ユーザレベル仮想記憶 API とその応用

田浦健次朗

目次

ユーザレベル仮想記憶 API

チェックポインティング

ネットワークページング

分散共有メモリ
- ハードウェア共有メモリ
- 分散共有メモリ

インクリメンタル Garbage Collection

ユーザレベル仮想記憶 API とは

- `mprotect(`*a*`,` *len*`,` *prot*`);`
  - アドレス範囲 $[a, a + len)$ に保護属性 *prot* を設定
- `char *` *a* `= mmap(`*a'*`,` *len*`,` *prot*`,` *flags*`,` *fd*`,` *offs*`);`
  - $[a, a + len)$ を割り当てるとともに, 保護属性 *prot* を設定
  - $a' = 0 \Rightarrow a$ =OS が探した空き領域
  - $a' \neq 0 \Rightarrow [a, a + len)$ が空いていれば $a = a'$
- *prot* — `PROT_READ`, `PROT_WRITE`, `PROT_EXEC` (それらの bit 和)
- `sigaction(SIGSEGV,` *act*`,` *oldact*`);`
  - Segmentation Fault 発生時に, *act* で指定したシグナルハンドラを実行

mprotect/mmap の保護属性設定の実装

- アドレス空間記述表に保護属性を記述し, 必要に応じてページテーブルを設定するだけ
  - 必要に応じて ≡ ページテーブルの設定で可能なアクセス ⊂ アドレス空間記述表で可能なアクセス
- OS にとっては, もともと他の目的に使っていた道具 (ページテーブル) を, ユーザに (も) 使わせているだけ
- もともと割り当てた領域は READ/WRITE 許可だったのを, READ/WRITE 禁止を可能にしているだけで, 安全性の問題もない
- むしろ「わざわざ READ/WRITE 禁止にして何の役に立つのか」が疑問

なぜ役に立つのだろうか?

- 思い出そう: OS 自身によるページテーブルのうまい利用法
  - 要求時ページング
  - mmap (e.g., 大きなファイルのランダムアクセス)
  - コピーオンライトによる物理メモリの節約
  - fork の高速化 (コピーオンライトの応用)
  - プロセス間共有メモリ
- 共通アイデア: ページフォルト, 保護違反の例外をキャッチして適切な処理を行う
- ⇒
  - ユーザに使わせればさらなる応用があるはず
  - そもそも実装も簡単 (ページテーブルを適切に設定するだけ) だし

多くの応用に共通の基本アイデア

- ある領域を READ (WRITE) 不可に設定
- 通常通りプログラムを実行
- 途中で保護違反が発生したら，その場所を READ (WRITE) 可に設定 + 実行を継続
- ポイント
  - 実行結果は通常と変わらない
  - 「実行中どこ (どのページ) にアクセスしたか」の情報が得られる

これまでに発明された応用

- {差分・並行}チェックポインティング
- ネットワークページング, 分散共有メモリ
- インクリメンタル Garbage Collection

興味のある人は,

- Andrew W. Appel and Kai Li. “Virtual Memory Primitives for User Programs”

ユーザレベル仮想記憶 API

チェックポインティング

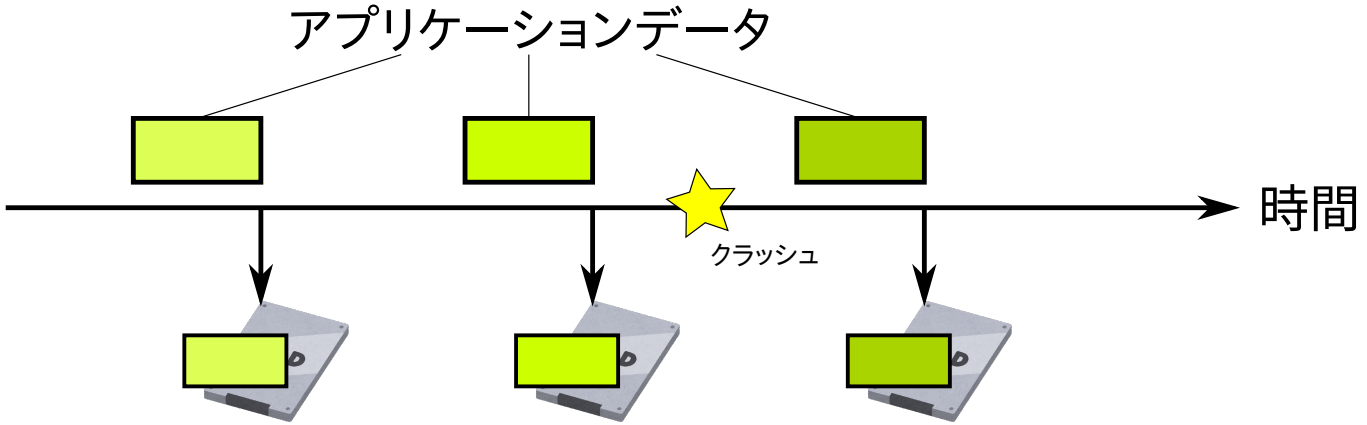
ネットワークページング

分散共有メモリ
- ハードウェア共有メモリ
- 分散共有メモリ

インクリメンタル Garbage Collection

チェックポインティングとは

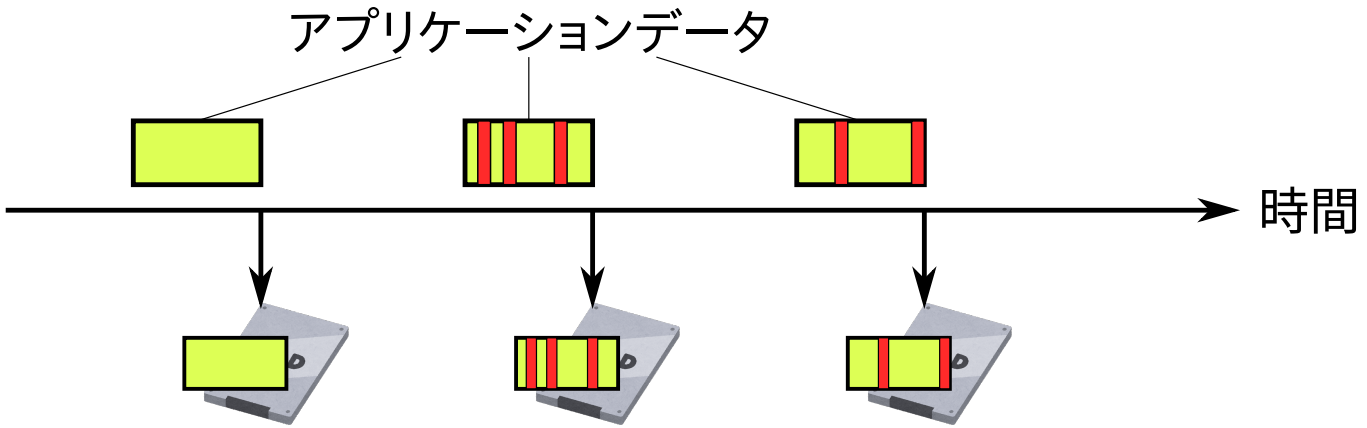
- 後から再開できるよう, 計算の途中状態を (通常 2 次記憶に) 保存すること
  - 保存した状態のことを「チェックポイント」と言う
- 主な目的: 耐故障性; 保存した状態から計算を再開
- 単純な方法:
  - 必要なアプリケーションデータをすべて, 保存



