# システムプログラム
# 最終レポート

2023 年 12 月 31 日 (日)

6321120

横溝　尚也

# 1 課題 1

## 1.1 課題内容

```
$ gcc sample.c -o sample
$ ./sample
$ ^C
```

**ターミナル上で上記の C プログラムのコンパイル，実行，ctrl-c での停止を行った際，OS が行う処理を可能な限り詳細に説明せよ．**
**※ プロセス，メモリ，ファイルシステム，シグナル，シェルの用語は用いること．**

## 1.2 全体の概要

ターミナル上でコンパイル、実行、プログラム停止までの大まかな流れについて説明し、詳細な説明を行う上での概要を説明する。説明の中には OS が行う処理でないものもありますがご容赦ください。まず初めに人間が人間語で書いたコードを機械が読める機械語へと変換する。この作業をコンパイルという。我々は機械語に翻訳するために gcc コマンドを実行する。このコンパイルの作業にもいくつか流れがあるため、初めにコンパイルの詳しい流れについて説明する。

コンパイルが終了するとターミナルが再び入力待ち状態になる。そこで我々は実行するための命令 *./sample* を入力する。この後に計算機で行われている処理は以下が挙げられる。

- プロセスの生成とその実行管理
- 必要なメモリ量の管理
- 必要なファイルへのアクセスなどのファイル管理

これらの処理はすべてカーネルで行われる。そのため、カーネルの概要とカーネルの各機能をそれぞれ説明していく。

カーネルによってさまざまな処理を行うことができるが、いつどんな処理をすればよいのかカーネルに命令する必要がある。実際我々はターミナルに命令を入力し、計算機が実行に必要な処理を自動で行い、再び入力待ち状態に戻る。このユーザとカーネルの仲介役として、うまくカーネルに命令を伝えるのがシェルである。このシェルについて説明を行う。

このような過程を経て最終的にプログラムが無事実行されていく。しかし、プログラム実行の上で意図しないイベントが発生することがある。実際我々は Ctrl-c によってプログラムを強制終了することが可能である。このようなときにプロセスに非同期的なイベントをプロセスに伝えるのがシグナルである。このシグナルの仕組みとその種類について最後に説明する。

## 1.3 コンパイルについて

コンパイル [2] とはコンパイラによってソースコードを何らかの中間形式に変換し、それを実行するインタプリタに渡すこと。またコンパイル言語は C,C++,java, などが挙げられる。これらの言語は機械語に変換するために以下の手順を追って翻訳する。

1. プリプロセス
2. コンパイル
3. アセンブル
4. リンク

これらの流れを追って説明していく。

### 1.3.1 プリプロセス

ソースプログラムがコンパイラによって コンパイルできる形式に変換することが目的である [3]。コンパイラ言語の中でも行われない言語も存在するが、C,C++ などではこの処理が行われる。マクロの展開や #include や#ifdef などのディレクティブの処理が行われる。前処理された結果は 「.i」という拡張子のついたファイルに保存される。gcc -E sample.c によりプリプロセスのみを行える。

### 1.3.2 コンパイル

プリプロセッサによって出力されたテキストファイル「.i」 をアセンブリ言語プログラムのテキストファイル「.s」 に翻訳する。gcc -S sample.c でアセンブリ言語記述のファイル生成まで行える。コンパイルには、いくつかの段階に分かれているため詳しく説明する。

**1. 字句解析**

字句解析 [1] とはプログラムの文字列を変数名、演算子、予約語など意味を持つ最小単位である字句 (トークン) に分解する。

**2. 構文解析**

字句解析によって分析したトークンをプログラム言語の文法に従って文法を解析する。また、IS 科 2 年の必修科目である情報科学実験 1 にて OCAML を使用した演習課題で行われたのが字句解析と構文解析であり、この処理に対応している。

**3. 意味解析**

意味解析 [1] とは変数の宣言としようとの対応付けやデータ型の整合性についてのチェックを行う。一般的には次に行う最適化のために 3 つ組み、 4 つ組み、逆ポーランドなどによる中間コードを生成する。

