情報処理学会論文誌　Vol.62　No.4　1051–1055 (Apr. 2021)

テクニカルノート

# 32 bit UNIXシステムの2038年問題に対するプログラム修正法の提案

大江 秀幸[1,a)]　松下 誠[2,b)]　井上 克郎[2,c)]

受付日 2020年8月2日, 採録日 2020年10月6日

**概要**：UNIXベースの32 bitシステムでは，2038年に時刻情報のオーバフローを起こすことが知られている．時刻情報が64 bitに拡張されたシステムでは特に問題とはならないが，すでに32 bitシステムで開発されたプログラムで，かつ2038年以降も稼働する場合では問題となる．本稿ではこのようなシステムを2038年以降も稼働させるための1つのプログラムの修正方法を提案する．

**キーワード**：2038年問題，32 bitシステム，UNIX，プログラムスライシング

# A Solution to the Year 2038 Problem for 32-bit Unix Systems

Hideyuki Oe[1,a)]　Makoto Matsushita[2,b)]　Katsuro Inoue[2,c)]

Received: August 2, 2020, Accepted: October 6, 2020

***Abstract:*** It is known that the time information digit overflow will occur in 2038 on UNIX-based 32-bit systems. This problem could be solved if the time information could be expanded from 32 bits to 64 bits, but it is a challenge to keep the time information 32 bit and to run even after 2038. In this paper, we propose a program modification method to solve the problem.

***Keywords:*** the year 2038 problems, 32 bit system, UNIX, program slicing

## 1. はじめに

32-bitのUNIXベースのOSでは，時刻情報をtime_t型と呼ぶ32ビット符号付き整数データとして1970年1月1日0時0分0秒（UTC）を起点（epoch）とした経過秒で管理している [1]．1970年からおよそ68年後の2038年にデータの最大値を超えて桁あふれを起こすため [2]，アプリケーションの動作にさまざまな影響を及ぼす恐れがある [3]（以下2038年問題とする）．

この問題への対策として，筆者らはepochを1970年から1998年に変更し，桁あふれの発生を2038年から2066年に遅らせる方法を提案した [4]．この提案は実際に20年間動作保証する組込みシステム開発に適用され，2020年現在，市場で稼働している．

しかしこの変更を実現するためには，時刻に関わるライブラリ関数，それぞれの入出力値や一時的に時刻情報を保存する変数のすべてをソースコードから抽出し，それぞれ修正対象箇所かどうかを人手で確認する必要があった．

本稿では，ソースコードからこの修正箇所を効率良く特定する手順を一般化した．まず，開発対象から修正箇所を抜き出すためにプログラムスライシング手法 [5], [6] を適用する．その後，得られたスライスより，対象とするアプリケーションの動作条件，制約事項により修正箇所を絞り込む．本稿では，実際にこの手順をFreeBSD上の3つのコマンド（date，stat，touch）に適用し，その効果を確認した．

1　関西デジタルソフト株式会社
Kansai Digital Soft Co., Ltd., Osaka 530–0005, Japan
2　大阪大学
Osaka University, Suita, Osaka 565–0871, Japan
a)　hideyuki.oe@digitalsoft.co.jp
b)　matusita@ist.osaka-u.ac.jp
c)　inoue@ist.osaka-u.ac.jp

© 2021 Information Processing Society of Japan

## 2. 提案手法

### 2.1 問題点と課題

2038年問題は，32 bitシステムにおいて32 bitの符号付き整数型で扱う時刻管理情報が桁あふれを起こすことが原因で起こる．この問題への対処法として，① time_t型を64 bit化する，② time_t型を32 bit符号なし整数型に変更する，③ time_t型変数の評価箇所で桁上がりを考慮して32 bitに収める，④ epochを変更し，影響を受ける部分をラッパ関数等で修正するという4つの方法が考えられた[4]．

筆者らが行った既報の提案では，開発量や修正による影響範囲の観点から，④を採用した．この選択により，標準ライブラリやデバイスドライバ，OS等の変更をいっさい行わず，アプリケーション部の修正だけで対応が可能であった．

本稿ではこの修正手順を明確化，一般化し，修正対象の関数や命令を効率良く見つける手法を提案する．

### 2.2 概要

修正対象を効率良く見つける手法を導くため，FreeBSDのツールを対象に2038年問題の修正方法を検討した．対策方針は既報の開発と同様に，時刻情報のオーバフロー回避のため，epochの起点をちょうど28年後ろにずらして，1998年1月1日0時0分0秒（UTC）に変更することとした．また修正による影響範囲を小さくするため，2.1節で示したものと同様の修正方針とし，デバイスドライバ，標準ライブラリやOS部の改変は行わないこととした．

