東京大学
情報理工学系研究科 創造情報学専攻
修士論文

遅延スレッドのために連結されたスタック領域のメモリ効率の改善

Improvement of memory efficiency of linked stack regions for lazy threads

折笠 雄太朗
Yutaro Orikasa

指導教員　千葉 滋 教授

2018 年 1 月

概要

本提案では子スレッドのスタック領域の割り当てを高速に行うための工夫をし，またそれによって発生する欠点をうまく回避する方法を提案する．多数のスレッドが生成された場合にスタック領域確保によるメモリ消費が増えるが，それを抑えるためにスタック領域をリンクリスト化する方法が有用である．その中でも遅延スレッド (Lazy Threads) は，既存のスレッドのスタック領域の未使用部分を「盗み出し」て，新たに生成するスレッドの実行に割り当てることにより，さらにメモリ消費を抑える．本提案ではスタック領域の大きさを元の遅延スレッドより大きく割り当て，さらに子スレッドの終了時に発生してしまう大きな空き領域を任意のスレッドに再利用させる．これにより，新規メモリの割り当て回数が減り，実行速度が高速化する．また利用不可能なスタック領域の発生を抑えられる．本研究では gcc の split stack 機構を用いて遅延スレッド並びに提案手法を実装し，評価実験を行った．

Abstract

We propose a plan for allocating stack regions, and we propose an approach for avoiding the disadvantage of the plan. Memory consumptions increase in case of many threads. For suppressing them, the methods of linked stack region is useful. Lazy Threads is one of the methods. It lowers memory consumptions further by "stealing" unused part of existing stack regions and allocating it to a new created thread. In this proposal we allocate bigger stack region than the original lazy threads, and we make any threads being able to reuse the big free areas. Using these the frequency of new memory allocations gets lower, and the speed of the execution gets faster. And they suppress creation of unusable memory areas. We implemented lazy threads and propose using gcc's split stack and did evaluation experiment.

目次

第 1 章

序論

1.1 並列計算とスレッドの軽量化

大規模な科学計算問題が与えられたとき，問題を均等に分割し並列に処理し，処理速度が CPU コア数でスケールさせる理想的である．しかし例えばアンバランスな木の全探索などでは，それは困難である．単純に木の 1 node に 1 スレッドを割り当てると全ての CPU コアを使い切れるが，スレッド生成・管理のコストが無視できない程大きくなりうる．このコストを嫌って，プログラマが自分でなるべくスレッドを作らずに，ロードバランスさせるコードを書くことは可能であるが，そのようなプログラムは実装が複雑になる．それよりも，ライブラリや言語処理系のスレッド機構が大量のスレッドを効率よくロードバランスさせながら実行するようにし，プログラマはそのスレッド機構を用いて，実装を気にせず多数のスレッドを生成するプログラムを書いた方が保守性の高いプログラムになることが期待できる．

上記のようなスレッド機構がすべき最適化の一つが，スレッド実行のためのメモリの消費量をなるべく少なくすることである．単純な実装では非常に多数のスレッドを同時に生成すると，各スレッドが用いるスタック領域が大量のメモリを消費してしまう．すぐに終了するスレッドや，ほとんど同期待ちしかしないようなスレッドが大量に生成されるようなプログラムでは，確保されたスタック領域がほとんど使われず無駄になってしまう．

そのような大量のメモリ消費を回避するための方法として，スタック領域をリンクリスト化する方法がある．これは，スタック領域として小さいメモリ領域を割り当てて，そのすべてを消費してしまった場合，それを検出し，新たなメモリ領域を確保し，既存のスタック領域にリンクする．このようなスタック領域の管理法を使用すると，スタック領域が小さなメモリ領域が複数連結されたリンクリストとなる．だが，そのような初期サイズとして小さなサイズのスタック領域を用いても，それよりも十分に小さなサイズの領域しか消費しないようなスレッドが大量に作られるようなプログラムでは使われないメモリ領域がいまだに多く発生してしまう．ここでスタック領域として割り当てるメモリのサイズを小さくすると，今度はメモリの確保の処理が頻発し，実行速度が悪化する可能性がある．

遅延スレッド (Lazy Threads [1]) では多数のスレッドを生成したときにもスタック領域を大量に消費してしまわないように，スレッド生成時にスタック領域を盗み出して割り当てる方

法を提案している．ここで言う「盗み出す」とは既存のスレッドのスタック領域の未使用部分を，生成するスレッドの実行に割り当ててしまうことを例えている．この手法により，スタック領域上の利用されていない領域の割合が減少し，メモリ効率が向上すると考えられる．しかし，やはり最初から大きなスタック領域を確保する場合と比べると依然としてスタック領域を消費しきってしまった場合のメモリ領域の確保処理は発生し，実行速度が悪化していると考えられる．

本研究では，遅延スレッドに加え，以下に述べる二つの手法を併用することで，メモリ使用量を抑えつつ高速化を図る方法を提案する．一つ目は元の遅延スレッドよりもスタック領域のサイズを大きくすることである．二つ目は，一つ目の手法を適用することで発生する子スレッド終了時の大きな空き領域を，状況に応じて任意のスレッドに再利用させる手法である．この再利用可能となった領域を本研究では fair-use 領域と呼んでいる．この二つの組み合わせにより，新規メモリの割り当て回数が減り実行速度が高速化し，また利用不可能なスタック領域の発生を抑えられメモリ消費量の増加を押さえられる．

本研究では遅延スレッドと提案手法の二つの方式を実装した．これらの手法はスタック領域のリンクリスト化を行うが，その実現ために，gcc の split stack 機能を利用した．また遅延スレッドと提案手法の二つの手法の性能を比較する評価実験を行なった．

1.2　本論文の構成

２章では，軽量なスレッドの利点の説明，並列計算を実現するための要素の説明，およびスレッドの高速化に関する既存研究の説明を行う．３章では提案手法の説明と，本研究における既存手法と提案手法の実装の説明を行う．４章では提案手法を評価と，既存手法との性能の比較を行うための実験を行う．５章で本研究のまとめと今後の課題について述べる．

第 2 章

背景

2.1 スレッド生成の手動最適化と自動最適化

複数の cpu コアを持つマシンが普及した現在では，マルチスレッドプログラミングによる並列化によりプログラムを高速化することはよく行われるようになってきている．しかし，スレッドを大量につくることにより並列化を行おうとすると，逆に性能が悪化することが考えられる．少量のスレッドであればスレッド自体にかかるコストの総量はプログラム全体で無視できるほど小さいが，大量のスレッドを作ってしまうとスレッド自体のコストが大きく積み上がってしまう．たとえば，木の探索を並列化するのに木の 1 ノードに 1 スレッドを割り当ててしまうような手法は，実装は簡単であるが，大量のスレッドが生成されてしまい，プログラムの動作速度が非常に遅くなると考えられる．また，メモリの消費量もスレッドの数に比例して大きくなると考えられる．

そのような問題を回避しつつ，マルチスレッドで動くプログラムの高速化をするために，cpu のコア数分だけスレッドを作り，そのなかでうまく仕事を分割するような最適化が考えられる．しかし，そのような最適化はプログラマが実装するものであるので，当然ながらプログラマの負担は増えることになる．とくに，簡単に等分割できないような問題では負担の増加は顕著である．そのような例として，たとえばアンバランスな木の並列な探索が挙げられる．

理想的には，大量のスレッド作る方式の，つまりプログラマにとって負担の少ない方式で実装を行いつつ，スレッドを提供する機構の側でスレッド自体のコストがなるべく小さくなるような最適化が自動的に行われればよい．そのようなスレッド機構の実現のための研究はすでに数多く行われているが，本研究もそれに類するものである．本研究ではプログラムの高速化とメモリ消費量の削減を同時に行うことを目的としている．

2.2 スレッドについて

スレッドとは平行に実行可能な処理の単位である．プロセス内に複数のスレッドが存在する場合，プログラマが各スレッドは同時平行で処理されることを想定している．言い換えると，無限に cpu コアを持つ計算機あった場合，その上で動く各スレッドは各コア上で実行され，全

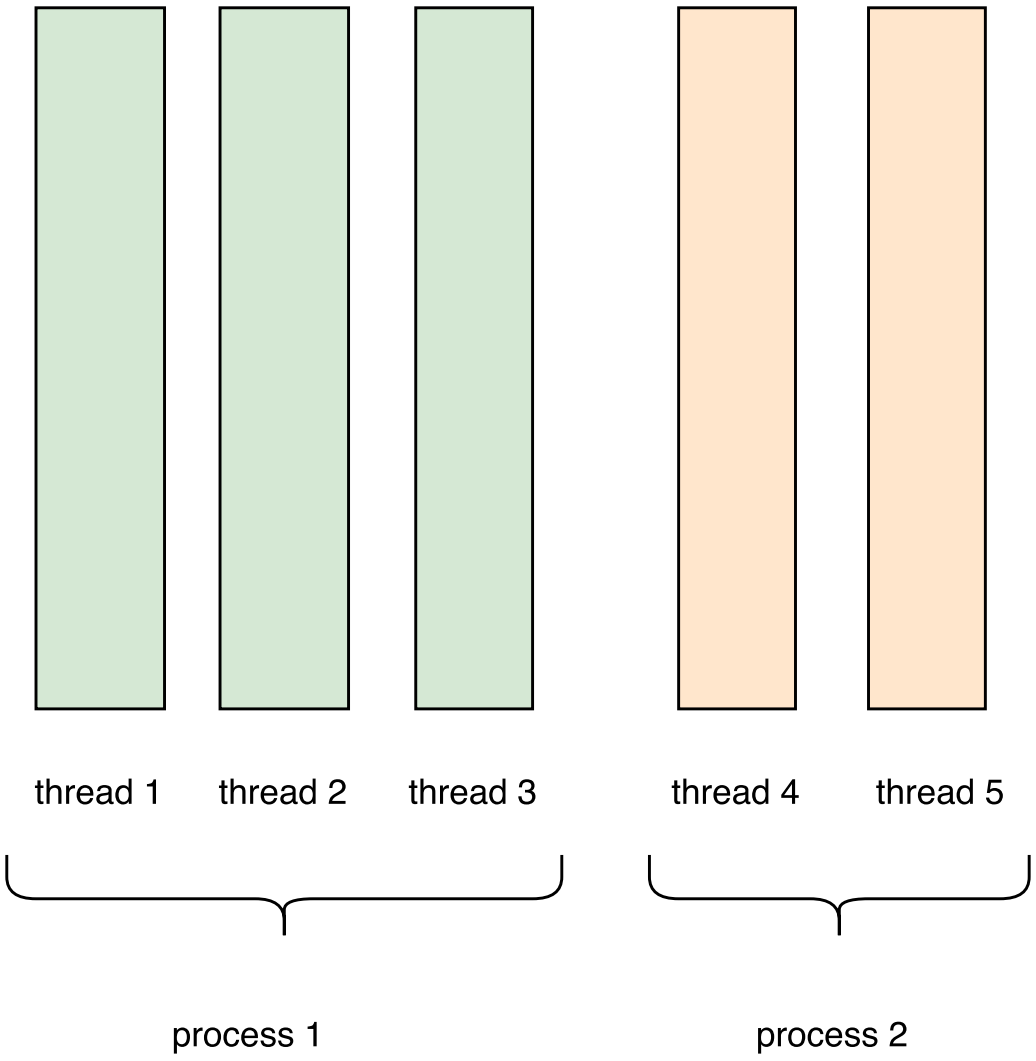

図 2.1. スレッドとプロセス

てのスレッドが並列に実行される．同様の処理の単位であるプロセスと違い，スレッドの場合，同一プロセス内にあるスレッドはメモリ空間を共有している．プロセスの場合，各プロセス毎に独立したメモリ空間を持つ．図 2.1 では同じ色が同じメモリ空間を表している．図中では process 1 と process 2 は独立した異なるメモリ空間で動作しており，process 1 中にある thread 1，2，3 は同じメモリ空間で動作している．また process 2 中の thread 4，5 も同じメモリ空間で動作している．

2.3　カーネルランドとユーザーランド

計算機上で動くプログラムには，カーネルランドで動く物とユーザーランドで動く物の二種類がある．カーネルランドで動くプログラムは cpu が特権モードと呼ばれるモードに設定された状態で動く．このモードで動作するプログラムは全てのメモリにアクセスすることができ

る．また，ハードウェアにも直接アクセスすることができる，すなわち，cpu に繋がっているハードウェアにどのような信号を送るかをプログラムすることができる．たとえば，cpu の n 番のピンを on にするというようなことである．これは，cpu の n 番のピンに電圧を掛けて，そこに繋がっているハードウェアに信号を送っている．また，カーネルランドでは，割り込みの on/off を切り替えたり，割り込みハンドラを設定したりすることが可能である．割り込みとは，cpu に繋がるハードウェアから要求があった場合，現在 cpu 上で実行中の処理を中断し，割り込みの種類ごとに設定された他の処理を実行することである．この，割り込みに応じて設定された処理のことを割り込みハンドラと呼ぶ．割り込みは cpu に繋がる外部ハードウェアからだけでなく，cpu 自身やプログラム上から発生させることもできる．そのようなものにタイマー割り込みやソフトウェア割り込みがある．タイマー割り込みはある一定時間が経過すると割り込みを発生させる．ソフトウェア割り込みはプログラマが cpu のソフトウェア割り込み命令を実行すると発生させることができる．ソフトウェア割り込みはシステムコールを実現するために利用できる．ソフトウェア割り込みが発生すると，カーネルランド上の割り込みハンドラが実行され，ユーザーランド上で割り込み命令を実行すれば実行からカーネルランドに制御が遷移するからである．加えて，マルチコアの場合，cpu の特権モード中では他の cpu を一時停止させることができる．これにより，クリティカルセクションを作ることができる，言い換えると，排他処理が行えるようになる．

クリティカルセクション中では，複数のスレッドが同時に行ってはならない処理を確実に一つのスレッドだけによって実行されるようにすることができる．別の言い方をすると，あるスレッドがあるクリティカルセクション内にいる場合，他のスレッドはそのクリティカルセクションに入れないようにして，クリティカルセクションに入ろうとするスレッドがあった場合そのセクションの手前でスレッドを一時停止させる．たとえば，ある変数に 2 つ以上の複数の cpu が同時に値を書き込んだり，ある cpu が値を書き込むのと同時に別の cpu が読み込む，といったことを防ぐことができる．複数の書き込みを同時に行ったり，書き込みと同時に読み込みを行った場合，不正なデータが書き込まれたり読み出されたりする場合がある．また，クリティカルセクションは二つ以上の一連の処理を不可分に行う為にも用いられる．

クリティカルセクションの実装は，スレッドがクリティカルセクションに入っているかを表すフラグを用いて実装される．ただし，このフラグの操作そのものを排他的に行う必要がある，つまり，クリティカルセクションを作る必要がある．このフラグ操作のためのクリティカルセクションを，ここでは 原始的なクリティカルセクションと呼ぶことにする．原始的なクリティカルセクション中では他のスレッドや割り込みハンドラに切り替わるのを防がなくてはいけない．このために cpu の全ての割り込みを禁止するがあるが，前述下通り cpu が特権モードで動作しているのでそれが可能である．ただし，このように他の cpu の実行を妨げてしまう原始的クリティカルセクションはプログラムのパフォーマンスを悪化させるので，できる限り短い方がよい．このような場合のクリティカルセクションの例を Listing 2.1 で説明する．`begin_cretical_section()` 関数と `end_cretical_section()` 関数が原始的クリティカルセクションを実装している．

Listing 2.1. 原始的クリティカルセクションの例

```
1  char* data;
2  int size;
3
4  void set_data(char* data_, int size_) {
5    begin_cretical_section();
6
7    data = data_;
8    size = size_;
9
10   end_cretical_section();
11 }
12
13 void get_data(char** out_data, int* out_size) {
14   begin_cretical_section();
15
16   *out_data = data;
17   *out_size = size;
18
19   end_cretical_section();
20 }
```

まず，一行目で char ポインタ型変数 data と，二行目で int 型変数 size を定義している．変数 data には char 型配列を，変数 size にはそのサイズをそれぞれ設定する．ただし，これら 2 変数は，変更する場合は同時に変更する必要があり，また，参照する場合は参照中にこれら 2 変数への変更が行われないようにしなくてはならない．次に 4 行目で set_data 関数を定義している．この関数は char ポインタ型変数 data_ と，それの指し示す先の char 配列のサイズを表す int 型変数 size_を引数として受け取る．5 行目で begin_cretical_section() 関数を呼び，クリティカルセクションを開始する．7, 8 行目で，引数として受け取った data_ と size_をそれぞれ data と size に設定している．この 2 変数の代入処理はクリティカルセクション中にあるので，他のスレッドによって同時に代入処理が行われることはない．最後に，10 行目でクリティカルセクションを終了している．13 行目では data と size を参照する get_data 関数を定義している．この関数は char ポインタのポインタである out_data と int ポインタ out_size の 2 引数を受け取る．14 行目でクリティカルセクションに入り，16, 17 行目で data と size の値を読み出し，引数 (ポインタ型) の指し示す先に書き込んでいる．data と size の読み出しは不可分に，つまり二つ同時に行う必要がある．言い換えると，16 行目と 17 行目の間，つまり data の読み込みと size の読み込みの間に data や size を更新する処理が入ってはいけない．begin_cretical_section() と end_cretical_section() で囲まれたクリティカルセクション中では割り込みや他のスレッドへの切り替えは発生しないので，2 変数を不可分に読み出すことができる．19 行目の end_cretical_section() 関数呼び出しでクリティカルセクションを終了している．カーネルはカーネルランドで動く．カーネルは適切に割り込みを設定し，ユーザーランドで動くプログラムをスケジュールする．また，計算機に繋がるハードウェアにアクセスし，ユーザーランドで動くプログラムが計算機に繋がるハードウェアに簡単にアクセスできるような抽象化されたインターフェースを提供する．た

とえば，ハードディスクは抽象化され，ユーザーランドのプログラムからはファイルシステムとして見える．別の例では，Ethernet や無線 LAN を経由してインターネット通信を行う場合も，通信経路はソケットとして抽象化され，ユーザーランドで動くプログラムを書くプログラマはネットワークカードなどの通信機器や tcp/ip といった通信規格を意識する必要がない．カーネルランド上で動き，cpu に繋がるハードディスクやネットワーク通信機器を直接操作し抽象化するためのプログラムはデバイスドライバと呼ばれる．

