Mini-Triangle Compiler Extension

Submitted [April 2017], in partial fulfillment of

the conditions for the award of the degree **[Bachelor of Science].**

Jinming YAO

4256249

Supervisor: Dr Henrik Nilsson

School of Computer Science

University of Nottingham

I hereby declare that this dissertation is all my own work, except as

indicated in the text:

Signature _____

Date _____/_____/_____

Abstract

Nowadays, there are many programming languages existed in the world and they are used in extensive fields to solve various problems. Every programming language has its own characteristic and performs well at some specific areas due to that. However, almost all languages have the similar back-end because most of the programs run on the similar hardware platform. This is due to compiler's contribution to translating the source code to binary code which can be recognized by hardware.

Multithreading programming technology is one special feature, which has been wildly used in many fields. It supports a concurrent program to execute two or more tasks parallel to improve the efficiency. Now many languages have supported multithreading like most famous languages: C/C++, Java, and Python. The basic multithreading features provided in above language are to start, wait and cancel a thread. Apart from these basic features, there are also some functions to deal with the synchronized problem.

To explore how multithreading program executes within the process, we are going to extend an existed language ***Mini-Triangle**** with multithreading feature in this dissertation. In more details, the main aims of the project are to design multithreading feature, add the new feature into the ***Mini-Triangle*** and rewrite the related compiler and interpreter to support execution. At the end of the dissertation, some classic concurrent problems wrote in the ***Extended Mini-Triangle*** will be provided.

(Keywords: compiler, interpreter, multithreading language and parallel programming)

Mini-Triangle*: Triangle is a Pascal-like language, but generally simpler and more regular. Its commands are similar to Pascal's, but for simplicity, there is only one conditional command and one iterative command. Unlike Pascal, Triangle has a let-in command with local declarations [Watt && Brown, 2004]. As for the Mini-Triangle used in this project, this is a small version of Triangle language and the base aspects like* ***calculation****,* ***procedures, functions, arrays, and records*** *have been achieved. The details about the basic language will be provided in the supplement.*

Contents

Chapter 1

Introduction

1.1 Project Description

The intention of this project is to extend ***Mini-Triangle*** with multithreading feature, which means to extend the origin compiler and its interpreter (The basic compiler and interpreter is my solution for G53CMP coursework part two). More detailed, this has three main stages: design multithreading grammar feature, modify the basic compiler to compile new feature and rewrite interpreter to run multithreading code. The extended language not only provides some functions which allow programmers to create, wait and cancel thread, but also support mutual exclusion functionality.

The most important parts of the project are the compiler and interpreter, which support the program wrote in the extended language to compile and execute. The compiler compiles well-written code to binary code and reports any language error when there has any grammar error in source code. The interpreter takes the binary code compiled by the compiler as input, executes it and shows correct input and output as the source code does. These two parts make up the extended language.

1.2 Motivation

The motivation of this project is to extend the ***Mini-Triangle*** with multithreading feature and then use the extended language to write some classic concurrence program. Through the dissertation, we want to give the reader basic idea of how multithreading work in compiler and interpreter. In addition to this, there has another important point brought to the readers is that how a programming language evolve.

1.3 Background Introduction

1.3.1 Compiler Introduction

The main work in this project is related to the compiler, the following part will give a brief introduction of compiler knowledge used in this project. However, more details about compiler architecture would be given in Appendix.

What is a compiler and what it does?

"A compiler is a computer program (or a set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language), with the latter often having a binary form known as object code. The most common reason for converting source code is to create an executable program."

[From Wikipedia]

How does compiler work?

As mention above, the work that compiler does is to do the transformation from source code to target code. This process in the most compiler is mainly broken into four stages *[Cooper && Torczon, 2012]*:

Scanner:

Most programs are written in the text file, which means a program to a compiler is a sequence of characters. In this stage, the compiler will scan the sequence of characters and match specific character sequence into the language token (like the keywords: ***if, then, else...***). This stage is not necessary but improves the efficiency of the parser because it reduces the complexity of source code sequence.

Parser:

After processed by the scanner, the source code has been translated to a sequence of tokens, which has been well defined in language grammar. In this stage, the compiler will parse the token sequence to an **abstract syntax tree**. In this project, we use happy parser [Appendix B: Happy Parser] to help us to form the tokens, which is a parser generator for Haskell.

Type checker:

After processed by the parser, the abstract syntax tree of the program has been built. However, we still need to do type checker to make sure the program does not have any type error. In this stage, the compiler focus on doing the contextual analysis to avoid program type error like assign an integer value to a character variable.

Code generator:

After the process of type checker, the abstract syntax tree has been well typed and most language error has been excluded. Therefore, we can generate code according to the abstract syntax tree.

1.3.2 Multithreading Introduction

Multithreading is the new feature we want to achieve in the extended language. To extend such feature base on Mini-Triangle, not only the knowledge of the compiler is needed but also the multithreading architecture. Although multithreading technology has been widely used and is familiar to most computer professionals, this part will still give a base description of the multithreading technology used in the dissertation.

What is a multithreading?

"In computer architecture, multithreading is the ability of a central processing unit (CPU) or a single core in a multi-core processor to execute multiple processes or threads concurrently, appropriately supported by the operating system. This approach differs from multiprocessing, as with multithreading the processes and threads share the resources of a single or multiple cores: the computing units, the CPU caches, and the translation lookaside buffer (TLB)."

[From Wikipedia]

1.4 Description of the work

As described above, the whole work in this project is to modify compiler and reconstruct the virtual machine model of the interpreter for the extended language and it can be divided into several more detailed parts in the following:

Grammar part:

The work in this part is to generate multithreading feature grammar for Mini-Triangle. The aim of this part is to provide a suitable multithreading grammar for next two parts. Therefore, there may be several kinds of possible grammar being come up. The base idea of the work content is to imitate grammar from other languages like C/C++, Java and other programming language support the multithreading feature.

Compiler part:

The work in this part is to compile the source code to the machine code defined in Interpreter part. It has four stages: Scanner, Parser, Type Checker, and Code Generator, which is same as the origin Mini-Triangle compiler. Most of the modification will be focused on Parser, Type Checker, and Code Generator because the new features in these stages are different from the existed compiler. In grammar part, the multithreading language features can be added to the grammar in several ways, so we need to choose the suitable one to implement.

Interpreter part:

The work in this part is to reconstruct the interpreter to execute the machine code compiled by above compiler. The origin interpreter has already supported to execute the program well in a single thread, what need to be done here is to add multithreading feature. The basic idea in this part is to achieve multithreading with less modification. With the knowledge of compiler and multithreading architecture, we may need to implement a simple thread control block (TCB) and thread context switch in the interpreter. And the stack model also needs to be redesigned for multithreading.

Chapter 2

Design

2.1 Language grammar

The ***Mini-Triangle*** used in this project has already support to define function and procedure. Currently, the most important work to do is to design multithreading part grammar to accept function or procedure as argument and then create a new thread to execute these operations. Therefore, we are going to consider how other languages define multithreading interface and then define a suitable grammar for Mini-Triangle. The following parts contain several temporary designs from other language and simple evaluation.

2.1.1 Java Multithreading

```java
public class Main {
    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        myThread.start();
    }
    public static class MyThread extends Thread{
        @Override
        public synchronized void run(){
            while(true){
                System.out.println("Hello from MyThread!");
            }
        }
    }
}
```

To write java concurrent program, we need to implement our procedure in a class extends from Thread class or implements of Runnable interface *[Thread, Oracle]*. The both two ways use Java thread library and act like an object. However, Mini-Triangle does not has such library and object concept in current design but we are now aiming to the language grammar.

The most important thing we can learn from Java is that we should package our multithreading part commands. However, the procedure defined in Mini-Triangle has already achieved above feature. Therefore, we have two choices: one is using the built-in procedure directly or defines a new kind of command sequence.

In order to distinguish between simple procedure and thread procedure, we decide to choose the second plan at this time. It will have a new kind of command sequence call thread, which is similar to procedure definition but it is only used for the definition of the thread.

Declaration -> proc *Identifier* (*ArgDel*) *Command*

thread *Identifier* (*ArgDel*) *Command*

```
proc foo(x : Integer)
let
    var a : Integer;
    var b : Integer
in
    putint(x + a + b)
```

```
thread foo(x : Integer)
let
    var a : Integer;
    var b : Integer
in
    putint(x + a + b)
```

2.1.2 C Multithreading

```c
#include <stdio.h>
#include <pthread.h>

void* foo(void* arg){
    while(1){
        printf("Hello from foo thread!\n");
    }
}

int main(int argc, char *argv[]){
    pthread_t id;
    pthread_create(&id, NULL, foo, NULL);
    pthread_join(id,NULL);
}
```

In C concurrent program, the concurrent commands are written in a procedure with a single argument, then use multithreading function from pthread library to create the new thread to execute concurrent commands and also can wait for some thread with specific id assigned from the system while Java uses the object to represent the thread. For Mini-Triangle, there already has two base type: integer and character. We choose integer to represent the specific id of the thread here.

Command -> threadStart (*Expression1*, *Expression2*)

Expression1 is an integer variable to represent the id of thread and Expression2 is a thread procedure or simple procedure. As for id of the thread, we now have two way to assign value: one is to assign the id by users and the other one is to assign by thread control block.

Besides, the C multithreading program also supports to pass a single variable from the main thread to the new thread in the function which create the thread, but the above threadStart function not. Therefore, the grammar should be modified to support above functionality.

The following definition comes from C Multithreading, the threadStart function does like C's pthread_create function, takes the procedure expression and a corresponding argument expression and then pass the argument to the procedure.

Command -> threadStart (*Expression1*, *Expression2, Expression3*)

Expression3 is the argument of Expression2, the other expressions are same as above definition.

2.1.3 Haskell Multithreading

```
module Main where

import Control.Concurrent


foo :: IO ()
foo = putStrLn "Hello from foo thread" >> foo

main = do
    id <- forkIO foo
    threadDelay 5000000
    killThread id
```

In Haskell concurrent program, it uses the function forkIO in Control.Concurrent library to create a lightweight thread. In our knowledge of Haskell, we know its type system is very strict. Therefore, we consider we can add this feature to our threadStart function.

In the above diagram, the argument of *forkIO* is *foo* with type IO(). But we can create an another function *foo2* with type a -> IO(), then pass the variable to *foo2* and use *forkIO* to create the thread. In my understanding of Mini-Triangle language grammar, the procedure call is a kind of command. Therefore, we can change the argument in threadStart to command directly, which means we pass variable when we pass the command to threadStart function.

