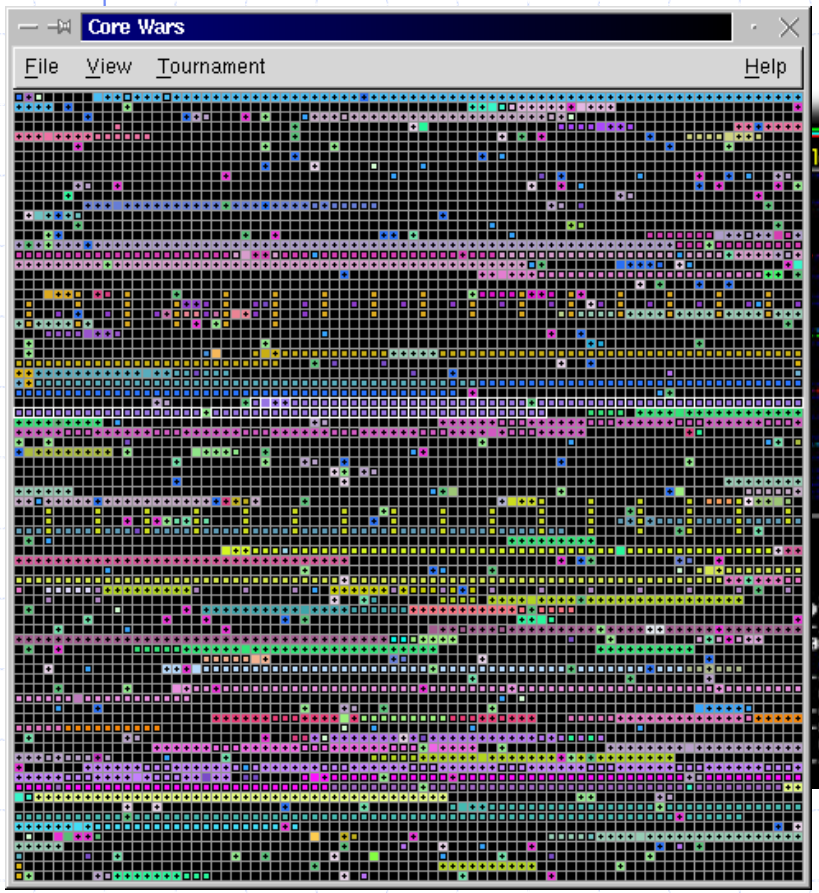
実行時自己書き換え住境

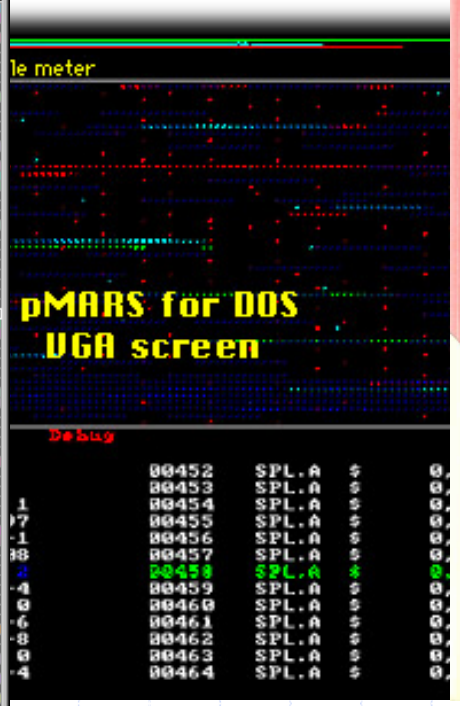
首藤 一幸

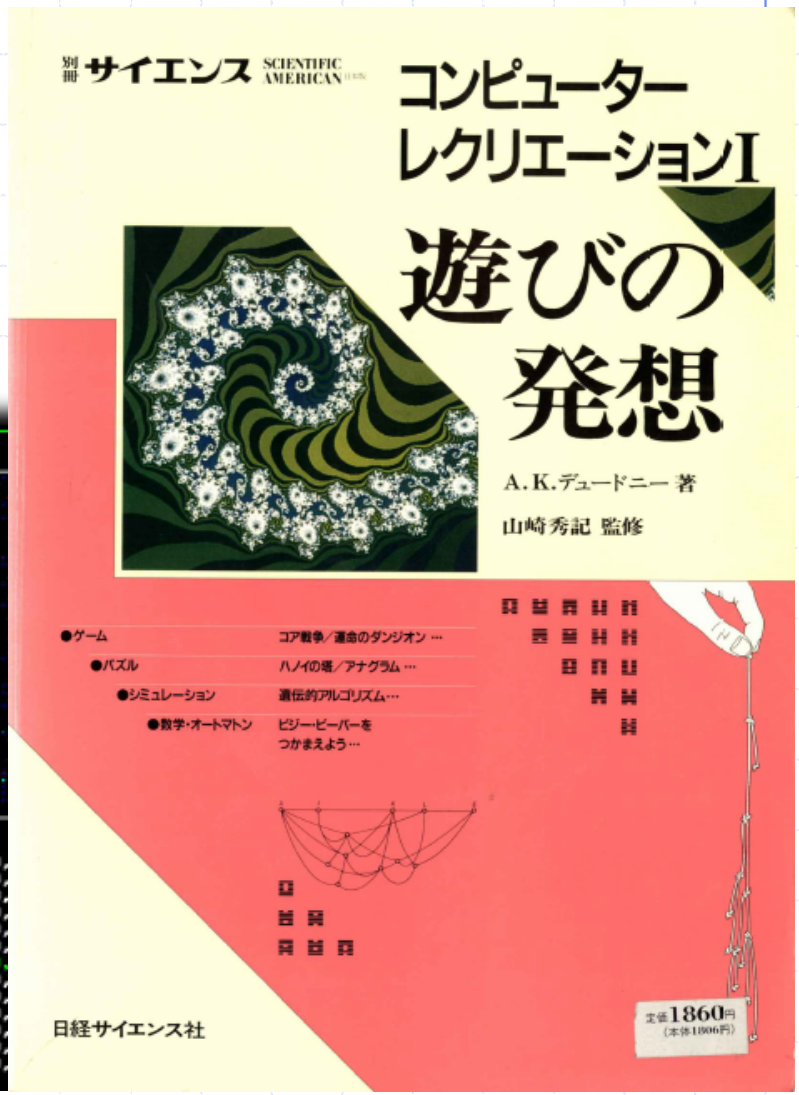
Core Wars (1984)

復刊希望

- 機械語プログラムどうしを戦わせるゲーム。
  - リング状のメモリ空間で相手を破壊し合う。
  - 自分のコードを実行しているスレッドが なくなったら負け。
  - 仮想マシンの命令は 10種類程度。