ユーザーランドで動くプログラムは，cpu が保護モードと呼ばれるモードに設定された状態で動く．このモードで動作するプログラムは全てのメモリにアクセスすることは出来ない．もしアクセスが許可されていないメモリアドレスに不正にアクセスした場合，たとえばカーネルがそのユーザーランドプログラムを強制終了したりする．詳しく説明すると，不正なメモリアクセスが発生した場合，cpu は例外を発生させ，計算機のモードをカーネルモードにして，例外ハンドラを実行する．そして，例外ハンドラが不正なメモリアクセスを行ったユーザーランドプログラムを強制終了する．

また，cpu に繋がるハードウェアに直接アクセスすることはできない．そのようなハードウェアを利用する場合，カーネルを経由して，デバイスドライバによって提供される抽象化されたインターフェースを利用して間接的にアクセスする．ユーザーランドのプログラムがカーネルの提供する機能にアクセスする場合はシステムコールを発行し，カーネルと通信することになる．

ユーザーランドのプログラムがクリティカルセクションを作る場合，つまり排他処理を行う場合もシステムコールを発行しカーネルに処理を移譲することになる．カーネルランドでは排他の為に自分以外の cpu を一時停止させることが可能であったが，ユーザーランド中では cpu は特権モードではないため，それは不可能である．ではどのようにするかというと，クリティカルセクションを実行中かどうかを格納するフラグ変数を用意し，クリティカルセクションに入るときにシステムコールを呼びカーネルによってフラグを立てる．フラグを立てるときに自分以外の cpu を一時停止およびスレッドの切り替えの禁止をして，自分以外のスレッドが同時にフラグを立てるのを防ぐ．クリティカルセクションを出るときも，同様にシステムコールを経由しカーネルランドでフラグを折る．このような仕組みをミューテックスと呼ぶ．

2.4 仮想メモリ

2.2 で見たように，別々のプロセスは独立したメモリ空間で動く．言い換えると同じメモリアドレスであっても，プロセス毎に格納されているデータは異なる．この仕組みを実現するのが仮想メモリである．

まず，物理メモリについて説明する．物理メモリは実際のハードウェアによって提供されるものであり，メモリ空間を一つ持つ．これを物理メモリ空間と呼ぶ．物理メモリ空間は通常カーネルランドからしかアクセスできない．

次に，仮想メモリについて説明する．これは，実際の物理メモリを，並べ替えて配置しているように仮想的に見せかけるためのものである．仮想メモリでは，実際の物理メモリの並び順

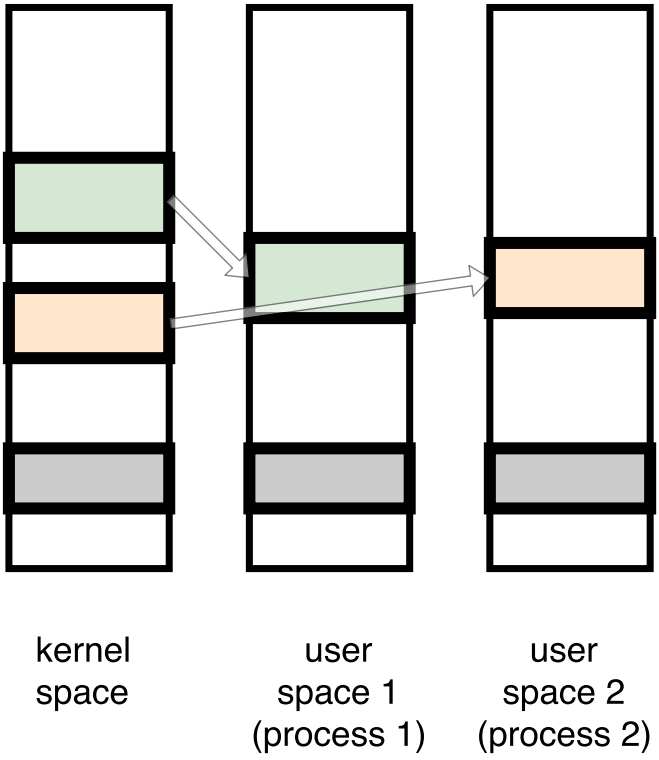

図 2.2. カーネル空間とユーザー空間

とは異なる順でメモリをマッピングしたり，対応する物理メモリがないようなメモリを作ることができる．図 2.2 はカーネルとユーザー空間プロセスの例である．kernel space は物理メモリ空間となっていて，user space が仮想メモリとなっている．kernel space と 各 user space にある灰色の領域はみな同じメモリ領域を指している．このメモリ領域は仮想メモリによりマップされていないメモリであるため，kernel space と 各 user space で同じアドレスに同じデータが置かれている．しかし，user space 1 と 2 の緑の領域と肌色の領域は同じアドレス上にあるのに違うデータとなっている．これは kernel space 上のデータが cpu の仮想メモリを行う機能 (MMU) によってマッピングされているからである．

2.5 コンテクストスイッチ

コンテクストスイッチは実行中のスレッドの切り替えである．スレッドの現在の動作状況を保存し一時停止して別の場所へ退避し，さらに別の退避されているスレッドの動作状況を復元したのちにスレッドの実行を再開する，ということを行う．実際には cpu のレジスタの保存と復元を行う．このとき，全てのレジスタを保存，復元する方法と，callee saved registers と呼ばれる，関数の return 時に復元されるレジスタのみ保存，復元する二つの方法がある．前者はタイマー割り込みなどをトリガーとして，動作中のスレッドを強制的に切り替える場合に利用され，後者はプログラマがコンテクストスイッチを行う関数を自発的に呼びスレッドを切り替える場合に利用される (このようなタイプのプログラムを協調的マルチタスクと呼ぶ) 以下に協調的マルチタスクにおけるコンテクストスイッチを x86_64 アーキテクチャ cpu で実装した場合の一例を示す．

Listing 2.2. 協調的マルチタスクにおけるコンテクストスイッチを x86_64 cpu で実装した場合の一例

```
1  save_context:
2          movq %rbx, 0(%rdi)
3          movq %rbp, 16(%rdi)
4          movq %r12, 24(%rdi)
5          movq %r13, 32(%rdi)
6          movq %r14, 40(%rdi)
7          movq %r15, 48(%rdi)
8
9          popq %rcx
10         movq %rsp, 8(%rdi)
11         movq %rcx, 56(%rdi)
12         pushq %rcx
13
14         mov $0, %rax
15         ret
16
17 restore_context:
18         movq 0(%rdi), %rbx
19         movq 8(%rdi), %rsp
20         movq 16(%rdi), %rbp
21         movq 24(%rdi), %r12
22         movq 32(%rdi), %r13
23         movq 40(%rdi), %r14
24         movq 48(%rdi), %r15
25
26         mov $1, %rax
27         jmp *56(%rdi)
```

`save_context()` 関数は現在のコンテスト，つまり cpu のレジスタ等の状態を保存する関数である．引数として 64 バイトの大きさのメモリ領域を指すポインタを受け取る．これは 8 バイト変数を 8 個分の配列であり，cpu のレジスタ 8 個分のデータが格納できる．この引数 (第

一引数) は x64 の呼び出し規約では rdi レジスタに格納される．2 行目から 15 行目までが save_context() 関数の処理である．まず，2 行目で rbx レジスタを保存している．保存先は rdi レジスタの指すアドレスのメモリである．3 行目で rbp レジスタを保存している．保存先は rdi レジスタの指すアドレスから 16 バイト進んだ位置，つまりオフセットが 16 バイトの位置である．4 行目で r12 レジスタを保存している．保存先は rdi レジスタの指すアドレスのオフセット 24 バイトの位置である．以下，同様に r13，r14，r15，レジスタを rdi レジスタの指す先からのオフセットが 32，40，48 である位置に保存する．9 行目でスタックのトップにある値を pop し， rcx に格納している．pop 時に rsp レジスタが指すものは save_context() 関数の return 先アドレスであり，これが rcx に格納され， rsp の値は 8 バイト分インクリメントされる．pop 時にデクリメントではなくインクリメントされるのは，x64 アーキテクチャではスタック領域は下 (アドレスが 0 に向かう方向) に伸びて行くからである．10 行目で現在の rsp レジスタの値を rdi レジスタの指すアドレスから 8 バイト進んだ位置に保存している．11 行目で rcx に格納されている return 先アドレスを rdi レジスタのオフセットが 56 の位置に保存し，12 行目で同じく rcx に格納されている return 先アドレスを push して，9 行目で pop する前の状態に戻す．14 行目で rax レジスタに戻り値として 0 を設定している．15 行目で ret 命令により save_context() 関数の呼び出し元に return している．次いで，17 行目から 27 行目に restore_context() 関数の定義が書いてある．restore_context() 関数は現在のコンテスト，つまり cpu のレジスタ等の状態を復元する関数である．引数として 64 バイトの大きさのメモリ領域を指すポインタを受け取る．これは save_context() の時と同様に 8 バイト変数を 8 個分の配列であり，cpu のレジスタ 8 個分のデータが保存され格納されている．また，同じく save_context() の時と同様にこの引数 (第一引数) は x64 の呼び出し規約では rdi レジスタに格納される．18 行目に rdi レジスタの指す位置にあるデータを rbx レジスタに設定する．この処理は save_context() 関数で保存された rbx レジスタを復元する処理である．19 行目で rdi レジスタのオフセット 8 バイト目にあるデータを rsp レジスタに設定し，復元している．以降，同様に rdi レジスタのオフセット 16，24，32，40，48 バイト目にあるデータを rbp， r12， r13，r14，r15 に設定し，復元している．26 行目で戻り値を格納する rax レジスタに戻り値として 1 を格納する．27 行目で rdi レジスタのオフセット 56 バイト目にあるデータが指す先にジャンプして restore_context()　関数は終了する．ジャンプ先である rdi レジスタのオフセット 56 バイト目にあるデータは，11 行目で save_context() 関数内で保存した， save_context() 関数の戻り先アドレスである． restore_context() 関数は自身の呼び出し元ではなく，rdi にレジスタ群を保存した save_context() 関数呼び出しの呼び出し元に return するが，その戻り値は 1 である． save_context() 関数の呼び出し使用者は， save_context() の戻り値が 0 か 1 かで，save_context() 関数がレジスタ群を保存して return したのか，レジスタ群を復元して return したのかを区別する．

2.6 プリエンプティブスレッド

一般的に計算機の cpu コアの数よりもスレッドの数の方が多い．なので，実際に計算機上で動くスレッドは cpu コア数分だけである．しかし，ある特定のスレッドが cpu コアを占有するわけにはいかない．これを解決する一つの方法は cpu コアを時間で分割することである．つまり，あるスレッドが cpu コア上で一定時間動作したら，別のスレッドに cpu コアを明け渡す．またこのときの分割された各時間のことをタイムスライスと呼ぶ．また，スレッドがタイムスライスを使い切ったときに，そのスレッドを強制的に中断し，別の中断しているスレッドを動作させる役目を持つプログラムをスケジューラと呼ぶ．スケジューラは，cpu コア上で動作させるスレッドをどのタイミングで切り替えるかを決定する権限を持つ．

プリエンプティブなスレッド機構では上で述べた通り，あるスレッドがある cpu コア上で一定時間動作するとその時点でコンテクストスイッチが発生し強制的に別のスレッドに切り替わる．以下に，プリエンプティブスレッドにおいてコンテクストスイッチが発生する場合の例を示す．

```
1 long factorial(long n) {
2   long tmp = 1;
3   while (n != 0) {
4     tmp *= n;
5     n -= 1;
6   }
7   return tmp;
8 }
```

この例では n の階乗を求める関数を定義している．1 行目から関数 factorial() の定義を開始している． factorial 関数は long 型の引数を n として受け取り， long 型の値を返す．2 行目で long 型の変数 tmp を 1 として初期化し，定義する．3 行目から 6 行目まで， n が 0 になるまでループを実行する．ループの中身を説明する．4 行目で tmp に tmp * n を代入し，5 行目で n を 1 デクリメントする．ループが終了すると，7 行目で tmp の値を return する．引数の n が非常に大きい場合，全ループが完了するまで処理に時間がかかることが予想される．そのような場合に，例えば，引数 n が 100 であり，ループの n が 50 のときの処理中に，タイムスライスを使い切ったとする．そうすると，スケジューラによって factorial() 関数を実行しているスレッドが停止され，他の停止中のスレッドを再開する処理が行われる．

2.7 ノンプリエンプティブスレッド

ノンプリエンプティブスレッドは 2.5 で登場した協調的マルチタスクのような，プログラマが自発的にスレッドを切り替えるタイプのスレッドである．

以下でノンプリエンプティブスレッドにおいて自発的にコンテクストスイッチ行う場合の例を示す．

```
1  long factorial(long n) {
2    long tmp = 1;
3    while (n != 0) {
4      tmp *= n;
5      n -= 1;
6      if (n % 10 == 0) {
7        yield();
8      }
9    }
10   return tmp;
11 }
```

この例でも，プリエンプティブスレッドの例と同様に n の階乗を計算する処理を題材として扱う．この例では n の階乗を求める関数を定義している．1 行目から関数 factorial() の定義を開始している． factorial 関数は long 型の引数を n として受け取り， long 型の値を返す．2 行目で long 型の変数 tmp を 1 として初期化し，定義する．3 行目から 6 行目まで，n が 0 になるまでループを実行する．ループの中身を説明する．4 行目で tmp に tmp * n を代入し，5 行目で n を 1 デクリメントする．6 行目で現在の n の値が 10 で割り切れるかどうかを判別している．もし 10 で割り切れる場合，7 行目の yield() 関数呼び出しが行われる．この yield() 関数は自発的にコンテクストスイッチを行う処理をする関数である．すなわち，Listing 2.2 の例で説明したように，現在のスレッド (factorial() 関数を実行しているスレッド) のレジスタ等を保存し，他の中断されたスレッドの，保存されたレジスタ等を復元する処理を行い，スレッドを再開する．また，再開するスレッドはスケジューラーによって適当に選ばれ，中断するスレッドも，のちに再開出来るようにするために，適切にスケジューラーの管理下に置かれる．このような処理を階乗の計算中に挟む理由は，階乗の計算は，引数 n が大きければ大きい程，求めるのに時間がかかるので， n が大きい場合に cpu の専有を防ぐためである．ループが終了すると，10 行目で tmp の値を return する．引数の n が非常に大きい場合，全ループが完了するまで処理に時間がかかることが予想される．そのような場合にこの例では処理を途中で中断し，他のスレッドを実行する．例えば，引数 n が 100 であった場合，途中で n が 10 で割り切れる値になったとき，すなわち n が 90，80，70…となったときに，現在の処理を中断しコンテクストスイッチを行い，他のスレッドを実行する．

2.8　native thread について

native thread は 図 2.3 のようにカーネルが管理するスレッドである．ユーザーランド内のプロセスが，システムコールを経由して作成する．通常，プリエンプティブなスレッドであり，カーネルが cpu のタイマー割り込みを利用してスケジューリングを行う．また，IO 処理やロック待ちなどでブロックすると，自動的にカーネルによって別の実行可能なスレッドが cpu コア上で実行される．

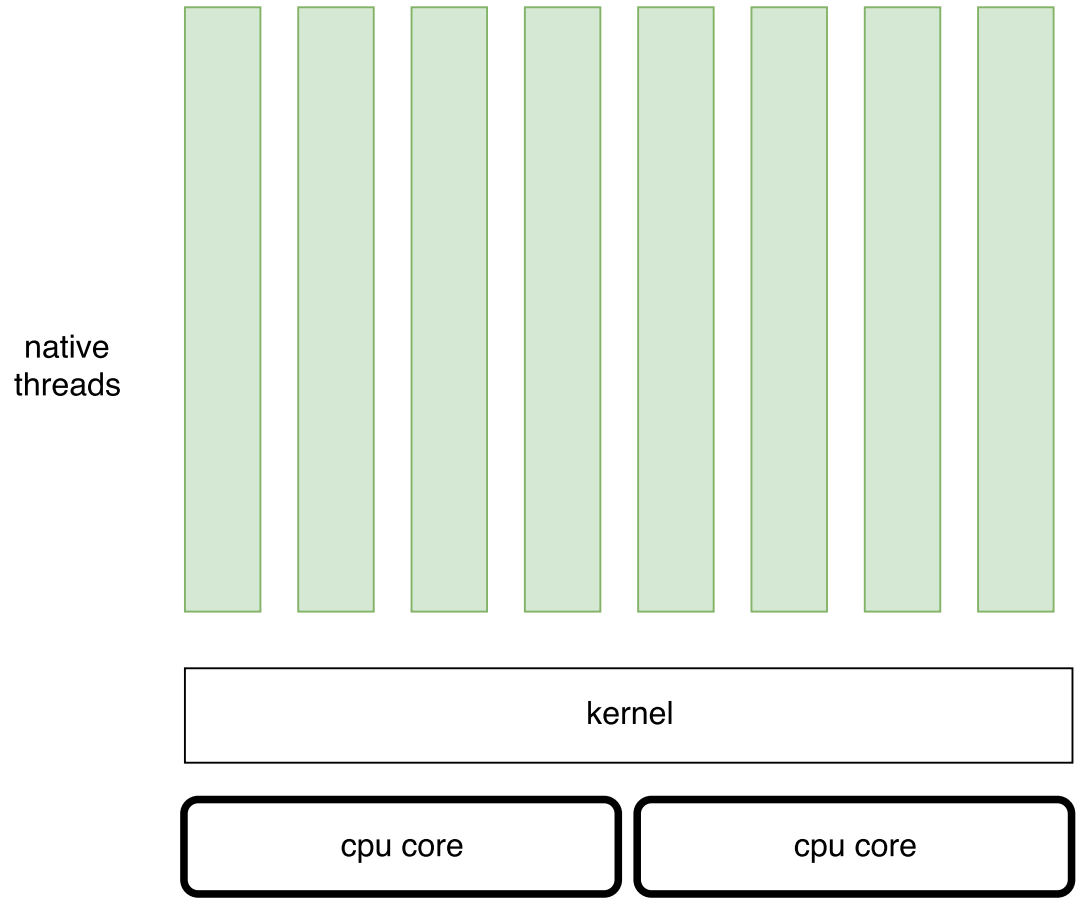

図 2.3. native thread の例

2.9 user land thread について

user land thread はユーザーランド上で実装されたスレッドである．スレッドの生成は破棄にシステムコールを経由しないため，大量にスレッドを作成するプログラムでは高速に動作すると考えられる．ノンプリエンプティブスレッド型の実装が多いが，SIGALRM シグナルを利用することでタイマー割り込みをユーザーランドで再現し，プリエンプティブスレッドを実装することも不可能ではない．

