶體不連續，且碎為小塊的情形)。為了解決這個問題，SIG使用事先配置好的一堆sigItem，從中挑選一個空白未佔用的sigItem使用，以避免每次都要進行記憶體配置／釋放的動作。因此，SIG在宣告時必須先告知sigItem的數量，這個數量代表了SIG所能處理的執行緒數，也限制同時等候一資源的執行緒數量。

第二個問題是等候串列中執行緒的排列順序。這個順序會影響到程式的執行速度(throughput)以及資源使用的公平性。等待串列的排列順序可以依照時間順序，先到先使用，或者依照執行緒的優先權來排均可。SIG採用的先到先使用(first-come-first-serve)策略，主要原因則是考慮公平性的問題，避免資源總是被高優先權的執行緒搶去，而使低優先權的執行緒挨餓(starving)：

```
------------------------------------------------------------------------------------------
----
class far sigItem: public _note{
        unsigned id;        //執行緒ID
        friend sigItem* findfree(sigItem *pool,unsigned size);
        friend class far SIG;
        public:
                sigItem()    {id=0;}
};

sigItem far *findfree(sigItem far *wt/* waiting thread*/
                                                        ,unsigned size)
{ /* 在sigItem的陣列中找尋空的sigItem，採用FIFO演算法*/
        for(int i=0; i<size; i++)
                if( wt[i].id==0 )
                        return( &wt[i]);
        return NULL;
}
```

圖6.1 sigItem物件(Semevent.h(I))與sigItem far * findfree(...)(Semevent.cpp(I))

圖6.1中顯示了sigItem物件的結構與搜尋法則。接下來定義SIG這個物件及其member functions。

```
------------------------------------------------------------------------------------------
----
class far SIG: public Message{
```

```
        private:
                        _list waitL;                    //等候串列
                        sigItem far *pool; //sigItem陣列
                        unsigned pool_size      //sigItem陣列的大小
        public:
                        SIG(unsigned size= 10);
                        ~SIG();
                        virtual int far pend(unsigned time_out=0 );
                        virtual int far post();
};
```

圖6.2 SIG物件(Semevent.h(II))

```
int far SIG::pend(unsigned time_out)
{
        MaskDispatcher;
        sigItem far *wt=findfree(pool,pool_size); //找尋空的位置
        if( wt == NULL)
                return ERR_SIGBUSY;
        wt->id=Tau.CurrentThreadID();                   //設定ID
        waitL.toHead();
        waitL.insert(wt);  // 加入等候串列
        if( time_out )      //若要使用time out, 則將執行緒放到DealQ,
                Tau.SuspendThread(wt->id,0,time_out);
        else                            //否則將執行緒懸置。
                Tau.SuspendThread(wt->id,1);
        SWITCH_CONTEXT;                                 //內文切換
        if( wt->id == Tau.CurrentThreadID()){           //再度取得執行權,
                //但自己的id仍然在等候串列中。
                MaskDispatcher;
                do{ //找尋自己的sigItem, 並從等候串列中去除。
                        sigItem *myItem=(sigItem*)waitL.focus();
```

```
                if(myItem->id == wt->id)        break;
            }while(waitL >>1);
            waitL.erase();                      // remove sigItem
            waitL.toHead;
            wt->id=0;                           //free sigItem
            return ERR_SIGTIMEOUT;              //return time out
        }
        return ERR_NOERROR
}
```

圖6.3 SIG :: pend() (Semevent.cpp)

```
int far SIG:: post()
{
        MaskDispatcher;
        int rc=ERR_NOERROR;
        waitL.toHead();
        if( !waitL.is_empty()){   //若等候串列中有執行緒,
            do{                   /*叫醒第一個執行緒*/
                sigItem far *wt = (sigItem far*)waitL.erase();
                unsigned id = wt->id;
                wt->id = 0;
                rc = Tau.ResumeThread(id,1);       //resume it
            }while( rc != ERR_NOERROR && !waitL.is_empty());
        }       /*若ResumeThread失敗, 表示此執行緒已不在核心中, 故須*/
            /*叫醒下一個*/
        else   rc=ERR_SIGEMPTY;
        return rc;
}
```

圖6.4 SIG::post() (Semevent.cpp)

SIG的用意是維護等待串列, 使得SEM與EVENT物件能更方便的運

作。這個結果在SEM與EVENT物件中可以很清楚地看出來。

```
-------------------------------------------------------------------------------------------
----
class far SEM: public SIG{
        private:
                int S;                  // 計數旗號，初始值為1。
        public:
                SEM(unsigned size=10) : SIG(size){S=1;}
                virtual int far pend(unsigned time_out=0);
                virtual int far post();
};

int far SEM :: pend(unsigned time_out)
{int rc=ERR_NOERROR;
        MaskDispatcher;
        --S;            //遞減計數值
        if( S < 0)
                if((rc=SIG::pend(time_out)) !=ERR_NOERROR){
                        ++S;
                        ReleaseDispatcher;
                        return rc;
                }
        ReleaseDispatcher;
        return rc;
}

int far SEM ::post()
{
        MaskDispatcher;
        ++S;            //遞增計數值
        if( S <= 0){
                SIG :: post();
```

```
            SWITCH_CONTEXT;
        }
        ReleaseDispatcher;
        return ERR_NOERROR;
}
```

