コンピュータとプログラミング
第7回（アセンブリ言語 第6回）
進んだ話題

大山恵弘

再帰関数呼び出し

再帰関数呼び出し

- 後半のC言語編で詳しく取り上げると思うが，先んじて少しだけ説明

```
#include <stdio.h>

int fact(int x)
{
  if (x <= 1) {
    return 1;
  } else {
    return x * fact(x - 1);
  }
}

int main(void)
{
  int a = fact(10);
  printf("%d¥n", a);
  return 0;
}
```

- factの実行の中で，さらにfactを呼び出して計算する
- 呼び出すたびに **x** が1ずつ減るので，呼び出しの連鎖はいつかは止まる
- factから返った後に掛け算がある

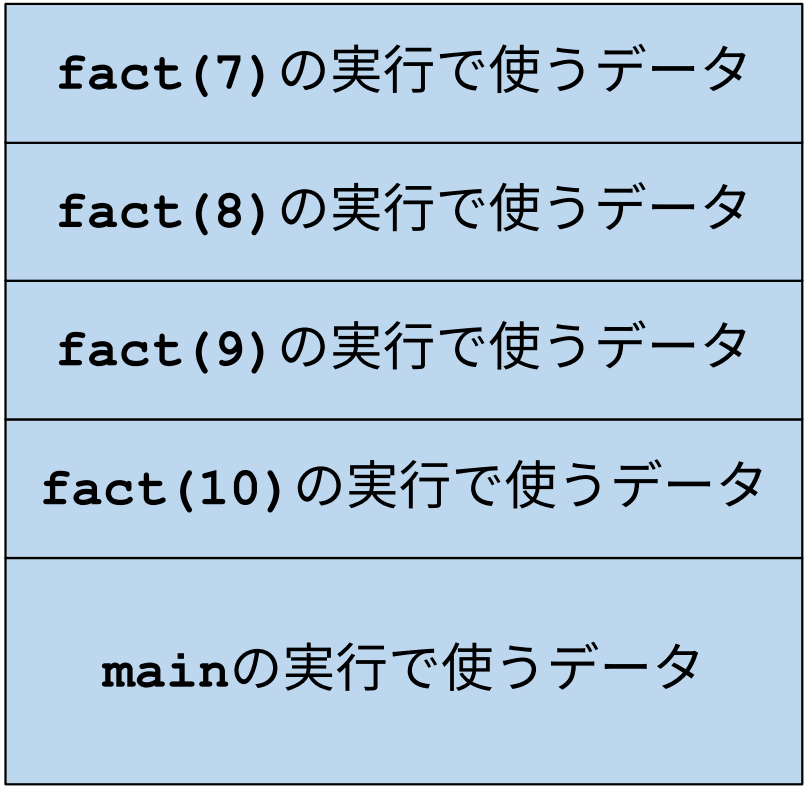
やる必要があること：（再帰）呼び出しで実行状態を保存

- fact(10) = 10 * fact(9)
  - 10は覚えておいてfact(9)を計算
- fact(9) = 9 * fact(8)
  - 9は覚えておいてfact(8)を計算
- ...
- fact(2) = 2 * fact(1)
  - 2は覚えておいてfact(1)を計算
- fact(1) = 1

- fact(N)のNが1増えるたびに，覚えておくべき数が1個増える
- しかし，計算に使えるレジスタは16個しかない
- fact(100)はどう計算する？

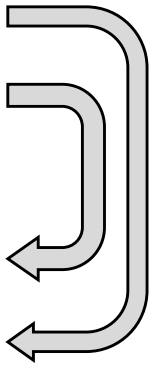
スタックを用いた，データの待避と復元

- 後で使うデータをスタック，すなわちメモリに積む（pushする）
- 必要になったら取り出す（popする）
  - 最後に積んだデータが最初に出てくる
- 再帰関数呼び出しのケース：
  - 関数を呼び出すたびに，覚えておくべき値をpush
  - 関数から返ってきたら値をpop