2.10 work steal

work steal はノンプリエンプティブスレッドのためのスケジューリングアルゴリズムである．lazy task creation [2] で初めて考案された．このアルゴリズムは，まずプロセッサごとに両端キューを用意する．両端キューには中断したスレッドが入っている．各プロセッサは自身の両端キューの下端からスレッドを取り出す．しかし，自身の両端キューが空の場合は他のプロセッサの両端キューの上端からスレッドを盗む．このとき，盗む側を thief と呼び盗まれる

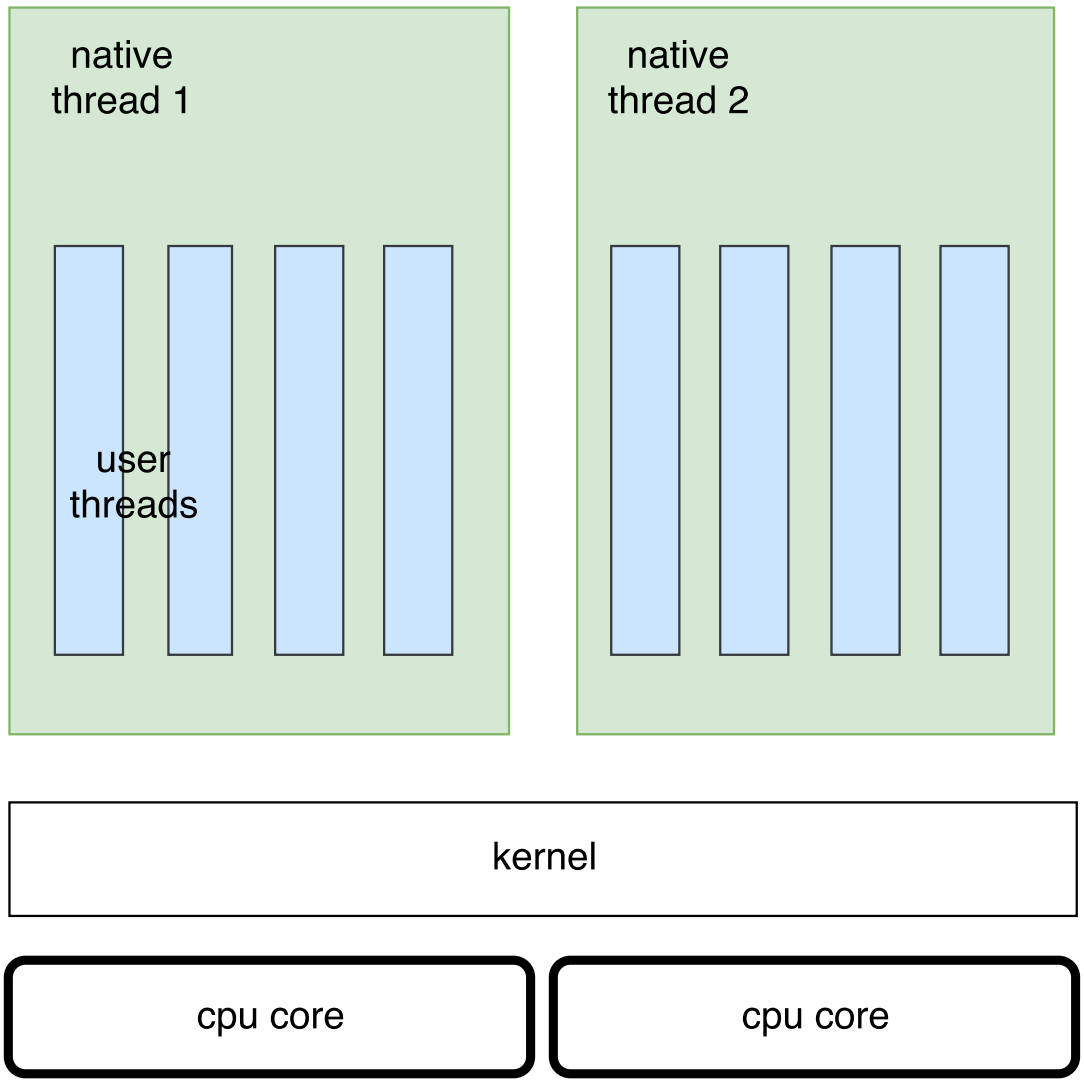

図 2.4. user thread の例

側を victim と呼ぶ．図 2.5 では プロセッサ worker 1 がプロセッサ worker 0 からスレッドを盗んでいる．また，スレッドが中断するときはプロセッサの自身の両端キューの下端にスレッドを挿入する．

2.11　split stack

split stack は，不連続なスタック領域を実現するための gcc の拡張機能である．スタック領域がリンクリスト状にすることで，スレッド生成時のスタック領域のサイズを小さくし，それを使い切ったら新たなメモリを確保しリンクリストにつなぐことでスタックオーバーフローを防いでいる．メモリの使い切りを検出するために，gcc は関数の処理の先頭にスタック領域の残りサイズをチェックをするコードを埋め込む．図 2.6 のようにスタック領域がリンクリスト状になっている．

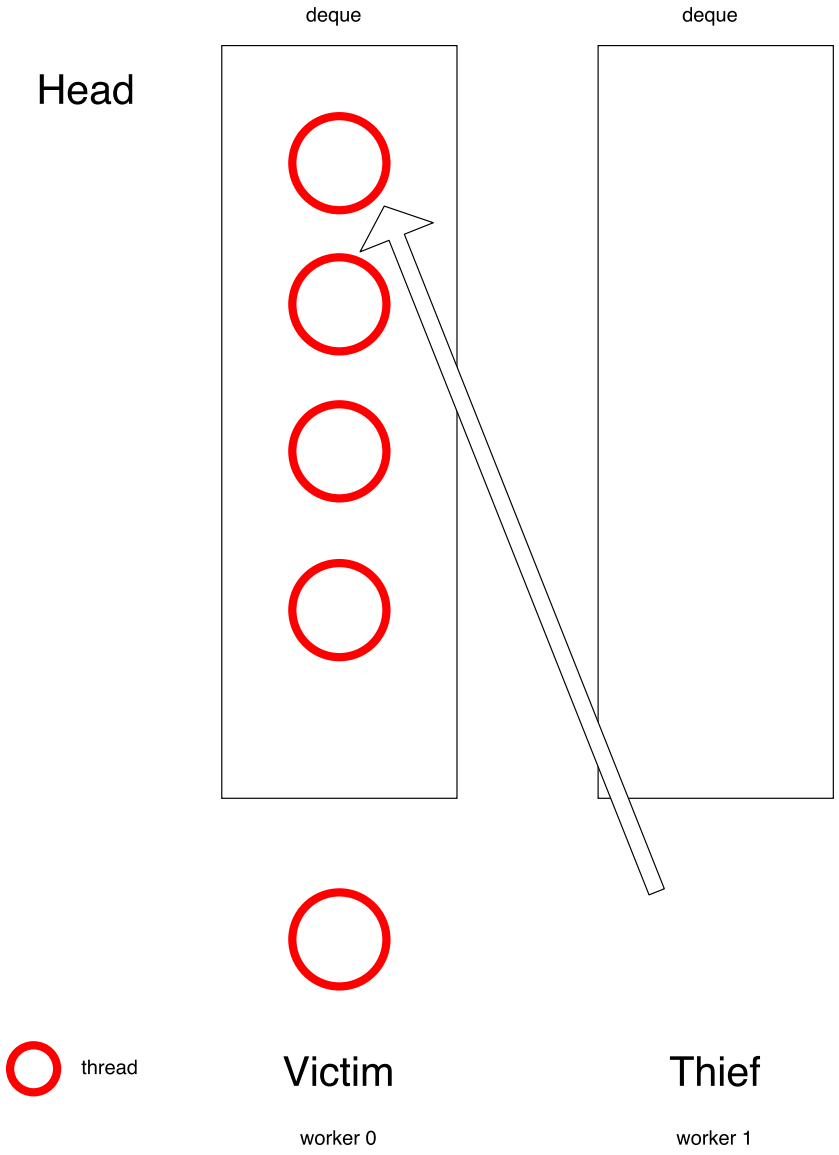

図 2.5. work steal の steal 時の両端キュー

実際の split stack のリンクリストの実装は 図 2.7 のようになっている．実際には図のように新しく確保された領域から古い領域を指す単方向リンクリストとなっている．また，各ノードはスレッド実行のためのスタック領域以外にメタデータ領域を持つ．各ノードのメタデータは一つ前のノードのメタデータ領域とスタックポインタの二つを指している．

本論文では各ノードのことをスタックブロックと呼ぶ．

実行ファイル中の split stack の構成を図 2.8 に示している．split stack はコンパイラが生成するコードと，cpu アーキテクチャ別にアセンブリ言語で書かれたライブラリ部分と，各アーキテクチャで共通の C 言語で書かれたライブラリ部分がある．各長方形の大きさが大まかな実行ファイルに占める割合となっている．

1 func:

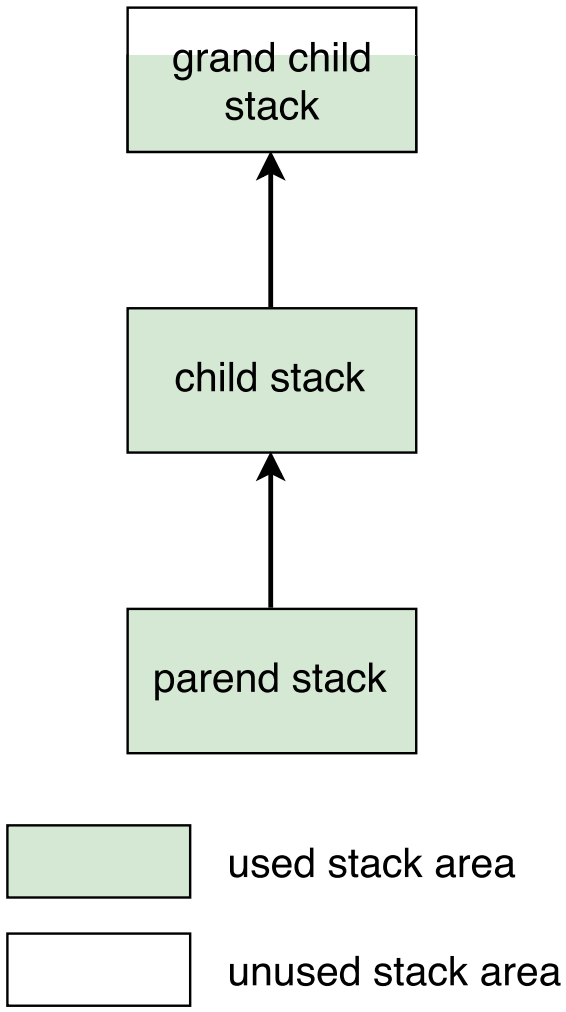

図 2.6. split stack の模式図

```
2           leaq −280(%rsp), %r11
3           cmpq %fs:112, %r11
4           jnb .L3
5           movl $280, %r10d
6           movl $0, %r11d
7           call __morestack
8           ret
9   .L3:
10          pushq %rbp
11          movq %rsp, %rbp
12          subq $272, %rsp
13          // ...
14          leave
15          ret
```

この例は gcc が生成したアセンブリ言語のコードを説明のために簡略化したものである．関数 `func` はスタック領域を 280 バイト利用する関数であり，2 から 9 行目に split stack に関係するコードがある．これは関数本体の処理がを実行される前に実行される．11 行目以降は

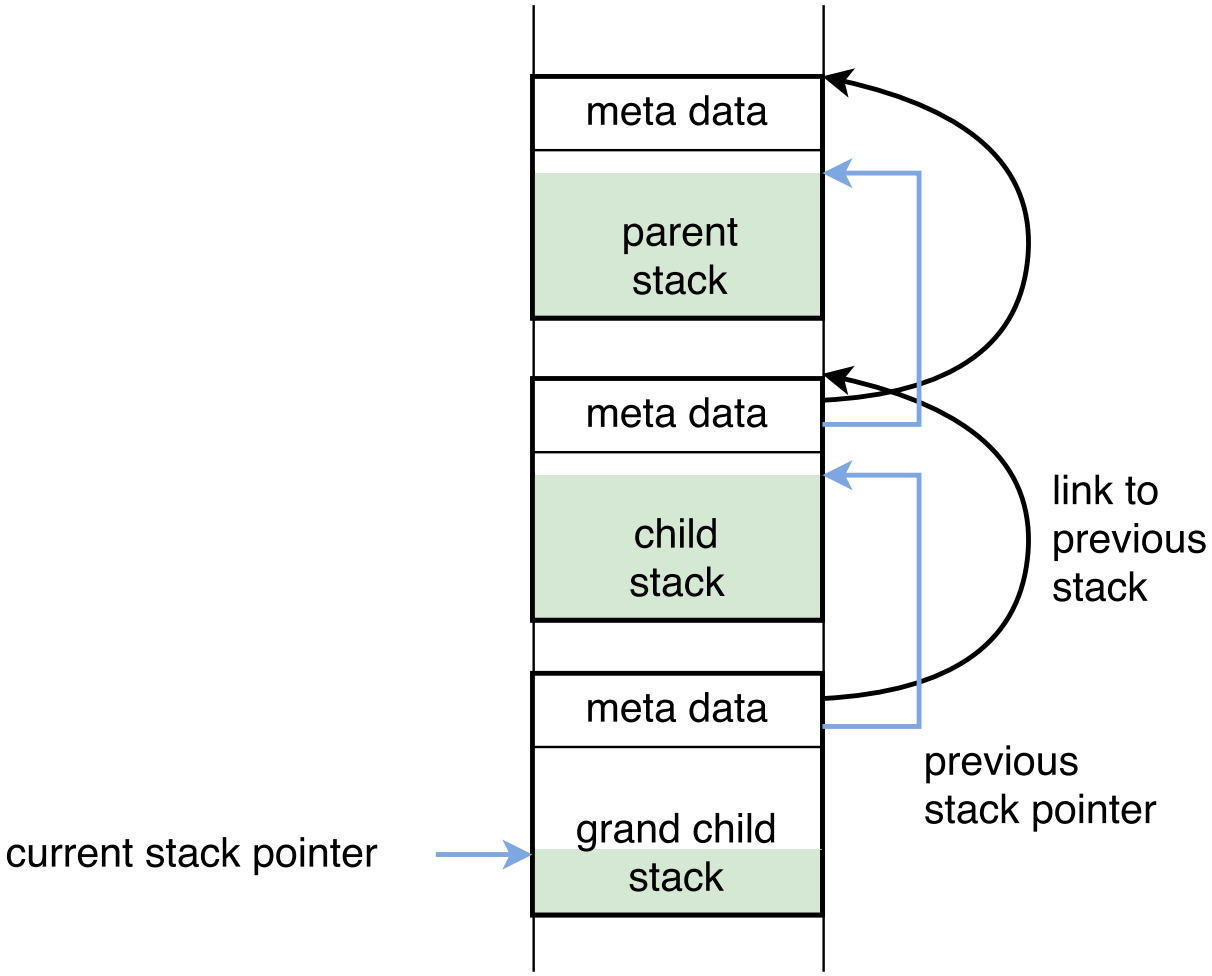

図 2.7. 実際の split stack のリンクリストの実装

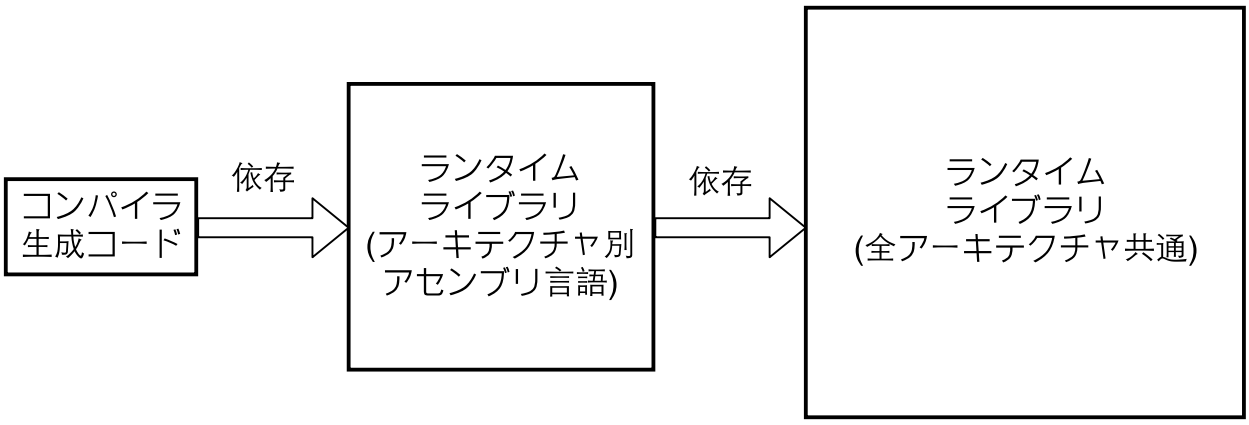

図 2.8. 実行ファイル中の split stack の構成

関数 func の本体の処理である．2 行目から 9 行目の処理の内容は，要約すると，関数本体を実行するのに必要なスタックを現在のスタック領域から確保できない場合は，新しいメモリ領域を確保して，そこからスタックを確保して関数本体を実行する．1 行目で func のシンボルを宣言している．2 行目で rsp から 280 引いたアドレスを r11 レジスタに書き込んでいる．このアドレスは，仮に split stack により新たなスタック領域の確保が行われなかった場合にスタックの top となる予定のアドレスである．3 行目では fs:112 の値と r11 の値を比較している． fs:112 という表記は， fs レジスタの指す位置から 112 バイト進んだ位置にあるデータを読み取る，ということを意味している．fs レジスタはスレッドローカルストレージを実装するために利用される場合もあるレジスタである．4 行目で，3 行目の比較の結果に応じてジャンプするかしないかを分岐している．3 行目の比較が， r11 が fs:112 より小さい値でないに場合ラベル .L3 にジャンプする．これが意味することを説明すると， func() 関数内でスタックの top になる予定のアドレスと，現在のスタック領域の上限のアドレスを比較し，スタック top になる予定のアドレスが現在のスタック領域の上限を超えていた場合ジャンプは発生せず，超えていた場合 .L3 へのジャンプが発生する．5 行目から 6 行目にかけて，split stack のランタイムライブラリ関数である __morestack() 関数の引数を設定している．__morestack() 関数は引数を r10 レジスタと r11 レジスタ経由で受け取る．引数の内容は，関数が利用するスタックの大きさと，スタック経由で関数に渡される引数の個数である．5 行目で func() 関数が使用するスタックの大きさを r10d レジスタに設定している． r10d レジスタは r10 レジスタを 32 ビットサイズのレジスタとして使用するためのものである．6 行目でスタック経由で渡される引数の個数である 0 を r11d レジスタに設定している．この例では関数 func() のスタック経由で渡される引数はないので， r11 レジスタに設定する値は 0 である．7 行目で __morestack() 関数を呼んでいる． __morestack() 関数は，新たなメモリ領域を確保し，現在のスレッドのスタック領域にリンクして追加する．このとき新規に確保される領域の大きさは 5 行目で r10d レジスタに設定したサイズ以上，つまり関数本体が使用するスタックの大きさ以上であることが保証される．また，スタック経由で渡される引数があった場合，それらが新規領域の側にコピーする処理が行われる．新規領域の確保等の処理が行われたあとは関数本体のコードを call 命令で呼び出す．この例の場合では，10 行目 (pushq 命令のある位置) のアドレスを call 命令で呼び出す．__morestack() 関数の動作をさらに詳しく説明すると， __morestack() 関数の return 先アドレスの次の命令のアドレスを call 命令で呼び出している．この例の場合であれば，__morestack() 関数の return 先アドレスは 8 行目の ret 命令であり，__morestack() 関数が呼び出す先はこの ret 命令の次の命令の位置である 10 行目の pushq 命令である．呼び出した関数本体が終了すると __morestack() 関数内に return する．この後__morestack() 関数内では新規に確保したメモリ領域と現在のスレッドのスタック領域のリンクが外されたのちに，新規領域の解放が行われる．その後 __morestack() 関数は return する．return 先は__morestack() 関数を呼び出した call 命令の次の命令であり，コンパイラはその命令が ret 命令であるコードを生成する．この例のアセンブリコードでは 8 行目の ret 命令へ戻り，8 行目の ret 命令が実行されると関数 func の呼び出し元へ戻る．10 行目から 15 行目までは関数 func の処理の本体である．10 行目で

