情報処理 (計算機) システム II・第 7 回

2024 年 5 月 31 日

今回の内容

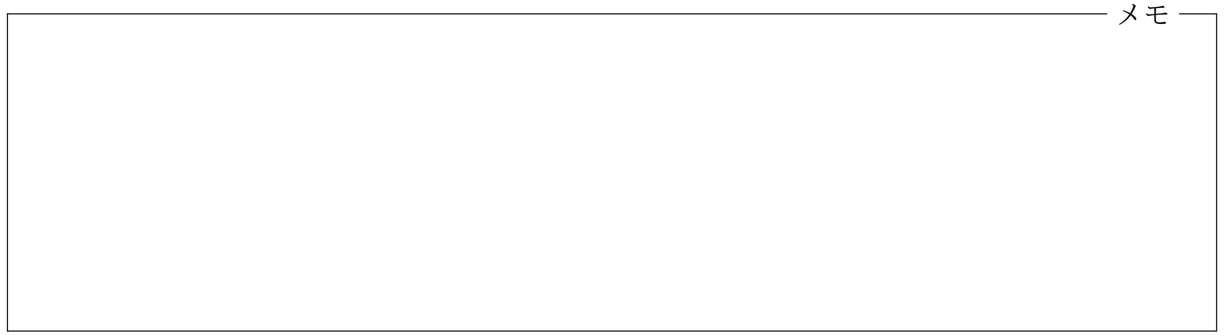
7.1 サブルーチンの呼び出しとスタック領域 7–1
7.2 ユーザプロセスへのメモリ割り当て 7–5

7.1 サブルーチンの呼び出しとスタック領域

C 言語における関数呼び出しの場合など、機械語プログラムがサブルーチンを呼び出す際には、呼び出されるサブルーチンプログラムの先頭に分岐するだけではなく、サブルーチンの実行終了後に呼び出した側のプログラムに戻って来て (これも分岐の一種) 呼び出した側の機械語プログラムを続行することが必要となります。どのアドレスに戻ってくるかは、どこからそのサブルーチンが呼び出されたかによって変わってきますので、固定したアドレスを分岐先に指定して呼び出し元に戻ることはできません。

そこで、サブルーチンを呼び出す側のプログラムは、戻り先となるアドレスを、呼び出されるサブルーチンの側に何らかの方法で伝えてやる必要があります。戻り先のアドレスは、サブルーチンを呼び出す機械語命令の次の機械語命令が置かれたアドレスとなります。この戻り先となるアドレスのことを**リターンアドレス**と呼びます。

リターンアドレスを伝える最も単純な方法は、それを特定のレジスタに格納しておくことです。呼び出されたサブルーチンの側では、この特定のレジスタに記憶されているアドレスに分岐して呼び出し元へ戻ることができます。ただし、この方法の難点は、呼ばれた側のサブルーチンが、また別のサブルーチンを (場合によっては自分自身を再帰的に) 呼び出す場合です。この呼び出しで、また、その特定のレジスタを使用しなければなりませんから、そこに格納されている (そのサブルーチン自身の) リターンアドレスを、どこか別の場所に退避しておかなければならなくなります。



スタック領域とスタックポインタ

リターンアドレスに限らず、サブルーチンが呼び出されると、そのサブルーチンが使用するメモリ領域[1]が新たに必要となるのが普通です。何重にもサブルーチンが呼び出される場合を考えると、この領域はサブルーチンが呼び出される度に増加していき、サブルーチンからその呼び出し元に戻る度に減少し元の状態に戻っていくことになります。

[1] C 言語の関数定義内で宣言された自動変数など。

通常、(仮想) アドレス空間の一部のメモリ領域を、このような用途に割り当てて、その端から[2]少しずつ使用していきます。この領域は**スタック領域**[3] (stack area) と呼ばれ、その領域の内、(その時) 使用されている部分を**スタック** (stack) と呼びます。多くの CPU は、スタック領域の内、その端からどこまでを使用しているのかを特定のレジスタに記憶するようにしています。このレジスタは一般に**スタックポインタ** (stack pointer) と呼ばれます。

スタック領域を使用する方法には2通りの方式があり、その1つは、スタック領域の先頭からアドレスの大きい方へ使用していく方式です。この場合、スタックポインタは、通常、使用している範囲の終り (使用中の最後のアドレスの次) のアドレスを記憶します。もう1つは、スタック領域の末尾からアドレスの小さい方へ使用していくもので、スタックポインタは、通常、使用している範囲の先頭のアドレスを記憶します[4]。どちらの場合でも、割り当てを開始した側の端をスタックの**底** (bottom) とよび、最後に割り当てた側 (スタックが伸びていく方向) の端を**トップ** (top) と呼びます。

