Vol.2021-DPS-186 No.12
Vol.2021-CSEC-92 No.12
2021/3/15

Rust 言語用動的メモリアロケータの検査フレームワークの提案

尾崎 純平[1,a)]　高野 祐輝[1,b)]　宮地 充子[1,2,c)]

概要：プログラミング言語の問題点がソフトウェアの脆弱性を導くことが多く，安全なプログラミング言語の確立は重要である．メモリ安全性とはぶら下がりポインタやオーバーラン等のメモリに関する問題を未然に防ぐことを意味する．C 言語や C++言語はメモリ安全性は担保されておらず，メモリ安全性の担保が可能な Rust が代替言語として注目されている．Rust のメモリ安全性に注目して隔離環境を実装した OS がいくつか存在するが，OS 実装の際はメモリアロケータを別途構築する必要があり，隔離環境の実装はメモリアロケータに依存している．しかし現時点でメモリアロケータの動作検証は充分に行われていない．本論文ではメモリアロケータの動作を検査するフレームワークの提案を行う．フレームワークはメモリの確保/開放の動作を対象とし，動作範囲と動作時間の 2 点について検査する．またフレームワークを用いて Rust で制作された既存のメモリアロケータの動作について検証し，パフォーマンスの比較を行った．

キーワード：メモリ安全性, メモリアロケータ

A Test Flamework of Dynamic Memory Allocator for Rust

Abstract: The establishment of secure programming languages is important because problems with programming languages often lead to software vulnerabilities. Memory safety is the ability to prevent problems related to memory such as dangling pointers and buffer overruns. The C and C++ languages do not guarantee memory safety, and Rust, which can guarantee memory safety, has been attracting attention as an alternative language. Although there are several operating systems that have implemented isolated environments focusing on the memory safety of Rust, the implementation of the isolated environment depends on the memory allocator, which must be prepared by the user. At present, however, the memory allocator has not been sufficiently verified. In this paper, we propose a framework for checking the behavior of memory allocators. The framework targets the memory allocation/deallocation operations, and is inspected for two points: operation range and operation time. We also verified the behavior of existing memory allocators created with Rust using the framework, and compared their performance.

Keywords: memory safety,memory allocator

1. はじめに

1.1 背景

伝統的に C 言語で行われていたシステムの開発を代替する言語として Rust が急速に人気を集めている [1]．安全でない言語を用いたプログラミングにおいて生じる脆弱性の 3 分の 2 は安全な言語の使用だけで除去することができると考えられており [2]，安全とされる言語として Rust を用いた環境構築が進められている [3].

現在，数々の Rust を用いた OS が構築されている [4], [5] が，これらの OS が使用しているメモリアロケータはメモリ安全性を担保していないため，OS の隔離環境も同様にメモリ安全性を担保することはできない．本研究ではメモリアロケータのメモリ安全性と性能を検査できるフレームワークを構築することを目的とする．検査対象は Rust で制作された既存のアロケータ [4], [5] であり，検査内容は

[1] 大阪大学
Osaka University
[2] 北陸先端科学技術大学院大学
Japan Advanced Institute of Science and Technology
[a)] ozaki@cy2sec.comm.eng.osaka-u.ac.jp
[b)] ytakano@cy2sec.comm.eng.osaka-u.ac.jp
[c)] miyaji@comm.eng.osaka-u.ac.jp

ⓒ 2021 Information Processing Society of Japan

アロケータの動作範囲と動作時間である．

本論文の構成は次の通りである．2 章では本論文を読む際に必要な基礎的な知識を準備として記載する．3 章では本論文に関する既存研究について記載する．4 章ではメモリアロケータの検査用フレームワークの設計原理について記述する．5 章では 4 章の設計原理に基づき検査用フレームワークを提案する．6 章では本フレームワークを用いて既存のアロケータの性能を比較/評価する．7 章では本研究のまとめと今後の課題について述べる．

2. 準備

2.1 メモリ安全性

メモリ安全性とは，メモリに関するバグから保護されている状態を指す．メモリに関するバグには次のような種類が挙げられる．

バッファオーバーラン 指定範囲外で入力が行われた際に，隣接するオブジェクトのデータに干渉し破壊する現象

競合状態 共有メモリへ同時に読み込みや書き込みが行われる現象

ぶら下がりポインタ ポインタの参照先となるオブジェクトが破壊され, 無効な値を指すようになる現象

2.2 メモリアロケータ

メモリアロケータとは，プログラムが使用するヒープ領域を動的に割り当てる機能であり，C 言語では malloc 関数，new 関数がこれに該当する．ここでは，本研究で検査するアロケータについて説明する．

