計算機システムⅡ・中野・水1講時

2020年1月29日

I 次の2つのプログラムは、あるCプログラムと、それをIntel 64アーキテクチャの64 bitモードの機械語プログラムに変換した結果である。ただし、関数 `main` に対応する部分の機械語プログラムは `0x400b5c` 番地から始まっている。また、関数呼び出しの際には、第1引数の値はレジスタ `rdi` に、第2引数の値はレジスタ `rsi` に格納し、戻り値はレジスタ `rax` に格納した状態で呼び出し元に戻るようにしている。この機械語プログラムを(関数 `main` を呼び出して)実行するものとして、以下の問いに答えなさい。(28点)

```
int *p;
int x;

void foo(int *q, int a) {
    if (*q [    ] a) {
        *q = a;
    }
}

int bar(int n) {
    return n + [    ];
}

int main() {
    x = 55;
    p = &x;
    foo(p, bar(50));
    return x;
}
```

```
0x400b4d:  movl   (%rdi), %eax
0x400b4f:  cmpl   %eax, %esi
0x400b51:  jg     0x400b55
0x400b53:  movl   %esi, (%rdi)
0x400b55:  retq
0x400b56:  movl   %edi, %eax
0x400b58:  addl   $0xa, %eax
0x400b5b:  retq
0x400b5c:  movl   $0x37, 0x6bc3a8
0x400b67:  movq   $0x6bc3a8, 0x6bc3a0
0x400b73:  movq   0x6bc3a0, %rbx
0x400b7b:  movl   $0x32, %edi
0x400b80:  callq  0x400b56
0x400b85:  movq   %rbx, %rdi
0x400b88:  movl   %eax, %esi
0x400b8a:  callq  0x400b4d
0x400b8f:  movl   0x6bc3a8, %eax
0x400b96:  retq
```

(1) 2つのプログラムが対応するように空欄を埋めなさい。

(2) 変数 `p` の値を記憶しているメモリアドレスの範囲を16進数表記で答えなさい。

(3) `0x400b4f` 番地の `cmpl` 命令が実行されるときのレジスタ `eax` の値を16進数表記で答えなさい。

(4) `0x400b51` 番地の `jg` 命令の次に実行される機械語命令のアドレスを16進数表記で答えなさい。

(5) `0x400b80` 番地の `callq` 命令が実行される際に、スタックにプッシュされる値(リターンアドレス)を16進数表記で答えなさい。

(6) `0x400b8a` 番地の `callq` 命令が実行されるときのレジスタ `rbx` の値を16進数表記で答えなさい。

学籍番号　　　　　　　　　　　　　　氏名　　　　　　　　　　　　　　　　　　**(裏面に続く)**

この期末試験の採点結果とこの科目の総合成績は、来週末までに、Emailで `txxxxxx@mail.ryukoku.ac.jp` 宛にお送りします。この連絡が不要な人は右の「連絡不要」を丸で囲んでください。

連絡不要

II 情報実習室の Linux 環境で、下の C プログラムの空欄部分を右の A、B、C のいずれかで埋めて関数 foo を定義したプログラムを 3 通り作成した。関数 foo を呼び出して、その実行に必要となる時間を測定したところ、実行時間が最も短いものから順に、0.14 秒、0.20 秒、6.7 秒という結果となった。A、B、C それぞれは、どの実行時間に対応していると考えられるか、理由とともに 200 〜 300 字程度で説明しなさい。(21 点)

A
```
sum += a[i][i];
```

B
```
sum += a[j][j];
```

C
```
sum += a[i][j];
```

```
#define N (20000)

int a[N][N];

int foo(void)
{
    int i, j;
    int sum = 0;

    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            [          ]
        }
    }
    return sum;
}
```

(次ページに続く)

III 情報実習室の Linux 環境にログインし、端末エミュレータ (新しい端末) を開き、下図右のような C プログラム bar.c を、下図左のようにコンパイルして実行した。下の候補の中から最も適当と思われる語句を選んで空欄を埋めなさい。ただし、下図右の C プログラム中の行頭の数字は行番号であり、このプログラムの一部ではない。(30 点)

```
t180000@s01cd0542-160:~$ cc -o bar bar.c
t180000@s01cd0542-160:~$ ./bar
データ? 23
データ? 12
データ? 43
データ? 67
データ? 42
データ? -1
最大値 = 67
t180000@s01cd0542-160:~$
```

bar.c

```
 1 #include <stdio.h>
 2
 3 int main() {
 4     int x;
 5     int max = -1;
 6
 7     while (1) {
 8         printf("データ? ");
 9         scanf("%d", &x);
10         if (x < 0)
11             break;
12         if (x > max)
13             max = x;
14     }
15     if (max >= 0)
16         printf("最大値 = %d\n", max);
17     return 0;
18 }
```