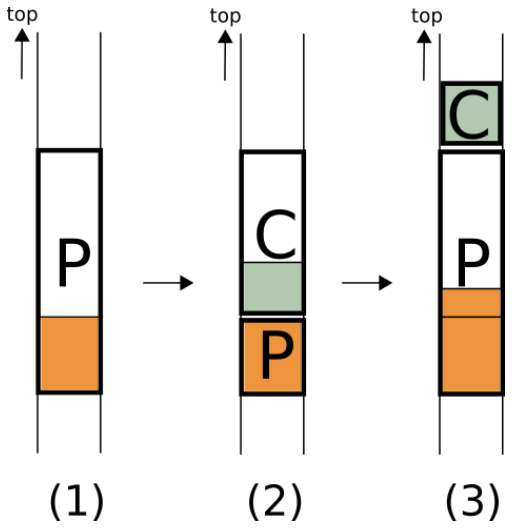

図 2.9. 子スレッドのスタック領域の割り当て

`rbp` レジスタをスタックに push する．11 行目で現在の `rsp` レジスタの値を `rbp` レジスタに設定している．12 行目で，スタック top のアドレスを格納するレジスタである `rsp` レジスタの中身を 272 引くことで，スタック top にローカル変数を置くための領域を 272 バイト確保している．この 272 バイトと 10 行目で push されたデータ 8 バイトを合わせると，2 行目や 5 行目で利用されているスタックのサイズである 280 バイトとなる．13 行目で何かしらの処理を行う．14 行目の leave 命令は `rsp` レジスタと `rbp` レジスタを復元する処理を行う．10 行目から 12 行目にかけて行われた変更が元に戻る．15 行目の ret 命令で呼び出し元に return する．ただし，7 行目の `call __morestack` を経由している場合は `__morestack` 内の処理に return し，そうでない，つまり 4 行目の jnb 命令 (jump 命令) でラベル `.L3` にジャンプした場合は `func` 関数の呼び出し元に return する．

2.12 大規模並列時のスレッドのスタック領域

並列プロセッサの普及により，性能を出すためにプログラマはスレッドを使った並列プログラムを書くことが求められている．しかし非常に多数のスレッドを同時に生成すると，各スレッドが用いるスタック領域が大量のメモリを消費し，場合によってはメモリ不足でプログラムが動作しなくなってしまう．とくに，スタック領域の大半は確保されてもほとんど使われないと考えられるので，各スレッドに大きなメモリ領域を割り当てるとメモリ効率が著しく悪化する．一方，各スレッドに割り当てるメモリ領域を小さくしすぎると，スタックオーバーフローでスレッドの実行が途中で中断してしまう危険性がある．

この問題を回避する手法のひとつが，スレッドに対するスタック領域の割り当てを遅延させる手法である．あるスレッド（親スレッドと呼ぶ）が新しいスレッド（子スレッドと呼ぶ）を生成したときには，子スレッド専用のスタック領域をすぐには割り当てず，親スレッドのスタック領域の未使用部分（現スタックポインタより上の部分）を使って子スレッドを実行す

る．親スレッドは他に空きコアができるまで停止する．例えば図 2.9 では，まず (1) 親スレッドがスタック領域 P のうち橙色の部分を使って動いているとする．ここで (2) 子スレッドが生成されると，スタック領域 P の未使用部分 C をスタック領域として子スレッドを実行する．

このようにすると，子スレッドの実行時間が短く，親スレッドの実行が再開される前に終了すれば，新たなスタック領域を確保する必要がなくなり，メモリ消費を小さくできる．一方，子スレッドが途中で同期のために停止した場合は，子スレッドが終了する前に親スレッドの実行を再開しなければならない．その場合，親スレッドの実行を再開する時点で子スレッドに専用のスタック領域を新たに割り当て，スタック領域の中身を旧領域から新領域にコピーして退避する．図 2.9 の (3) に示すように，子スレッドのスタック領域 C を元の元のスタック領域 P の外に移し，スタック領域 P を元の大きさに戻す．これにより親スレッドが自身のスタック領域を利用できるようにする．また子スレッドが実行再開後に新しいスタック領域を利用するように，停止した子スレッドのスタックポインタやフレームポインタを新しいスタック領域を指すように変更する．多数のコアをもつマシンなどで，子スレッドが同期のため停止する前に親スレッドの実行を別のコアを使って再開する場合は，タイマー割り込みなどを用いて子スレッドをいったん強制的に停止させる．

しかしながら，スタック領域の中身（図 2.9 のスタック領域 C の緑の部分）を旧領域から新領域にコピーすると，コピーの時間が実行時ボトルネックとなる可能性がある．また旧スタック領域内をさすポインタが存在する場合は，そのポインタが新領域をさすように，値を変更しなければならない．このため，この手法は C 言語には適さないという問題もある．この問題を避けるには，親スレッドと子スレッドを同時に動かさないこととし，親スレッドが一時停止して，子スレッドの実行が再開するときは，親スレッドのスタック領域の中身を退避して，子スレッドのスタック領域の中身を親スレッドのスタック領域へ復元する，という手法が考えられる．つまり図 2.9 の (3) でコピーしたスタック領域 C を再び元の位置に戻してもよいが，スタック領域のコピーのオーバーヘッドが非常に大きくなることが予想される．また多数のコアがある場合，この手法では多数のコアを有効活用できない可能性がある．

2.13　関連研究

2.13.1　lazy threads

Lazy threads [1] は，多数のスレッドを生成したときにもスタック領域を大量に消費してしまわないように，子スレッドに親スレッドのスタック領域を盗み出して割り当てるものである．この方式では， 2.12 で説明した方式のように，子スレッドは親スレッドのスタック領域の未使用部分を使って実行される．しかしながら 2.12 の方式と異なり，子スレッドはそのスタック領域を終了するまで親スレッドに返却しない．言わば盗んでしまう．子スレッドが終了する前に親スレッドの実行を再開するときは，親スレッドは新しいメモリ領域を得て，それを既存のスタック領域にリンクリストのように連結して用いる．新しいメモリ領域として大きな領域は確保しないものとし，メモリの消費を抑えている．

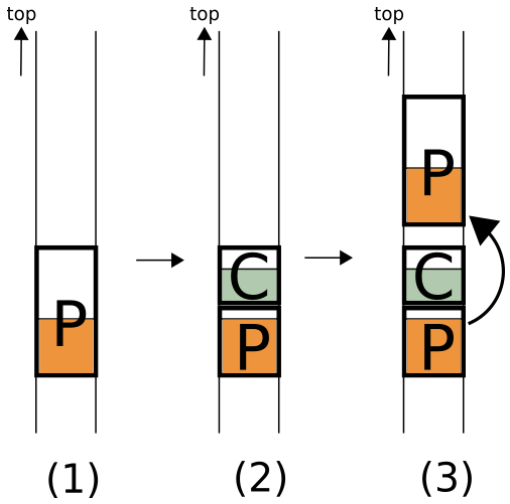

図 2.10. lazy threads 方式における親スレッドからのスタック領域の割り当て

図 2.10 に示すように，親スレッドのスタック領域 P の一部をスタック領域に使って子スレッドを動かす (2) までは図 2.9 と同じであるが，親スレッドの実行を再開する (3) では，子スレッドのスタック領域 C はそのままにして，親スレッドのスタック領域を追加で確保し，元のスタック領域と連結して用いる．lazy threads 方式では，各スレッドに一回に割り当てるスタック領域を小さくし，全体としてメモリの消費量を抑える．例えば図 2.10 の (1) で親スレッドのスタック領域 P は，図 2.9 に比べて小さくなっている．

2.13.2 アルゴリズム

lazy threads のメモリ管理手法をユーザーランドスレッドライブラリに適用した場合のアルゴリズムは以下のようになる．

lazy threads 方式では，スレッドの切り替えはレジスタの退避によって行い，スケジューリングには work steal アルゴリズムを利用する．その上で，スタック領域の管理にスタック領域の盗出し法を用いる．

スレッドを生成すると親スレッドは一停止する．このとき thread entry と呼ぶデータ構造を作ってレジスタ等を退避し，生成する子スレッドの実行を開始する．作成した thread entry は論理コアごとの deque に入れて管理し，スケジューリングをおこなう．親スレッドの一時停止時に退避した entry は親スレッドを実行していた論理コアの deque の末尾に push する．子スレッドが終了したとき，親スレッドの実行が一度も再開されていない場合は，親を deque から取り除き，親の実行を再開する．そうでない場合は以下のように work steal スケジューリングを行う．各論理コアは自分の deque の末尾から thread entry を取り出す．もし，自分の deque が空なら，空でない他の論理コアの deque の先頭から thread entry を取り出す．論理コアは，取り出した thread entry 中の退避されたレジスタなどを復元し，そのスレッドを再開する．

子スレッドのスタック領域の割り当て (盗出し) は，本方式では以下のように行う．子スレッ

ドのスタック領域には，親スレッドのスタック領域の上方の余りを使う．上方とは，スタックの成長方向のことを指す．上方の余りとは，親スレッドのスタックポインタ (以下 sp) からマージン領域のサイズ分進んだ位置から，親スレッドの元のスタック領域の上限の間の領域のことである．マージン領域は後述する split stack の動作のに使用するスタック領域である．子スレッドのスタック領域の開始位置が決まったら，その位置を親スレッドのスタック領域の新しい上限とする．親が再開する前に子が終了した場合は，親のスタック領域の上限を増やし，子が使用していたスタック領域を親に返却する．親が再開した後で子が終了した場合も子が使用していたスタック領域は親に返却するが，それは後述するように親のスタック領域が縮んで，子スレッドが使用していた領域と連続領域となったときである．

子スレッドの実行中に親スレッドの実行を再開するときは，新しいメモリ領域を確保し，それを短くなった既存のスタック領域と連結して利用する．短いスタック領域を連結して一つのスタック領域として用いるために, gcc の split stack 機能を用いる．この機能により, 関数呼び出しのたびに，呼び出す関数の実行に必要なスタック領域の大きさ (以下 $frame\text{-}size$) と，スタックポインタ sp とスタック領域の上限の間の大きさを比較する．比較の結果，$frame\text{-}size$ 分の領域が sp より上にある場合はそのまま関数を実行する．領域がない場合は，まずスタック領域の上限のすぐ上に終了した子スレッドのスタック領域があれば，その分だけスタック領域の上限を増やす（このとき子スレッドが盗んだスタック領域が親に返却される）．その上で再び $frame\text{-}size$ と，sp とスタック領域の上限の間の大きさを比較し，$frame\text{-}size$ 分の領域が sp より上にある場合は呼ばれた関数を実行する．領域がない場合，新たにメモリ領域を確保し，既存のスタック領域に追加して連結する．確保するメモリの大きさは，固定長の小さなものか，それが $frame\text{-}size$ 未満なら，$frame\text{-}size$ と split stack のためのマージン領域の和である．呼ばれた関数は追加されたメモリ領域をスタック領域の一部として実行される．

関数の実行が終了して戻るとき，その関数の呼び出し時に既存のスタック領域にメモリ領域を追加して連結していた場合，追加した領域をスタック領域から外す．外したメモリ領域の一部が子スレッドやその子孫のスレッドにより使用されていなければ，そのメモリ領域を解放する．

2.13.3　メモリ効率

lazy threads 方式では，スタック領域を比較的小さい領域に分割して管理する．したがって多数のスレッドが生成され，その多くが短い時間で終了するようなプログラムの場合に，スタック領域のためのメモリを効率よく利用して実行できる．スタック領域のために全体として必要なメモリ量を小さく抑えられることが期待できる．

類似の手法として StackThreads/MP[3] で用いられているスタック領域の管理法があるが，状況によっては利用されないメモリ領域が大量に発生する場合がある．

StackThreads/MP では，論理コア毎に連続した 1 本のスタック領域を用意し，それが同一コア上で実行される複数のスレッドによって共有される．例えば，親スレッドと子スレッドが交互に実行された場合，それぞれのスタックフレームが 1 本のスタック領域上に交互に並ぶこ

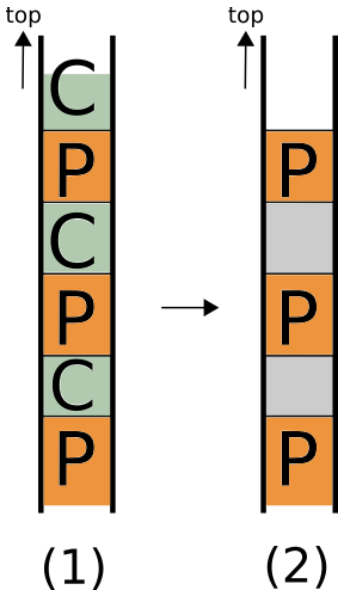

図 2.11. StackThreads/MP のスタック領域の管理

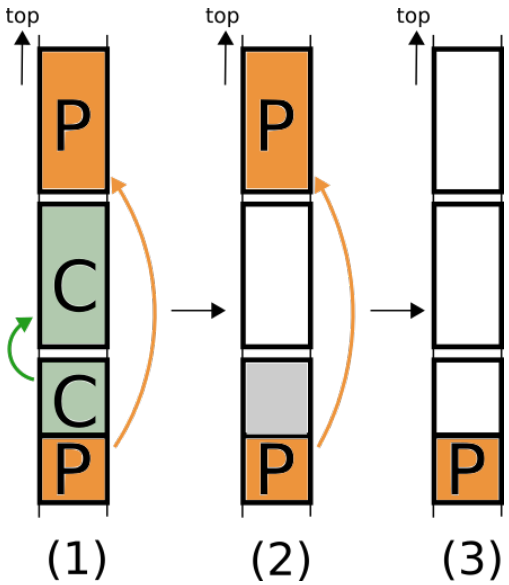

図 2.12. lazy threads 方式での歯抜けのスタック領域の扱い

とになる．したがって子スレッドが先に終了した場合スタック領域上に多数の歯抜けの領域が図 2.11 のようにできてしまう．この灰色で示した歯抜けの領域は親スレッドが終了するまで再利用されない．StackThreads/MP では，短い細切れのスタック領域を連結して用いることができないからである．

一方，lazy threads では，子スレッドと親スレッドが交互に実行された後に，先に子スレッドだけが終了した場合も，子スレッドがスタック領域として利用していたメモリの大半を再利用できる．子スレッドのスタック領域は，親スレッドから盗んだスタック領域と，新規に子スレッドが確保して占有するスタック領域の 2 つがある．子スレッドが占有する領域は，子ス

レッドの終了（正確にはその領域を確保した関数呼び出しの終了）と共に解放され，再利用可能になる．親スレッドのスタック領域から子スレッドが盗んだ領域も，親スレッドが既にそのスタック領域を利用していなければ，その領域全体を解放して再利用できる．親スレッドがまだそのスタック領域を利用中の場合は再利用できないが，そのような領域は小さい．なぜなら，子スレッドに盗まれた領域は，親のスタック領域の上方の未使用領域だけであるからである．

例えば図 2.12 のように，親スレッドと子スレッドのスタック領域が (1) のように配置されていたとする．P が親の，C が子のスタック領域を示す．スタック領域 C の下側の四角部分が，親スレッドから盗んだ領域である．ここで子スレッドが終了すると，(2) のように歯抜けの未使用領域ができるが，白いスタック領域は全体を解放でき，灰色のスタック領域だけが未使用のまま残る．この灰色のスタック領域は (3) で，親スレッドのスタック領域が縮み，スタック領域 P の下側の部分だけが残ったときに，親スレッドのスタック領域に連結されて以後使われる．

2.13.4　その他の関連研究について

StackThreads/MP や lazy threads 以外にも，多数のスレッドを効率よく管理する手法の研究は数多くある．

Lazy Task Creation [2] は本手法で採用している work steal スケージューリング・アルゴリズムに加え， 2.12 で言及したスタック領域をコピーする方法を lisp 処理系上で提案している．Lisp ではスタック領域内を指すポインタは存在せず，スタック領域のコピー時にポインタ値の変更をする必要がないため，この手法は有効である．一方，lazy threads では C 言語上での使用を想定しており，スタック領域をコピーしない．このためポインタ値の変更をする必要がなく，メモリをコピーするコストもない．

Cilk-5 [4] は work steal スケジューリングを行う並列計算言語である．lazy threads や本論文の提案手法と異なり，スレッドが作成されるごとにそのスレッドの実行に十分な大きさのスタック領域を割り当てる．例えば再帰的にスレッドが作られるようなプログラムでは，スタック領域が余っている状態で子スレッドを多数生成するので，使われていないスタック領域が大量に発生する．lazy threads ではスタック領域の余っている部分を子スレッドに割り当てるので，よりメモリ効率がよい．

Tascell [5] は，ある論理コアが他の論理コアからタスクを要求されると，ある過去の時点へバックトラックしタスクを生成する仕組みの並列計算言語である．lazy threads や本論文の提案手法と異なり，親子のタスクが同時並行に動くことができないが，スレッドの生成と管理のコストが非常に小さい．