- 問題点:
  - データが大きい場合に時間がかかる → 差分チェックポインティング
  - その間アプリケーションが停止している → 並行チェックポインティング

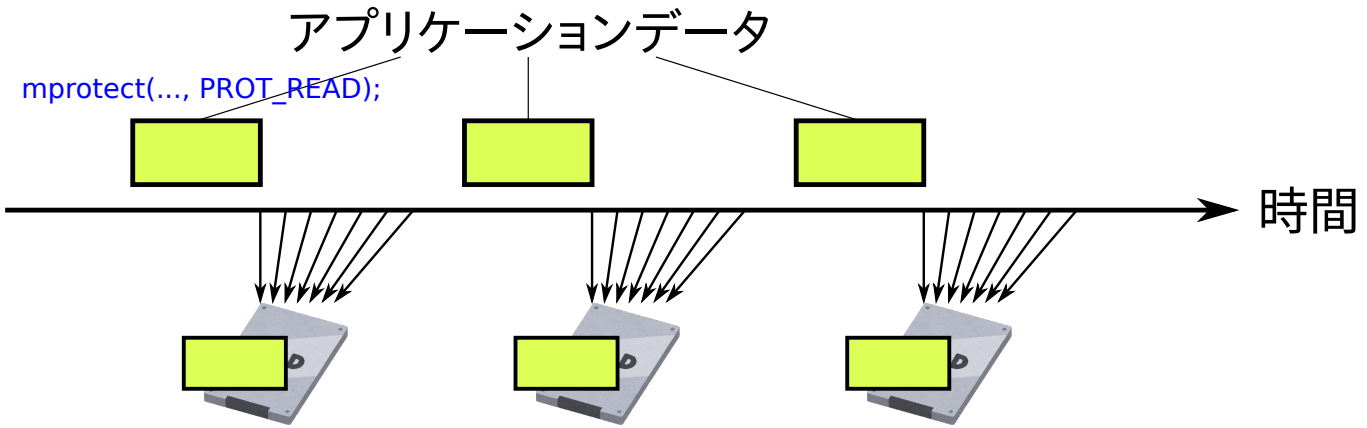
差分チェックポインティング

- チェックポイント保存時, 前回との「差分」のみ保存
  - 「差分」= 前回のチェックポイント以降に変更 (≈ 書き込み) があった部分
- 書き込まれた部分を知るために仮想記憶 API を使う
  1. チェックポイント取得後: データを書き込み禁止 (read-only) に設定
  2. 書き込み違反 (SEGV 発生) 時: 違反対象ページを記録; 書き込み可に設定
  3. 次回のチェックポイント取得時: 2. で記録されたページのみ保存



並行チェックポインティング

- 目的: ユーザプログラムの停止時間を短くする
- 基本: チェックポイントの保存と, ユーザプログラムの実行を並行して行う
- 手法:
  1. チェックポイントに到達した際, データを保存するかわりに, データを書き込み禁止に設定
  2. すぐにユーザプログラムを再開, それと並行してチェックポイントを保存
  3. あるデータに書き込みが起きたらそのページの (書き込み前の) データを保存



ユーザレベル仮想記憶 API

チェックポインティング

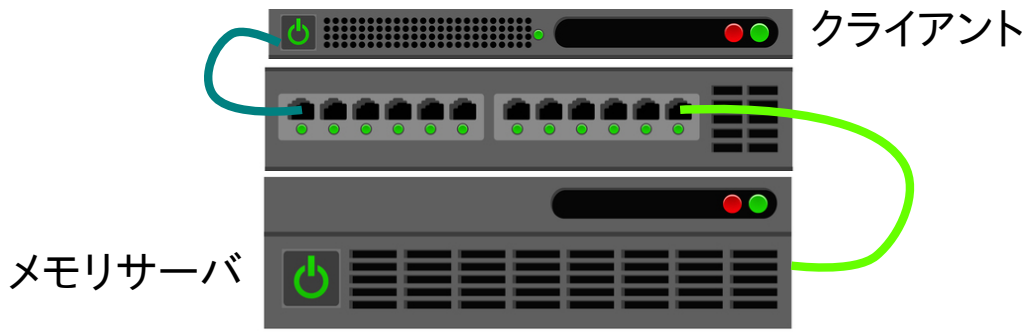
ネットワークページング

分散共有メモリ
- ハードウェア共有メモリ
- 分散共有メモリ

インクリメンタル Garbage Collection

ネットワークページング

- 主記憶の少ない計算機が, ページング領域として, 2 次記憶の代わりに, 主記憶を多く搭載したサーバ (メモリサーバ) の主記憶を使う (主記憶の拡張)
- 2 次記憶の速度 < ネットワークの速度の場合に有効



参考: 2次記憶とネットワークの速度

- 2 次記憶

SATA HDD	≈ 100MB/sec
SATA SSD	≈ 500MB/sec
NVMe SSD	≈ 1-5GB/sec
Optane Persistent Memory	≈ 2-7GB/sec

- ネットワーク

Gigabit Ethernet	ラップトップ	≈ 100MB/sec
10Gb Ethernet	普通のサーバ	≈ 1GB/sec
100Gb Ethernet	高性能サーバ	≈ 10GB/sec
400Gbps Infiniband NDR 4x	スパコン	≈ 50GB/sec