プッシュ命令とポップ命令

CPU には、スタックに対する基本的な操作である**プッシュ**と**ポップ**という2つの操作を行う機械語命令が用意されているのが普通です。それぞれ、プッシュ (push) 命令とポップ (pop) 命令と呼ばれます。これらは 通常、それぞれ1つのオペランド[5]をとり、次のような操作を行います。

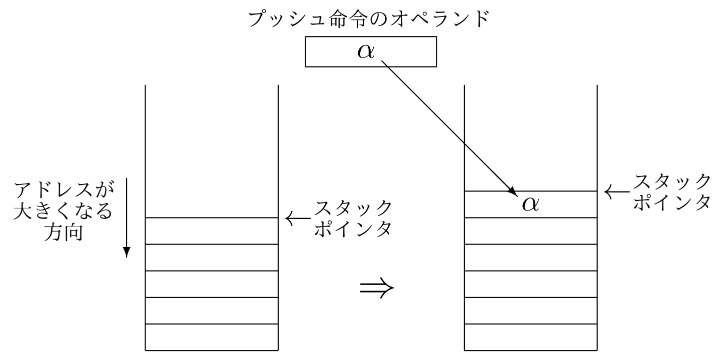
プッシュ命令 オペランドのデータ大きさ (バイト数) だけスタックを拡大してできる新しい領域に、オペランドのデータをコピーします。このデータがスタックの新しいトップとなります。アドレスが大きくなる方向へスタックが伸びる方式では、スタックポインタの現在の値をメモリアドレスとしてオペランドのデータを書き込み、その後、スタックポインタの値をそのデータの大きさだけ増やします。また、アドレスが小さくなる方向へスタックが伸びる方式の場合は、書き込みの前にスタックポインタの値を減らし、その新しいスタックポインタの値をデータの格納先のアドレスとして使用します。

[2]アドレスの小さい方の端 (領域の先頭) から大きい方へ使用していく場合と、アドレスの大きい方の端 (領域の末尾) から小さい方へ使用いしていく場合の2通りがあります。Intel の IA-32 や Intel 64 アーキテクチャでは後者が採用されています。

[3]セグメント方式のアドレス変換機構を使用する場合は、通常、独立したセグメントとします。

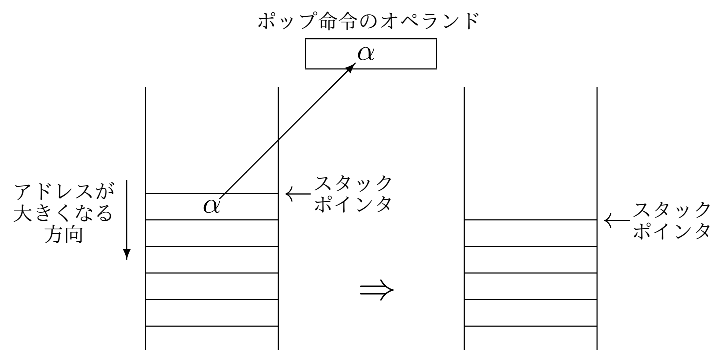
[4]こちらの方が主流です。

[5]CPU の機械語命令の操作の対象を、その機械語命令の **オペランド (operand)** と呼びます。



プッシュ命令の効果
(アドレスが小さくなる向きに伸びるスタックの例)

ポップ命令 スタックのトップに位置するデータをオペランドへコピーするとともに、その大きさだけスタックを縮めます。アドレスが大きくなる方向へスタックが伸びる方式では、コピーする前に、まず、オペランドのデータの大きさだけスタックポインタの値を減らし、その新しいスタックポインタの値をメモリアドレスとして使用します。また、アドレスが小さくなる方向へスタックが伸びる方式の場合は、スタックポインタの現在の値をメモリアドレスとしてコピーを行い、その後で、スタックポインタの値を増やします。



ポップ命令の効果
(アドレスが小さくなる向きに伸びるスタックの例)

レジスタの値のスタックへの退避

機械語プログラムがサブルーチンを呼び出す場合、特定のレジスタにリターンアドレスを格納する方法を紹介しましたが、呼ばれたサブルーチンが、さらにサブルーチンを呼び出す場合は、この特定のレジスタの値を退避する必要があることを指摘しました。このような際の、レジスタに置かれた値の退避先としてスタックが用いられます。

まず、そのレジスタの値をスタックにプッシュし退避しておきます。そして、同じレジスタに新しいリターンアドレスを書き込んでサブルーチンに分岐します。そのサブルーチンから戻って来たところで、退避しておいた値をスタックから同じレジスタにポップすれば、そのレジスタの元の値 (このサブルーチン自身のリターンアドレス) を復元することができます。

CPU には限られた数のレジスタしか内蔵されていませんから、計算の途中結果などを格納するために必要な一時的な記憶領域が不足しがちになります。また、複雑な計算の途中で、その途中結果を記憶したままサブルーチンを呼び出さなければならない場合もありますので、そうなると、せっかくレジスタに記憶していた途中結果を、呼び出したサブルーチンが壊してしまうことになりかねません。スタックへのデータのプッシュ (退避) やポップ (復元) は、機械語プログラムにおける非常に基本的な道具となっています。

メモ

呼び出し命令と復帰命令

サブルーチンの呼び出しの際のリターンアドレスを特定のレジスタに格納してしまうと、サブルーチンの呼び出しの度に、そのレジスタの値のプッシュやポップが必要となってしまいますので、スタックを直接使ってリターンアドレスをサブルーチンに伝える方式が採用されることがよくあります[6]。スタックを直接利用してリターンアドレスを伝える方式の場合、サブルーチンを呼び出す側では、呼び出したいサブルーチンに分岐する前に、リターンアドレスをスタックへプッシュしておきます。呼び出されたサブルーチンの側では、呼び出し元へ戻る際にスタックからリターンアドレスをポップし、そのアドレスへ分岐します。多くの CPU には、この 2 種類の作業を行うことのできる特別な分岐命令がそれぞれ用意されています。

呼び出し命令 call 命令とも呼ばれる命令で、スタックを使ってサブルーチンを呼び出す際に必要な 2 つの操作、つまり、リターンアドレスのプッシュと、指定したアドレスへの分岐をまとめて行います。プッシュされるリターンアドレスは、この呼び出し命令の次の命令が置かれたアドレスとなります。

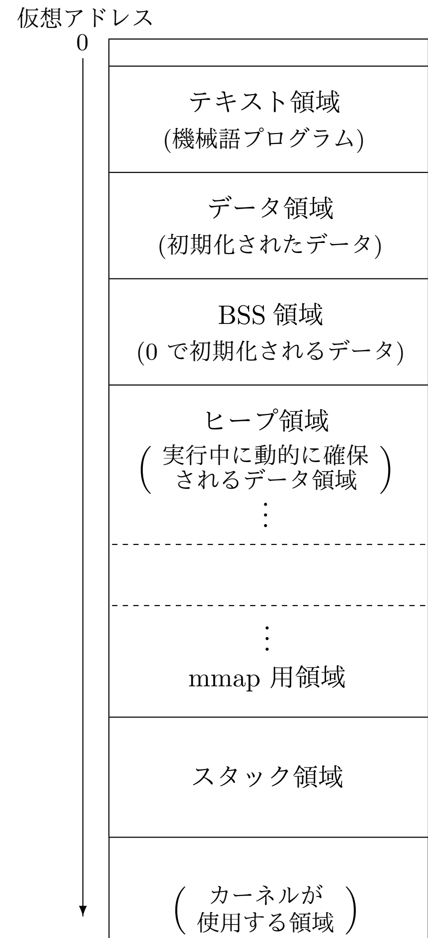
復帰命令 return 命令とも呼ばれる命令で、サブルーチンから呼び出し元へ戻る際に必要な 2 つの操作、つまり、リターンアドレスのポップと、ポップして得られたアドレスへの分岐をまとめて行います[7]。

[6]スタックを直接使ってリターンアドレスをサブルーチンに伝える方式でも、サブルーチンを全く呼び出すことのないサブルーチンを呼び出す際に限って、レジスタにリターンアドレスを格納する方法が使われる場合があります。

[7]return 命令は、プログラムカウンタをデスティネーションオペランドとしたポップ命令と考えることができます。

7.2 ユーザプロセスへのメモリ割り当て

ユーザプロセスが起動した際には、その実行に必要なメモリ空間がカーネルによって割り当てられます。たとえば、Linux などの Unix 系 OS のユーザプロセスのアドレス空間は右の図のようになっています[8]。