作業空間をある過去の時点へ復元してコピーすることを特徴とする並列計算言語である．Pentomino パズルの全解探索アルゴリズムの並列化を例にすると，tascell ワーカが他のアイドル状態のワーカからタスクを要求されると，作業空間 (パズルの盤面) を過去の時点に復元してコピーし，タスクを生成する．ワーカは自分のタスクを復元前に戻し計算を再開し，また

他方のワーカ上では新規タスクはコピーした盤面でパズルのピースの別の置き方を計算する．コピーのコスト削減しているが，復元処理はプログラマが書く必要がある．

Lithe [6] は intel TBB と openMP を性能を損なうことなく同時に利用することを目指した研究である．複数のスレッドライブラリを同時に利用すると競合が発生しパフォーマンスが落ちる可能性がある．著者らが作成した Lithe をスレッドライブラリの下層に挿入することによって，アプリケーションコードを変更することなく複数のスレッドライブラリをパフォーマンスを落とすことなく組み合わせる事ができる．また，Lithe の利用者はスタック管理に遅延スレッド方式を使うような最適化を行う事ができる．

Dynamic Feedback [7] は実行時情報ロックの取得と解放に関係する部分の処理を高速化するコンパイラの研究である．このコンパイラは同期に関していくつかの戦略に基づいたコードを生成し，実行時に得られた情報によってどの戦略のコードを実行するかが選ばれる．

Lazy binary-splitting [8] は，loop の並列化時のスケジューリング方式を改良する研究である．

Varuna [9] は構成が変化しやすい計算マシンで計算速度の低下を防ぐための研究である．Varuna は posix スレッドか intel TBB を使用しているプログラムであれば，ソースコードを書き換えることなく，コンパイル時に Varuna ライブラリをリンクするだけで利用できる．

2.14　2 章のまとめ

プログラマの負担を減らしつつ並列計算の高速化を行うために，スレッド機構の最適化が必要である．そのような最適化の対象として，スレッドの速度の向上とスレッドの消費メモリの削減の二つがある．大量のスレッドを生成するようなプログラムを高速に動作させるためには，スレッド一つ一つが高速に動作する必要がある．同時に，スレッド一つ一つのメモリ消費量を削減することも重要である．なぜなら，大量にスレッドを生成するとそれに比例してメモリ消費量も増えてしまうからである．

スレッドのメモリ消費を削減する方法として，スタック領域をリンクリスト化する方法がある．この方法ではスタック領域が不足するたびにリンクリスト化されたスタック領域に新しい領域を継ぎ足す．こうすることで大きすぎるスタック領域を確保するのではなく，必要に応じた大きさのスタック領域だけで実行することができる．

リンクリスト化されたスタック領域では，各ノードの大きさを変えることでスレッドの速度とスレッドの消費メモリが変化するが，この二つの間にはトレードオフの関係がある．ノードのサイズを大きくするとスレッドの実行速度が上がるが，消費メモリが増えてしまう．逆に，ノードのサイズを小さくするとメモリ消費量が削減されるが，スレッドの実行速度が遅くなってしまう．

lazy threads [1] はスタック領域の盗み出しによってスレッドのメモリ消費を抑える手法である．この手法もスタック領域のリンクリスト化を行なっているが，各ノードには小さいメモリを割り当てている．よって，この手法ではスレッドの実行速度の低下が予想される．３章ではこの問題の解決を試みる．

第 3 章

スレッドの高速化と省メモリ化を行う手法の提案

本研究の提案は，lazy threads 方式のメモリ割り当てを高速化するためスタックブロックの大きさを大きくし，スタック領域が足りなくなり追加のメモリ確保が行われる回数を減らすことである．さらに fair-use の手法を提案する．この手法は，子スレッドの終了時に発生する大きな空き領域を，親スレッド以外が再利用できる状況を作りだす手法を提案する．提案手法では子スレッドと親スレッドが別々のプロセッサ上で並列に動作している場合に子スレッドが終了したとき，終了した子スレッドの領域が任意のスレッドから再利用可能になる．例えば，図 3.1 では，(1) で 終了したスレッドである child thread of thread 0 があり，thread 1 の直

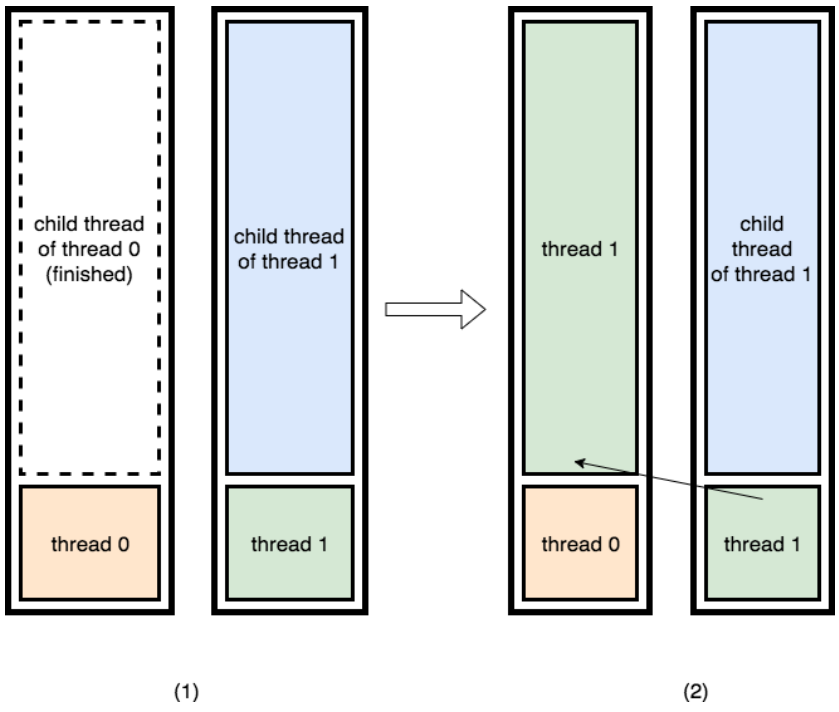

図 3.1. 終了した子スレッドの領域が再利用される場合

上に自身の子スレッドを作成し一時停止していたとする．そして， thread 0 は 自身の終了した子スレッドの領域を再利用できない状況，具体的にはスケジューリングによって steal されて動作が再開してしまった場合であったとする．この状況下で， thread 1 の動作が再開すると thread 1 は自身の真上にスタック領域を伸ばすことができない．提案手法ではこのような場合でも新規のスタックブロックを確保せずに，終了して fair-use 領域となった child thread of thread 0 のスタック領域を再利用し，メモリ消費を抑えることができる．lazy threads 方式ではスタックブロックの大きさを大きくすると，再利用されないメモリ領域が大きくなってしまう．具体的には，子スレッドが終了したときに発生する空き領域がスタックブロックの大きさに比例して大きくなるが，直下に親スレッドのスタックトップがない場合に再利用されず，メモリ消費が増えてしまう．

本研究では，提案手法と lazy threads 方式で用いられる，リンクリスト状に連結されたスタック領域の管理に gcc の split stack 機能を用いる．Split stack は関数呼び出し時にスタック領域の残りサイズを計算し，不足が検出された場合に新たにメモリ領域を確保し，既存のスタック領域に追加する．

3.1 アルゴリズム

提案する方式では，スレッドの切り替えはレジスタの退避によって行い，スケジューリングには work steal アルゴリズムを利用する．その上で，スタック領域の管理に lazy threads[1] の手法に加え，fair-use の手法，つまり提案手法である終了した子スレッド領域の再利用を行う．以下でその手法のアルゴリズムについて説明する．

スレッドを生成すると親スレッドは一停止する．このとき thread entry と呼ぶデータ構造を作ってレジスタ等を退避し，生成する子スレッドの実行を開始する．作成した thread entry は論理コアごとの deque に入れて管理し，スケジューリングをおこなう．親スレッドの一時停止時に退避した entry は親スレッドを実行していた論理コアの deque の末尾に push する．子スレッドが終了したとき，親スレッドの実行が一度も再開されていない場合は，親を deque から取り除き，親の実行を再開する．そうでない場合は以下のように work steal スケジューリングとスタック領域の管理を行う．各論理コアは自分の deque の末尾から thread entry を取り出す．もし，自分の deque が空なら，空でない他の論理コアの deque の先頭から thread entry を取り出す．論理コアは，取り出した thread entry 中の退避されたレジスタなどを復元し，そのスレッドを再開する．さらに，使用していたスタック領域を fair-use 領域としてメモリプールの中に登録する．

子スレッドのスタック領域の割り当ては，本方式では lazy threads 方式と同様，以下のように行う．子スレッドのスタック領域には，親スレッドのスタック領域の上方の余りを使う．上方とは，スタックの成長方向のことを指す．上方の余りとは，親スレッドのスタックポインタ (以下 sp) からマージン領域のサイズ分進んだ位置から，親スレッドの元のスタック領域の上限の間の領域のことである．マージン領域は後述する split stack の動作のに使用するスタック領域である．子スレッドのスタック領域の開始位置が決まったら，その位置を親スレッドの

スタック領域の新しい上限とする．親が再開する前に子が終了した場合は，親のスタック領域の上限を増やし，子が使用していたスタック領域を親に返却する．親が再開した後で子が終了した場合も子が使用していたスタック領域は親に返却するが，それは後述するように親のスタック領域が縮んで，子スレッドが使用していた領域と連続領域となったときである．

子スレッドの実行中に親スレッドの実行を再開するときは，まず fair-use 領域が登録されているメモリプールの中に親スレッドの再開時に必要なサイズを持つメモリ領域があるかを調べ，あればそのメモリ領域を短くなった既存のスタック領域と連結して利用する．そのようなメモリ領域が見つからなかった場合，新しいメモリ領域を確保し，それを短くなった既存のスタック領域と連結して利用する．分割されたスタック領域を連結して一つのスタック領域として用いるために，gcc の split stack 機能を用いる．この機能により，関数呼び出しのたびに，呼び出す関数の実行に必要なスタック領域の大きさ (以下 $frame\text{-}size$) と，スタックポインタ sp とスタック領域の上限の間の大きさを比較する．比較の結果，$frame\text{-}size$ 分の領域が sp より上にある場合はそのまま関数を実行する．領域がない場合は，まずスタック領域の上限のすぐ上に終了した子スレッドのスタック領域があれば，その分だけスタック領域の上限を増やす（このとき子スレッドが盗んだスタック領域が親に返却される）．その上で再び $frame\text{-}size$ と，sp とスタック領域の上限の間の大きさを比較し，$frame\text{-}size$ 分の領域が sp より上にある場合は呼ばれた関数を実行する．領域がない場合，まず fair-use 領域が登録されているメモリプールの中に親スレッドの再開時に必要なサイズを持つメモリ領域があるかを調べ，あればそのメモリ領域を短くなった既存のスタック領域と連結して利用する．そのような領域が得られなかった場合，新たにメモリ領域を確保し，既存のスタック領域に追加して連結する．確保するメモリの大きさは，固定長の小さなものか，それが $frame\text{-}size$ 未満なら，$frame\text{-}size$ と split stack のためのマージン領域の和である．呼ばれた関数は追加されたメモリ領域をスタック領域の一部として実行される．

関数の実行が終了して戻るとき，その関数の呼び出し時に既存のスタック領域にメモリ領域を追加して連結していた場合，追加した領域をスタック領域から外す．外したメモリ領域の一部が子スレッドやその子孫のスレッドにより使用されていなければ，そのメモリ領域を解放する．

3.2　実装

3.2.1　ユーザーランドスレッドの実装

コンテクストスイッチの実装

ユーザーランドスレッドライブラリのコンテクストスイッチは 2.5 の協調的マルチタスク向けのものを行うようになっており，実装も 2.5 で解説されているものと同様である．callee saved register 群の保存と復元を行う．

work deque

これはプロセッサ毎，つまり worker 毎に用意される両端キューである．両端キューの中身は 中断された，もしくは未実行のスレッドであり，中断されたスレッドの保存されたレジスタ群へのポインターもしくは関数ポインタと引数の組へのポインタである．本研究の実装では実際は 3.2.1 で述べられている ThreadData 構造体のポインタが両端キューの中身である．

この両端キューは work steal スケジューリングアルゴリズム [2] を実装するのに使用される．work steal スケジューリングでは，各プロセッサが自身の両端キューの下端側からスレッドを取り出し実行し，自身の両端キューが空になった場合は他のプロセッサの両端キューの上端からスレッドを盗む．

このスケジューリングアルゴリズムでは，一つの両端キューが複数のスレッドからアクセスされることになるので適切に排他処理を行う必要がある．たとえば，複数のプロセッサが一つの両端キューの上端からスレッドを取り出そうとしたり，あるプロセッサが自身の持つ両端キューの下端からスレッドを取り出すのと，別のプロセッサがその両端キューの上端からスレッドを取り出すのが同時に起こることが考えられる．

本研究の実装では両端キューへのアクセスの前にプロセッサがロックを取得することで排他的に両端キューにアクセスするようにする．ロックは各プロセッサの持つ両端キューごとに取得するようになっている．ロックを行うのはすべてのアクセスにおいて，つまり両端キューの下端へのスレッドの挿入，下端からのスレッドの取り出し，上端へのスレッドの挿入，上端からのスレッドの挿入，の各操作に対してである．本研究は Linux 上で動作することを想定しているが，Linux 上では，ロックがアトミック変数を用いて高速に動作するように実装されていることが期待される．Linux にはアトミック変数を用いたロックの実装を支援する futex システムコールが存在しているからである．また，このアトミック変数と futex システムコールを用いたロックは競合が発生しない限り，カーネルランドを経由せずにユーザーランドのみで処理が完結するので高速に動作する．アトミック変数と futex システムコールを併用するロックでない，単純なロックの実装では第 2 章で説明したように，一般にはカーネルランドを経由し自分以外のプロセッサを一時停止する処理などを行う必要があるため，アトミック変数を利用したロックよりも低速である．ただし，アトミック変数と futex システムコールを併用し実装されたロックであっても，ロックの競合が発生した場合は単純なロックの実装と同様に低速になる．

他の研究，例えば cilk5 [4] では両端キューの下端へのスレッドの挿入時にはロックを行わず，また，下端からの取り出しも上端からの取り出し (スレッドの steal) と同時に発生したことが検出されない限りロックを実行しないようなアルゴリズムが考案されているが，本研究の実装では実装を簡略化するために，ロックが競合のない場合に高速に動作することを期待して，両端キューの操作ごとにロックを行なっている．

worker

worker はプロセッサ毎に作成され， 3.2.1 で述べられている両端キューからスレッド (スレッドライブラリによって提供されるユーザーランドスレッドを指す) を取り出し実行する処理者である．この worker はネイティブスレッドをプロセッサ個数分作成し，各ネイティブスレッド上で上記の処理を行うことで実装される．worker がスレッドを開始すると，まず worker のコンテクスト (現在のレジスタ群) を保存してから，最初のスレッド (両端キューの下端から取り出されたもの) を開始する．単純な実装では，スレッドの切り替え時に一度 worker のコンテクストを復元してからスレッドの両端キューからの取り出し及び取り出したスレッドへの切り替えを行うが，本研究の実装では，スレッドの切り替え時にはスレッド内でスレッドの両端キューからの取り出しと切り替えを行う．このような実装は他のユーザーランドスレッドライブラリ [10] でも採用されている高速化手法である．この実装では，スレッドの切り替え時に行われるレジスタ群の保存と復元はあるスレッドから別のスレッドへの 1 回だけ行われるので，単純な実装より高速に動作するはずである．単純な実装では，スレッドから worker へと，worker から切り替え先のスレッドへの 2 回レジスタ群の保存と復元が行われてしまい非効率である．本研究で実装したスレッドライブラリでは，すべてのスレッドが終了した時点で worker コンテクストに戻る．ただし，本実装では一番目に作られたスレッドが終了した段階ですべてのスレッドが終了したものと見なしている．

スレッドの切り替え，つまりレジスタ群の保存と復元が行われたあとには，切り替え前にプロセッサ上で実行していたスレッドに対して処理を行う必要がある．その処理は，切り替え前に実行していたスレッドが終了していた場合はそのスレッドのスタック領域の解放を行い，終了していない場合はそのスレッドを両端キューの下端に挿入することである．

スレッドを表すオブジェクト

スレッドを表すオブジェクトである `ThreadData` 構造体が用意されている．これは，レジスタ群を保存しておくためのメモリ領域と，スレッドとして実行したい関数の関数ポインタおよびスレッドとして実行したい関数に渡す引数 (ポインタ) を持つ．また，この `ThreadData` 構造体の示すスレッドが開始前なのか中断したものなのかを表すフラグ変数を持つ．関数ポインタは `void*` とスレッド開始前にプロセッサで動作していたスレッドの `ThreadData` のポインタを受け取りスレッド終了後に次に実行するスレッドの `ThreadData` を返す．

スレッドライブラリのインターフェース

本研究のスレッドライブラリは以下のようなインターフェースを持つ．

```
1 template <typename Fn, typename... Args>
2 auto start_main_thread(Fn fn, Args... args);
```

この関数は main thread を開始する関数である．main thread は本ユーザーランドスレッドライブラリにおいて最初に作られるスレッドである．この関数は第一引数 `fn` として main

thread として実行したい関数を受け取る．第二引数以降は第一引数として渡した関数 `fn` に渡す引数である．戻り値は `fn` の戻り値である．この関数は main thread が終了するまでブロックする．main thread が作成するスレッド及びそれらの子孫のスレッドは cpu のコア数分の並列度で実行される．

```
1 template <typename Fn, typename... Args>
2 auto start_main_thread_with_worker_size(std::size_t worker_size, Fn fn, Args... args);
```

この関数は worker 数を指定して main thread を開始する関数である．第一引数としてスレッドライブラリの worker 数を指定する．main thread が作成するスレッド及びそれらの子孫のスレッドは cpu のコア数分の並列度で実行される．その他は `start_main_thread` と同様である．

```
1 template <typename Fn, typename... Args>
2 auto create_thread(Fn fn, Args... args);
```

この関数はスレッドを作成し開始する関数である．この関数は第一引数 `fn` としてスレッドとして実行したい関数を受け取る．第二引数以降は第一引数として渡した関数 `fn` に渡す引数である．戻り値は `fn` の戻り値の型の future オブジェクトである．future オブジェクトには `fn` の実行が完了すると `fn` の戻り値が代入される．この値は future オブジェクトを `join()` 関数に渡すことで `join()` 関数の戻り値として得ることができる．この関数は作成したスレッドの終了を待たない．この関数はユーザーランドスレッド内で呼び出される必要がある．つまり，main thread またはその子孫のスレッド内で呼び出される必要がある．

```
1 template <class Future>
2 decltype(auto) join(Future&& future);
```