月刊ASCIIでも2回(?) 特集記事に。

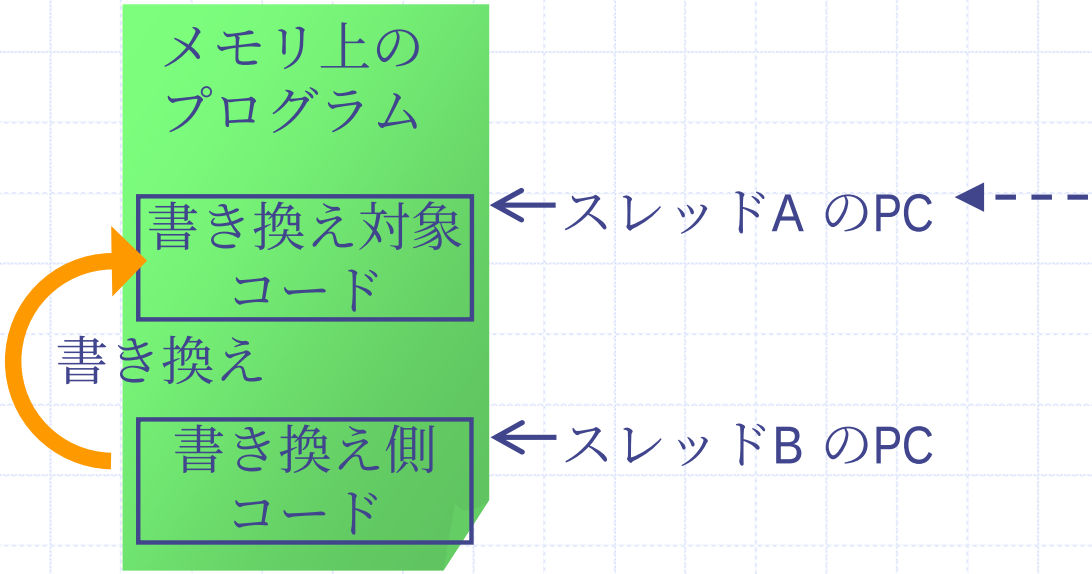
Core Wars 動物図鑑

- Imp（小鬼）
  - `MOV 0 1`
  - 自身を次のアドレスにコピーする。永遠に。
- Imp Pit（小鬼地獄）
  - ```
    MOV #0 -1
    JMP -1
    ```
  - 1番地手前に即値 0 を書き込み続けて、小鬼を上書きする。
- Dwarf（一寸法師）
  - ```
    DAT -1
    ADD #5 -1
    MOV #0 @-2
    JMP -2
    ```
  - 5番地おきに即値 0 を爆撃する。永遠に。

マルチスレッドなプログラム

- 実行の流れ（スレッド）が複数あって、メモリ空間を共有する。
  - スレッドごとにプログラムカウンタ(PC)があり、それぞれプログラム中の別の場所を実行する。
  - 同一プロセッサ上での並行処理かもしれないし、SMPなど、複数プロセッサでの並列処理かもしれない。

Q. これってリアル Core Wars ですか？



書き換え中の個所を実行するスレッドがあるかもしれず。

実は、自己書き換えでなくとも、外部からの書き換えでも同じこと。

なんで自己書き換えなんかするの？

私が見た例

- HotSpot VM (SunのJava仮想マシン)
  - アプリのスレッドを任意の個所で停止させるために、ソフトウエア割り込み命令を上書きする。後で元のコードを書き戻す。
    ガーベジコレクションやOn-stack replacement 目的。

- IBM の Java仮想マシン (by IBM東京基礎研)
  - メソッド探索せずに呼び出すことができていた(仮想)メソッドの呼び出しに探索が必要になった場合、JITコンパイル済みのコードを書き換える。
    あらかじめ、直接呼び出すコードと探索を行うコードの両方を生成しておく。

なんで自己書き換えなんかするの？

私が書いた例

- shuJIT (x86用 Java JITコンパイラ)
  - 一度だけ実行すればよい処理を、
    二度目以降はパスするため。
    性能のため、条件分岐は避けたい。

  - 参考：初回実行時のみの処理
    - クラスの初期化
      - Static変数アクセス、staticメソッド呼び出し、インスタンス生成
    - インスタンス生成時の諸チェック
      - Interfaceやabstractクラスではないか？
        もしそうならIllegalAccessErrorをthrow
    - Interface呼び出し時の呼び出し先解決結果チェック
      - メソッドが見つからなければ
        IncompatibleClassChangeErrorをthrow

自己書き換えの例

shuJIT に実装した
手法 その１

- 生成しておく機械語命令列:

nop
nop
初回実行時のみの処理
「nop」を「jmp done」に書き換え
done: ……

- 一度実行された後:

