

多緒開放式程式開發環境

張欽隆

中央研究院資訊科學研究所

evirt@iis.sinica.edu.tw

廖賀田

淡江大學資訊管理學系

htliaw@mail.im.tku.edu.tw

摘要

整合式開發環境可以簡化軟體開發的瑣碎工作，但在此種開發環境中，程式員仍需面對冗長的編譯時間並受限於封閉式的執行環境。本論文提出一個多緒開放式的互動發展環境，它具有下列特色：(1) 以直譯中間碼的方式改善編譯耗時的缺點，(2) 以互動的方式提供開放的執行環境，(3) 提供多緒使用介面讓程式測試更有彈性。透過本環境，程式員將可節省程式開發時間並可動態地測試程式。

關鍵詞：開發環境、多緒、執行緒管理、互動、開放。

Multi-threaded Open Integrated Development Environment

Chin-Lung Chang

Institute of Information Science,
Academia Sinica, NanKang,
Taipei 115, Taiwan 115, Taiwan, R.O.C

evirt@iis.sinica.edu.tw

Heh-Tyan Liaw

Department of Information Management
Tamkang University

htliaw@mail.im.tku.edu.tw

ABSTRACT

Integrated development environments may simplify routine works in software development. But a programmer in such an environment still has to endure the lengthy compiling time and to be limited by the closed runtime environment. A multi-threaded open integrated development environment is proposed in this paper. The features of the environment are: (1) interpreting intermediate codes to reduce compiling time, (2) supporting an open runtime environment in an interactive manner, (3) supporting a multi-threaded user interface to make the testing of programs more flexible. In this environment, a programmer may reduce the time of development, and may test programs dynamically.

Keywords: development environment, multi-thread, thread management, interactive, open

壹、緒論

一、研究動機

傳統開發環境多為文字介面，程式員需在命令列一步步操控。為了方便，許多公司便提供了整合式的開發環境(Integrated Development Environment, 簡稱 IDE), 這些常見的 IDE 只是簡化程式員的操作，還不能完全滿足開發軟體的需求。依其所提供的功能，我們可將 IDE 分為三種等級如下：

1. 整合式開發環境(Integrated Development Environment, IDE)

此為軟體開發環境的初級系統，只能滿足最基本的開發需求。程式員必須先寫好整個程式，才能編譯、連結與執行。

2. 開放式開發環境(Open Integrated Development Environment)

開放式的開發環境將變數和副程式開放出來讓程式員自由取用。程式即使不完整，仍可單獨執行某個副程式，其間還可隨時添加(全域)變數，並可在測試過程中自由更動它的內含值。這種開發環境使[編寫-編譯-連結-測試]的程式開發循環巨幅縮併。此種開發環境以 forth[5]環境，lisp[11]，及 SmallTalk[3]為代表。

3. 多緒開放式開發環境(Multi-threaded Open Integrated Development Environment)

在開放式的互動直譯器上運作時，程式員所執行的副程式未必能立刻結束工作，這時候直譯器的交談介面將進入凍結狀態。為了要使程式員在某個副程式正執行時仍能操作直譯器，尤其是能為執行中的副程式改變某個使用中的全域變數，因此開發環境應該要提供多緒(multi-threading)能力。目前多緒程式的寫作已經相當普及，因此多緒的開發環境就更是急需。

目前常見的開發環境如下：

1. Microsoft Visual Studio

Microsoft 的 Visual Studio 內含了 Visual C/C++、Visual Basic 和 Visual J++ 等等的開發環境。Visual C/C++、Visual J++ 屬於傳統編譯式的程式開發環境，因此程式員仍需經過冗長的編譯流程。而 Visual Basic 雖內含直譯器，但並不提供開放的能力，也就是程式員無法觸及程式內的變數與函式。

2. EiC

EiC, An embeddable/extensible interactive bytecode C interpreter[6], EiC 是一套 C 語言的交談式直譯器(interactive interpreter), 它少了一些標準 C 語言所提供的功能，但提供了