(1) [　　　　] はキーボードから「./bar」という文字列を読み取ると、[　　　　] (あるいはそれに類する) システムコールを利用して自らのプロセスを複製して新しいプロセスを生成し、その仮想アドレス空間に execve を使って bar の機械語プログラムなどを取り込み、その実行を開始する。

(2) この「./bar」は相対パス名と呼ばれる書き方である。カレントディレクトリの絶対パス名が /home/t190000 である場合、同じファイルを [　　　　] という絶対パス名で指し示すことができる。

(3) bar のプロセスが起動された後、C プログラムの 9 行目で呼び出されている関数 scanf は、[　　　　] というシステムコールを使って、標準入力 (ファイル記述子 [　　] 番) から入力された文字列を読み取り、十進表記の整数値と解釈して、引数 &x で指定されたアドレスにその整数値を書き込む。このアドレスは、OS (カーネル) のメモリ管理機能によってプロセスごとに用意された [　　　　] アドレス空間におけるアドレスである。

(5) bar.c の 4 〜 5 行目に宣言されている変数 x や max は、bar のプロセスのメモリ空間内の [　　　　] 領域と呼ばれる部分に割り当てられる。

(6) Linux などで採用されている [　　　　] マルチタスク方式の OS では、ユーザープロセスが CPU の [　　　　] モードで実行されている状態でも、[　　　　] からの割り込み信号を受けると、CPU が [　　　　] モードに移行し、カーネルがあらかじめ設定しておいたプログラムが実行されることでプロセスのスケジューリングの処理を行うことができる。

空欄の候補

bar、../bar、/home/bar、/home/./bar、/home/t190000/bar、/home/t190000/bar.c、0、1、2、3、4、BSS、close、execve、fork、main、open、printf、read、scanf、write、アイドル、アドレス、イベント、カーネル、カレント、キーボード、キャッシュ、シェル、スタック、タイマー、テキスト、データ、ディレクトリ、パイプライン、ヒープ、ファイル、ブートストラップ、プリエンプティブ、プロセス、メモリ、ルート、レジスタ、親、階層的、仮想、協調的、子、実行可能、数理、絶対、相対、特権、非特権、物理

学籍番号　　　　　　氏名

(裏面に続く)

IV　デマンドページング方式の仮想記憶システムを持ったオペレーティングシステムが稼働しており、あるプロセスが実行されている。このプロセスが使用できる (主記憶装置の) 物理ページは常に 6 ページであり、ページフォルトが起こると、(必要なら) 最後にアクセスされた時刻が最も遠い過去であるような物理ページを (補助記憶装置に) ページアウトし、(必要なら) アクセスしたい仮想ページの内容を (補助記憶装置から) その物理ページにページインする。(21 点)

(1) このプロセスがある時点までにアクセスした仮想ページの番号をアクセスした順に並べると次のようになった。

$$5, 0, 3, 3, 8, 1, 8, 5, 6, 4, 3, 3, 7, 9, 2, 1, 4, 4, 6, 2, 8, 8, 6, 4, 7, 4, 3, 8, 4, 4$$

この時点で (6 つの) 物理ページに割り当てられている仮想ページの番号をすべて挙げなさい。

(2) (1) に続いて、このプロセスが

$$5, 3, 2, 5, 1, 0, 7, 2$$

の順に、ここに挙げた番号の仮想ページにアクセスするとする。これら 8 回のアクセスでは、いつページフォルトが起こると考えられるか。ページフォルトが起こる仮想ページの番号を、ページフォルトが起こる順に列挙しなさい。

(3) このプロセスが、0 番から 19 番までの 20 個の仮想ページの内の 1 つをランダムに選んでアクセスすることを繰り返すとすると、1 回のアクセスでページフォルトが発生する確率はどの程度と考えられるか。

(定期試験問題終り)

参考資料: Intel 64 アーキテクチャ 64 bit モードの主要な命令群

各命令のアセンブリ言語 (情報実習室の Linux 環境) での書式

主要なデータ転送命令

命令	説明
mov α, β	α の値を β へコピーする
xchg α, β	α と β の値を交換する
lea α, β	メモリオペランド α のアドレスそのものを β へ書き込む
push α	rsp をオペランドの大きさ (バイト数) だけ減じて、そのアドレスに α をコピーする
pop α	アドレス rsp のデータを α にコピーし、その大きさ (バイト数) だけ rsp を増やす

主要な整数演算命令

命令	説明
add α, β	整数値 α、β について、$\beta+\alpha$ の計算結果を β へ書き込む
sub α, β	整数値 α、β について、$\beta-\alpha$ の計算結果を β へ書き込む
addc α, β	整数値 α、β について、$\beta+\alpha+\mathtt{CF}$ の計算結果を β へ書き込む
subb α, β	整数値 α、β について、$\beta-(\alpha+\mathtt{CF})$ の計算結果を β へ書き込む
cmp α, β	整数値 α、β について、$\beta-\alpha$ の計算結果に合わせて状態フラグ群を設定する
neg α	整数値 α について、$-\alpha$ の値を α へ書き込む
inc α	整数値 α について、$\alpha+1$ の値を α へ書き込む
dec α	整数値 α について、$\alpha-1$ の値を α へ書き込む

主要なビット演算命令

命令	説明
and α, β	ビット列 α、β について、α と β のビット毎の論理積を β へ書き込む
or α, β	ビット列 α、β について、α と β のビット毎の論理和を β へ書き込む
xor α, β	ビット列 α、β について、α と β のビット毎の排他的論理和を β へ書き込む
not α	ビット列 α をビット毎に反転した結果を α へ書き込む
shl α, β	ビット列 β を α (省略すると 1) bit だけ左シフトし β へ書き込む
shr α, β	ビット列 β を α (省略すると 1) bit だけ論理的右シフトし β へ書き込む
sar α, β	ビット列 β を α (省略すると 1) bit だけ算術的右シフトし β へ書き込む
rol α, β	ビット列 β を α (省略すると 1) bit だけ左ローテートし β へ書き込む
ror α, β	ビット列 β を α (省略すると 1) bit だけ右ローテートし β へ書き込む
rcl α, β	ビット列 β を CF を含めて α (省略すると 1) bit だけ左ローテートし β へ書き込む
rcr α, β	ビット列 β を CF を含めて α (省略すると 1) bit だけ右ローテートし β へ書き込む

主要な分岐命令

命令	説明
jmp *α	メモリオペランド α の値のアドレスへ分岐する
jmp a	アドレス a にプログラムカウンタ相対モードで分岐する
je a	$\mathtt{ZF}=1$ なら、アドレス a にプログラムカウンタ相対モードで分岐する
jne a	$\mathtt{ZF}=0$ なら、アドレス a にプログラムカウンタ相対モードで分岐する
jb a	$\mathtt{CF}=1$ なら、アドレス a にプログラムカウンタ相対モードで分岐する
jae a	$\mathtt{CF}=0$ なら、アドレス a にプログラムカウンタ相対モードで分岐する
ja a	$\mathtt{CF}=0$ かつ $\mathtt{ZF}=0$ なら、アドレス a にプログラムカウンタ相対モードで分岐する
jbe a	$\mathtt{CF}=1$ または $\mathtt{ZF}=1$ なら、アドレス a にプログラムカウンタ相対モードで分岐する
jl a	$\mathtt{SF}\neq\mathtt{OF}$ なら、アドレス a にプログラムカウンタ相対モードで分岐する
jge a	$\mathtt{SF}=\mathtt{OF}$ なら、アドレス a にプログラムカウンタ相対モードで分岐する
jg a	$\mathtt{SF}=\mathtt{OF}$ かつ $\mathtt{ZF}=0$ なら、アドレス a にプログラムカウンタ相対モードで分岐する
jle a	$\mathtt{SF}\neq\mathtt{OF}$ または $\mathtt{ZF}=1$ なら、アドレス a にプログラムカウンタ相対モードで分岐する
js a	$\mathtt{SF}=1$ なら、アドレス a にプログラムカウンタ相対モードで分岐する
jns a	$\mathtt{SF}=0$ なら、アドレス a にプログラムカウンタ相対モードで分岐する
jo a	$\mathtt{OF}=1$ なら、アドレス a にプログラムカウンタ相対モードで分岐する
jno a	$\mathtt{OF}=0$ なら、アドレス a にプログラムカウンタ相対モードで分岐する
call *α	%rip (プログラムカウンタ) の現在の値 (次の機械語命令の置かれている 64 bit 長のアドレス) をスタックにプッシュし、メモリオペランド α の値のアドレスへ分岐する
call a	%rip (プログラムカウンタ) の現在の値 (次の機械語命令の置かれている 64 bit 長のアドレス) をスタックにプッシュし、アドレス a にプログラムカウンタ相対モードで分岐する
ret	スタックからアドレス (64 bit 長) をポップし、そのアドレスへ分岐する

ただし、ZF はゼロフラグ、SF はサインフラグ、CF はキャリーフラグ、OF はオーバーフローフラグで、減算に対しては、最上位ビットでの

繰り下がりの発生を CF = 1 で示します。各命令のオペランドの大きさ(ビット長)が自明でない場合は、movq や movl、movw、movb のように、命令のニーモニックの末尾に q、l、w、b を付して、オペランドの大きさが、それぞれ 64 bit、32 bit、16 bit、8 bit であることを示すことができます。また、call 命令や ret 命令では、レジスタ %rsp がスタックポインタとして使用されます。

レジスタの指定法　Intel 64 アーキテクチャの 64 bit モードでは、次の 16 個の 64 bit 汎用レジスタを使用することができます。

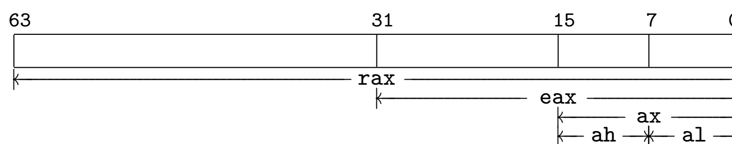
rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp, r8, r9, r10, r11, r12, r13, r14, r15

これらのレジスタは、その一部のビット列だけを、次の表のように 32 bit、16bit、8bit のレジスタとして使用することもできます。

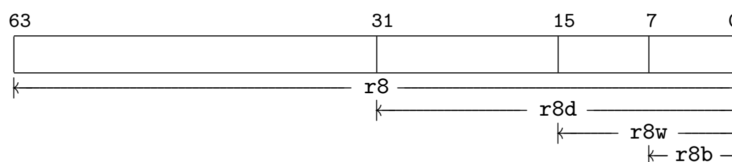
eax, ebx, ecx, edx, esi, edi, ebp, esp	それぞれ、レジスタ rax, ..., rsp の下位 32 bit
ax, bx, cx, dx, si, di, bp, sp	それぞれ、レジスタ rax, ..., rsp の下位 16 bit
al, bl, cl, dl, sil, dil, bpl, spl	それぞれ、レジスタ rax, ..., rsp の下位 8 bit
ah, bh, ch, dh	それぞれ ax, ..., dx の上位 8 bit

r8d, r9d, ..., r15d	それぞれ、レジスタ r8, r9, ..., r15 の下位 32 bit
r8w, r9w, ..., r15w	それぞれ、レジスタ r8, r9, ..., r15 の下位 16 bit
r8b, r9b, ..., r15b	それぞれ、レジスタ r8, r9, ..., r15 の下位 8 bit

つまり、64 bit の汎用レジスタ rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp は、その一部分を、次のような名前で指定できます (rax の例)。



また、r8, r9, ..., r15 については次のようになります (r8 の例)。



ただし、32 bit 長のデータ転送命令や演算命令で、eax, ebx, ecx, edx, esi, edi, ebp, esp, r8d, r9d, ..., r15d がデスティネーションオペランドとして指定された場合は、これらを含んでいる 64 bit 汎用レジスタ (rax, ..., r15) の上位 32 bit はすべて 0 となります。

データ転送命令や演算命令のオペランドの書式　データ転送命令や演算命令のオペランドでは、つぎのような書式で各種のアドレッシングモードを指定することができます[1]。メモリはバイトアドレッシングで、多バイト長のデータはリトルエンディアン方式で格納されます。

$\$c$	整数値の定数 c (即値アドレッシング)
a	メモリアドレス (定数) a (絶対アドレッシング)
%r	レジスタ r (レジスタ直接アドレッシング)
d(%b)	b をベースレジスタ、変位を d (省略すると 0) としたレジスタ相対アドレッシング
d(%rip)	変位を d (省略すると 0) としたプログラムカウンタ相対アドレッシング
d(%b, %i, s)	b をベースレジスタ (省略可)、i をインデックスレジスタ、変位を d、スケール値を s (1, 2, 4, 8 のいずれか) としたインデックス修飾付きレジスタ相対アドレッシング

以下は、これらのアドレッシングモードを使用したデータ転送命令の例です。

mov $0x12345678, %rax	定数 0x0000000012345678 をレジスタ %rax に書き込む
mov 0x12345678, %rbx	メモリアドレス 0x12345678 に置かれたデータ (64 bit 長) をレジスタ %rbx にコピーする
mov %rcx, 0x1c(%rbp)	レジスタ %rbp の値に定数 0x1c を加えたメモリアドレスに、レジスタ %rcx の値を書き込む
mov 0x1c(%rax,%rsi,4), %edx	%rax の値に、%rsi の値を 4 倍したものを加え、さらに 0x1c を加えたメモリアドレスに置かれたデータ (32 bit 長) を %edx にコピーする