**図1**に，手順を示す．

<u>ステップ ①　スライシング基準の選定</u>

time_t型変数が該当プログラムの実行系列で最後に利用される箇所を見つけて，スライシング基準とする．実行系列での最後の利用箇所については，対象によっては一意に決まらない場合もあるため，コードレビュー等で最後に利用される可能性のある命令をすべて抽出する．

<u>ステップ ②　静的スライスの抽出</u>

求めたスライシング基準より，データ依存行，制御依存行を求めて，静的スライスを得る．ただし，OS部，標準ライブラリ，デバイスドライバ等の関数の呼び出し文については，入出力の仕様からステップ③と同様の判断基準で，静的スライスとして含めてさらに伝搬させるか否かを判断する．静的スライスの抽出には，CodeSonar [7] 等のツールを用いると効率的だが，本研究では対象が比較的小規模なので目視により行った．

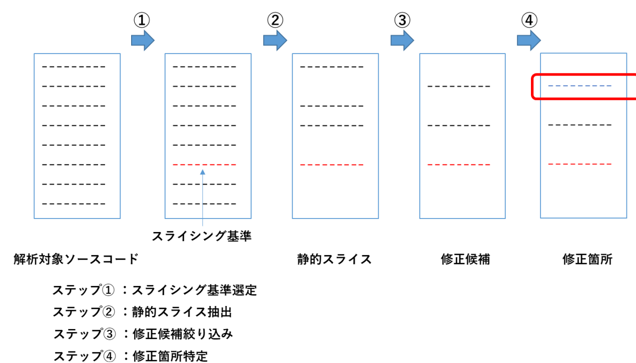


図1　修正箇所の特定手順

**Fig. 1** Process of identifying modification parts.

<u>ステップ ③　修正候補の絞り込み</u>

ステップ②で得られた静的スライスの各文（スライス文と呼ぶ）から修正箇所を絞り込むため，2038年問題に関係しないスライス文を対象から除外する．除外する箇所は，呼び出し関数，アプリケーションの機能，処理内容の3つの視点で選定する．

（1）呼び出す関数の機能

時刻情報を扱わない関数呼び出しを持つスライス文は，対象外とする．たとえば実行制御に関わっている場合でも，時刻情報を扱わない関数は対象外とする．

（2）アプリケーションの機能

アプリケーション中の特定の機能が，直接的，間接的に時刻情報に影響しない場合には，その機能を実現しているスライス文を修正対象外と判断する．たとえばコマンドライン引数をとるアプリケーション中で，時刻情報の更新に影響しない引数を処理するスライス文は，修正対象外とする．

（3）スライス文の処理内容

time_t型変数への，符号付き整数型32 bitを超えない定数の代入箇所は，修正対象外と判断する．たとえばtime_t型変数への初期値0の代入箇所は，修正対象外とする．

<u>ステップ ④　修正箇所の特定とその修正方針</u>

修正候補の各文に対して，修正箇所の特定を以下の手順で行う．

（1）標準ライブラリ関数の呼び出し

time_t型変数に関連するライブラリ関数は，ラッパ関数から間接的に呼び出すように変更する．関数mktimeを例に示す．mktimeは年月日と時刻の情報を表すstruct tm型の値の引数をとり，epochからの経過秒であるtime_t型の値を返す関数である．アプリケーションからmktimeを直接呼び出す代わりに，mktimeのラッパ関数（関数wrapper_mktimeとする）を用意し，wrapper_mktime内で，epochの起点変更分（28年）を年の情報から減算したうえで，mktimeを呼び出すようにする．

（2）時刻情報を評価する箇所

時刻情報を直接評価する命令がある場合は，epoch変更前の時刻情報を扱う箇所か，epoch変更後の時刻情報を扱う箇所かを識別する．そのうえで，epoch変更後の時刻情報を扱う場合には，起点変更分（28年）を考慮した評価に改める．

### 2.3 修正候補の絞り込みの具体例

FreeBSDのコマンドtouchを例に，本手法のステップ③

© 2021 Information Processing Society of Japan

```
332         if ((*p == '.' || *p == ',') && isdigit((unsigned char)p[1])) {
333                 p++;
334                 val = 100000000;
335                 while (isdigit((unsigned char)*p)) {
336                         tvp[0].tv_nsec += val * (*p - '0');
```