2.2.1 Buddy allocator

Buddy allocator は 1963 年に開発されたアルゴリズムであり，メモリ領域を 2^L 個のブロックに分割して割り当てを行う．2^L の L をオーダーと呼び，要求されたメモリの大きさに応じてメモリを 2 等分することにより確保/開放の時間を小さく抑えることができる．しかし，要求されたメモリ領域がブロックの大きさより遥かに小さい場合，メモリ領域のほとんどが無駄に確保されてしまう．

2.2.2 Slab allocator

Slab allocator は前述した buddy allocator の欠点を解決するために開発されたアロケータであり，単一のブロック内に小さなメモリ領域を複数割り当てることが可能になっている．そのためにスラブという同サイズのオブジェクトを複数個格納するためのメモリ領域を用いており，同サイズのオブジェクトを管理するスラブをまとめたものをキャッシュと呼ぶ．スラブはオブジェクトの埋まり具合に応じて Full，Partial，Empty の 3 種類の状態に分類され，スラブの使用状況を管理することができる．スラブが扱うオブジェクトの大きさは 13 種類に分けられており，最小で 32byte まで取り扱っているため，無駄に確保してしまうメモリ領域を大幅に削減することができる．

2.2.3 Linked list allocator

Linked list allocator は空のメモリ領域の位置と大きさを管理するリストを用いたアロケータである．メモリの確保が要求された際にはリストから十分な大きさの空メモリを探索し，最初に見つかった領域を確保する．メモリ開放が要求された際は開放した領域をリストに登録し，隣接している空メモリが存在している場合は合体させる．

2.3 フレームワーク構築に用いる Rust 言語の機能

ここでは，フレームワーク構築の際に実際に使用した Rust 言語の機能について説明する．

A.now() 関数/elapsed() 関数

now() 関数/elapsed() 関数は，それぞれ Instant という型を用いる関数である．Instant はある二つの瞬間から経過時間を測定する型であり，now() 関数は呼び出された瞬間に対応した Instant を作成し，elapsed() 関数は Instant が作成されてから経過した時間を返す関数である．よって，now() 関数を開始部分，elapsed() 関数を終了部分に呼び出すことで時間の測定が可能になる．

B. スライス

スライスとは連続した要素への参照のことであり，最初の要素へのポインタとそこからアクセスできる要素数を持っている．スライスには次の 2 種類が存在する．

共有スライス & [T] で記述される型．複数のスレッドで共有することができるが，要素の変更ができない

可変スライス & mut [T] で記述される型．他のスレッドと共有はできないが，要素の読出しや変更ができる

C. カスタムアロケータ

Rust はメモリアロケータを独自で用意したものに置き換えて動作させることができ，これをカスタムアロケータと呼ぶ．カスタムアロケータは原則として alloc/dealloc の 2 つの関数を用意する必要があり，#[global_ allocator] で定義することによりアロケータの置き換えができる．

ソースコード 1 カスタムアロケータ

```
1  use std::alloc::{GlobalAlloc, Layout, alloc};
2  struct MyAllocator;
3  unsafe impl GlobalAlloc for MyAllocator {
4      unsafe fn alloc(&self, layout: Layout) ->
       *mut u8 {
5      // ここにalloc 部分の実装
6      }
7      unsafe fn dealloc(&self, ptr: *mut u8,
       _layout: Layout) {
8      // ここにdealloc 部分の実装
9      }
10 }
11 #[global_allocator]
12 static A: MyAllocator = MyAllocator;
```

まず，1 行目でカスタムアロケータの作成に必要な関数を用意し，3 行目でカスタムアロケータを宣言する．次に 5〜12 行目でカスタムアロケータの設定を行う．6〜8 行目部分で alloc，9〜11 行目部分で dealloc の動作を設定する．最後に 13〜14 行目で global allocator に設定することによりカスタムアロケータによってメモリアロケーションを行うことができるようになる．