不要な処理を
スキップ

jmp done
初回実行時のみの処理
「nop」を「jmp done」に書き換え
done: ……

自己書き換えの例（余談）

- 書き換えのためには、その時点のプログラムカウンタ(PC)の値を把握する必要がある。
  - 書き換え対象の位置を、PCからの相対アドレスで特定するため。

- x86 では PC は汎用レジスタのようには読み出せない。どうする？

自己書き換えの例（余談）

- プログラムカウンタの取得

  E8 00 00 00 00　　call 次のpopl命令
  XX　　popl レジスタ

- レジスタに popl 命令自身を指すプログラムカウンタが入る。

自己書き換えの例

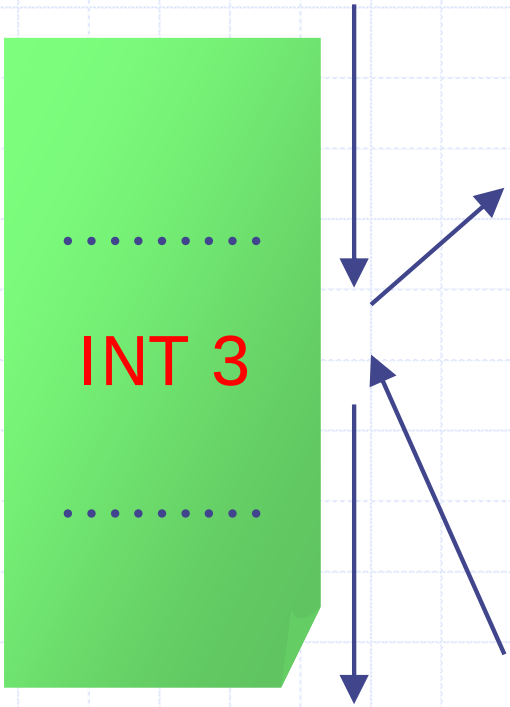
shuJIT に実装した
手法 その２

- ソフトウェア割り込みを使う方法
- 例えば x86 の INT 3命令
  - SIGTRAPを発生させる１バイト命令 (0xCC)。
    デバッガ用なので、デバッガを使いにくくなるという問題も。
  - INT $0x10 といった２バイト命令でもよいが、１バイト命令なら書き換え時にatomicity（後述）を気にせずに済む。

コード生成時:
生成したコードを

「INT 3」で上書き
しておく。

※本当は、シグナルハンドラ中でのシグナル発生を防ぐため、トランポリンを生成し、ハンドラからそこに戻る。



シグナルハンドラ:

void signal_handler(…) {

・※初回限定の処理を行う

・INT 3を本来のコードで上書き

・シグナルコンテクスト中の
プログラムカウンタを設定

}

○|￣|＿

- ソフトウェア割り込みを使うことで
  コード生成量を減らすことができた。
- しかしメモリ消費量は減らなかった。
  - コードの量:
    - ジャンプ上書き　1516 KB
    - ソフトウェア割り込み　1408 KB
    - 減った
  - 例外表まで含めたメモリ消費量
    - ジャンプ上書き　1682 KB
    - ソフトウェア割り込み　1773 KB
    - 増えた ＿|￣|○
- 例外表：
  - シグナル発生時に、Javaバイトコード上のプログラムカウンタを得るための表。
  - ソフトウェア割り込みを活用するために必要になった。

Core Wars ふたたび

- Core Wars
  - メモリ書き換えで他者のスレッドを妨害する。
- マルチスレッドな普通のプログラム
  - 実行時書き換えの際に、
    他のスレッドを妨害してはいけない。

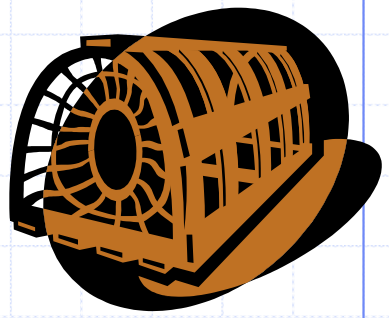
- 妨害:
  - 書き換え途中の新旧混ざったコードを実行してしまう。
  - 命令パイプラインに書き換え前のコードが残っており、新旧混ぜて実行してしまう。
  - ……

他のスレッドを妨害しないために

- 一貫した状態のメモリを見る、見せるために。

- Atomic に書き換える
  - Atomic: 他のスレッドからは、書き換え前か、書き換え後か、どちらかの状態しか見えないこと。
  - 具体的には
    - 1バイト単位のアクセス (486以降)
    - 16/32ビット境界に整列された16/32ビット単位のアクセス (486以降)
    - 64ビット境界に整列された 64ビット単位のアクセス (P5以降)
    - 整列されていない、キャッシュライン (32バイト) に収まるアクセス (P6のみ)
  - Atomicにメモリを書き換える方法が、プロセッサごとに用意されている。
    - x86 だと XCHG命令など。
- ロック
  - モニタ、セマフォ、…
  - アプリ側に（コード書き換え用）ロックを取得させるというのは、性能上あり得ない。
- 一時停止
  - 差し支えない位置で、他のスレッドを止める。やっかい。
- 罠

罠



- Core Wars の Imp Pit（小鬼地獄）のように (???)
  - MOV #0 -1
    JMP -1
  - 1番地手前に即値0を書き込み続ける。

- 書き換え対象領域の先頭で、他のスレッドを足止めする
  1. 書き換え対象の先頭２バイトに、
     トラップ EB FEを（atomicに）書き込む。
     EB FE: 自身へのジャンプ命令
  2. ３バイト目以降を書き換える。
  3. 先頭２バイトを書き換える。

マルチスレッドプログラミングでは

Core Wars的なテクニックがリアルに使える！

バイナリアンの時代がやってきた！

というのはたぶん言いすぎ。

ところで、ハックリタイ症候群って何？