待避と復元のコードの例

```
main:
…
add $100, %rcx # 中間計算結果
sub $4, %rdx   # 中間計算結果
push %rcx      # 待避
push %rdx      # 待避
call func
pop %rdx       # 復元
pop %rcx       # 復元
…
 （ここで rcx と rdx を用いて計算）
…
```



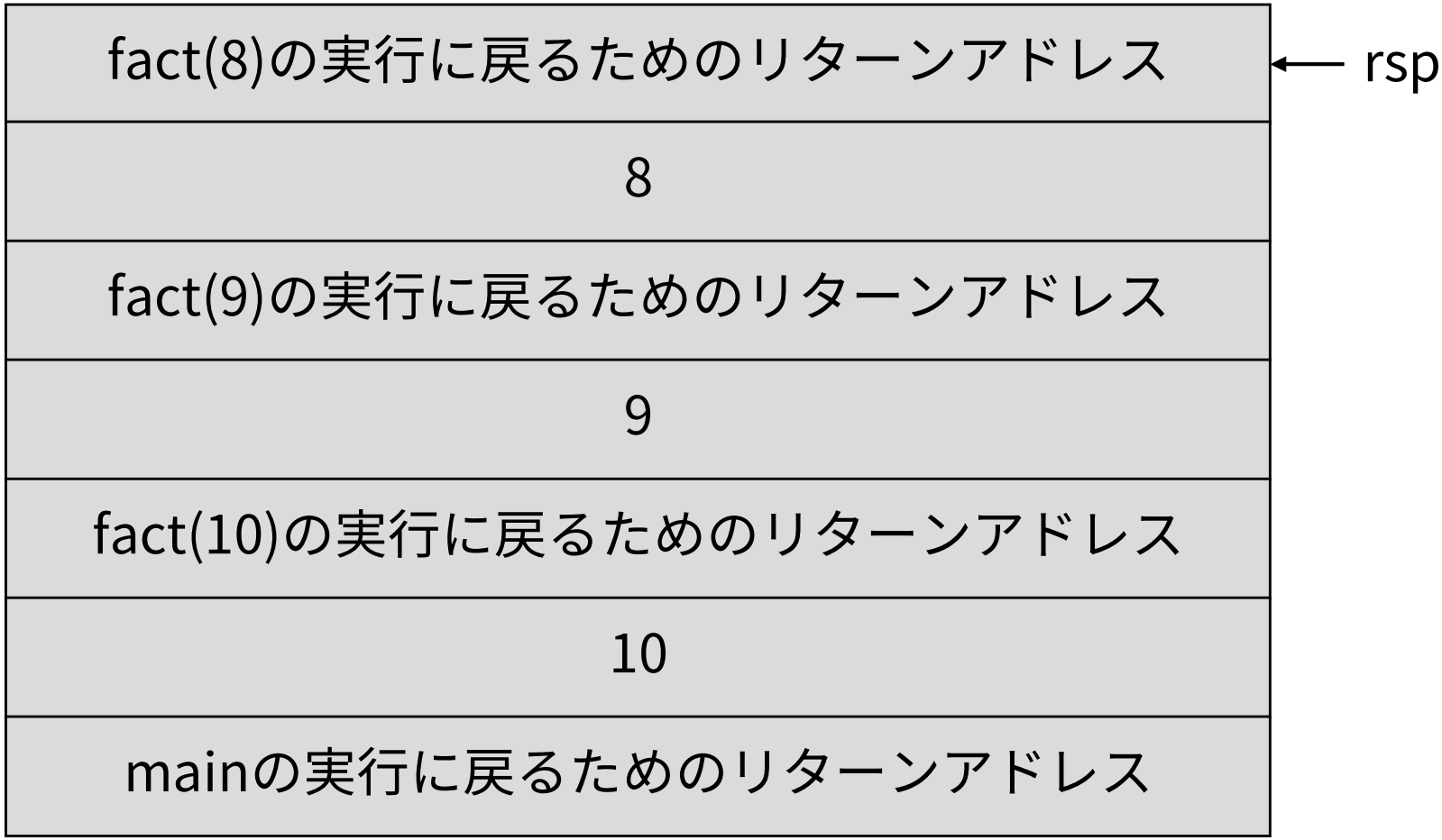
```
func:
…
mov $1, %rcx # 破壊される
…
add $5, %rdx # 破壊される
…
ret
```