図 2 対象外の関数呼び出しの例

**Fig. 2** An example of non-target function call.

```
79      Aflag = aflag = cflag = mflag = timeset = 0;
80      atflag = 0;
81      ts[0].tv_sec = ts[1].tv_sec = 0;
82      ts[0].tv_nsec = ts[1].tv_nsec = UTIME_NOW;
83
84      while ((ch = getopt(argc, argv, "A:acd:fhmr:t:")) != -1)
85         switch(ch) {
86         case 'A':
87            Aflag = timeoffset(optarg);
88            break;
```

図 3 アプリケーション機能による確認例

**Fig. 3** An example of confirming application feature.

である修正候補の絞り込みについて具体的に示す．

(1) 呼び出す関数の機能

関数 main から呼び出す関数の中には，2038 年問題の解決に無関係な関数もある．このような関数の例として，**図 2** に main から呼び出す関数 stime_darg の抜粋を示す．332 行目，および 335 行目で関数 isdigit を呼び出しているが，isdigit は入力の文字列が数値かどうかを評価する関数であり，時刻の情報は扱わない．よって，isdigit の呼び出し部分および isdigit 自体は修正対象外とする．

(2) アプリケーションの機能

**図 3** に関数 main の抜粋を示す．87 行目の関数 timeoffset は，-A オプションを指定した際に呼ばれる関数であり，経過秒を返す関数である．時刻情報ではあるが，年単位の情報を扱わないため，修正対象から除外する．

(3) スライス文の処理内容

図 3 の 79 行目から 82 行目のように，固定の初期値を変数に代入する処理では，年情報のオーバフローは起こらないため，修正対象から除外する．

## 2.4 修正箇所の特定の具体例

2.3 節と同様に，本手法のステップ ④ である修正箇所の特定について，具体的に示す．

(1) 標準ライブラリ関数の呼び出し

**図 4** に関数 main より呼び出される関数 stime_arg1 を示す．272 行目に，当該関数内で最後に時刻を扱う標準ライブラリ関数である mktime がある．この関数については，**図 5** のようにラッパ関数 wrapper_mktime を用いて epoch の起点を変更する．これにより，main では epoch を修正した時刻を扱えるようになる．このように各ライブラリ関数について，年でのオーバフローの発生が起こりうるかどうかを確認し，オーバフローの可能性がない場合には修正対象としない．

```
219  static void
220  stime_arg1(const char *arg, struct timespec *tvp)
221  {
     …
252               yearset = ATOI2(arg);
253               if (yearset < 69)
254                  t->tm_year = yearset + 2000;
255               else
256                  t->tm_year = yearset + 1900;
257            }
258            t->tm_year -= 1900;    /* Convert to UNIX time. */
259            /* FALLTHROUGH */
260         case 8:                    /* MMDDhhmm */
261            t->tm_mon = ATOI2(arg);
     …
271         t->tm_isdst = -1;        /* Figure out DST. */
272         tvp[0].tv_sec = tvp[1].tv_sec = mktime(t);
273         if (tvp[0].tv_sec == -1)
274            goto terr;
     …
281  }
```

図 4 関数 stime_arg1

**Fig. 4** Function stime_arg1.

```
219  static void
220  stime_arg1(const char *arg, struct timespec *tvp)
221  {
     …
271         t->tm_isdst = -1;        /* Figure out DST. */
272         tvp[0].tv_sec = tvp[1].tv_sec = wrapper_mktime(t);
273         if (tvp[0].tv_sec == -1)
274            goto terr;
     …
281  }
…    …
     static time_t
     wrapper_mktime(struct tm *ptm)
     {
        time_t ret;

        ptm->tm_year -= 28;
        ret = mktime(ptm);

        return ret;
     }
```

図 5 stime_arg1 の修正箇所

**Fig. 5** Modified parts of stime_arg1.

(2) 時刻情報を評価する箇所

図 4 に示す関数 stime_arg1 では，254 行目に t->tm_year に値を設定する命令がある．この命令が修正対象かどうかを検討する必要がある．この命令実行前には time_t 型の変数値に対して加減算を行っていない．時刻を修正していないため，yearset と比較する値も修正前のプログラムと同等であり修正対象外とする．

