Programming Languages Project Report

Implementing Asynchronicity using OCaml

Kappagantula Lakshmi Abhigna - IMT2019040

July 11, 2022

Introduction

This report discusses our approach on the project 'Implementing Asynchronicity using OCaml'.

Problem Statement

The aim of this project is to implement an extension of the OCaml Language for enabling some concurrent programming features. According to our understanding, we have to implement some part of the async library from scratch. In order to clearly understand the requirements, we need to have knowledge the following concepts.

Concurrency

A concurrent program is the one which can run to produce optimal results even when it is not executed sequentially. It involves splitting a computation into concurrent pieces called 'Concurrent Modules'.

Concurrency enables the programmer to code such that even when one part of the code makes a *blocking call*, the other parts of the code continue executing. A blocking call is any function call which makes the code 'wait' for some action, i.e. user inputs, signals, etc.

For example:

- Multiple windows open in a browser
- Multiple processor cores on a single chip
- Serveral users at the same time on a website

Models for concurrent programming

- Shared memory: In this model, concurrency modules interact through shared memory space.
- Message passing: Here, communication channels exist between the modules. The incoming messages are handled with the help of a queue.

Kinds of concurrent modules

- Process: An isolated instance of a running program, which also holds a private portion in the memory space. Processes normally do not have a shared memory. Each process runs independent of the others.
- Thread: A thread is a portion of execution. Each operation can be executed on separate threads which may or may not have a shared memory space. It takes effort to preserve a private memory for a thread. If message transfer is occurring between threads, a special queue needs to be set up.

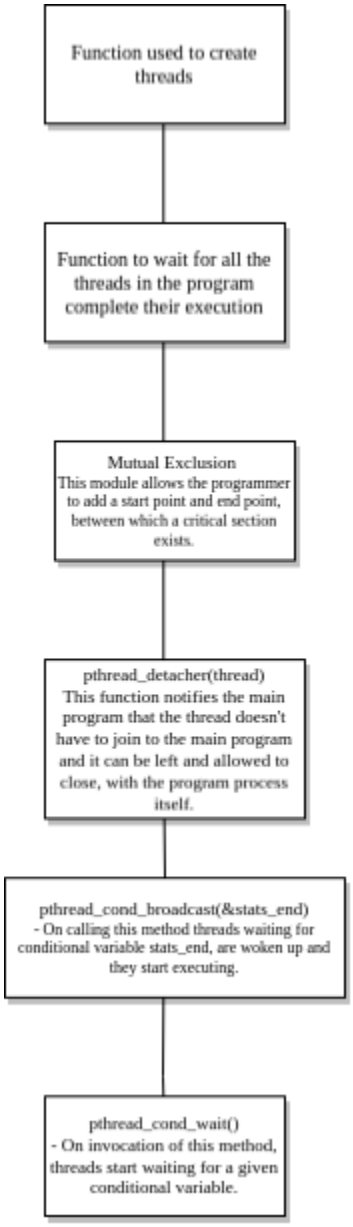
Some Important Concepts

- Time slicing: A processor switches between threads after some intervals of time to optimally utilize all available processors.
- Interleaving: This is a specific ordering of the process and read-write operations to prevent any clashes or errors during said operations.
- In message passing, interleaving occurs not between the operations themselves, but between the instructions and messages.

Solution Outline

Concurrency Extension Contents

The following diagram shows the modules we plan to include in the extension:



Choosing an Approach

After reading up on Concurrency and the different components involved, we tried searching for more resources to develop an idea or a starting point for our project. We finally landed on the official OCaml source code and searched through the library code to find the part where threads were implemented. We found it here.

Upon reading through and understanding the code, we observed that to create threads, the developers had wrapped OCaml and C together using the 'external' keyword. We decided to take the same approach, but since using what they had used was too complex for our understanding, we used the POSIX API. We also

dabbled with the idea of implementing kernel-level threads, but we decided to stick to user-level threads so that the functions used could be user defined and we already knew how they worked.

The 'external' Keyword

User-defined primitives, written in C, can be linked with OCaml code and called from OCaml functions using the 'external' keyword. It is used as:

external name : type = C-function-name

The Code

In our code, we used the external keyword and defined the following functions:

- create_thread: string → int64
- join_thread: int64 → unit
- self_thread: unit → int64
- sleep : int → unit
- sem_create: unit → int
- sem_wait: int → int
- sem_post: int → int

Each of these functions do as the name suggests and are linked to C functions which call the respective POSIX library functions. A locking mechanism was also implemented using semaphores (the functions sem_create, sem_wait and sem_post).

One obstacle we faced was how to include the linker '-pthread' while compiling using ocamlopt. We figured out that we had to add '-cclib' before the linker at the end of the compile statement.

We also planned to create a test case using a small multi-threaded application, but due to time constraints and projects in courses, we could not implement it. Our test file hence contains three threads. After the first one, the system sleeps for a specified amount of time after which the second thread executes. This is our final implementation.

The thread function

As of now, our thread function can be defined only in the C program but we have tried to implement it in OCaml using the 'callback' functions. The problem which arised however was that we could not figure out how to pass a function between OCaml and C; even passing simple variables and parameters required a lot of type conversions because the parameters passed to the C functions could only be of a specific type - 'value'. Also, defining the thread function in OCaml meant that we had to do continuous transitions between OCaml and C and that process was too cumbersome.

References

1. https://web.mit.edu/6.005/www/fa14/classes/17-concurrency/
   The above link elaborates over the concepts of Concurrency and it's types. The link helped us understand the basic concepts of concurrency and thread operations.

2. https://begriffs.com/posts/2020-03-23-concurrent-programming.html
   The above link explains concurrent programming by exploring the POSIX threads (pthread) API in the C Language. This helped us find the modules which we could implement for our project.

3. https://ocaml.org/manual/libthreads.html
   The above link would help us see how OCaml's threading library can be used. This would help us design a similar interface for our extension.

4. https://github.com/ocaml/ocaml/
   This is the link to the actual OCaml source code.

5. https://v2.ocaml.org/manual/intfc.html
   This is the OCaml Manual on their website.