階乗を計算するコードの例

```
        .text
        .global fact
        .global main
fact:
        cmp $1, %rdi # x <= 1?
        jle L1
        push %rdi    # xを待避
        sub $1, %rdi # x-1を計算
        call fact
        pop %rdi     # xを復元
        mul %rdi     # xとfact(x-1)の答の掛け算
        ret
L1:
        mov $1, %rax # xが1なので1を返す
        ret
main:
        mov $10, %rdi
        call fact
        call finish
```

```
$ gcc max.s lib.s
$ ./a.out
Finished.
rax=0x0000000000375f00 (3628800)
rbx=...
```

再帰関数呼び出しによる階乗の計算中のスタック



進んだ話題：末尾再帰呼び出し → ループ

- 関数末尾の再帰呼び出しをループのように実行する最適化が存在する
  - 関数呼び出しなのにスタックにデータを積まない
  - 関数呼び出しが分岐命令に変換される
  - コンパイラの定番技術

```
#include <stdint.h>
uint64_t fact_aux(uint64_t r,
uint64_t n)
{
  if (n == 1) {
    return r;
  } else {
    return fact_aux(r * n,
                    n - 1);
  }
}

uint64_t fact(uint64_t n)
{
  return fact_aux(1, n);
}
```

gcc -S -O2

どこにも call 命令が無い！

```
fact_aux:
        movq    %rdi, %rax
        cmpq    $1, %rsi
        je      .L5
.L2:
        imulq   %rsi, %rax
        subq    $1, %rsi
        cmpq    $1, %rsi
        jne     .L2
.L5:
        ret
```

飛び先が実行時に決まるCALL/JMP命令（間接関数呼び出し / 間接ジャンプ）

- **`call *`***exp*：*exp*の値を計算し，その値を関数の開始アドレスとみなして呼び出す
  - **`call *`**レジスタ
    - 例：**`call *%rax`**：raxレジスタに入っている値を関数の開始アドレス（ラベル）とみなして呼び出す
  - **`call *`**定数**`(`**レジスタ**`,`** レジスタ**`,`** 定数**`)`**
    - 例：**`call *-0xe0(%rbx, %rsi, 8)`**：オペランドの値を計算し，その値を関数の開始アドレスとみなして呼び出す
- jmp命令も同様のオペランドを取れる
  - **`jmp *%rax`** など
  - ただし，関数呼び出しではなく単なるジャンプ（リターンアドレスを積まない）
- 典型的には，実行すべきコードが実行時になってから決まるプログラムで役立つ

スタックについての他の話題

演習用のファイルlib.sが使うスタック

- 前回の授業で，**`lib.s`** の中でスタック領域を作っていると言ったかもしれないが訂正
- Linux上のgcc用の標準ライブラリが，プログラムを実行する際の初期化処理の中で，スタック領域を作成
- スタック領域はどのアドレスからどのアドレスまで？
  - その情報を得るのは少し難しい：
    対象プログラムを止めて，対象プログラムのPIDを調べて，
    `cat /proc/`*pid*`/maps`** を実行すればわかる
    - **`[stack]`** と書かれた行にスタックの開始・終了アドレスが出力される
- 教員がCOINSのサーバでsample.sから作った実行可能ファイルについて調べたときには，スタックの範囲は
  0x7fffffdd000から0x7fffffff000（サイズ0x22000）

スタックオーバフロー

- スタックのために確保したメモリ領域の境界を超えてさらにスタックに値を入れよう（pushしよう）とすると発生する
- 突然プログラムが異常終了することもあれば，スタックオーバフローというエラーが表示されて終了することもある
  - OSやライブラリや場合による
- COINSのサーバで，無限に入れ子で関数呼び出しをするプログラムを実行したらどうなったか？
  → 「**Segmentation fault (コアダンプ)**」と表示されて終了
  - 内部的には，実はスタックのサイズは0x22000から0x800000に増やされていたが，増やす限界が来たのか，最終的には異常終了

```
        .text
        .global g
        .global main
g:      push $1
        call g
        ret
main:   call g
        call finish
```