**4. 最適化**

実行時の処理時間や容量が少なくなるようにレジスタの有効利用を目的としてプログラムの再編成を行う。レジスタ [1] とは、CPU 内に内蔵されている少量で高速な記憶装置のこと。

**5. コード解析**

目的プログラムとして出力するコードを生成する。

### 1.3.3 アセンブル

アセンブラ言語「.s」を機械語命令に翻訳し，リロケータブル・オブジェクト・プログラムと呼ばれる形にパッケージ化して「.o」の形にする。gcc -c sample.c で次に行うリンクを行わないアセンブルまでを行ったファイル生成ができる。

#### 1.3.4 リンク

コンパイラによって生成された目的プログラムをロードモジュールにする。プログラムに使用しているライブラリモジュールなど実行に必要なものリンカによってをまとめ上げる。

## 1.4 コンパイル後の動作

コンパイルによって機械語に翻訳することができた。ユーザ側が./sample と入力することで計算機が行うことについて詳しく説明していく。

まず初めにユーザが実行コマンドを入力するとコンパイルしたファイルがメインメモリ (主記憶) に格納される。このときメインメモリに格納されているのはプログラムとデータの両方であり、両方とも格納されていなければ命令は実行できない。これがのちに説明するカーネルの機能であるメモリ管理が重要な理由の一つである。

その後、メインメモリ内の命令を一つずつ CPU で処理していく。以下がその流れである。

1. 命令を一つずつ読み出す (命令フェッチ)
2. 命令の解読 (デコード)
3. アドレス計算
4. オペランドのフェッチ
5. 実行

これらの過程を経て命令が実行される。次の命令をプログラムカウンタが読み出すことで次の実行サイクルへと移る。

#### 1.4.1 高速処理のためのアルゴリズム

プロセッサでは上記のような流れでメインメモリ内に格納された命令が実行されていく。その際に CPU が高速で命令を次々と実行するためのアルゴリズムがいくつか存在する。以下がその例である。

1. メモリ・ファイルの抽象化
2. パイプライン処理
3. メモリ間のデータの効率受け渡し

1. メモリとファイルの抽象化に関しては課題 2 で詳しく説明する。パイプライン処理とは一つの命令をステージという単位に分割し、各ステージを並列して処理を行うことである。パイプラインにもさまざまな形態があり、より効率の良いアルゴリズムを使用することで高速処理が可能となる。しかし、パイプラインにハザード (制御ハザード、データハザード) といった問題点がるため、投機実行や遅延分岐などによって問題を解消する必要がある。

他にはメモリとメモリの間でのデータの受け渡しを効率よく行うためにキャッシュメモリを利用している。キャッシュメモリとは CPU の処理速度とメインメモリへのアクセス速度の差を緩和するためのものである。プログラムの一部をメインメモリからキャッシュメモリにコピーしておき、CPU はキャッシュメモリからデータにアクセスすることで処理の高速化を図っている。

## 1.5 カーネルの概要

OS とは基本ソフトウェアのことであり、狭義で制御プログラムのことを指す。制御プログラムとはカーネル、デバイスドライバ、ファイルシステムで構成される。その中でもカーネルが OS の中核をなす部分である。他のプログラムがカーネルの機能を利用して動いている。概要で述べたとおり、カーネルの機能は以下のとおりである。

1. プロセス管理
2. メモリ管理
3. ファイル管理
4. デバイス管理

これらに関しては課題２でも触れるため、ここでは説明を省略する。

## 1.6 シェルについて

OS の中核をなすカーネルで様々な処理が行われているがユーザが入力した命令をカーネルに伝えるインターフェイスとなるのがシェルである。我々が普段、コンパイルする際などに入力しているターミナルはシェルへの入出力をサポートするソフトウェアである。