## 2.5 修正後の実行例

FreeBSD のコマンド touch，stat，date のソースコードを修正した．これらの実行結果を，**図 6**，**図 7** に示す．

修正前のコマンド touch で 2040 年 1 月 1 日 0 時 0 分のファイルを作成しようとすると，エラーが返る．修正したコマンド touch.new で 2040 年 1 月 1 日 0 時 0 分のファイルを作成した場合，ファイルの作成に成功し，同様に 2038 年問題に対応したコマンド stat.new でファイルの作成年月日を確認すると，2040 年 1 月 1 日 0 時 0 分となっており，正常に実行できたことが確認できる．

© 2021 Information Processing Society of Japan

```
ooe@FreeBSD:~/test % touch -t 204001010000 aaa
touch: out of range or illegal time specification: [[CC]YY]MMDDhhmm[.SS]

ooe@FreeBSD:~/test % touch.new -t 204001010000 aaa
ooe@FreeBSD:~/test % stat.new -F aaa
-rw-r--r-- 1 ooe ooe 0 Jan  1 00:00:00 2040 aaa
ooe@FreeBSD:~/test %
```

図 6 実行結果（touch, stat）
**Fig. 6** Execution result (touch, stat).

```
ooe@FreeBSD:~ % date
1992年 7月19日 日曜日 22時00分08秒 JST
ooe@FreeBSD:~ % date.new
2020年 7月19日 日曜日 22時00分11秒 JST
ooe@FreeBSD:~ % date -j 204001010000
date: nonexistent time
ooe@FreeBSD:~ % date.new -j 204001010000
2040年 1月 1日 日曜日 00時00分00秒 JST
ooe@FreeBSD:~ %
```

図 7 実行結果（date）
**Fig. 7** Execution result (date).

コマンド date.new の実行結果を図 7 に示す．date.new では，システム時刻をあらかじめ 1998 年起点の時刻に変更する（28 年分減算する）．修正前のコマンド date で現在時刻を表示させると，2020 年現在の 28 年前である 1992 年を表示する．epoch を修正した date.new では，期待どおりに 2020 年を表示する．また修正前の date で 2040 年 1 月 1 日 0 時 0 分に日付を変更しようとするとエラーが返り，修正した date.new では 2038 年を超える 2040 年に正しく変更できる．

## 3. 議論

2038 年問題に対応する際，修正コストを極力抑えるため，修正規模を小さく，またなるべく簡単に対応したい．既報のやり方で 2038 年問題に対応する場合，影響する可能性のある，プログラム中のすべての箇所を確認する．複雑なことは行わない代わりに，標準ライブラリ関数も含めた確認が必要だったため，確認の必要なプログラム規模は大きく，調査コストが大きくなった．調査範囲を絞り込むため，grep 等の正規表現を用いた検索手法の利用も考えられるが，int 型で宣言した一時変数に time_t 型の値を代入する場合や，ポインタ変数からエイリアスとしてアクセスする場合等には，修正対象箇所として抽出できない．

プログラムスライシングを用いて修正対象箇所を限定すれば，調査するプログラムの規模を抑えたうえで修正箇所を漏れなく抽出できる．しかし，対象とするプログラムによっては，プログラムスライシングでは修正候補箇所の絞り込みが不十分な場合があり，確認の必要なプログラム規模が大きくは減少しないおそれがある．

表 1 提案手法による調査規模の減少
**Table 1** Size reduction by the proposed method.

| コマンド | 元の行数* | 静的スライス結果 | 修正候補の確認行数 | 修正行数 | 追加行数 |
|---|---|---|---|---|---|
| Touch | 303 | 273 (10%) | 178 (41%) | 18 | 15 |
| Date | 871 | 564 (35%) | 258 (70%) | 11 | 35 |
| Stat | 771 | 565 (27%) | 249 (68%) | 3 | 10 |

*空行、コメント行除く．() 内は元の行数からの削減率．

そこで提案手法では，修正候補のスライス文からプログラムで 2038 年までの時刻情報を扱う箇所と，時刻に影響するライブラリ関数をコールする箇所に着目し，それ以外の箇所は修正対象外として確認対象から外すことにした．提案手法により，確認が必要な箇所が減少する様子を**表 1** に示す．表中，静的スライス結果は手順のステップ ② の結果，残るソースコードの行数であり，修正候補の確認行数はステップ ③ の結果，残るソースコードの行数である．

表 1 に示すとおり，コマンド touch で 41%，コマンド date で 70%，コマンド stat で 68%，確認が必要なプログラムの量を減少させることができた．より大規模なプログラムで，時刻情報に関わるプログラム規模の割合が相対的に下がる場合には，さらに大きな効果を見込むことができる．