- 結局言いたいこと： スタック領域のサイズ（入れ子で関数呼び出しができる回数）は有限

他の様々な命令

NOP命令

- 何もしない命令
- 何のためにあるのか？
  - アラインメント： 関数の開始アドレスやジャンプ先アドレスを，キリのいい値（8の倍数など）にする
    - 関数と関数の間をNOPで埋めたりする
  - 動的コード生成
    - コード内にNOPで場所を空けておく
    - 空けておいた場所に後から命令列を書き込んで実行する
  - 可読性向上
    - 区切りの印としてNOP命令の列を入れる
  - 時間つぶし
    - NOPだけのループを100万回まわる，など

```
...
100403927: pop    %rsi
100403928: pop    %rdi
100403929: pop    %rbp
10040392a: retq
10040392b: nop
10040392c: nop
10040392d: nop
10040392e: nop
10040392f: nop
100403930: mov    %rsi,%rax
100403933: add    $0x38,%rsp
100403937: pop    %rbx
100403938: pop    %rsi
100403939: pop    %rdi
10040393a: pop    %rbp
10040393b: retq
10040393c: nop
10040393d: nop
10040393e: nop
10040393f: nop
100403940: push   %rdi
100403941: push   %rsi
100403942: push   %rbx
100403943: sub    $0x20,%rsp
100403947: movzwl 0x10(%rcx),%ebx
...
```

関数間の「スキマ」

関数

関数間の「スキマ」

LOOP命令

- rcxレジスタの値を1減らし，結果が非零ならば，
  オペランドのアドレスにジャンプする

```
      mov $X, %rsi
      mov $Y, %rdi
      mov $256, %rcx
L1:
      mov (%rsi), %rbx
      mov %rbx, (%rdi)
      add $8, %rsi
      add $8, %rdi
      loop L1
```

ラベル **X** の領域からラベル **Y** の領域に，
64 bitのデータ256個をコピー

```
      mov $str1, %rsi
      mov $str2, %rdi
      mov $100, %rcx
L1:
      mov (%rsi), %bl # 8bit reg
      mov %bl, (%rdi)
      inc %rsi
      inc %rdi
      loop L1
```

ラベル **str1** の領域からラベル **str2** の
領域に，100バイトをコピー

LOOP系命令

- loopz（loopeも同じ）：rcxレジスタの値を1減らして，結果が非零であり，かつ，ZF=1であるならば，オペランドのアドレスにジャンプする
- loopnz（loopneも同じ）：rcxレジスタの値を1減らして，結果が非零であり，かつ，ZF=0であるならば，オペランドのアドレスにジャンプする

「最大N個の要素に対して同じ処理を実行していくが，途中に〇〇な要素があったらそこで打ち切る」というような処理の記述で役立つかも知れない

浮動小数

- x86は浮動小数のためのレジスタを備えている
  - xmm0, xmm1, ..., xmm7
- x86は浮動小数演算のための命令を備えている
  - fadd, fsub, fmul, fdiv, fcom, fabs, fcos, fsin, fptan, fsqrt
- 使う機会は少ない上，整数演算と同じ部分も多いので，扱わない

OSが使う命令

- in (input from port): デバイスからデータを受け取る
  - **`in $0x64, %eax`**： 0x64番ポートから入力を受け取りeaxに入れる
- out (output to port): デバイスにデータを送る
  - **`out %eax, $0x60`**： eaxの値を0x60番ポートに送る
- hlt (halt): 命令の実行を停止し，CPUを停止状態にする
- invd: キャッシュをフラッシュする（捨てる）
- invlpg: TLBエントリ（メモリアクセス高速化のためにCPUが一時的に保持する管理データ）を無効化する
- vmenter/vmexit: 仮想マシンを実行する/仮想マシンの実行から抜ける