Command -> threadStart (*Expression1*, *Command1*)

Command1 is the concurrent command to run in the thread created by threadStart function. The other expression is same as above definition.

2.1.4 Basic Multithreading Functions

In above other language multithreading examples, some models of thread definition and its threadStart function has been put forward. However, there also has some other multithreading functions to use the above thread procedure like **kill** and **join** in C/C++ multithreading. These functions help programmers to scheme the threads. Therefore, we want to add the similar functionality in our extended language. The multithreading functions we are going to implement is **threadStart**, **threadWait** and **threadCancel**:

threadStart is a function to create a thread and execute commands; it has three arguments: id of the thread, thread identifier and single argument.

threadWait is a function to block the current thread until some specific thread finish; it has one argument: id of the thread.

threadCancel is a function to kill some specific thread, is has one argument: id of the thread.

After the multithreading related functions were decided, then we need to consider how these functions implement in our compiler. Base on our knowledge of compiler, we now have two different way to implement the multithreading, one is to define the functions in language grammar level and the another one is in library level. The details of the difference of two ways will be given in implementation part.

2.1.5 Synchronized Functions

After achieving above functions, the language has been able to support to run a simple concurrent program. However, there has another important point in concurrent programming, which is to do communication between threads like sharing variables. Doing communication between threads is not only to share the data structure simply but also need to make the program access the correct data structure at the correct time. In C/C++ multithreading program, it provides MUTEX lock and semaphore lock to guarantee the correct access right of each thread *[Himanshu, 2012]*. This feature can also be found in Java and Haskell multithreading programming. If the above multithreading functions are considered as the foundation of writing a multithreading program, then the following functions are used to writing a well synchronized program.

Therefore, this feature will also be added in the extended language. However, it needs a special data structure to support lock operation. Like MUTEX lock in C, the lock and unlock operation use a variable typed with **pthread_mutex_t**, which is used to store the status of the mutual lock. But in current Mini-Triangle, there are only integer and character which had already been implemented. Therefore, we need to implement such a special data type or use existed type to represent. In essence, the special data structure only stores the status of the mutual lock: lock or unlock. Thus we decide to use an integer value to represent it like Boolean value in C: 0 means UNLOCK while other value means LOCK. Base on above design, we are going to implement two more multithreading function:

threadLock is a function to block the current thread until the mutual lock is available, lock the mutual lock when it is available and then execute next command.

threadUnlock is a function to unlock a mutual lock, generate warning when unlock a mutual lock not locked.

After achieve above mutual lock function, the extended language is well enough to write a concurrent program like doing number calculation with same variable by two or more threads without invalid calculation when mutual lock functions were well used. But some other synchronized program like producer && consumer problem may need more multithreading functions. Although it can be achieved with the simple mutual lock, it would be pretty complex in algorithm design.

With the experience of writing producer && consumer problem in C, we decide to design another two multithreading function **semPost** and **semWait** like semaphore library in C. These two functions are similar to threadLock and threadUnlock function but have differences on the special variable. The special variable for the mutual lock is used to represent the status of the lock while the variable for semPost and semWait is used to represent semaphore. When the semaphore counter is 0, semWait function will block the current thread until the counter of the semaphore increase. Otherwise, the semaphore will decrease by 1 and continue to execute next command. Thus we define the following two semaphore functions:

semPost is a function to add a semaphore to the given variable who represent semaphore.

semWait is a function to decrease semaphore counter by 1; when the counter of the semaphore is 0, this function will block the thread until the counter increase.

2.2 Interpreter Instruction

The multithreading functions need to implement has been listed in the last section, this part will explain the new interpreter instructions used in implementation part for each multithreading function. Besides, the multithreading model will also be related when design interpreter instruction.

2.2.1 threadStart

The **threadStart** function is the entrance of concurrence program, which is used to generate new thread and the related information of the thread. As described in the design section of **threadStart** function, this function takes necessary information like thread identifier and concurrent code, then used these to create a correct thread.

Base on the origin interpreter model, there has a variable called program counter (**PC**), which is used to tell the interpreter which instruction is going to execute. Therefore, the new thread will also have its own program counter, which is different to the main thread. Besides program counter, there are still some variables who acts like program counter and these variables should also be processed as same as the program counter.

What is more, what the **threadStart** operation does is like procedure call in Mini-Triangle, but the difference is that **threadStart** operation does more multithreading operation. Therefore, we consider designing a new interpreter instruction **THREADSTART** for threadStart operation. This instruction will read the necessary data from the stack, copy non-shared variables for the new thread and then add the new thread to thread control table.

After the new thread was created, the main thread needs to continue its next instruction. But there has another problem here: the new thread will come back to the threadStart operation point because we use the built-in procedure to define the concurrent part code. So the new thread will execute the instructions below **THREADSTART** while it should be halted. Therefore, we add another instruction **THREADHALT,** which has same functionality to **HALT** but aim to the thread expected the main thread.

With above design, **THREADHALT** should always appear after **THREADSTART** for the new thread to halt but the main thread should skip **THREADHALT** because it is for the new thread. The solution to above situation is that we generate a new label here for the main thread to jump after **THREADSTART** instruction. Therefore, these two instruction work like following:

THREADSTART *label*	id, ca, ... -> ...	Create a thread to execute code from label ca and assign thread identifier to id variable. Then the thread executes this command jump to *label* to execute the rest commands.
THREADHALT	... -> ...	Halt current thread and do garbage collection on its stack or others.

2.2.2 threadWait

The **threadWait** function is used to block the current thread until the target thread halt. The work this function does is to check thread control table whether the target thread existed. To achieve doing such operations on thread control table, we create a new interpreter instruction **THREADWAIT**. The other information needed to execute this instruction is the identifier of the target thread and we need to evaluate the identifier and push it to stack in advance. Then **THREADWAIT** instruction read the identifier, check and decide to wait or continue next instruction. Here we use a loop of checking thread control table to represent wait process: while the target thread existed then jump back to prepare next **THREADWAIT**. Therefore, the **THREADWAIT** instruction work like following:

THREADWAIT *label*	id... -> ...	Check whether some thread existed in thread control table; If the thread existed then jump back to the *label* set before this instruction to block current thread, else continue next instruction. Throw error when waiting for itself.

2.2.3 threadCancel

The **threadCancel** function is used to kill a thread except for main thread. The work this function does is to check whether the target thread existed, halt the target thread if it existed in thread control table, else generate an error message. Therefore we create another interpreter instruction **THREADCANCEL** to check and kill the thread. Before executing this instruction, the identifier should be evaluated and push to stack and then thread control block knows which thread is going to be killed. Therefore, the **THREADCANCEL** instruction work like following:

THREADCANCEL	id... -> ...	Check whether some thread existed in thread control table; if the thread existed then kill the thread by the control block, else do nothing. Throw error when try to kill the main thread.

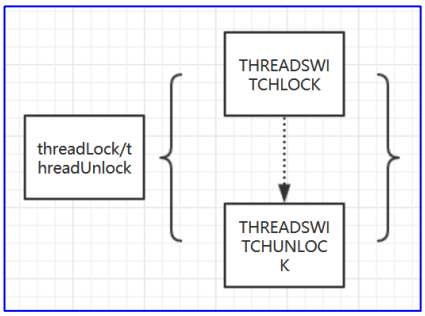
2.2.4 threadLock && threadUnlock

The threadLock and threadUnlock are used in concurrence program when two or more thread tries to access the critical sections at the same time *[Pratt && Zelkowitz, 2001]*. These functions are used to guarantee there are not more than one thread access the critical sections at the same time. Because there may have concurrent problems when two or more threads do modification in the critical section: some modification may not be saved.

There has a classic critical section problem: there has a variable contains an integer and its initial value is 0, two threads do the calculation on this variable and each thread add one to the variable ten times, what is the variable at final? Absolutely, the variable can be any value between 10 and 20 because doing the calculation is not unit operation in the program and part calculation may not be saved. There are many solutions to solve such critical section problem: like Peter's solution and mutual lock. But to solve this critical section problem, the most easiest way is used the mutual lock to scheme threads.

From above critical section problem, we know the result caused the problem is that doing the calculation is not unit operation and it has several steps: read value to a template register, add one and then re-assign the value to the variable. However, the problem will not exist when such calculation becomes a unit operation. Therefore, our threadLock and threadUnlock function should also be a unit operation. To make these operations unit, there will a special register added to TAMState, who contains a Boolean value and thread switch can only happen when its value is false. To modify the value in this register, we have two new instruction **THREADSWITCHLOCK** and **THREADSWITCHUNLOCK**.

THREADSWITCHLOCK	... -> ...	Set the switch lock to true
THREADSWITCHUNLOCK	... -> ...	Set the switch lock to false



The thread does not be switched between threadLock and threadUnlock because the switch lock is enabled at the beginning and disable after the operation is finished.

Then the content of **threadLock** and **threadUnlock** between switch lock should be considered. From the definition of **threadLock,** it is known that **threadLock** is similar to **threadWait**, both of them block the current thread until meeting the special condition. With this similarity, the structure of **threadWait** can be used here directly and just modify on the judgment condition. The **threadWait** takes identifier as input while **threadLock** use mutual lock variable. Therefore, we design another instruction **THREADLOCK** for threadLock operation. It is same to **THREADWAIT**, **THREADLOCK** also needs a label to jump back to achieve block functionality.

THREADLOCK *label*	lock,address... -> ...	Check whether the mutual lock is available; if it is available then lock it and continue next instruction, else unlock switch lock and jump back to the *label* set before this instruction to block current thread.

The **threadUnlock** operation is not complex as **threadLock**, the work **threadUnlock** does is to unlock the mutual lock and generate a warning message if try to unlock a mutual lock who is not locked. Therefore, we add **THREADUNLOCK** to the interpreter to achieve threadUnlock operation.

THREADUNLOCK	lock,address...->...	Check whether the mutual lock is available; if is not available then unlock it, else generate a warning message.

2.2.5 semPost && semWait

The content of these two semaphore functions is same as **threadLock** && **threadUnlock**: check the condition and then decide to block or not. Therefore, these two functions are decided to implement in library level with above already defined interpreter instructions. Because it is no need to implement two similar functions in the same level.

2.2.6 sleep

The sleep function is widely used in many languages to block current thread for a while. It will also be implemented in the extended language with new instruction **THREADSLEEP**.

THREADSLEEP	t...->...	Block current thread for t millisecond and then continue next instruction.