更安全，如 pointer-safe，互動操控模式，可擴充式介面和內嵌到其它語言等特色。

3. CINT

Cint[10]是一套 C/C++的互動式直譯器，作者宣稱其能與 95%的 AnsiC, 90%的 C++相容，並提供 source code debugger 的能力以供除錯。Cint 可以直接執行使用，但也可以利用一些工具來將程式員的 C/C++程式碼內建到 Cint interpreter 中，Cint 可以在多個平台上執行，Cint 在正常使用的情況下可以提供程式員在互動環境內使用 C++撰寫程式。我們實際使用這套系統的經驗是它不太穩定，常常會當機。尤其是它的錯誤偵測與恢復的部份作得不好，時常需要重新啟動。

4. Tcl/Tk

Tcl 為 Tool Command Language 的縮寫[7]，為 John Ousterhout 所發展的一套程式語言，它通常都和 Tk(Tool kit)這套圖型元件整合在一起。Tcl 扮演著和 Shell Script 相同的角色，可整合各種程式資源。Tcl 有許多特色，其中尤以整合式圖型元件最為人所注意。Tcl 本身還沒有物件導向的語法。

5. Python

Python[9]是一套擁有互動式介面的物件導向式 Script 語言，主要的發展者為 Guido van Rossum, Python 支援多型(polymorphism) 重載(overloading) 複繼承(multiple inheritance) 等。Python 亦提供互動式操作介面。

6. TkF

我們先前開發了一套改良的 Forth 系統，稱為 TkF[1]。TkF 是一套擁有圖形使用介面的 Forth 系統，且提供許多傳統 Forth 未支援的能力，如:型別支援、參考變數、指標變數、區域變數和互動式控制流程等等的特色，TkF 結合了視窗介面並強化傳統 Forth 系統，除了提供一個友善的 Forth 發展環境外，TKF 不但支援多種資料型態，更可支援程式設計師自定的資料型態。此外，變數參照與遞迴呼叫的支援，使得 Forth 程式設計師能更有彈性地來發展程式。總之，TkF 加強了傳統 Forth 的能力，並提供一個更友善的介面使得開發 Forth 程式更方便更有效率。

另外，江文傑[2]在民國 86 年提出的「設計並實作一多人視覺化程式發展環境」一文中，他認為建構一套大型的軟體系統通常需要多人的合作開發，因此其論文中所提之視覺化發展環境主要著重在多人合作程式設計(collaborative programming)，和視覺化物件導向程式設計(visual object-oriented programming)的方面。

上述的開發環境除了第一項的 Microsoft Visual Studio 屬於封閉式開發環境外，其餘皆屬開放式的開發環境，但都還未提供多緒的使用介面。本論文提出的 TkF2 改良了先前的 TkF，新加入多執行緒的能力，而成為一套「多緒開放式的程式開發環境」。這套直

譯器可以達成以下的目的:1.加速的程式的開發, 2.縮短初學者的程式學習時間, 3.支援多緒的程式測試環境, 4.提供更具彈性的執行緒程式寫作方式。

本論文共分五章, 第一章為簡介, 第二章討論本直譯器的系統模型, 在第三章描述本直譯器的使用介面, 第四章詳述整個直譯器系統的規劃和開發細節。第五章是結論。

貳、程式環境的抽象模型

一、多緒開放式的程式開發環境的需求

多緒開放式的程式開發環境在軟體開發與測試上非常重要。為解說這個重要性, 我們用一個實例來作展示。這個實例姑且以 C++ 語言來表達, 但它的精神與所用的語言無關。

假設我們有一個程式, 這程式能讓一些球在視窗內任意移動。以下為一典型的程式片斷:

```
void main() {  
// .....  
    Ball b1;                // 產生 Ball 物件  
    b1.SetSize(10);         // 設定 b1 的 size  
    b1.SetVelocity(2,4);    // 設定 b1 的速度向量  
    b1.move();              // 呼叫 move 來移動球體  
// .....  
}
```