この関数は `create_thread()` 関数が return した future オブジェクトをから結果を取り出す関数である．引数として future オブジェクトを受け取る．この関数は 引数で渡された future オブジェクトを return した `create_thread(fn, args...)` 関数呼び出しで作成されたスレッドの `fn` 関数の実行が終了するのを待ち， `fn` の結果を return する．この関数の戻り値の型は `decltype(fn(args...))` である．つまり，`fn` の戻り値が参照であった場合は，この関数の戻り値の型も参照型となる．

```
1 void yield(bool push_to_head = false);
```

この関数は現在のスレッドを一時停止し他の一時停止しているスレッドにプロセッサを明け渡す関数である．引数として true を渡すと現在のスレッドが両端キューの上端に挿入される．そうでない場合現在のスレッドが両端キューの下端に挿入される．その場合 `yield()` の呼び出し順によっては無限ループが発生する場合がある．

3.2.2 lazy thread の実装

gcc の split stack を用いてコンパイラの改造をせずに実装している．gcc が埋め込んだ split stack のコードがスタック領域の不足を検出すると，本研究で実装した改変版の split stack ラ

ンタイムライブラリの処理が実行される．

quick resume

本研究における lazy thread および提案手法の実装では，子スレッド生成時に一時停止した親スレッドが他のプロセッサ上で再開していない場合，子スレッドの終了時に親スレッドに return するという高速化を行なっている．これは 3.2.1 で述べられている ThreadData 構造体に ThreadData 型ポインタの `last_created_child` メンバ変数を追加することで実現している．親スレッドが子スレッドを生成するときに，中断した親スレッドを表す ThreadData の `last_created_child` に子スレッドを表す ThreadData のポインタを代入する．そして，子スレッド終了時に，次に実行するスレッドの ThreadData の `last_created_child` が子スレッド自身の ThreadData 構造体のポインタであるかどうかを確認する．もしそうであれば，次のスレッドは自身の親スレッドであり，かつ自身を生成してから他のプロセッサ上で再開していないことを示しているので，C++ の関数の return によって親スレッドに return する．もし違う場合は次のスレッドをコンテクストを復元する等して実行する．

splitstack context の保存と復元，初期化

本研究の lazy thread 及び提案手法の実装では，スレッドの一時停止時にレジスタ群を保存する前に split stack のコンテクストを保存している．split stack のコンテクストとは，現在のスレッドのスタック領域の上限アドレスなどである．これは split stack のランタイムライブラリの `__splitstack_getcontext()` 関数を使用して行なっている．この `__splitstack_getcontext()` は本実装では改造を行なっていない．また，一時停止したスレッドの再開時にレジスタ群を復元したあとに，split stack のコンテクストを復元している．これは split stack のランタイムライブラリの `__splitstack_setcontext()` 関数を使用して行なっている．この `__splitstack_setcontext()` 関数は本実装ではデバッグ出力の追加以外の改造は行なっていない．

3.2.3　lasy thread と提案手法の実装のための split stack ランタイムライブラリの改造及び処理の追加

__generic_morestack() 関数の改造

`__generic_morestack()` 関数は split stack のランタイムライブラリの関数であり，新規スタックの確保とスタック経由で渡される関数の引数を古いスタックから新しいスタックへコピーする処理を行う関数である．戻り値として新規スタックのアドレスを返す．このランタイム関数は，関数の呼び出し時にスタック領域が上限を超えてしまうと判断された場合に呼ばれるものである．

本研究の lazy thread 及び提案手法の実装では，このランタイム関数を新規のスタックを確保する前に，現在使用中のスタック領域の上方に未使用領域がある場合にそれを自分自身の領域にマージする処理を行うように変更した．実際の実装では，後述の 3.2.3 で説明されている

merge_finished_children_stack() 関数を呼んでいる．マージが行われ十分なスタック領域が確保された場合，新規スタックの確保は行われないので，戻り値として 0 を返すように変更している．この 0 を返す仕様は改造前の split stack にはないものであり，この変更に合わせて，__generic_morestack() 関数を呼び出す__morestack() 関数 (cpu アーキテクチャ毎にアセンブリ言語で実装されている，本研究では x64 アーキテクチャ向けコードのみを対象としている) の実装も変更した．

__generic_releasestack() 関数の改造

__generic_releasestack() 関数は split stack のランタイムライブラリの関数であり，引数として渡されたスタックを解放する関数である．解放するスタックは __generi_morestack() 関数で確保されたものでなくてはならない．

本研究の lazy thread 及び提案手法の実装では，この関数にメモリブロックの参照カウンタを下げる処理を行うように変更した．lazy thread 及び提案手法では，確保されたメモリブロックは複数のスレッドによって使用される．よって，本実装ではメモリブロックの解放タイミングを参照カウンタで管理している．

merge_finished_children_stack() 関数の追加

本研究の lazy thread 及び提案手法の実装では，このランタイム関数を新規のスタックを確保する前に，現在使用中のスタック領域の上方に未使用領域がある場合にそれを自分自身の領域にマージする処理である merge_finished_children_stack() を新たに追加した．この関数は現在のスタック領域のすぐ上方にあるスレッドのスタック領域があり，かつそれが終了したスレッドのものであるとき，これを現在のスタック領域にマージする．この処理をマージが不可能になるまで繰り返す．ただし，マージ対象スレッドが終了していない子スレッドを持つ場合は，その領域を除いてマージしなくてはならない．

allocate_segment_from_current_segment()

この関数は現在のスレッドのスタックの未使用部分を盗み出して返却する関数である．この関数はまず merge_finished_children_stack() 処理を実行したのち，現在の現在のスレッドのスタックの未使用部分を切り出す．この切り出しは，現在のスタック領域を表すメタデータ中の，スタックサイズを格納する変数を更新し，未使用部分を指すメタデータを新たに作成することで行われる．ただし，得られた領域のサイズが要求されたスタック領域サイズかスタック領域の最小サイズのうち大きい方の値を下回ってしまう場合，この関数は失敗したものとして 0 を返却する．3.2.1 で述べられている create_thread() 関数内から間接的に呼ばれる．

3.2.4　提案手法の実装

提案手法では，子スレッドが終了するとき，quick resume かどうかを判別し，もしそうならば親スレッドは他のプロセッサに盗まれたと判断し，子スレッドの使用していたスタック領域はメモリプールの中にいれられる．

また，スレッド作成時に親スレッドから領域の盗み出しに失敗したら，メモリプール中に要求される大きさのスタック領域があるか調べ，あればそれを使う．なければ新規にメモリを確保する．

第 4 章

実験

実験として，二つのマイクロベンチマークを実装し実験した．実験ではスタックブロックのサイズを増やすとて提案手法では実行時間が増えず，既存手法では実行時間が増えることを期待している．実験は Intel Xeon CPU E5-2637 を搭載するマシンで行なった．このマシンの cpu は 8 コア 16 スレッドである．またメモリは 512GB 搭載している．

4.1 ベンチマーク 1: 行列演算付き並列フィボナッチ数列

マイクロベンチマークの一つは行列演算付き並列フィボナッチ数列計算である．これは Listing 4.1 で記述されている疑似コードで示されているものである．実験では，このベンチマークでは提案手法と既存手法でメモリ消費量が変化しないことを確認するのが目的である．このベンチマークで 20 番目のフィボナッチ数を二つのプロセッサで並列に計算するときにスレッドとそのスタックがどのような挙動をするのかを説明する．以下 n 番目の フィボナッチ数を $fib(n)$ とする．このベンチマークでは，$fib(n)$ の計算は，再帰関数呼び出しに近い形で行われる．$fib(n)$ の計算時には $fib(n-1)$ をスレッドを作成し計算し，$fib(n-2)$ を関数呼び出しとして実行し計算する．ベンチマークプログラムが開始されると，各プロセッサ毎に worker が作られる．worker はスレッドを実行するプログラムである．その後，ベンチマーク上では最初のスレッドが作成される．このスレッドは worker 0 上で実行される．このスレッドが作成されるときにスタックブロックが確保され，スレッドにスタック領域となる．このスレッドがこのスタック領域上で $fib(20)$ を計算する．$fib(20)$ の計算では，まず $fib(19)$ を計算する子スレッドが作られる．この子スレッドのスタック領域は $fib(20)$ を計算する親スレッドの未使用領域上に割り当てられる．work steal スケジューリングアルゴリズムにより $fib(19)$ は worker 0 上で実行される．本研究では提案手法と既存手法を実装には work steal スケジューリングアルゴリズムを採用している．$fib(19)$ が作成されたのち，子スレッドを作成し中断した $fib(20)$ の計算スレッドを，worker 1 が再開する．これも work steal スケジューリングアルゴリズムによる．再開後には $fib(20)$ の計算スレッドは $fib(18)$ を関数呼び出しとして計算する．このとき，現在のスタック領域には未使用領域はないため，新たにスタックブロックが確保される．なぜならば，$fib(20)$ のスタック領域の未使用領域はすで

に $fib(19)$ を計算する子スレッドに割り当てられてしまったからである．この後，$fib(18)$ は $fib(17)$ をスレッドとして実行し，$fib(16)$ を関数呼び出しとして実行する．$fib(16)$ の関数呼び出し時にも，$fib(18)$ の計算時と同様に新規のスタックブロック確保が行われると予想される．これは既存手法であっても，メモリ消費量削減を目指し既存手法を改良した提案手法であっても変わらないと予想される．なぜ提案手法でメモリ消費量が改善されないのかを説明する．提案手法では，もし $fib(19)$ の計算スレッドが終了していた場合は，$fib(16)$ を関数呼び出し時にスタックブロックを確保する前に，終了したスレッドが使用していた領域をスタック領域として利用しようとする．しかし，$fib(19)$ の使用していた領域が再利用されることがないと予想される．なぜならば，$fib(19)$ は $fib(16)$ の計算が開始する前に終了していないことが予想されるからである．$fib(19)$ が終了しないのは，計算に時間がかかる処理だからである．このベンチマークでは，各 $fib(n)$ の計算時に行列の加算が一度行われる．これは実際のアプリケーションを想定し，計算が瞬時に終了しないような挙動を再現するためである．よって $fib(16)$ の開始と $fib(19)$ の終了には誤差の範囲では収まらない明確な時間差があると予想される．明確な時間差があるとは，例えば本研究で実装したスレッド機構のスケジューリング処理や，オペレーティングシステムのスケジューリング処理などによって，スレッドの開始や再開のタイミングがずれたとしても，それによる時間差が無視できるほど大きい時間差がベンチマークの各計算で行われるということである．さらにこの後もスレッドの生成と関数呼び出しが繰り返されるが，上の議論と同様に，提案手法でも既存手法でもメモリ消費は変化しない．

4.1.1　実装の詳細

以降でこのベンチマークの詳細を説明する．このベンチマークでは 64×64 の大きさの行列を用いる．行列の各要素は `double` 型である．この実験は Listing 4.1 中の `fibo()` 関数を再帰的に子スレッドとして実行していくものである．`fibo()` 関数の説明をする．この関数は引数として数値 `n` と行列 `mat` と，出力用行列 `mat_out` を受けとる．関数内の処理は，まず関数のスタックフレーム内にローカル変数として二つの行列を用意する．これらを変数 `ret1` と `ret2` とする．次に，引数 `n` が 0 か 1 であった場合は何もせず `n` を return し関数を抜ける．そうでない場合は，まず `fibo(n - 1, mat, ret1)` をスレッドを実行するスレッドを作成する．作成したスレッドが行なった行列演算の結果は `ret1` に格納される．つぎに，`fibo(n - 2, mat, ret2)` を関数呼び出しとして実行する．呼び出し先が行なった行列演算の結果は `ret2` に格納される．そして，スレッドとして実行した `fibo(n - 1, mat, ret1)` と `fibo(n - 2, mat, ret2)` の行なった行列演算の結果である `ret1` と `ret2` を足し合わせ出力用行列 `mat_out` に代入する．最後に作成したスレッドが終了するのを待ち，スレッド `fibo(n - 1, mat, ret1)` の戻り値と関数呼び出し `fibo(n - 2, mat, ret2)` の結果を足し，それを return する．

Listing 4.1. フィボナッチベンチマークの疑似コード

```
1  Mat as matrix double[64 * 64];
2
3  def fibo(n, mat, mat_out) {
4    ret1, ret2 as Mat in stack frame;
5
6    if ( n == 0 || n == 1) {
7      return n;
8    }
9
10   child = create_thread fibo(n − 1, mat, ret1);
11   n2 = fibo(n −2, mat, ret2);
12
13   mat_out = ret1 + ret2;
14   r = n2 + join(child);
15
16   return n;
17
18 }
```

4.1.2 実験結果

この実験を n を 20 として，スタックブロック (新規に確保されスタック領域に追加されるメモリ領域) のサイズを 8kb，16kb，64kb，2048kb として行なった．このベンチマークでは worker 数を 16 として実行した．つまりプロセッサが 16 個同時に利用される．また，スタックブロックのアロケーションには malloc を使用している．ベンチマーク実行時のメモリ消費量の推移のグラフが，図 4.1 4.2 4.3 4.4 である．また，このベンチマークのメモリ消費量の最大値と実行時間 (10 回動かした平均) をまとめた表が 4.1 と 4.2 である．さらに，スタックブロックのアロケーションに malloc を使用するようにした同じベンチマークを実行した．そのベンチマークのメモリ消費量の最大値と実行時間 (10 回動かした平均) をまとめた表が 4.3 と 4.4 である．

スタックサイズ	lazy-threads	proposal
8kb	25194496	25935872
16kb	28577792	25853952
64kb	38760448	36388864
2048kb	58679296	46993408

表 4.1. malloc を使ったフィボナッチベンチマークの最大メモリ消費量 (byte)

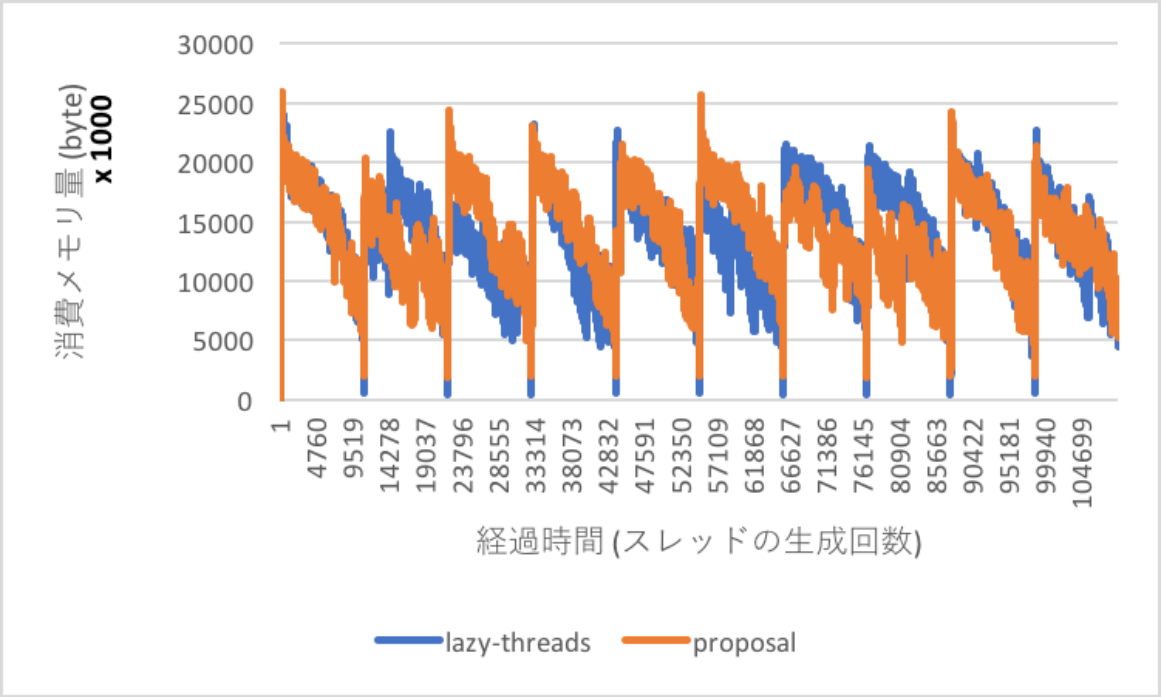

図 4.1. フィボナッチベンチマークのメモリ消費量の推移 (スタックブロックサイズ 8kb)

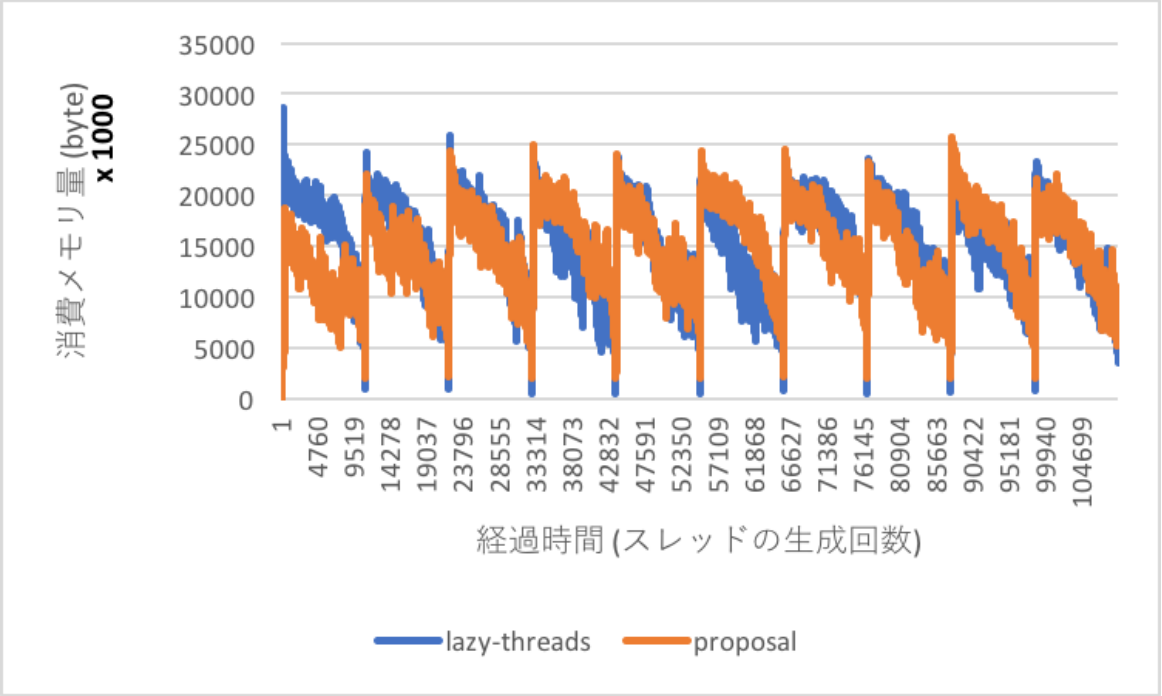

図 4.2. フィボナッチベンチマークのメモリ消費量の推移 (スタックブロックサイズ 16kb)