Chapter 3

Implementation

3.1 Compiler Part

3.1.1 Why implement in language level?

In design part, how concurrent function being implemented has been mentioned and we think there has two way to implement the concurrent function: one is implement concurrent in language level and the other one is library level. The first way needs to do the modification on language grammar as shown in the design part, the concurrent functions become keywords in Mini-Triangle and will be processed by the compiler. The second way needs to write a concurrent library and we do not need to change a lot on compiler because the concurrent functions are written in the library, which is as same level as functions like *getint* and *getchar.*

Before implementation, both two approaches seem possible, but we decide to choose the first one because the existed compiler does not yet support passing procedure in the procedure call. We still need to extend the type system of Mini-Triangle, but choosing the first plan gives us more room to do the modification on the compiler, which make the implementation easy and clear.

3.1.2 First implementation of threadStart

Command -> threadStart (*Expression1*, *Expression2*)

The threadStart function is the core of multithreading, the current design is mainly focused on creating and running the thread. Therefore, it is not strict with the procedure format and we assume all procedure we use in this implementation does not have arguments.

Implementation of Scanner&&Parser Part

First, we need to extend Abstract Syntax Tree with **threadStart** data structure. There have id expression and procedure expression inside its data structure:

```
| CmdThreadStart {
      ctsId     :: Expression,
      ctsProc   :: Expression,
      cmdSrcPos :: SrcPos
}
```

Second, add token **ThreadStart** for the terminal syntax **threadStart** and then modify the scanner to be able to scan **threadStart**.

Finally, we need to modify the happy parser to parse the command to the correct data structure:

- Add token declaration

```
THREADSTART { (ThreadStart, $$)}
```

- Add CmdThreadStart parser:

```
| THREADSTART '(' expression ',' expression ')'
    { CmdThreadStart {ctsId = $3, ctsProc = $5, cmdSrcPos = $1}}
```

Implementation of Type checker

To implement the type checker of the threadStart, we need to create the type rule for the threadStart first. As mentioned above, the current threadStart command takes an Integer and a procedure identifier as arguments, so the rule of the threadStart could be written like following:

$$\frac{\Gamma \vdash e1 : Integer, \Gamma \vdash e2 : Void}{\Gamma \vdash threadStart(e1, e2)} \mathrm{T} - ThreadStartCommand$$

Expression e1 is an integer, expression e2 typed with void has zero arguments.

According to the above rule, the type checker for threadStart:

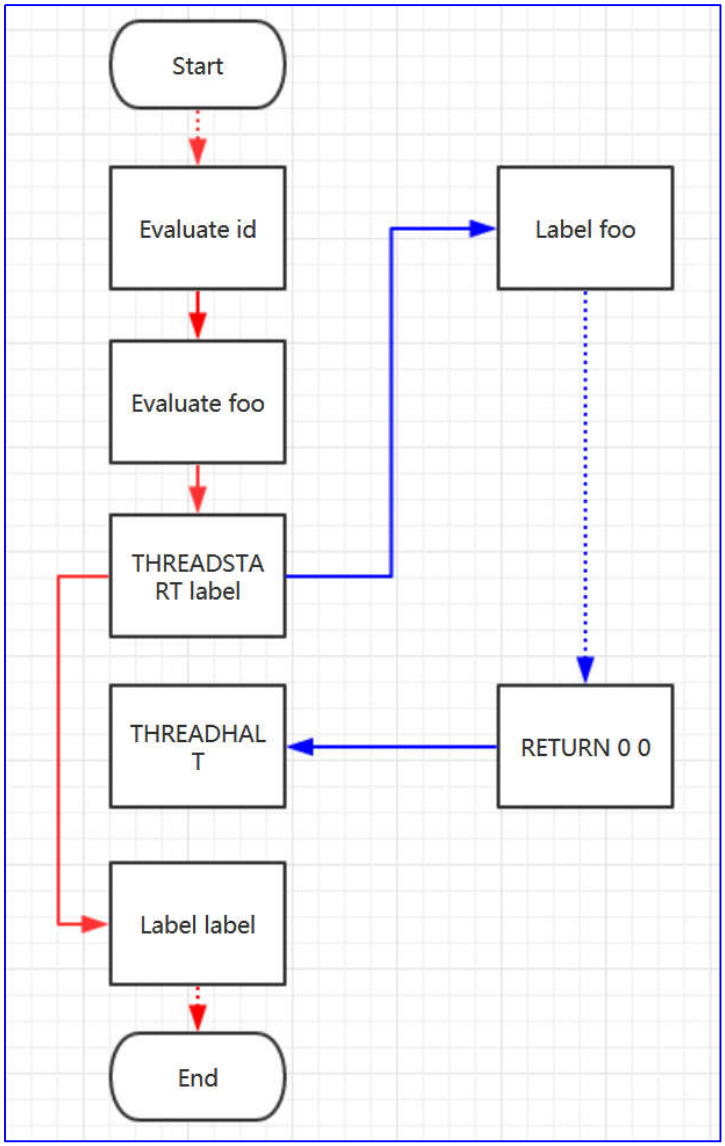
```
chkCmd env (A.CmdThreadStart {A.ctsId = c,A.ctsProc = cp,A.cmdSrcPos = sp}) = do
    c' <- chkTpExp env c Integer
    (s, cp')   <- infNonRefTpExp env cp
    case s of
        Arr ts t -> do
            require (length ts == 0) sp
                    ("Thread proc require 0 argument!")
            require (t == Void) sp
                    (notProcMsg t)
            return (CmdThreadStart {ctsId = c',ctsProc = cp',cmdSrcPos = sp})
        _        -> do
            return (CmdThreadStart {ctsId = c',ctsProc = cp',cmdSrcPos = sp})
        where
        notProcMsg t = "Not a procedure; return type is " ++ show t
```

Implementation of Code generator

The final step in compiler part is to generate code from abstract syntax tree. In the design part, we have added a new machine instruction called (**THREADSTART** *label*), which is used to create the thread according to the top value in the stack. But we do not consider how this machine instruction work and only focus on generating the machine instructions.

THREADSTART *label*	id, ca, ... -> ...	Create a thread to execute code from label ca and assign thread identifier to id variable. Then the thread executes this command jump to *label* to execute the rest commands.

Assume we are going to execute command **threadStart(id,foo)** now: the first step is to evaluate expression **id** and **foo**, then generate a new label for **THREADSTART** to jump after the new thread being created. In the new thread, it receives the instruction index from THREADSTART instruction and executes from this index, which is similar to the procedure call in existed language. Meanwhile, it will jump back after all instructions in **foo** function being executed. At this case, this thread should terminal rather than continue executing the instructions followed by **THREADSTART**. Therefore, we should add another new instruction **THREADHALT** after **THREADSTART** to terminate the thread.



threadStart

Note: red means main thread, blue mean new thread.

```
execute majl env n (CmdThreadStart {ctsId = c,ctsProc = cp}) = do
    lblJump <- newName
    evaluate majl env c
    evaluate majl env cp
    emit (THREADSTART lblJump)
    emit (THREADHALT)
    emit (Label lblJump)
```

3.1.3 Second implementation of threadStart

Command -> threadStart (*Expression1*, *Command*)

Communication between threads is important in the concurrent program and most multithreading language support passing initial variable to the new thread when it is created. But the first implementation of threadStart only supports to start the new thread due to its grammar definition.

But the second definition of threadStart imitated from Haskell replaces the procedure expression with the command to achieve passing variable by passing both procedure call and initial variable in a single command like the followings.

```
threadStart(id,putint100())
threadStart(id,putint(100))
threadStart(id,add(100,200))
```

Besides the procedure call, other commands like assigning value and even whole let-in frame can also be put into the threadStart function.

```
threadStart(id, x := 100)
threadStart(id, let
                    var i : Integer;
                    var j : Integer
                in
                    putint(i+j)
            )
```

Implementation of Scanner&&Parser Part

Base on the first implementation, we now need to change all procedure expression to command so that new threadStart data structure can store commands.

```
| CmdThreadStart {
      ctsId     :: Expression,
      ctsCmds   :: Command,
      cmdSrcPos :: SrcPos
}
```

With the change of the threadStart, we also need to modify happy parser to parse the command to the new data structure.

```
| THREADSTART '(' expression ',' command ')'
    { CmdThreadStart {ctsId = $3, ctsCmds = $5, cmdSrcPos = $1}}
```

Implementation of Type checker

Meanwhile, the type checker rule for threadStart become easier than the first version and we only need to verify the integer expression and command.

$$\frac{\Gamma \vdash e1 : Snk\ Integer, \Gamma \vdash c1 : Command}{\Gamma \vdash threadWait(e1,\ c1)}\ \mathrm{T} - ThreadWaitCommand$$

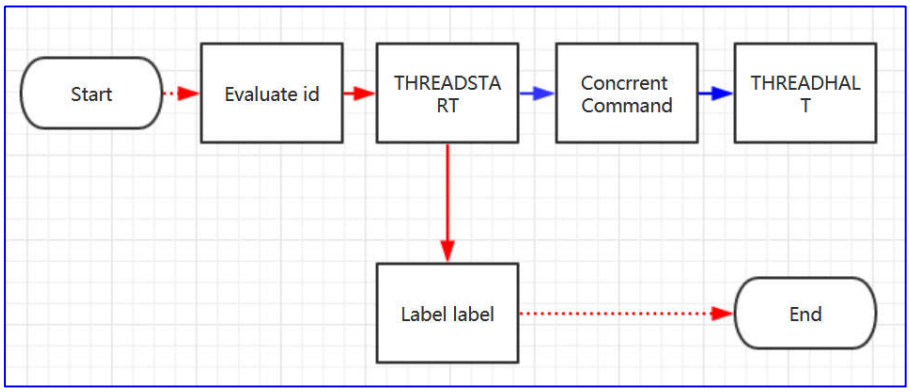
According to the above rule, the type checker for threadStart:

```
chkCmd env (A.CmdThreadStart {A.ctsId = c,A.ctsCmds = cd,A.cmdSrcPos = sp}) = do
    -- c' <- chkTpExp env c Integer
    (s, c') <- infTpExp env c                    -- env |- x : s
    (t, c'')<- sinks_nonreftype s c'             -- sinks(s,t), not reftype(t)
    cd' <- chkCmd env cd
    return (CmdThreadStart {ctsId = c'',ctsCmds = cd',cmdSrcPos = sp})
```

Above identifier type is Snk integer because the identifier is assigned by the interpreter, we can use integer when identifier assigned by users.