シェルでは文字列マッチングをするグロブ、入出力接続、コマンド履歴、数値演算や変数宣言、分岐など基本的なプログラムを書くことができる。また、ファイル操作、プログラム実行、テキストの印刷などの処理をシェルスクリプトというスクリプト言語にコーディングして実行ことができる [4]。

### 1.6.1 シェルの種類

シェルには以下のような種類が存在する。

- sh
- bash
- ksh
- zsh
- csh
- tcsh

この中でも bssh とは、UNIX で標準採用されているシェルである。最も基本的なシェルであり、多くの OS でサポートされている。また基本的なプログラムであればこの bash で記述可能である。以下に bash を用いて名前を入力し、その名前を出力する簡単なプログラムを記述した。


図 1: bash の実行例

## 1.7 シグナルによるプログラムの停止

コンパイルにより機械語に翻訳したものをメインメモリに格納し、１命令ずつ実行を行っていく。実行途中にメインメモリに書かれていないイベント (何らかのエラーなど) が発生したとき、その緊急度・重要度に応じて割り込んで処理を行わなければいけない。そのときにシグナルによってプロセスに非同期イベントを伝達する。このシグナルによってユーザの命令した強制終了が実行される。ここではシグナルについて詳しく説明する。

シグナルにはいくつかの種類がある。授業で学習したシグナルの一覧は以下である。

| シグナル名 | シグナル番号 | 詳細 |
|---|---|---|
| SIGHUP | 1 | 説明クローズ |
| SIGINT | 2 | Ctrl-c |
| SIGUQUIT | 3 | Ctrl-￥￥ |
| SIGILL | 4 | 不正な命令実行 |
| SIGKILL | 9 | 強制終了 |
| SIGSEGV | 11 | 不正なメモリアクセス |
| SIGSTOP | 19 | プロセスの一時停止 |
| SIGTSTP | 20 | Ctrl-z |

表 1: シグナルの種類

シグナル番号２番によってプログラム実行時に割込みが生じプログラムの実行が強制終了される。再びターミナルで入力待ち状態に戻ることで課題内容にある一連の流れが終了する。

# 2　課題 2

## 2.1　課題内容

**OS の以下の 3 つの機能が抽象化して提供している事項を説明せよ．**

**1．プロセス**

**2．仮想メモリ**

**3．ファイルシステム**

## 2.2　プロセス

プロセスとは CPU から見た処理の単位である。CPU ではメインメモリにあるプログラムを順に実行していく中で、いかに効率よく処理時間を短縮して実行するかがポイントになる。プロセスを分割して実行することはできず、メインメモリにあるプログラムをどのように制御するかをプロセス管理という。また類義語としてスレッドという用語があるが、スレッドとは一つのプロセスの中で並行動作することに対して、プロセス管理とはいくつかのプロセスを並行動作により処理効率を高めるといった違いがある。

### 2.2.1　プロセスの抽象化の目的

処理効率を高めるプロセス管理の手法として並行処理がある。これは CPU で実行するプロセスを見かけ上並行してプロセスが処理されているように見せかけ、効率を高める考え方である。まず計算機には一般的に複数の CPU が存在する。これはひとつの CPU で実施できるプロセスは一つであるため、CPU の個数が多ければ多いほど処理を同時に行える数が多くなり処理時間は早くなる。この CPU 数 (コア数) は一般的に 4,6,8 個ほどである。この CPU 数に対して、あるプログラムのプロセス数は数百に及ぶ。よって (CPU コア数)<<<(プロセス数) であることが大半である。以下に自分の計算機のプロセス数を示す。

プロセス制御において、CPU の割り当てを強制的に変更する方式をプリエンティブ方式という。これに対してプロセス自体が OS に制御を戻すか、プロセスが終了するまでほかのプロセスが実行されることがない方式をノンプリエンティブ方式という。プリエンティブ方式におけるプロセス切り替えをどのタイミングで行えば効率の良い処理となるかを考える。