3. 既存研究

本章では Rust を用いて開発された OS に関する既存研究について述べる．

3.1 RedLeaf [4]

RedLeaf は言語の安全性がオペレーティングシステム (OS) の編成に対してどこまで影響するかを調査するために，Rust でゼロから開発された新しい OS である．最大の特徴としては障害の隔離をドメイン単位で行うことができるという点が挙げられる．RedLeaf はドメインごとに確保されるヒープ (プライベートヒープ) とは別に共有ヒープを設け，プライベートヒープは基本的に共有ヒープとのみデータのやり取りを行う．この構造によりプライベートヒープのデータは他のプライベートヒープに干渉することを防ぎ，障害が発生した際には対象のプライベートヒープとそのヒープが干渉している共有ヒープ内のデータのみ解放の処理を行えばよくなる．この構造を実装するために RedLeaf は Box<T>型という Rust 側で用意された所有権という特徴を強制する型を用いており，これによって二つ以上のヒープが同じアドレスを参照することを防ぐことができる．

3.2 Theseus [5]

Theseus は OS のコードの中で状態の流出をどの程度回避できるかを調べるためにゼロから開発された OS である．Theseus の最大の特徴は，OS がセルという単一の構成要素で作られているという点である．セルはソフトウェアで定義されたモジュール性の単位であり，コンパイル時には単一のオブジェクトファイルとして存在する．実行時にはまずセルオブジェクトファイルを見つけて解析し，そのセクションをメモリにロードする．そして他のセルとの依存性を検証し，必要に応じて欠落しているセルを再帰的にロードすることによって構築されている．Theseus には “セル同士に永続的な境界線を設ける” という原則があり，これはセルを 1 つずつ動的に確保してセルが確保するメモリ領域に間隔を開けることにより実装している．

4. 提案

4.1 設計原理

本章ではメモリアロケータの検査装置がどうあるべきかを述べる．メモリアロケータの性能の評価基準として次の 2 つが挙げられる．

- 指定された範囲の確保，開放の動作が正常かどうか
- 実行時間はどれほどの長さなのか

1 つ目はメモリアロケーションが安全に実行できているのかどうかを，2 つ目はメモリアロケータが処理速度に与える影響を評価する．よって，検査装置には上の 2 つの性能を検査する機能を入れる必要がある．

またアロケータにも様々な種類が存在しており，それぞれ入出力の形式は異なるが，検査装置としてはアロケータごとにプログラムを書き換えるより可能な限り共通のコードで稼働できることが望ましい．

4.2 検査手法

メモリアロケータの検査フレームワークの作成において，本研究ではカスタムアロケータを用いる．具体的には，カスタムアロケータの alloc/dealloc 関数を以下の手順で実行するように設計する．

alloc 関数　時間の測定開始→検査対象のアロケータの実行→時間の測定終了→色の検査→色付け

dealloc 関数　色の検査→色付け→時間の測定開始→検査対象のアロケータの実行→時間の測定終了

また、色の検査/色付け/時間の測定をそれぞれ次の考えのもとに設計する．

4.2.1 動作範囲の検証

メモリの確保/開放が正常に動作しているか否かを判断する手段として，本研究では疑似的にメモリ領域を色付けする．alloc 関数によって確保した領域を赤，dealloc 関数によって解放した関数を青と色付けした場合，alloc 関数，dealloc 関数の動作はそれぞれ次の条件を満たせばよい．

alloc 関数　確保の対象となるメモリ領域の色が赤以外であればよい

dealloc 関数　開放の対象となるメモリ領域の色が赤であればよい

本研究では赤=1，青=2 とおき，スライスを用いて対象のアドレスに値を入力することで色付けを行う．この色付けにより，検証は次のように行える．

alloc 関数　確保の対象となるメモリ領域の値が 1 以外であればよい

dealloc 関数　開放の対象となるメモリ領域の値が 1 であればよい

また，色の検査は alloc は動作後，dealloc は動作前に実行し，色付けは alloc/dealloc の動作後に実行する．

4.2.2 動作時間の検証

alloc/dealloc の動作にかかる時間を計測するために，本研究では計測開始/計測終了に now() 関数と elapsed() 関数を用いる．注意点として，検査装置自体の動作時間が影響すると正確な値を得ることができなくなるので now() 関