Implementation of Code generator

In the second version threadStart function, the thread procedure call is replaced by concurrent command. And we can keep all most structure of the first version threadStart function and replace the key part. Therefore, we can generate the code with following order.




```
execute majl env n (CmdThreadStart {ctsId = c,ctsCmds = cd}) = do
    lblJump <- newName
    evaluate majl env c
    emit (THREADSTART lblJump)
    execute majl env n cd
    emit (THREADHALT)
    emit (Label lblJump)
```

3.1.4 Implementation of threadWait && threadCancel

Command -> threadWait (*Expression1*)

Command -> threadCancel (*Expression1*)

After achieving threadStart function in the compiler, the next step is to implement other multithreading functions to arrange threads like join and stop in C multithreading. We name they **threadWait** and **threadCancel** here: threadWait is used to block the current thread until the target thread finish and threadCancel is used to kill the target thread.

Implementation of Scanner&&Parser Part

First, we need to extend Abstract Syntax Tree with **threadWait && threadCancel** data structure. There have id expression and procedure expression inside its data structure:

```
| CmdThreadCancel {
      ctcId     :: Expression,
      cmdSrcPos :: SrcPos
}

| CmdThreadWait {
      ctwId     :: Expression,
      cmdSrcPos :: SrcPos
}
```

Second, add tokens **ThreadWait** && **ThreadCancel** for the terminal syntax **threadWait** && **threadCancel** and then modify the scanner to be able to scan **threadWait** && **threadCancel**.

Finally, we need to modify the happy parser to parse the command to the correct data structure:

- Add token declaration

```
THREADCANCEL    { (ThreadCancel, $$)}
THREADWAIT      { (ThreadWait, $$)}
```

- Add CmdThreadWait && CmdThreadWait parser:

```
| THREADCANCEL '(' expression ')'
    { CmdThreadCancel {ctcId = $3, cmdSrcPos = $1}}
| THREADWAIT '(' expression ')'
    { CmdThreadWait {ctwId = $3, cmdSrcPos = $1}}
```

Implementation of Type checker

The type checker rule for threadWait and threadCancel is much simpler than threadStart and we only need to make sure the expression represent the identifier of some thread.

$$\frac{\Gamma \vdash e1 : Integer}{\Gamma \vdash threadWait(e1)} \mathrm{T} - ThreadWaitCommand$$

$$\frac{\Gamma \vdash e1 : Integer}{\Gamma \vdash threadCancel(e1)} \mathrm{T} - ThreadCancelCommand$$

Implementation of Code generator

In this part generator, we use two new machine instructions:

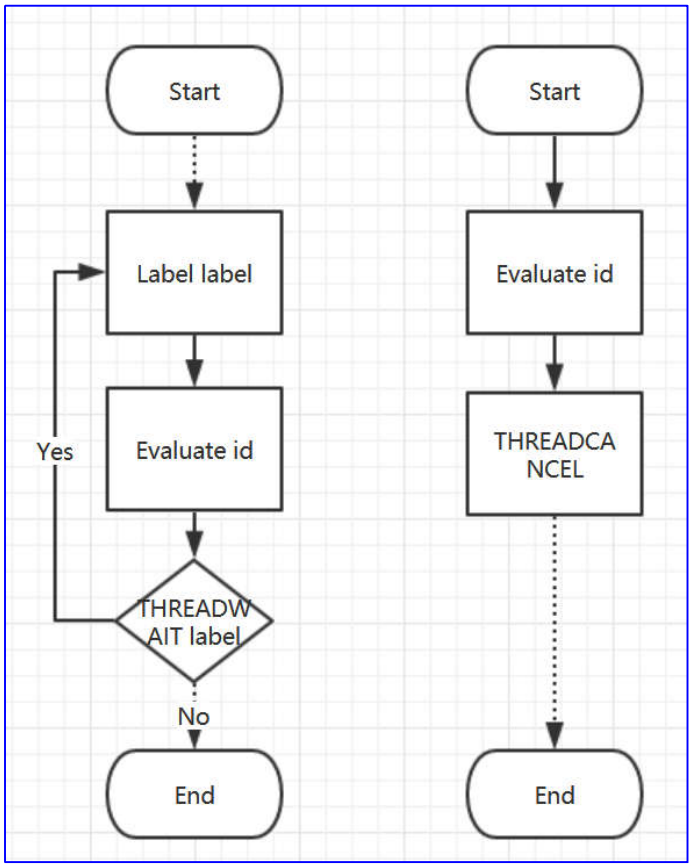
THREADWAIT *label*	id... -> ...	Check whether some thread existed in thread control table; if the thread existed then jump back to the *label* set before this instruction to block current thread, else continue next instruction. Throw error when waiting for itself.
THREADCANCEL	id... -> ...	Check whether some thread existed in thread control table; if the thread existed then kill the thread by the control block, else do nothing. Throw error when try to kill the main thread.

Assume we are going to execute **threadWait(id)** now: the first step is to evaluate the expression id, then generate a new label for **THREADWAIT** to jump back when target thread still running. This is like busy waiting in other concurrent programs, the **THREADWAIT** will check the thread control table over and over again until the target thread halt. But there has a possible deadlock that some threads are waiting for themselves, so waiting for the thread itself is not allowed. Therefore, we can generate the code with following order.

The another multithreading function **threadCancel(id)** is similar to **threadWait(id)**: the first step is to evaluate the expression id but we do not need the special label for **THREADCANCEL.** This instruction will pass the id to thread control block to kill the target thread. However, there will be nothing happen when target thread does not exist. But the program will halt when someone tries to kill the main thread, which is not allowed in the current design. Therefore, we can generate the code with following order.

```
execute majl env n (CmdThreadCancel {ctcId = c}) = do
    evaluate majl env c
    emit (THREADCANCEL)

execute majl env n (CmdThreadWait {ctwId = c}) = do
    lblJump <- newName
    emit (Label lblJump)
    evaluate majl env c
    emit (THREADWAIT lblJump)
```



threadWait && threadCancel

3.1.5 Implementation of threadLock && threadUnlock

Command -> threadLock (*Expression1*)

Command -> threadUnlock (*Expression1*)

The content of threadLock and threadUnlock has been mentioned in design part. Now we are going to implement these two function.

Implementation of Scanner&&Parser Part

First, we need to extend Abstract Syntax Tree with **threadLock && threadUnlock** data structure. There have id expression and mutual lock expression inside its data structure:

```
| CmdThreadLock {
      ctlLock   :: Expression,
      cmdSrcPos :: SrcPos
}

| CmdThreadUnlock {
      ctuUnlock :: Expression,
      cmdSrcPos :: SrcPos
}
```

Second, add tokens **ThreadLock** && **ThreadUnlock** for the terminal syntax **threadLock** && **threadUnlock** and then modify the scanner to be able to scan **threadLock** && **threadUnlock**.

Finally, we need to modify happy parser to parse the command to the correct data structure:

- Add token declaration

```
THREADLOCK      { (ThreadLock, $$)}
THREADUNLOCK    { (ThreadUnlock, $$)}
```

- Add CmdThreadLock && CmdThreadUnlock parser:

```
| THREADLOCK '(' var_expression ')'
    { CmdThreadLock {ctlLock = $3, cmdSrcPos = $1}}
| THREADUNLOCK '(' var_expression ')'
    { CmdThreadUnlock {ctuUnlock = $3, cmdSrcPos = $1}}
```

Implementation of Type checker

The type checker rules for threadLock and threadUnlock are not complex like threadStart. But there has additional point we need to consider: the mutual lock expression. In design part, we tend to use an integer to present the status. This argument is different to the integer argument in normal procedure: the program can do both reading and writing on this argument. Therefore, the type of the argument should be Snk Integer here.

$$\frac{\Gamma \vdash e1 : Snk\ Integer}{\Gamma \vdash threadLock(e1)}\mathrm{T}-ThreadLockCommand$$

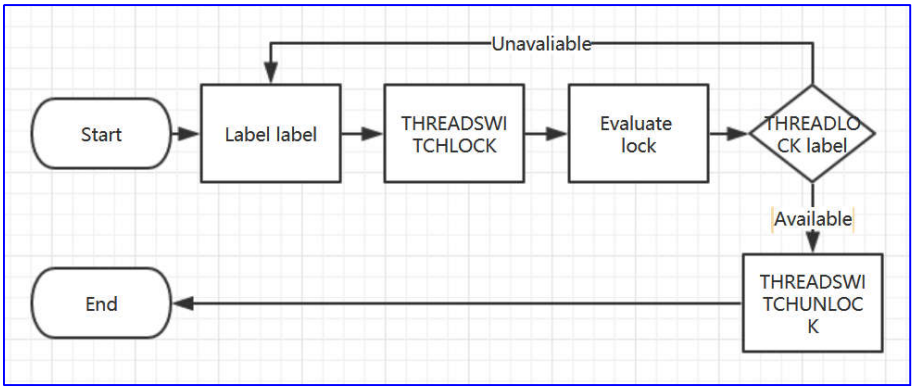
$$\frac{\Gamma \vdash e1 : Snk\ Integer}{\Gamma \vdash threadUnlock(e1)}\mathrm{T}-ThreadUnlockCommand$$

According to the above rules, the type checker for threadLock and threadUnlock:

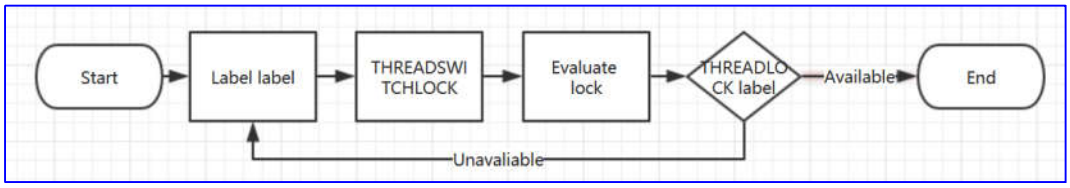
```
chkCmd env (A.CmdThreadLock {A.ctlLock = c,A.cmdSrcPos = sp}) = do
    (s, c') <- infTpExp env c                   -- env |- x : s
    (t, c'')<- sinks_nonreftype s c'            -- sinks(s,t), not reftype(t)
    return (CmdThreadLock {ctlLock = c'',cmdSrcPos = sp})
chkCmd env (A.CmdThreadUnlock {A.ctuUnlock = c,A.cmdSrcPos = sp}) = do
    (s, c') <- infTpExp env c                   -- env |- x : s
    (t, c'')<- sinks_nonreftype s c'            -- sinks(s,t), not reftype(t)
    return (CmdThreadUnlock {ctuUnlock = c'',cmdSrcPos = sp})
```

Implementation of Code generator

With the design of threadLock and threadUnlock operation, we assume we are going to execute **threadLock(lock)** here to display how code generator works with threadLock operation. Because the threadLock is a unit operation in our design, the instruction to enable and disable thread switch **THREADSWITCHLOCK** and **THREADSWWITCHUNLOCK** should be generated at the beginning and the end. Between these two instructions, the first step is to evaluate the status of mutual lock and then decide to continue or jump back depend on the status. Therefore, we can generate the code with following order.