など多数

雑多な命令

- cpuid: CPUの識別番号やCPUが提供する機能の情報を取得する
- rdtsc: time stamp counter（1 CPUクロック程度の短い時間の経過ごとに1増える64bitのCPU内カウンタ，TSC）を読む
  - Time stamp counterの上位32bitがrdx，下位32bitがraxに入る
- sysenter/sysexit, syscall/sysret: システムコール（OSカーネルが提供する手続き）を呼び出す/システムコールからリターンする
- aesdec: AES暗号の復号処理の1ラウンドを実行する
- prefetch: メモリからキャッシュに先行的にデータを読む
- rdrand: CPUが内部で生成した乱数を読む
- sha256msg2: SHA256の最後の計算を実行する
- ud2: 無効命令（実行するとエラーで終了する）

進んだ話題：
CPUID命令によるCPU情報の取得

```
        .text
        .global main
main:
        mov $0, %rax
        mov $0, %rbx
        mov $0, %rcx
        mov $0, %rdx
        cpuid
        call finish
```

```
$ gcc cpuinfo.s lib.s
$ ./a.out
Finished.
rax=0x000000000000000d (13)
rbx=0x00000000756e6547 (1970169159)
rcx=0x000000006c65746e (1818588270)
rdx=0x0000000049656e69 (1231384169)
rsi=...
```

"Genu" のASCIIコード

"ntel" のASCIIコード

"inel" のASCIIコード

進んだ話題：
RDTSC命令による時間経過の計測

```
        .data
N:      .quad 10000000000
        .text
        .global dloop
        .global main
dloop:
        mov $0, %rcx
L1:
        cmp N, %rcx
        jge L2
        inc %rcx
        jmp L1
L2:
        ret
main:
        rdtsc
        mov %rax, %r8
        mov %rdx, %r9
        call dloop
        rdtsc
        sub %r8, %rax
        sbb %r9, %rdx
        call finish
```

dloopの実行中のTSCの増分
（一種の経過時間として利用可能）

```
$ gcc lib.s dummyloop.s
$ ./a.out
Finished.
rax=0x000000001ce36288 (484663944)
rbx=0x0000000000000000 (0)
rcx=0x000000002540be400 (10000000000)
rdx=0x0000000000000002 (2)
rsi=0x00007ffc98852b48
(140722867350344)
rdi=0x0000000000000001 (1)
r8 =0x0000000064459b03 (1682283267)
r9 =0x000000000001ac339 (1753913)
...
```

dloopの実行前のTSCの値

進んだ話題：
RDRAND命令による乱数の取得

```
    .text
    .global main
main:
    mov $0, %rax
    rdrand %rax
    call finish
```

```
$ gcc rdrand_test.s lib.s
$ ./a.out | grep rax
rax=0xf515c04245b34ead (-7865111690365300003)
$ ./a.out | grep rax
rax=0xc00442bafac66e82 (-4610486747681886590)
$ ./a.out | grep rax
rax=0x2ab11e3280b59cb2 (3076273222727343282)
$ ./a.out | grep rax
rax=0x04c26f8a9032292d (342959162548955437)
$ ./a.out | grep rax
rax=0x5f8733bf6163be70 (6883527452524789360)
$
```

乱数生成に成功したらCFが1に，
失敗したらCFが0になるので，
本来ならばCF (RFLAGS) も見るべき

その他いろいろ

同じ，または，ほぼ同じ動作をする命令（1/2）

- （ほぼ）同じ動作をする異なる命令の組が存在する
  - raxレジスタに0をセット：
    `mov $0, %rax` や **`and $0, %rax`** や **`xor %rax, %rax`**
  - raxレジスタの値を2倍にする：
    2をかけたり，**`shl $1, %rax`**（左に1ビットシフト）を実行したり
  - raxレジスタの値を半分にする：
    2で割ったり，**`shr $1, %rax`**（右に1ビットシフト）を実行したり
  - rbxレジスタの値が0かどうかを検査する：
    `cmp $0, %rbx` や **`test %rbx, %rbx`**
  - rcxレジスタの値が奇数かどうかを検査する：
    `and 1, %rcx` で最下位ビットを取り出したり，rcxレジスタをdiv命令で2で割り剰余を見たり
  - 他の組も見つけてみよう！