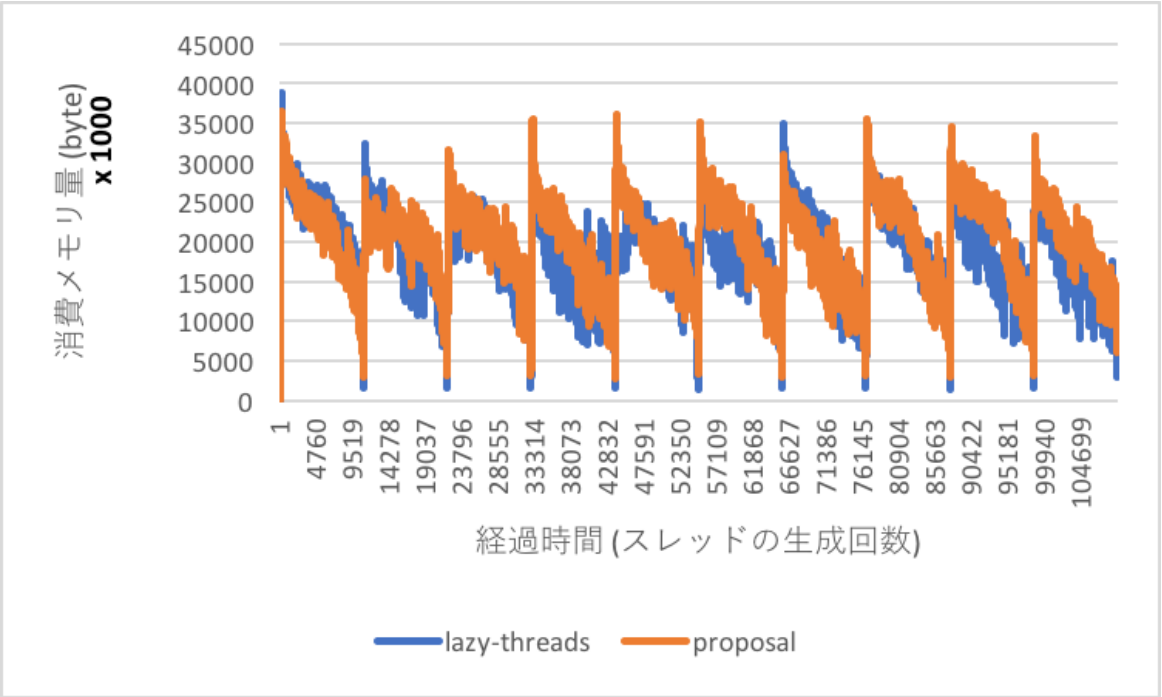

図 4.3. フィボナッチベンチマークのメモリ消費量の推移 (スタックブロックサイズ 64kb)

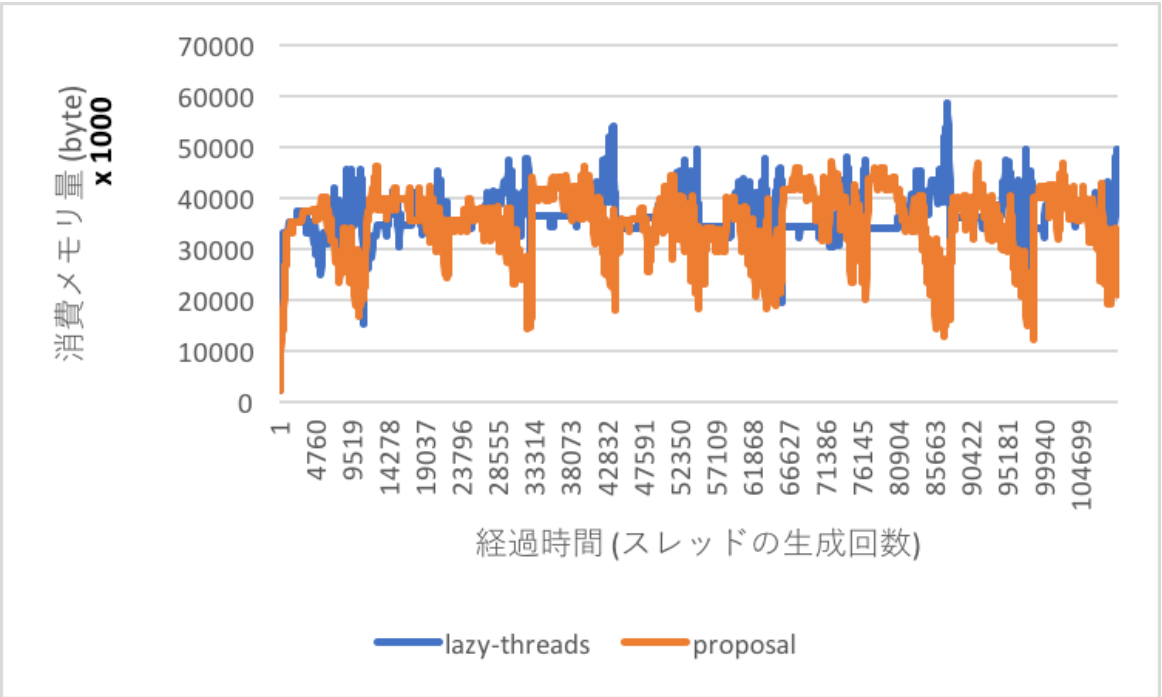

図 4.4. フィボナッチベンチマークのメモリ消費量の推移 (スタックブロックサイズ 2048kb)

スタックサイズ	lazy-threads	proposal
8kb	121	142
16kb	100	122
64kb	144	108
2048kb	63	96

表 4.2. malloc を使ったフィボナッチベンチマークの経過時間 (ミリ秒)

スタックサイズ	lazy-threads	proposal
8kb	26230784	27275264
16kb	28352512	28823552
64kb	37896192	39342080
2048kb	58986496	53604352

表 4.3. mmap を使ったフィボナッチベンチマークの最大メモリ消費量 (byte)

スタックサイズ	lazy-threads	proposal
8kb	3060	2942
16kb	2918	3010
64kb	2924	3081
2048kb	116	104

表 4.4. mmap を使ったフィボナッチベンチマークの経過時間 (ミリ秒)

4.2　ベンチマーク 2: スレッドの生成と同期の繰り返し

マイクロベンチマークのもう一つはスレッドの生成と同期を繰り返すベンチマークである．以降このベンチマークをベンチマーク 2 と呼ぶことがある．このマイクロベンチマークは，疑似コードとして 4.2 で記述されているものである．このベンチマークは要約すると，一つの親スレッドが子スレッドの生成と，それ以前に生成したスレッドの終了待ちを再帰関数呼び出しをしながら交互に行うものである．この実験は提案手法は既存手法よりもメモリ消費量か少なくなることを観測するために行なっている．メモリ消費量か少なくなるという予想がなされるのは，提案手法では，親スレッドのメモリ消費量が既存手法より少なくなることが，期待されるからである．また，このベンチマークでは，木やグラフのトラバースを行いながら各ノードで時間のかかる処理を並列に実行するようなアプリケーションを想定して，それを再現するような処理をする．詳細に述べると，1 つの親スレッドが再帰関数呼び出しで深さ優先探索を行いながら，各ノードでは子スレッドを作成し，時間のかかる処理をさせるようなアプリケーションである．

以下で 4.2 の解説を行う．まず，親スレッドが新規に確保されたスタックブロックを全て自

分のスタック領域として実行を開始する．そして，スタック領域上に作業領域が置かれ，スタック領域を消費する．次に，親スレッドが子スレッドを作成する．このとき子スレッドのスタック領域は親スレッドの未使用領域を切り出して割り当てる．子スレッドはしばらくは終了しない．子スレッドが中断し親スレッドが動作するか，子スレッドと親スレッドが別のプロセッサ上で並列に動作することになる．次に親スレッドが関数呼び出しをして追加の作業領域をスタック領域上に確保しようとする．ただし，現在の親スレッドのスタック領域上には残り領域が少なく作業領域は確保できない．なので，新たにスタックブロックを確保し，親スレッドのスタック領域に追加する．親スレッドは追加された領域上に作業領域を確保する．そして，さらに子スレッドを作成する．この子スレッドも現在のスタックブロックの未使用領域を割り当てて動作する．そして，次の時点で初めに作成した子スレッドが終了したものとする．この子スレッドが終了した時点で，子スレッドの使用していた領域が解放され，未使用領域となる．そして，親スレッドはさらに関数呼び出しを行い作業領域を確保しようとする．再び親スレッドはスタック領域の残りが少なく作業領域が確保できないという事態が発生する．このとき，提案手法では先ほど子スレッドが終了し発生した未使用領域を親スレッドのスタック領域として再利用する．なので新たなメモリが確保されることがなく，メモリ消費が増えない．しかし，既存手法では新たにスタックブロックを確保し親スレッドの領域に追加するので，メモリ消費が増えてしまう．このあと，ベンチマークでは二番目に作成した子スレッドが終了したとする．そして，親スレッドの作業領域もさらに確保されて行く．そのあと，提案手法では終了した子スレッド領域の再利用が親スレッドにより行われ，既存手法では新規のメモリ消費が行われる．そして，子スレッドの終了，親スレッドの作業領域の追加消費，子スレッドの作成が順に繰り返され，提案手法ではメモリ消費が増えず，既存手法でのみメモリ消費が増えるということが起こると予想される．ただし，提案手法でもスタックブロックを消費しきってしまった場合はメモリ消費が起こる．本研究ではスタックブロックのサイズをいくつかのパラメータでこのベンチマークを行なっているが，提案手法ではスタックブロックのサイズを増やしてもメモリ消費量は変わらないことが予想され，既存手法ではスタックブロックのサイズを大きくするにつれてメモリ消費量が多くなることが予想される．

4.2.1 実装の詳細

以降でこのベンチマークについて詳細に説明する．

親スレッドで再帰的に実行される関数 `parent` がある．この関数は引数として `count` と `prev_child` と `prev_child_wait` の 3 つを受け取る．`count` は再帰呼び出しの残り実行回数である．`prev_child` は前回の `parent` 関数の呼び出しで作成された子スレッドを表すオブジェクトである．`prev_child_wait` は 引数で渡された `prev_child` スレッドに待ち続けるかどうか，つまり終了するべきかを伝えるフラグを表す変数の参照である．まず，`parent()` 関数はスタックフレーム内に作業用領域として サイズが 0x2000，つまり 8192 の配列を確保し全要素を 0 で初期化する．これをローカル変数 `array` とする．配列の各要素は `char` 型，つまり 8 バイトである．この作業領域 `array` は現実のアプリケーションで確

保され，なんらかの作業に用いられることを想定し，それをベンチマーク中で再現しているものとする．次に，もし count が 0 の場合は参照引数 prev_child_wait に false を設定し， join(prev_child) で引数で渡された prev_child スレッドが終了を待ち，それが終了すると parent 関数は return する．count が 0 以外の場合の処理が以降で続く．まず，child_wait ローカル変数を定義し，true で初期化する．次に子スレッドとして child() 関数を実行する．このとき子スレッドとして作成されるスレッドのスレッドオブジェクトを child_thread とする．child() 関数の定義は後述する．child() 関数には child_wait ローカル変数を渡す．これは参照渡しとなる．そして， prev_child_wait 参照引数に false を設定する．これにより 引数で渡された prev_child スレッドが終了可能となる．ついで，終了可能となった prev_child スレッドを待つ．最後に 自分自身である parent() 関数を再帰的に呼び出す．このとき，引数として， count は自身に渡された count から 1 引いたものを渡し，さらに child_wait 変数の参照と child_thread スレッドオブジェクトを渡す．以上で parent() 関数は終了である．次に child() 関数の定義を説明する．child() 関数は引数として child_wait を受け取る．これはスレッドとして実行された自分自身が終了してもよいかを表すフラグの変数であり，参照として渡される．この引数の内容は親スレッドである parent() 関数内で変更される．child() 関数は実行されると，child_wait が false になるのを待つ．変数が変更されるのを待つ処理はスレッド機構によって提供されるが，これは変数が変更されているかをチェックし，変更されていなければ自分自身を中断し，他の中断しているスレッドを再開するという処理を実行する．ベンチマークの開始時には parent_start() 関数が呼ばれる．この関数は parent() 関数と同様に，まずスタックフレーム内に作業用領域として サイズが 0x2000，つまり 8192 の配列を確保し全要素を 0 で初期化する．配列の各要素は char 型，つまり 8 バイトである．次に，child_wait ローカル変数を定義し，true で初期化する．そして， child() 関数に child_wait 変数を参照として渡し，スレッドとして実行する．このスレッドを child_thread とする．最後に parent(RECURSIVE_CALL_SIZE, child_wait, child_thread) を関数呼び出しとして実行する．RECURSIVE_CALL_SIZE は parent() 関数の再帰呼び出しの深さである．また，child_wait 変数は参照を渡す．

4.2.2　実験結果

この実験を n を再帰の深さを 125 として，スタックブロック (新規に確保されスタック領域に追加されるメモリ領域) のサイズを 8kb，16kb，64kb，2048kb として行なった．このベンチマークでは worker 数を 2 として実行した．つまりプロセッサが二つ同時に利用される．また，スタックブロックのアロケーションには malloc を使用している．ベンチマーク実行時のメモリ消費量の推移のグラフが，図 4.5 4.6 4.7 4.8 である．また，このベンチマークのメモリ消費量の最大値と実行時間 (10 回動かした平均) をまとめた表が 4.5 と 4.6 である．さらに，スタックブロックのアロケーションに mmap を使用するようにした同じベンチマークを実行した．そのベンチマークのメモリ消費量の最大値と実行時間 (10 回動かした平均) をまとめた表

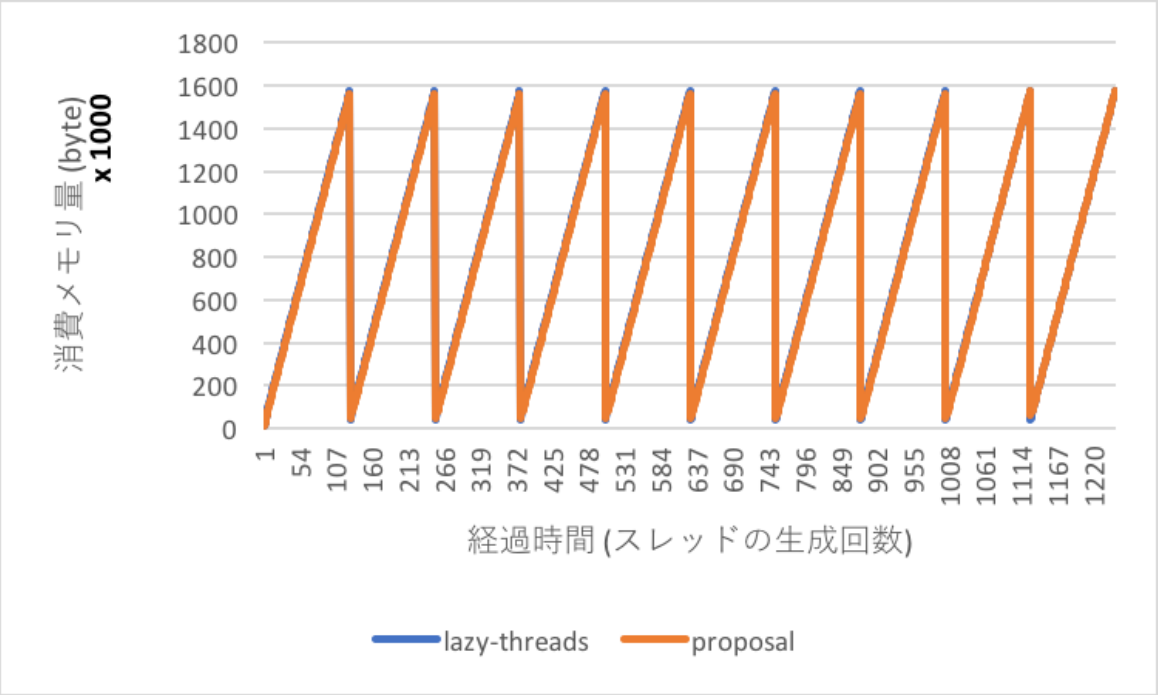

図 4.5. ベンチマーク 2 のメモリ消費量の推移 (スタックブロックサイズ 8kb)

が 4.7 と 4.8 である.