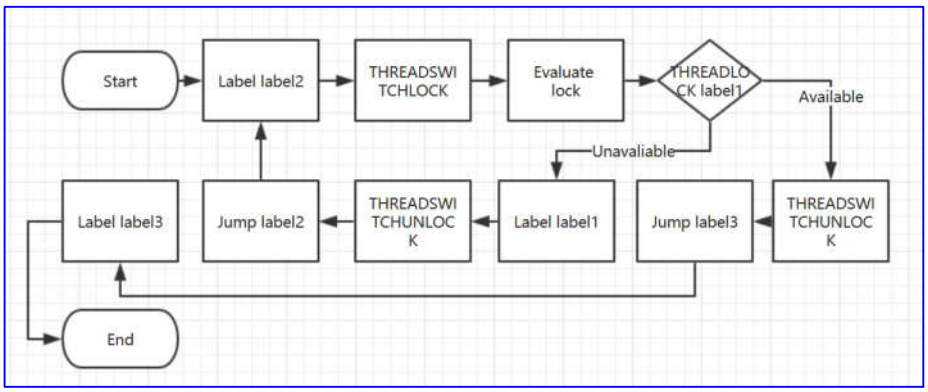


But here we find that functionality of **THREADSWITCHUNLOCK** should be merged to **THREADLOCK** operation because there will be a deadlock in above flow chart: one thread lock switch lock so other thread will not execute but this thread is waiting for a mutual lock being unlocked by other thread. Therefore, we decide to add switch unlock functionality to **THREADLOCK** instruction to avoid above deadlock problem.

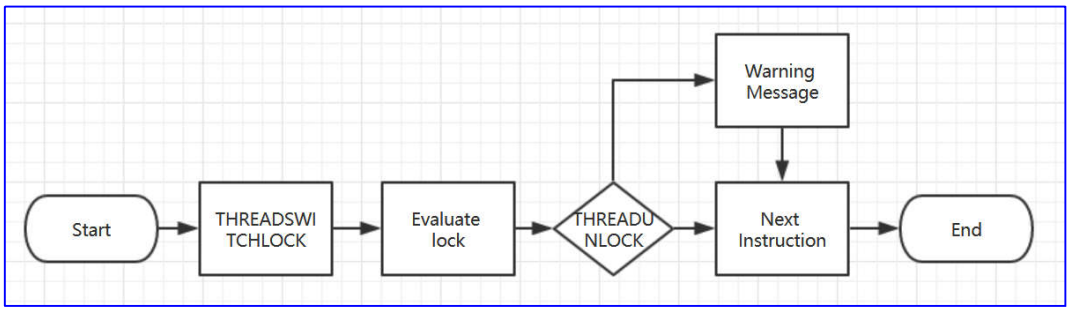



```
execute majl env n (CmdThreadLock {ctlLock = c}) = do
    lblJump <- newName
    emit (Label lblJump)
    emit (THREADSWITCHLOCK)
    evaluate majl env c
    evaluate majl env c
    emit (LOADIB 1)
    emit (THREADLOCK lblJump)
```

However, there exists other solution like following flow chart: use three labels to manage the structure of instructions. But this solution is much complex than merging two instruction's functionality. Therefore, the solution merging functionality is being chosen to generate code for **threadLock operation**.



Then we use the same method to implement the command **threadUnlock(lock).** Same as threadLock, we also merge the switch unlock functionality to **THREADUNLOCK**. Therefore, we can generate the code with following order.




```
execute majl env n (CmdThreadUnlock {ctuUnlock = c}) = do
    emit (THREADSWITCHLOCK)
    evaluate majl env c
    evaluate majl env c
    emit (LOADIB 1)
    emit (THREADUNLOCK)
```

3.1.6 Implementation of semPost && semWait

All above multithreading functions are implemented in language level, but the **semPost** and **semWait** will be different because we tend to implement these two functions in library level. That means these two functions can be used like all most normal function like *getint, putint, getchr* and *putchr*. It is better to give the explanation how interpreter instruction work and why to generate instruction like that trough implementing these two functions.

The first step to implementing these two function is to define their type: how many arguments and what type of each argument the function need and result type it returns. It is clear to know the result type is Void because these two semaphore functions do not return anything. As the argument, it should be both readable and writable, which mean the argument type is Snk Integer.

```
("semWait", Arr [Snk Integer] Void,          ESVLbl "semWait"),
("semPost", Arr [Snk Integer] Void,          ESVLbl "semPost"),
```

The second step is to consider the instruction structure: assume we are implementing **semWait(sem)**. We should evaluate the semaphore counter value first and then check whether the counter is enough to decrease one from it. Next, we need to do the comparison between counter with zero and then decide to decrease and return or jump back for next **semWait** operation. With above flow, we write semWait function like following:

```
-- sem_wait
    Label "semWait",
    THREADSWITCHLOCK,
    LOAD (LB (-1)),
    LOADIB 1,
    LOADL 0,
    GTR,
    JUMPIFZ "semFalse",
    LOAD (LB (-1)),
    LOADIB 1,
    LOADL (-1),
    ADD,
    LOAD (LB (-1)),
    STOREIB 1,
    THREADSWITCHUNLOCK,
    RETURN 0 1,
    Label "semFalse",
    THREADSWITCHUNLOCK,
    JUMP "semWait",
```

As for the semPost function, it is much simpler than semWait because it only increases the semaphore counter. Therefore, we only need write operation these between switch lock like following:

```
-- sem_post
    Label "semPost",
    THREADSWITCHLOCK,
    LOAD (LB (-1)),
    LOADIB 1,
    LOADL 1,
    ADD,
    LOAD (LB (-1)),
    STOREIB 1,
    THREADSWITCHUNLOCK,
    RETURN 0 1,
```

3.1.7 Implementation of sleep

The sleep function is used to block current thread for a while. This function is also implemented in library level. This function takes an integer *t* as the argument, which means to sleep for how many milliseconds and no return result. Here we use **THREADSLEEP** to achieve sleep functionality directly and continue executing after *t* millisecond.

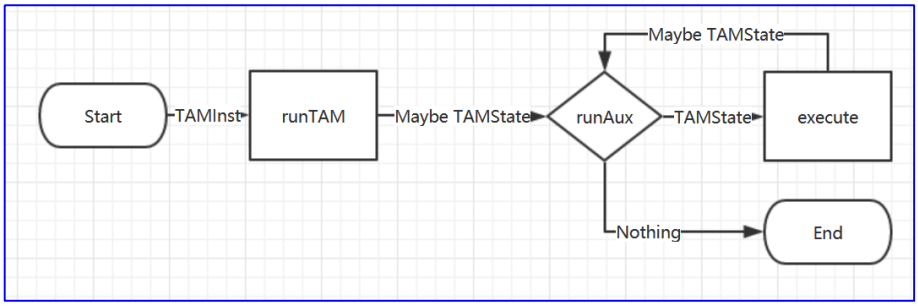
```
-- sleep
    Label "sleep",
    LOAD (LB (-1)),
    THREADSLEEP,
    RETURN 0 1
```

3.2 Interpreter Part

3.2.1 Introduction of Interpreter

The interpreter is a program to execute the machine instruction compiled by the extended Mini-Triangle compiler, which is like Java Run-Time Environment to execute Java class compiled by Java compiler. This part will give the basic introduction about how interpreter work and its new structure for running multithreading program.

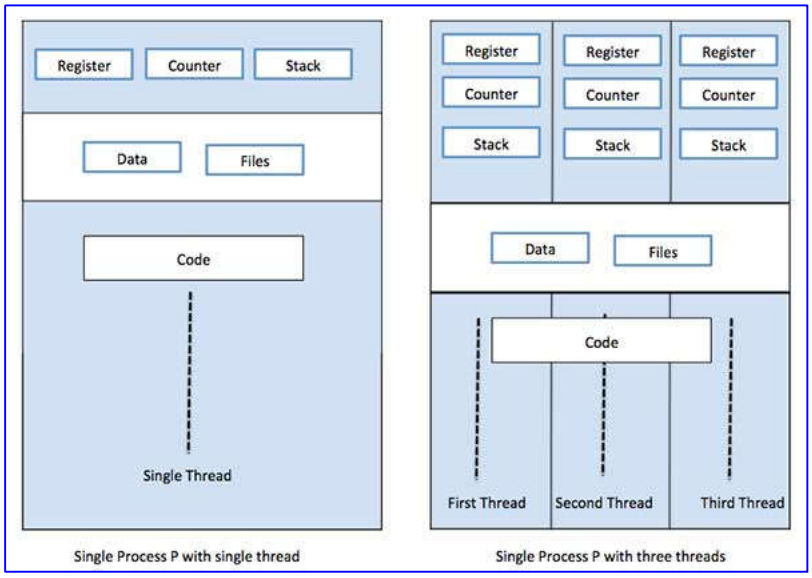
Existed Execution Structure:



The interpreter takes TAM instruction as input and then generate an initial state at the beginning by **runTAM function**. Then **runAux function** will check whether the TAMState is valuable, pass it to **execute function** to execute the instruction and then return new TAMState back to **runAux function** for next instruction. The whole execution is a loop of **runAux function** and **execute function**. However, the program break from the loop when **runAux function** receives **Nothing** returned from **execute function,** which means the program reach the end or any crash happen in **execute function**. During the execution, the TAMState hold all information used in the program including instructions, instruction counter, stack and stack's related register.

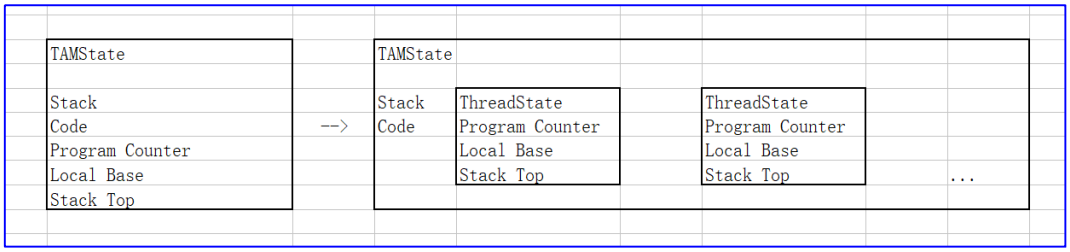
To achieve multithreading base on such structure, we has two way to implement it: one is true multithreading that means we will use existed multithreading functions in our program to achieve multithreading and the another one is not true multithreading but it is still used in current central processing unit design, it will has a thread control table to manage threads and execute each thread fro a while and then switch to another one to achieve multithreading. In the following implementation, we choose the second method to implement because it gives more sense of how multithreading work in single core computer.