圖6.5 SEM物件(Semevent.h)與SEM::pend(), SEM::post()
(Semevent.cpp)

```
class far EVENT : SIG{
        private:
            unsigned trig:1;          //事件觸發旗標
        public:
            EVENT(unsigned size=1) : SIG(size){trig =0;}
            virtual int far pend(unsigned time_out=0);
            virtual int far post();
};
int far EVENT :: pend(unsigned time_out)
{       int rc=ERR_NOERROR;
        if( !trig){     //事件未發生則進入等待。
                        MaskDispatcher;
                        if(( rc=SIG :: pend(time_out)) !=ERR_NOERROR){
                                ReleaseDispatcher;
                                return rc;
                        }
                        ReleaseDispatcher;
        }
        trig=0;
        return rc;
}
int far EVENT :: post()
{       int count=0;
```

```
        MaskDispatcher;
        trig=1;
        while( SIG::post() == ERR_NOERROR) //叫醒所有等候事件的執行
                                           //段。
                count++;    //若count不等於0, 表示有執行緒被叫
                            //醒,
        if( count )         {SWITCH_CONTEXT;}//而須要執行內文切
                                           //換。
        else                ReleaseDispatcher;
        return count;
}
```

圖6.5 EVENT物件(Semevent.h)與EVENT::pend(), EVENT::post() (Semevent.cpp)

SEM與EVENT物件的使用, 這裡用一個程式來示範:FACE.CPP。這個程式中, face()和print()會使用螢幕設備, 因為Turbo C++的螢幕I/O模組是不可重覆進入的, 若不協調兩者的動作的話, 程式馬上會當掉, 因此須要旗號。Kb()負責鍵盤的界面, 當Kb_trig()偵查有鍵盤動作時, 便會送出EVENT叫醒Kb()進行處理。FACE.CPP的動作是在文字螢幕上產生一群ASCII碼為1的符號(一張臉), 而這群符號的移動速度均不同。程式中事實上只有一個符號的程式碼, 藉由τ - OS產生一群相同的工作, 只是初始參數不同而已(代表不同之移動速度)。本範例除了展示旗號的功能外, 也明確顯示單工與多工之差別。讀者若仍對多工無甚概念, 可使用單工之方式撰寫本範例並加以比較, 可加深印象。

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <bios.h>
#include "tauos.h"        /* τ - OS的主含括檔*/
char *string[]={"This is a demo of Tau-OS.        ",
```

```
                "Press Enter to create another face (Max is 20).",
                "Press Esc to exit program.          ",
                "Thank you for playing me!!          "
                                                    };
/* face : 在螢幕上顯示一個移動中的"臉"。*/
void far face(void far *arg)
{
        int y= *(int* ) arg; //將參數轉型成整數
        int prex=1,x=1,step=1;
        int wait;
                MaskDispatcher;
                wait=random(20)+1;    //決定移動速度
                ReleaseDispatcher;
                while(1){
                        if(video.pend() == ERR_NOERROR){
                /*此block的程式碼必須在取得video旗號之後才能執行*/
                                /*video的宣告在TauOS.H中          */
                                gotoxy(prex,y);
                                putch(‘ ’);//清除前一張臉
                                gotoxy(x,y);
                                putch(1);           //畫新的臉
                                video.post();       //釋放video旗號
                        }
                        if( x==80)          step=-1;
                        if( x==1)           step=1;
                        prex=x;
                        x=x+step;
                        TauTimeDelay(wait);    //使Task延遲wait個ticks
                }
}

/* print : 固定4秒鐘在螢幕底顯示一行訊息*/
void far print(void far *)
```

```
{int i=0;
        while(1){
                        i=i%4;
                        if(video.pend() == ERR_NOERROR){
                                gotoxy(5,25);
                                printf("%s",string[i]);
                                video.post();
                        }
                        i++;
                        TauTimeWait();          //等待下一個週期
        }
}

EVENT kb_press;       //鍵盤的事件

/* kb : 處理鍵盤事件*/
void far kb(void far *)
{       int count=0;
        unsigned id;
        union{
                char byte[2];
                int word;
        }key;

        while(1){
                        kb_press.pend();        //等候鍵盤事件
                        key.word = bioskey(0);          //讀入新的按鍵
                        switch(key.byte[1]){            //解讀scan code
                                case 28:    //Enter
                                        if(count ==20)     break;
                                        count++;
                                                //產生新的工作
                                        TauCreateSliceTask(id,face,
```

```
                                        (ARG)&count,1024,15);
                                break;
                        case 1:     //ESC
                                TauShutDown;
                                break;
                }
        }
}

/* kb_trig : 檢查是否按鍵, 並觸動kb_press事件*/
TASK kb_trig(void far*)
{
        while(1){
                if(bioskey(1))
                        kb_press.post();
                else
                        TauGiveUpSlice();       //放棄執行時間
        }
}

/* TauMain : τ - OS的程式進入點*/
void far TauMain(int, char*[])
{       unsigned id;
        clrscr();
        randomize();
        /*產生工作*/
        TauCreateTimeTask(id,print,0L,4096,14,80);
        TauCreateEventTask(id,kb,0L,4096,15);
        TauCreateSliceTask(id,kb_trig,0L,4096,15);

        _setcursortype(_NOCURSOR);
        /*啟動多工*/
        TauStart(20);       //週期:0.05秒
```

```
        _setcursortype(_NORMALCURSOR);
}
```

圖6.5 FACE.CPP

6-4結語

本章介紹旗號(semaphore)與事件(event)之基本原理與程式設計。誠如第二章所述，旗號是增加系統額外負擔的動作，如非得已，盡量少用之。例如對本章之範例(FACE.CPP)而言，如果螢幕驅動及繪圖的指令均為可再進入者(reentrant)，則無須使用旗號。若讀者已有這些自寫的指令，只要稍加修改即可。但注意不可有BIOS call。另外，本書之附錄E含另一範例說明旗號之一有趣應用(哲學家問題)，供讀者參考。