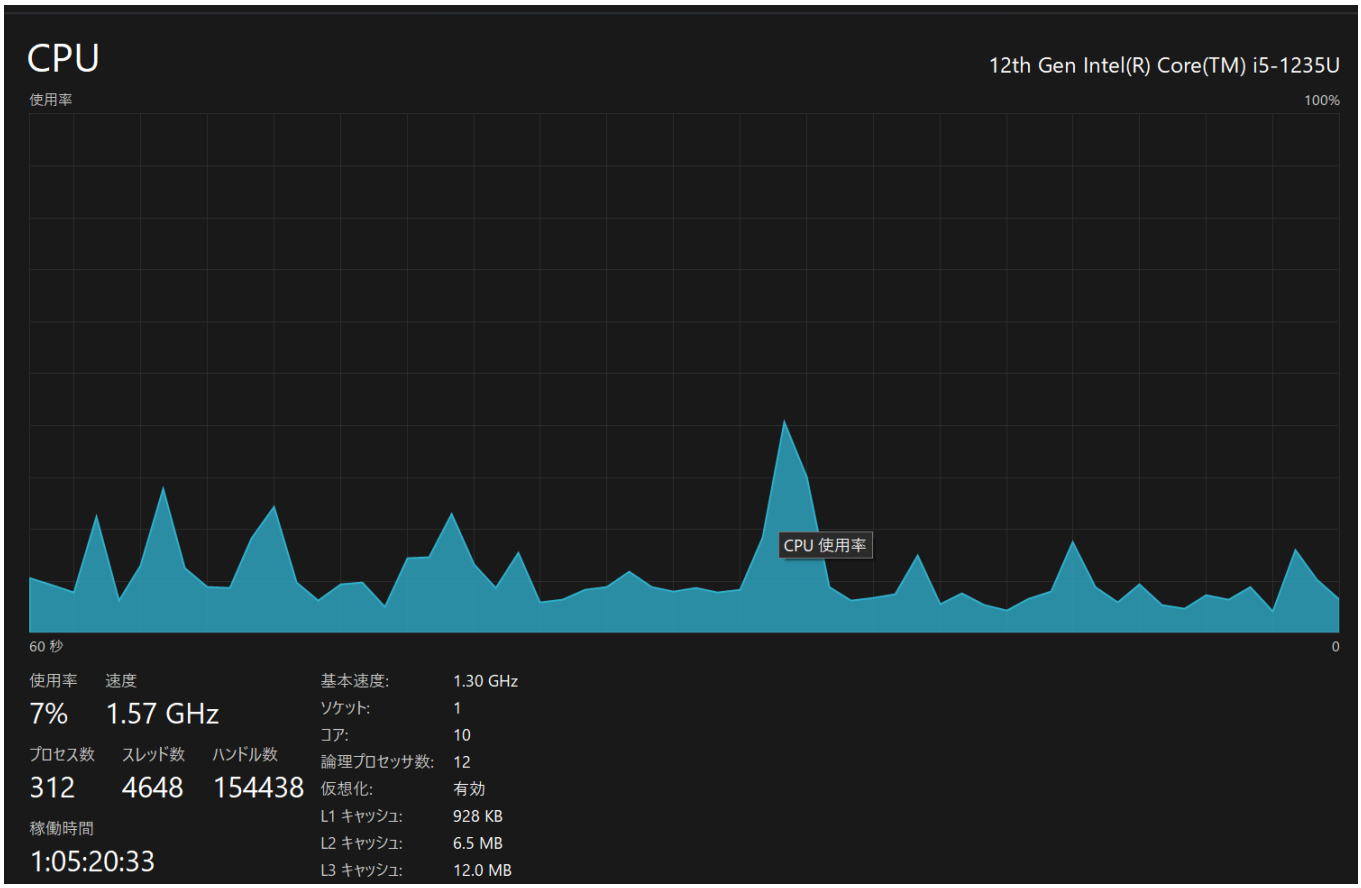


図 2: bash の実行例

### 2.2.2 プロセスの抽象化のアルゴリズム

プロセス切り替えを行う上でのきっかけとして考えられるのは以下である。

- マウスのクリックなどによるイベント発生時
- 一定時間による切り替え

これらを使いながら定められたプロセスのスケジューリング (どのタイミングでどのプロセスからどのプロセスへ移行するのか定めたスケジュール) 方式について説明する [2]。