テキスト領域　機械語プログラムや、そのプログラムが使用する読み取り専用のデータ[9]が置かれます。テキストセグメント (text segment) やテキストセクション (text section) と呼ばれることもあります。この領域は読み取り専用で、プログラムの実行中に書き変わることはありません。同じプログラムが複数起動される場合は、その複数のプロセス間で同じ物理ページが共有されます[10]。

通常、プログラムの起動時には、あたかもテキスト領域の仮想ページがプログラムファイル[11]中にページアウトしているかのような状態でプロセスが生成されます。CPU がテキスト領域の仮想ページにアクセスするときにページフォルトが発生し、そのときのページインの処理としてプログラムファイル中に記録されている機械語プログラム等が物理ページにコピーされます。次に同じプログラムを起動した際には、その物理ページのデータが失われていないのであれば、すでに存在している物理ページをそのまま利用します。

データ領域　プログラムで使用される初期値を持つデータのための領域です。C プログラムの静的変数[12]が 0 以外の初期値で初期化される場合、この領域に割り当てられます。データセグメント (data segment) やデータセクション (data section) と呼ばれることもあります。この領域は読み書きがともに可能で、同じプログラムが複数起動されている場合でも、独立した物理ページが使用されます。

---

[8] OS によっては mmap 用領域がスタック領域の後に取られる場合もあります。

[9] たとえば、C プログラムの文字列リテラル (文字列定数) など。

[10] Unix 系 OS で採用されているデマンドページング方式の仮想記憶システムの働きです。

[11] 第 4 回の配布資料参照

[12] 大域変数 (関数定義の外で宣言される変数や配列) や、`static` というキーワード付きで宣言される局所変数 (関数定義やブロックの内部で宣言される変数や配列) のことです。

テキスト領域と同様に、あたかもデータ領域の仮想ページがプログラムファイル中にページアウトしているかのような状態でプロセスが生成され、ページフォルトが発生してはじめて、プログラムファイル中のデータが物理ページにコピーされますが、このページの内容が一旦書き換えられると、プログラムファイルではなく、補助記憶装置内の独立した領域[13]にページアウトします。

BSS領域　0 で初期化されるデータのための領域です。BSS セグメント (bss segment) や BSS セクション (bss section) と呼ばれることもあります[14]。C 言語の静的変数が 0 で初期化されている場合[15]、この領域に割り当てられます。この領域の初期値は 0 と決まっていますので、プログラムファイル中のデータを物理ページにコピーする必要はありませんが、このページの内容が一旦書き換えられると、補助記憶装置内の独立した領域にページアウトする必要があります。

ヒープ領域　プログラムの実行中に、必要に応じてメモリが割り当てられる領域です。Unix 系 OS では、`sbrk` や `brk` と言ったシステムコールにより、この領域の終りを変更することができます。たとえば、C 言語の標準ライブラリに含まれる `malloc` や `calloc` などの関数は、動的に (プログラムの実行中に) メモリを割り当てることができますが、これらの関数は、このヒープ領域を拡張することで必要な領域を確保し、そこからメモリの割り当てを行うことができます[16]。

[13] 前回解説したページング方式の仮想記憶システムにおけるスワップ領域やスワップファイルの一部です。

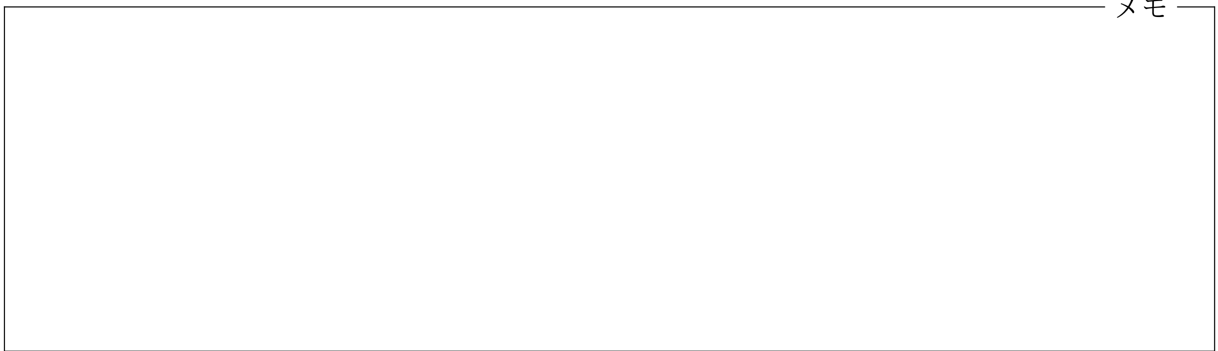
[14] BSS は Block Started by Symbols の頭文字で、あるアセンブリ言語において、特定のメモリ領域を名前を付けて予約するための疑似命令 (アセンブラーへの指示) の名前に由来します。