上述的程式碼在傳統的 IDE 下, 經過 C++ 編譯器編譯執行後, 若程式員希望修改一些參數來做其它的測試時, 只好重新啟動編輯器, 將程式碼修改後, 再編譯執行一次。這實在是無謂的時間浪費。相對地, 若我們在一個擁有多緒開放介面的 C++ 開發環境下撰寫並執行這個程式片斷, 交談操作的情況如下:

(> 為直譯器提示符號, thread 是系統外加的語法)

```
> Ball b1;                // 產生 Ball 物件  
> b1.SetSize(10);         // 設定 b1 的 size  
> b1.SetVelocity(3,3);    // 設定 b1 的速度向量  
> thread { b1.move(); }   // 開一條新的 thread 來執行  
> b1.SetSize(20);         // 修改球體的大小  
> b1.SetVelocity(5,5);    // 修改速度向量
```

(註解部份為本論文所加的說明, 操作時不含這些註解)

我們可以看到，在多緒開放式的使用介面下執行時，我們可以開出一條執行緒來執行 `b1.move()` 這個函式。之所以要開出另一條執行緒來執行是因為 `b1.move()` 這個函式若直接執行時將會造成互動介面的凍結。

開出新執行緒執行 `b1.move()` 後，系統就立即返回互動介面，於是使用者便可以繼續操控。如上例我們對球體的屬性做了修改，於是程式的行為和表現將會馬上隨著改變，透過這種多緒開放式的使用介面支援，我們將可以做到在執行期變更變數的內容，如此我們便可透過這種方式來測試系統在不同的情況下的反應，而不需重新修改程式。

更甚者，只要程式員能在發展系統時將同步化的問題考慮進去，我們甚至可以隨時開一條新的執行緒以進行更多的測試，以上例為例，當我們已經有一個球體在視窗內移動時，我們可以再產生另一個球體且也讓它在同一個視窗內移動，如：

```
> Ball b2;
> b2.SetSize(5);
> b2.SetVelocity(2,2);
> thread { b2.move(); } // 讓 b2 在視窗內移動
```

這種極具彈性的軟體發展與測試環境將可大幅降低程式員的負擔，程式員可輕鬆地在開發環境進行發展與測試，待一切無誤後，再呼叫編譯器將程式碼編譯成可執行檔。

二、多緒開放式的程式開發環境模型

開發環境所管理的元件可以粗分為靜態元件和動態元件兩種，靜態元件包括變數和函式，而動態元件包括執行緒以及立即式的流程控制。靜態元件的使用語法已決定於所選的程式語言，而動態元件的使用還需作更進一步的探討。我們以 Unix Shell(ex. Bash[3]) 的工作控制機制為基礎，將動態元件的管理模型分為“個別狀態模型”(individual status model)和“整體狀態模型”(global status model)兩種，分別介紹如下：

1. 個別狀態模型

我們將個別執行緒的執行狀態分為前景執行狀態、背景執行狀態、暫停狀態、停止狀態共四種。各個狀態可互相移轉，最後變成停止狀態。以下解說各個狀態：

(1) 前景執行狀態 (Foreground Running State)

前景執行狀態是指執行緒正在執行之中，並且可取得標準輸入，此時該執行緒可以接收使用者的輸入，也可以輸出資料到標準輸出。本文將這個狀態記為 `TS_FG`。

(2) 背景執行狀態 (Background Running State)

背景執行狀態是指執行緒正在執行之中，但它並無標準輸入。因此它無法取得使用者的輸入，但可以輸出資料到標準輸出。本文將這個狀態記為 TS_BG。

(3) 暫停狀態 (Suspend State)

當執行緒位在此狀態，它不佔用中央處理器(CPU)，當然也沒有輸出入。本文將這個狀態記為 TS_SP。

(4) 停止狀態 (Stop State)

當執行緒結束執行之後，尚保留有一些資訊如執行緒識別碼、結束狀態等等以供系統或程式員運用。此時狀態為停止狀態。本文將這個狀態記為 TS_ST。

上述狀態的轉移乃是透過四種訊號的輸入所產生，這四種訊號分別為 SigSusp、SigStop、SigBG 和 SigFG 這四種，下面我們分項介紹這四種訊號的意義及產生的狀況：