数は色の検査より後に，elapsed() 関数は色付けより前に行う．

4.3 検査指標

上で述べた機能を用いてアロケータの評価を行うため，2 つの検査指標を設定する．

MODE size MODE size は要求するメモリ領域の大きさによって動作がどのように影響するかを検査する機能である．$1+2^i$(i=1,2,3,...) の大きさで i を変更しながら alloc/dealloc の動作と検査を行う

MODE loop MODE loop は alloc/dealloc の動作が動作回数によって影響するかどうかを検査する機能である．要求するメモリ領域の大きさを 1000byte とし，alloc/dealloc 動作を 1000 回行う．alloc/dealloc 動作が異常であった回数を記録し，全て正常に動作しているのか否かを検査する．

4.4 本研究で検査するアロケータ

次に，本研究で検査するアロケータを列挙する．

linked-list-allocator [6] linked list allocator の動作を Rust で実装したアロケータ

buddy_system_allocator [7] buddy allocator の動作を Rust で実装したアロケータ．オーダーの最大値を 32 としている．

buddy_and_slab_allocator [8] 64KiB 未満のサイズで slab allocator，それ以上で buddy allocator の動作を行うアロケータ．

5. 実装

本章では，4 章で述べた機能を Rust 上で実装した工程をソースコードを用いて解説していく．

5.1 検査装置の構造

5.1.1 動作範囲の検査

alloc/dealloc の動作が正常に動作しているかを検査するフレームワークを実装した．

カスタムアロケータ内部における色の検査方法についてソースコード 2 を用いて説明する．まず alloc で要求されたメモリ範囲の色情報をスライスを作成することによって取り出し，確保範囲の色が条件を満たしているかどうか (alloc の場合は赤色 (1) でないかどうか) を一つずつ検査する．検査後に全ての色が条件を満たしている場合は正常値として 0 を，そうでない場合は異常値として 1 をそれぞれ検査結果を記録するグローバル変数に返す．dealloc の検査は条件を赤色 (1) であるかどうかに変更して同様の操作を行う．

次に，main() 内部における色の検査結果の出力部分についてソースコード 3 を用いて解説する．1 行目で検査結果を記録するグローバル変数の値をもとに動作の正常/異常を判別し，結果に応じて 2 行目または 4 行目で標準出力に出力する．

ソースコード 2 色の検査 (alloc)

```
1  let checkalloc = unsafe { slice::
       from_raw_parts(ptr,layout.size()) };
2  let mut count = 0;
3  for i in 0.. layout. size(){
4      if checkalloc[i] != 1{
5          count += 1;
6      }
7  }
8  if count != layout. size(){
9      c_alloc = 1;
10 } else {
11     c_alloc = 0;
12 }
```

ソースコード 3 色の検査結果の出力 (alloc)

```
1  if unsafe{c_alloc} == 0{
2      println!("alloc is correct")
3  } else {
4      println!("alloc is incorrect")
5  }
```

ソースコード 4 色付け (alloc)

```
1  let paintred = unsafe {slice::
       from_raw_parts_mut(ptr,layout. size())};
2  for i in 0..layout.size() {
3      paintred[i] = 1;
4  }
```

ソースコード 5 動作時間の計測 (alloc)

```
1  let start = Instant::now();
2  let ptr = L.alloc(layout);
3  let end = start.elapsed();
4  t_secs_alloc = end.as_secs();
5  t_nanos_alloc = end.subsec_nanos();
```

また，alloc/dealloc の動作後の色付けにおけるカスタムアロケータ内部の動作についてソースコード 4 を用いて解説する．色付けは検査とは異なり値の変更を行うため，対象となる範囲の色情報を可変式スライスによって取り出す．そして検査時と同様に 1 つずつ移動しながら色を変更することにより色付けを完了させる．

5.1.2 動作時間の検査

alloc/dealloc の効率を比較するため，時間を検査するフレームワークを実装した．

カスタムアロケータ内部の動作についてソースコー

ソースコード 6 動作時間の出力 (alloc)

```
let mut tsa = 0;
let mut tna = 0;
.....................................................
tsa = unsafe{t_secs_alloc};
tna = unsafe{t_nanos_alloc};
println!("time for alloc = {}.{:08}s", tsa,
    tna);
```