- 注: 速度は構成 (ディスク数やネットワークのリンク数) によって変わる

ネットワークページング アルゴリズム

- 前提:
  - クライアントが使える物理メモリ量 $= P$ ページ
  - クライアントが用いるメモリ領域 $= L$ $(> P)$ ページ
- クライアントは $\leq P$ ページだけを (mmap で) 割り当てた状態
- 残りの $(L - P)$ ページにアクセスすると, SEGV が発生
  $\rightarrow$
  1. すでに P ページ割当済みであれば適当なページをメモリサーバに送信; 解放 (munmap)
  2. アクセされたページを割当て (mmap)
- $\approx$ OS のメモリ管理をユーザプログラム内で行っている
  - OS 内のページフォルト $\approx$ ユーザプログラムの SEGV

ユーザレベル仮想記憶 API

チェックポインティング

ネットワークページング

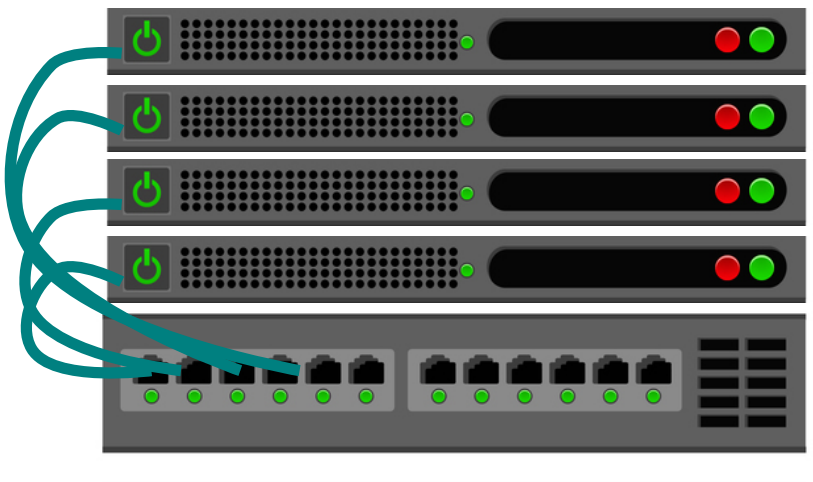
分散共有メモリ

- ハードウェア共有メモリ
- 分散共有メモリ

インクリメンタル Garbage Collection

分散共有メモリとは

- 複数の計算機で「あたかも」共有メモリを持っているかのように見せる技術
- ≈ OpenMP やスレッドプログラミングを用いた並列処理を, 複数の計算機でも可能にする
  - ⇔ ソケットやメッセージによる通信



そもそも共有メモリとは

- 「あるスレッドが書き込みをするだけで他のスレッドにそれが伝わる $(*)$」ことが本質

T_0	T_1
`a[100] = 200;`	
...	...
	`x = a[100];  /* 200 */`

- ハードウェア共有メモリ
  - 普通, 共有メモリと言えばこの意味
  - 1 つの計算機の中で複数の CPU (やその中のコア) が物理的に同じ主記憶を使っている
- 「分散」共有メモリ
  - 複数の計算機間で, $(*)$ を実現するソフトウェア
- まずはハードウェア共有メモリの実現方法を述べる
- 分散共有メモリでも考え方は共通

ユーザレベル仮想記憶 API

チェックポインティング

ネットワークページング

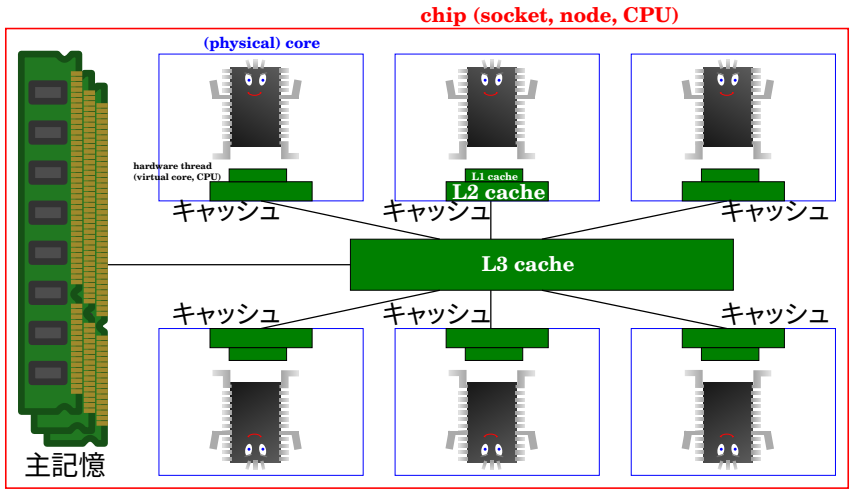
分散共有メモリ

- ハードウェア共有メモリ
- 分散共有メモリ

インクリメンタル Garbage Collection

ハードウェア共有メモリ

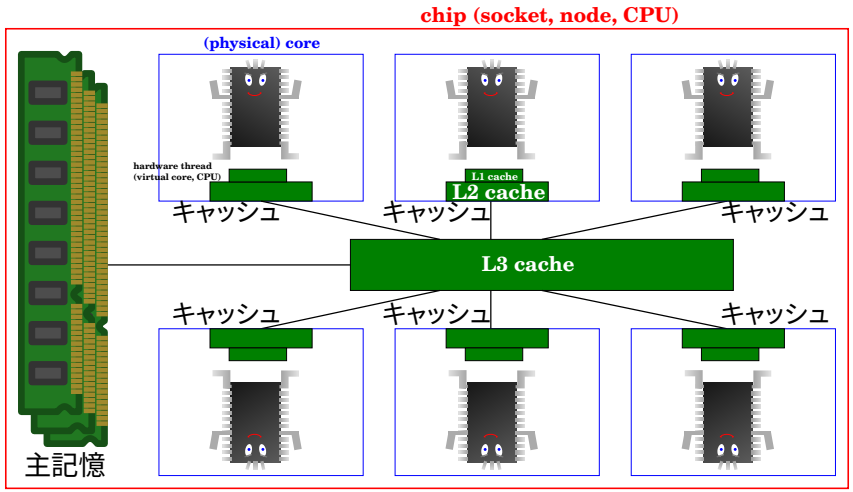
- 1 つの計算機の中では同一の物理メモリを使っていると言っても, CPU はメモリアクセス命令のたびに主記憶にアクセスしているわけではない (キャッシュ)



- したがって, 1 つの計算機内でも (∗) を実現するのは容易ではない
- . . . どころか, 極めて複雑な, キャッシュ一貫性 (cache coherency) プロトコルで実現されている