(1) SigSusp，執行緒暫停。

以下為 SigSusp 產生的狀況：a. 終端機前使用者按下 ctrl+z 時，前景執行緒會收到 SigSusp。b. 背景執行緒若需要標準輸入時，則會收到 SigSusp 並進入暫停狀態。c. 其它執行緒使用 susp 指令對某執行緒發出 SigSusp 訊號。

(2) SigStop，執行緒結束。

以下為 SigStop 產生的狀況：a. 終端機前使用者按下 ctrl+c 時，前景執行緒會收到 SigSusp。b. 其它執行緒使用 kill 指令對某執行緒發出 SigStop 訊號。c. 執行緒正常結束。

(3) SigBG，背景執行，讓處於暫停狀態的執行緒切換到背景執行。

以下為 SigBG 產生的狀況：a. 直譯器執行緒使用 bg 指令，產生 SigBG，使得接收此訊息的背景暫停執行緒可以在背景執行。

(4) SigFG，前景執行，讓正在背景執行或暫停的執行緒切換到前景執行。

以下為 SigFG 產生的狀況：a. 直譯器執行緒使用 fg 指令，產生 SigFG，使得接收此訊息的背景暫停或背景執行的執行緒可以在前景執行。

我們將影響執行緒狀態的輸入訊息分為 SigSusp、SigStop、SigBG 和 SigFG 這四種。它們影響執行緒執行狀態的情形如下表：

	SigSusp	SigStop	SigBG	SigFG
TS_FG	TS_SP	TS_ST	X	X
TS_BG	TS_SP	TS_ST	TS_BG	TS_FG
TS_SP	TS_SP	TS_ST	TS_BG	TS_FG
TS_ST	TS_ST	TS_ST	TS_ST	TS_ST

(X：表在此狀態下，不可能收到此訊息，故狀態不變)

表一：影響執行緒狀態的輸入訊息

每一條執行緒在一開始被執行時只有兩種可能的狀態，一為前景執行狀態 TS_FG，另一為背景執行的狀態 TS_BG，執行緒在 TS_FG 及在 TS_BG 的差別，在於此執行緒能否取得標準輸入。

下面我們分項介紹執行緒的每一種執行狀態遇到各種訊息時狀態的轉移，需注意的是，下述所指的執行緒並不包含直譯器執行緒，因直譯器執行緒為執行緒管理者的角色，因此會對上述的訊息做特殊的處理或是忽略：

(1) 執行緒處於 TS_FG 狀態時

當執行緒處於 TS_FG 狀態時，直譯器執行緒一定位於背景執行狀態，因此 susp、stop、bg 和 fg 等執行緒控制指令不可能透過直譯器執行緒下達，因為直譯器執行緒無法取得標準輸入，此時只能透過熱鍵控制指令來讓目前的前景執行緒改變狀態，也就是此時執行緒只可能收到 SigSusp 或 SigStop 這兩種訊號。當位於 TS_FG 狀態的執行緒收到 ctrl+z 所產生的 SigSusp 訊號時，狀態會轉移到暫停狀態 TS_SP，若收到 ctrl+c 所產生的 SigStop 訊號時，狀態則轉移到停止狀態 TS_ST。

(2) 當執行緒處於 TS_BG 狀態時

當執行緒處於 TS_BG 狀態時，此時前景執行緒可能為直譯器執行緒或是其它的執行緒，此時可能接收到的訊息有 SigSusp、SigStop、SigBG 和 SigFG。當收到 SigSusp 時，狀態會轉移到 TS_SP 進入暫停狀態，若收到 SigStop 則轉成 TS_ST 進入終止狀態，收到 SigBG 時則狀態不變，收到 SigFG 時，則轉到 TS_FG 前景執行的狀態。

(3) 當執行緒處於 TS_SP 狀態時

當執行緒處於 TS_SP 狀態時，此時必定處於背景狀態，可能收到到訊息有 SigSusp、SigStop、SigBG、SigFG 四種。收到 SigSusp 時狀態不變，收到 SigStop 時進入 TS_ST 終止狀態，若收到 SigBG 則轉入 TS_BG 進入背景執行的狀態，而收到 SigFG 則進入 TS_FG 前景執行狀態。