The first step is to add thread control block (TCB) feature. In existed structure, the state holds all information who acts like process control block in operation system. Therefore, we decide to imitate the structure of process model in our interpreter. The following graph shows the structure of the single process with single thread and three threads *[From TutorialsPoint]*.

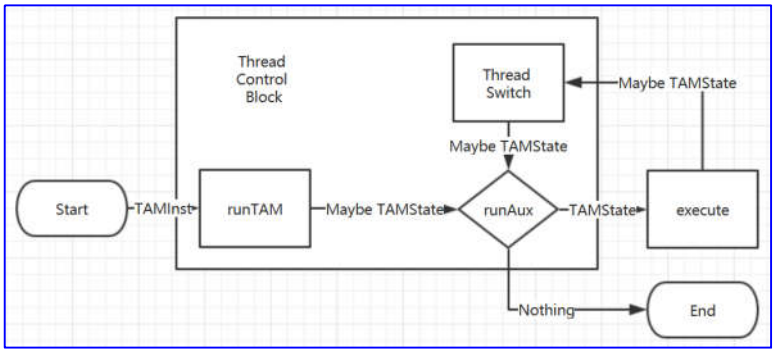


Multithreading model

Learn from above process model, we decide to add a sub-state name **ThreadState** to hold the information that is only used in the current thread and there will have multiple **ThreadState** in the interpreter state. Besides, the shared data like instructions will be stored in interpreter state directly and can be accessed by any running thread. The following model is the first version interpreter, the stack is shared with all threads.



Base on above state model, we need to re-arrange the execution structure: add thread switch functionality and thread management block. In the following graph, we can see the difference between the new execution structure and the old one: the interpreter still generate the initial state at the beginning and pass it thread management block, then the thread management block will select one thread, switch to it and execute a certain number of instructions. When there is more than one thread in the state, the switch functionality starts working: it will collect current thread data, compress them and store it in thread table, then select the next thread, decompress them and start next execution loop. The following diagram show structure of new interpreter, the detail of thread control block will be given in thread context switch implementation part.



3.2.2 Thread context switch implementation

In the last section, the structure of thread control block has been given. In this section, we are going to explain implement the thread context switch functionality.

```
data ThreadStatus = Runable |
                    Running |
                    Occupied |
                    Sleep UTCTime |
                    Dead deriving (Show,Eq)

data ThreadState = ThreadState {
    thID   :: MTInt,
    thSU   :: ThreadStatus,
    thPC   :: MTInt,
    thLB   :: MTInt,
    thST   :: MTInt,
    thStk  :: Stack
} deriving (Show,Eq)

data TAMState = TAMState {
    tsCode :: Array MTInt TAMInst,
    tsCdSz :: MTInt,
    tsCT   :: MTInt,

    tsID   :: MTInt,
    tsSU   :: ThreadStatus,
    tsPC   :: MTInt,
    tsLB   :: MTInt,
    tsST   :: MTInt,
    tsStk  :: Stack,

    tsME   :: Memory,
    tsTH   :: [ThreadState]
} deriving (Show,Eq)
```

The above diagram shows the definition of ThreadStatus, ThreadState and new TAMState implemented in Reference Stack Model. Besides some base data like Local Base and Stack Top, there have some new registers for multithreading execution.

ThreadStatus is the status of the thread, it has five kinds of status:
Runnable: thread created with Runable status;
Running: thread is running;
Occupied: this thread occupied the interpreter so thread context switch will not work;
Sleep (t :: UTCTime): this thread will not execute until the time t;
Died: this thread halt and should be removed;

ThreadState contains ThreadStatus, thID and base data:
ThreadStatus: this thread's status;
thID: this thread's id

TAMState contains current running thread's base data, memory and a thread table:
tsCT: this is a counter from 0, which is used to initiate the id for new thread
tsTH: the thread table contain non-activated threads

Base on above definition, then the requirement of thread switch function is decided. In thread switch function, it should check whether the current thread is **Occupied** because it does work when current occupies the interpreter; then collect the current thread's information and

compress; the final step is to choose a new thread and decompress. Therefore, the thread switch function is implemented like the following:

```
switchThread :: TAMState -> IO TAMState
switchThread s@(TAMState {tsID = id,
                          tsSU = su,
                          tsPC = pc,
                          tsLB = lb,
                          tsST = st,
                          tsStk = stk,
                          tsTH = th}) = do

            if su == Occupied then return s           -- check thread lock
            else do
                (h,t) <- switchThreadHelper (cThread:th)
                return $ s { tsID  = thID h,
                             tsPC  = thPC h,
                             tsLB  = thLB h,
                             tsST  = thST h,
                             tsSU  = thSU h,
                             tsStk = thStk h,
                             tsTH = t}
    where
        empty []     = True
        empty _      = False
        cThread      = ThreadState{thID = id,
                                   thPC = pc,
                                   thLB = lb,
                                   thST = st,
                                   thSU = su,
                                   thStk = stk}
```

In above code, there has a function named switchThreadHelper: this function is used to select a thread to execute. There have two methods to do the selection: one is random pick and the other one is to execute threads in circularly.

3.2.3 Implementation of new interpreter instructions

Interpreter instructions need to be implemented with interpreter model, the following instruction implementations are under second version stack model. The details of first and second version stack model will be given in next part including the result why we choose the second version stack model.

THREADSTART *label*	id, ca, ... -> ...	Create a thread to executes code from label ca and assign thread identifier to id variable. Then the thread execute this command jump to *label* to execute the rest commands.

This instruction is the core of threadStart operation, whose functionality is to generate new thread and then add the thread to thread control table. During the implementation of **THREADSTART**, the first step is to read the necessary information from stack: we only need identifier variable address in the second implementation because we need to generate a new identifier; the second step is to ask stack management for stack room for new thread; the third step is to add the new thread to thread tables; the final step is to move the program counter to the label set after concurrent code. Therefore, **THREADSTART** is implemented like the following:

```
THREADSTART name -> do
    case pop s of
        Right (idAdd , s') ->
            case (writeValue (ref,id1) (tsME s') >>=
                 (\me' -> addRefN (tsStk s') me')) of
                Right me'' ->
                    jump name (s'{tsCT  = ct + 1,
                                  tsTH  = (newThr : th),
                                  tsSU  = Running,
                                  tsME  = me''})
                Left err -> abort err
            where
                id1 = ct + 1
                newThr = ThreadState{thID  = id1,
                                     thLB  = tsLB s',
                                     thPC  = tsPC s',
                                     thST  = tsST s',
                                     thStk = tsStk s',
                                     thSU  = Runable}
                ref = tsStk s !! addrToIx (SB idAdd)
        Left err        -> abort err
```

THREADWAIT

THREADWAIT *label*	id... -> ...	Check whether some thread existed in thread control table; if the thread existed then jump back to the *label* set before this instruction to block current thread, else continue next instruction. Throw error when waiting for itself.

This instruction is used for threadWait operation to achieve busy waiting until the target thread finishes. From the threadWait operation, we know how **THREADWAIT** work: the first step is to read the thread identifier from the stack; the second step is to check the identifier in thread table, if the thread exists then jump back, else continue next instruction. However, there has another important thing that no thread can wait for itself because it must cause deadlock. So we need to add one more condition to compare the identifier to itself and halt the problem if waiting for itself happen. Therefore, **THREADWAIT** is implemented like following:

```
THREADWAIT name -> do
    case pop s of
        Right (id1,s') -> if (id == id1) then abort (waitSelf id)
                          else if (isRunning id1) then jump name s'{tsSU = Running}
                          else continueR s'{tsSU = Running}
        Left err       -> abort err
    where
        isRunning n = elem n (map (thID) th)
```

THREADCANCEL

THREADCANCEL	id... -> ...	Check whether some thread existed in thread control table; if the thread existed then kill the thread by the control block, else do nothing. Throw error when try to kill the main thread.

This instruction is used for threadCancel operation to kill some specific thread except main thread. From the threadCancel operation, we know how **THREADCANCEL** work: the first step is same as **THREADWAIT** to read the thread identifier from stack; the second step is to tell thread control block to kill the thread and ask stack management to clean job, however, this instruction won't do anything if the thread to kill does not exist or has already finished. But in our implementation of threadCancel operation, killing the main thread is not allowed so we add one more condition to do the comparison, the program will halt if try to kill the main thread. But consider in another point, the result of halting the program is same to kill the main thread because non-main threads will not continue when main thread halt. Therefore, the instruction **THREADCANCEL** is implemented like following:

```
THREADCANCEL -> do
    case pop s of
        Right (id1,s')  -> do
            if id1 == 0 then abort cancelMain
            else if (id1 == id) then continue $ do
                me' <- releaseRefN (tsStk s') (tsME s')
                return s'{tsSU = Dead,tsST = 0,tsStk = [],tsME = me'}
            else if (isRunning id1) then continue $ do
                me' <- releaseRefN (concat $ map thStk $ toCancel id1) (tsME s')
                return s'{tsTH = map (p id1) th,tsME = me'}
            else abort $ threadNotE id1
        Left err        -> abort err
    where
        isRunning n  = elem n (map (thID) th)
        toCancel id2 = filter (\t -> thID t == id2) th
        p id2 t = if (thID t) == id2 then t{thSU = Dead,thStk = [],thST = 0} else t
```

THREADSWITCHLOCK && THREADSWITCHUNLOCK

THREADSWITCHLOCK	... -> ...	Set the switch lock to true
THREADSWITCHUNLOCK	... -> ...	Set the switch lock to false

These two instructions are used for enabling and disabling the thread switch functionality in thread control block. So there should be a special register contain thread switch lock value in TAMState. However, we do not tend to add such register but add a new thread status **Occupied**. When the current thread's status is **Occupied**, then thread switch will not work until it becomes normal status **Running**, which does same work as switch lock. Therefore, these two instructions are implemented like followings:

```
THREADSWITCHLOCK   -> continueR (s {tsSU = Occupied})
THREADSWITCHUNLOCK  -> continueR (s {tsSU = Running})
```

THREADLOCK

THREADLOCK *label*	lock,address... -> ...	Check whether the mutual lock is available; if it is available then lock it and continue next instruction, else unlock switch lock and jump back to the *label* set before this instruction to block current thread.