ソースコード 7 MODE size

```
let num_of_alloc = 24;
println!("MODE size");
for i in 1..num_of_alloc+1 {
    if i > 1 {
    // dealloc の検査結果の出力部分
    }
    let base: i32 = 2;
    let exp: u32 = i;
    let a = 1 + base.pow(exp);
    let b = a as usize;
    println!("{}...allocate size = {}",i,a);
    let mut vec = Vec::<u8>::with_capacity(b);
    tsa = unsafe{t_secs_alloc};
    tna = unsafe{t_nanos_alloc};
    // alloc の検査結果の出力部分
    }
// 最後のdealloc 動作の検査結果の出力部分
}
```

ド 5 を用いて解説する．まず検査対象のアロケータの alloc/dealloc 動作の直前に now() 関数を呼び出すことによりインスタンスを作成する．次に alloc/dealloc 動作の直後に elapsed() 関数を呼び出すことにより now() 関数でインスタンスが呼び出されてからの経過時間を測定し，計測結果をグローバル変数に記録する．

次に main() 内部における動作時間の出力部分についてソースコード 6 を用いて解説する．時間の出力は alloc/dealloc の動作結果の出力後に行うが，出力の際に使用する関数がメモリアロケーションを必要とするためグローバル変数をそのまま出力しようとすると alloc/dealloc の検査結果の出力自体の dealloc 動作の時間を返してしまう．そこで事前にグローバル変数から時間を受け取る変数を作成し，出力前に時間の情報を記録することで本来の検査対象となる動作時間を返すようにしている．

5.2 検査装置の機能

5.2.1 MODE size

MODE size の構造についてソースコード 7 を用いて解説する．まず 1 行目で要求するサイズの最大値をループ回数を指定することによって決定し，7〜10 行目で 2^i の値を作成する．制作した値を用いて 12 行目に alloc 動作を行い，13〜14 行目でグローバル変数を用いて経過時間を記録した後に 15 行目の位置で alloc の検査結果を出力する．dealloc はループの動作時と終了時に行われるため，最後以外は 16 行目で 1 度以上ループ処理が行われたことを確認してから 5 行目の位置で dealloc の検査結果を出力し，最後のみ 17 行目の位置で出力を行う．

ソースコード 8 MODE loop(変数の設定部分)

```
let mut l_alloc = 0;
let mut l_dealloc = 0;
let looptime = 1000;
let l_size = 1000000;
let mut totalalloc_s = 0;
let mut totalalloc_n = 0;
let mut totaldealloc_s = 0;
let mut totaldealloc_n = 0;
```

ソースコード 9 MODE loop(alloc の検査部分)

```
totalalloc_s += unsafe{t_secs_alloc};
totalalloc_n += unsafe{t_nanos_alloc};
if unsafe{c_alloc} == 1 {
    l_alloc += 1;
}
```

ソースコード 10 MODE loop(alloc の出力部分)

```
if unsafe{l_alloc} == 0 {
    println!("alloc is correct");
} else {
    println!("alloc is incorrect");
}
```

5.2.2 MODE loop

動作前の設定についてソースコード 8 を用いて解説する．まず 1〜2 行目で alloc/dealoc が異常な動作を行った回数を記録する変数を設定する．次に 3〜9 行目でループを行う回数と要求するメモリ領域の大きさを指定し，5 行目〜8 行目で合計時間を記録する変数を設定する．

次に，alloc/dealloc の検査部分についてソースコード 9 を用いて解説する．まず 1〜2 行目で alloc/dealloc の動作時間を加算する．次に c_alloc/c_dealloc を用いて動作の正常/異常を判別し，異常値 (1) が確認された場合は l_alloc/l_dealloc のカウントを進めることにより異常の発生を記録する．

次に，alloc/dealloc の出力部分についてソースコード 10 を用いて解説する．異常な動作が一度も行われていない場合は異常な動作を確認した回数が 0 となることを利用し，l_alloc(l_dealloc) が 0 の場合は正常，0 以外であれば異常

ソースコード 11　MODE loop(検査部分)

```
1  println!("MODE loop(loop for {} times)",
       looptime);
2  for n in 1..looptime+1 {
3      // dealloc 動作後の処理
4          }
5      }
6      let mut vec = Vec::<u8>::with_capacity(
       l_size);
7      // alloc 後の処理
8  }
9  //最後のdealloc 後の処理
```

ソースコード 12　MODE loop(出力部分)

```
1  let ave_alloc_s = unsafe{totalalloc_s/1000};
2  let ave_alloc_n = unsafe{totalalloc_n/1000};
3  let ave_dealloc_s = unsafe{totaldealloc_s
       /1000};
4  let ave_dealloc_n = unsafe{totaldealloc_n
       /1000};
5  if unsafe{l_alloc} == 0{
6      println!("alloc is correct");
7  } else {
8      println!("alloc is incorrect");
9  }
10 if unsafe{l_dealloc} == 0 {
11     println!("dealloc is correct");
12 } else {
13     println!("dealloc is incorrect");
14 }
15 println!("average time for alloc = {}.{:08}s",
        ave_alloc_s, ave_alloc_n);
16 println!("average time for dealloc = {}.{:08}s
       ", ave_dealloc_s, ave_dealloc_n);
```

を文章で出力するように分岐を設定している．

上で説明した機能を省略すると，MODE loop のループ部分の構造はソースコード 11 のようになる．

出力部分についてソースコード 12 を用いて解説する．まず，1〜4 行目で alloc/dealloc の平均動作時間を算出する．次に 5〜9 行目/10〜14 行目で alloc/dealloc の検査結果を出力し，15〜16 行目で平均動作時間を出力する．

5.3　共有可能なコード

本研究で制作した検査装置のコードは全て検査対象となるアロケータの外部に組み込まれている．そのため，検査対象のアロケータを変更する際にはセットアップ部分となるコードのみを書き換えればよく，アロケータの動作方法や環境に応じてコードを作り直す必要が無くなる．

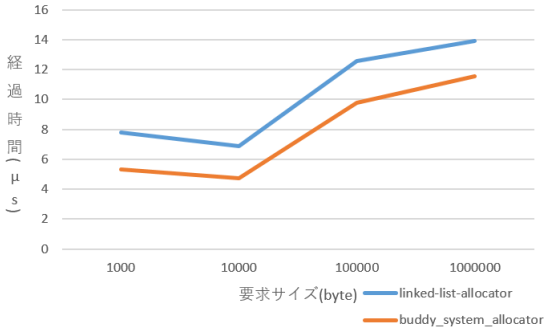

図 1　動作時間 (alloc, 最適化前,linked list,buddy)

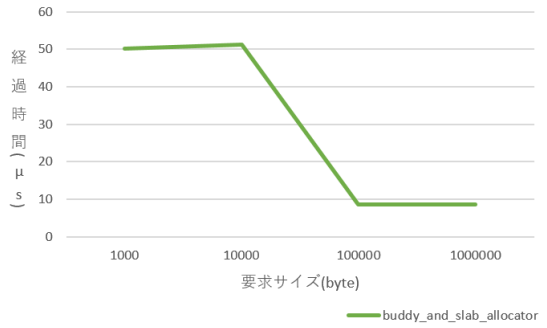

図 2　動作時間 (alloc, 最適化前,buddy_and_slab)

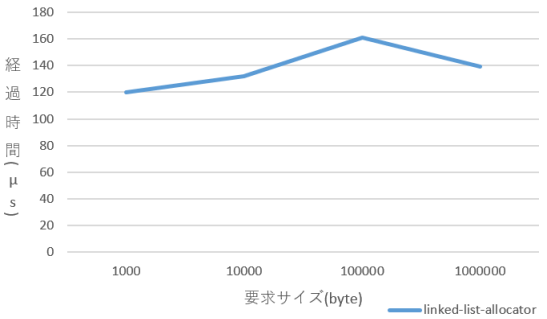

図 3　動作時間 (alloc, 最適化後,linked list)

6.　結果・考察

6.1　検査結果

5 章で実装した検査フレームワークを用いて linked-list-allocator,buddy_system_allocator, buddy_and_slub_allocator の検査結果について述べる．表 1 では alloc/dealloc の動作が MODE size と MODE loop で正常であったか否かをまとめている．また，表 2 と表 3 では要求サイズごとの alloc/dealloc の動作時間を MODE loop の平均時間を元にまとめている．

6.2　考察

まず表 1 より alloc/dealloc の動作範囲について検証する．linked-list-allocator と buddy_and_slab_allocator は全て正常値を出力したが，

表 1 alloc/dealloc の動作の正常/異常

アロケータ名	alloc(size)	dealloc(size)	alloc(loop)	dealloc(loop)
linked-list-allocator [6]	✓	✓	✓	✓
buddy_system_allocator [7]	× (i=25 以上)	× (i=25 以上)	✓	✓
buddy_and_slub_allocator [8]	✓	✓	✓	✓