最も単純なキャッシュ一貫性プロトコル — Modify-Shared-Invalid (MSI)

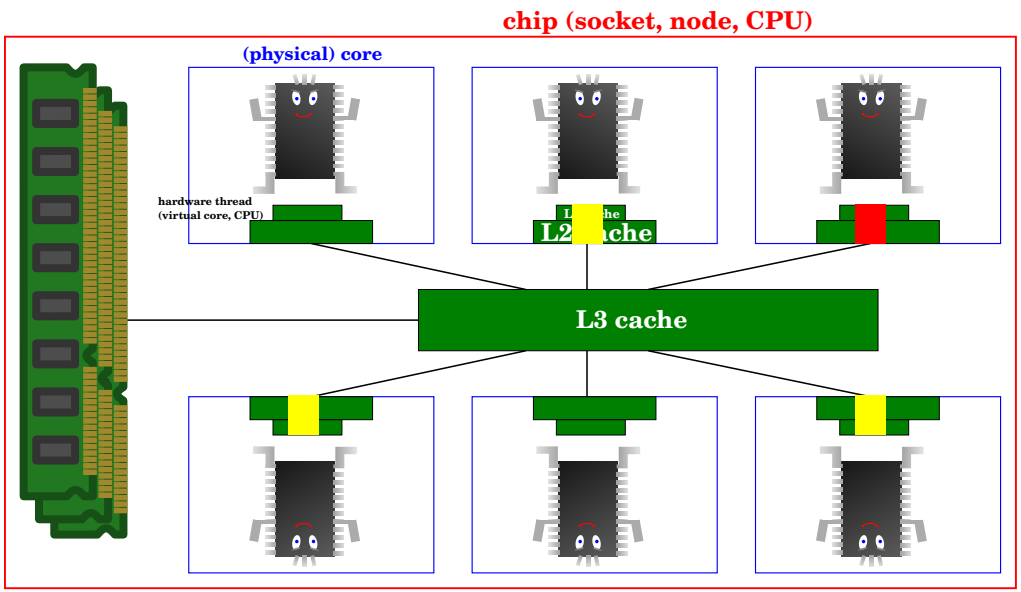
- 各コアは一定量 (≪ 主記憶量) のキャッシュを持つ



- キャッシュ内の各ブロック (管理の単位．普通は 64 バイト) は以下の 3 つのどれかの状態
  - Modified ( ) $\iff$ 他のコアに通知せずに write/read 両方が可能
  - Shared ( ) $\iff$ 他のコアに通知せずに read が可能
  - Invalid ( ) $\iff$ 他のコアに通知せずには何もできない