(4) 當執行緒處於 TS_ST 狀態時

當執行緒處於 TS_ST 狀態時，不管收到何種訊息，狀態皆不變。

2. 整體狀態模型

上述的狀態機是用來描述個別執行緒的執行狀態的轉移。本論文須探討多執行緒的控制介面，即整體狀態模型。我們把多執行緒操作狀態分成兩大類：

(1) FTi ($0 \leq i \leq \text{MAX}$)

此狀態表直譯器執行緒為前景執行，此時有 i 條執行緒在背景執行或暫停。

(2) Ti ($1 \leq i \leq \text{MAX}$)

此狀態表直譯器執行緒為背景執行，此時有 1 條執行緒在前景，另有 $i-1$ 條執行緒在背景執行或暫停。

這兩大類的狀態主要用來表示執行緒和直譯器執行緒的前景和背景的切換以及執行緒數量的增減，能夠影響上述兩類狀態的輸入有下面幾種：

(1) 建立前景執行的執行緒，以 NewFT 代表

在直譯器執行緒內以產生前景執行緒的語法產生執行緒時。

(2) 建立背景執行的執行緒，以 NewBT 代表

在直譯器 thread 內以產生背景執行緒的語法產生執行緒時。

(3) 前景執行緒執行結束，以 EndFT 代表

前景執行緒執行結束的原因：a. 正常結束。b. 終端機前使用者按下 ctrl+c 時。

(4) 背景執行緒執行結束，以 EndBT 代表

背景執行緒執行結束的原因：a. 正常結束。b. 終端機前使用者在直譯器執行緒內使用 kill 指令對某背景執行緒發出 SigStop 訊號時。

(5) 前景執行緒暫停執行，以 SuspFT 代表

終端機前使用者按下 ctrl+z 時，執行緒會進入背景暫停狀態，此時前景執行緒變成直譯器。

(6) 讓背景執行或暫停的執行緒變成前景，以 FG(i) 為代表 ($1 \leq i \leq \text{MAX}$)

終端機前使用者在直譯器執行緒內使用 fg 指令，可使得接受訊號的背景執行緒變成前景執行，直譯器執行緒轉為背景執行。

下表為上述的狀態轉移圖：

	NewFT	NewBT	EndFT	EndBT	SuspFT	FG(i),(1<=i<=MAX-1)
FT0	T1	FT1	FT0	X	FT0	X
FT(i),(1<=i<=MAX-1)	T(i+1)	FT(i+1)	FT(i)	FT(I-1)	FT(i)	T(i)
FTMAX	FTMA X	FTMA X	FTMAX	FT(MAX-1))	FTMA X	TMAX
T1	X	X	FT0	X	FT1	X
T(i),(1<=i<=MAX-1)	X	X	FT(i-1)	T(I-1)	FT(i)	X
TMAX	X	X	FT(MAX-1))	T(M-1)	FTMA X	X

(X：表在此狀態下不可能收到此輸入，因此狀態不變)

表二：執行緒狀態轉移表

系統一開始時，一定是處於 FT0 的狀態，即直譯器為前景執行緒，且沒有其它執行緒在背景執行或暫停。在 FT0 的狀態下，因為沒有其它的背景執行緒存在，因此 EndBT 和 FG(i) 都無任何作用，又因直譯器會忽略暫停和停止訊息，因此若收到 SuspFT 和 EndFT 時，將直接忽略，因此若想轉換到其它的狀態，只能透過 NewFT 或 NewBT 這兩種訊息。收到 NewFT 時，直譯器執行緒轉成背景執行，新產生的執行序則為前景執行，因此狀態由 FT0 轉到 T1，若收到 NewBT，則新產生的執行緒會在背景執行，直譯器執行緒依然為前景，因此狀態由 FT0 轉到 FT1。