表 2 動作時間 (alloc)

アロケータ名	size = 1000	size = 10000	size = 100000	size = 1000000
linked-list-allocator(最適化前)	7.77 μs	6.86μs	12.57μs	13.92μs
linked-list-allocator(最適化後)	120.00μs	132.00μs	161.00μs	139.00μs
buddy_system_allocator(最適化前)	5.32μs	4.73μs	9.77μs	11.57μs
buddy_system_allocator(最適化後)	11.0μs	17.00μs	7.00μs	8.00μs
buddy_and_slub_allocator(最適化前)	50.16μs	51.25μs	8.70μs	8.60μs
buddy_and_slub_allocator(最適化後)	6.00μs	8.00μs	10.00μs	7.00μs

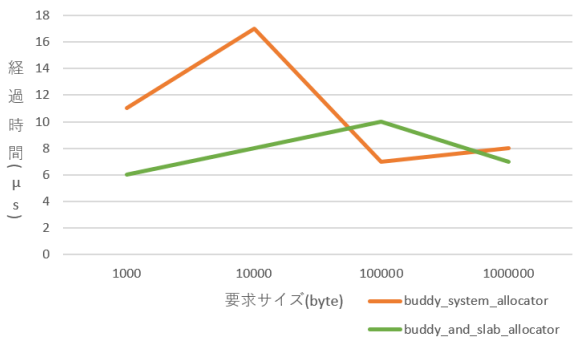

図 4 動作時間 (alloc, 最適化後,buddy,buddy_and_slab)

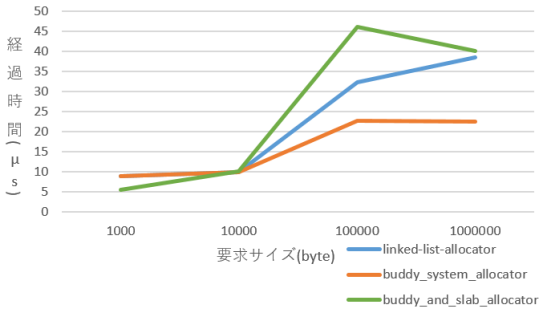

図 5 動作時間 (dealloc, 最適化前)

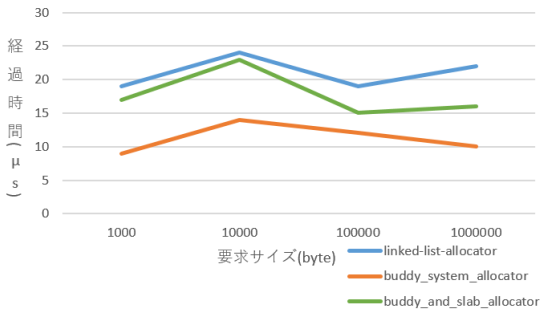

図 6 動作時間 (dealloc, 最適化後)

buddy_system_allocater のみ i=25 異常の alloc 要求時にエラーを出力した．これはアロケータ用に確保したサイズが 2^{26}byte であり，要求されるサイズが $1 + 2^{25}$byte と動作領域の半分を上回ったために領域の分割が一度も行えなくなったことが原因と推測される．よって，動作範囲の検査は全て正常に行われたことから alloc/dealloc の動作は全て正常であると考えられる．

次に表 2 と図 1〜図 4 より，alloc の動作時間について述べる．最適化前は linked-list-allocator と buddy_system_allocator の動作時間は要求されるサイズが大きくなるほど動作時間が長くなる傾向が見られ，buddy_and_slab_allocator は 10000byte 以下の動作時間が 100000byte 以上の動作時間より遥かに長くなった．最適化後については linked-list-allocator の動作時間は一転して遥かに長くなり，buddy_system_allocator はサイズが大きいほど動作時間が短くなり，buddy_and_slab_allocator は 10000byte 以下ではサイズが小さいほど，100000byte 以上ではサイズが大きいほど動作時間が短くなった．よって，linked-list-allocator は最適化との相性が悪く，buddy_system_allocator と buddy_and_slab_allocator は想定通りの動作が行われたため最適化との相性が良い．