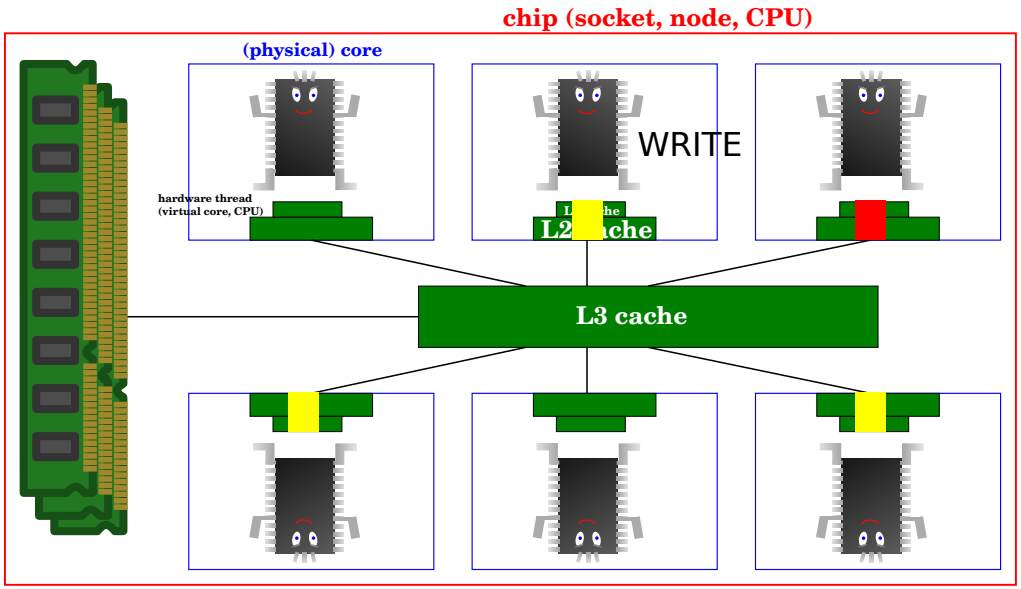
MSIの動作例

Modified ( ) が不在の状態



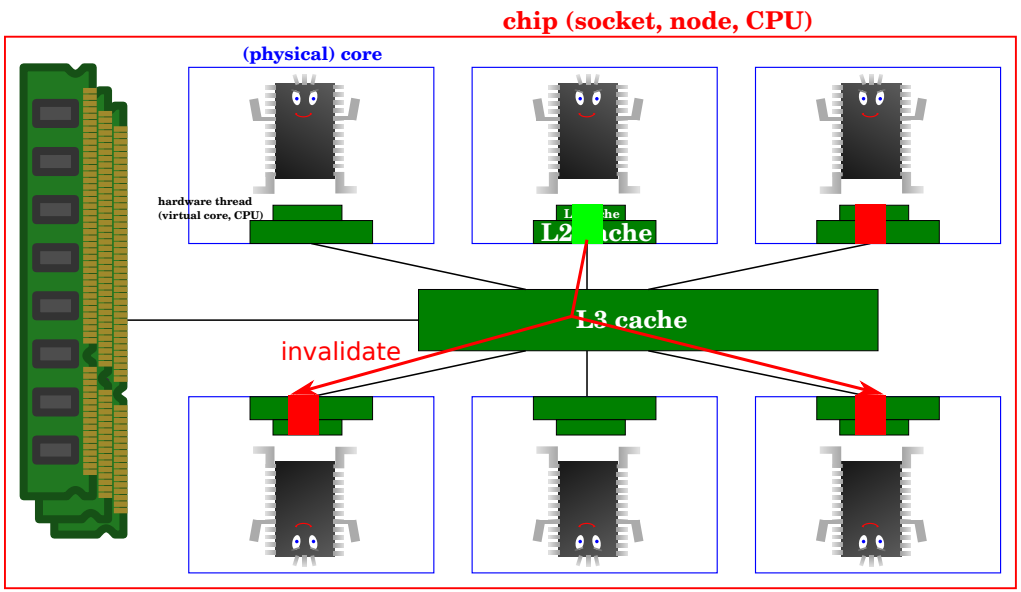
MSIの動作例

誰かが書き込む



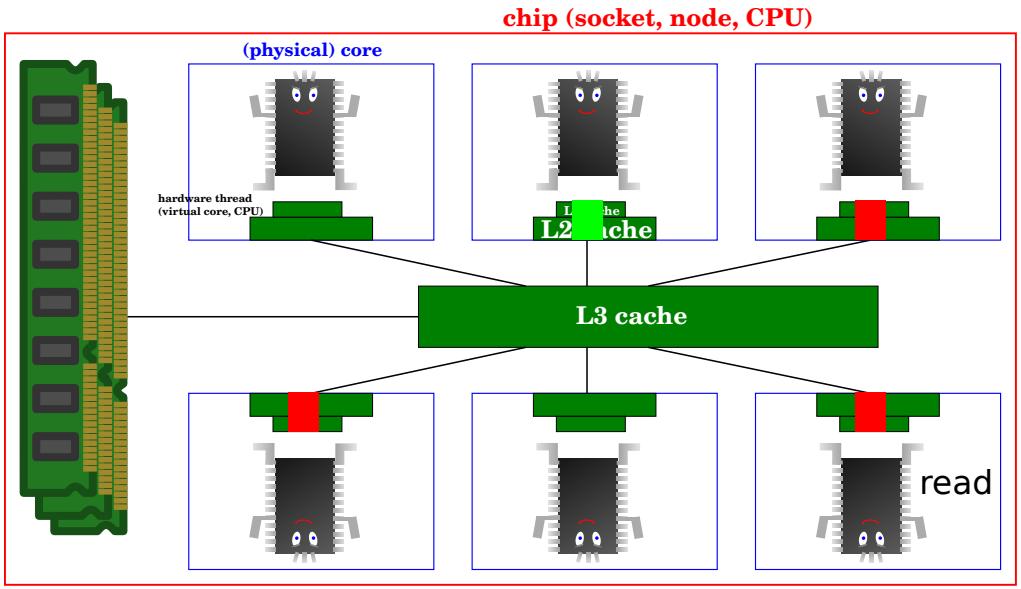
MSIの動作例

Shared (■) の全員を Invalid (▬) に



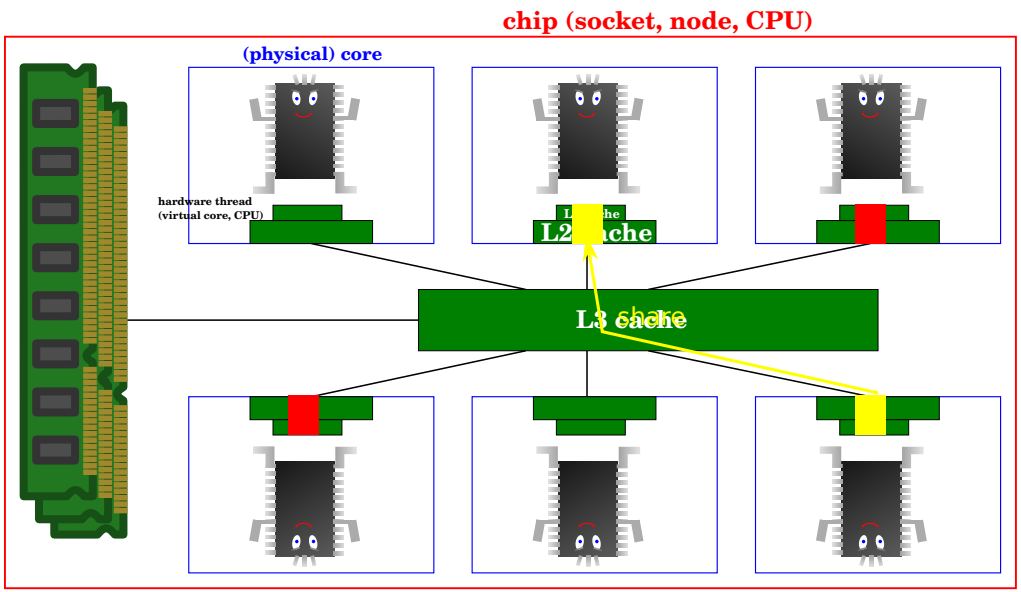
MSIの動作例

誰かが読み込む



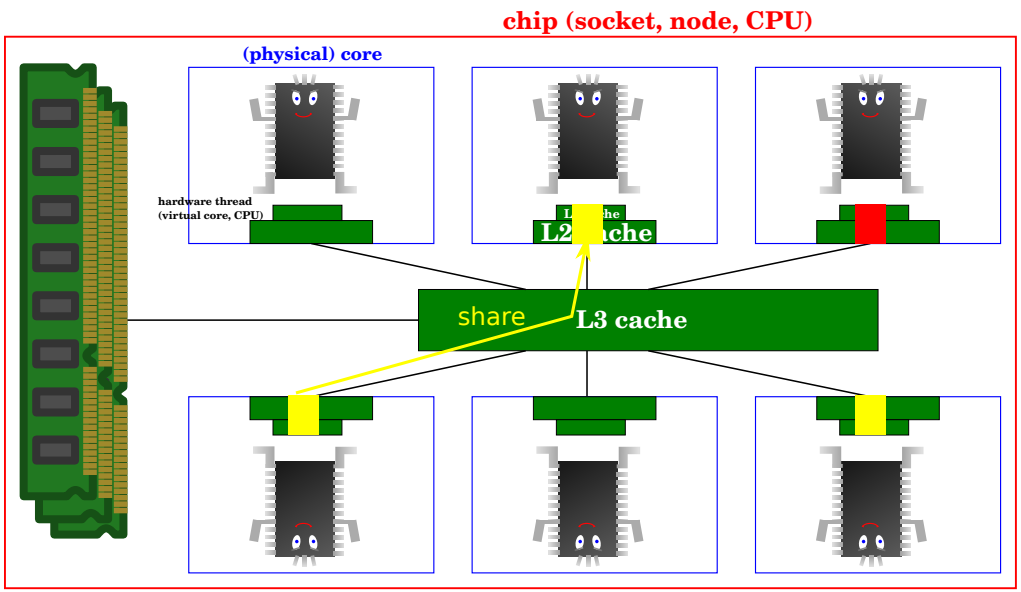
MSIの動作例

Modified (■) の人を Shared (■) に



MSIの動作例

Shared (■) は何人いても良い



MSIの不変条件と動作

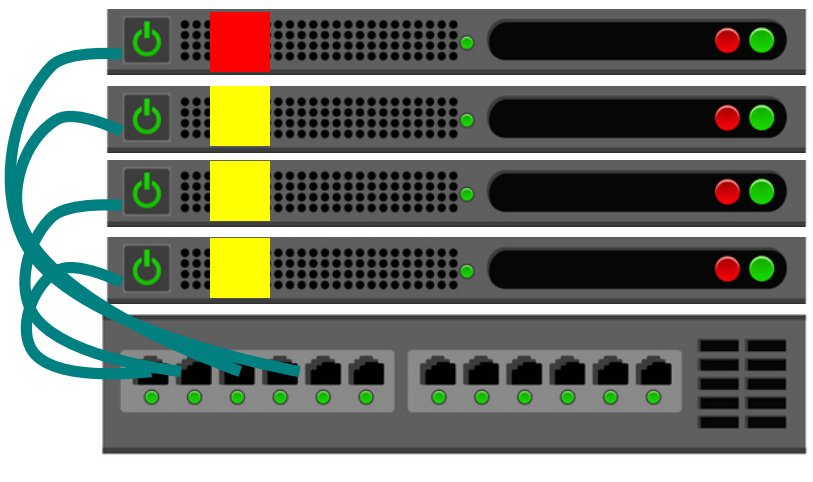
- 1ブロックに対するキャッシュの状態は以下のどちらかを保つ
  - 1個の Modified (■) + 任意個の Invalids
  - 任意個の Shared (■) / Invalid (■)
- 状態とアクションに応じてプロトコルを発動

	Modified	Shared	Invalid
read	hit	hit	read miss
write	hit	write miss	read miss; write miss

  - hit：何もしない (cache hit)
  - write miss：他のコアを Invalid (■) に
  - read miss：Modified なコアを Shared (■) に

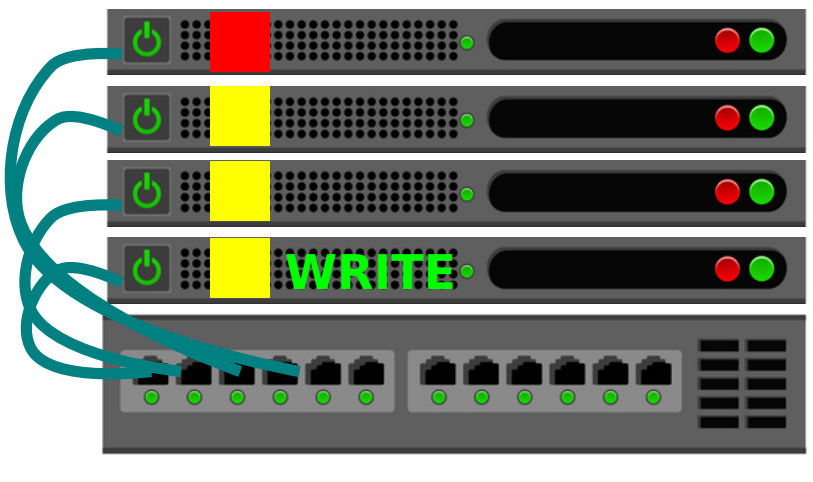
分散共有メモリの動作

- ≈ MSI をソフトウェアで実現
- 管理の単位はページ (当然) で, キャッシュの状態を, ページの保護属性 (mprotect) で実現
  - M : 読み書き許可 (`PROTO_READ|PROTO_WRITE`)
  - S : 読み出しのみ許可 (`PROTO_READ`)
  - I : 読み書き禁止 (`PROTO_NONE`)



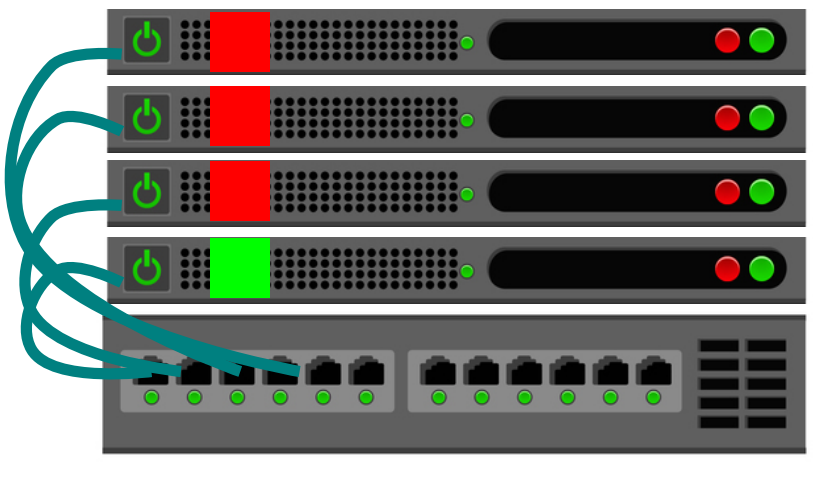
分散共有メモリの動作

- ≈ MSI をソフトウェアで実現
- 管理の単位はページ (当然) で, キャッシュの状態を, ページの保護属性 (mprotect) で実現
  - M : 読み書き許可 (PROTO_READ|PROTO_WRITE)
  - S : 読み出しのみ許可 (PROTO_READ)
  - I : 読み書き禁止 (PROTO_NONE)



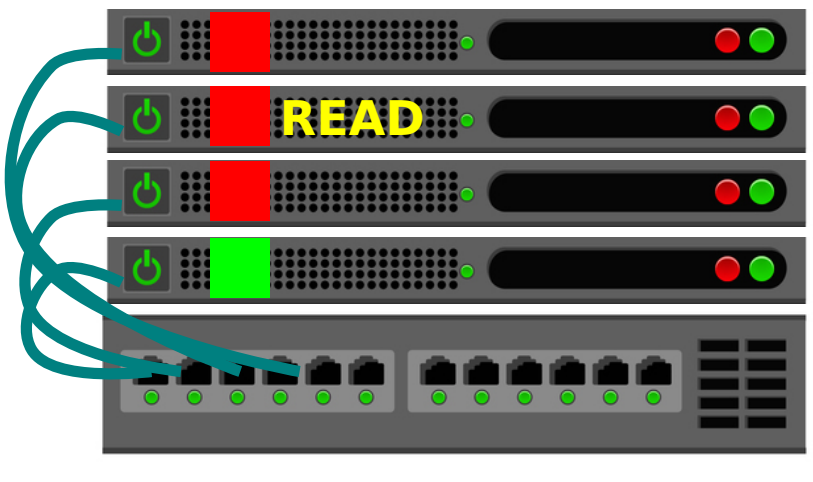
分散共有メモリの動作

- ≈ MSI をソフトウェアで実現
- 管理の単位はページ (当然) で, キャッシュの状態を, ページの保護属性 (mprotect) で実現
  - M : 読み書き許可 (`PROTO_READ|PROTO_WRITE`)
  - S : 読み出しのみ許可 (`PROTO_READ`)
  - I : 読み書き禁止 (`PROTO_NONE`)



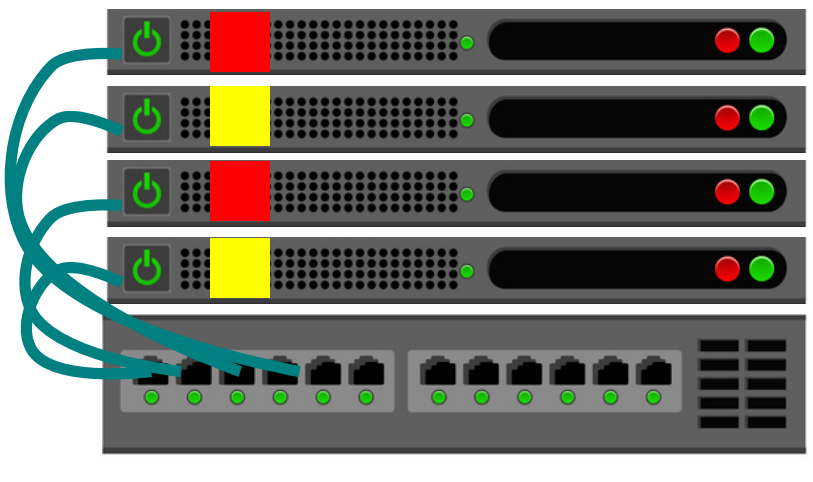
分散共有メモリの動作

- ≈ MSI をソフトウェアで実現
- 管理の単位はページ (当然) で, キャッシュの状態を, ページの保護属性 (mprotect) で実現
  - M : 読み書き許可 (PROTO_READ|PROTO_WRITE)
  - S : 読み出しのみ許可 (PROTO_READ)
  - I : 読み書き禁止 (PROTO_NONE)



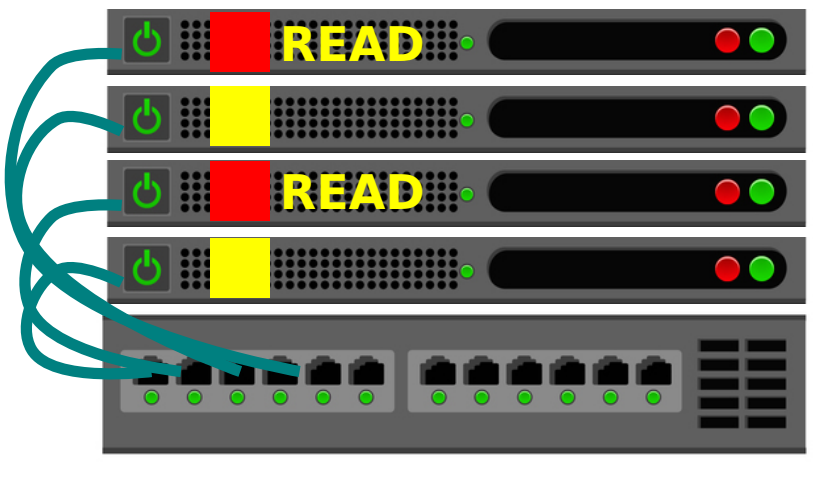
分散共有メモリの動作

- ≈ MSI をソフトウェアで実現
- 管理の単位はページ (当然) で, キャッシュの状態を, ページの保護属性 (mprotect) で実現
  - M : 読み書き許可 (`PROTO_READ|PROTO_WRITE`)
  - S : 読み出しのみ許可 (`PROTO_READ`)
  - I : 読み書き禁止 (`PROTO_NONE`)



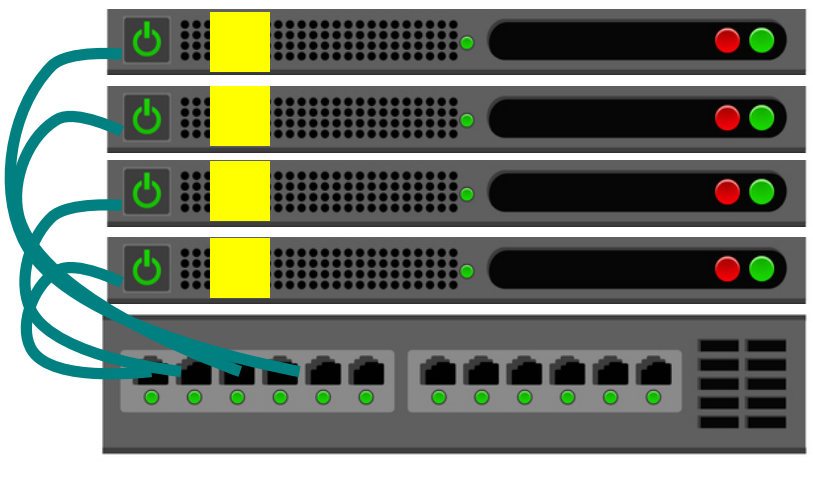
分散共有メモリの動作

- ≈ MSI をソフトウェアで実現
- 管理の単位はページ (当然) で, キャッシュの状態を, ページの保護属性 (mprotect) で実現
  - M : 読み書き許可 (PROTO_READ|PROTO_WRITE)
  - S : 読み出しのみ許可 (PROTO_READ)
  - I : 読み書き禁止 (PROTO_NONE)



分散共有メモリの動作

- ≈ MSI をソフトウェアで実現
- 管理の単位はページ (当然) で, キャッシュの状態を, ページの保護属性 (mprotect) で実現
  - M : 読み書き許可 (`PROTO_READ|PROTO_WRITE`)
  - S : 読み出しのみ許可 (`PROTO_READ`)
  - I : 読み書き禁止 (`PROTO_NONE`)