This instruction is used for threadLock operation to check the lock status: lock the mutual lock if it is available else jump back to achieve busy waiting. From the implementation of threadLock operation, we know the lock status and its variable address has been pushed to the stack, then **THREADLOCK** needs to read these two value and do correct operation depend on the lock status: if the lock is available then lock the mutual lock through its address, else jump back. Besides, the **THREADLOCK** also has the functionality of **THREADSWITCHUNLOCK** to unlock thread switch lock. Therefore, this instruction is implemented like the following:

```
THREADLOCK name -> do
    case popN 2 s of
        Right (val:add:_,s') ->
            if val == 0 then do
                case writeValue (ref,1) (tsME s') of
                    Right me' -> continueR s'{tsSU = Running,tsME = me'}
                    Left err  -> abort err
            else jump name s'{tsSU = Running}
            where
                ref = tsStk s !! addrToIx (SB add)
        Left err              -> abort err
```

THREADUNLOCK

THREADUNLOCK	lock,address...->...	Check whether the mutual lock is available; if is not available then unlock it, else generate a warning message.

This instruction is used for threadUnlock operation to unlock the mutual lock. In threadLock operation, we used to consider to generate a warning message when the mutual lock has already been unlocked but this functionality will not be added because it is not important while other language does so. From the implementation of threadUnlock operation, we know lock status and its variable address has been pushed to the stack, then **THREADUNLOCK** needs to read these two variables, unlock the mutual if it is locked or do nothing if it is already unlocked. Therefore, the instruction is implemented like the following:

```
THREADUNLOCK -> do
    case popN 2 s of
        Right (val:add:_,s') ->
            if val == 0 then continueR s'{tsSU = Running}
            else do
                case writeValue (ref,0) (tsME s') of
                    Right me' -> continueR s'{tsSU = Running,tsME = me'}
                    Left err  -> abort err
            where
                ref = tsStk s !! addrToIx (SB add)
        Left err              -> abort err
```

THREADSLEEP

THREADSLEEP	t...->...	Block current thread for *t* milliseconds and then continue next instruction.

To achieve this function in the interpreter, we add another thread status **Sleep UTCTime**. The thread status contains a **UTCTime *t***, which means this thread continue next instruction at time ***t.*** Therefore, we only need to generate the **UTCTime** when executing **THREADSLEEP** instruction like followings:

```
THREADSLEEP        -> do
    case pop s of
        Right (msec,s') -> do
                           cTime <- getCurrentTime
                           let wakeTime = addUTCTime ((fromIntegral msec) / 1000) cTime
                           continueR (s' {tsSU = Sleep wakeTime})
        Left err        -> abort err
```

THREADHALT

THREADHALT	... -> ...	Halt current thread and do garbage collection on its stack or others.

This instruction is used to halt current thread, the work needs to be done in this instruction is to tell memory management to clean this thread's stack and set current thread's status to **Died**. Then the thread control table will remove this thread in next thread switch.

```
THREADHALT         -> continue $ do
                        me' <- releaseRefN stk me
                        return (s {tsSU = Dead,tsST = 0,tsStk = [],tsME = me'})
```

3.2.4 Connection between Interpreter and Stack Model

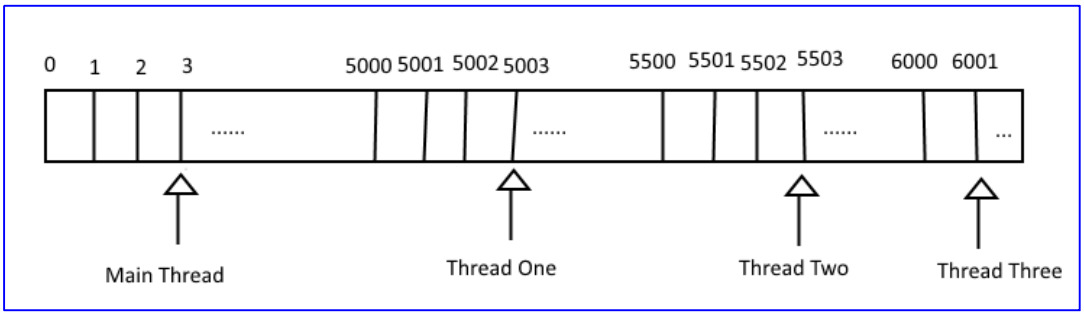
Stack model is an important part for the interpreter to keep data and all calculation and read or write operations base on the variables in the stack. However, the origin stack is not suitable for multithreading operation. If there has two threads share a stack and do a calculation, then the stack will be messy because the thread does not know which value belonged to it. Therefore, we need to create a multiple stack model *[Koopman, 1989]* for differenct thread in multithreading interpreter *[Pratt && Zelkowitz, 2001]*. The following two stack models are developed in implementation.

Apart from keeping variables, the stack model should also provide interfaces for the interpreter to access the variables: the most important two operation is **pop** and **push**. But there may have some situations that we need to pop or push multiple values, we prefer to add **popN** and **pushN** to do **pop** and **push** for multiple times. Besides above operation, there are still many function need to support interpreter. To make the connection between interpreter and stack model, some important functions need to be implemented in stack model will be listed in the following table:

Pop	Pop the top value from stack
Push *n*	Push a value *n* to stack
PopN	Pop the top n values from stack
PushN *ns*	Push n values *ns* to stack
WriteStk *n i*	Overwrite the value *n* at index *i*
WriteStkN *ns i*	Overwrite the values *ns* begin at index *i*
ReadStk *i*	Read the value at index i
ReadStkN *n i*	Read n values begin at index *i*

Besides above base functions, some stack models also have its own special function like generateNewStack in Stack Chain Model and release function in Reference Stack Model. These functions combine the interpreter and stack together to execute the program. However, the implementation of these functions are not difficult if the stack model is decided but the difficulty is the structure of stack model. So the implementation of above functions in stack model will not be including in stack model introduction and we mainly focus on the structure of stack model: how it support multithreading program.

3.2.5 First Stack Model: Stack Chain Model



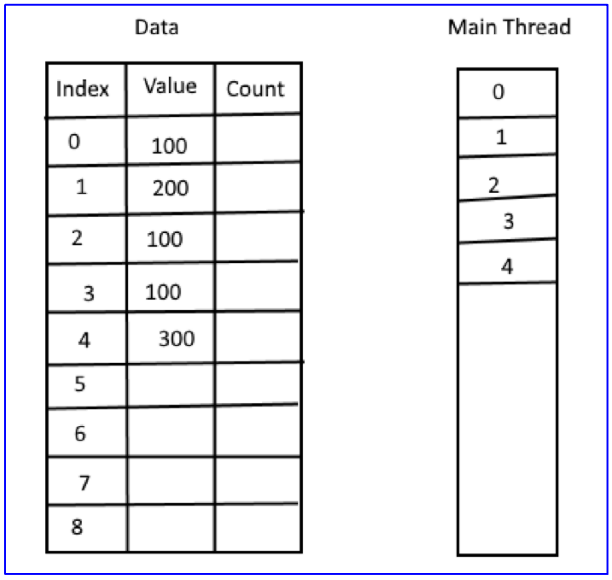
Stack Chain Model contains one single stack who has several pieces stack frames. Each thread has it own stack frame: the main thread start from index 0 while other non-main threads start from index 5000 and each thread's stack frame size is 500. This stack model works well with all most multithreading program in the design, but it has problems when threads share variables like the followings:

```
let
    var id0 : Integer;
    var x   : Integer := 100
in
begin
    let
        var y : Integer := 200
    in
        let
            proc f()
            while true do
            begin
                putint(y);
                sleep(1000)
            end
        in
            threadStart(id0,f());
    threadWait(id0)
end
```

This program creates a thread id0 to print variable y each second forever and the main thread terminals when thread id0 terminals. However, this program will crash with the stack overflow error when running on interpreter with the above stack model. Because the variable y is defined in main thread and store in main thread's stack but it will be popped when main thread execute the command **threadWait(id0).** Therefore, the thread id0 will get the stack overflow error when it tried to get the variables y.

Although the stack model has problems in concurrent communication, it is still valuable to implement to help to find other solution to solve it. In Stack Chain Model, we know the problem happen because threads share stack and one thread want to access the variable in other thread's stack but the variable has been released. Therefore, we need to consider how to manage these shared variables between threads when the thread who created them finished.

3.2.6 Second Stack Model: Reference Stack Model



In this stack model, the stack contains two parts: data structure and reference table. As shown in above figure, values are stored in data structure rather than reference table and the table in thread keeps all references to the value. In the data structure, each cell contains three attributes: index, value and reference count.

Index is a unique id for data cell, it is assigned from zero and throw memory error when running out of the data structure. When index reaches the end of the data structure, it will start from zero again to search available id.
Value is the data stored in the stack in Stack Chain Model.
Reference count trace how many times current cell being referenced and the cell will be released when reference count become zero.

The data structure was store in TAMState, which can be accessed by any running thread and each thread contains the reference table. When push value to stack, the memory management will use the first available space in the data structure, push its index to reference table and then interpreter know where should the value be stored in the data structure. As for pop operation, the first step is to pop the index from the reference table, then use the index to read the value in the data structure and finally decrease the reference counter. When reference counter become 0, the value will be removed from the data structure and its index will be set to available to the interpreter. As for other read and write operation, they are achieved through reference index to the data structure.

In implementation part, we have already mentioned the new thread will have a copy of stack of its father thread. But the fact is that new thread has a copy of its father thread's reference table when the new thread is created. Meanwhile, the reference counter in data structure will be updated with the copying of the reference table action. When thread halts, the interpreter will decrease the reference counter of the remind indexes in this thread's reference table to release space in the data structure for future use.

Chapter 4

Evaluation

4.1 Introduction of evaluation

In the following part, there will use two usual concurrent examples to evaluate the extended language. During the evaluation, there will be discussion focus on the advantages and disadvantages of the extended language's new feature.

