継続を中心とした言語 Gears OS のデモンストレーション

宮城 光希[1,a)] 河野 真治[1,b)]

概要：現代の OS では拡張性と信頼性を両立させることが要求されている。信頼性をノーマルレベルの計算に対して保証し、拡張性をメタレベルの計算で実現することを目標に Gears OS を設計中である。Gears OS は Continuation based C(CbC) によってアプリケーションと OS そのものを記述する。CbC はこの Code Gear と Data Gear の単位でプログラムを記述する。システムやアプリケーションを記述するために Code Gear と Data Gear を柔軟に再利用する必要がある。このときに機能を接続する API と実装の分離が可能であることが望ましい。Gears OS の信頼性を保証するために、形式化されたモジュールシステムを提供する必要がある。本論文では、Interface を用いたモジュールシステムの説明とその応用としての並列 API について考察する。並列 API は継続を基本とした関数型プログラミングと両立する必要がある。ここでは、CbC の goto 文を拡張した par goto 文を導入する。par goto 文を用いることによって Gears OS は並列処理を行う。また、本研究ではハードウェア上でメタレベルの処理、および並列実行を可能とするために、raspberry pi 上での Gears OS の実装についての考察を行う。

キーワード：OS, プログラミング言語, CbC, Gears OS

1. メタ計算を使った並列処理

プログラムの処理において、プログラミング言語で記述される部分をノーマルレベルの計算と呼ぶ。コードが実行される時には、処理系の詳細や使用する資源、あるいは、コードの仕様や型などの言語以外の部分が服属する。これをメタレベルの計算という。

従来の OS では、メタ計算はシステムコールやライブラリーコールの単位で行われる。しかし、メタ計算は性能測定あるいはプログラム検証、さらに並列分散計算のチューニングなど細かい処理が必要で実際のシステムコール単位では不十分である。

Gears OS はノーマルレベルの計算とメタレベルの計算を階層化することで、ノーマルレベルの計算の信頼性をメタレベルから保証できることを目指して開発している。

メタ計算を通常の計算から切り離して記述するためには処理を細かく分割する必要がある。しかし、関数やクラスなどの単位は容易に分割できない。そこで当研究室ではメタ計算を柔軟に記述するためのプログラミング言語の単位として Code Gear、Data Gear という単位を提案している。これによりシステムコードよりも細かくバイトコー

[1] 琉球大学工学部情報工学科
[a)] mir3636@cr.ie.u-ryukyu.ac.jp
[b)] kono@ie.u-ryukyu.ac.jp

ドよりも大きなメタ計算の単位を提供できる

ノーマルレベルとメタレベルを共通して表現できる言語として Continuation based C(以下 CbC)[6] を用いる。CbC は関数呼出時の暗黙の環境 (Environment) を使わずに、コードの単位を行き来できる引数付き goto 文 (parametarized goto) を持つ C と互換性のある言語である。この CbC を用いて、Code Gear と Data Gear、さらにそのメタ構造を導入する。これらを用いることで、検証された Gears OS を構築したい。

2. Gears OS

Gears OS は処理の単位である Code Gear とデータの単位である Data Gear を用いて開発されている。

Code Gear は CbC における最も基本的な処理単位である。関数に比べて細かく分割されているのでメタ計算をより柔軟に記述できる。Data Gear はデータの単位であり、int や文字列などの Primitive Type を持っている。Code Gear、Data Gear にはそれぞれメタレベルの単位である Meta Code Gear、Meta Data Gear が存在し、これらを用いてメタ計算を実現する。

Gears OS では、並列実行するための Task を、実行する Code Gear と、実行に必要な Input Data Gear 、Output Data Gear の組で表現する。この Input Data Gear、Output Data Gear の依存関係が解決された Code Gear を並列実行する。

Code Gear は任意の数の Input Data Gear を参照して処理を行い、Output Data Gear を出力し処理を終える。また、接続された Data Gear 以外には参照を行わない。処理やデータの構造が Code Gear、Data Gear に閉じているため、これにより実行時間、メモリ使用量などを予測可能なものにすることができる。

Gears OS では メタレベルの計算 を Meta Code Gear、Meta Data Gear で表現する。(図 1)

Meta Code Gear は通常の Code Gear の直後に遷移され、メタレベルの計算を実行する。Meta Code Gear で OS の機能であるメモリ管理やスレッド管理を行う。

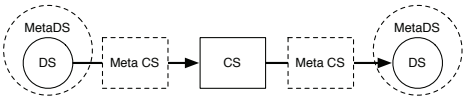

図 1: Gears でのメタ計算

CbC は軽量継続 (goto 文) による遷移を行うので、継続前の Code Gear に戻ることはなく、状態遷移ベースのプログラミングに適している。CbC は LLVM[4]、gcc[2] 上で実装されており、メタレベルでの記述は、Perl による変換スクリプトにより生成される stub と goto meta によって Code Gear で記述される。

3. interface

Code Gear と Data Gear は Interface と呼ばれるまとまりとして記述される。Interface は使用される Data Gear の定義と、それに対する操作を行う Code Gear の集合である。Interface は複数の実装を持つことができ、Meta Data Gear によって定義される。Interface の操作に対応する Code Gear の引数は Interface に定義されている Data Gear を通して行われる。一つの実行スレッド内で使われる Interface の Code Gear と Data Gear は Meta Data Gear に格納される。この Meta Data Gear を Context という。ノーマルレベルでは Context を直接見ることはできない。

メタ計算側ではこれらの Context を常に持ち歩いているので goto 文で引数を用いることはなく、行き先は Code Gear の番号のみで指定される。

これにより Interface 間の呼び出しを C++ のメソッド呼び出しのように記述することができる。Interface の実装は、Context 内に格納されているので、オブジェクトごとに実装を変える多様性を実現できている。

呼び出された Code Gear は必要な引数を Context から取り出す必要がある。これは スクリプトで生成された stub Meta Code Gear で行われる。Gears OS でのメタ計算は stub と goto のメタ計

算の 2 箇所で実現される。

4. Gears OS の構成

Gears OS は以下の要素で構成される。

- Context
- TaskQueue
- TaskManager
- Worker

図 2 に Gears OS の構成図を示す。

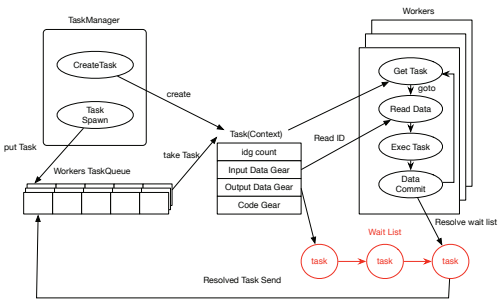

図 2: Gears OS の構成図

Context には Data Gear の Data Type の情報が格納されている。この情報から確保する Data Gear のサイズなどを決定する。Context は Task でもあり、Task は通常の OS のスレッドに対応する。Task は実行する Code Gear と Data Gear をすべて持っている。TaskManager は Task を実行する Worker の生成、管理、Task の送信を行う。Gears OS における Task Queue は Synchronized Queue で実現される。Worker は TaskQueue から Task である Context を取得し、Task の Code Gear を実行し、Output Data Gear の書き出しを行っている。Input/Output Data Gear の依存関係が解決されたものから並列実行される。

5. Gears OS の並列処理

Gears OS では実行の Task を Code Gear と Input/Output Data Gear の組で表現する。Input/Output Data Gear によって依存関係が決定し、それにそって並列実行を行う。Gears OS では 並列実行する Task を Context で表現する。Context には Task 用の情報として、実行される Code Gear、Input/Output Data Gear の格納場所、待っている Input Data Gear のカウンタ等を持っている Task の Input Data Gear が揃っているかを TaskManager で判断し、 実行可能な Task を Worker に送信する。Worker は送信された Task が指定した Code Gear を実行し、Output Data Gear を書き出す。Context を複製して複数の CPU に割り当てることにより並列実行を可能になる。

6. 並列構文

Gears OS の並列構文は par goto 文で用意されている (Code1)。

```
__code code1(struct Integer* integer1, struct
      Integer* integer2, struct Integer*
     output) {
    par goto add(integer1, integer2, output,
         __exit);
    goto code2();
}
```

Code 1: par goto による並列実行

par goto の引数には Input/Output Data Gear と実行後に継続する __exit を渡す。par goto で生成された Task は __exit に継続することで終了する。

この par goto 文は通常のプログラミングの関数呼び出しのように扱うことができる。

par goto 文でも 通常の goto 文と同様にメタへの goto 文へ置き換えられるが、par goto 文では通常の goto 文とは異なる meta code gear へと継続する。Gears OS の Task は Output Data Gear を生成した時点で終了するので、par goto では直接 __exit に継続するのではなく、Output Data Gear への書き出し処理 (Commit) に継続される。

Code Gear は Input に指定した Data Gear が全て書き込まれると実行され、実行した結果を Output に指定した Data Gear に書き出しを行う。Code Gear の引数である __next が持つ引数の Data Gear が Output Data Gear となる。

書き出し処理は Data Gear の Queue から、依存関係にある Task を参照する。参照した Task が持つ実行に必要な Input Data Gear カウンタのデ

クリメントを行う。カウンタが 0 になると Task が待っている Input Data Gear が揃ったことになり、その Task を TaskManager 経由で 実行される Worker に送信する。

7. raspberry pi 上での Gears OS の実装

Gears OS をハードウェア上で実装するために、ARM[1] プロセッサを搭載したシングルボードコンピュータである Raspberry Pi[5] 上での Gears OS 実装する。ハードウェア上でのメタレベルの計算や並列実行を行うために、Linax 等に比べてシンプルである xv6 の機能の一部を Gears OS に置き換えることで実現させる。xv6 は UNIX V6 を x86 へ再実装したものであるが、ここでは Raspberry pi 用に移植した xv6_rpi_port[3] を用いて実装した。

xv6_rpi_port を Gears OS で置き換えるためには xv6_rpi_port を CbC コンパイラを用いてコンパイルした kernel が Raspberry Pi 上で動作する必要がある。そこで GCC 上で実装した CbC コンパイラを ARM 向けに build し xv6_rpi_port のクロスコンパイルを行い動作させることができた。

8. 今後の課題

本研究では Gears OS のプロトタイプの設計と実装、並列実行機構の実装を行った。Code Gear 、Data Gear を処理とデータの単位として用いて Gears OS を設計した。Gears OS は Code Gear と Input/Output Data Gear の組を Task とし、並列実行を行う。

Code Gear と Data Gear は Interface と呼ばれるまとまりとして記述される。Interface は使用される Data Gear の定義と、それに対する操作を行う Code Gear の集合である。Interface は複数の実装をもち、Meta Data Gear として定義される。従来の関数呼び出しでは引数をスタック上に構成し、関数の実装アドレスを Call するが、Gears OS では引数は Context 上に用意された Interface の Data Gear に格納され、操作に対応する Code Gear に goto する。

並列処理を行う際は Context を生成し、 Code Gear と Input/Output Data Gear を Context に設定して TaskManager 経由で各 Worker の SynchronizedQueue に送信される。Context の設定はメタレベルの記述になるため、ノーマルレベルでは par goto 文という CbC の goto 文に近い記述で並列処理を行える。この par goto は通常のプログラミングの関数呼び出しのように扱える。

また、ハードウェア上での実装のため gccbc での xv6_rpi_port のクロスコンパイルを行なった。

今後の課題は、gccbc でクロスコンパイルした xv6_rpi_port が実際のハードウェア上で動作することが可能となったため、CbC で記述された Gears OS も ARM 向けにコンパイルが可能となった。xv6_rpi_port の機能を Gears OS の機能と置き換えていくことで、ハードウェア上でのメタ計算、並列処理を可能とすることが課題である。

参考文献

[1] ARM Architecture Reference Manual: *http://infocenter.arm.com/help/topic/com.arm.doc.subset.architecture.reference/index.html.*

[2] GNU Compiler Collection (GCC) Internals: *http://gcc.gnu.org/onlinedocs/gccint/.*

[3] Huang, Z.: xv6_rpi_port, https://github.com/zhiyihuang/xv6_rpi_port (2013).

[4] Lattner, C. and Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California (2004).

[5] Raspberry Pi — Teach, Learn, and Make with Raspberry Pi : https://www.raspberrypi.org.

[6] TOKKMORI, K. and KONO, S.: Implementing Continuation based language in LLVM and Clang, *LOLA 2015* (2015).

[7] 河野真治，伊波立樹，東恩納琢偉：Code Gear、Data Gear に基づく OS のプロトタイプ，情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS) (2016).