同じ，または，ほぼ同じ動作をする命令（2/2）

- 命令が異なるので当然バイナリコードも異なる
  - 例：**`mov $0, %rax`** → **`48 c7 c0 00 00 00 00`**
    `and $0, %rax` → **`48 83 e0 00`**
    `xor %rax, %rax` → **`48 31 c0`**
  - コード難読化やマルウェアの変異に利用されることがある
- 実行速度も異なる
  - 例：一概には言えないが一般に，シフト命令や加減算命令は速く，乗除算命令やOSが使う命令は遅い
  - 速い命令を使うと，プログラムの動作を速くできる

他の言語とアセンブリ言語の連携（C言語の例）

- C言語コードとアセンブリ言語コードを組み合わせて，1つのプログラムとして使える
  - 関数の単位で分割する
- アセンブリ言語のコードからC言語の関数の呼び出し
  - C言語をコンパイルする環境の規約に従って引数をレジスタやスタックに配置し，call命令で関数を呼び出せばOK
  - アセンブリ言語のコードからprintf関数などが呼び出せる
- C言語のコードからアセンブリ言語の関数の呼び出し
  - 同じ規約に従ってアセンブリ言語の関数を実装し，C言語のコードでは普通に関数呼び出しを実行すればOK
- 今年度のこの科目では扱わないが，興味のある人は試してみてほしい

ケーススタディ

- Linux（OSカーネル）のアセンブリ言語コードを読む
  - Linux カーネル総本山： https://www.kernel.org
  - アセンブリ言語のコード
    - **`arch/x86/kernel/*.S`**
    - **`arch/x86/entry/*.S`**
    - **`arch/x86/crypto/*.S`**
    - **`arch/x86/boot/*.S`**
    - **`arch/x86/lib/*.S`**

    など
- 一般のアプリケーションのアセンブリ言語コードを読む

OSカーネルのシステムコール処理に突入する部分のソースコード

```
SYM_CODE_START(entry_SYSCALL_64)
     UNWIND_HINT_ENTRY
     ENDBR

     swapgs
     /* tss.sp2 is scratch space. */
     movq %rsp, PER_CPU_VAR(cpu_tss_rw + TSS_sp2)
     SWITCH_TO_KERNEL_CR3 scratch_reg=%rsp
     movq PER_CPU_VAR(pcpu_hot + X86_top_of_stack), %rsp

SYM_INNER_LABEL(entry_SYSCALL_64_safe_stack, SYM_L_GLOBAL)
     ANNOTATE_NOENDBR

     /* Construct struct pt_regs on stack */
     pushq     $__USER_DS                    /* pt_regs->ss */
     pushq     PER_CPU_VAR(cpu_tss_rw + TSS_sp2)  /* pt_regs->sp */
     pushq     %r11                        /* pt_regs->flags */
     pushq     $__USER_CS                    /* pt_regs->cs */
     pushq     %rcx                        /* pt_regs->ip */
SYM_INNER_LABEL(entry_SYSCALL_64_after_hwframe, SYM_L_GLOBAL)
     pushq     %rax                        /* pt_regs->orig_ax */

     PUSH_AND_CLEAR_REGS rax=$-ENOSYS

     /* IRQs are off. */
     movq %rsp, %rdi
     /* Sign extend the lower 32bit as syscall numbers are treated as
int */
     movslq    %eax, %rsi
...
```

まったく歯が立たなくもないのでは？

マクロの多さにより，とっつきにくくはなっているが．．．

時間をかければ解読できる（かも）！

アプリケーションのバイナリを逆アセンブルして解読

violet01の /usr/local3/coins/linux/ruby-2.6/bin/ruby の逆アセンブル結果

```
0000000000400890 <main>:
  400890:       55                                  push   %rbp
  400891:       53                                  push   %rbx
  400892:       48 83 ec 28                         sub    $0x28,%rsp
  400896:       89 7c 24 0c                         mov    %edi,0xc(%rsp)
  40089a:       31 ff                               xor    %edi,%edi
  40089c:       48 89 34 24                         mov    %rsi,(%rsp)
  4008a0:       48 8d 35 f1 01 00 00                lea    0x1f1(%rip),%rsi
  4008a7:       64 48 8b 04 25 28 00 00 00          mov    %fs:0x28,%rax
  4008b0:       48 89 44 24 18                      mov    %rax,0x18(%rsp)
  4008b5:       31 c0                               xor    %eax,%eax
  4008b7:       e8 c4 ff ff ff                      callq  400880 <setlocale@plt>
  4008bc:       48 89 e6                            mov    %rsp,%rsi
  4008bf:       48 8d 7c 24 0c                      lea    0xc(%rsp),%rdi
  4008c4:       e8 87 ff ff ff                      callq  400850 <ruby_sysinit@plt>
  4008c9:       48 8b 2c 24                         mov    (%rsp),%rbp
  4008cd:       8b 5c 24 0c                         mov    0xc(%rsp),%ebx
  4008d1:       48 8d 7c 24 10                      lea    0x10(%rsp),%rdi
  4008d6:       e8 95 ff ff ff                      callq  400870 <ruby_init_stack@plt>
  4008db:       e8 50 ff ff ff                      callq  400830 <ruby_init@plt>
  4008e0:       48 89 ee                            mov    %rbp,%rsi
  4008e3:       89 df                               mov    %ebx,%edi
  4008e5:       e8 56 ff ff ff                      callq  400840 <ruby_options@plt>
...
```

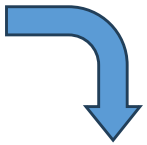
他のCPU（アーキテクチャ）

- 異なる命令のセットを提供
- 要注目で学ぶ価値ありのもの
  - Arm
  - RISC-V
  - （CPUではないが）WebAssembly
- 廃れており苦境にいるもの
  - SPARC
  - MIPS
  - SuperH
  - Power

個人の感想です

Armの命令列（1）

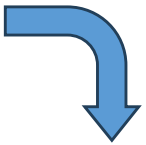
```
int fact(int x)
{
  if (x == 1) {
    return 1;
  } else {
    return x * fact(x - 1);
  }
}
```



```
000104d4 <fact>:
   104d4:       e3500001        cmp     r0, #1
   104d8:       0a000005        beq     104f4 <fact+0x20>
   104dc:       e92d4010        push    {r4, lr}
   104e0:       e1a04000        mov     r4, r0
   104e4:       e2400001        sub     r0, r0, #1
   104e8:       ebfffff9        bl      104d4 <fact>
   104ec:       e0000094        mul     r0, r4, r0
   104f0:       e8bd8010        pop     {r4, pc}
   104f4:       e3a00001        mov     r0, #1
   104f8:       e12fff1e        bx      lr
```

Armの命令列（2）

```
int fib(int x)
{
  if (x < 2) {
    return 1;
  } else {
    return fib(x - 1) + fib(x - 2);
  }
}
```



```
000104d4 <fib>:
   104d4:       e3500001        cmp     r0, #1
   104d8:       da000008        ble     10500 <fib+0x2c>
   104dc:       e92d4038        push    {r3, r4, r5, lr}
   104e0:       e1a04000        mov     r4, r0
   104e4:       e2400001        sub     r0, r0, #1
   104e8:       ebfffff9        bl      104d4 <fib>
   104ec:       e1a05000        mov     r5, r0
   104f0:       e2440002        sub     r0, r4, #2
   104f4:       ebfffff6        bl      104d4 <fib>
   104f8:       e0850000        add     r0, r5, r0
   104fc:       e8bd8038        pop     {r3, r4, r5, pc}
   10500:       e3a00001        mov     r0, #1
   10504:       e12fff1e        bx      lr
```