4.2 Simple concurrent calculation

```
let
    var number : Integer := 0;
    var lock : Integer := 0;
    var id0 : Integer;
    var id1 : Integer
in
    let
        proc thread()
        let
            var i : Integer := 123
        in
            while i < 133 do
            begin
                threadLock(lock);
                number := number + 1;
                threadUnlock(lock);
                i := i + 1
            end
    in
        begin
            threadStart(id0,thread());
            threadStart(id1,thread());
            threadWait(id0);
            threadWait(id1);
            putchr('n');
            putchr('u');
            putchr('m');
            putchr('b');
            putchr('e');
            putchr('r');
            putchr(':');
            putint(number);
            putchr('\n')
        end
```

The above code is an example to implement concurrent calculation in extended Mini-Triangle. There has three thread: main thread, thread id0 and thread id1. Thread id0 and thread id1 do the calculation on the variable *number*, add one to the variable ten times. Main thread creates these two threads and waits for these two threads finish their work and then print out the final value of *number*. When we comment **threadLock(lock)** and **threadUnlock(lock)** in **thread()** definition, then the program will have concurrent problem because the critical section *number* is operated by two thread at same time: the final value of *number* can be any number between 10 and 20 and the result of execution also prove it. But we need threadLock and threadUnlock here to prevent critical section problem, the result of execution is that the value of *number* is always 20 when we use the mutual lock. Besides using mutual functionality, we also achieve Peter's solution in extended Mini-Triangle.

4.3 Producer and consumer problem

As we know the producer-consumer problem is a classic example of a synchronization problem. This problem describes two threads, the producer and the consumer, who share a common, fix-size buffer used as a queue. The producer's job is to generate data, put it into the buffer and repeat while the consumer's job is to consume the data from the buffer, one piece at a time [From Wikipedia]. This problem is to make sure that the producer does not generate data when the buffer is full and the consumer does not consume data when the buffer is empty. With the new multithreading function semPost and semWait, this problem has been achieved in extended Mini-Triangle. However, we do not focus on how producer-consumer problem was solved in extended Mini-Triangle. The most important aspect is that the extended Mini-Triangle support producer-consumer problem to execute well. If you want to know more details about how producer-consumer problem achieve in extended Mini-Triangle, it is a better way to review the source code in extended Mini-Triangle.

Chapter 5

Future work

5.1 Introduction of future work

In conclusion, this dissertation has achieved multithreading support in Mini-Triangle and some classic concurrent examples are also given in evaluation part. However, there are still some works can be done to make the language better. Some of these work may have been mentioned in design and implementation part but not been chosen to implement due to various kinds of reasons. In the following part, we are going to discuss these work, which can make the language better.

5.2 Code optimization

Code optimization is one important step in the compiler, which make the code execute efficiently. Although the program can execute without code optimization, it might be slower than optimized code. In this dissertation, the origin compiler itself contain code optimization but its optimization rule is in conflict with the multithreading feature, which makes the code optimization disable in the extended compiler. Meanwhile, the code optimization is not including in the work plan. However, adding code optimization will make the program smaller and more efficient.

5.3 True Concurrent Interpreter

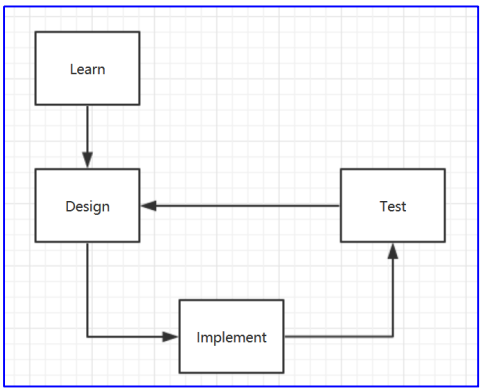
In this dissertation, the interpreter is not true concurrence: we execute each thread for a while and switch to the other one. The reason why we choose the analog concurrence has been mentioned in implementation part. The most important reason is that we do not want to make big modification on origin compiler and interpreter and then achieve multithreading support. However, true concurrence is much efficiently than the analog one. Besides, the implementation will be easier because some multithreading work will be done in the programming language (the language we use to build the interpreter). So achieving true concurrence is still meaningful for the extended language.

Chapter 6

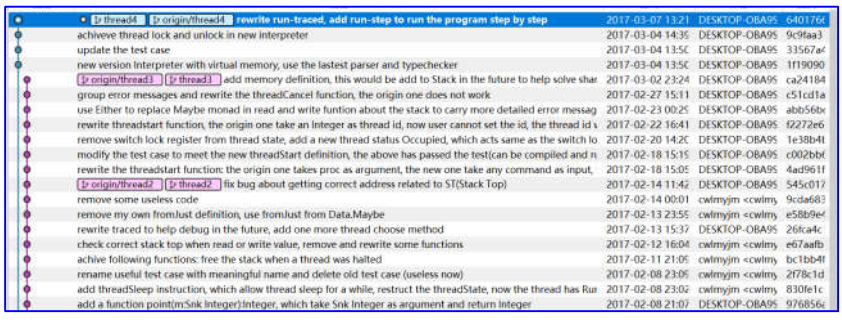
Summary and reflection

6.1 Project management

This project lasts for seven months from October 2016 to April 2017 and it can be divided into three parts: the first part is to learn fundamentals of compiling, the second part is to design the multithreading feature for Mini-Triangle and the third part is to implement the multithreading feature came up in the second part. The second part and third part of the project was carried on at the same time because once any fatal error happens in implementation part then I need to go back to the second part to re-consider the design of multithreading feature, which is a bit like V-Model in software development.



The learning part takes first three months while doing the design and implement in last four months. Within the above three parts, there are more details on work management to arrange the project. For example, there are many multithreading functions need to be implemented in this project but the most important one is threadStart, so implementing threadStart becomes the sally port in design and implementation part and once it is done then other multithreading function become much easier. Therefore, I focus on achieving the threadStart functionality in design and implementation part first. That means we need to find the sally port of the project, and use it to lead whole project's progress.



Besides time management, I also use git to manage my project code. By using git in my project, it is easy to track the modification in compiler and interpreter, which help me a lot in debug because the modification becomes much more in the final process than the beginning. Tracking the modification provides a clear flow chart to help me understand what happen in my modification history.

6.2 Contributions and reflections

6.2.1 Contributions

In this project, I achieve to extend the Mini-Triangle with multithreading feature. The extended Mini-Triangle provide the following interfaces: threadStart, threadWait, threadCancel, threadLock, threadUnlock, semPost and semWait to write multithreading program in Mini-Triangle language. I also write a simple concurrent calculation and producer and consumer problem in the extended Mini-Triangle and these programs execute well.

But above multithreading features are all based on Reference Stack Model, which solved the shared variables problem in Stack Chain Model. This is also the most difficult part of the project because it decides that concurrent program can execute without exception when threads do communication with others.

6.2.2 Reflections

6.2.2.1 The reflection on time plan:

Although the time plan has been set at the beginning of the project, but it was adjusted several times for unexpected delay of the task. This problem is caused by unfair time allocation to different work: both easy part and difficult part gains similar time intervals. However, the aim of the project was achieved at the final but the result is not satisfied because some aspects of the extended Mini-Triangle are not done in this project like code optimization and true concurrence. But I think these aspects can be achieved if the time plan of the project is better. Therefore, the time plan in future project should be considered seriously to avoid such situation happen again. To avoid it in the future, the first thing is that we should take into full consideration about the tasks in the work plan, mark the difficulty of each and then arrange them in the order of priority. Besides, the work plan also needs being updated in every development period to make sure it is not outdated.

6.2.2.2 The reflection on implementation:

Before design and implement the new feature, I have learned basic knowledge of compiler and multithreading. But it is still not enough in the process of implementation, to add multithreading feature does not only need the knowledge of compiler and multithreading but also machine architecture and more. Although I have achieved the aim of the project, the project can still be improved because there are still some multithreading problems in current interpreter. For example, the *getint* and *getchr* functions who ask users for input will block all other threads because the interpreter we implemented is not true concurrence and the thread executes in circularly. To implement true concurrence need more knowledge about computer architecture. Therefore, I realized that the base of computer language is much more complex than what I learn from school, which means that making the better project need more knowledge.

6.2.2.3 The reflection of the whole project

Although I have achieved the aim of the project, the whole project is not like the initial expectation. Most of the works in the project are allocated to last four months, which make the time not enough for some tasks. This is the most serious problem in the project, which must be improved in feature project, or it will lead serious results.

Apart from this, the another aspect I realized is that the project is individual rather than group project, which means I am the only one person to develop the project. The advantage is that I can decide most aspects in the project but the disadvantage is much more. Because it is an individual project, my friend may only know I am doing something about compiler but nothing else. Same to me, I only know my friend's project is about AI. This makes something difficult in the project, you may only solve the problem in your project yourself because your friend cannot help you without the basic knowledge. But this problem will not exist in group project because group members take part in the project together. Therefore, it is better to invite a friend to take part in your project as a listener and you do same in your friend's project through the project. This can make both your and your friend's project easier and learn more knowledge than working alone.

Besides, the another disadvantage is that is too free in the individual project. The project manager can arrange all most everything in the project. However, this is not like group project because the group manager arranges work depends on the participants. It will consider every participant's ability and let everyone do his/her best but the individual project trains every aspect. Any aspect is not good enough will influence other aspects, so we are not only to do the project but also train other aspects. And too much freedom in the project also makes the participant lazy because only yourself manage the progress. But it is not serious to people who has strong possessiveness but I am not. Therefore, I think it can be solved same as the last problem: invite a friend to supervise you and you do same to him/her. This can make both you and your friend know the progress of each other project and improve the efficiency.

Bibliography

1 David A Watt & Deryck F Brown[2004],

Programming Language Processors In Java

2 Keith D. Cooper & Linda Torczon[2012], Engineering A Compiler

3 Philip J. Koopman, Jr. [1989], Stack Computers: the new wave

4 Terrence W. Pratt and Marvin V. Zelkowitz [2001],

Programming languages design and implementation

5 TutorialsPoint, Operating System - Multi-Threading, Last accessed 30 March 2017 at https://www.tutorialspoint.com/operating_system/os_multi_threading.htm

6 Himanshu Arora [2012], How to Use C Mutex Lock Examples for Linux Thread Synchronization, Last access 31 December 2016 at http://www.thegeekstuff.com/2012/05/c-mutex-examples/?refcom

7 Oracle, Thread Library, Last access 21 December 2016 at https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

8 Wikipedia, Multithreading, Last access 1 October 2016 at https://en.wikipedia.org/wiki/Multithreading_(computer_architecture)

9 Wikipedia, Compiler, Last accessed 1 October 2016 at https://en.wikipedia.org/wiki/Compiler

10 Wikipedia, Consumer and producer problem, Last access 27 March 2017 at https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem