計算機システムⅡ

2023 年度

九州工業大学 情報工学部

目次

前書き

本講義の目的

我々が生活する実社会では、スーパーコンピュータ、汎用大型機、パーソナルコンピュータ、タブレット端末、スマートフォン、自動車や家電製品等の組み込み機器等、様々な規模・性能を持った計算機が存在し、その上では、様々な目的を持った各種アプリケーションプログラムが実行されている。しかし、これら多くのシステムの基本的構成や、動作原理には大きな相違はない。

我々、情報関連の技術者を目指して学ぶ者は、立場の相違はあっても、計算機システムがどのような構成になっており、どのように動作するかを理解する必要がある。そのため、本講義では、「計算機システムI」の講義を前提として、情報技術者の立場から、プログラマがプログラミング言語で書いたアプリケーションプログラムが、最終的にどのような形でCPU 内で実行されるかを理解し、それがオペレーティングシステムの助けを借りて、どのように様々な計算資源を活用するかを理解することを目的とする。

また、本講義では、これらを広く概観するため、詳細や理論的側面に踏み込むことはせず、代表的な実例を挙げることにより、計算機システムに関する包括的な概念を得ることを目的とする。

第１講　計算機システムの概観

本講では、「計算機システムⅠ」の講義の簡単な復習も含めて、計算機システムの概観を以下の観点から理解することを目的とする。これにより、本講以降の講義内容の概要を把握する。

・ハードウェアとしての計算機システムの概観
・プログラマの書いたソースプログラムを、CPU で実行可能な機械語のプログラムまで変換し、実行するまでの過程
・CPU の基本的な構造と、機械語プログラムの実行過程
・アプリケーションプログラムの実行の際に、CPU やメインメモリ、入出力装置等の計算資源を利用するに当ってのオペレーティングシステム（OS）の役割

1.1 計算機ハードウェアの基本構成

計算機の概念的な構成を図 1.1 に示す。

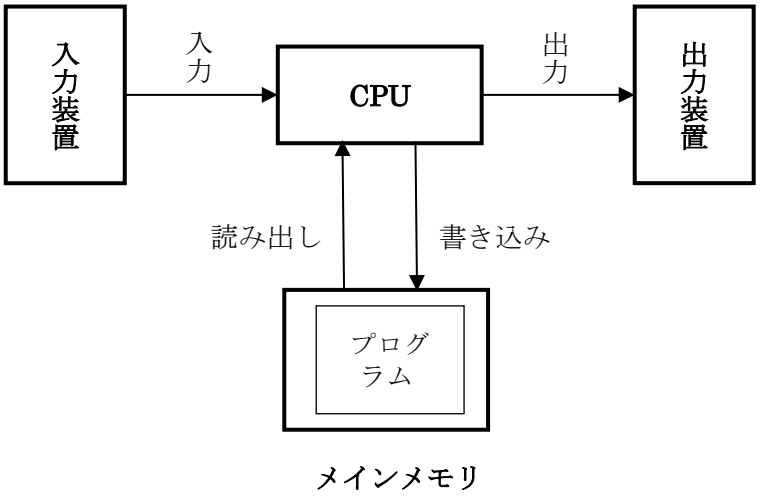


図 1.1 計算機の概念的構成

・CPU (Central Processing Unit)
　メインメモリに格納されているプログラムの命令を、一つずつ取り出し、解釈、実行する。その際に必要なデータもメインメモリとの間でやり取りする。
・メインメモリ（main memory)
　メインメモリは、（最後に格納された）命令やデータを保持（記憶）する。
　メインメモリには一連の（メモリ）アドレス（番地）が振られており、CPU はメモリアドレスを指定して、命令の読出し、データの読み書きを行う。

・入出力装置　(input/output device)

人間や機械等、計算機外部とのインタフェースを司る。

例として、キーボード、マウス、ディスプレイ、プリンタ、カメラ、オーディオ機器、ネットワークインタフェース、各種センサ等が挙げられる。

ハードディスク（ディスク装置)、CD/DVD 等の補助記憶装置も入出力装置として扱われる。

1.2 コンパイルから実行まで

1.2.1 一連の流れ

プログラムのコンパイルから実行までの一連の流れを図 1.2 に示す。

プログラマが書いたテキストのプログラムを、ソースプログラムという。

これらは、一般に C 言語などのプログラミング言語で書かれる。

プログラミング言語で書かれたプログラムは、コンパイラによりコンパイルされ、機械語のプログラムに変換される。

機械語のプログラムは、最終的にはそのまま実行可能なロードモジュールとして、補助記憶装置上のファイルに格納（メインメモリと異なり電源オフでも保持）される。

プログラムの実行は、オペレーティングシステム（OS）の一部であるローダがロードモジュールをメインメモリにロードし、CPU が機械語命令を 1 つずつ読み込み（命令フェッチ)、解釈、実行することにより行われる。

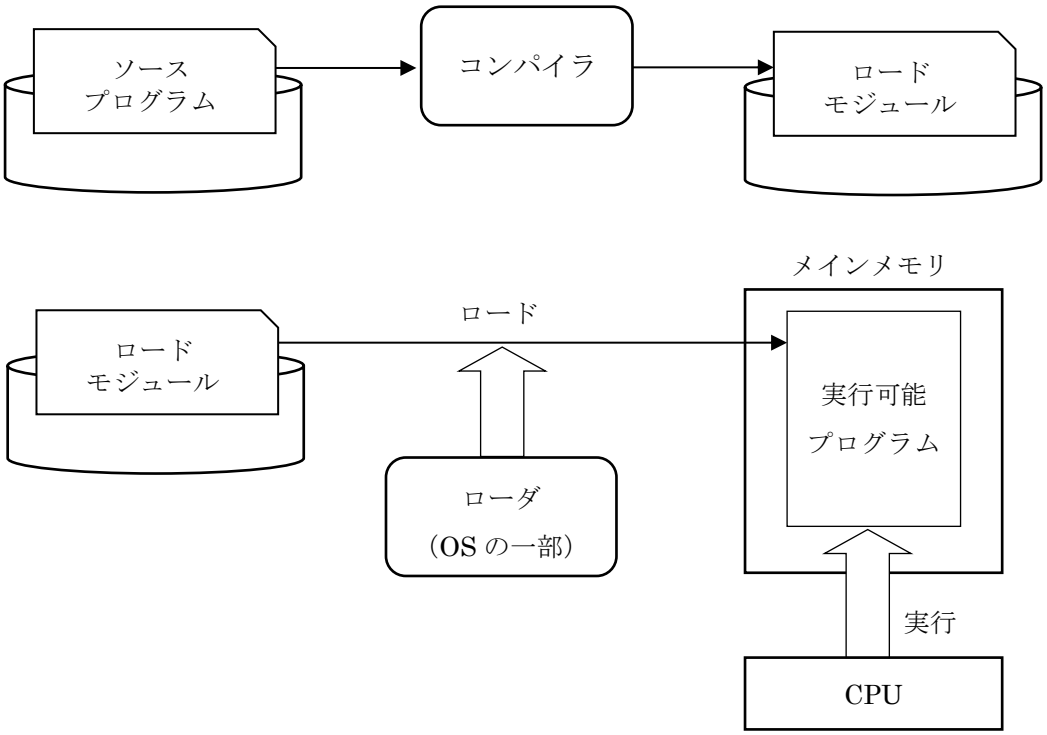


図 1.2 コンパイルから実行までの流れ

ソースプログラム（source program）

プログラマが書くテキストで書かれたプログラム。一般的には、プログラミング言語で書かれる。

コンパイラ（compiler）

プログラミング言語で書かれたプログラムを機械語のプログラムに変換する。

ロードモジュール（load module）

実行可能な機械語のプログラム

ローダ（loader）

ロードモジュールをメインメモリにロードする。OS の一部

1.2.2 プログラミング言語、アセンブリ言語、機械語の関係

これらの関係を図 1.3 に例示する。

プログラミング言語

C 言語などのプログラマ向けの言語

CPU に依存しない。

コンパイラを用いて機械語に変換される。

アセンブリ言語

機械語を人間に理解しやすく記号化した言語

CPU の命令セットに固有（図 1.3 の例は、MIPS のアセンブリ言語）

アセンブラを用いて機械語に変換

機械語

メモリにロードして、CPU が直接実行できるバイナリ形式

CPU の命令セットに固有（図 1.3 の例は、MIPS の機械語）

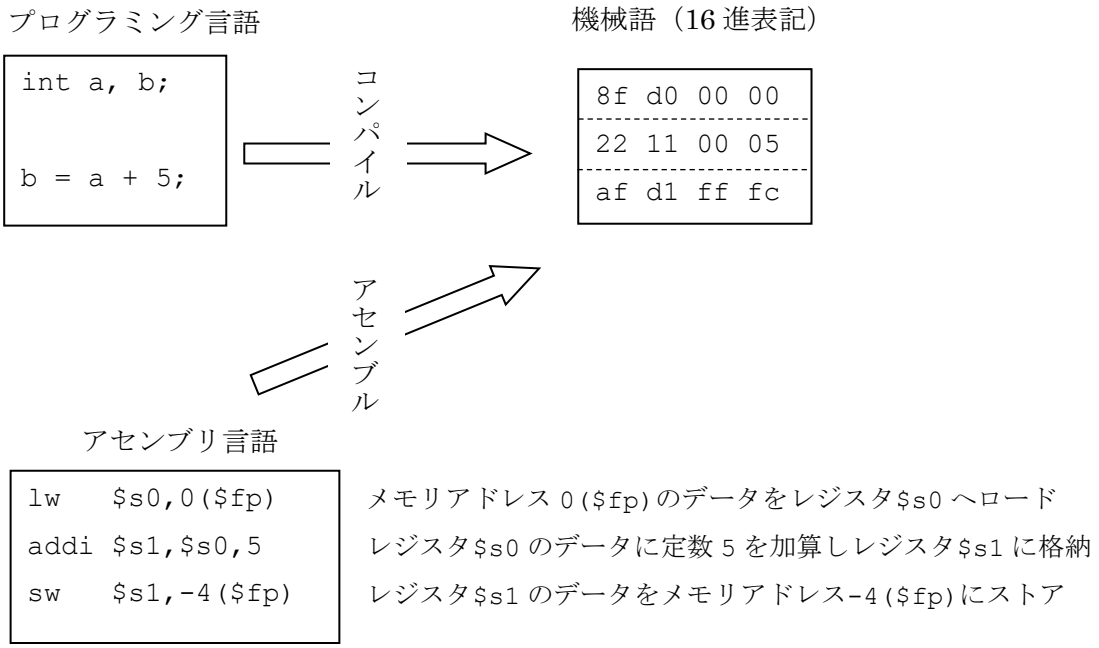


図 1.3 プログラミング言語、アセンブリ言語、機械語の関係

1.3 CPU の構成と動作

1.3.1 CPU の概念的な構成

CPU の概念的な構成を図 1.4 に示す。

・命令デコーダ

命令コードを解読する。

（例：加算命令を解読し、演算回路に対して加算の実行を指示する等）

・レジスタ

メモリから読み出したデータや演算結果などを一時的に保持する。

・演算回路

算術演算や論理演算を行う回路

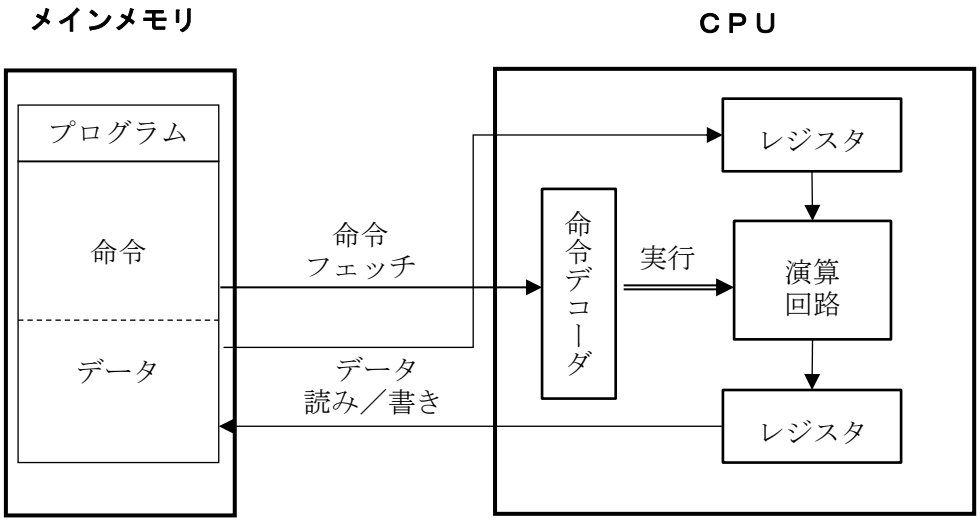


図 1.4 CPU の概念的構成

1.3.2 機械語プログラムの実行

CPU は図 1.5 に示す命令実行サイクルにより、命令を１つずつ、順番に実行する（逐次実行)。フェッチする（そしてデコードし、実行する）命令のメモリアドレスは、一般にプログラムカウンタ（詳細は第３部、但し、第２部の割り込みやプロセスにも関係する）と呼ばれる CPU 内の特殊なレジスタが保持する。

機械語プログラムの命令を単位とした実行は、命令実行前の計算機の状態から命令実行後の計算機の状態へと 1 命令実行するたびに状態遷移していくものと考えられる。計算機の状態は、入出力装置を除くと、メインメモリ内に保持される命令やデータ、CPU 内のレジスタに保持される命令やデータにより定まる。

メインメモリにとってはデータも命令もビット列であり 16 進表記が便利である。これを CPU が、数値やメモリアドレス、文字コード、命令コードなどとして利用する。

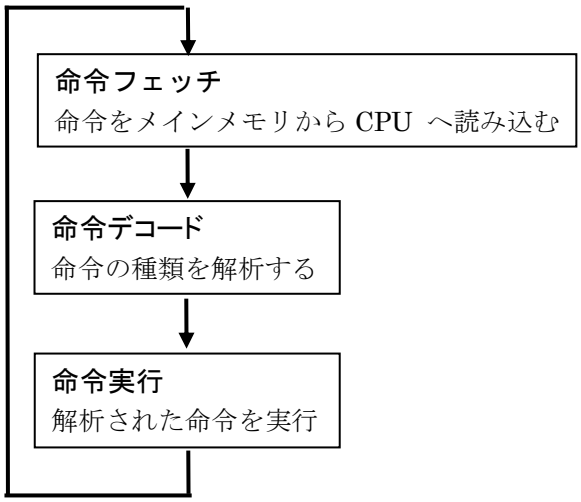


図 1.5 命令実行サイクル

1.3.3 命令の種類（CPU には何ができるか）

・転送命令

レジスタ間、メモリ・レジスタ間のデータのコピー

アクセス（読み書き）対象をレジスタ名やメモリアドレスで指定

・算術演算命令

加減乗除の演算

・論理演算命令

論理和、論理積、排他的論理和等

・プログラムの実行を制御する命令

ジャンプ（分岐）命令、サブルーチンコール命令など

1.4 オペレーティングシステム（OS: operating system）

1.4.1 身近なオペレーティングシステムの例

パーソナルコンピュータ用

Windows、macOS、Linux 等

スマートフォン用

Android、iOS 等

本学の学生証、交通系 IC カード等

FeliCa OS

1.4.2 オペレーティングシステムの役割

・CPU、メインメモリ、入出力装置など計算資源の詳細を、プログラマを含む利用者から

隠す。

利用者は、複数のプログラムをどのように実行させるかを知る必要はない。

（複数のウィンドウをオープンしている場合、複数のプログラムが並行して実行されている。）

利用者は、複数のプログラムをメインメモリのどこにロードするかを知る必要はない。

利用者は、ハードディスクの構造や操作を知ることなく、またハードディスクか USB メモリかなど装置の相違を意識せずに、ファイルにアクセスできる。

・計算資源を効率的に、正しく管理する。

プログラムの実行効率を高くする。

メインメモリ上に無駄なくプログラムを配置し、また他のプログラムの領域にアクセスできないようにする。

ハードディスクに無駄なくファイルを配置し、また他人のファイルに勝手にアクセスできないようする。

1.4.3 オペレーティングシステムの仕事の例

・複数のプログラムのどれを実行するかを決め、CPU を割り当てる。

・複数のプログラムを並行して実行するため、同時にメインメモリ上に複数のプログラムをロードし、他のプログラムの領域と明確に区別する。

・メインメモリの容量を超える大きなプログラム（データも含む）も実行できるようにする。

・ファイルの生成、削除、検索、読み書き等の操作を行う。

・どのファイルが、ハードディスクのどこに格納されているかの管理や、アクセス制御を行う。

・入出力装置の制御を行い、実際の入出力を行う。

第１部

言語処理系

第２講　プログラムと言語処理系

本講では、プログラマにより書かれたプログラムが、最終的に CPU により実行可能な形になるまでの過程を、以下の観点から理解することを目的とする。
・プログラムを書くための言語とそれを処理する様々なレベルの言語処理系の関係
・コンパイラとインタプリタの相違

2.1 プログラムを書くための言語と言語処理系

・機械語（machine language）
メモリにロードし、CPU が直接実行できる形式の言語
バイナリ形式（0 と 1 の並び）
CPU の命令セットにより異なる（機械依存）。

・アセンブリ言語（assembly language）
機械語を人間にとってわかり易いように、記号化した言語
テキスト（文字）として表現される。
機械語と１対１に対応する。
アセンブラ（assembler）により機械語に変換される。
命令セットにより異なる（機械依存）。また同一命令セットでも複数の形式が存在する場合がある。

・プログラミング言語（programming language）
人間にとって読み書きのし易い計算機言語（C 言語など）
テキスト（文字）として表現される。
アセンブリ言語等に対して、高水準言語（high-level language）とも呼ばれる。
コンパイラ（compiler）により、機械語、あるいはアセンブリ言語に変換される。
または、インタプリタ（interpreter）により、そのプログラムが実行される。
CPU とは独立である。即ち、CPU に関係のない共通の言語である。

2.2 トランスレータ（translator）

コンパイラやアセンブラなど，ある言語で書かれたプログラムを、それと同等の（同じ動作をする）別の言語で書かれたプログラムに変換するソフトウェア
トランスレータへの入力プログラム（変換前のプログラム）を、ソースプログラム（source program）、あるいは、一般的にテキストで書かれているので、ソーステキスト（source

text) と呼ぶ。また、これを格納したファイルを、ソースファイル (source file) と呼ぶ。

トランスレータの出力プログラム（変換後のプログラム）を、オブジェクトプログラム (object program)、特にこれが機械語の場合、オブジェクトコード (object code) と呼ぶ。また、これを格納したファイルを、オブジェクトファイル (object file) と呼ぶ。

例）トランスレータの一種である C コンパイラは、C 言語で書かれたソースプログラムを、一般的には、その計算機の CPU が実行できる機械語のプログラム（オブジェクトコード）に変換（翻訳）する。

2.3 アセンブラ

アセンブリ言語で書かれたプログラムを機械語のプログラムに変換するソフトウェア

機械語の命令コードを人間にわかりやすく記号化した命令の表記をニーモニック (mnemonic) という。

アセンブリ言語のプログラムは、ニーモニックを用いてコーディングする。

アセンブラは、個々のニーモニックを対応する機械語の命令等に 1 対 1 で変換する。

ニーモニックは、オペコード部とオペランド部からなる。

オペコードは、命令の種類を表す（データ転送命令、加算命令等)。

オペランドは、データ転送命令や加算命令の場合、命令の対象となるデータまたは格納先を表す（即値なら値そのもの（定数）を指定、レジスタなら名称で指定、メモリならメモリアドレス（の求め方）を指定)。

MIPS の例)

`a = a + 5` に相当するアセンブリ言語と機械語

アセンブリ言語のニーモニック　`addi $s0, $s0, 5`

```
          ↑      ↑
          |      オペランド部
       オペコード部
```

（レジスタ`$s0` の値と定数 5 を加算し、結果を`$s0` に格納する。）

機械語のコード（２進表記）`001000  10000  10000  0000000000000101`

```
addi    $s0    $s0       即値 5
```

2.4 コンパイラ

プログラミング言語で書かれたプログラムを、機械語、あるいはアセンブリ言語のプログラムに変換するソフトウェア

第３講、第４講で詳細に講義する。

分割コンパイル (separate compilation)

メインルーチン (main routine)、サブルーチン (subroutine)、あるいは手続き

(procedure)、関数 (function) などのプログラム単位（あるいはこれらを格納したファイル）ごとにコンパイルを行うこと

分割コンパイルにより生成された個々のオブジェクトモジュール (object module) は、リンカ (linker) により結合されてロードモジュール (load module) が生成される。

ロードモジュールは、オペレーティングシステム（OS）内のローダによりメモリにロードされ、CPU により実行される。

クロスコンパイラ (cross compiler)

ある CPU を搭載した計算機で、別の CPU の機械語のコードを生成するコンパイラ。

通常、ある計算機上のコンパイラは、その計算機でそのまま実行するオブジェクトコードを生成する。

しかし、例えば、組み込みソフトウェアの開発において、開発用計算機上のクロスコンパイラは、対象となる組み込み用 CPU のオブジェクトコードを生成する。

2.5 リンカ (linker)

分割コンパイルにより生成された複数のオブジェクトモジュールやライブラリを結合して、１つのロードモジュールを生成するソフトウェア

分割コンパイルにより生成された個々のオブジェクトモジュールには、大域変数やサブルーチンなどの外部参照が解決されていない。

リンカにより、大域変数やサブルーチンのアドレスが決定される。

2.6 インタプリタ

ソースプログラム（ソーステキスト、あるいは中間言語のプログラム）を１ステップずつ解釈して実行していくソフトウェア

コンパイラ（トランスレータ）がプログラムを一括して**翻訳**するのに対して、インタプリタは逐次的にプログラムを解釈していく**通訳**の役割を果たす。

インタプリタが主に用いられる言語の例

Prolog, Lisp や、Python, Ruby, Perl などのスクリプト言語

2.7 言語処理系 (language processor / language system)

人間が書いたプログラム、即ち、人間が理解できる言語で書かれたプログラムを計算機上で実行させるための処理を行うソフトウェア

上述のように、言語処理系は、トランスレータ（アセンブラやコンパイラ)、インタプリタ、及びリンカを含む。

・Java 言語の場合

Java 言語で書かれたソースプログラムは、Java コンパイラによりコンパイルされ、機械語における命令列に相当する Java bytecode と呼ばれる中間コードに変換される。

この中間コードは、Java virtual machine (Java VM) により解釈・実行される。

即ち、Java VM は、Java bytecode のインタプリタである。

練習問題 1

以下の用語を簡単に説明しなさい。

1. 機械語、アセンブリ言語、プログラミング言語
2. コンパイラとアセンブラ
3. ソースプログラムとオブジェクトプログラム
4. トランスレータとインタプリタ
5. リンカ

第３講　コンパイラの概観と字句解析の概要

本講及び次講では、プログラミング言語で書かれたプログラムを機械語のプログラムに変換するコンパイラについて理解することを目的とする。特に本講では、コンパイラの概観と、その最初の段階である字句解析について、以下の観点から理解する。

・コンパイラの全体構成とコンパイルの過程についての概観
・プログラムを構成する単語に当たる字句について、その構成と表現法
・プログラムから字句を切り出す字句解析器の構成

3.1 コンパイラの概観

コンパイルの過程は、大きく分けて次の４つの処理からなる。

1）字句解析（lexical analysis）

字句解析では、ソースプログラムのテキストを字句（lexical element: 字句要素とも）に分割し、内部表現に変換する。

字句とは、自然言語の単語に当たる。

コンパイラ内部で字句解析を行う部分を、字句解析器（lexical analyzer）、あるいはスキャナ（scanner）と呼ぶ。

2）構文解析（syntax analysis, parsing）

構文解析では、字句の列をプログラミング言語の文法に合致しているかのチェックを行いながら、文法構造を解析し、構文木（syntax tree）などの中間的な内部表現を生成する。合致しない場合は構文エラーとする。

コンパイラ内部で構文解析を行う部分を、構文解析器（syntax analyzer）、あるいはパーサ（parser）と呼ぶ。

3）意味解析（semantic analysis）

型の検査などを行う。

例）言語によっては文字と文字の積は無意味であり、型エラーとする。

```
char a, b, c;
a = b * c;
```

構文解析では型エラーを検知できない（型の整合性は文法で表現できないので）。

言語によっては、型の自動変換を行う。

例）

```
float a, b;
int c;
a = b * c;
```

の場合、c の値をいったん float に変換して、乗算を行う。

4) コード生成（code generation）

構文解析器が生成した構文木などの中間的な内部表現から、それぞれの CPU の機械語コードを生成する。

字句解析から意味解析までは、それぞれのプログラミング言語独自のものであり、そのプログラムが実行される CPU には依存しないが、コード生成は CPU に依存する。

コード生成に当たっては、コードの最適化が行われる場合もある。

コードの最適化（code optimization）とは、生成するコードを、実行効率を上げる、コードを小さくする、実行時に必要な記憶領域を小さくするなどの観点からより良いコードに変換することである。

3.2 字句と正規表現

3.2.1 字句

C 言語においては、字句は以下のように分類される。

・識別子（identifier）

変数名や関数名などに用いられる。

先頭が英字で、残りは英字または数字である。この場合の英字には、下線（ _ ）も含まれる。

・キーワード（keyword）

予め決められた用途に用いる語

例）int, while, struct など

・定数リテラル（constant literal）

表記自体がある型の特定の値を表しているもの

例）95　　十進の整数

　　0137　八進の整数

　　0x5f　十六進の整数

　　314.1592, 3.141592e2　浮動小数点定数

　　'C'　　文字定数

・文字列リテラル（string literal）

二重引用符 " " で囲まれた文字の列

例）"string literal"

・演算子（operator）

例）+, *, =, <=, &&, ! など

・区切り記号

例）;, {, } など。カンマ（ , ）については演算子として使われることもある。

3.2.2 字句の定義

字句は正規表現で定義することができる。

例えば、C 言語における識別子は、先頭が英字で残りが英数字であるので、次のように定義できる。

例）識別子の正規表現

α を A|B|C| … |Y|Z|a|b|c| … |y|z|_ の略記（英字を表す）、δ を 0|1|2|3|4|5|6|7|8|9 の略記（数字を表す）とすると、

識別子の正規表現は、

α(α | δ)*

となる。

ここで、縦棒（ | ）は「または」を、アスタリスク（ * ）は 0 回以上の繰り返しを表す。

また、

[A-Za-z_][0-9A-Za-z_]*

と表記する記法もある。

ここで、[文字の列] は、この中の文字のいずれか 1 文字。A-Z は、A から Z までの 1 文字、即ち大文字 1 文字を表す。従って、[A-Za-z_] は英字 1 文字を表す。

即ち、識別子は、英字のあとに 0 個以上の英字または数字が続く。

3.3 字句解析器と有限オートマトン

字句解析器は、文字の列であるソースプログラムを入力とし、文字を順に読み込んでいき、字句を切り出してトークン（token）と呼ばれる内部表現の列に変換する。

例）

以下の C 言語のプログラムの一部

```
int sum, i;
sum = 0;
for (i = 1; i <= 10; i++)
    sum += i;
```

は、次のように字句の切り出しが行われる。

int sum , i ; sum = 0 ; for (i = 1 ; i <= 10 ; i ++) sum += i ;

字句解析器

字句は、既に述べた通り正規表現で定義できる。

字句の種類ごとに正規表現で定義し、その正規表現で定義された文字列を受理する有限オートマンを構成できるので、字句解析器は有限オートマトンを応用して構成するこ

とができる。

例）整数を解析する有限オートマトン

整数は、十進、八進、十六進を含めて正規表現として次のように表せる。

```
[1-9][0-9]*|0([0-7]*|[Xx][0-9A-Fa-f]+)
```

ここで、“+”は、1回以上の繰り返しを表す。

これを受理する有限オートマトンを図 3.1 に示す。二重丸は受理状態を表す。

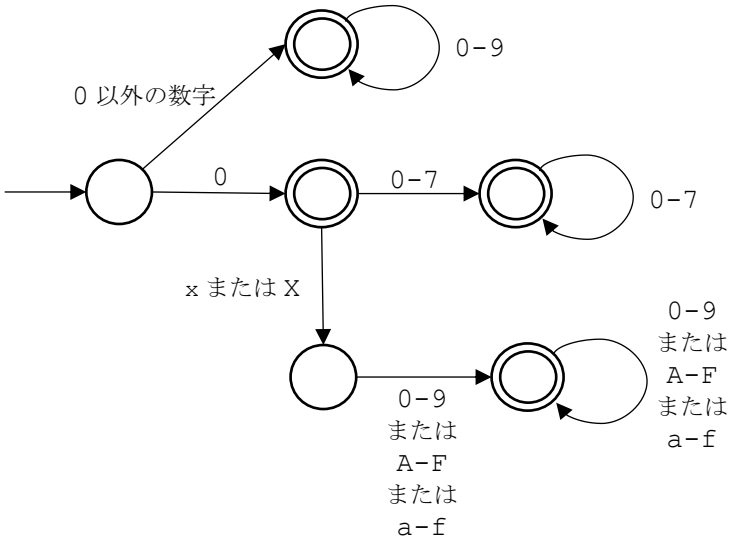


図 3.1 整数の解析を行うオートマトン

しかし、現実のプログラミング言語の字句解析では以下の点が異なる。

1) 最も長い字句を切り出す。

例）

abc という文字列は1つの識別子とみなす。

a, ab も識別子となり得るが、最長の abc を識別子とする。

+=, ++, <= などの演算子も同様である。

2) 字句の種類ごとに優先度を設ける。

例）

キーワード if は、英字の並びなので識別子にもなり得るが、キーワードに高い優先度を与えることにより、if をキーワードとすることができる。

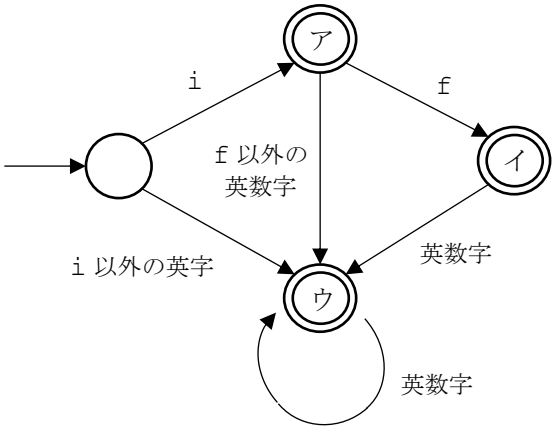
3) 言語、あるいはコンパイラによっては、識別子の長さに上限がある場合がある。

練習問題 2

1. 以下の C 言語の字句は、3.2.1 項に示すどの種類に属するか。

i)	`5`	ii)	`'5'`
iii)	`"5"`	iv)	`"x86"`
v)	`x86`	vi)	`0x86`
vii)	`++`	viii)	`for`
ix)	`"for"`	x)	`for0`

2. 以下の図に示す有限オートマトンのそれぞれの受理状態で受理される文字列は、C 言語の字句としてはどのような種類に分類されるか。



第４講　構文解析とコード生成の概要

本講では、前講に引き続き、コンパイラの残りの部分を、簡単な例を挙げ、以下の観点から理解することを目的とする。

・プログラミング言語の文法の定義と記述法
・プログラムの構文を解析する構文解析
・機械語のプログラムを生成するコード生成

4.1 プログラミング言語の構文と構文解析

プログラミング言語は、以下の３点から定義される。

例として、あるプログラミング言語における if 文を挙げる。

1) プログラムの構成要素の構造を規定する構文 (syntax)

`if ( <式> ) <文>1 else <文>2`

2) 構文的に正しいプログラムを実行するための制約 (constraint)

<式> は、真または偽を値として持たなければならない。

3) プログラムの構成要素がどのように実行されるかを定義する意味 (semantics)

まず <式> を評価し、その値が真ならば <文>1 を、偽ならば <文>2 を実行する。

構文解析の対象となるのは、構文のみであり、制約や意味は意味解析の対象である。

4.2 構文の記述

例として四則演算からなる算術式の定義の簡単な例を挙げる。

あるプログラミング言語の算術式の文法は、以下の４つの項目から定義される。

生成規則

<式> → <式> + <項> | <式> − <項> | <項>
<項> → <項> * <因子> | <項> / <因子> | <因子>
<因子> → (<式>) | *識別子* | *定数*

非終端記号

<式>, <項>, <因子>

終端記号

+, −, *, /, (,), および *識別子*, *定数*

出発記号

<式>

→ や | は、構文を定義する記号であり、メタ記号 (meta-symbol) と呼ばれる。ここ

で、縦棒（ | ）は、「または」を意味する。

4.3 構文解析

前節で定義した算術式の構文解析を例として挙げる。

ソースプログラムのテキスト

```
a + 3 * (b + c)
```

は、上に挙げた算術式の生成規則と合致する解析木を求める形で図 4.1 のように解析できる。

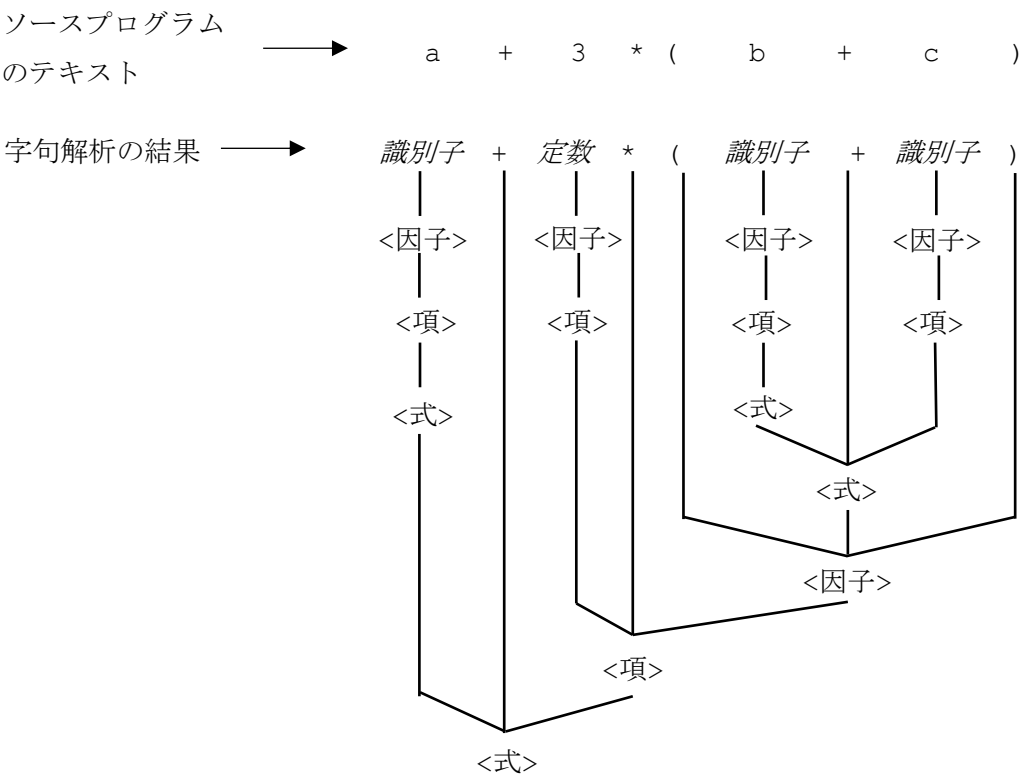


図 4.1 構文解析の例

4.3.1 演算子の優先度

多くのプログラミング言語では、乗除算が加減算より高い優先度を持つ。

例）a + b * c

は、b * c を先に計算し、その結果を a の値に加える。

これは、生成規則を以下のように定めることにより可能となる。

<式> → <式> + <項> | <項>

<項> → <項> * <因子> | <因子>

<因子> → (<式>) | *識別子*

これにより、図 4.2 のように構文解析が行われる。

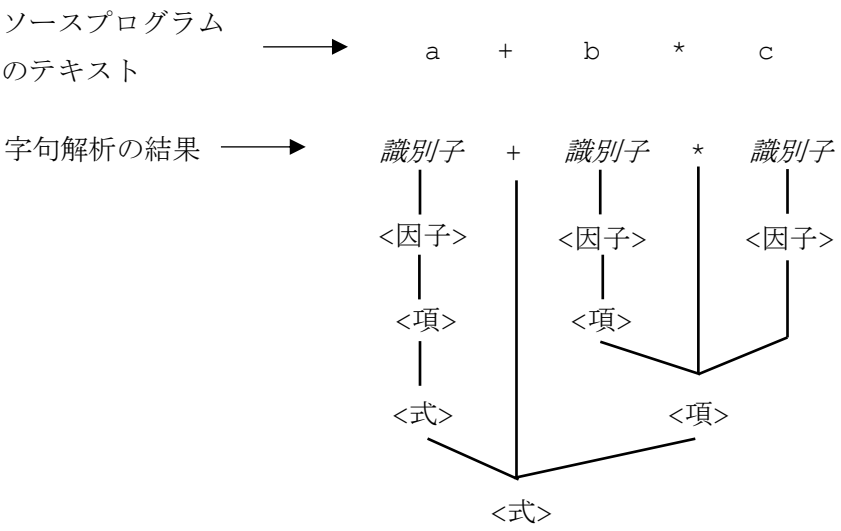


図 4.2 乗算を優先する構文解析

この解析木は、<項> である b * c を先に計算し、<式> である a に加えることを示している。

また、

```
a * ( b + c )
```

の場合、(b + c) が (<式>) であり、*識別子* と同じ <因子> としての扱いを受けるので、カッコ内を先に計算することになる。

4.3.2 演算子の結合性

a / b / c のように優先度の同じ演算子が並ぶ同じ場合、多くのプログラミング言語では、左側の計算を先に行う。即ち、

```
a / b / c
```

は (a / b) / c に相当し、「演算子 / は、左結合である」という。

結合性は、生成規則に表現されており、図 4.3 のように構文解析を行うことができる。

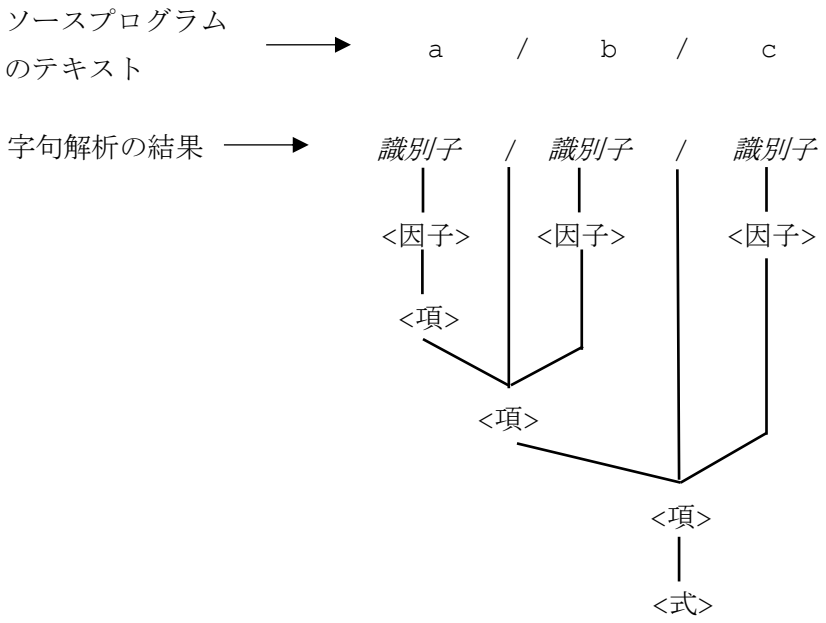


図 4.3 左結合の構文解析

逆に、代入演算子 = は、右結合である。即ち、

```
x = y = 3
```

は x = (y = 3) に相当する。即ち、先に y = 3 が行われ、次に同じ値が x に代入される。

4.3.3 中間言語

構文解析器は、コード生成に用いるため、構文解析を進めながら構文木などの中間表現を生成する。中間表現を書くための言語を中間言語という。

例）a + b * c や (a + (b * c)) の構文木は図 4.4 のようになる。

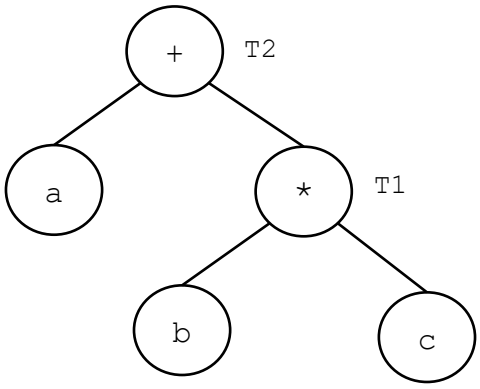


図 4.4 構文木の例

ここでは、構文木から別の中間言語の内部表現である四つ組（quadruple）の列を生成する例を示す。

ここでの四つ組は次の形を持つものとする。

（演算子，結果，被演算子$_1$，被演算子$_2$）

上記の構文木のノードに一時変数 T1, T2 を割り当てることにより、次の四つ組の列を得る。

```
(*, T1, b, c)
(+, T2, a, T1)
```

4.4 コード生成

構文解析により得られた中間表現に対して、意味解析により型の検査等が行われる。

その後、例えば四つ組が得られた後、変数にメモリやレジスタが割り当てられ、それを元にそれぞれの CPU の機械語が生成される。

例）

```
int a, b, c, y;
y = a + b * c;
```

の場合、次の四つ組の列が得られる。

```
(*, T1, b, c)
(+, T2, a, T1)
(=, y, T2,  )
```

意味解析により、変数 a,b,c,y は整数型であり、演算 *,+ は整数演算であることが分かる。

コード生成では、a,b,c,y にメモリを、T1,T2 にレジスタを割り当てることにより、以下のアセンブリ言語に対応する機械語のコードを生成する。

```
lw   $s0,-4($fp)  # b に割り当てられたメモリの値をレジスタ$s0 にロード
lw   $s1,-8($fp)  # c に割り当てられたメモリの値をレジスタ$s1 にロード
mul  $t0,$s0,$s1  # レジスタ$s0 と$s1 の値を掛け算し、結果を$t0 に格納
lw   $s2,0($fp)   # a に割り当てられたメモリの値をレジスタ$s2 にロード
add  $t1,$s2,$t0  # レジスタ$s2 と$t0 の値を足し算し、結果を$t1 に格納
sw   $t1,-12($fp) # レジスタ$t1 の値を y に割り当てられたメモリにストア
```

練習問題 3

1. 4.2 節に例示した文法について、以下の問に答えなさい。

   i) “ + ” と “ * ” はどちらの優先度が高いか。

   ii) “ * ” と “ / ” はどちらの優先度が高いか。

   iii) “ – ” は、右結合か、左結合か。

2. C 言語における右結合の演算子の例を挙げよ。

課題 I

問題 I　以下の C 言語のプログラムから字句を切り出し、3.3 節の例のように表記しなさい。

```
int max2(int i0, int i1)
{
  if (!(i0 <= i1))
    i1 = i0;
  return i1;
}
```

問題 II　以下に定義する文法を考える。

生成規則

$E \rightarrow E$ << $B \mid E$ >> $B \mid B$

$B \rightarrow$ （E） | I

$I \rightarrow$ x | y | z

非終端記号

E, B, I

終端記号

x，y，z，<<，>>，（，）

出発記号

E

字句解析後の終端記号の列

i)　x << y >> z

ii)　x << (y >> z)

のそれぞれについて、図 4.1 のように「字句解析の結果」と解析木を書き、図 4.4 に示す構文木を作成しなさい。但し、<<，>> は演算子である。

問題 III　演算子 <<，>> について、以下の問に答えなさい。

i) 2 つの演算子のそれぞれは、左結合か、右結合か。簡単な理由とともに答えよ。

ii) どちらの演算子の優先度が高いか、あるいは、双方同じか。簡単な理由とともに答えよ。

提出

講義開始時の受講案内、及び Moodle 上の指示に従って提出すること。

練習問題解答（第 1 部）

練習問題 1（第 2 講）

1.

機械語：(2.1 節参照)

（例文）メモリにロードされ CPU が直接実行できる CPU の命令セットに固有のバイナリ形式の言語

アセンブリ言語：(2.1 節参照)

（例文）機械語命令に 1 対 1 で対応するニーモニックで書かれた命令セットに固有の言語

プログラミング言語：(2.1 節参照)

（例文）人間にとって読み書きし易いテキストとして表現される計算機言語

2.

コンパイラ：(2.4 節参照)

（例文）プログラミング言語で書かれたプログラムを、機械語、あるいはアセンブリ言語のプログラムに変換するソフトウェア

アセンブラ：(2.3 節参照)

（例文）アセンブリ言語で書かれたプログラムを機械語のプログラムに変換するソフトウェア

3.

ソースプログラム：(2.2 節参照)

（例文）コンパイラやアセンブラの入力となるプログラマが書くプログラム

オブジェクトプログラム：(2.2 節参照)

（例文）コンパイラやアセンブラにより出力されるプログラム

4.

トランスレータ：(2.2 節参照)

（例文）コンパイラやアセンブラなど、ある言語で書かれたプログラムを、それと等価な別の言語で書かれたプログラムに変換するソフトウェアの総称

インタプリタ：(2.6 節参照)

（例文）ソースプログラムを 1 ステップずつ解釈して実行していくソフトウェア

5. (2.5 節参照)

（例文）複数のオブジェクトモジュールを結合し、実行可能なロードモジュールを出力する言語処理系

練習問題 2（第 3 講）

1. （3.2.1 項参照）

i)	定数リテラル（十進整数）	ii)	定数リテラル（文字定数）
iii)	文字列リテラル	iv)	文字列リテラル
v)	識別子	vi)	定数リテラル（十六進整数）
vii)	演算子	viii)	キーワード
ix)	文字列リテラル	x)	識別子

2. （3.3 項参照）

ア： `i` （一文字）

イ： `if`

ウ： 上記以外の文字列で、先頭が英文字で、それ以降が 0 個以上の英数字

即ち、アは識別子、ウは識別子あるいは `if` 以外のキーワード、イはキーワード `if`

練習問題 3（第 4 講）

1.

i) “ * ” の方が高い。

ii) 双方同じである。

iii) 左結合

2.

=、+=、-= 等の代入演算子

!、~ 等の単項演算子

第２部

オペレーティングシステム

第５講　オペレーティングシステムの概観と利用者との関わり

本講では、利用者からの視点に立ち、オペレーティングシステムの概観と我々利用者との関わりについて以下の観点から理解する。

・オペレーティングシステムの目的と基本的な役割
・計算機の様々な利用形態とそれらの相違
・どのようなオペレーティングシステムが存在するか
・エンドユーザから見たオペレーティングシステム
・プログラムの実行から見たオペレーティングシステム

5.1 オペレーティングシステム（OS）の基本的役割

5.1.1 ハードウェアの仮想化、抽象化

OS は、ハードウェアをプログラムから使いやすくするインタフェースを提供する。

例）利用者は、ファイルの名前さえ知っていれば、それを利用することができる。
データがどこにどのような形で格納されているかを知る必要はない。
利用者やプログラマは、ファイルが格納されている補助記憶装置の構造や操作法を知る必要はない。
装置に依らず、同一の方法でファイルのオープン、読み出し、書き込みができる。

5.1.2 計算資源の管理

OS は、計算資源を正しく、効率的に利用するための管理を行う。

計算資源の例

ハードウェア資源

CPU、メインメモリ、入出力装置（補助記憶装置、キーボード、マウス、ディスプレイ、プリンタ、ネットワークインタフェース等)

ソフトウェア資源

プログラム、データ、それらを格納したファイル等

計算資源の管理の例

個々のプログラムが早く終了するように、あるいは全体として多くのプログラムが実行されるように、プログラムの実行に CPU を割り当てる（効率の向上)。

ハードディスク内に、スペースの無駄がないようにファイルを格納し（効率の向上)、他人のファイルに許可なくアクセスできないようにする（正当性の保証)。

5.1.3 システムの階層構造

利用者も含めたシステムの階層構造を図 5.1 に示す。

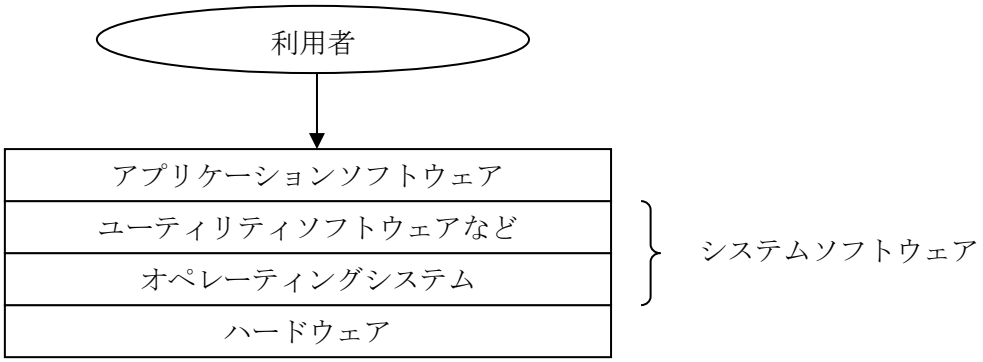


図 5.1 システムの階層構造

・オペレーティングシステム

狭義のオペレーティングシステム（核（kernel）と呼ぶ）は、ハードウェアを隠蔽し、それより上位のプログラムが直接ハードウェアにアクセスできないようにしている。

・ユーティリティソフトウェア（utility software）

ユーティリティソフトウェアには、各種コマンド、シェル（shell）などのコマンドインタプリタ、ウィンドウシステム、コンパイラ等が含まれる（古くは、OS を入手するとそれに付随してきた）。

狭義のオペレーティングシステムとユーティリティソフトウェアを合わせて、広義のオペレーティングシステムと捉える場合がある。

一般に OS というと、後者の意味で捉えることが多い。

例）CentOS, Ubuntu は後者の例で、それぞれ前者の例である Linux のディストリビューションの１つである。

例）Android も OS と捉えられているが、核は Linux である。

オペレーティングシステムとユーティリティソフトウェアの層を総称して、システムソフトウェアと呼ぶこともある。

・アプリケーションソフトウェア（application software）

アプリケーションソフトウェアは、エンドユーザが直接操作するソフトウェアである。

ワードプロセッサや文書管理プログラム、表計算プログラム（スプレッドシート）、プレゼンテーションプログラム、ウェブブラウザ、会計・経理などの事務処理プログラム、機械設計、建築・土木設計などの設計支援プログラム（CAD: computer-aided design）等

エンドユーザにとって、「パソコンができる」とは、これらアプリケーションソフトウェアが使いこなせることである。

5.2 計算機の利用形態

5.2.1 対話処理（interactive processing）

コマンド入力やマウスクリックにより要求を出し、応答を見て、次の要求を出す利用形態。
短い応答時間（レスポンスタイム）が求められる（応答性を重視する）。
時分割（タイムシェアリング）処理とパーソナル処理により実現する。
例）ソフトウェア開発、文書作成、ウェブブラウジング、銀行 ATM からの預金・引き出し・送金の操作等

5.2.2 バッチ処理（batch processing）

ひと固まりの仕事（ジョブ（job））を一括して処理する形態。実行中に利用者との相互作用はない。
スループット（5.2.4 項参照）を重視する。
ターンアラウンド時間（5.2.4 項参照）が短いことが求められるが、応答性に対する制約は厳しくない。
例）定型的な事務処理（月末締めの給与計算、毎月の公共料金の請求事務、銀行口座における毎月決まった日付に行われる給与の振り込み、公共料金の引き落としなど）
ひたすら速度が要求される大規模な数値計算（毎日の天気予報など）
← 利用者との相互作用は速度低下を引き起こすので。

5.2.3 リアルタイム処理（real time processing）

人間が相手でなく、組み込み機器等の機械が相手となる。
入力 － センサや測定器
出力 － 機器の制御
応答時間に厳しい制約がある（リアルタイム性）。
例）自動車の燃料噴射制御： 次の燃料噴射時期に燃料の噴射を、量、時期とも正確に行わなければならない。
ロボットの障害回避： 障害物を検知したら、衝突する前に回避行動を終了しなければならない。

5.2.4 性能の評価指標

・スループット（throughput）
単位時間当たりの仕事の処理量
単位：トランザクション / 秒
例）銀行口座からの公共料金の引き落としの場合、1秒間に何件の引き落としができるか。

・応答時間（response time）
利用者が要求を出して、システムが何らかの応答を返すまでの時間
例）マウスクリックからメニューを表示するまでの時間

・ターンアラウンド時間（turnaround time）
利用者が要求を出して、全ての処理が終了するまでの時間
例）コンパイルコマンド入力から、ロードモジュールが作成されるまでの時間

応答時間とターンアラウンド時間は、応答性を測る尺度である。
スループットと応答性は、必ずしも両立しない。
スループットを上げようと思えば、計算機を休ませないために、1台の計算機でなるべく多くの仕事を処理することになるが、そうすると新たに要求された仕事は長い時間待たされることになり、応答性は低下する。

5.3 OSの種類

汎用大型機（IBM等）
OS/360 → MVS
パーソナルコンピュータ
Windows
Linux （UNIX 系）
macOS（UNIX 系）
モバイル機器
iOS、Android
組込み機器

5.4 利用者から見たOS

5.4.1 システムの立ち上げと立ち下げ

システムの立ち上げ （bootstrap）
・電源 オン → ROM (read-only memory) 上のプログラム（ブートローダ）が起動
・ブートローダが補助記憶装置から IPL (initial program loader) をメインメモリへロード

・初期プログラムローディング（initial program loading）

IPL が OS を補助記憶装置からメインメモリへロード

・OS が周辺機器のテストと初期化、システム用プログラムを起動

システムの立ち下げ（shutdown）

・プログラム実行の停止

・メインメモリのデータ（特にファイル関連）を補助記憶装置に格納（保持）

・電源 オフ

5.4.2 ログインとログアウト

ログイン（login）

セッションの開始

ID とパスワード入力（利用者認証）

コマンドインタプリタ（シェル）の起動

ログアウト（logout）

セッションの終了

コマンドインタプリタ（シェル）の終了

セッション（session）

ログインからログアウトまで

その間、利用者の情報や利用状況を保持

5.4.3 利用者認証（user authentication）

計算機を利用する利用者が本人であることを確認すること

方法： パスワード、IC カード、生体認証（指紋、虹彩）など

5.5 プログラムから見た OS

5.5.1 プログラムの構成

実行中のプログラムは、メモリ上に以下の 4 つの領域を持つ。

・コード（code）領域

プログラムのロード時に静的に確保され、機械語の命令コードの列を保持する。UNIX の場合、text 領域と呼ばれ、読み出し専用の領域となっている。

・データ（data）領域

プログラムのロード時に静的に確保され、データを保持する。

C 言語の場合、`static` 変数や関数の外に定義される大域変数が割り当てられる。

これらのデータは、プログラムの開始から終了まで存在する

UNIX の場合、プログラムのロード時に、各大域変数等が、指定された初期値に初期化される data 領域と、ゼロクリアされるだけの bss 領域に分かれる。

・ヒープ（heap）領域

プログラムのロード時に程々の大きさで確保される。プログラムの実行中に必要に応じて動的に確保・解放したいメモリ領域は、このヒープ領域に割り当てられる。

C 言語の場合、実行時に大きさを指定できる動的なメモリ領域は `malloc()` により確保され、`free()` により解放される。

・スタック（stack）領域

プログラムのロード時に程々の大きさで確保される。関数への引数、関数内の一時的な局所変数などは、関数コール時に連続した領域となるよう、このスタック領域に自動的に確保され、リターン時に自動的に解放される。

この関数呼び出しごとに確保される連続した領域をフレームと呼ぶ（表 11.1 のフレームポインタは、現在実行中の関数のフレームを指し示すレジスタである）。

即ち、フレームは、関数のコールからリターンまで存在する。

図 5.2 に、5 つそれぞれの領域のメモリ空間内での相対的な位置を示す。
名称は UNIX による。スタック領域とヒープ領域は拡張可能である。

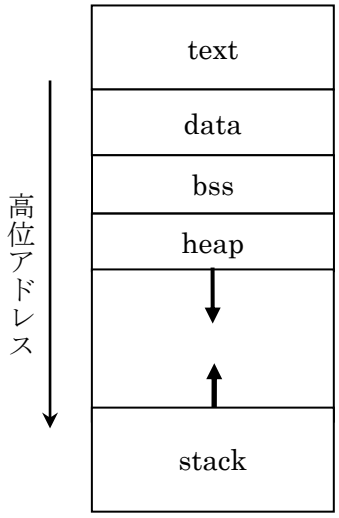


図 5.2 メモリ空間内の 5 つの領域の割り当て

5.5.2 システムコール（system call）

スーパーバイザコール（supervisor call: SVC）、カーネルコール（kernel call）ともいう。
プログラム（プロセス）から、OS へサービスを要求するインタフェース
計算資源への直接のアクセスを OS にのみに許可し、利用者のプログラム（プロセス）は、OS を介して資源へアクセスする。

仮想化・抽象化されたハードウェアへのアクセスの実現

資源の保護の実現

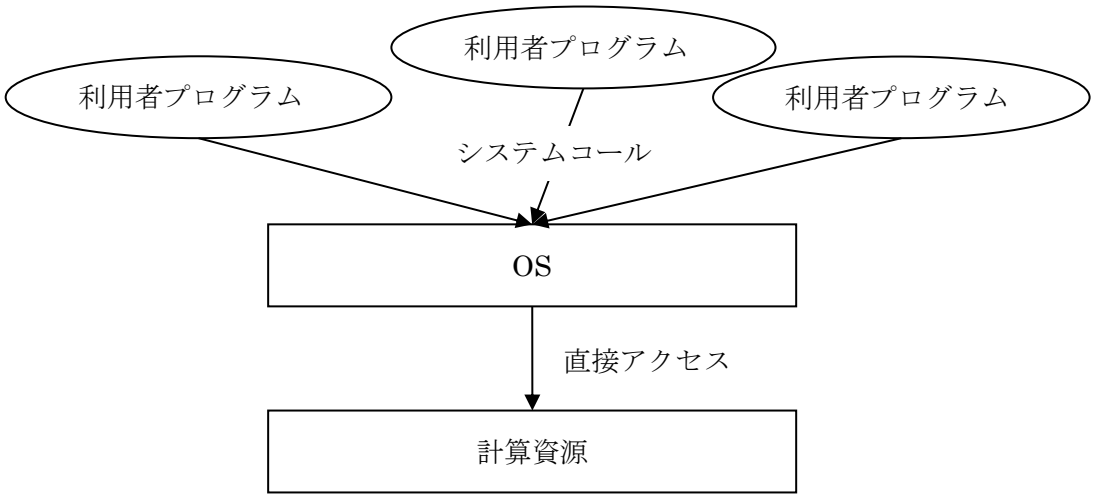


図 5.3 システムコール

5.5.3 システムコールの種類

・ファイル関係

生成、削除、オープン、クローズ、名前の変更、読み出し、書き込み

例）UNIX における read システムコール

```
read(fd, buf, size)
```

fd: オープンされたファイルをプログラム内で識別するファイルディスクリプタと呼ばれる番号（整数）

buf: ファイルから読み込んだデータを格納するプログラム内のバッファへのポインタ（格納先の先頭メモリアドレス）（7.2.2 項参照）

size: 読み込むべきデータの大きさ　バイト単位

・その他の入出力要求等の入出力装置の操作

・プロセス関係

生成、消去、同期・通信

・メモリの割り当て要求（ヒープ領域の拡張など）

・スリープ（実行待機）

・日付、時刻、システム情報（ネットワークアドレスなど）管理

練習問題 4

1. 以下の計算機の処理形態は、どのような利用形態に適するか。具体的な例とともに述べなさい。
   i) 対話処理
   ii) バッチ処理
   iii) リアルタイム処理
2. 以下の各領域には、何が割り当てられるか（何が確保されるか)。C 言語を例として答えなさい。
   i) データ領域
   ii) ヒープ領域
   iii) スタック領域
3. ログインの必要性を述べなさい。
4. システムコールの必要性を述べなさい。

第６講　ファイルシステム

本講では、オペレーティングシステムの構成要素の中で、最も利用者に身近なファイルシステムを以下の観点から理解することを目的とする。

・ファイルとは何か
・ファイル名の付け方とその管理法
・ハードディスクの構成
・ファイルシステムの仕組み
・資源保護のためのアクセス制御

6.1 ファイル (file) とは

関連するデータの集まり　データを保管・管理する入れ物

仮想的、抽象的補助記憶装置

ファイルシステムの役割

媒体上の物理的な領域割り当て、検索、データの入出力

補助記憶装置の種類

ハードディスク（ディスク装置、HDD とも呼ばれる)、フラッシュメモリ（SSD、USB メモリ、SD カードなどの種類がある)、CD、DVD など

6.2 ディレクトリと名前付け

6.2.1 ファイル名

ファイル名	.	拡張子

固定長ファイル名　MVS、MS-DOS 等、古い OS

可変長ファイル名　UNIX 系、Windows 等

6.2.2 ディレクトリ（directory）

元来はファイルの管理簿の意味。現実的には、ファイルをグループ化するための特殊なファイル

ディレクトリ内にディレクトリを作成することにより、階層的なディレクトリ構造を作ることができる。

Windows では、ディレクトリに対応するものをフォルダと呼ぶ。

6.2.3 階層的なディレクトリ構造

関連するファイルをグループ化、階層化する（図 6.1 参照)。図 6.1 においては、ディレクトリを四角、ファイルを楕円で表現している。

階層的なディレクトリ構造の利点

・階層的な名前付けにより、柔軟性を保ちつつ名前の一意性を保証する（名前の衝突を防ぐ)。

同じファイル名でも、ディレクトリが異なれば、別のファイルである。

・利用者がファイルを整理し、検索するのが容易

・アクセス制御、コピー、削除等、関連するファイルの一括操作が可能

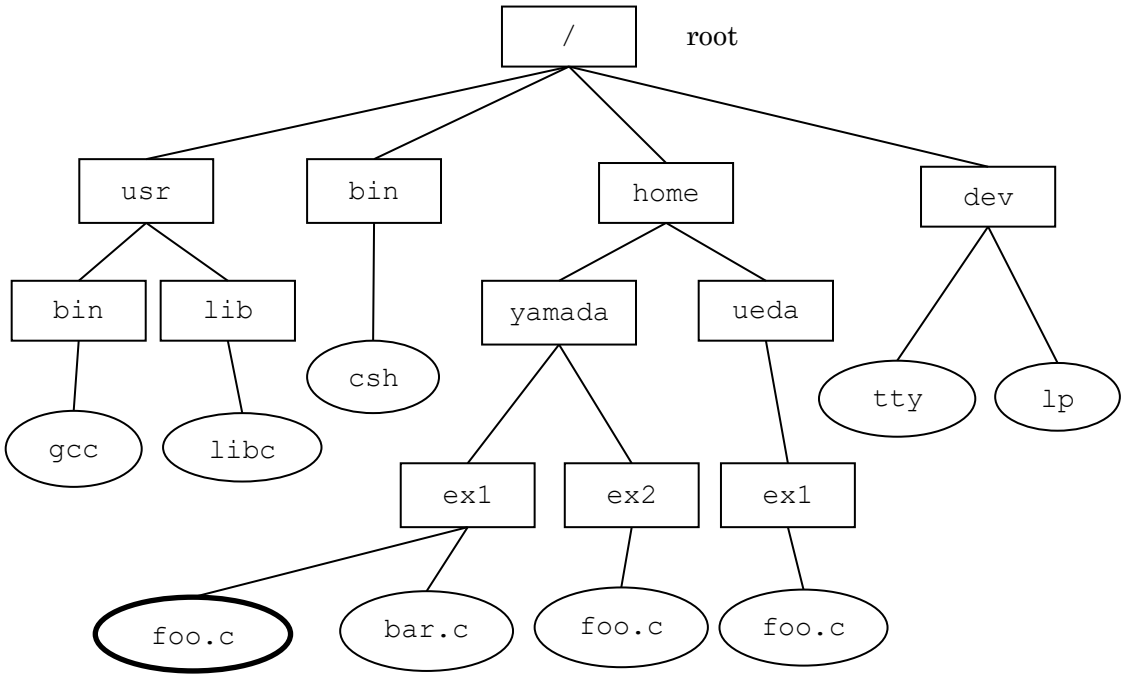


図 6.1 UNIX の階層的ディレクトリ構造

6.2.4 階層的名前付け

階層的なディレクトリ構造を用いることにより、ファイルやディレクトリに対して階層的な名前付けができる。

UNIX の例

・絶対パス名　root (/) からたどる経路で表す。

例）　ディレクトリ：　/home/yamada/ex1　（または、/home/yamada/ex1/）

ファイル：　/home/yamada/ex1/foo.c

・相対パス名　カレントディレクトリ (current directory)からの経路

（現在作業をしているディレクトリ）

例）カレントディレクトリが /home/yamada のとき
/home/yamada/ex1/foo.c の相対パス名は ex1/foo.c
または ./ex1/foo.c

“.” カレントディレクトリを示す。

“..” カレントディレクトリの親ディレクトリを示す。

例）カレントディレクトリが /home/ueda/ex1 のとき
/home/yamada/ex1/foo.c の相対パス名は、例えば
../../yamada/ex1/foo.c
./../../yamada/ex1/foo.c
./../../../yamada/./././ex1/foo.c （例のための例）

login 時のカレントディレクトリをホームディレクトリと呼ぶ

6.3 補助記憶装置の構成

6.3.1 データのディスク上での位置

図 6.2 に簡略化したハードディスクの構造を示す。ディスクの記録面を同心円状に分割した一周分の記録領域はトラックと呼ばれる。複数の記録面（複数のディスクの両面）は共通の回転軸を持ち、ある半径の円筒上に位置するトラックの集合はシリンダと呼ばれる。

データは、ブロックと呼ばれる（物理レコード、セクタともいう）トラックを分割した固定長の領域に格納される。

ブロックは物理的入出力の単位となる（ハードディスクへの 1 回の操作で読み書きされる）。

古くは〔シリンダ、トラック、ブロック〕の組がブロックを特定するのに用いられた。

現在は、ブロックの識別に Logical block addressing (LBA) と呼ばれるブロックの一意の通し番号が広く用いられる。

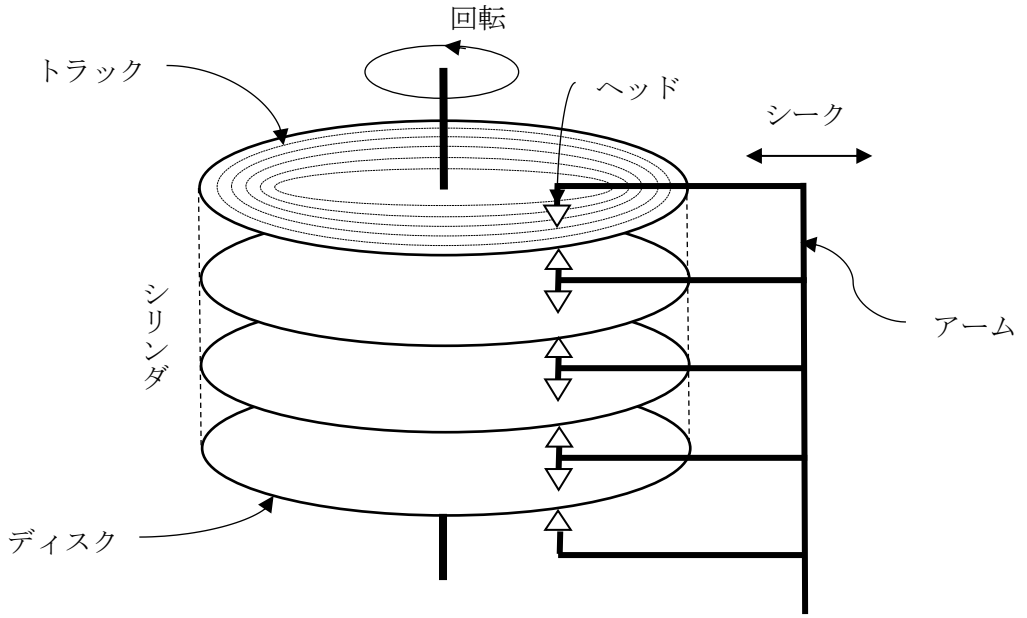


図 6.2 ハードディスクの構造

6.3.2 ディスクに対するアクセス手順

1. シーク（seek） アームを移動させてヘッドを指定されたシリンダに位置付ける。
   平均シーク時間 10ms 前後
2. トラックの選択　電気的にヘッドを選択する。
   時間は無視できるほど短い。
3. 回転待ち　指定されたブロックがヘッドの位置まで回転するのを待つ。
   平均回転待ち時間は、ディスクが半回転する時間

6.3.3 アクセス時間

入出力要求から入出力完了までの時間

ハードディスクの場合

アクセス時間 ＝ シーク時間 ＋ 回転待ち時間 ＋ データ転送時間

6.4 ファイルシステム

OS の重要な構成要素のひとつ

ファイルを標準的な方法で統一的に管理する仕組み

利用者からは、名前からファイルを検索する機構

ここでは、UNIX 系の OS を例に挙げる。

6.4.1 領域割り当て

一つのファイルを構成する複数のブロックの保持法

UNIX 系の例

UNIX では、inode と呼ばれる構造体にファイル管理に必要な情報を格納し、更にそのファイルを構成するブロックを指し示す索引を保持する。

inode には inode 番号と呼ばれる一連の番号が付されている。

inode に格納されるファイル管理に必要な情報

・保護情報（アクセス制御に用いる）（6.5.2 項）

・ファイルの型

通常のファイル、ディレクトリ、ブロックデバイス（ハードディスク等)、キャラクタデバイス（キーボード等)、ネットワークインタフェース等の種別

注）UNIX においては、入出力装置も論理的にはファイルとして扱われる。

・所有者 ID、グループ ID（利用者やグループに割り当てられた整数値）

・時刻印（最終更新時刻、最終参照時刻等）

・大きさ（バイト単位、使用ブロック数）

・物理的位置への索引（図 6.3 参照）など

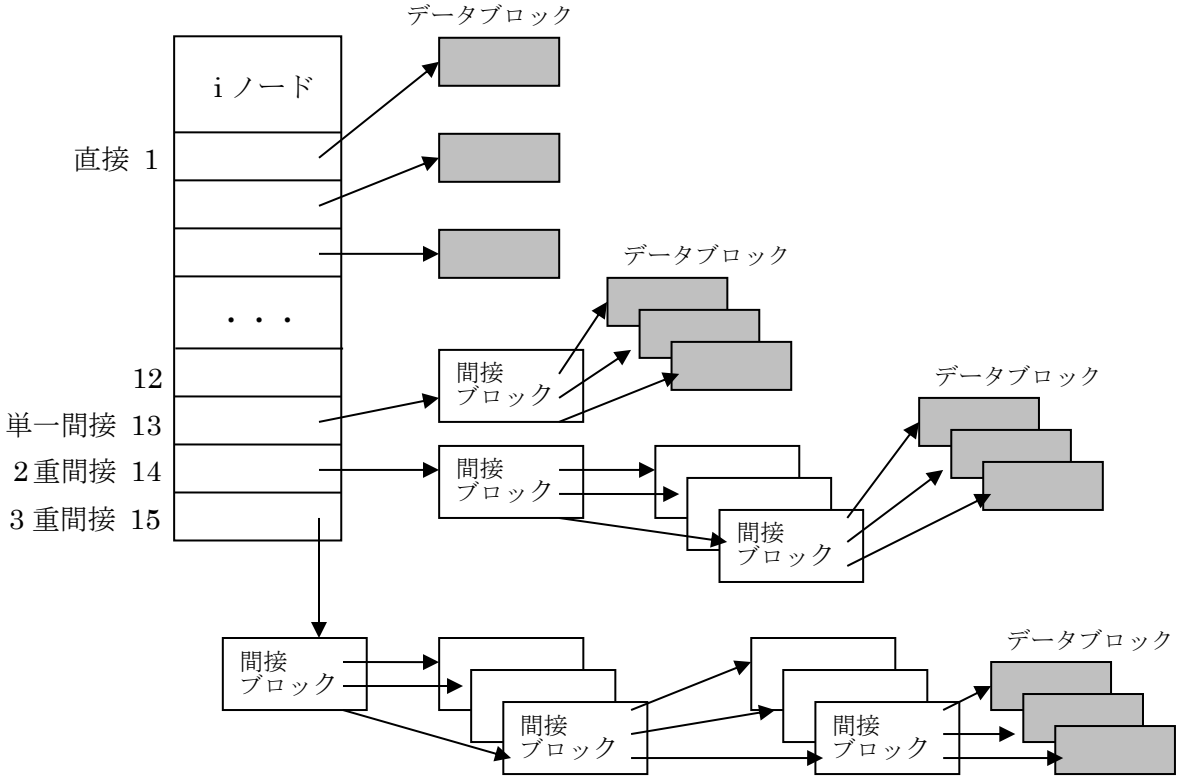


図 6.3　inode 内のデータブロックへの索引

6.4.2 ディレクトリの実現

UNIX におけるディレクトリは、そのディレクトリ内のファイルの名前と inode 番号の対を記録したファイルとして実現される。

ディレクトリに、その下位にあるディレクトリの名前と inode 番号を記録することにより、階層的なディレクトリ構造を構成できる。

6.4.3 空き領域管理

ファイルシステムは、使用されていないブロックや、ファイル消去により不要となったブロックの管理をおこなう。

UNIX の場合、リスト構造で管理する。

6.4.4 ファイルの生成

付けられたファイル名と新たに確保された inode の番号をディレクトリに登録し、inode に管理情報を記録する。

ファイルの格納に当たっては、空きブロックリストから空きブロックを確保し、そのブロックのアドレスを inode に索引として登録して、データをブロックに書き込む。

6.4.5 ファイルの検索

ディレクトリから該当するファイル名を検索し、それに対応する inode 番号から inode を特定する。

その inode の索引をたどってデータブロックを見つける。

6.4.6 ファイルの消去

ファイル名をディレクトリから消去し、使用していた inode を解放し、データブロックを空きブロックリストに返す。

データブロック内に書かれたデータは、そのブロックが他のファイルのために利用されて上書きされるまでそのまま残る。

6.5 アクセス制御

6.5.1 アクセス制御

アクセス制御（access control）とは、プログラム（正確には、プロセス）が資源にアクセスするとき、その正当性を保証すること

〔アクセスを行う主体，アクセスされる対象，アクセス権〕の組を規定する。

アクセス権とは、許可される操作をいう。

即ち、アクセス制御とは、例えば、あるプログラムが、あるファイルに、例えば書き込みを行おうとするとき、それを許可するか、拒否するかの判定を行うこと。

6.5.2 アクセス制御リスト

アクセス制御リストとは、ファイル（対象）ごとに定められる〔利用者（主体），許可される操作（アクセス権)〕のリスト

UNIX における例（9-bit で、 inode 内に格納される）

所有者　　グループ　　その他

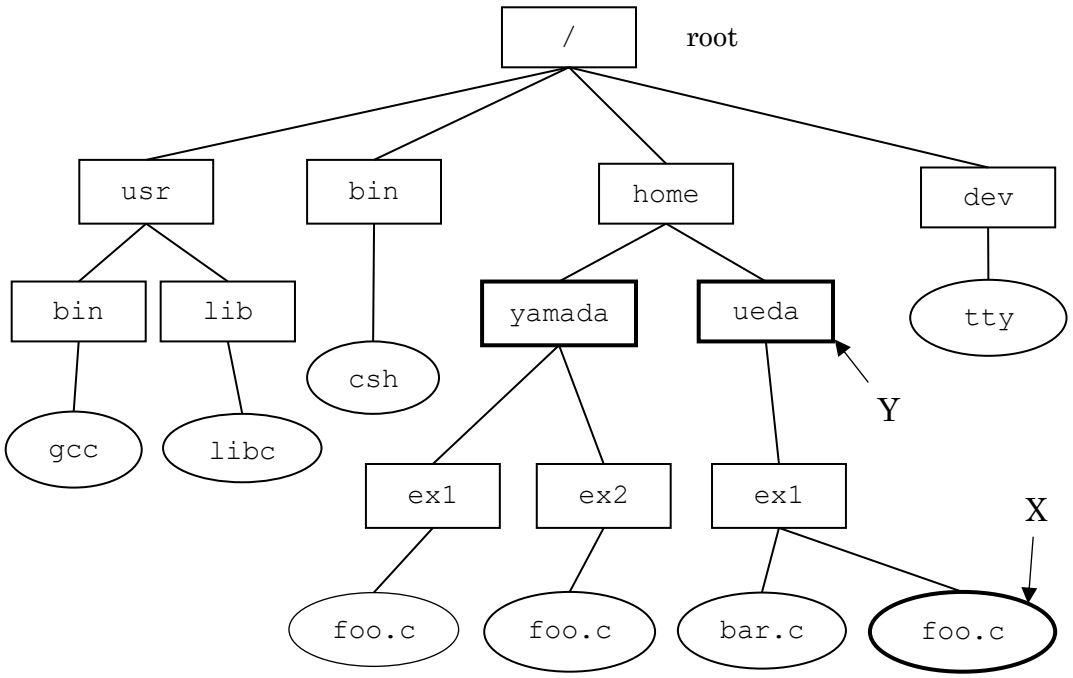
r w x　　r w x　　r w x

r: 読出可，w: 書込可，x: 実行可

ファイルのオープン時（あるいは実行時）に、所有者 ID 等とアクセス制御リストに基づいてアクセス権のチェックが行われる。

練習問題 5

1. 補助記憶装置の例を挙げなさい。
2. 下図の木構造ディレクトリに関して、以下の問に答えよ。
   i) X のファイルの絶対パス名は何か。
   ii) カレントディレクトリが Y であるとき、X の相対パス名は何か。
   iii) カレントディレクトリが /home/yamada のとき、X の相対パス名は何か。



3. 1 分間 7,200 回転（7,200rpm）のハードディスクにおいて、 平均回転待ち時間を有効数字 2 桁まで求めよ。
4. ファイルをファイルシステムから消去しただけでは、ファイルのデータを復元できる場合がある。その理由を述べなさい。

第７講　入出力

本講では、入出力に関するオペレーティングシステムの役割を以下の観点から理解する。ここは、ハードウェアとソフトウェアの境目に当たる。

・CPU と周辺装置との関係としての割り込みについて、その目的、種類、仕組み
・入出力データを一時的にメインメモリ内に格納するバッファリング
・プログラムからハードウェアまでの階層構造

7.1 割り込み

割り込み（interruption）とは、
CPU の命令実行とは独立に、あるいはその実行結果として発生する事象を捉える手段

7.1.1 割り込みの種類

例）

1. 入出力割り込み
   入出力装置の動作完了、誤動作など
2. （狭義の）外部割り込み
   タイマの時間切れ等
3. マシンチェック割り込み
   CPU 誤動作、電源異常など
4. リスタート割り込み
   計算機を再起動させるときに発生させる。
5. プログラム割り込み
   プログラムの実行時エラーにより発生
   オーバーフロー、ゼロ除算、メモリアクセス保護違反、特権命令違反等
6. SVC 割り込み
   システムコールを実行しようとするとき発生させる。

外部割り込み（広義の）
命令の実行とは独立して発生する。上記の１ 〜 4
内部割り込み（トラップ、割り出しともいう）
命令の実行の結果発生する。上記の 5, 6

割り込み処理ルーチン
割り込みに対して適切な処理を行う OS の核内のプログラム
特権モード（privileged mode）

特権命令が実行できるモード　OS の核の実行モード

非特権モード

特権命令が実行できないモード　通常のプログラムの実行モード

特権命令

入出力操作、仮想記憶管理、割り込みに関する命令（OS の核のみが実行できる）

特権モードと非特権モードに分けることにより、資源への直接アクセスを OS のみに許すことが可能となる。

割り込みとは、非特権モードから特権モードへの移行の手段と捉えることもできる。

7.1.2 割り込みの仕組み

割り込みが発生すると、図 7.1 に示すように、利用者プログラムから OS の核に実行制御が移り、実行モードが非特権モードから特権モードに移行する。

割り込み処理終了後は、利用者プログラムへ復帰する。

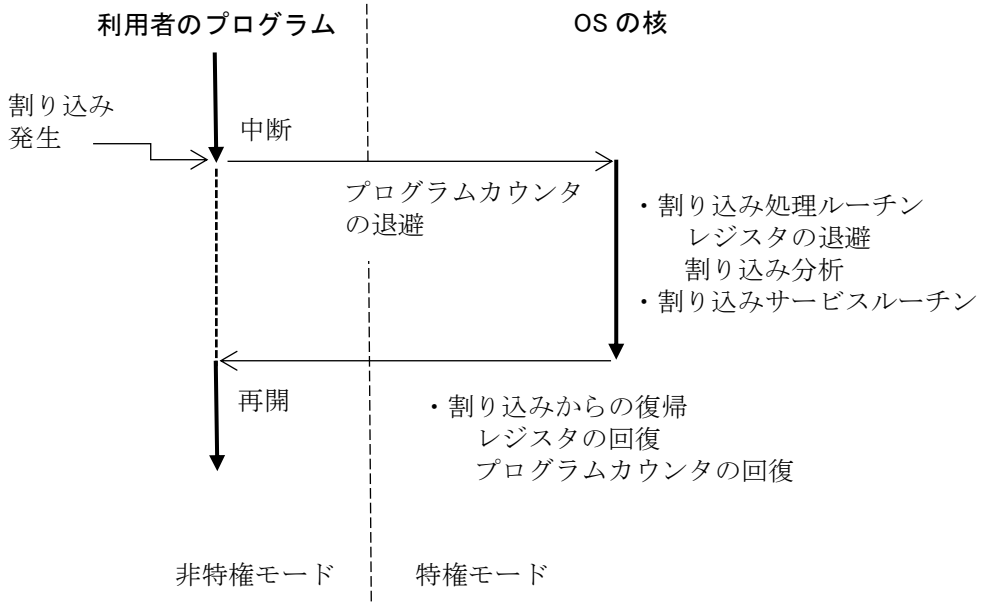


図 7.1 割り込みの仕組み

7.1.3 システムコールの実現

システムコールはサブルーチンコールでなく割り込みとして実現される。

・非特権モードから特権モードへの移行

・異なるアドレス空間への移行

7.1.4 OS のカーネル（核、kernel）

OS の中心となる部分で、最も狭い意味での OS を指す。

以下の特徴を持つ。

・特権モードで動作する。

・仮想記憶管理（第 9 講参照）の対象とならず、メインメモリに常駐する。

・スケジューリング（第 8 講参照）の対象とならず、OS の立ち上げ後は割り込みによって起動される。

7.2 バッファリング

バッファ（buffer）

入出力データを一時的に保持するためにメインメモリ上に確保された領域

バッファリング（bufferig）

バッファを用いて入出力の効率を上げる技法

7.2.1 バッファによる入出力効率の向上

1. 入出力のデータの大きさの調整

プログラムの入出力単位　比較的小さい

ハードディスクなどの入出力単位　比較的大きい

2. メモリアクセスと入出力の速さの差

メモリアクセスは速い

入出力は遅い

→　待ち合わせ時間の解消

1, 2 の結果、遅い入出力装置に対する度々のアクセスによる待ち合わせ時間を解消できる。

3. キャッシュ（cache）としての効果

データを共有する場合や再利用する場合に効果が大きい。

一般的に、OS の核内に特定の領域を設け、多数のバッファを予め用意している（バッファプール）。これはキャッシュとしても利用できるので、UNIX では、バッファキャッシュと呼ぶ。

7.2.2 ダブルバッファリング

OS が核内に用意する入出力用のバッファと、入出力ライブラリがプログラム内に用意する CPU のデータ処理用のバッファの２つのバッファを設けること（図 7.2 参照）

出力の例

プログラムからの標準出力関数を用いた出力は、プログラム内のバッファに書き込みが行われる（変数の領域からバッファの領域へのデータのコピー）。

このバッファが満杯になると、write システムコールにより、核内のバッファにデータがコピーされる。

核内のバッファからのハードディスクへの実際の出力は後に行われるが、それまでの間、プログラム内のバッファに上書きができる。

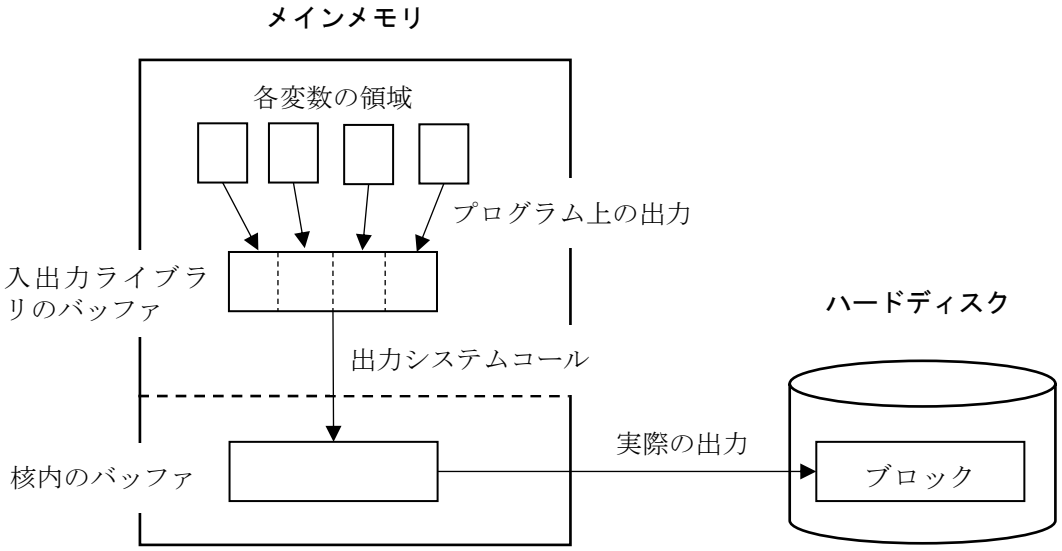


図 7.2 ダブルバッファリング

7.3 ファイルと入出力の階層

入出力を行うソフトウェアの階層を図 7.3 に示す。

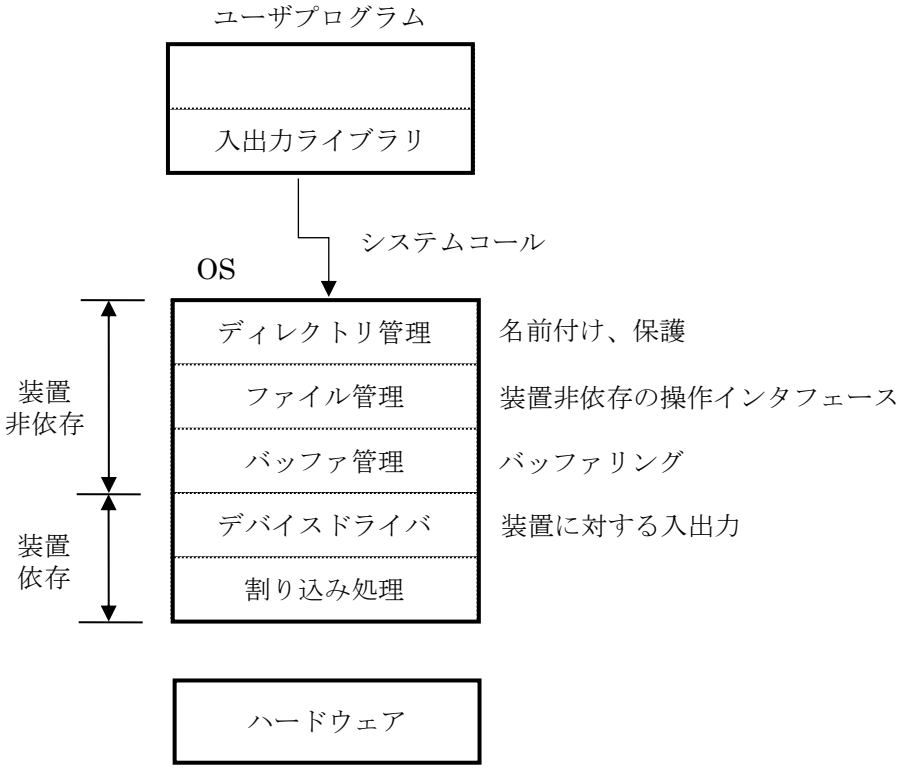


図 7.3 入出力を行うソフトウェアの階層

デバイスドライバ (device driver)

OS 中で、入出力装置（デバイス）に依存した部分

装置ごとにインストールされる。

デバイスコントローラに指示を与える。

練習問題 6

1. 割り込みについて、以下の問に答えなさい。

i) 外部割り込みと、内部割り込みの相違を述べなさい。

ii) 外部割り込みの具体的な例を挙げなさい。

iii) 内部割り込みの具体的な例を挙げなさい。

2. 入出力のためにバッファを用いることが入出力の効率向上に寄与する理由を 3 点挙げなさい。

第８講　プロセス管理とスケジューリング

本講では、CPU、メインメモリ、入出力装置等の計算資源を共有しながら並行にプログラムを実行していく仕組みであるプロセスについて、以下の観点から理解する。

・プロセスとは何か
・並行に動いている複数のプロセスの実行を切り替える仕組み
・プロセスの実行に関する状態とそれらの遷移
・複数のプロセスの実行順序を決めるスケジューリング

8.1 プロセス（process）とは

解釈が異なるだけで、実体としては同じもの

・原始的解釈　計算機内で実行中のプログラム
・古典的解釈　仮想的 CPU と仮想記憶からなり、プログラムを実行する主体（図 8.1）
　スレッド　実行制御の流れ　＝　プロセスから切り離した仮想的 CPU

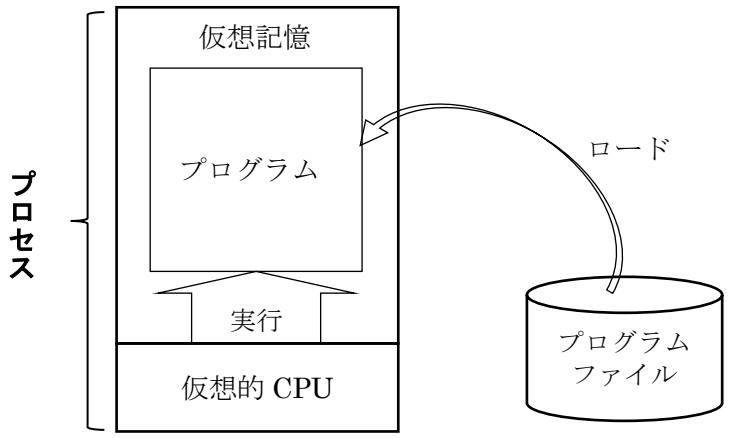


図 8.1 プロセスの古典的解釈

・現代的解釈　利用者が依頼した作業を、利用者の代理として計算機内で実行する主体（図 8.2）
　動作はプログラムに記述された内容に従う。
　プログラム実行に際し、OS に資源を要求する。

複数のプロセスが並行に動作する。

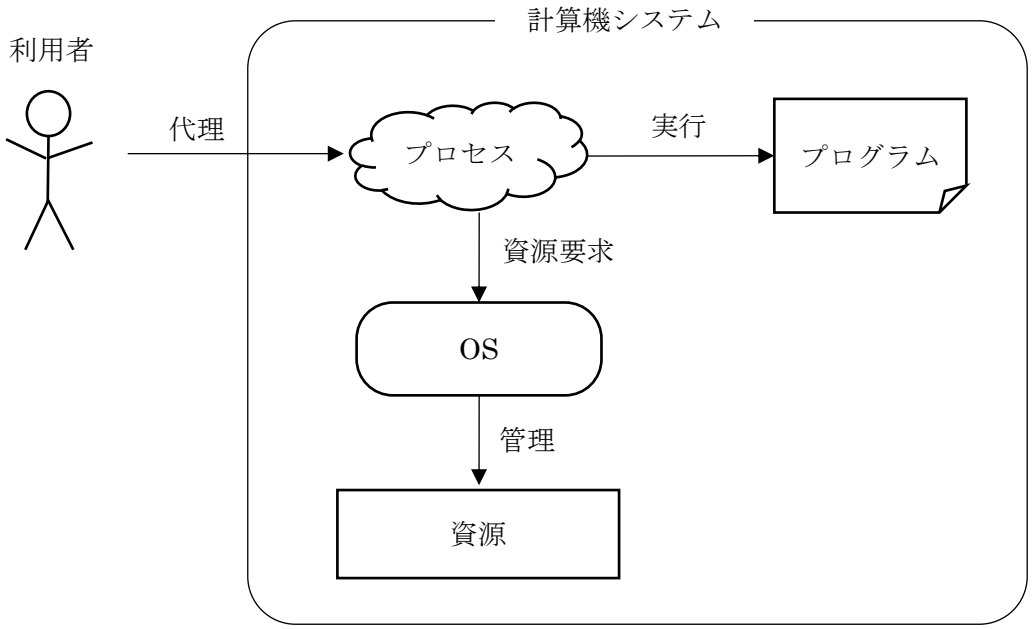


図 8.2 プロセスの現代的解釈

8.2 プロセスの切り替え

複数のプロセスを並行に実行するためのプロセス切替えの発生する契機

・入出力要求を出したプロセスは、入出力が完了するまで CPU を必要としないので、CPU を他のプロセスに譲る。

・時分割処理において、定時間（quantum time あるいは time slice）が経過すると、CPU は他のプロセスに割り当てられる。

例）プロセス A, B, C が時分割多重により CPU を共有する様子を図 8.3 に示す。各プロセスには、性能は劣るがあたかも自分専用の CPU を持っているかのような効果がある（CPU の仮想化）。

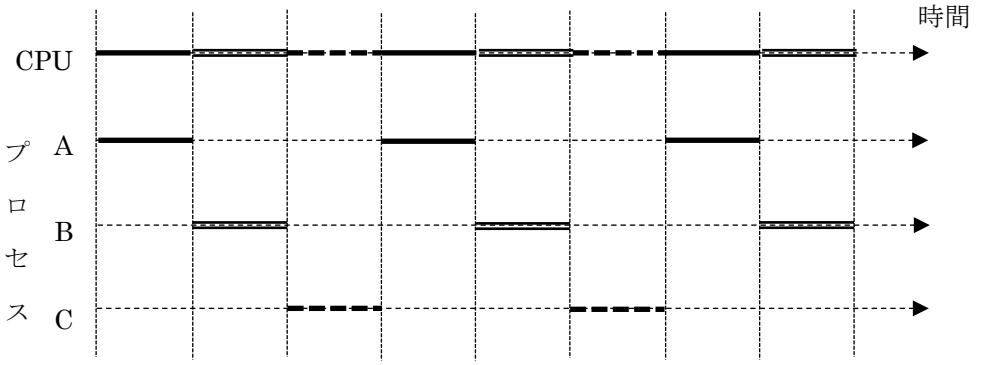


図 8.3 CPU の時分割多重

プロセス切替えの手順

図 8.4 に、プロセスの切り替え（コンテクストスイッチ（context switch）ともいう）の手順を示す。

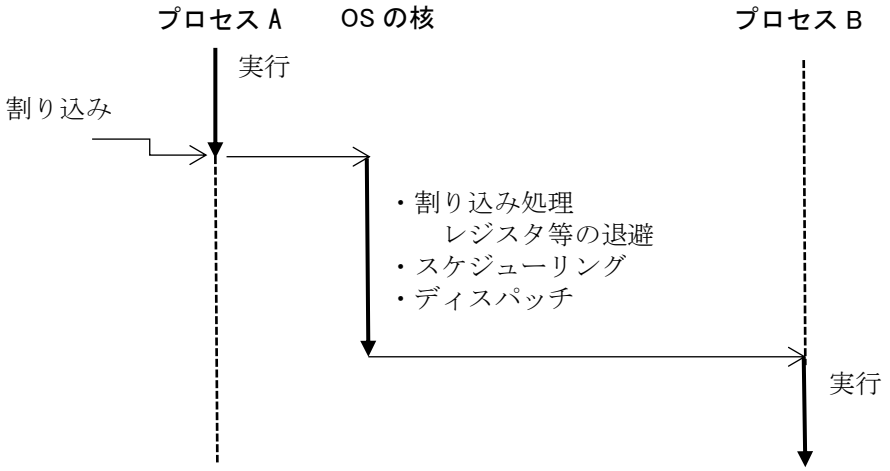


図 8.4 プロセス切り替え

1. 入出力システムコールの発行や、タイマ割り込みなどの割り込みによりプロセス A の実行が中断される。
2. 通常の割り込み処理により、プロセス A の（プログラムカウンタを含む）レジスタ等の実行状態が退避される。
3. CPU スケジューラ（scheduler）により、次に実行するプロセス B を選択する。プロセスの実行順序を決め、次に実行すべきプロセスを選択することを CPU スケジューリング（scheduling）という。
4. ディスパッチャ（dispatcher）により、スケジューラの選択したプロセス B を起動する。これは、先にプロセス B の中断に際して退避した状態を回復して再開することになる。これをディスパッチ（dispatch）という。

8.3 プロセスの状態

プロセスの状態と状態遷移を図 8.5 に示す。

・実行状態（running state） CPU が割り当てられてプログラムを実行している状態

・実行可能状態（ready state） CPU が割り当てられさえすれば、プログラムを実行できる状態

新たに生成されたプロセスは、まずこの状態になる。

他のプロセスが CPU を保持しているので、CPU が空くまでこれらのプロセスは実行可能待ち行列（ready queue）につながれる。

・待ち状態（waiting state） システムに発生する何らかの事象発生を待っている状態。

CPU を必要としていない。

事象発生待ちの例

入出力完了待ち、タイマの時間経過待ち、他のプロセスとの同期等

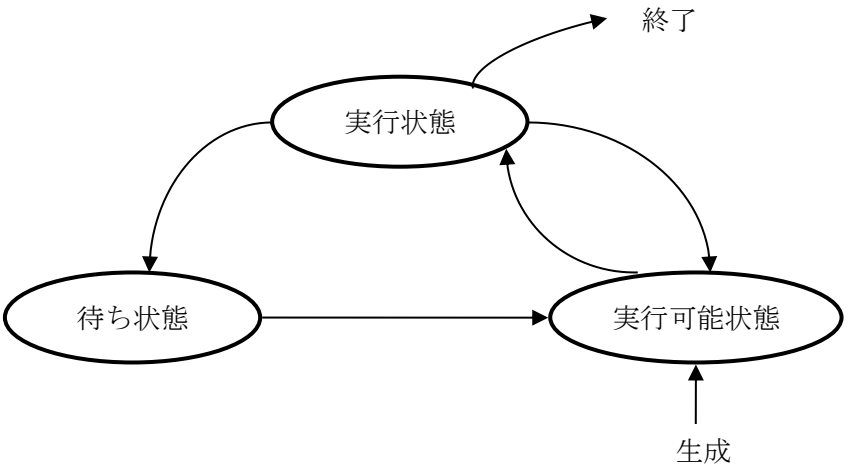


図 8.5 プロセスの状態遷移図

プロセスの状態遷移

実行状態	→	実行可能状態	CPU の解放
実行可能状態	→	実行状態	CPU の割り当て
実行状態	→	待ち状態	事象の発生待ち（CPU は解放）
待ち状態	→	実行可能状態	待っていた事象の発生

プロセス生成と切り替え、及び状態遷移の例（図 8.6 参照）

shell（シェル，殻の意味から）は、コマンドが入力されるとコマンドを解析し、新たな実行可能状態のプロセス（子プロセスと呼ぶ）を生成してそのプロセスにコマンドを実行させる。

shell（この場合は、親プロセスとなる）は、子プロセスの終了まで待ち状態となる。

子プロセスは、CPU が割り当てられている間は実行状態となり、コマンドプログラムをロードし、その実行を終了すると、親プロセスへその旨を通知する（プロセスの同期）。

shell は、子プロセスの終了通知を受けて実行可能状態となり、CPU が割り当てられると実行状態となる。

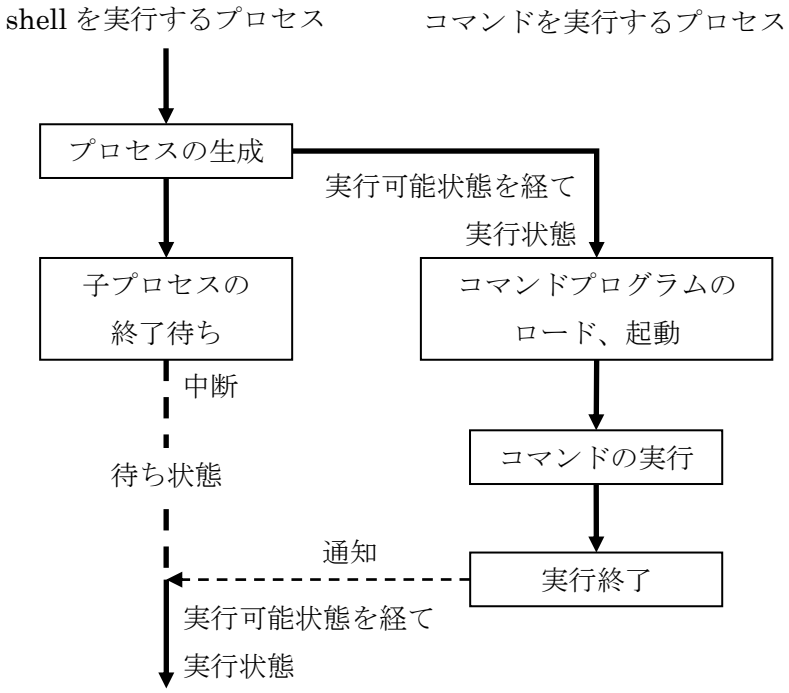


図 8.6 shell におけるプロセス生成

8.4 スケジューリングアルゴリズム

8.4.1 スケジューリングの評価基準

1. スループット（処理効率）
2. 応答性

   バッチ処理　ターンアラウンドタイム

   対話処理　応答時間（レスポンスタイム）

   最大値を最小にするのか、平均値を最小にするのか
3. 公平性
4. 応答時間の予測可能性

   リアルタイム処理では重要

8.4.2 基本的なスケジューリングアルゴリズム

基本的なスケジューリングアルゴリズムの例を挙げる。

a）到着順（FCFS: first-come first-served）（図 8.7）

新たに実行可能になった順。到着したプロセスは、ready queue の最後尾につながれる。最も単純なアルゴリズム

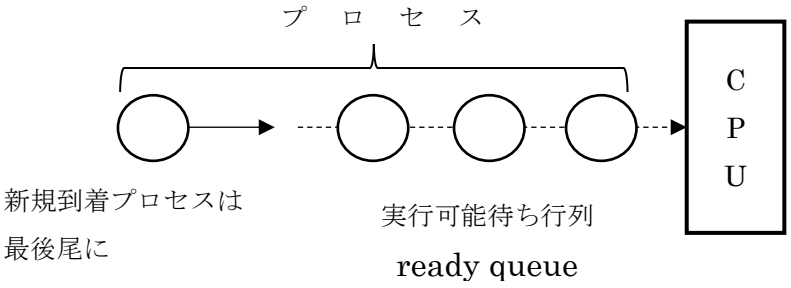


図 8.7 到着順

b) ラウンドロビン（round robin）（図 8.8）

プロセスは、割り当てられた定時間（quantum time, または time slice）経過すると ready queue の最後尾へ移される。

公平に CPU を割り当てられる（全てのプロセスに一定の応答時間を保証)。

対話処理に用いられる時分割（タイムシェアリング）処理向け

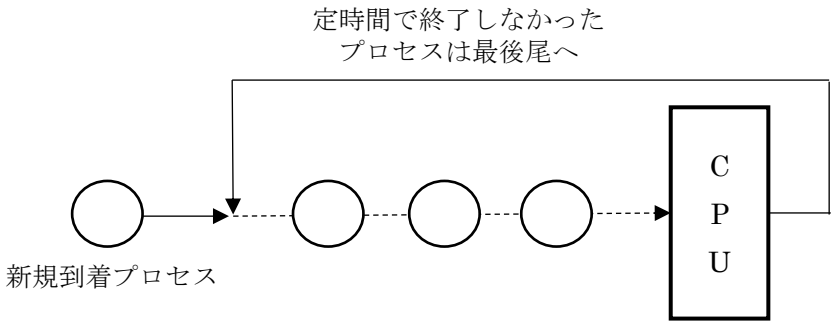


図 8.8 ラウンドロビン

例）

定時間を 2 としたラウンドロビンによる各プロセスへの CPU の割り当て

図 8.9 は、横軸を時間としたタイミングチャート

プロセス A, B, C は、以下の表に示す時刻にシステムへ到着し（△)、それぞれの処理時間 CPU を割り当てられ（太線)、処理を終了する（▲)。

ターンアラウンド時間は、プロセスの到着から終了までの時間

仮定： この間新たなプロセスの到着はない。

各プロセスは入出力を行わない。

プロセス切り替えの時間は無視

プロセス	到着時刻	処理時間
A	0	10
B	3	12
C	7	6

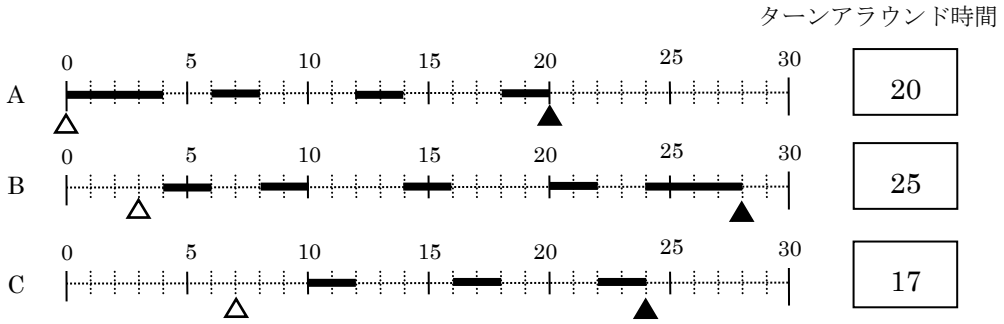


図 8.9 ラウンドロビンにおける CPU 割り当て

c) 優先度順（priority）（図 8.10）

プロセスは、実行の優先度を与えられ、優先度別の ready queue へつながれる。
優先度の高い ready queue につながれたプロセスから順に実行される。

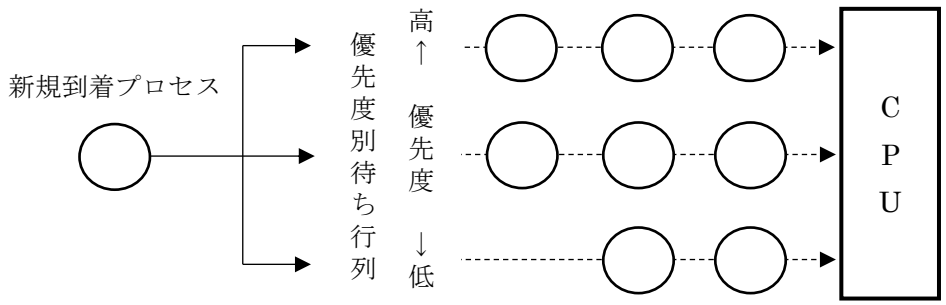
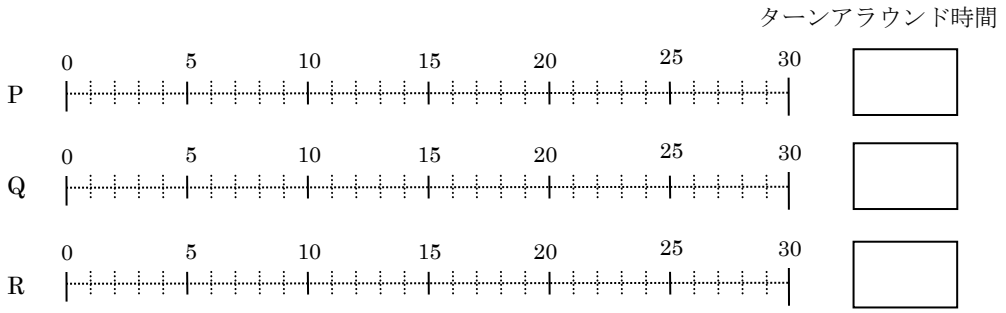


図 8.10 優先度順

練習問題 7

1. 実行可能状態と待ち状態の相違を述べなさい。
2. プロセスが待ち状態から実行可能状態へ状態遷移する契機となる事象の具体例を挙げなさい。
3. 定時間を２としたラウンドロビンによるプロセスのスケジューリングを行う。下の表に示す時刻に３つのプロセスがシステムに到着し、表に示す処理期間を CPU で消費する。下図のタイミングチャートに各プロセスのシステムへ到着時刻（△)、CPU を割り当てられている時間（太線)、終了時刻（▲）を記入し、各プロセスのターンアラウンド時間を求めよ（P, Q, R の ready queue 内での順番に注意すること)。但し、この間新たなプロセスの到着や入出力はなく、プロセス切り替えの時間は無視するものとする。

プロセス	到着時刻	処理時間
P	0	10
Q	1	8
R	3	12



第９講　仮想記憶

本講では、メインメモリの容量より大きな、個々のプロセス専用の仮想的な記憶装置を提供する仮想記憶について、以下の観点から理解することを目的とする。
・仮想記憶とは何か
・代表的な仮想記憶の技法であるページングの仕組み

9.1 仮想記憶（virtual memory）とは

9.1.1 仮想記憶の目的

・プロセスにメインメモリの大きさにとらわれないより容量の大きい仮想的な記憶装置を提供する。
　メインメモリの容量を気にせずに、大きなプログラムをロードでき、大きなデータを扱える。
・プロセスごとに、それ専用の一連の仮想的なアドレスが振られたアドレス空間を提供する。
　他のプロセスのプログラムの領域を侵すことがない（記憶領域の保護）。
　一般的にロードモジュールは、仮想アドレスが振られている（リンクの段階で静的なアドレスは決定できる）。

9.1.2 仮想記憶の実現

メインメモリと補助記憶装置を組み合わせる。
　プログラム全体がメインメモリにロードできない場合、今必要なプログラムの一部をメインメモリに置く。
　残りを補助記憶装置に置き、必要な時にはメインメモリにロードする。
仮想アドレス空間（virtual address space）
　仮想記憶にアクセスするためのアドレスを仮想アドレスという。
　仮想アドレスにより形成されるアドレス空間を仮想アドレス空間と呼ぶ。これはプログラムに割り当てられた論理的なアドレス空間である。
　プログラムは、仮想アドレスを用いて仮想記憶にアクセスする。
物理アドレス空間（physical address space）
　メインメモリにアクセスするためのアドレスを物理アドレスという。
　物理アドレスは、プログラムがロードされた際のメインメモリのアドレスで、プログラム実行時に決定される。
　物理アドレスにより形成されるアドレス空間を物理アドレス空間と呼ぶ。

後述のアドレス変換機構が仮想アドレスから物理アドレスへの変換を行い、メインメモリへのアクセスが実現される。

9.2 ページング (paging)

仮想アドレス空間と物理アドレス空間をページと呼ばれる固定長の記憶領域に分割し、ページ単位で仮想アドレスと物理アドレスをマッピングする技法（図 9.1 参照)。

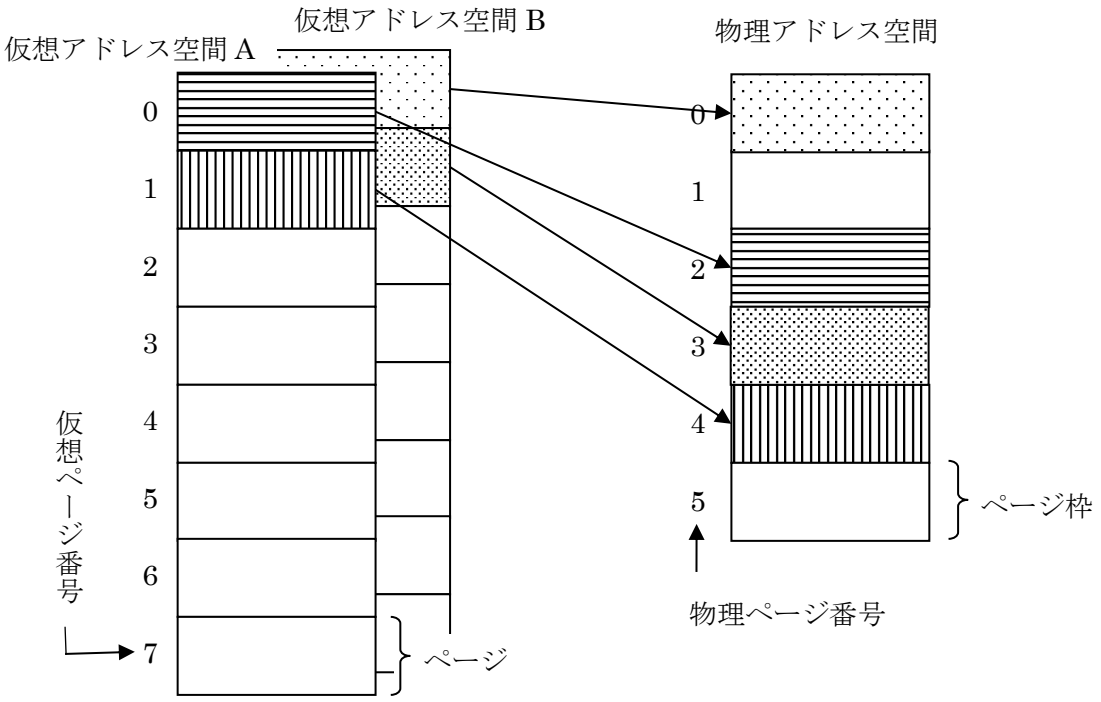


図 9.1 ページングによる仮想アドレス空間

アドレス変換

仮想アドレスから物理アドレスへの変換は概念的には図 9.2 のとおりである。

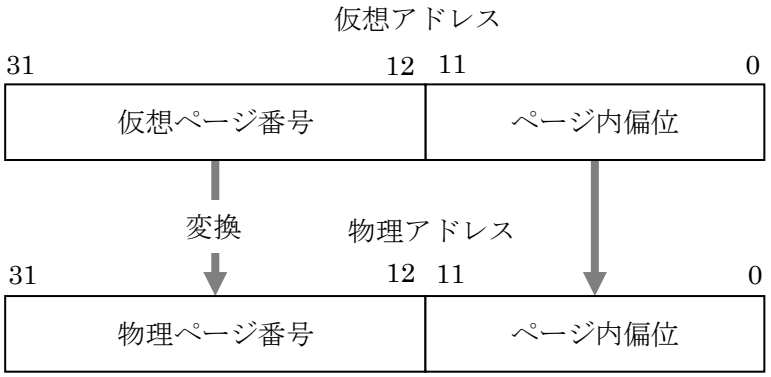


図 9.2 アドレス変換

例）図 9.2 の例は、仮想アドレス　32 bits、　ページサイズ 4KB
　　仮想アドレス空間　4GB、　ページ内偏位　12 bits、　仮想ページ番号　20 bits

アドレス変換は、メインメモリ上（核の領域）のページテーブルにより行われる（図 9.3 参照）。

仮想ページ番号をインデックスとしてページテーブルを参照し、物理ページ番号を得て、これとページ内偏位を結合して物理アドレスを得る。

ページテーブルは、プロセスごとに用意される。

アドレス変換は、memory management unit（MMU）と呼ばれるハードウェアにより自動的に行われる。

但し、メインメモリ上にページテーブルを置くと、プログラム上１回の仮想記憶へのアクセスに対して、実際には２回のメインメモリアクセスが必要となり、プログラムの実行速度が低下するので、一般的に MMU 内にアドレス変換の高速化のための機構を持っている（translation look-aside buffer (TLB)）。

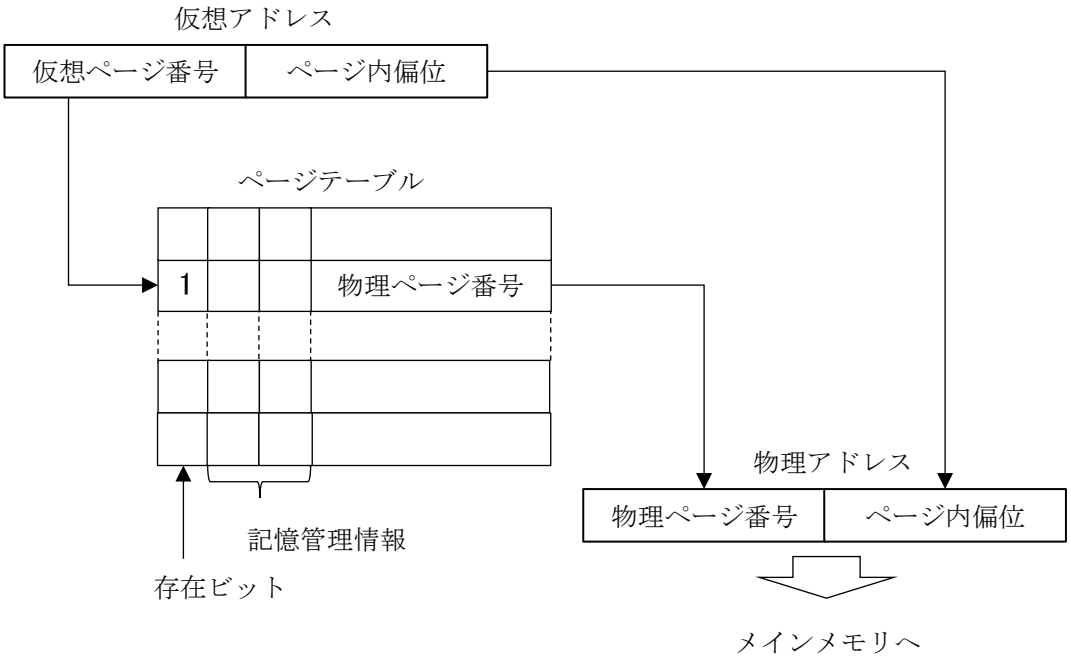


図 9.3 ページテーブルによるアドレス変換

9.3 ページフォルト（page fault）

プロセスが仮想アドレスを用いて仮想記憶にアクセスしようとする際に、MMU だけでは物理アドレスへと変換してのメインメモリアクセスを実現できない事態をページフォルトと呼ぶ。ページフォルトに直面した MMU はページフォルト割り込みを発生させる。

ページフォルトが発生すると、補助記憶装置からのページ読み込みを待つため、プロセスは待ち状態となる。→　プログラムの実行効率が低下する。

手順

1. 必要とするページの仮想ページ番号をインデックスとして参照するページテーブルの存在ビットが 0 なら（図 9.3 参照）MMU によりページフォルト割り込みが発生（以下 OS 内）
2. 空きページ枠を探し、あれば、ページの読み込み
   空きページ枠へ必要なページを補助記憶装置から読み込む（ページイン）。
   なければ、ページ枠を 1 つ選び、ページの置換え（page replacement）（図 9.4）
   そこからメインメモリ上のページを補助記憶装置へ掃き出す（ページアウト）。
   掃き出されたページの仮想ページ番号をインデックスとしてページテーブルの存在ビットを 0 に
   そこへ必要なページを補助記憶装置から読み込む（ページイン）。
3. 読み込まれたページの仮想ページ番号をインデックスとして、ページテーブルにページ枠の物理ページ番号を登録、存在ビットを 1 にし、プロセスの待ち状態を解除

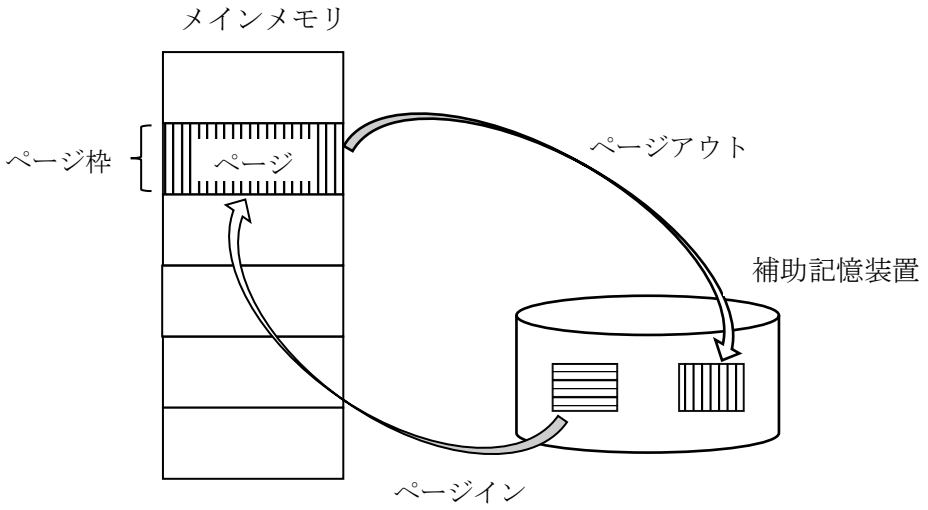


図 9.4 ページ置換え

練習問題 8

1. 仮想記憶の目的を 2 つ述べよ。
2. 仮想記憶において、メインメモリにロードできないプログラムはどこに置かれるか。
3. 仮想アドレス空間と物理アドレス空間の相違を述べよ。
4. 仮想アドレスから物理アドレスへの変換を行う際に使用されるテーブルを何と呼ぶか。

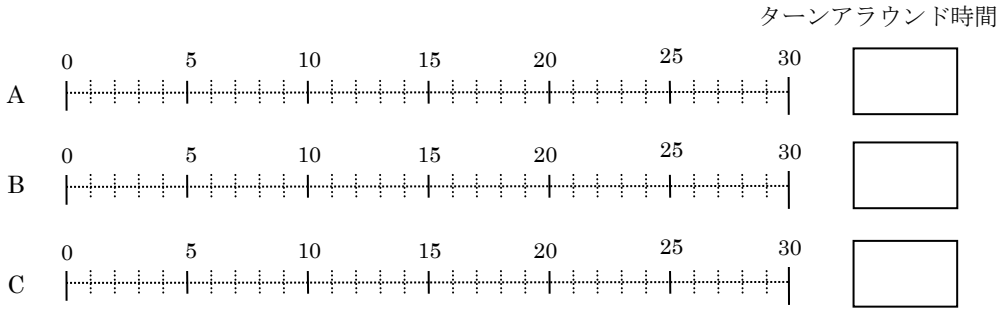
課題 II

問題 I　身近にある UNIX 系の OS について、以下の問題に答えなさい。

1. 用いた OS の名前を答えなさい（バージョン番号は不要）。
2. コマンド“`ls -l`”を実行すると、カレントディレクトリ内の各ファイルに対してどのような情報が得られるか。表示される項目（意味）を 6 個以上列挙しなさい。
3. コマンド“`ps axu`”（もしオプション“`u`”が受け付けられない時は“`ps axl`”）を実行すると各プロセスに対してどのような情報が得られるか。表示される項目（意味）を 7 個以上列挙しなさい。

問題 II　定時間を 2 としたラウンドロビンによるプロセスのスケジューリングを行う。下の表に示す時刻に 3 つのプロセスがシステムに到着し、表に示す処理期間を CPU で消費する。下図のタイミングチャートに各プロセスのシステムへ到着時刻（△）、CPU を割り当てられている時間（太線）、終了時刻（▲）を記入し、各プロセスのターンアラウンド時間を求めよ（A, B, C の ready queue 内での順番に注意すること）。但し、この間新たなプロセスの到着や入出力はなく、プロセス切り替えの時間は無視するものとする。

プロセス	到着時刻	処理時間
A	0	12
B	1	8
C	7	4



問題 III　仮想アドレスが 32 ビット、仮想ページ番号が 16 ビット、ページ内偏位が 16 ビットの仮想記憶において、あるプロセスのある時点におけるページテーブルが以下のようになっているとする。以下の問に答えなさい。

仮想ページ番号	存在ビット	物理ページ番号
0x0000		
・・・		
0x03a8	1	0x246e
0x03a9	0	
0x03aa	1	0x28f2
・・・		

(1) この仮想記憶におけるページサイズは、何 KB か。

(2) このとき、プログラムカウンタ（1.3.2 項参照）の値が 16 進数で `0x03a807a0` であったとする。実際にフェッチされる命令コードは、メインメモリの何番地に存在するか、32 ビットの物理アドレスを 16 進数で答えなさい。

提出

講義開始時の受講案内、及び Moodle 上の指示に従って提出すること。

練習問題解答（第２部）

練習問題４（第５講）

1.

i)（5.2.1 項参照）

（例文）ソフトウェア開発、文書作成、ウェブブラウジング、銀行 ATM からの預金・引き出し・送金の操作等、利用者がコマンド入力やマウスクリックにより要求を出し、応答を見て、次の要求を出す利用形態

ii)（5.2.2 項参照）

（例文）月末締めの給与計算、毎月の公共料金の請求事務、銀行口座における毎月決まった日付に行われる給与の振り込み、公共料金の引き落としなどの定型的な事務処理や天気予報作成などのスーパーコンピュータでの大規模数値計算

iii)（5.2.3 項参照）

（例文）自動車の燃料噴射制御やロボットの障害回避など、応答性に厳しい制約が課せられている組み込み機器等

2.（5.5.1 項参照）

i)（例文）全ての関数から参照される大域変数や `static` 変数

ii)（例文）実行時に大きさを指定して `malloc()` 関数により確保され、`free()` 関数により解放される、動的なメモリ領域

iii)（例文）関数への引数や関数内の一時的な局所変数

3.（5.4.2 項参照）

（例文）・システムを利用しようとしている利用者が正当な利用者であることを認証するため

・認証され、識別された利用者に対する適切な権限の付与や、サービスの提供を行うため

・ログイン・ログアウトの間をセッションとして、利用者の情報や利用情報を保持するため

4.（5.5.2 項参照）

（例文）計算資源への直接アクセスを OS のみに許可し、利用者のプログラムからはシステムコールにより OS を介して資源アクセスすることにより、仮想化・抽象化されたハードウェアへのアクセスと、資源の保護を実現するため

練習問題5（第6講）

1.（6.1節参照）

（例）ハードディスク（または、ディスク装置、HDD）

SSDやUSBメモリ、SDカードなどのフラッシュメモリ

CD、DVDなどの光ディスク等

2.（6.2.3節参照）

i) `/home/ueda/ex1/foo.c`

ii) `./ex1/foo.c` あるいは `ex1/foo.c`

iii) `../ueda/ex1/foo.c` あるいは `./../ueda/ex1/foo.c`

3.（6.3.2節参照）

1分間　7,200回転は、毎秒120回転。

半回転に必要な時間は、1÷120÷2 ＝ 0.00416…

従って、平均回転待ち時間は、4.2ms（ミリ秒）

4.（6.4.6節参照）

（例文）ファイルシステムからファイルを消去することは、ファイルを管理・検索する情報をファイルシステムから消去し、ファイルを構成していたブロックを空きブロックとして解放するだけであるので、ファイルのデータそのものは上書きされるまで空きブロック内に残っており、これを元にファイルのデータを復元することは可能である。

練習問題6（第7講）

1.（7.1.1節参照）

i)（例文）外部割り込みは、命令の実行とは独立して生起する事象により発生する。
内部割り込みは、命令の実行の結果生起する事象により発生する。

ii)（例文）キーボードの押下、マウスのクリック、ハードディスクに対する入出力の完了、ネットワークに対するパケットの送信・受信完了などの入出力の完了
ハードディスクに対する読み書きのエラー、ネットワークに対するパケットの送信・受信の失敗などの入出力の誤動作
タイマの時間切れ、CPU誤動作、電源異常、計算機の再起動操作など

iii)（例文）演算のオーバーフロー、ゼロ除算、メモアクセス保護違反、特権命令違反などのプログラム実行時のエラーや、システムコールを発行するときの割り込みなど

2. （7.2.1 節参照）

（例文）・比較的小さなプログラムの入出力単位と、大きなハードディスクなどの入出力単位を調整し、時間の掛かる入出力の回数を減少させるため

・比較的速いメモリのアクセス速度と、遅いハードディスクなどのアクセス速度を調整することにより、待ち合わせ時間を解消するため

・データを共有する場合や再利用する場合に、キャッシュとして効果が期待できる。

練習問題 7（第 8 講）

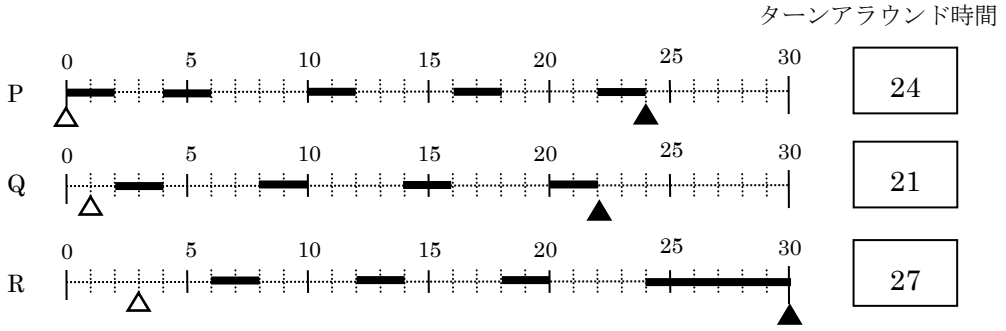
1.（8.3 項参照）

（例文）実行可能状態と待ち状態は、ともに CPU を割り当てられていない状態であるが、実行可能状態は CPU を必要としているが現在別のプロセスに CPU が割り当てられているため、CPU が割り当てらさえすれは実行可能な状態であり、待ち状態は、CPU を必要としていない状態

2.（8.3 項参照）

（例文）要求していたハードディスクからの読み込み完了、待っていたキーボードの押下やマウスのクリック、ネットワークからのパケット到着等の入出力完了、何らかの応答待ちの際のタイムアウト、シェルにおけるコマンドを実行する子プロセスの終了など

3. （8.4.2 項 b 参照）



※　R の到着直後の実行可能待ち行列内のプロセスは、Q, P, R の順につながれている。

練習問題 8（第 9 講）

1. （9.1.1 項参照）
   （例文）プロセスにメインメモリの大きさにとらわれないより容量の大きい仮想的な記憶装置を提供することと、それ専用の一連の仮想的なアドレスが振られたアドレス空間を提供すること
2. （9.1.2 項参照）　補助記憶装置
3. （9.1.2 項）
   （例文）仮想アドレス空間は、実行前にプログラムに割り振られたアドレス空間で、物理アドレス空間は、実行時にプログラムがロードされた際のメインメモリのアドレス空間
4. （9.2 節参照）　ページテーブル

第３部

計算機ハードウェア

第 10 講　計算機ハードウェアの全体構成

本講では、「計算機システム I」の復習を兼ねて、一般的な計算機ハードウェアの全体構成を、以下の観点から理解することを目的とする。

・一般的な CPU の基本構成と基本動作
・メインメモリの構成と CPU との関係
・入出力装置も含めた計算機の全体構成

10.1 CPU (Central Processing Unit)

10.1.1 CPU の基本的な構成

ある種の CPU の基本的な構成を図 10.1 に示す。

・プログラムカウンタ
　次に実行すべき命令のアドレスを保持するレジスタ。命令フェッチの度に自動的に増加する。
・命令レジスタ
　フェッチされた命令のコードを一時的に保持するレジスタ
・命令デコーダ・制御回路
　命令レジスタ内の命令のコードを解読し、制御回路により他の回路を制御する。
　（例：加算命令を解読し、ALU に対して加算の実行を指示する等）
・汎用レジスタ
　メモリから読み出したデータや演算結果、プログラム内で参照されるメモリのアドレスなどを一時的に保持するなど、汎用的に用いられるレジスタ
・ALU (Arithmetic Logic Unit)
　算術演算や論理演算を行う論理回路

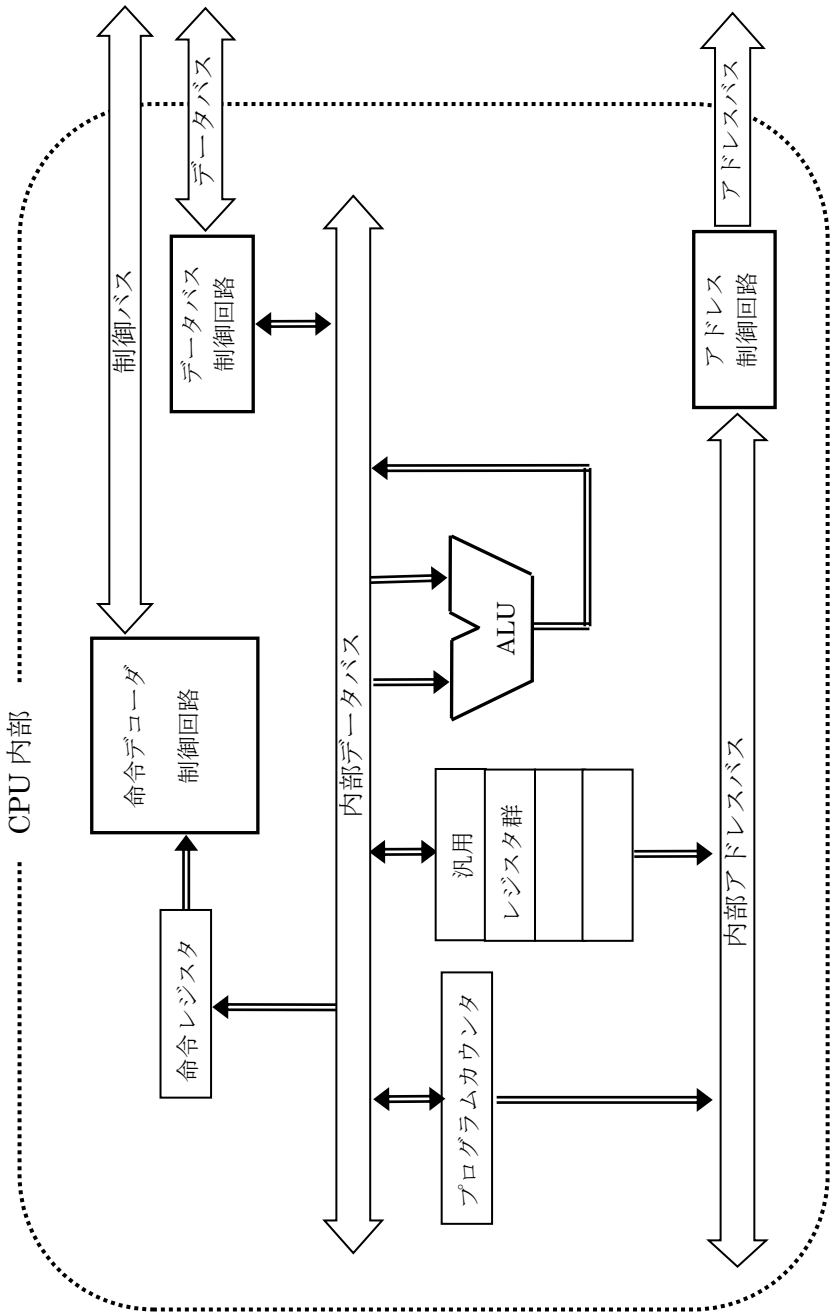


図 10.1 CPU の基本構成

10.1.2 CPU の基本的な動作

CPU は図 10.2 に示す命令実行サイクルを、ただひたすらに繰り返す。

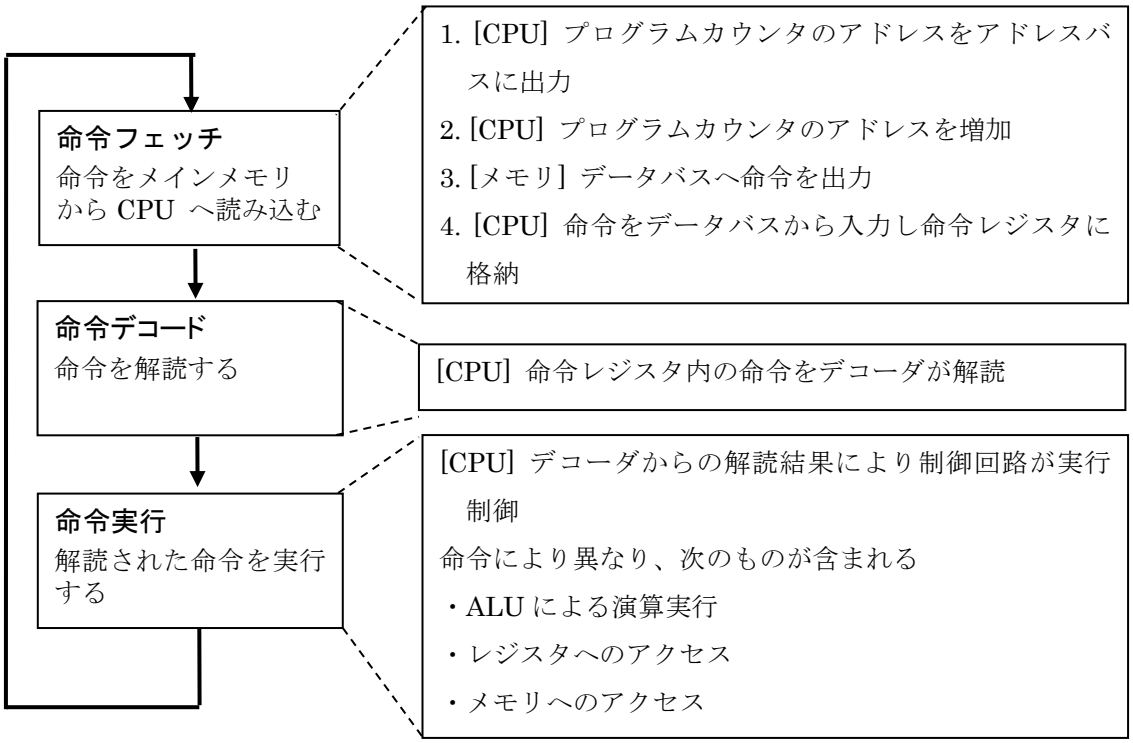


図 10.2 命令実行サイクル

10.2 メインメモリ

メインメモリは、(最後に格納された) 命令やデータを保持 (記憶) する。

図 10.3 左のようにメインメモリには 1Byte 単位でアドレス (番地) が振られる。32-bit アドレスの場合、0x00000000 番地 から 0xffffffff 番地まで 4GB

メインメモリには、実行したいプログラム (を構成する命令や初期データ) をロードできる。一部に ROM (read-only memory) を用いた場合、その部分のプログラムは (簡単には) 変更できないものの、電源オフでもプログラムを保持し続けられる。

データには、入力データ、演算の途中結果、出力待ちのデータ等がある。

メインメモリは図 10.3 右のように CPU とバスを介して接続されている。

CPU のメインメモリからのデータの読み出しの手順 (命令フェッチと同様)

①CPU がアドレスをアドレスバスに出す。

②CPU が読み出し要求を制御バスに出す。

③メモリがデータをデータバスに出す。

④CPU がデータバスからデータを取り込む。

CPU のメインメモリへのデータの書き込みの手順

①CPU がアドレスをアドレスバスに出す。

②CPU がデータをデータバスに出す。

③CPU が書き込み要求を制御バスに出す。

④メモリがデータバスのデータを該当するアドレスに書き込む。

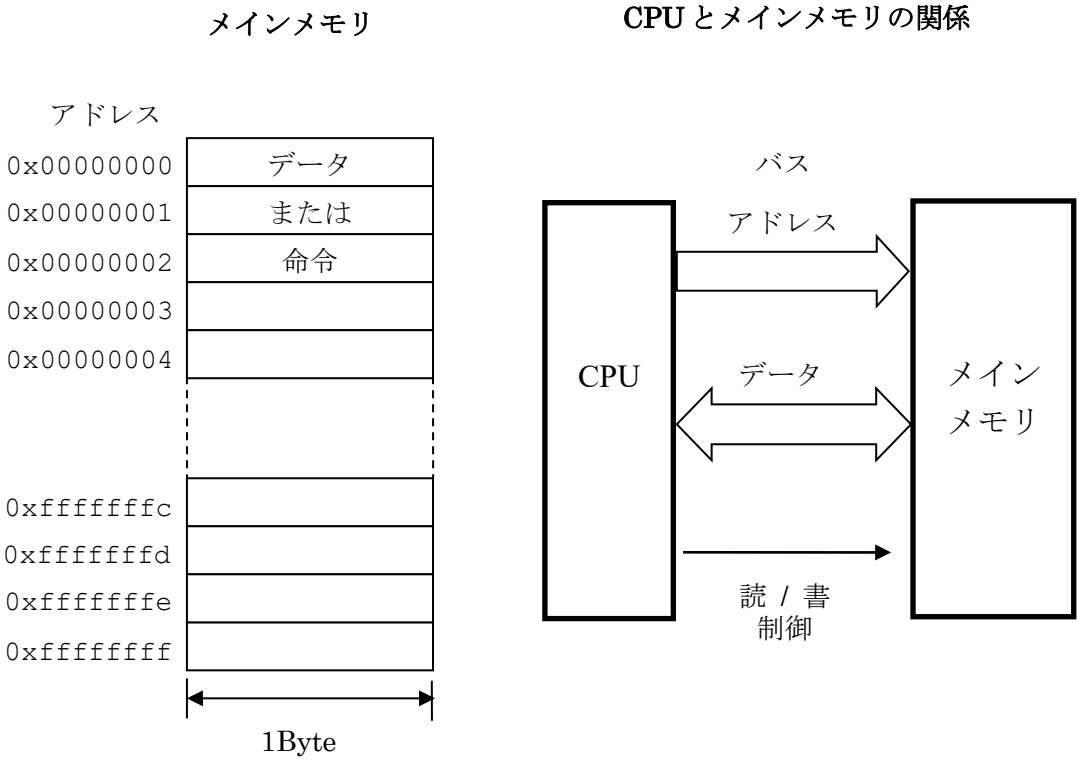


図 10.3 メインメモリと CPU の関係

10.3 計算機の全体構成とバス

汎用大型機やスーパーコンピュータ以外の計算機は、基本的に図 10.4 に示す通り、概念的にバスに全ての構成要素が接続され、バスを介してデータのやり取りが行われる。

DMA 転送

CPU が直接入出力に介在し、CPU 経由でメインメモリと入出力装置の間でのデータ転送を行うと時間が掛かる。

そこで、DMA (Direct Memory Access : 直接メモリアクセス) コントローラが入出力装置とメインメモリの間のデータ転送を制御することで、入出力の効率を高める。

ハードディスクを入出力装置の例とした DMA 転送の手順

① CPU → DMA コントローラ

ディスク上のアドレス、メインメモリ内のバッファのアドレス、転送バイト数を設定

② CPU → ディスクコントローラ
入出力開始要求
③ DMA コントローラ ←→ ディスクコントローラ
必要な情報（ディスク上のデータの位置、メインメモリ内のバッファのアドレス、データサイズ）のやり取り
④ DMA 転送 ハードディスク ←→ メインメモリ内のバッファ
実際のデータの転送
⑤ 転送終了後：DMA コントローラ → ディスクコントローラ
ディスク入出力の終了を指示
⑥ ディスクコントローラ → CPU
入出力完了割り込み

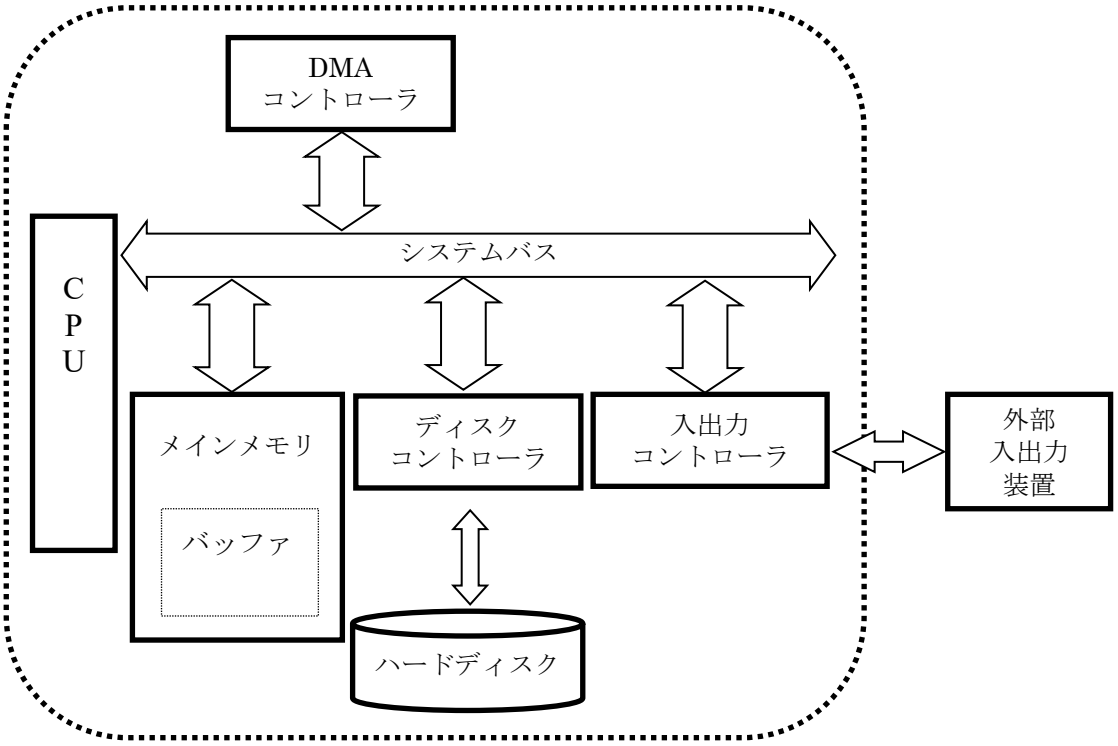


図 10.4 計算機の概念的な全体構成

練習問題 9

1. 次の略語のフルスペリングを答えなさい。
   i) CPU
   ii) ALU
   iii) DMA
2. 次の CPU 内のレジスタの機能を説明しなさい。
   i) プログラムカウンタ
   ii) 命令レジスタ
   iii) 汎用レジスタ

第 11 講　命令セットアーキテクチャと CPU の実行過程

本講以降では、「計算機システム I」の内容を基礎として、具体的な命令セットアーキテクチャの実例として MIPS を取り上げ、基本的な CPU の構成と命令の実行過程を、以下の観点から理解することを目的とする。

・命令セットアーキテクチャの実例としての MIPS アーキテクチャ

・CPU の実行過程の実例

11.1 MIPS アーキテクチャ

11.1.1　MIPS について

これ以降、MIPS と呼ばれる命令セットアーキテクチャを具体的な題材として、命令セットアーキテクチャの概念を学ぶ。

命令セットアーキテクチャとは、ソフトウェア技術者から見た CPU の基本的な構成のことで、どのような命令の種類があるか、機械語の形式がどのようになっているか、どのように対象となるデータの所在場所を指し示すか（レジスタの構成も含む）が含まれる。

従って、実装の方法はアーキテクチャに含まれない。

MIPS の CPU の商用実装としては、R2000, R3000, R4000 … 等がある。これらは、日本のメーカでは、日本電気（NEC）、東芝、ソニー、任天堂などの機器に用いられている。

32 ビット版と、64 ビット版があるが、本講義では、簡単化のため 32 ビット版を取り上げる。

これは、メモリアクセスや演算の単位が 32 ビットであることを表す。

ここで、32 ビット を 1 word（語）と呼ぶ。即ち、1 word = 4 Bytes.

11.1.2　レジスタ構成

表 11.1 に示す通り、MIPS は 32-bit レジスタを 32 本有する（広義の汎用レジスタ）。

演算対象となるデータはレジスタに格納する。

表 11.1 のレジスタのうち、特殊な用途に用いられる `$zero`, `$ra` 以外はハードウェア的な役割は固定されていない本当の汎用レジスタであるが、ソフトウェア的な使用法は規約に定められている。

それぞれの利用法は、命令の紹介の際に述べる。

表 11.1 MIPS のレジスタ構成

名称	番号（十進）	用途
$zero	0	常に 0
$at	1	アセンブラが一時的に使用
$v0 ～ $v1	2 ～ 3	サブルーチンの戻り値
$a0 ～ $a3	4 ～ 7	サブルーチンの引数
$t0 ～ $t7	8 ～ 15	一時変数（セーブされることを想定しない）
$s0 ～ $s7	16 ～ 23	一時変数（セーブされることを想定）
$t8 ～ $t9	24 ～ 25	一時変数（セーブされることを想定しない）
$k0 ～ $k1	26 ～ 27	カーネル用に予約
$gp	28	広域ポインタ
$sp	29	スタックポインタ
$fp	30	フレームポインタ
$ra	31	サブルーチンのリターンアドレス

11.1.3 命令の形式

MIPS の機械語の命令は、全て 32-bit 固定長である。
命令の形式として、機械語におけるオペランドの種類に応じて、以下の 3 つの形式がある。

・R 形式

表 11.1 のレジスタ（register）を指定するオペランドを 3 つ持つ。
レジスタとレジスタの値を演算し、結果をレジスタに格納する。

31 0

opcode	rs	rt	rd	シフト量	機能
(6)	(5)	(5)	(5)	(5)	(6)

opcode: オペコード　命令の種類を表す
rs: 第 1 ソースレジスタ（読み出し元）
rt: 第 2 ソースレジスタ
rd: デスティネーションレジスタ（格納先）
シフト量: R 形式ではシフト命令で用いる定数を指定するオペランドも 1 つ持てる
機能: オペコードのバリエーション（R 形式の場合の実質的なオペコード）

例）
ニーモニック `add $t0, $s0, $s1`

動作　　レジスタ$s0 と $s1 の値を加算し、結果をレジスタ $t0 に格納する。
（$t0 ← $s0 + $s1）
機械語コード 000000 10000 10001 01000 00000 100000
add $s0 $s1 $t0 add

・I 形式

表 11.1 のレジスタを指定するオペランドを 2 つ、16 ビットの即値 (immediate value) として値そのものを指定するオペランドを 1 つ持つ。

31 0

opcode	rs	rt	即値
(6)	(5)	(5)	(16)

opcode: オペコード　命令の種類を表す
rs: ソースレジスタ（転送命令の場合は、ベースレジスタ）
rt: ターゲットレジスタ
即値: 即値定数（転送命令の場合は、偏位）

次講以降でニーモニックの構文を示す際は、ソースレジスタとして$*rs*、ベースレジスタとして$*rb* を用いる。また，機械語でのターゲットレジスタについては、実際の動作における役割に応じて、ソースレジスタとして$*rs*、第 2 ソースレジスタとして$*rt*、デスティネーションレジスタとして$*rd* を用いる。また、即値には *C* を用いる。

例）
ニーモニック addi $t0, $s0, 5
動作　　レジスタ$s0 の値と定数 5 を加算し、結果をレジスタ $t0 に格納する。
（$t0 ← $s0 + 5）
機械語コード 001000 10000 01000 0000000000000101
addi $s0 $t0 即値 5

・J 形式

26-bit のアドレスを指定する単一のオペランドを持つ。

31 0

opcode	アドレス
(6)	(26)

例）

ニーモニック `j` *address*
動作　　*address* から求められる番地に無条件ジャンプする（詳細は、14.3 節）。
機械語コード `000010` **********

`j`　　*address* の値そのもの

11.2 CPU の実行過程

加算命令の例

`$s0` と`$s1` の値を加算し、`$t0` に格納する。

ニーモニック　`add $t0, $s0, $s1`

コード　`000000` `10000` `10001` `01000` `00000` `100000`

`add`　`$s0`　`$s1`　`$t0`　`add`

・命令フェッチ（IF）

・命令デコード（ID）

この場合、オペコードが `000000` であるので、R 形式であることが分かる。

デコードの結果、以下のことが判明する。

機能フィールドから命令の種類は加算命令（`add`）（従って、シフト量は不使用）

第 1 ソースレジスタは`$s0`

第 2 ソースレジスタは`$s1`

デスティネーションレジスタは`$t0`

・命令実行（EX）

レジスタ`$s0` と`$s1` の値を ALU で加算する。

・書き込み（WB）

ALU の出力をレジスタ`$t0` に格納する。

この場合の実行過程を、図 11.1 に示す。

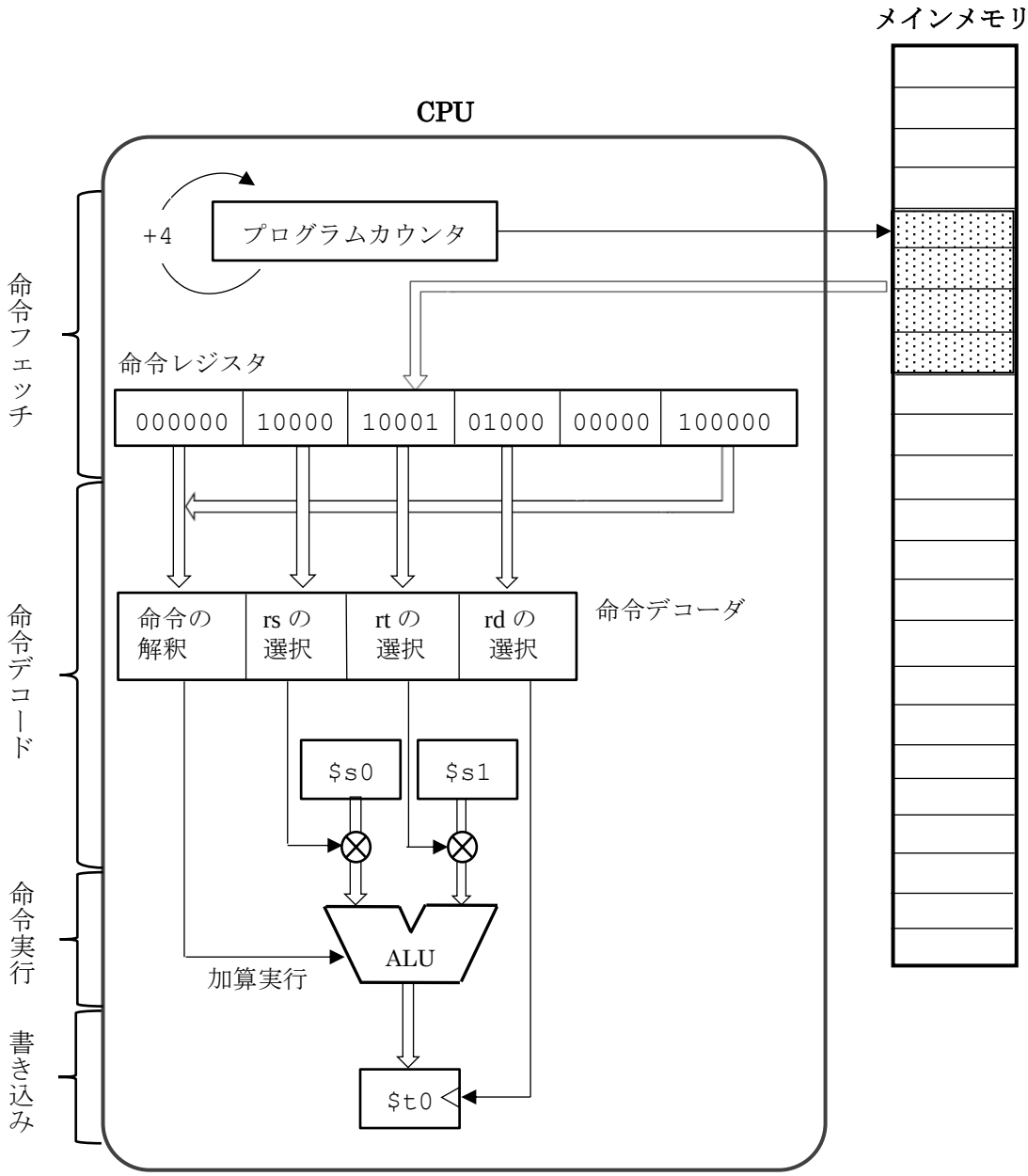


図 11.1 加算命令の実行過程

ロード命令の例

指定されたアドレスのメモリからデータを読み出し、$s1 に格納する。

ニーモニック　lw $s1, 64($s0)

コード　　100011　10000　10001　0000000001000000

　　lw　　$s0　　$s1　　64

・命令フェッチ（IF）

・命令デコード（ID）

デコードの結果、以下のことが判明する。

オペコードからⅠ形式であり、命令の種類はロード命令（lw）

転送命令だから、ベースレジスタは$s0

転送命令だから、偏位は即値の 64（詳細は、13.1.1 項および 13.2 節参照）

Ⅰ形式のターゲットレジスタ（ロード命令ではデスティネーションレジスタ）は$s1

・命令実行（EX）

メモリアドレスを計算する。この場合、$s0+64

・メモリアクセス（MEM）

上記のアドレスのメモリから 4 バイトを読み出す。

・書き込み（WB）

メモリから読み出された値をレジスタ$s1 に格納する。

ストア命令の例

$s1 の値を指定されたアドレスのメモリに書き込む。

ニーモニック　sw $s1, 64($s0)

コード　　101011　10000　10001　0000000001000000

　　sw　　$s0　　$s1　　64

・命令フェッチ（IF）

・命令デコード（ID）

デコードの結果、以下のことが判明する。

オペコードからⅠ形式であり、命令の種類はストア命令（sw）

転送命令だから、ベースレジスタは$s0

転送命令だから、偏位は即値の 64

Ⅰ形式のターゲットレジスタ（ストア命令ではソースレジスタ）は$s1

・命令実行（EX）

メモリアドレスを計算する。この場合、$s0+64

・メモリアクセス（MEM）

上記のアドレスのメモリへ、$s1 の 4 バイトの値を書き込む。

練習問題 10

1. MIPS は、機械語プログラムの命令のオペランドとして指定可能な 32-bit レジスタを何本持つか。また、そのうち 2 本は特殊な用途に用いられるが、その用途を述べなさい。
2. MIPS の 3 つの命令形式について、それぞれどのようなオペランドを持つか。
3. MIPS アーキテクチャを採用した CPU におけるロード命令

```
lw     $s0, -4($fp)
```

の実行過程を 5 ステップで述べなさい。

第 12 講　算術・論理演算命令

本講以降では、具体的な命令セットの実例として MIPS を挙げることにより、命令セットアーキテクチャに対する理解を深めることを目的とする。そのため、MIPS の全ての命令を紹介する訳ではない。本講では、命令セットの理解を深めることを目的として、以下の命令を取り上げる。

・整数の算術演算命令
・論理演算命令
・シフト命令

12.1 算術演算命令（整数）

・add

ニーモニック　`add $rd, $rs, $rt`

動作　レジスタ `$rs` と `$rt` に格納されている値を読み出し、2 の補数表現の 32-bit 整数として加算し、結果をレジスタ `$rd` に格納する。
（`$rd` ← `$rs` + `$rt` ）

例）`add $t0, $s0, $s1`

コード　R 形式

000000	10000	10001	01000	00000	100000
add	$s0	$s1	$t0		add

・add immediate

ニーモニック　`addi $rd, $rs, C`

動作　レジスタ `$rs` の値に定数 *C* を加算し、結果をレジスタ `$rd` に格納する。即値 *C* は、2 の補数表現の 32-bit 整数に符号拡張される。
（`$rd` ← `$rs` + *C* ）

例）`addi $t0, $s0, 5`

コード　I 形式

001000	10000	01000	0000000000000101
addi	$s0	$t0	即値 5

※符号拡張

2 の補数表現のデータを、より長いビット長の 2 の補数表現のデータに拡張する場合、値の同一性を保つため、拡張部分となる上位ビットをデータの符号ビットのコピーで埋めること

例）16 ビット長のデータから 32 ビット長のデータに符号拡張する場合

データが十進表現で +100 のとき

```
                    0000000001100100
                   ↓
0000000000000000 0000000001100100
```

（上位 16 ビットを 符号ビットの 0 で埋める）

データが十進表現で －100 のとき

```
                    1111111110011100
                   ↓
1111111111111111 1111111110011100
```

（上位 16 ビットを 符号ビットの 1 で埋める）

・subtract

ニーモニック　sub $*rd*, $*rs*, $*rt*

動作　　レジスタ $*rs* から $*rt* の値を減算し、結果をレジスタ $*rd* に格納する。

（$*rd* ← $*rs* - $*rt* ）

例）sub $t0, $s0, $s1

コード　R 形式

000000	10000	10001	01000	00000	100010
sub	$s0	$s1	$t0		sub

・multiply

ニーモニック　mult $*rs*, $*rt*

動作　　レジスタ $*rs* と $*rt* の値を符号付整数と解釈して乗算し、結果の 64 ビットを専用レジスタ HI（上位）と LO（下位）に格納する。

（[HI:LO] ← $*rs* * $*rt* ）

専用レジスタ HI, LO から広義の汎用レジスタへの転送には、mfhi (move from high), mflo (move from low) を用いる。※mfhi, mflo 命令から 2 命令以内に乗除算命令を実行してはならない。

例）mult $s0, $s1

コード　R 形式

000000	10000	10001	00000	00000	011000
mult	$s0	$s1			mult

・move from high

ニーモニック　mfhi $*rd*

動作　　レジスタ HI の値をレジスタ $*rd* に格納する。

（$*rd* ← HI ）

例）mfhi $t1

コード　R 形式　000000　00000　00000　01001　00000　010000

mfhi　$t1　mfhi

・move from low

ニーモニック　mflo $*rd*

動作　レジスタ LO の値をレジスタ $*rd* に格納する。

（$*rd* ← LO ）

例）mflo $t0

コード　R 形式　000000　00000　00000　01000　00000　010010

mflo　$t0　mflo

・divide

ニーモニック　div $*rs*, $*rt*

動作　レジスタ $*rs* の値を $*rt* の値で符号付整数と解釈して除算し、商を LO、剰余を HI に格納する。

（LO ← $*rs* / $*rt*, HI ← $*rs* % $*rt* ）

例）div $s0, $s1

コード　R 形式　000000　10000　10001　00000　00000　011010

div　$s0　$s1　div

12.2 論理演算命令

・and

ニーモニック　and $*rd*, $*rs*, $*rt*

動作　レジスタ $*rs* と $*rt* の値のビットごとの論理積をレジスタ $*rd* に格納する。

（$*rd* ← $*rs* & $*rt* ）

例）and $t0, $s0, $s1

コード　R 形式　000000　10000　10001　01000　00000　100100

and　$s0　$s1　$t0　and

・and immediate

ニーモニック　andi $*rd*, $*rs*, *C*

動作　レジスタ $*rs* の値と定数 *C* のビットごとの論理積をレジスタ $*rd* に格納する。即値 *C* は、2 の補数表現の 32-bit 整数に符号拡張される。

（$*rd* ← $*rs* & *C* ）

例）andi $t0, $s0, 15

コード　I形式　001100　10000　01000　0000000000001111

andi　$s0　$t0　即値 15

この例では、$s0 の上位 28 ビットを 0 にした（下位 4 ビットはそのままの）値を $t0 に残すことになる。このことを $s0 の上位 28 ビットを「マスクする」（見えなくする）という。

・or

ニーモニック　or $*rd*, $*rs*, $*rt*

動作　レジスタ $*rs* と $*rt* の値のビットごとの論理和をレジスタ $*rd* に格納する。

（$*rd* ← $*rs* | $*rt* ）

例）or $t0, $s0, $s1

コード　R形式　000000　10000　10001　01000　00000　100101

or　$s0　$s1　$t0　or

・or immediate

ニーモニック　ori $*rd*, $*rs*, *C*

動作　レジスタ $*rs* の値と定数 *C* のビットごとの論理和をレジスタ $*rd* に格納する。即値 *C* は、2 の補数表現の 32-bit 整数に符号拡張される。

（$*rd* ← $*rs* | *C* ）

例）ori $t0, $s0, 1

コード　I形式　001101　10000　01000　0000000000000001

ori　$s0　$t0　即値 1

この例では、$s0 の最下位ビットを 1 にした（それ以外のビットはそのままの）値を $t0 に残すことになる。このことを $s0 の最下位ビットに「1 を立てる」という。

・exclusive or

ニーモニック　xor $*rd*, $*rs*, $*rt*

動作　レジスタ $*rs* と $*rt* の値のビットごとの排他的論理和をレジスタ $*rd* に格納する。

（$*rd* ← $*rs* ^ $*rt* ）

例）xor $t0, $s0, $s1

コード　R形式　000000　10000　10001　01000　00000　100110

xor　$s0　$s1　$t0　xor

・nor

ニーモニック　nor $*rd*, $*rs*, $*rt*

動作　　レジスタ $*rs* と $*rt* の値のビットごとの NOR をレジスタ $*rd* に格納する。

（$*rd* ← ~($*rs* | $*rt*) ）

例）nor $t0, $s0, $zero

コード　R 形式　000000　10000　00000　01000　00000　100111

nor　$s0　$zero　$t0　nor

MIPS においては、NOT 命令（否定 : 全ビット 0/1 を反転）は用意されていない。上述のように、0 との NOR を取ることにより NOT 命令の代用とする。

12.3 シフト命令

・shift left logical

ニーモニック　sll $*rd*, $*rt*, *C*

動作　　レジスタ $*rt* を左に *C* ビットシフトし、結果をレジスタ $*rd* に格納する。左シフトして空いた下位ビットには 0 を格納する。0 ≦ *C* < 32

（$*rd* ← $*rt* << *C* ）

例）sll $t0, $s0, 2

コード　R 形式　000000　00000　10000　01000　00010　000000

sll　$s0　$t0　2　sll

上記の例の 2-bit 左シフトは、4 を掛けることと同等である。

・shift right logical

ニーモニック　srl $*rd*, $*rt*, *C*

動作　　レジスタ $*rt* を右に *C* ビットシフトし、結果をレジスタ $*rd* に格納する。右シフトして空いた上位ビットには 0 を格納する。

（$*rd* ← $*rt* >> *C* ）

例）srl $t0, $s0, 2

コード　R 形式　000000　00000　10000　01000　00010　000010

srl　$s0　$t0　2　srl

上記の例の 2-bit 論理的右シフトは、符号なし整数として 4 で割ることと同等である。

・shift right arithmetic

ニーモニック　sra $*rd*, $*rt*, *C*

動作　　レジスタ $*rt* を右に *C* ビットシフトし、結果をレジスタ $*rd* に格納する。右シフトして空いた上位ビットには $*rt* の値の符号拡張を行う。

（$*rd* ← $*rt* >> *C*　　符号拡張を行う）

例）`sra $t0, $s0, 2`

コード　R 形式　`000000`　`00000`　`10000`　`01000`　`00010`　`000011`

`sra`　`$s0`　`$t0`　`2`　`sra`

練習問題 11

1. MIPS の次のアセンブリ言語の命令を機械語のコードに変換し、二進数で表しなさい。
   i)　`add  $t1, $s1, $a0`
   ii)　`addi  $t0, $s2, -2`
2. 上の問 ii) の命令実行前のレジスタ`$s2` の値が十六進で `0x00000007` のとき、実行後のレジスタ`$t0` の値を十進数で答えなさい。
3. レジスタ`$s0` の値からレジスタ`$s1` の値を引き、結果をレジスタ`$t0` に格納する MIPS の命令をアセンブリ言語で書きなさい。
4. レジスタ`$s0` の値を `1` 増加させる（即ち、`$s0` の値に `1` を加え、結果を`$s0` に残す）MIPS の命令をアセンブリ言語で書きなさい。
5. レジスタ`$s1` の値の最下位ビットのみを残し、上位 31 ビットを 0 にしてレジスタ`$t1` に格納する（即ち、最下位ビットを残して、上位 31 ビットにマスクを掛ける）MIPS の命令をアセンブリ言語で書きなさい（1 命令で)。
6. レジスタ`$s0` の値の 2 倍をレジスタ`$s1` に格納する MIPS の命令をアセンブリ言語で書きなさい（乗算命令を用いずに 1 命令で)。（ヒント：　$a + a = 2a$ である。また、1 ビット左シフトも 2 倍になる。）

第 13 講　データ転送命令とアドレシングモード

本講では、前講に引き続き、命令セットの理解を深めることを目的として、データ転送命令を取り上げ、データの所在場所を示すアドレシングモードと実効アドレスを解説する。また、プログラム例を通して、C 言語などのプログラミング言語と、アセンブリ言語の関係を理解することを目的とする。

13.1 データ転送命令

一般的に、レジスタはアクセス対象として非常に高速であるが、その本数はメインメモリの容量と比較して非常に少ない。

そこで、必要なデータの大部分はメインメモリに置かれる。(ユーザプログラムにおいては仮想記憶も活用)

しかし、演算はレジスタを対象に行われる（特に、MIPS では、演算の対象となるのは全てレジスタ)。

そこで、メインメモリのデータをレジスタに転送し、演算終了後にメインメモリに格納する。

ここで、「転送」とは、記憶素子間でのデータのコピーである。

13.1.1　メインメモリからレジスタへの転送命令

・load word

ニーモニック　`lw $`*rd*`, `*C*`(`$*rb*`)`

動作　連続する 4 バイト（word）のデータをメモリからロードし、32-bit の値としてレジスタ `$`*rd* に格納する。

メモリの先頭アドレス（ユーザプログラムでは仮想アドレス）は、ベースレジスタ `$`*rb* の値に偏位 *C* を加えた値。偏位 *C* は、2 の補数表現の 32-bit 整数に符号拡張される。

（`$`*rd* ← memory`[$`*rb* + *C*`]`）

例）`lw $s1, 4($s0)`

コード　I形式　`100011 10000 10001 0000000000000100`

`lw $s0 $s1 4`

例えば、32-bit 整数 2 つからなる C 言語の構造体へのポインタ（メモリアドレス）を `$s0` が保持するとき、2 つ目の整数値をメモリからロードして`$s1` に格納

例）`lw $s0, -4($fp)`は、`$fp` が保持するメモリアドレスを基準（ベース）として偏位`-4` のアドレスを先頭に 4 バイト（`$fp-4` から`$fp-1` まで, 図 13.1 のｂ参照）の

データをメモリからロードし 32-bit 値として$s0 に格納

・load half-word

ニーモニック　lh $*rd*, *C*($*rb*)

動作　連続する 2 バイト（half word）のデータをメモリからロードし、16 ビットから 32 ビットへの符号拡張を行ってレジスタ $*rd* に格納する(即ち、データは符号付の整数であると見做す)。メモリの先頭アドレスは、同上

例）lh $s1, 8($s0)

コード　I形式　100001　10000　10001　0000000000001000

lh　$s0　$s1　8

・load half-word unsigned

ニーモニック　lhu $*rd*, *C*($*rb*)

動作　連続する 2 バイト（half word）のデータをメモリからロードし、16 ビットから 32 ビットへのゼロ拡張を行ってレジスタ $*rd* に格納する。（即ち、データは単なるビット列や符号なし整数であると見做す)。メモリの先頭アドレスは、同上

例）lhu $s1, 0($s0)

コード　I形式　100101　10000　10001　0000000000000000

lhu　$s0　$s1　0

・load byte

ニーモニック　lb $*rd*, *C*($*rb*)

動作　1 バイトのデータをメモリからロードし、8 ビットから 32 ビットへの符号拡張を行ってレジスタ $*rd* に格納する。メモリの先頭アドレスは、同上

例）lb $s1, 0($fp)

コード　I形式　100000　11110　10001　0000000000000000

lb　$fp　$s1　0

・load byte unsigned

ニーモニック　lbu $*rd*, *C*($*rb*)

動作　1 バイトのデータをメモリからロードし、8 ビットから 32 ビットへのゼロ拡張を行ってレジスタ $*rd* に格納する。メモリの先頭アドレスは、同上

例）lbu $s0, -4($fp)

コード　I形式　100100　11110　10000　1111111111111100

lbu　$fp　$s0　-4

13.1.2 レジスタからメインメモリへの転送命令

・store word

ニーモニック　sw $*rs*, *C*($*rb*)

動作　レジスタ $*rs* の値を連続する 4 バイト（word）のデータとしてメモリにストア（格納）する。

メモリの先頭アドレス（ユーザプログラムでは仮想アドレス）は、ベースレジスタ $*rb* の値に偏位 *C* を加えた値。偏位 *C* は、2 の補数表現の 32-bit 整数に符号拡張される。

（$*rs* → memory[$*rb* + *C*]）

例）sw $s1, -8($fp)

コード　I形式　101011　11110　10001　1111111111111000

sw　$fp　$s1　-8

$fp が保持するメモリアドレスを基準（ベース）として偏位-8 のアドレスを先頭とする 4 バイト（$fp-8 から$fp-5 まで, 図 13.1 の y 参照）のメモリに$s1 の値を 4 バイトのデータとしてストア

・store half-word

ニーモニック　sh $*rs*, *C*($*rb*)

動作　レジスタ $*rs* の値の下位 16 ビットを連続する 2 バイトのデータとしてメモリにストア（格納）する。

メモリの先頭アドレスは、同上

例）sh $s1, -12($fp)

コード　I形式　101001　11110　10001　1111111111110100

sh　$fp　$s1　-12

・store byte

ニーモニック　sb $*rs*, *C*($*rb*)

動作　レジスタ $*rs* の値の下位 8 ビット（1 バイト）をメモリに格納

メモリの先頭アドレスは、同上

例）sb $s1, -16($fp)

コード　I形式　101000　11110　10001　1111111111110000

sb　$fp　$s1　-16

補足

MIPS には、レジスタ間の転送命令は用意されていない。これは、

```
addi $rd, $rs, 0
```

で代用する。

例）　`addi $s1, $s0, 0`　　レジスタ`$s1`にレジスタ`$s0`の値をコピーする

これは、他の命令セットアーキテクチャにおける

```
move $rd, $rs
```

に相当する。

同様に、レジスタに小さな定数を即値としてセットする転送命令も用意されていない。これは、

```
addi $rd, $zero, C
```

で代用する。

例）　`addi $s0, $zero, 10`　　レジスタ`$s0`に定数`10`をセットする

これは、他の命令セットアーキテクチャにおける

```
movei $rd, C
```

に相当する。

13.2 アドレシングモードと実効アドレス

オペランドによって命令の対象となるデータの所在場所を指定する方式をアドレシングモード（addressing mode）と呼ぶ。分岐命令の分岐先となる命令の所在場所を指定することもある。

所在場所がメモリの場合、命令実行時に求められる実際のメモリアドレス（ユーザプログラムでは仮想アドレス）を実効アドレス（effective address）と呼ぶ。

いくつかのアドレシングモードでは、その時々のベースレジスタやプログラムカウンタの値を基準として、実効アドレスが求められる。

MIPS には、以下のアドレシングモードがある。

レジスタアドレシング（register addressing）

機械語でもアセンブリ言語でもオペランドにデータの所在場所となるレジスタを指定する。

例）　`lw $s1, 64($s0)`

ベースアドレシング（base addressing）

機械語では 2 つのオペランドによってアセンブリ言語では 1 つのオペランドにベース

レジスタと、偏位（displacement）となる定数を指定する。実効アドレスは、ベースレジスタが保持する値に偏位を加えて求められる。

例） `lw $s1, 64($s0)`

ベースレジスタが`$s0`、偏位が 64 である。実効アドレスは、`$s0+64` である。

即値アドレシング（immediate addressing）

機械語でもアセンブリ言語でもオペランドに定数そのものを指定する。データの所在場所が命令内であると考えることもできる。

例） `addi $t0, $s0, 5`

PC 相対アドレシング（PC-relative addressing）

条件分岐命令の分岐先の命令の所在場所の実効アドレスを、現在のプログラムカンタ（PC）の値を基準として求められるよう I 形式のオペランドに相対アドレスを指定する。相対アドレスは、アドレスの差（相対値）を表す符号付整数である。

例）`beq $s0, $s1, LABEL`

ここでアセンブリ言語のオペランドの例に用いた“`LABEL`”は、分岐先の命令の所在場所に付した一種の識別子であり、ラベルと呼ばれる。

詳細、およびその実際の使用例は、「14.1 節 条件分岐命令」を参照。

擬似直接アドレシング（pseudodirect addressing）

オペランドに（相対値ではなく）絶対アドレスを（間接的でなく）直接指定する。

例）`j LABEL`

このアセンブリ言語でのラベル指定オペランドの例の場合、本来の直接アドレシング（direct addressing）では、`LABEL` のアドレスを 32 ビットで指定したいが、MIPS の機械語の命令長が 32 ビット固定であるので、本来 32 ビットのオペランドは少々工夫しても機械語命令の中に収まらない。

そこで、MIPS では、J 形式の 26 ビットのオペランドによって、疑似的に直接アドレシングをまねた擬似直接アドレシングが用いられる。

詳細、および実際の使用例は、「14.3 節 無条件ジャンプ命令」を参照。

13.3 プログラム例

C 言語で書かれた次のプログラムは、

```
int a, b;
b = a + 5;
```

MIPS のアセンブリ言語により、以下のように書ける。

```
lw    $s0, 0($fp)      #局所変数 a の領域はスタック上に取られる
addi  $s1, $s0, 5
sw    $s1, -4($fp)     #局所変数 b の領域はスタック上に取られる
```

2 つの整数の和の 2 倍を求めるプログラム例は、次の通り。

C 言語

```
int a, b, y;
y = (a + b) * 2;
```

MIPS のアセンブリ言語

```
lw    $s0, 0($fp)     #局所変数 a の領域はスタック上に取られる（図 13.1）
lw    $s1, -4($fp)    #局所変数 b の領域はスタック上に取られる
add   $t0, $s0, $s1
sll   $s3, $t0, 1
sw    $s3, -8($fp)    #局所変数 y の領域はスタック上に取られる
```

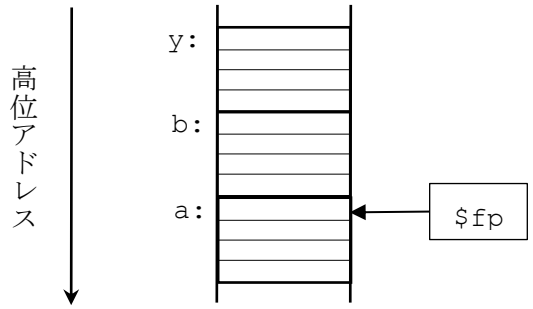


図 13.1 スタックのイメージ

練習問題 12

1. 以下の MIPS のアセンブリ言語の命令の下線部のオペランドのアドレシングモードを何と呼ぶか。

   i)　　`lw  $s0, -4($fp)`

   ii)　　`lw  $s0, -4($fp)`

   iii)　　`addi $s0, $zero, 10`

2. 上の問 iii) の命令実行後のレジスタ`$s0` の値を十進数で答えなさい。

3. MIPS のアセンブリ言語の命令

   ```
   lw  $s0, -4($fp)
   ```

   に関して、以下の問に答えなさい。

   i)　このアセンブリ言語の命令を機械語のコードに変換し、二進数で表しなさい。

   ii) この命令の実行時のフレームポインタ `$fp` の値が十六進で `0xff000008` であるとき、このアセンブリ言語の命令の第２オペランドの実効アドレスは何か、十六進数で答えなさい。(即ち、メモリの何番地のデータがレジスタにロードされるか。)

第 14 講　プログラム実行の制御構造

本講では、分岐や繰り返しなどのプログラムの実行の流れを制御する方法を、以下の観点から理解することを目的とする。

・条件によってプログラムを分岐させるか否かの指示
・無条件のジャンプ
・サブルーチンコールとリターン
・プログラミング言語の if 文、for 文、関数コールなどのアセンブリ言語による実現

14.1 条件分岐命令

条件分岐命令は、条件が成立すれば指定されたアドレスの命令から実行を行い（分岐する)、そうでなければ、この命令の次の命令を実行する。

・branch on equal

ニーモニック　beq $*rs*, $*rt*, *C*

動作　　レジスタ $*rs* と $*rt* の値が等しいとき、指定されたアドレスに分岐する。

（if ($*rs* == $*rt*) goto *PC* + 4*$C$ ）

例）beq $s0, $s1, LABEL

コード　I形式　000100　10000　10001　****************

beq　$s0　$s1　(LABEL-.)/4-1

ここで、アセンブリ言語における一種の識別子である LABEL は、13.2 節で述べた通り、分岐先の命令の先頭に付けられるラベルと呼ばれるもので、メモリのアドレスの代わりとなる。また、“.” は、この beq 命令自身のアドレスの代わりとなるもので、両者の差をとることで、ある種の相対アドレスが計算できる。但し、beq 命令フェッチ後の PC の値は “.+4” である点や、4 の倍数をそのまま用いるのは得策ではない点に注意

ラベルの使用例)

```
loop:  addi $s0, $s0, 1
        ・・・・
        ・・・・
       beq $s0, $s1, loop
```

条件分岐命令の I 形式の第 3 オペランドの値から分岐先の命令の所在場所となる実効

アドレスを求めるには、アドレシングモードとして、PC 相対アドレシングが用いられる（13.2 節参照）。

命令フェッチ後 PC の値は 4 増加しているので(MIPS の命令長は 4 バイト固定であるので)、この命令の次の命令のアドレスを基準とした相対値となる。

但し、命令は 4 の倍数のアドレスからの 4 バイトのメモリに格納されるので、実際のオペランドの値 *C* は、4 の倍数のアドレスの相対値の 1/4（下位の 2 ビットの 00 を省く）である符号付整数である。

実効アドレスは、*PC* + 4**C* （*C* はオペランドに与えられた符号付整数の定数）である。

条件分岐命令の命令形式はI形式が用いられるので、*C* は 16 ビットの符号付整数で与えられ、現在の命令から概ね±2^{17} 番地のアドレスを指定することができる。即ち、条件分岐命令は、この命令から前後 2^{15} 命令に分岐することができる。次の命令が 0 の位置であり、この命令自体は−2^{15} 以上 0 未満の範囲（これを前半分と呼ぶ（前日など）か後と呼ぶ（後戻りなど）かは日本語が難しい点の一つだが、分岐の意味なら後方（backward））の−1 に位置する。多くの場合、C 言語の if 文による分岐や、for 文、while 文によるループを実現するための分岐先の範囲は、この−2^{15} 以上 2^{15} 未満の範囲に収まる。

アセンブリ言語のプログラムを書く際は、

```
beq $s0, $s1, LABEL
```

のように、第3オペランドとして分岐先命令に付けられるラベル（この場合は、“LABEL”）を記入する。アセンブルに際して、アセンブラがアドレスの相対値をもとに、I形式のオペランドの値を計算してくれる。

・branch on not equal

ニーモニック　bne $*rs*, $*rt*, *C*

動作　　レジスタ $*rs* と $*rt* の値が等しくないとき、指定されたアドレスに分岐する。

（if ($*rs* != $*rt*) goto *PC* + 4**C* ）

例）bne $s0, $s1, LABEL

コード　I形式　000101　10000　10001　****************

bne　$s0　$s1　(LABEL-.)/4-1

“.” は、この bne 命令自身のアドレスの代わりとなる。

14.2 条件分岐命令と共に用いる比較命令

この命令の次の条件分岐命令で用いるために、大小比較を行った結果をレジスタに残す。

・set on less than

ニーモニック　slt $*rd*, $*rs*, $*rt*

動作　符号付整数として、レジスタ $*rs* の値が $*rt* の値より小さければ、レジスタ $*rd* に 1 を、そうでなければ 0 を格納する。

（$*rd* ← ($*rs* < $*rt*) ）

例）slt $t0, $s0, $s1

コード　R 形式

000000	10000	10001	01000	00000	101010
slt	$s0	$s1	$t0		slt

・set on less than immediate

ニーモニック　slti $*rd*, $*rs*, *C*

動作　符号付整数として、レジスタ $*rs* の値が符号拡張した即値 *C* の値より小さければ、レジスタ $*rd* に 1 を、そうでなければ 0 を格納する。

（$*rd* ← ($*rs* < *C*) ）

例）slti $t0, $s0, 5

コード　I 形式

001010	10000	01000	0000000000000101
slti	$s0	$t0	即値 5

14.3 無条件ジャンプ命令

無条件で命令実行の順序を変える（ジャンプする）命令

・jump

ニーモニック　j *C*

動作　指定されたアドレスに無条件ジャンプする。

（プログラムカウンタに *C* から求められたジャンプ先のアドレスを格納する）

（goto (*PC*&0xf0000000) + 4**C* ）

例）j LABEL

コード　J 形式

000010	**************************
j	(LABEL&0x0fffffff)/4

jump 命令の J 形式のオペランドの値からジャンプ先の命令の所在場所となる実効アドレスを求めるには、アドレシングモードとして、擬似直接アドレシングが用いられる。（13.2 節参照)。

本来の直接アドレシングは、オペランドに絶対アドレスを直接指定する。

例えば、アセンブリ言語で j LABEL の場合、ジャンプ先である LABEL として、32 ビットのアドレスを与えるべきである。

しかし、MIPS の機械語の命令長は 32 ビット固定長であり、オペコードが 6 ビットあるため、オペランドは 26 ビットに納めなければならない（J 形式）。

そのため、機械語のコードとしてのオペランドの値は、ジャンプ先 32 ビットアドレス LABEL の下位 28 ビットの 1/4（26 ビット）とする（命令は 4 の倍数のアドレスのメモリに格納されるので、アドレスの下位 2 ビットは常に 00）。アセンブルに際しては、アセンブラがこの J 形式のオペランドの値を計算してくれる。

32 ビットの実効アドレスは、PC の上位 4 ビットと、オペランドの値を 4 倍した値（下位 28 ビットになる）を結合して得られる（擬似直接アドレシング）。

これにより、jump 命令では、アドレスの上位 4 ビットが共通となるアドレス、即ち 32 ビットでアドレシングできる全メモリ空間中、1/16 の範囲でジャンプができる。

・jump register

ニーモニック　`jr $`*rs*

動作　　レジスタ `$`*rs* が値として保持するアドレスに無条件ジャンプする。
（`goto $`*rs*）

例）`jr $ra`

コード　R 形式　000000　11111　00000　00000　00000　001000

jr　　$ra　　jr

アドレス（値）の所在レジスタの指定にはレジスタアドレシングが用いられている。ジャンプ先の命令の所在場所は、いわば「間接アドレシング」で得られており、擬似直接アドレシングとは異なり、（仮想記憶等が有効な）全メモリ空間のどこでもよい。

・jump and link

ニーモニック　`jal` *C*

動作　　サブルーチンコール用。戻り先アドレスをレジスタ `$ra` に格納して、指定されたアドレスに無条件ジャンプする。J 形式のオペランドのアドレシングモードは、擬似直接アドレシング

従って、サブルーチンからのリターンは、

```
jr $ra
```

を用いることになる。

また、サブルーチン内からサブルーチンを呼ぶ場合は、戻り先アドレスを格納した `$ra` の値を他のレジスタやメインメモリ（典型的には、スタック）に一時退避しなければならない。

例）`jal LABEL`

コード　J 形式　000011　**********

jal　　(LABEL&0x0fffffff)/4

14.4 制御構造

C 言語のようなプログラミング言語における if 文、while 文、関数呼び出しなどの制御構造は、MIPS のアセンブリ言語により以下の例のように書ける。

最下位ビットが 1 であることを検知する

レジスタ $s0 の最下位ビットが 1 のときに、ある処理をするプログラム

```
        andi   $t0, $s0, 1
        beq    $t0, $zero, L1
        最下位ビットが 1 の時の処理
L1:     次の命令
```

絶対値を求める

C 言語

```
if (a < 0)
  a = -a;
```

MIPS のアセンブリ言語

```
        slti   $t0, $s0, 0
        beq    $t0, $zero, L1
        sub    $s0, $zero, $s0
L1:     次の命令
```

1 から 10 までの和を求める

C 言語

```
n = 10;
s = 0;
for (i = 1; i <= n; i++)
  s = s + i;
```

MIPS のアセンブリ言語

```
# n = $s0
# s = $s1
# i = $s2
        addi  $s0, $zero, 10     # n = 10
        addi  $s1, $zero, 0      # s = 0
        addi  $s2, $zero, 1      # i = 1
```

```
LOOP:  slt   $t0, $s0, $s2      # n < i ?
       bne   $t0, $zero, L1     # if (i > n) goto L1
       add   $s1, $s1, $s2      # s = s + i
       addi  $s2, $s2, 1        # i++
       j     LOOP               # goto LOOP
L1:    次の命令
```

２引数の内、小さくない方を返す関数

C 言語

```
int max(int x, int y)
{
  if (x >= y)
    return x;
  else
    return y;
}
```

MIPS のアセンブリ言語

```
# x = $a0
# y = $a1
MAX:
      slt   $t0, $a0, $a1     # x < y ?
      bne   $t0, $zero, L1    # if (x < y) goto L1
      addi  $v0, $a0, 0       # if (x >= y)
                              #   set x to return value
      j     L2                # goto L2
L1:   addi  $v0, $a1, 0       # if (x < y)
                              #   set y to return value
L2:   jr    $ra               # return
```

練習問題 13

1. MIPS のアセンブリ言語の命令

```
j   LABEL1
```

において、この命令が存在するメモリのアドレスが、十六進数で 0x10206088 であり、0x102060a4 番地にジャンプしたいとする。このアセンブリ言語の命令を機械語のコードに変換し、二進数で表しなさい。

2. MIPS のアセンブリ言語の命令

```
bne   $s1, $s0, LABEL2
```

において、分岐先は 5 命令先（次の次の次の次の次の命令）であるとする。このアセンブリ言語の命令を機械語のコードに変換し、二進数で表しなさい。

3. MIPS の命令

```
jr   $ra
```

は、どのような命令で、どのように利用できるか述べなさい。

課題 III

問題　次の 32 ビットの MIPS のアセンブリ言語のプログラムについて、以下の問に答えなさい。

```
1.          addi $s0, $zero, 6
2.          addi $s1, $zero, 1
3. LOOP:    sll  $s1, $s1, 1
4.          addi $s0, $s0, -1
5.          beq  $s0, $zero, EXIT
6.          j    LOOP
7. EXIT:    sw   $s1, -4($fp)
```

(1) 1 行目のニーモニックを機械語の命令のコードに変換しなさい（二進数)。
(2) 5 行目のニーモニックを機械語の命令のコードに変換しなさい（二進数)。
(3) 7 行目の下線部のオペランドのアドレシングモードを何と呼ぶか答えなさい。
(4) 7 行目の命令実行時に、フレームポインタ`$fp` の値が十六進数で `0x10010010` であったとする。このときの実効アドレスは何か。十六進数で答えなさい。
(5) 5 行目の命令は何回実行されるか答えなさい。
(6) 7 行目の実行によってメモリに格納される値を十進数で答えなさい。
(7) このプログラムは何をするプログラムか説明しなさい。即ち、1 行目でレジスタ `$s0` に設定された 1 以上の値（*m* とする）に対して、7 行目でどのような値がメモリに格納されるかについて、*m* を使用して説明すること。

提出

講義開始時の受講案内、及び Moodle 上の指示に従って提出すること。

総合練習問題

（提出の必要はないが、各自で解答すること。解答例は、107 ページ以降に掲載している）

次のように文法が与えられている。

生成規則

<代入文> → *識別子* = <式> ;