[15] C 言語では、初期値を特に指定せずに静的変数を宣言すると、その変数 (や配列) は 0 で初期化される決まりとなっています。

[16] これらの関数は、ヒープ領域を使用せずに、(特に大きな領域が必要な場合に) 後述の mmap 領域を使って動的なメモリ割り当てを行う場合もあります。

mmap 用領域　Unix 系 OS には、その仮想記憶システムの一部として、ファイルの内容の一部を仮想アドレス空間の一部に対応させて、ユーザプロセスがメモリの読み書きを行うことで、ファイルの内容の読み書きを可能にする機能が用意されています。ファイルの内容と仮想アドレス空間の一部との対応は、`mmap` というシステムコールを使って作ることができます。ヒープ領域と後述のスタック領域の間の仮想アドレス空間は、この `mmap` システムコールを利用して、ファイルの一部を仮想アドレス空間に対応付けするのに使用されます。

実行中のプログラムが使用しているライブラリの機械語プログラムや、そのライブラリプログラムが使用するデータもこの領域に置かれます。ユーザプログラムがライブラリを使用する場合は、この `mmap` システムコールで、ライブラリファイルの一部を仮想ページに対応させることで、ライブラリの機械語プログラムをユーザプロセスのアドレス空間に取り込みます。また、`mmap` は名前のない新しいファイル[17]を仮想アドレス空間に対応させることで、読み書き可能な新しいメモリ領域を確保することができますので、ライブラリ関数が使用するデータ領域はこの機能を利用して確保されます。



スタック領域　実行中のプログラムがサブルーチン[18]の呼び出しを行う際のリターンアドレス[19]や、呼び出されたサブルーチンなどが一時的に使用する記憶領域が置かれます。C プログラムの自動変数[20]は、このスタック領域に割り当てられます[21]。プロセスの起動時には、スタック領域はほとんど使用されていませんが、サブルーチンの呼び出しの入れ子が深くなるにつれて、必要な記憶領域が大きくなっていきます。また、サブルーチンから呼び出し元へ戻る (リターンする) ごとに必要な記憶領域の大きさが元に戻っていきます。

スタック領域として確保される大きさは、そのプロセスを起動した際に決定されますので、サブルーチン呼び出しの入れ子が深くなり続けると、いずれはスタック領域の範囲を外れてしまい、メモリアクセスに関する例外が発生してしまいます[22]。

情報処理 (計算機) システム II ・第 7 回・終わり

[17]実際にはスワップ領域やスワップファイルの一部です。

[18]C 言語における関数など。

[19]サブルーチンの実行の終了後に戻って来るべきアドレス

[20]`static` というキーワードを付けずに宣言された局所変数など。

[21]ライブラリ関数 `alloca` が割り当てるメモリも、自動変数と同様にこのスタック領域に割り当てられます。

[22]Unix 系 OS では、この例外はセグメンテーション違反 (Segmentation Violation または Segmentation Fault) と呼ばれ、**シグナル**と呼ばれる仕組みでカーネルからユーザプロセスに伝達されます。

付録: Unix 系 OS における C プログラムの変数のメモリ割り当て

```
int a;                                              // BSS 領域
int b = 123;                                        // データ領域
int c[10];                                          // BSS 領域
int d[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };      // データ領域
static int e;                                       // BSS 領域
static int f = 123;                                 // データ領域

void
foo(int x)                                          // スタック領域
{
    int y;                                          // スタック領域
    int z = 123;                                    // スタック領域
    static int p;                                   // BSS 領域
    static int q = 123;                             // データ領域

    ⋮
}

int
main(int argc, char *argv[])                        // スタック領域
{
    static int s = 123;                             // データ領域
    static int t;                                   // BSS 領域
    int i;                                          // スタック領域
    int j = 456;                                    // スタック領域
    int k[10] = { 1, 2, 3 };                        // スタック領域

    ⋮
}
```