### 2.2.3 到着順方式

名前の通り到着順でプロセスを割り当てるためこれはノンプリエンティブ方式にあたる。これを多段にして行われることもある。

### 2.2.4 処理時間順方式

名前の通り処理時間の短いタスクから行うスケジューリングである。

### 2.2.5 優先順位順方式

優先順位によってプロセスを割り当てる。優先順位の高いプロセスが実行待ち状態になるとプロセスの切り替えが発生する。この優先順位の割り当てにはいくつかのアルゴリズムが存在し、静的・動的どちらで割り当てることもある。

このアルゴリズムには低い優先順位のプロセスには一向に CPU の割り当てが行われない (スタベーション) という欠点がある。この欠点を補ったアルゴリズムとして待ち時間を考慮して優先順位をつける方式も存在する。

### 2.2.6 ラウンドロビン方式

ある一定時間 (タイムスライス) プロセスを実行し、すべて実行されなかった場合にはプロセス待ち行列の末尾に並びなおし、再び自分のプロセス割り当てがを待つ方式である。これの利点としては、実装が簡単であること、またどのプロセスにも平等に処理時間が与えられるためスタベーションのような現象が起こらない。しかしこれはプロセスの粒度 (各プロセスの規模) が同程度の時に優位に働く方式であるが、ある一つのプロセスが大規模であったときにそのプロセスには他よりも多くのタイムスライスを与えるべきである。そこでタイムスライスを各プロセスごとに何らかのアルゴリズムで割り当てるアルゴリズムも存在する。

例えば未終了プロセスに対して次のタイムプロセスを以下のように計算してタイムスライスを与える。

$$(\text{次のタイムスライス}) = (\text{現在のタイムスライス}) - (\text{実行時間}) + (\text{残り時間}) \times \frac{1}{2} + (nice\ \text{値・優先順位の数値}) \times (-1)$$

これにより応答性は向上するが、プロセス数が多い時にはこのタイムスライスを計算することのオーバーヘッドが生じ、逆に非効率となることもある。

## 2.3 メモリ

計算機は物理メモリの格納されているプログラムとデータを用いて実行を行う。物理メモリ容量には限界があり、我々が使用したいメモリ量よりも少ない場合に問題が生じる。そのようなときに仮想メモリによる抽象化によって見かけ上のメモリ量を増やすことでメモ容量の確保をしている。例えば容量が 10GB しかない物理メモリを持つ計算機があるとする。物理メモリには 4GB のプログラムと 7GB のプログラムを同時に配置することはできない。そのときに仮想メモリに 1GB 文のプログラムを退避することで順にプログラムを実行することを可能としている。仮想メモリへの退避方法や物理メモリと仮想メモリの紐づけについて説明を行っていく。

### 2.3.1 ページング方式

物理メモリと仮想メモリの紐づけ (マッピング) を行う方式の一つである。メモリをページ単位で分割し管理する。ただしページは可変長であり、状況に応じて容量を変えながら仮想メモリにプログラムを格納する。

物理メモリと仮想メモリのマッピングはページングテーブル一覧にその情報が格納されており、物理メモリと仮想メモリのデータの移動をしたいときに適宜マッピングテーブルを参照してアドレスを取得する。このとき MMU によってこのアドレス変換が行われる。アドレス変換を効率化するための工夫として MMU にもキャッシュ機能が存在する。