余談：世界を震撼させた，Intel CPUに存在した脆弱性Meltdownについての論文

```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

Listing 2: The core of Meltdown. An inaccessible kernel address is moved to a register, raising an exception. Subsequent instructions are executed out of order before the exception is raised, leaking the data from the kernel address through the indirect memory access.

このコードのAT&T表記

```
xor %rax, %rax
retry:
movb (%rcx), %al
shl $0xc, %rax
jz retry
movq (%rbx, %rax), %rbx
```

理解できる！

software vulnerability exists that can be exploited to gain kernel privileges or leak information. The attacker targets secret user data, e.g., passwords and private keys, or any other valuable information.

5.1 Attack Description

dress, *i.e.*, whether this virtual address is user accessible or only accessible by the kernel. As already discussed in Secti… ugh a per- missi… ded by the h… ystems al- … ress space

As … to a valid physi… CPU can acces… ly differ- ence … the CPU raises … level does not a… the user space cannot simply read the contents of such an address. However, Meltdown exploits the out-of-order execution of modern CPUs, which still executes instructions in the small time window between the illegal memory access and the raising of the exception.

In line 4 of Listing 2, we load the byte value located at the target kernel address, stored in the RCX register, into the least significant byte of the RAX register repre-

Lipp et al., "Meltdown: Reading Kernel Memory from User Space," USENIX Security Symposium 2018.

余談：アセンブリ言語ゴルフ

- お題の処理をできるだけ少ない命令数で実現する知的競技
- 例
  - 与えられた整数の約数の個数をraxに入れて終了する関数
  - 与えられた整数2つの最小公倍数をraxに入れて終了する関数
  - 与えられた整数が素数なら1，合成数なら0をrax に入れて終了する関数
- 「78命令でできた！」「私は175命令でできた！」「世界記録は63命令」「32命令だけの神コード！」などと競う

コードゴルフ関連情報

- Wikipedia: コードゴルフ
  https://ja.wikipedia.org/wiki/%E3%82%B3%E3%83%BC%E3%83%89%E3%82%B4%E3%83%AB%E3%83%95
- Elf Golf
  http://shinh.skr.jp/binary/fsij061115/
- Code Golfについて
  https://www.slideshare.net/shinh/code-golf
- コードゴルフとは？初心者向けチュートリアル、開発に役立つ情報、ツール、実例などのトピック集
  https://qiita.com/tags/%e3%82%b3%e3%83%bc%e3%83%89%e3%82%b4%e3%83%ab%e3%83%95
- プログラミング初級者から上級者まで一緒に楽しめるコードゴルフ大会開催のコツ
  https://style.biglobe.co.jp/entry/2020/08/26/130000
- C++でコードゴルフをする話
  https://qiita.com/gengar-094/items/f978bb1ed33253086d14

おわりに

- 低レイヤの理解なくして，
  情報やコンピュータの真の理解なし

- 低レイヤの世界は（人にもよるが）
  すごく楽しいし，奥が深く飽きがこない

中間テスト

- 日時と場所： 2025年6月6日（金） 3限 12:15-13:30　3A204
- 試験時間：60分
  - 12:15から用紙の配布を開始し，数分後に試験開始
- 範囲： UNIXとアセンブリ言語の部分全体
- 形式： 筆記試験
  - 講義で配布した資料（スライド，演習課題，小テストの問題と解答，演習用のプログラムや資料を含む）を印刷した紙と，それらの紙や白紙やノートに手書きで書き込みを加えた紙のみが持ち込み可
  - 印刷などの手書き以外の方法で情報を書き込んだ紙は持ち込み不可
  - 時計以外の全ての電子機器の利用不可
  - 座席自由，学生証を机上に提示

中間テストに参加できない場合

- どうしてもこの日に試験を受けられない人は，6月4日（水）までに担当教員の大山（oyama@cs.tsukuba.ac.jp）にメールで連絡すること
  - 受けられない理由を記載すること
  - その日と翌日中に対策を検討し，6月5日（木）の21:00までには（多くの場合もっと早くに），どういう措置を取るかを連絡します
- その後，病気や事故により急遽やむをえず試験に参加できなくなった人も，できるだけ早く連絡して指示を仰ぐこと