狀態轉到 T1 時，此時直譯器執行緒為背景執行，程式員無法透過直譯器產生新的執行緒，或是進行執行緒的操控，因此 NewFT，NewBT 和 FG(i) 訊息無法被發出，又因 T1 表有一執行緒為前景，且唯一的背景執行緒為直譯器執行緒，因此 EndBT 也無法有所作用。總而言之，在 T1 狀態時，只能收到 EndFT 和 SuspFT 這兩個輸入，收到 EndFT 時，表前景執行緒執行結束，直譯器執行緒將回到前景執行，因此狀態將轉換到 FT0。若是收到 SuspFT 時，此時直譯器執行緒進入前景，執行緒本身則進入背景，因此狀態轉換到 FT1。

狀態轉到 FT1 時，大部份情況與 FT0 的狀態類似，不同的是，此時有一條執行緒在背景，因此在 FT1 時，可以多收到 EndBT 和 FG(i) 這兩種訊息，收到 EndBT 時，表背景執行緒執行結束，因此狀態轉換到 FT0，若收到 FG(i) 時，表原本處於背景的執行緒變成前景，因此直譯器轉入背景，此時狀態也轉換到 T1。

當狀態轉換到 T2 時，大部份的情況和 T1 相似，只是此時背景執行緒除了直譯器執行緒外還有另一條執行緒的存在，因此在 T2 時可以接受 EndBT 輸入，收到 EndBT 時，表有一背景執行緒執行結束，因此少了一條執行緒，狀態轉移到 T1。

最後，當狀態處於 FTMAX 時，表執行緒數量已達極限，無法再新增任何的執行緒，因此若收到 NewFT 或是 NewBT 時，狀態依然不變。

參、使用介面

一、TkF2 基本語法介紹

TkF2 是改自 TkF 的一套 forth 語言系統，因此在使用上承襲了許多 forth 語言的特色，forth 語法最大的特色就是採用後序式表示法：先寫資料再寫操作方式。在 TkF/TkF2 內程式員要對任何資料做動作時，必需先將資料置入資料堆疊後，再進行資料操作，以【1.3 x + .】為例，程式員輸入 1.3 和 x 時，表示將 1.3 置入堆疊後，再將 x 這個變數的參照(reference)也置入堆疊，而【+】這個指令會從堆疊內取出 1.3 和 x 內的值後求其和，再將所得的和置入堆疊，最後【.】會將堆疊最頂層的資料取出並印出。另外，因 TkF/TkF2 的流程控制繼承自 Forth[5]，故在此不另加說明。

二、靜態元件的管理

靜態元件即變數和函式，TkF2 在此部份的語法比 Forth[4]和 TkF 完整。限於篇幅，不在本論文中討論。

三、動態元件的管理

Bash[4]的工作管理介面是一種成熟且相當完善的多程序操控介面，本直譯器的執行緒控制介面將以 Bash 的工作管理介面為模型的藍本，如此不但可以避開開發新的介面模型所可能遭遇的問題，也可以降低使用者的學習成本。所謂的動態元件管理，在本直譯器下所指為執行緒的管理，下面我們分項描述本直譯器的動態元件管理介面：

1. 執行模式與指令集

相對於 Bash 的執行機制，我們以執行緒來模擬處理程序，也就是直譯器的地位等同於 Bash，而直譯器所產生的執行緒等同於處理程序，因此我們制定了下面的執行模式：

(1) ...[enter]

直接於直譯器執行緒內執行 enter 之前的程式碼，此時直譯器執行緒為前景。

(2) \ ...[enter]

以 \ 開頭到[enter]結尾中間的程式碼會在一條新的執行緒內執行，此時新的執行緒為前景，直譯器執行緒則會等待新執行緒的執行結束再往下繼續做。

(3) \ ... [&enter]

以 \ 開頭到 [&enter]結尾中間的程式碼會在一條新的執行緒內執行，此時新的執行緒為背景，直譯器執行緒不會等待新執行緒的執行結束，而是一開出新執行緒後就馬上返回