ページング方式にも問題点は存在する。それがページフォルトである。ページテーブルに対象となる仮想メ

モリのアドレス変換を参照し、その物理アドレスが正常に割り当てられていないときに発生する割り込みである。

### 2.3.2 スワッピング方式

物理メモリの容量が不足したときに、ハードディスクに不要なデータを退避 (スワップアウト) してメモリ容量を大きく見せること。

## 2.4 仮想メモリを使用する際の注意点

計算機の仮想メモリ容量は自分で設定可能である。しかしこの仮想メモリ量を大きくしても基本的に計算機のカクツキやフリーズは解消されない [5]。それはあくまでも物理メモリ内のプログラムが実行されるためである。また、仮想メモリの容量を大きくしすぎてもデータの読み込みに時間がかかり、かえって処理速度が落ちることも考えられる。

## 2.5 ファイルシステム

メインメモリ上のプログラムを実行する際にファイルを参照することがある。そのとき効率的に記憶媒体からファイルを探し出す必要がある。また、その記憶媒体の容量にも限界があるため、どのようにファイルを格納するかも問題である。これらのファイルシステムの性質について説明していく。

### 2.5.1 ファイル管理

多数のファイルをただ端から記憶媒体に並べたとき、ほしいファイルがどこにあるのか検索が困難である。検索の効率化やセキュリティ向上のためにディレクトリによるファイル管理、パスによるファイル検索ができる。ディレクトリとはファイルを格納しておくためのフォルダのことである [2]。これらによってファイル情報の修正も容易になっている。

### 2.5.2 VFS:Visual File System

ファイルを格納しておく記憶媒体製品は多数存在し、各記憶媒体によって仕様がかなり異なる。記憶媒体に合わせて計算機を選んでいては汎用性がない。そのため、どの記憶媒体でも様々なアプリケーションで使用できるように統一的なインターフェイスが存在し、それが VFS である。アプリケーション (ユーザ) が要求したファイルへの修正に対して、VFS は記憶媒体固有の表現に変換し、ファイル情報の修正を可能としている。

### 2.5.3 ファイルシステムの代表例

ファイルシステムは多種多様であるが、代表例が以下である。

- FAT(16,32)
- exFAT
- NTFS
- HFS+

自分のノートパソコンは WindowsOS であるので Windows で主に使用されるファイルシステムについて調

べた。

### 2.5.4 FAT と NTFS

FAT(File Allocation Table) と NTFS(New Technology File System) とは [6] どちらも Microsoft によって提供されただいるシステムである。以前には FAT が、今では NTFS が windows OS のデフォルトファイルシステムである。FAT のバージョン (12,16,32) が新しい規格になるほどクラスタ数が増加している。クラスタ数とはデータ領域を一定のサイズに区切った単位のことであり、クラスタ数が大きければ処理速度は速くなるが、ひとまとまりを大きくしているため容量の無駄遣いが増える。

NTFS は exFAT と同じ最大容量を持ちながら、暗号化の強化、ファイル単位での圧縮が可能となった。大規模なファイル情報の更新中に障害が発生したときに、ファイル情報に矛盾が起きてしまう可能性がある。この時にデータの不整合を起こさない機能としてジャーナリングファイルシステム [7] がある。NTFS にはこのジャーナリングファイルシステムが機能としてあるため、広く使用されているファイルシステムである。ジャーナリングファイルシステムは NTFS のほかにも HFS+ などに搭載されている。

# 3 課題 3

## 3.1 課題内容

**メモリのスタック領域とヒープ領域について解説し，それぞれの特徴を生かした使用用途を例示せよ**

## 3.2 スタック領域

### 3.2.1 スタック領域とは

スタック領域 [9] とはメモリ領域を確保した逆順でメモリの解法 (LIFO:Last In First Out) を行うことである。アルゴリズムで学習したスタックと同じ手法でメモリの確保・解法を動的に行う。

### 3.2.2 スタック領域の利点・欠点

コンパイル時点で自動的にサイズが確定するメモリ領域のため自分でメモリ領域の確保・解法を気にしなくてよい利点がある。一方で不必要なメモリ領域までメモリを消費し、効率的なメモリ管理をしたいときにはスタック領域は不適切である。

### 3.2.3 スタック領域の使用用途

基本的にスタック領域を使用できるときはスタック領域を使用する方が好ましい。理由としてはスタック領域の方がメモリ管理が容易、コードの簡潔化、実行時間の短縮などの観点である。