次に表 3 と図 5〜図 6 より，dealloc の動作時間について述べる．全てのアロケータで最適化前はサイズが大きいほど長くなり，最適化後はサイズを問わずほぼ一定の値をとるという傾向が見られた．また最適化前後で比較するとサイズが小さいときは最適化前のほうが短く，サイズが大きいときは最適化後のほうが短くなるため，最適化によってサイズに依存する処理が無くなる代わりに追加された処理が共通で行われていると推測される．以上より，linked-list-allocator,buddy_system_allocator, buddy_and_slub_allocator は連続しない alloc/dealloc 動作について安全であると考えられる．また動作時間より，最適化の導入前は buddy_system_allocator が，最適化の導入後では buddy_and_slub_allocator が検査を行ったアロケータ内で最も動作効率が良いと考えられる．

ⓒ 2021 Information Processing Society of Japan

表 3 動作時間 (dealloc)

アロケータ名	size = 1000	size = 10000	size = 100000	size = 1000000
linked-list-allocator(最適化前)	8.98μs	9.99μs	32.36μs	38.52μs
linked-list-allocator(最適化後)	19.00μs	24.00μs	19.00μs	22.00μs
buddy_system_allocator(最適化前)	6.59μs	7.05μs	22.79μs	22.59μs
buddy_system_allocator(最適化後)	9.00μs	14.00μs	12.00μs	10.00μs
buddy_and_slub_allocator(最適化前)	5.55μs	10.16μs	46.00μs	39.97μs
buddy_and_slub_allocator(最適化後)	17.00μs	23.00μs	15.00μs	16.00μs

7. 結論

本研究ではメモリアロケータのメモリ安全性の検査と性能の比較を行うため検査フレームワークを作成し，既存のメモリアロケータの評価と比較を行った．まず，確保/開放を行うメモリ領域に数値を代入することによって疑似的に色を付け，色から動作の成否を判別する機能を実装した．また，メモリアロケータの動作時間を記録し，性能の比較要素として出力する機能も実装した．このフレームワークはアロケータの性能の比較に加え，アロケータが想定通りのアルゴリズムで動作しているかどうかをサイズごとの動作時間によりある程度の確認が可能となった．しかし，本フレームワークは alloc や dealloc を 2 回以上連続して要求する動作を想定しておらず，複数回の連続した alloc/dealloc 要求に関する影響の検査は行っていない．よってこれを今後の課題とし，アロケータの性能のより詳しい判別が可能なフレームワークの実装を行っていく．

謝辞 本研究の一部は文部科学省「Society5.0 に対応した高度技術人材育成事業成長分野を支える情報技術人材の育成拠点の形成 (enPiT)」さらに文部科学省の平成 30 年度「Society 5.0 実現化研究拠点支援事業」と「ソフトバンク株式会社」の助成を受けています．

参考文献

[1] Kulkarni, C., Moore, S., Naqvi, M., Zhang, T., Ricci, R. and Stutsman, R.: Splinter: Bare-Metal Extensions for Multi-Tenant Low-Latency Storage, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018* (Arpaci-Dusseau, A. C. and Voelker, G., eds.), USENIX Association, pp. 627–643 (2018).

[2] Cutler, C.: The benefits and costs of writing a POSIX Kernel in a high-level language, PhD Thesis, Massachusetts Institute of Technology, Cambridge, USA (2019).

[3] Levy, A., Campbell, B., Ghena, B., Giffin, D. B., Pannuto, P., Dutta, P. and Levis, P.: Multiprogramming a 64kB Computer Safely and Efficiently, *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, ACM, pp. 234–251 (2017).

[4] Narayanan, V., Huang, T., Detweiler, D., Appel, D., Li, Z., Zellweger, G. and Burtsev, A.: RedLeaf: Isolation and Communication in a Safe Operating System, *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, USENIX Association, pp. 21–39 (2020).

[5] Boos, K., Liyanage, N., Ijaz, R. and Zhong, L.: Theseus: an Experiment in Operating System Structure and State Management, *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, USENIX Association, pp. 1–19 (2020).

[6] Oppermann, P.: linked-list-allocator, https://github.com/phil-opp/linked-list-allocator.

[7] jiegec, C.: buddy_system_allocator, https://github.com/rcore-os/buddy$_s ystem_a llocator$.

[8] 高野祐輝: memalloc, https://github.com/ytakano/memalloc.

ⓒ 2021 Information Processing Society of Japan