互動模式繼續執行等待使用者的輸入。

除了上述的執行模式外，在 Bash 環境裡，提供了許多用來管理處理程序的機制。我們的多緒使用介面也提供這些工作管理命令來管理所有的執行緒，列示如下：

- (1) `susp n`，`n` 為執行緒管理代碼，用來暫停一條執行緒的執行。
- (2) `kill n`，`n` 為執行緒管理代碼，用來終止一條執行緒的執行。
- (3) `ctrl+c`，終止前景執行緒的執行，當前景為直譯器執行緒時，且有程式指令正在直譯器執行緒執行時，`ctrl+c` 亦可終止之。
- (4) `ctrl+z`，暫停前景執行緒的執行，當前景執行緒為直譯器執行緒時，無作用。
- (5) `jobs`，列出所有執行緒的執行狀況。
- (6) `bg n`，`n` 為執行緒管理代碼，`bg` 可以讓一個被暫停的執行緒在背景執行，此時前景為直譯器執行緒。
- (7) `fg n`，`n` 為執行緒管理代碼，`fg` 可以讓一個被暫停的執行緒在前景執行，此時直譯器執行緒變成背景並且會等待前景執行緒的執行結束。

2. 執行緒的產生語法

我們亦提供了一套 `forth` 指令集，使程式也可以產生及操控執行緒，分項介紹如下：

- (1) `threadbeg threadend`

在 `threadbeg` 和 `threadend` 之間的程式指令將會在一條新的執行緒內執行，`threadbeg` 會在產生出一條執行緒後，將此執行緒的代碼置於資料堆疊上以供後續的指令使用。

- (2) `threadwait`

用來等待一條執行緒的執行結束，`threadwait` 會到資料堆疊上去取出執行緒代碼來決定要等待哪一條執行緒的結束。

- (3) `setfg`

將資料堆疊上的執行緒代碼所代表的執行緒設為前景，此時直譯器執行緒變成背景。

肆、使用實例

大部份的程式員在開發系統的同時，就會盡可能地測試系統是否有問題，但以傳統開發環境而言，程式員每次一做修改就必需經過編譯、連結、執行的過程，若系統發展至一定規模時，有時只是為了測試某個小修改是否正確，卻必需重新編譯一次系統，這種方式不但耗時，更是無謂的時間浪費。若是在交談式多緒直譯器下發展程式，則可以將這種時間的浪費降至最低，我們來看看下面這些例子：

- (1) 假設有一個函式會依照傳入的整數的參數不同，而印出不同的字元，如下：

```
int fundef symbolmsg
char argdef ch
    blkbeg
        ch . cr           // 印出字元
        2000 sleep       // 暫停 2 秒
    blkend
funend
```

於是，在互動模式下，程式員可以傳入不同的參數值，來檢驗此函式是否運作正常，如下面的執行情形：

```

(0)0- '*' symbolmsg
*
    ok[0]
(0)0- '%' symbolmsg
%
    ok[0]
(0)0-

```

(2) 承上例，我們有一個函式，會呼叫 symbolmsg 來印出字元，但這個函式為一無窮迴圈每次一個迴圈就會呼叫 symbolmsg 一次，這個函式傳給 symbolmsg 的參數乃是一個全域的字元變數，因此只要一執行此函式，就會不斷地印出此全域變數內的字元，函式內容如下：

```

char vardef tvar // 全域變數
'*' tvar =: // 初始為 '*'
int fundef loopmsg // 定義 loopmsg 函式
    blkbeg
        begin 1 while // 無窮迴圈
            tvar symbolmsg // 呼叫 symbolmsg 函式
        repeat
    blkend
funend

```

這個 loopmsg 函式為一個無窮迴圈函式，因此若我們直接在互動模式下執行，則整個互動介面將會被凍結住，程式員將無法再行輸入程式指令，此時如果互動介面不會被凍結，程式員將可以直接修改影響 loopmsg 的全域變數值，使得程式員將可以在程式的執行期修改變數內容，以修正函式的某些行為。本論文的直譯器另一個強調的重點在於多緒的使用介面，我們可以透過直譯器提供的多緒使用介面輕易地讓 loopmsg 於另一條新的執行緒上執行，如下：

```

(0)0- \ loopmsg &
(0)0-