具体的には関数の呼び出し、再帰などがある [11]。プログラムに作成するにあたってある関数を呼び出すことは頻発する。そのときスタックの LIFO の構造が、関数を呼び出して実行したのちに、元の関数に戻るアルゴリズムが適するためである。また、C 言語などの自動変数をスタックに確保するときにも使用される [12]。

## 3.3 ヒープ領域について

### 3.3.1 ヒープ領域とは

ヒープ領域 [9] とはスタック領域とは異なり順番に依存せず、メモリの確保・開放はソフトウェア側が行う。

### 3.3.2 ヒープ領域の利点・欠点

プログラムが実行時に動的にメモリ管理を行うため、プログラムにメモリ管理について明記しないといけない欠点がある [10]。また、確保したメモリを開放しないとメモリリークとなったり、メモリの確保・解法を繰り返すとフラグメンテーション (メモリの断片化) が発生する。それにより使用可能なメモリ量が減少してしまう。一方でコンパイル時にメモリ確保量が決まっていないためメモリ確保量を変えることもでき、柔軟性が高い。

### 3.3.3 ヒープ領域の使用用途

ヒープ領域が使用される用途として、大規模なデータ構造がプログラムの途中で変化するときに使用される。データ構造が変化するとき、必要なメモリ量が変化する。その時に明示的に領域の確保・解法を行うことで効率的にメモリ管理を行うためである。C 言語の malloc 関数や C++ の new 演算子でメモリを確保することにも使用される [12]。

# 4　課題 4

## 4.1　課題内容

**1．2 の補数を説明せよ．**
**2．計算機上の整数値（符号あり）において，2 の補数を用いて負の数を表現する理由を説明せよ．**

## 4.2　補数について

補数 [1] とはある基数法において、ある自然数 a に足したとき桁が 1 つ上がる（桁が 1 つ増える）数のうち最も小さい数のこと。2 進数を採用している計算機では”１の補数”と”２の補数”が存在する。その中で 2 の補数はあるビット列”10101010”をビット反転 (0→1,1→0) を行った数に 1 を足すことで求められる。例えば、符号なしエンコード (後に説明) において $(00011000)_2 = (24)_{10}$ の 2 の補数は $(11101000)_2 = -(24)_{10}$ と表せる。

## 4.3　計算機における数の表現方法

人は日常で 10 進数を使用し、あらゆる数値を 10 進数で表現するのに対し、先述した通り計算機では 2 進数で数値、文字列などを表現している。計算機では-10 といった数も 0,1 の 2 値で表す必要がある。このとき負の数をどのように表現するかが問題となる。そこで以下の負の表現方法が存在する。

## 4.4　符号ありエンコーディング

まず初めに符号なしエンコーディングとはビット列 1 バイト単位で 10 進変換を行った数値のこと。例えば $(11111111)_2 = (255)_{10}$ と変換できる。

これに対して、符号ありエンコーディングとは、１バイトのビット列を変換する際に符号が正負のどちらであるか考慮しながら変換すること。先頭１ビットを符号を表すビットとし、0 なら正、1 なら負として変換を行う。例えば、$(11111111)_2 = -(1)_{10}$ と変換できる。ここで直感的には先頭ビットで符号の判別、２ビット目以降で数値を表すのなら $(11111111)_2 = -(127)_{10}$ と変換するのが直感的である。しかしこの変換は計算機の演算の上で不都合が生じる。下記でその説明を行う。