また，touch と同様に date でも関数 mktime を用いており，touch で作成したラッパ関数がそのまま利用できる．時刻を扱うライブラリ関数のラッパ関数群をあらかじめ用意すれば，さらに修正コストを抑えた 2038 年問題の解決が期待できる．

一方で本手法は，プログラムスライシングを前提とした手法であるため，スライシング基準を適切に設定できないと，修正対象から漏れてしまう．シグナルハンドラ中の処理等は，スライシング基準選定時に別に確認する必要がある．

なお，Linux カーネル 5.6 では，time_t 型変数の 64 bit 化が行われている（2020 年 5 月現在）[8]．しかし，time_t 型の変数の 64 bit 化が行われた場合でも，32 bit システム下で作成したファイルの属性等を含めて考えた場合，単純には 2038 年問題は解決しない．特に稼働中のシステムに対するモダナイゼーションを検討する際，コスト面等の事情によりハードウェア更新やリビルド等の大規模な修正が選択できない場合には，対象アプリケーションのみを修正対象とする本手法が代替案となりうる．

© 2021 Information Processing Society of Japan

## 4. おわりに

32 bit Unix システムの 2038 年問題の解決法の 1 つを提案し，FreeBSD 上の 3 つのツールに適用して，その効果を確認した．2038 年問題への対応方法はいくつか考えられるが，特に対応コスト抑制の優先度が高い場合には，本稿で紹介した手法が，有力な選択肢となりうる．

### 参考文献


[1] time(3), FreeBSD 11.1-RELEASE Library Functions Manual (2017).

[2] Holzmann, G.J.: Out of Bounds, *IEEE Software*, Vol.32, No.6, pp.24–26 (2015).

[3] 鈴木孝知，中村建助：「西暦 2038 年問題」でトラブル相次ぐ，日経コンピュータ (2004-4-1)，入手先 ⟨http://tech.nikkeibp.co.jp/it/members/NC/ITARTICLE/20040325/1/⟩（参照 2020-07-19）．

[4] 大江秀幸，安藤友康，松下　誠，井上克郎：組込み機器開発における 2038 年問題への対応事例，デジタルプラクティス，Vol.10, No.3, pp.588–602 (2019).

[5] 下村隆夫：プログラムスライシング技術と応用，共立出版 (1995).

[6] Weiser, M.: Program Slicing, *IEEE Trans. Softw. Eng.*, Vol.SE-10, No.4, pp.352–357, DOI: 10.1109/TSE.1982.5010248 (1984).

[7] CodeSonar, available from ⟨https://www.grammatech.com/codesonar-cc⟩ (accessed 2020-07-25).

[8] Vaughan-Nichols, S.J.:「2038 年問題」への対応進む Linux，入手先 ⟨https://japan.zdnet.com/article/35149495/⟩（参照 2020-07-19）．

**大江 秀幸**　（学生会員）

1991 年関西大学工学部金属工学科卒業．同年より NEC テレコムシステム（現，NEC 通信システム）株式会社，2002 年より松下 AVC マルチメディアソフト（現，パーソル AVC テクノロジー）株式会社，2018 年より関西デジタルソフト株式会社に所属．主に組み込みソフトウェア開発に従事．大阪大学大学院情報科学研究科博士課程に在学中．技術士（情報工学部門）．

**松下 誠**　（正会員）

1998 年大阪大学大学院基礎工学研究科博士後期課程修了．同年同大学基礎工学部情報工学科助手．2002 年大阪大学大学院情報科学研究科コンピュータサイエンス専攻助手．2005 年同専攻助教授．2007 年同専攻准教授．博士（工学）．リポジトリマイニング，プログラム解析の研究に従事．日本ソフトウェア科学会，ACM 各会員．

**井上 克郎**　（正会員）

1984 年大阪大学大学院基礎工学研究科博士後期課程修了（工学博士）．同年大阪大学基礎工学部情報工学科助手．1984～1986 年ハワイ大学マノア校コンピュータサイエンス学科助教授．1991 年大阪大学基礎工学部助教授．1995 年同学部教授．2002 年より大阪大学大学院情報科学研究科教授．ソフトウェア工学，特にコードクローンやコード検索等のプログラム分析や再利用技術の研究に従事．本会フェロー．


© 2021 Information Processing Society of Japan