<式> → <式> + <項> | <式> - <項> | <項>

<項> → (<式>) | *識別子* | *定数*

非終端記号

<代入文>, <式>, <項>

終端記号

=, +, -, (,), ;, および *識別子*, *定数*

出発記号

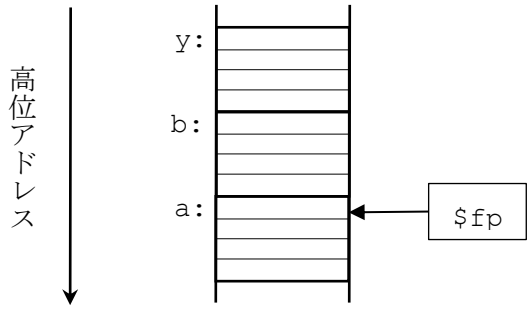
<代入文>

入力文字列

```
y = a - (b + 2);
```

に対して、

1. 図 4.1 のように「字句解析の結果」と解析木を書きなさい。
2. 図 4.4 のような構文木を作成しなさい。
3. 上問の構文木に従って、MIPS のアセンブリ言語のプログラムを書きなさい。但し、変数 a, b, y は、int 型であり、その大きさは 32 ビットである。また、これらの変数は、下図のようにスタック上に割り当てられるとする。



練習問題解答（第 3 部）

練習問題 9（第 10 講）

1. （10.1 節、10.3 節参照）
   i) Central Processing Unit
   ii) Arithmetic Logic Unit
   iii) Direct Memory Access
2. （10.1.1 項参照）
   i) 次に実行すべき命令のアドレスを保持するレジスタ
   ii) フェッチされた命令のコードを一時的に保持するレジスタ
   iii) メモリから読み出したデータや演算結果、プログラム内で参照されるメモリのアドレスなどを一時的に保持するなど、汎用的に用いられるレジスタ

練習問題 10（第 11 講）

1.（11.1.2 項参照）
   32 本
   `$zero` 値が 0 に固定される
   `$ra` サブルーチンのリターンアドレスを格納する。
2. （11.1.3 項参照）
   R 形式 3 つのレジスタとシフト量
   I 形式 2 つのレジスタと、16-bit 定数（即値、または偏位）
   J 形式 26-bit アドレス
3.（11.2 節参照）
   （命令フェッチ）命令をメインメモリから命令レジスタに読み込む
   （命令デコード）命令を解読する
   （命令実行）ロードすべきメモリアドレスの計算を行う
   （メモリアクセス）メインメモリからデータを読み出す
   （書き込み）読み出されたデータをレジスタに格納する

練習問題 11（第 12 講）

1.（命令表参照）

i) 000000 10001 00100 01001 00000 100000
add $s1 $a0 $t1 add

ii) 001000 10010 01000 1111111111111110
addi $s2 $t0 -2

2. 5

3. sub $t0, $s0, $s1

4. addi $s0, $s0, 1

5. andi $t1, $s1, 1

6. add $s1, $s0, $s0 または、 sll $s1, $s0, 1

練習問題 12（第 13 講）

1.（13.2 節参照）

i) レジスタアドレシング

ii) ベースアドレシング

iii) 即値アドレシング

2. 10

3.（命令表参照）

i) 100011 11110 10000 1111111111111100
lw $fp $s0 -4

ii) 0xff000004

練習問題 13（第 14 講）

1.

```
000010  00 0000 1000 0001 1000 0010 1001（14.3 項参照）
  j           (LABEL1&0x0fffffff)/4
```

（解説）（14.3 節 「無条件ジャンプ命令」を参照）

この命令（正確には次の命令）のアドレスと、ジャンプ先のアドレスの上位 4 ビットは同一であるので、オペランドのアドレシングモードが擬似直接アドレシングである j 命令でジャンプ可能である。

ジャンプ先アドレスは、２進数で

0001 0000 0010 0000 0110 0000 1010 0100

である。オペランドは、擬似直接アドレシングであるので、上位 4 ビットと下位 2 ビットを除いた、

0000 0010 0000 0110 0000 1010 01

となる。

2.

```
000101 10001 10000 0000000000000100
  bne    $s1    $s0   (LABEL2-.)/4-1
```

（解説）（14.1 節 「条件分岐命令」を参照）

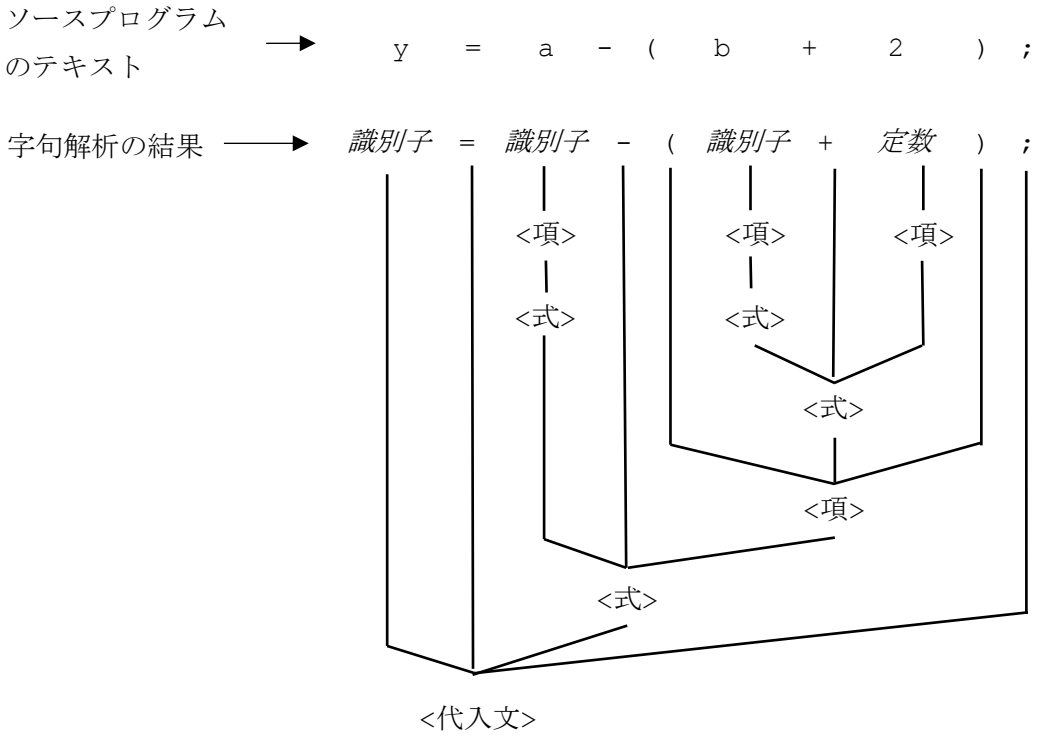
bne 命令の第３オペランドのアドレシングモードは PC 相対アドレシングである。この命令実行時には、PC は１つ先の命令のアドレスを持つので、この PC の値から 4 つ先の命令に分岐することになる。従って、第３オペランドの値は、十進数で 4。

3. （14.3 節参照）

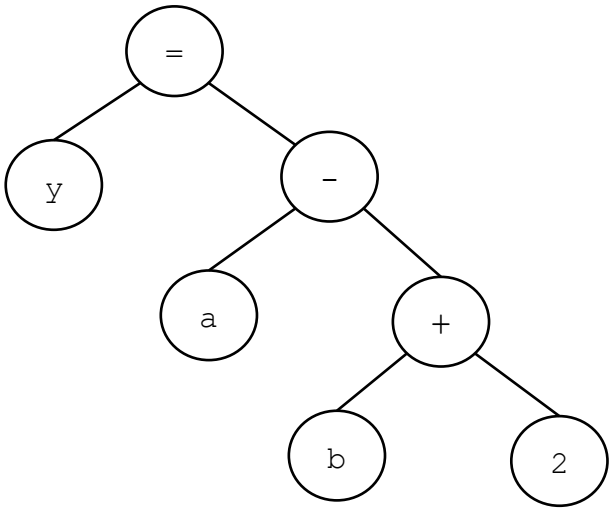
（例文）レジスタ $ra が値として保持するアドレスに無条件ジャンプする。jal (jump and link) 命令では、戻り先アドレスを $ra に格納するので、サブルーチンからのリターンに用いることができる。

総合練習問題の解答

1.



2.



3.

```
lw    $s0, -4(fp)
addi  $t0, $s0, 2
lw    $s1, 0(fp)
sub   $s2, $s1, $t0
sw    $s2, -8(fp)
```

第 15 講　総合演習

注意点
○　講義で解説を行うが、それまでに Moodle 上の指示に従って各自で解答しておくこと。

問題 I　以下の記述のうち、正しい文に○を、誤った文に×を、解答欄に記入しなさい。

(1) 本学の学生証のような IC カードは、あまりに小さすぎるので OS を持っていない。

(2) スマートフォン上のアプリケーションプログラム（いわゆる「スマホのアプリ」）は、ファイルとして補助記憶装置上に置かれている。

(3) 命令をメインメモリから読み出し、CPU 内のレジスタに格納する動作を、命令フェッチと言う。

(4) C 言語のようなプログラミング言語は、CPU の命令セットに依存する。

(5) 関数呼び出し毎に確保されるフレームは、関数呼び出し時にスタック上に確保され、リターン時に解放される。

(6) UNIX においては、ファイル名はディレクトリに記録される。

(7) 特権命令とは、システム管理者や、特権的な地位にある利用者のみに実行を許された命令である。

(8) CPU 内の汎用レジスタには、プログラム内で参照されるメモリのアドレスを格納することができない。

(9) CPU 内にある ALU は、Arithmetic Logic Unit の略であり、算術演算や論理演算を行う論理回路である。

(10) コンパイラのコード生成は、CPU の命令セット毎に異なる機械語のコードを生成する。

問題 II　以下の各問の解答として最適なものを選択肢の中から 1 つ選択し、記号を解答欄に記入しなさい。

(1) 次の選択肢のうち、応答性の制約が最も厳しいのはどれか。

a: スーパーコンピュータによる地球温暖化シミュレーション
b: 自動車の障害物検知・衝突回避
c: ウェブブラウザからのネットショッピング
d: 毎月の電話料金請求業務

(2) ソースファイルの説明として正しいのは次のどれか。

a: トランスレータへの入力ファイル
b: トランスレータが使用する中間ファイル
c: トランスレータの出力ファイル
d: トランスレータのロードモジュールが格納されたファイル

(3) 補助記憶装置の入出力の効率向上にバッファが必要な理由として、誤っているのは次の

どれか。

a: ブログラムとハードディスクなどの入出力単位のサイズを調整するため。

b: メモリとハードディスクなどのアクセス時間の差を調整するため。

c: データの共有や再利用に際して、キャッシュとしての効果を期待するため。

d: 入出力に際して、タイムラグなく、直接ハードディスクなどにアクセスするため。

(4) C 言語において、一つの字句として扱われないのは次のどれか。

a: `0x86` **b**: `x86` **c**: `0*86` **d**: `"0*86"`

(5) C 言語における字句 `while1` は、次のどの種類に属するか。

a: キーワード **b**: 識別子

c: 定数リテラル **d**: 文字列リテラル

(6) アドレシングモードの説明として正しいのは次のどれか。

a: 仮想アドレスから物理アドレスに変換する方式

b: アセンブラがオペランドの実効アドレスを計算する方式

c: プログラム内の次に実行する命令のアドレスをモニタリングする方式

d: オペランドによって命令の対象となるデータ等の所在場所を指定する方式

(7) 以下の MIPS のアセンブリ言語の命令の内、レジスタ`$s1` の値を 1 増加させる命令として正しいのは次のどれか。

a: `add $s1, $s1, 1` **b**: `add $s1, 1, $s1`

c: `addi $s1, $s1, 1` **d**: `addi $s1, 1, $s1`

(8) 以下の MIPS のアセンブリ言語の命令の内、レジスタ`$s1` の値を符号付整数として 2 倍する命令として正しいのは次のどれか。

a: `add $s1, $s1, $s1` **b**: `sll $s1, $s1, 2`

c: `srl $s1, $s1, 2` **d**: `sra $s1, $s1, 1`

(9) OS のカーネル（核）の説明として正しくないのは次のどれか。

a: 特権モードで実行される。 **b**: 割り込みにより起動される。

c: メインメモリに常駐する。 **d**: 最も高い優先度でスケジュールされる。

(10) 十進数 -5 は、8 ビット 2 の補数表現で表すと、`11111011` となる。これを 16 ビット整数に符号拡張すると、次のどれになるか。

a: `00000000 11111011` **b**: `11111111 11111011`

c: `10000000 11111011` **d**: `00000000 00000101`

問題 III 以下の各問の解答として最適な語句を下の選択語群の中から選択し、解答欄に記入しなさい。

(1) login 時に、login ID とパスワードを照合し、利用者の正当性を確認することを何というか。

(2) ハードディスクがアームを移動させてヘッドを指定されたシリンダに位置付ける動作

を何と呼ぶか。

(3) CPU 内で次に実行すべき命令のアドレスを保持するレジスタを何と呼ぶか。

(4) UNIX において、プログラムの機械語の命令列が格納されるメモリ領域は何と呼ばれるか。

(5) プログラミング言語で書かれたプログラムを、機械語、あるいはアセンブリ言語のプログラムに変換する言語処理系を何と呼ぶか。

選択語群　セッション管理、利用者認証、アクセス制御、ディジタル署名、シーク、トラック選択、回転待ち、データ転送、命令レジスタ、フレームポインタ、プログラムカウンタ、インデクスレジスタ、スタックポインタ、スタック領域、ヒープ領域、データ領域、テキスト領域、アセンブラ、インタプリタ、コンパイラ、リンカ

問題 IV　次に与えられた文法に関して、以下の問に答えなさい。

生成規則

〈加法式〉 → 〈加法式〉 + *識別子* | 〈加法式〉 - *識別子* | *識別子*

〈式〉 → 〈式〉 & 〈加法式〉 | 〈加法式〉

非終端記号

〈加法式〉および〈式〉

終端記号

+，-，&，および *識別子*

出発記号

〈式〉

(1) 入力文字列

```
x1 - x2 & c3
```

に関して、字句解析の結果、x1，x2，c3 は識別子、-，& はそれぞれそのまま終端記号として扱えることが分かった。字句解析の結果となる終端記号列を上記の文法に従って構文解析し、解析木を解答欄に書きなさい。

(2) 上問の解析木に対応する構文木を解答欄に書きなさい。ただし、入力文字列の一部を用いて識別子を区別できるようにすること。

(3) 演算子として +，- は、どちらの優先度が高いか。以下の選択肢から選びなさい。

+ の方が高い、　　- の方が高い、　　双方同じ

(4) 演算子として +，& は、どちらの優先度が高いか。以下の選択肢から選びなさい。

+ の方が高い、　　& の方が高い、　　双方同じ

(5) 演算子として & は、右結合か、左結合か、それとも不明か。

問題V　C言語の文　`c = getc(fp);` は、ファイルポインタ `fp` で参照されるファイルから読み込んだ文字を `int` 型変数 `c` に代入するものである。以下の文章はその実行過程を表すが、括弧内に下の選択語群より最も適した語句を選択し、解答欄に記入しなさい。

プロセスが標準入出力関数 `getc()` を呼ぶと、入出力ライブラリが用意する（　A　）から変数 `c` にデータがコピーされる。もしこの（　A　）が空であれば、ファイルからの読み出しのためのシステムコールである（　B　）が発行され、核（カーネル）に実行制御が移り、これを実行したプロセスは入力の完了を待つため（　C　）状態に遷移し、プログラムの実行は中断される。

核に制御が移った後、核内の（　A　）が空なら、補助記憶装置からデータが読み出され（　D　）転送により核内の（　A　）へデータが格納される。（　D　）転送終了後、割り込みの一つである（　E　）により、核は、核内の（　A　）から入出力ライブラリが用意する（　A　）へコピーを行った後、（　C　）状態にあったプロセスは（　F　）状態に遷移する。その後、（　F　）状態にあったプロセスにCPUが割り当てられると実行状態に遷移し、システムコールから復帰することでプログラムの実行を再開する。

選択語群　`open` システムコール、タイマ割り込み、デバイスドライバ、ディレクトリ、バッファ、`read` システムコール、プログラム割り込み、`fork` システムコール、待ち、実行可能、DNA、DDT、DMA、DHC、入出力割り込み

問題VI　定時間を2としたラウンドロビンによるプロセスのスケジューリングを行う。下の表に示す時刻に4つのプロセスがシステムに到着し、表に示す処理期間をCPUで消費する。但し、この間新たなプロセスの到着や入出力はなく、プロセス切り替えの時間は無視するものとする。

プロセス	到着時刻	処理時間
P	0	12
Q	3	6
R	5	8
S	9	4

(1) 解答用紙のタイミングチャートに各プロセスのシステムへの到着時刻（△）、CPUを割り当てられている時間（太線）、終了時刻（▲）を記入し、各プロセスのターンアラウンド時間を求めなさい。

(2) あるプロセスが CPU を保持している間に、他のプロセスは＊＊行列に繋がれて CPU が空くまで待つ。＊＊を答えなさい。（＊＊行列を英語表記で解答してもよい）

(3) プロセスの切り替えについて考える。プロセスの中断に際して一旦退避した状態を回復して再開させることを何と呼ぶか。

(4) スケジューリングの評価基準の一つに応答性がある．応答性を測る尺度にはターンアラウンド時間と応答時間がある．応答時間とは何か説明しなさい。

問題 VII　計算機の実行過程に関して、以下の問に答えなさい。

(1) CPU がメインメモリからデータを読み出す過程を、以下のステップを並べ替えて示しなさい。但し、順番は **a, b, c** 等の記号で解答すること。

a. メインメモリがデータをデータバスに出す。

b. CPU がデータバスからデータを取り込む。

c. CPU が読み出し要求を制御バスに出す。

d. CPU がアドレスをアドレスバスに出す。

(2) MIPS アーキテクチャを採用した CPU における「`add $t0, $s0, $s1`」の実行過程を、以下のステップを並べ替えて示しなさい。ただし、パイプライン処理は考慮する必要はない。順番は **a, b, c** 等の記号で解答すること。

a. ALU の出力をレジスタ`$t0` に格納する。

b. 命令をフェッチする。

c. レジスタ`$s0` と`$s1` の値を ALU で加算する。

d. 命令をデコードし、レジスタ`$s0` と`$s1` の値を読み出す。

問題 VIII　次の MIPS のアセンブリ言語のプログラムについて、以下の問に答えなさい。

```
1.          addi $s0, $zero, 0
2.          addi $s1, $zero, 1
3.          addi $s2, $zero, 4
4.          addi $s3, $zero, 5
5.   LOOP:  add  $s0, $s0, $s3
6.          ________________          # $s2 ← $s2 - 1
7.          beq  $s2, $zero, EXIT
8.          j  LOOP
9.   EXIT:  sw $s0, -12($fp)
```

(1) 5 行目の命令を機械語のコードに変換しなさい。

(2) 7 行目の命令を機械語のコードに変換しなさい。

(3) 9 行目の命令実行時に、フレームポインタ`$fp` の値が十六進数で `0x1001000c` であったとする。このときの実効アドレスは何か。十六進数で答えなさい。

(4) 6 行目の命令として、レジスタ`$s2` から 1 を減算し、結果をレジスタ`$s2` に格納する命令をアセンブリ言語で書きなさい。

(5) 5 行目の命令は何回実行されるか。

(6) 9 行目の実行によってメモリに格納される値を十進数で答えなさい。

(7) このプログラムは何をするプログラムか。即ち、3、4 行目でレジスタ`$s2`、`$s3` に設定された値（n と m とする）に対して、9 行目でどのような値がメモリに格納されるか。

【参考】　パイプライン処理

A. 1 パイプライン処理とは

パイプライン処理（pipelining）とは、処理速度の向上を目的として、処理過程を直列に連結し、流れ作業のように処理することで、複数の処理を並列に効率よく実行する仕組みである。

パイプライン処理は、一般的に以下の方法により処理の高速化を図る。

・ある処理機能を複数ステージに分割する。

・ステージごとに独立処理できるハードウェア機構を装備する。

・これらの複数のステージを時間的にオーバラップさせる。

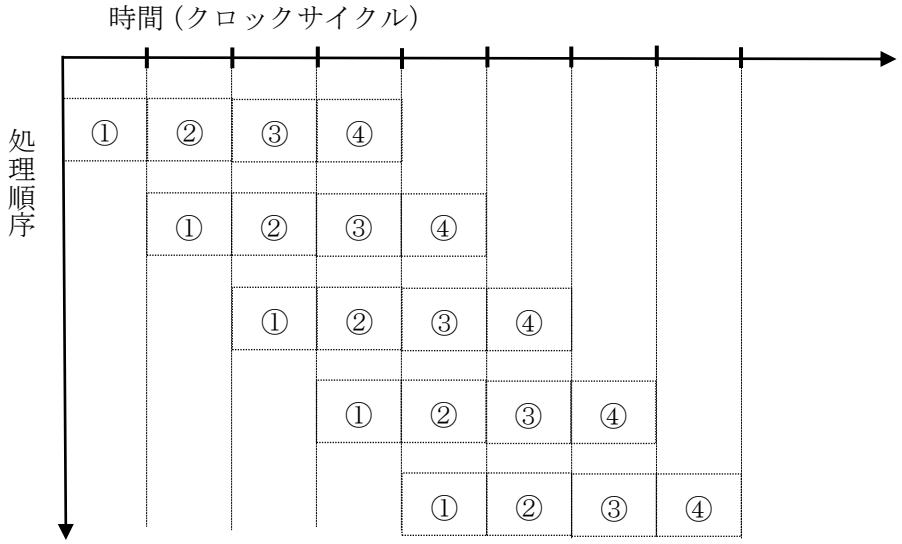


図 A.1 パイプライン処理の概念図

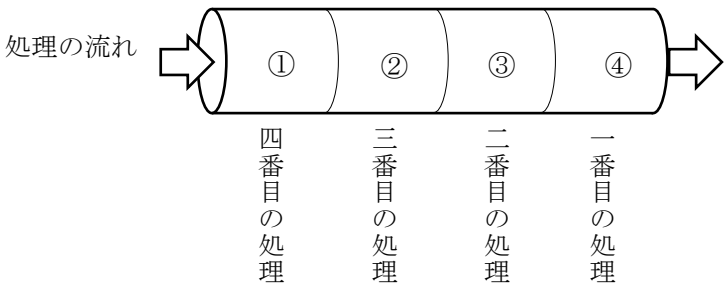


図 A.2 パイプの中を処理が流れるイメージ図

A. 2 MIPS におけるパイプライン処理

ここでは、命令パイプライン（(instruction pipeline: 命令処理におけるパイプライン）の一例として、MIPS を取り上げる。

MIPS のパイプライン処理は、次の 5 ステージからなる。

1. 命令フェッチ（IF）

   CPU がプログラムカウンタの値をアドレスバスに出し、メモリからの 4 バイトを命令レジスタに格納する。

   プログラムカウンタの値を 4 増加させる。

2. 命令デコード（ID）

   命令レジスタ内の命令を命令デコーダでデコード(解読)する。

   必要なデータをレジスタから読み出す。

3. 命令実行（EX）

   ALU での演算実行や、メモリへのアクセスの際のアドレス生成（実効アドレスを求める）を行う。

4. メモリアクセス（MEM）

   メモリへの読み出しや書き込みを行う。

5. 書き込み（WB）

   演算結果やメモリから読み出されたデータをレジスタに書き込む。

具体的な命令実行におけるパイプライン処理の様子を図 A.3 に示す。

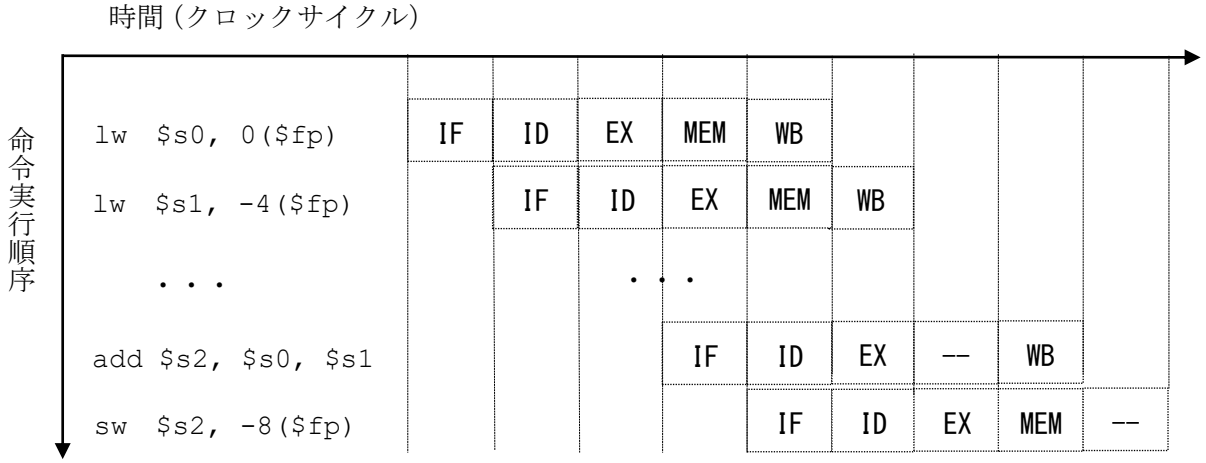


図 A.3 MIPS におけるパイプライン処理の例

・`lw $s0, 0($fp)`

1. IF
2. ID　　命令をデコードするとともに、`$fp` のデータを読み出す。

3. EX　実効アドレスの計算を行う。ここでは、0+$fp
4. MEM　上記で計算されたからアドレスのメモリからデータを読み出す。
5. WB　読み出されたデータを$s0 に格納する。

・add $s2, $s0, $s1

1. IF
2. ID　命令をデコードするとともに、$s0, $s1 のデータを読み出す。
3. EX　$s0, $s1 の値を ALU で加算する。
4. --　何もしない（メモリアクセスは不要)。
5. WB　ALU の出力を$s2 に格納する。

・sw $s2, -8($fp)

1. IF
2. ID　命令をデコードするとともに、$s2, $fp のデータ読み出す。
3. EX　実効アドレスの計算を行う。ここでは、-8+$fp
4. MEM　上記で計算されたアドレスのメモリに$s2 のデータを書き込む。
5. --　何もしない（レジスタへのデータ格納は不要)。

A.3 パイプラインハザード

ここまでは、パイプライン処理は何の問題もなくスムーズに処理が進むと仮定してきた。しかし、パイプライン処理においては複数の命令の異なるステージの処理を同時に行うため、次のクロックサイクルで次の命令を実行できないことがある。これをパイプラインハザード (pipeline hazard) と呼ぶ。これにより、パイプラインの処理効率が低下する。

例えば、

```
add   $t1, $s1, $s2
sub   $t3, $t1, $t0
```

において、add 命令により$t1 に値が書き込まれる第 5 ステージ（WB）終了まで、$t1 の値を必要とする次の sub 命令の第 3 ステージ（EX）を実行できない。

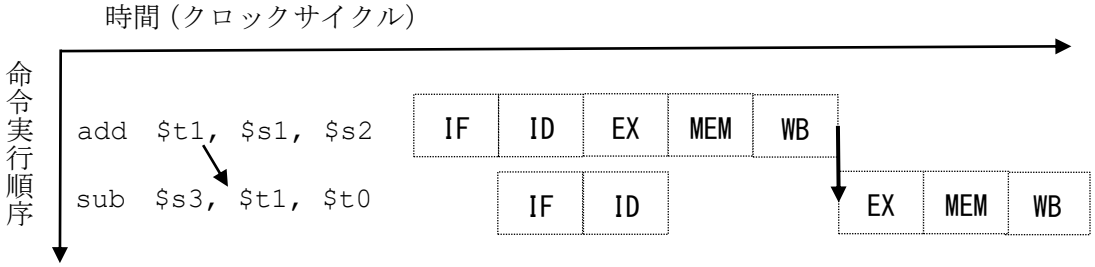


図 A.4 MIPS におけるパイプラインハザードの例