## 4.5　2 の補数を用いて負の数を表現する理由

符号あり整数値において先頭ビットで符号を、２ビット目以降で数値を表現する。$(111)_2 = -(3)_{10}$ としてしまうと以下の演算に矛盾が生じてしまう。

$$3 + (-3) = 0$$

$$(011)_2 + (111)_2 = (110)_2 \neq (0)_2$$

ではこの演算に矛盾 (0 が二通りで表される) が生じないようにするにはどうしたらよいのか？まずは 1 の補数 (ビット反転) について説明する。

任意の n 桁のビット列が存在したとする。この 1 の補数、すなわちビット反転したものと元の数値を足し算すると以下のように全ビット 1 の数値となる。

$$\underbrace{1001010100,,,}_{n\text{ 桁}} \quad + \quad \underbrace{0110101011,,,}_{n\text{ 桁}} \quad = \quad \underbrace{1111111111,,,}_{n\text{ 桁}}$$

次に 2 の補数 ( 1 の補数に 1 を足したもの) を考える。2 の補数とは上記の計算式に 1 を足せばよいので以下の式が成り立つ。

$$\underbrace{1001010100,,,}_{n\text{ 桁}} \quad + \quad \underbrace{0110101011,,,}_{n\text{ 桁}} \quad +1 \quad = \quad \underbrace{1000000000,,,}_{n+1\text{ 桁}}$$

1 を足したことである数値にその 2 の補数を加算した値は、n+1 桁の 1000000000,,, となる。ここで初めの 1 ビットを除けば、普段自分たちが 10 進数で使用する $(0)_10$ と同じような理想の数値 0000000000,,, が登場している。このため、1 バイト単位でビット列を認識するのならば繰り上がった数値は考慮せずに計算を行えば (任意の数値)+(ある数値の 2 の補数)=$(00000000)_2$ が成り立つ。

1 の補数では 11111111,00000000 の 2 通り 0 を表す数値が存在してしまうのに対し、2 の補数は 0 が一意に定まる。また減算を行う際に 2 の補数の方が計算が容易であることから 2 の補数を使用したほうが一般的に使用されている [3]。負の数の加算を可能にするということは正の数と正の数の減算など加減算を可能にすることを意味し、2 の補数のおかげで減算が実現している。

# 5　参考文献

# 参考文献


[1] 応用情報技術者合格教本 (令和 05 年), 大滝みや子・岡嶋裕史

[2] コンパイル型言語 - Wikipedia
https://ja.wikipedia.org/wiki/%E3%82%B3%E3%83%B3%E3%83%91%E3%82%A4%E3%83%AB%E5%9E%8B%E8%A8%80%E8%AA%9E

[3] コンパイラは何をしているのか
http://nenya.cis.ibaraki.ac.jp/TIPS/compiler.html

[4] シェルスクリプト - wikipedia
https://ja.wikipedia.org/wiki/%E3%82%B7%E3%82%A7%E3%83%AB%E3%82%B9%E3%82%AF%E3%83%AA%E3%83%97%E3%83%88

[5] 仮想メモリとは？ 設定方法まで詳しくご説明します！
https://www.dospara.co.jp/5info/cts_str_pc_vmemory.html

[6] FAT32、NTFS、exFAT の違いとファイルシステムの変換
https://www.partitionwizard.jp/partitionmagic/differences-between-fat32-exfat-and-ntfs.html

[7] ジャーナリングファイルシステム - wikipedia
https://ja.wikipedia.org/wiki/%E3%82%B8%E3%83%A3%E3%83%BC%E3%83%8A%E3%83%AA%E3%83%B3%E3%82%B0%E3%83%95%E3%82%A1%E3%82%A4%E3%83%AB%E3%82%B7%E3%82%B9%E3%83%86%E3%83%A0

[8] 電子計算機コンピューター内部の負の数の表現
http://www.yamamo10.jp/yamamoto/lecture/2003/2E/7th/index.php
[9] ヒープ領域とは？スタック領域との違いや具体的な管理方法
https://it-trend.jp/development_tools/article/32-0041
[10] ヒープとスタックの違いを解説 - プログライフ
https://proglife.net/heap-stack/
[11] メモリとスタックとヒープとプログラミング言語
https://keens.github.io/blog/2017/04/30/memoritosutakkutohi_puto/
[12] メモリの 4 領域
https://brain.cc.kogakuin.ac.jp/~kanamaru/lecture/MP/final/part06/node8.html