スタックサイズ	lazy-threads	proposal
8kb	1572864	1572864
16kb	2097152	1564672
64kb	5300224	1732608
2048kb	135254016	7827456

表 4.5. malloc を使ったベンチマーク 2 の最大メモリ消費量 (byte)

スタックサイズ	lazy-threads	proposal
8kb	0	2
16kb	1	2
64kb	7	3
2048kb	1	2

表 4.6. malloc を使ったベンチマーク 2 の経過時間 (ミリ秒)

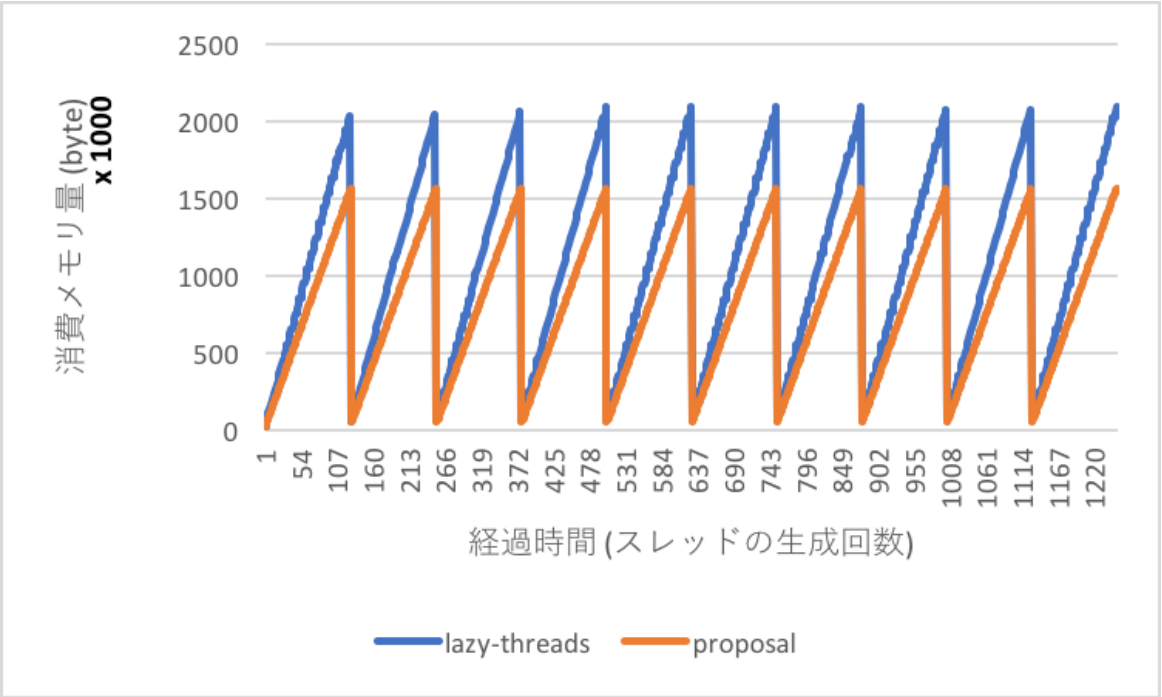

図 4.6. ベンチマーク 2 のメモリ消費量の推移 (スタックブロックサイズ 16kb)

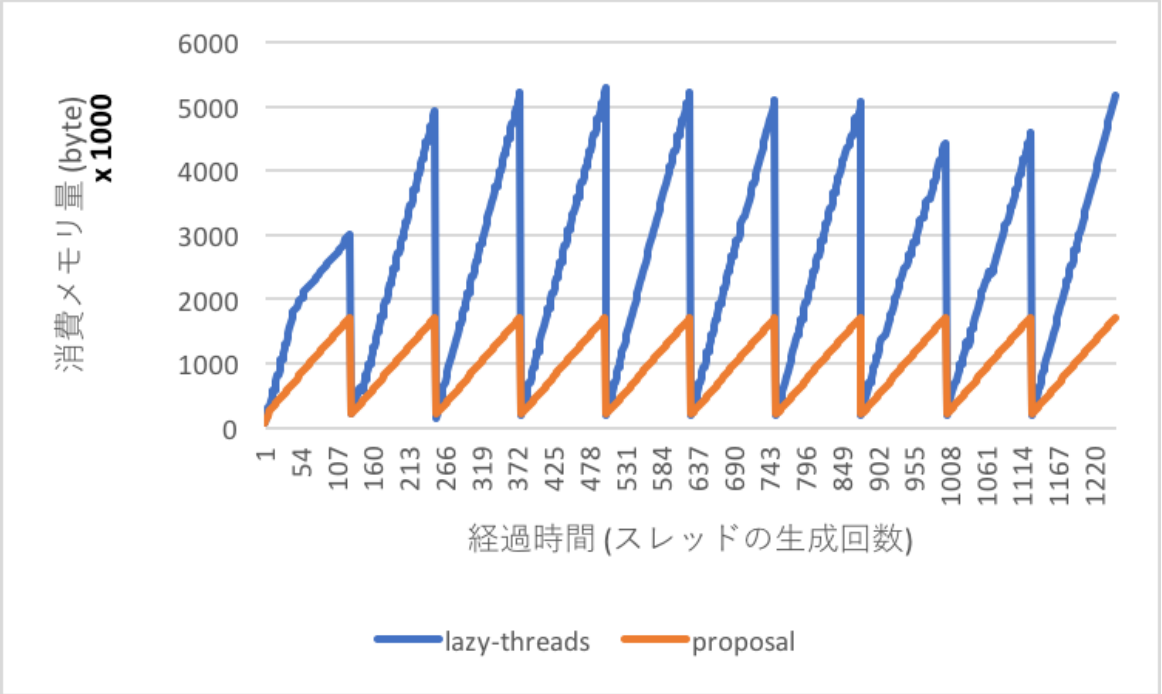

図 4.7. ベンチマーク 2 のメモリ消費量の推移 (スタックブロックサイズ 64kb)

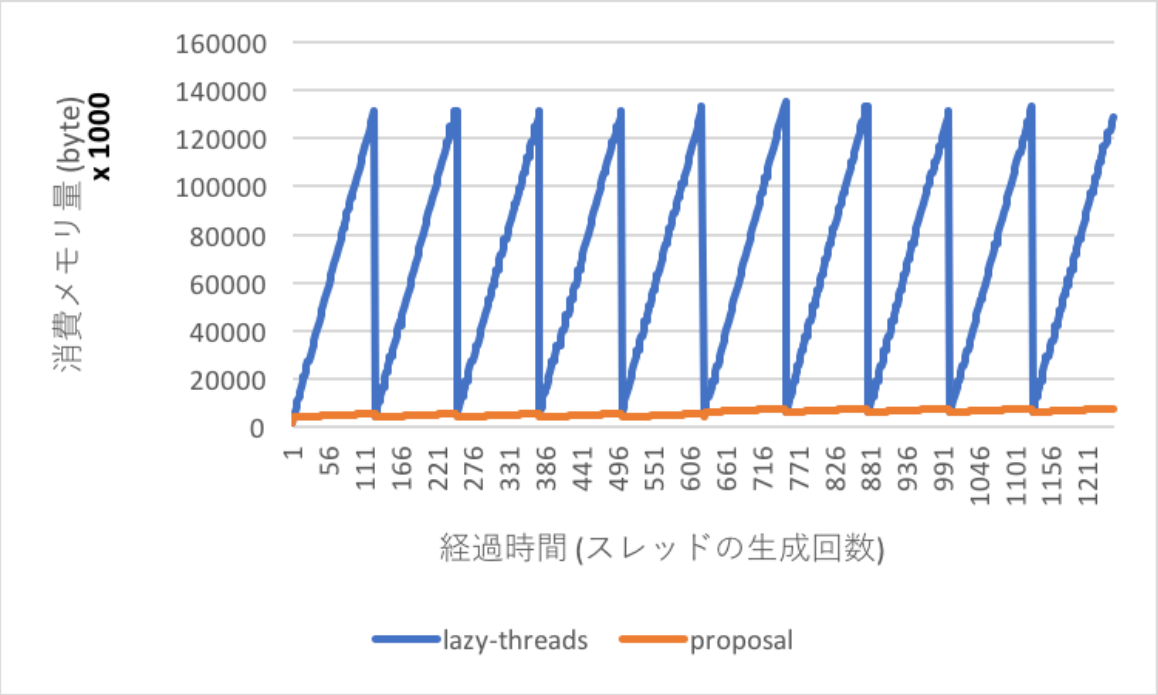

図 4.8. ベンチマーク 2 のメモリ消費量の推移 (スタックブロックサイズ 2048kb)

スタックサイズ	lazy-threads	proposal
8kb	1572864	1560576
16kb	2084864	1564672
64kb	5287936	1732608
2048kb	137342976	5726208

表 4.7. mmap を使ったベンチマーク 2 の最大メモリ消費量 (byte)

スタックサイズ	lazy-threads	proposal
8kb	4	2
16kb	4	1
64kb	2	3
2048kb	2	2

表 4.8. mmap を使ったフィボナッチベンチマークの経過時間 (ミリ秒)

4.2.3 追加実験

追加実験として，この実験を `n` を再帰の深さを 60000 として，スタックブロック (新規に確保されスタック領域に追加されるメモリ領域) のサイズを 8kb から倍にしていき 65536kb まで変化させて行なった．このベンチマークでは worker 数を 2 として実行した．つまりプロ

Listing 4.2. ベンチマーク 2 の疑似コード

```
1
2  def parent(count, prev_child, &prev_child_wait) {
3
4    array as char[0x2000] = {};
5
6    if (count == 0) {
7      prev_child_wait = false;
8
9      join(prev_child);
10
11     return;
12   }
13
14   child_wait = true;
15
16   child_thread = create_thread child(child_wait);
17
18   prev_child_wait = false;
19   join(prev_child);
20
21   parent(count − 1, child_thread, child_wait);
22 }
23
24 def child(&child_wait) {
25   wait child_wait == false;
26 }
27
28 def parent_start() {
29
30   array[0x2000] = {};
31
32   child_wait = true;
33
34   child_thread = create_thread child(child_wait);
35
36   parent(RECURSIVE_CALL_SIZE, child_wait, child_thread);
37 }
```

セッサが二つ同時に利用される．また，スタックブロックのアロケーションには `malloc` を使用している．

追加実験の際に本スレッドライブラリの実装に変更を加え，提案手法及び遅延スレッド手法の実装の高速化を図っている．この変更で C++ 標準ライブラリの future と promice を利用していた部分を置き換えている．変更前の実装ではスレッドライブラリの実装に C++ 標準ライブラリの future と promice を使用していたが，作成した子スレッドの実行結果を親スレッドが取得する際に futex システムコールがそのつど呼ばれていた．

このベンチマークのメモリ消費量の最大値と実行時間 (10 回動かした平均) をまとめた表が表 4.9 と表 4.10 である．また，図 4.9 はベンチマーク 2(追加実験) の実行時間と最大メモリ消

費量に関する散布図である．

stack size (kb)	lazy-threads	proposal
8	737,316,864	737,316,864
16	979,689,472	737,329,152
32	1,461,424,128	737,378,304
64	2,418,286,592	737,476,608
128	4,350,603,264	737,673,216
256	8,224,813,056	738,066,432
512	16,040,583,168	738,852,864
1024	31,506,661,376	740,425,728
2048	62,833,704,960	743,571,456
4096	124,909,932,544	749,862,912
8192	245,504,090,112	762,445,824
16384	487,324,073,984	787,611,648
32768	989,713,993,728	837,943,296
65536	1,997,921,693,696	938,606,592

表 4.9. malloc を使ったベンチマーク 2 の最大メモリ消費量 (byte)

4.3 実験結果の評価

以上二つの実験の結果から以下の結論が得られる．提案手法では，スタックブロックのサイズを大きくすると，遅延スレッドよりメモリ消費量を大きく減らせるようなプログラムがある．しかし，スタックブロックのアロケーターとして `malloc` を使用する場合，期待されていたスタックブロックのサイズを大きくすることによる実行速度の改善は，提案手法，遅延スレッドともに起こらなかった．しかし，スタックブロックのアロケーターとして `mmap` を使用する場合，実行速度が改善するようなプログラムが見つけられた．`malloc` はメモリのプーリングを行い高速にメモリ確保をしていると思われるが，これによりメモリ確保のコストがそれ以外の計算と比べて無視できるほど小さくなっていると考えられる．

ベンチマーク 2 の追加実験からは，提案手法の方が遅延スレッドよりもスタックブロックの変化に対する実行時間の変化が安定していると言える．遅延スレッドを利用する場合スタックブロックのサイズを適切に選ぶ事ができれば提案手法よりも高速に動作するが，適切な値から外れた場合スタックブロックの変化に対する実行時間の変化が提案手法よりも大きい．

stack size (kb)	lazy-threads	proposal
8	1168	2068
16	1964	1486
32	1734	1614
64	3390	1699
128	1980	1754
256	2144	2046
512	3124	1810
1024	1601	1584
2048	766	1264
4096	1313	1752
8192	6270	1215
16384	11910	2003
32768	4639	1649
65536	4583	1702

表 4.10. malloc を使ったベンチマーク 2 の経過時間 (ミリ秒)

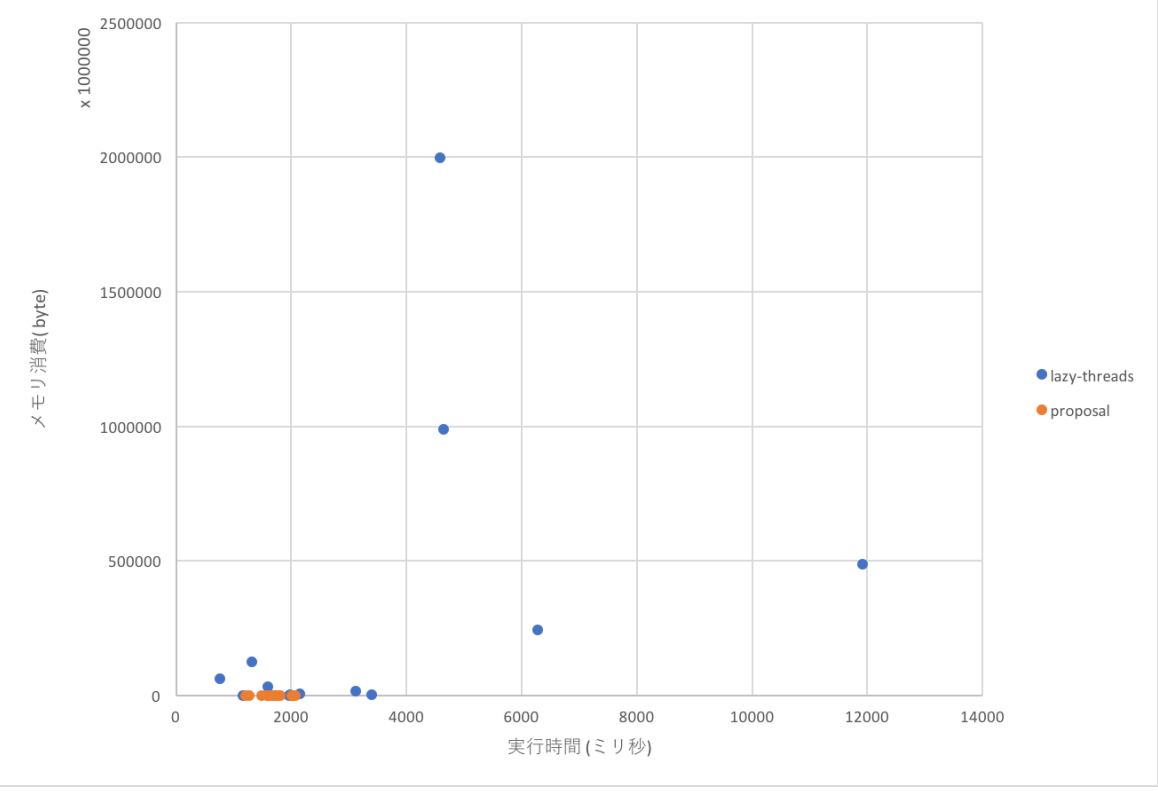

図 4.9. ベンチマーク 2(追加実験) の実行時間と最大メモリ消費量に関する散布図

第 5 章

まとめと今後の課題

5.1 まとめ

本研究では遅延スレッド [1] において，スタック領域のサイズを大きくし，それによって発生する大きな空き領域が再利用できる状況を増やす fair-use の手法を提案した．遅延スレッドは大量にスレッドが生成されるようなプログラムで利用されていないスタック領域のサイズを減らすための手法である．しかし，遅延スレッドではプログラムによっては実行速度が低下する場合が考えられる．なぜならばスタック領域が小さいメモリが連結されたリンクリスト状になっており，それらの領域を使い切ったときのメモリの確保処理が頻繁に発生しうるからである．そこで，本研究ではスタック領域のサイズを大きくし，メモリの確保回数を減らすことで実行速度の改善を狙った．また，スレッド終了時に発生する領域を場合に応じて任意のスレッドが再利用可能にする fair-use 手法を導入することで，スタック領域のサイズを大きくすることにより発生する再利用不可能な領域の増大を抑える手法を提案した．提案手法と遅延スレッドの実現において必要なリンクリスト化されたスタック領域は，gcc の split stack が提供するものを使用して，二つの手法を実装した．また，この実装を用いて二つの手法の利用時のプログラムの実行速度とメモリ使用量を比較する実験を行なった．実験の結果，スタック領域サイズが同じ場合，提案手法の方が高速に動作するという結論は得られなかった．また，スタック領域のサイズを大きくするにつれて，提案手法の方がメモリ消費量が相対的に少なくなるという結果が得られた実験と，メモリ消費量に大きな差は出ないという結果が得られた実験があった．しかし追加実験の結果，適切なスタックブロックのサイズを選ぶ事ができれば遅延スレッド手法の方が提案手法よりも高速に動作するが，選ぶ事ができなかった場合遅延スレッド手法は提案手法に比べて実行速度が悪化する現象が確認された．

5.2 今後の課題

提案手法が遅延スレッドよりも高速に動作するようなプログラムを考案し，実験を行うことが今後の課題である．本研究の実験では提案手法の方が従来手法である遅延スレッドよりも，速度面で改善されていることを示すような結果は得られなかった．

参考文献

[1] Seth Copen Goldstein, Klaus Erik Schauser, and David E. Culler. Lazy threads. *J. Parallel Distrib. Comput.*, Vol. 37, No. 1, pp. 5–20, August 1996.

[2] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pp. 185–197, New York, NY, USA, 1990. ACM.

[3] Kenjiro Taura, Kunio Tabata, and Akinori Yonezawa. Stackthreads/mp: Integrating futures into calling standards. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '99, pp. 60–71. ACM, 1999.

[4] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pp. 212–223. ACM, 1998.

[5] Tasuku Hiraishi, Masahiro Yasugi, Seiji Umatani, and Taiichi Yuasa. Backtracking-based load balancing. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pp. 55–64. ACM, 2009.

[6] Heidi Pan, Benjamin Hindman, and Krste Asanović. Composing parallel software efficiently with lithe. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pp. 376–387. ACM, 2010.

[7] Pedro C. Diniz and Martin C. Rinard. Dynamic feedback: An effective technique for adaptive computing. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pp. 71–84. ACM, 1997.

[8] Alexandros Tzannes, George C. Caragea, Rajeev Barua, and Uzi Vishkin. Lazy binary-splitting: A run-time adaptive work-stealing scheduler. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pp. 179–190, New York, NY, USA, 2010. ACM.

[9] Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. Adaptive, efficient, parallel execution of parallel programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pp. 169–180. ACM, 2014.

[10] 中島. 高効率な i/o と軽量性を両立するマルチスレッド処理系の設計と実装. Master's thesis, 東京大学, 2011.

謝辞

本研究を進めるにあたり，研究方針や論文構成，発表手法，研究アイディアなど多岐にわたって手厚くご指導賜りました指導教員の千葉滋教授に心より感謝致します．研究を進める過程で多くの困難があった中，とても大きく強い支えとなってとなってくださいました．

また、論文の執筆や研究の発表に関して多くのご指導を頂きました市川和央氏に厚くお礼を申し上げます．日々本研究の議論にお付き合い頂き，様々な示唆を与えてくださった山崎徹郎氏に感謝致します．

最後に，日頃から研究室生活やミーティングを通じてご支援くださった千葉研究室の皆様に心より感謝致します．