```

此時 loopmsg 會在另一條執行緒上執行，於是直譯器執行緒便得以立即返回互動模式讓程式員繼續輸入指令：

```

(0)0- '$' tvar =:

```

上面這條指令主要是修改 tvar 這個全域變數為\$字元，於是 loopmsg 變成輸出\$字元。

伍、結論

傳統開發環境雖然省去開發人員不少的時間，但實際上程式員所面對的開發流程還是沒變，這種每修改一次就需重新做一次編譯、連結、執行的循環對程式員來說只是時間上的浪費。因此我們針對目前的整合式開發環境進行檢討與改進，我們提出多緒開放式的程式開發環境，我們透過直譯器的方式縮短了編譯連結的循環，加速了程式系統的開發，另外我們也為直譯器加入了多執行緒的能力，使得程式員可以輕鬆地開發多執行緒程式，最後我們為了讓執行緒的管理可以更簡易，我們參考了 Bash 的工作管理介面，為直譯器加入執行緒管理介面。本文所探討的便是多緒開放式程式環境相較於傳統整合式開發環境所帶來的改進，列舉如下：

1. 降低初學者學習門檻

透過互動模式的輔助，初學者可以在互動模式下學習程式的語法，而無需面對繁複的編譯連結流程。

2. 加速程式的開發

直譯器內程式是以中間碼的型式執行，因此可以節省產生機器碼及連結所耗費的時間。

3. 動態的程式測試

在直譯器內，程式的內容是開放的，因此程式員可以透過互動模式來碰觸程式的內容，幫助程式員測試程式與除錯。

4. 更具彈性的執行緒產生方式

在多緒開放式程式開發環境內，程式員透過執行緒的執行緒管理界面可以任意讓某些程式區段在執行緒內執行，而無需對程式做修改。

5. 方便的多緒程式除錯環境

撰寫多緒程式時很容易造成程式的死結，然而在多緒開放式程式開發環境內開發程式時，程式員可以透過互動環境取得執行緒的各種資訊，因此便可透過這些資訊判斷出程式內可能產生死結的問題點。

本論文後續發展方向主要有兩項：

1. C/C++/Java 的多緒開放式程式開發環境

本文乃以 forth 的語法為主，但 forth 語法低階，使用不易，因此未來希望可以為 C/C++/Java 等主要的程式語言加入本論文所探討的多緒開放式程式開發環境。

2. GUI 介面的支援

目前整個多緒開放式程式開發環境暫時仍為一文字介面的系統，因此在使用上的便利性仍不及擁有視窗介面的整合開發環境，因此後續工作是為其加入視窗使用介面。這不但可以提高系統的易用性，而且還可以讓不同的執行緒擁有各自的執行視窗，以增進程式測試或除錯上的便利性。

參考文獻

- [1] 劉勉志, 盧豎昌, 廖賀田, "視窗式程式寫作之 Forth 系統", 資訊管理展望, vol.2, no.1, 1999 Jul, pp.35-46。
- [2] 江文傑, 設計並實作一多人視覺化程式發展環境, 交通大學資訊工程學系學位論文, 1997。
- [3] Adele Goldberg, *Smalltalk-80, The Interactive Programming Environment*, Addison-Wesley, 1984。
- [4] Chet Ramey, Bash Reference Manual, http://www.gnu.org/manual/bash-2.02/html_node/bashref_toc.html, 1998。
- [5] Elizabeth Rather, *Programming Languages – Forth*, American National Standards Institute, 1993。
- [6] Edmond J. Breen, Extensible Interactive C, <http://www.kd-dev.com/~eic/documentation/>, 1998。
- [7] John Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994。
- [8] Leo Brodie, *Starting Forth*, Prentice-Hall, 1982。
- [9] Mark Lutz, *Programming Python*, O'Reilly & Associates, 1996。
- [10] Masaharu Goto, CINT Reference, <http://root.cern.ch/root/Cint.phtml?ref>, 1998。
- [11] Paul Graham, *ANSI Common LISP*, Prentice Hall, 1995。