[1]前ページの表に挙げたデータ転送命令や演算命令では、2 つのオペランド α と β が、ともにメモリオペランドとなることはありません。また、レジスタ相対アドレッシングモード (インデックス修飾付きを含む) で、ベースレジスタやインデックスレジスタとして使用できるのは 64 bit 長の汎用レジスタのみで、%rsp はインデックスレジスタとしては使用できません。

分岐命令のアドレッシングモード　`jmp *`α や `call *`α の `*` は間接アドレッシングであることを示しています。α の部分には、即値以外のアドレッシングモードが使用できますが、実際に、分岐先アドレスとなるのは、α に格納されている値です。このため、α 自身がレジスタ直接アドレッシングであっても、`*`α ではレジスタ間接アドレッシングを指定したことになり、そのレジスタに格納されているアドレスに分岐します。

一方、`jmp` a, `je` a, ..., `call` a では、a で絶対アドレスを指定しますが、実際には、この命令の次の命令が置かれるアドレス[2]から a までの変位を指定したプログラムカウンタ相対アドレッシングで分岐する命令となります。

以下は、いろいろなアドレッシングモードを使用した分岐命令の例です。

`jmp *rax`	レジスタ `rax` に格納されているアドレスに分岐する (レジスタ間接アドレッシング)
`jmp *(rax)`	レジスタ `rax` に格納されているメモリアドレスに格納されているアドレスに分岐する (レジスタメモリ間接アドレッシング)
`jmp *0x12345678`	メモリアドレス 0x12345678 に格納されているアドレスに分岐する (絶対メモリ間接アドレッシング)
`jmp 0x12345678`	アドレス 0x12345678 に (プログラムカウンタ相対モードで) 分岐する
`je 0x12345678`	`ZF` $= 1$ の場合に限ってアドレス 0x12345678 に (プログラムカウンタ相対モードで) 分岐し、`ZF` $= 0$ の場合は何もしない

条件分岐命令のニーモニック　条件分岐命令のニーモニック `je`, `jne`, ..., `jno` は、それぞれ以下のような意味を持っています。

`je`	Jump if Equal	`jne`	Jump if Not Equal
`jb`	Jump if Below	`jae`	Jump if Above or Equal
`ja`	Jump if Above	`jbe`	Jump if Below or Equal
`jl`	Jump if Less	`jge`	Jump if Greater or Equal
`jg`	Jump if Greater	`jle`	Jump if Less or Equal
`js`	Jump if Sign	`jns`	Jump if Not Sign
`jo`	Jump if Overflow	`jno`	Jump if Not Overflow

`je` から `jle` までの 10 通りの条件分岐命令のニーモニックは、`cmp` α, β や `sub` α, β という命令を実行した際に設定される各フラグの状態に基づく、α を基準とした β の大きさを表しています。above や below は符号なし整数としての大小を、greater や less は符号付き整数とての大小を意味します。

たとえば、次のような 2 つの機械語命令を順に実行すると、`rax` と `rbx` に格納されているビット列を符号付き整数とみなして、`rbx` $>$ `rax` が成り立っている場合にのみ、アドレス 0x1234 へ分岐します。

```
cmp %rax, %rbx
jg 0x1234
```

参考資料：Intel Core i3-4330T プロセッサのキャッシュ構成

Intel 社の Core i3-4330T プロセッサには、それぞれ 3 GHz のクロック周波数で動作する 2 つの CPU コアが内蔵されており、次の表のような 3 つのレベルのキャッシュメモリが搭載されています。

	L1 キャッシュ		L2 キャッシュ	L3 キャッシュ
	命令用	データ用		
配置	CPU コアごと			2 つのコアで共有
容量	32 KiB	32 KiB	256 KiB	4 MiB
ラインサイズ	64 B	64 B	64 B	64 B
方式	8-way セット アソシアティブ	8-way セット アソシアティブ	8-way セット アソシアティブ	16-way セット アソシアティブ
書き込み処理	—	write back	write back	write back

(参考資料終り)

[2] IA-32 や Intel 64 アーキテクチャでは、命令の実行の前にプログラムカウンタが更新されますので、分岐命令が実行されるときのプログラムカウンタの値はその分岐命令の次に置かれている命令のアドレスとなっています。