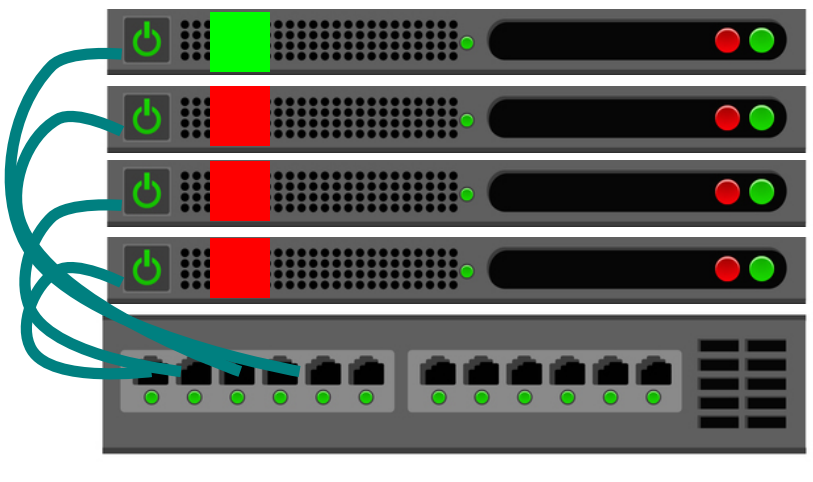


分散共有メモリの動作

- ≈ MSI をソフトウェアで実現
- 管理の単位はページ (当然) で, キャッシュの状態を, ページの保護属性 (mprotect) で実現
  - M : 読み書き許可 (PROTO_READ|PROTO_WRITE)
  - S : 読み出しのみ許可 (PROTO_READ)
  - I : 読み書き禁止 (PROTO_NONE)

分散共有メモリの動作

- ≈ MSI をソフトウェアで実現
- 管理の単位はページ (当然) で, キャッシュの状態を, ページの保護属性 (mprotect) で実現
  - M : 読み書き許可 (`PROTO_READ|PROTO_WRITE`)
  - S : 読み出しのみ許可 (`PROTO_READ`)
  - I : 読み書き禁止 (`PROTO_NONE`)



ユーザレベル仮想記憶 API

チェックポインティング

ネットワークページング

分散共有メモリ
- ハードウェア共有メモリ
- 分散共有メモリ

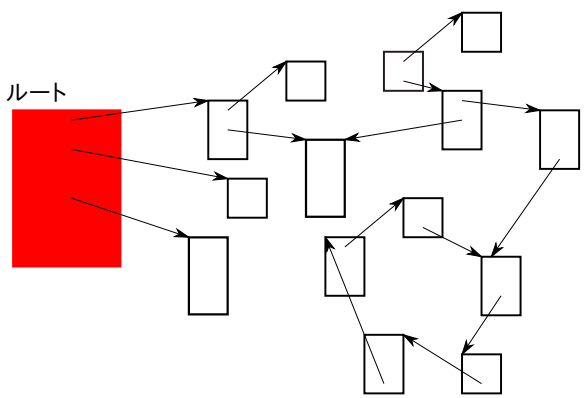
インクリメンタル Garbage Collection

Garbage Collection (GC), インクリメンタル GC

- Garbage Collection (GC)
  - 今後使われないメモリ領域を自動的に発見・回収する技術
  - C 言語で言うならば, malloc した領域を free しなくてもよくする技術
  - どうやって? ⇒ 次スライド
- インクリメンタル GC
  - GC を「少しずつ」行うことで, プログラムの「停止時間」を短くする技術

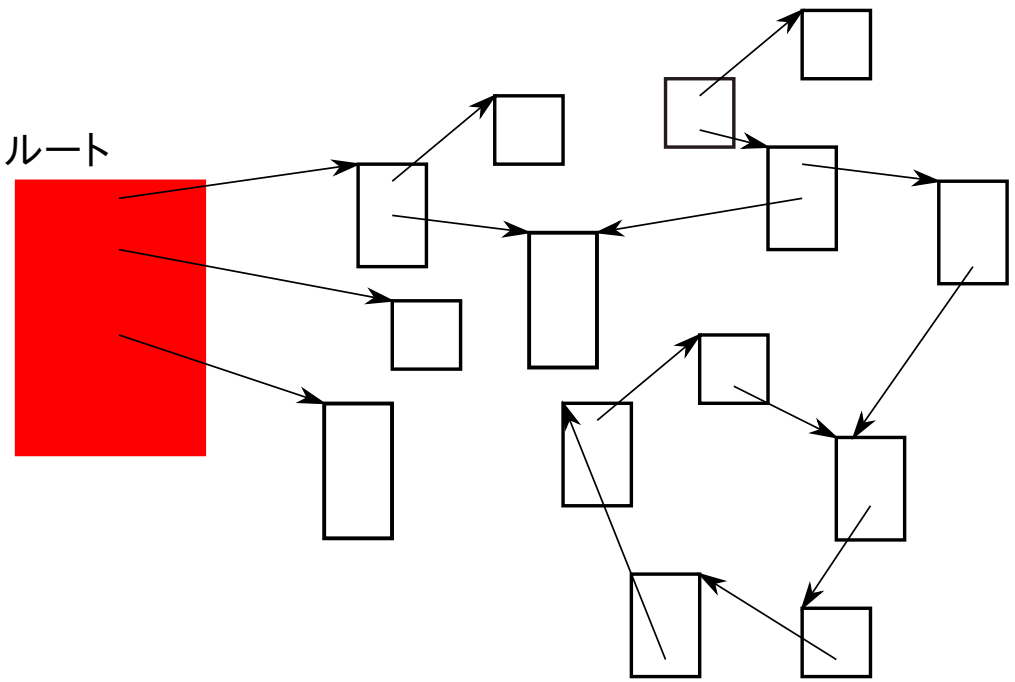
GC の基本原理

- プログラムが使うメモリ領域は, プログラムの大域変数/局所変数 (ルート), およびそこからポインタを任意回たどってたどり着く領域 (だけ) である
- GC の動作
  1. ルートからポインタをたどって, 使われうるメモリ領域を見つけていく
  2. これ以上たどれるポインタがなくなったところで，見つけられていない領域を回収
- 注: C 言語のようにポインタを使って任意のアドレスにアクセスできる言語では, 厳密に正しいとは言えないが, 仕様上, そのようなプログラムの動作は未定義で, 「そのようなプログラムはサポート外」としても有用性は損なわれないだろう



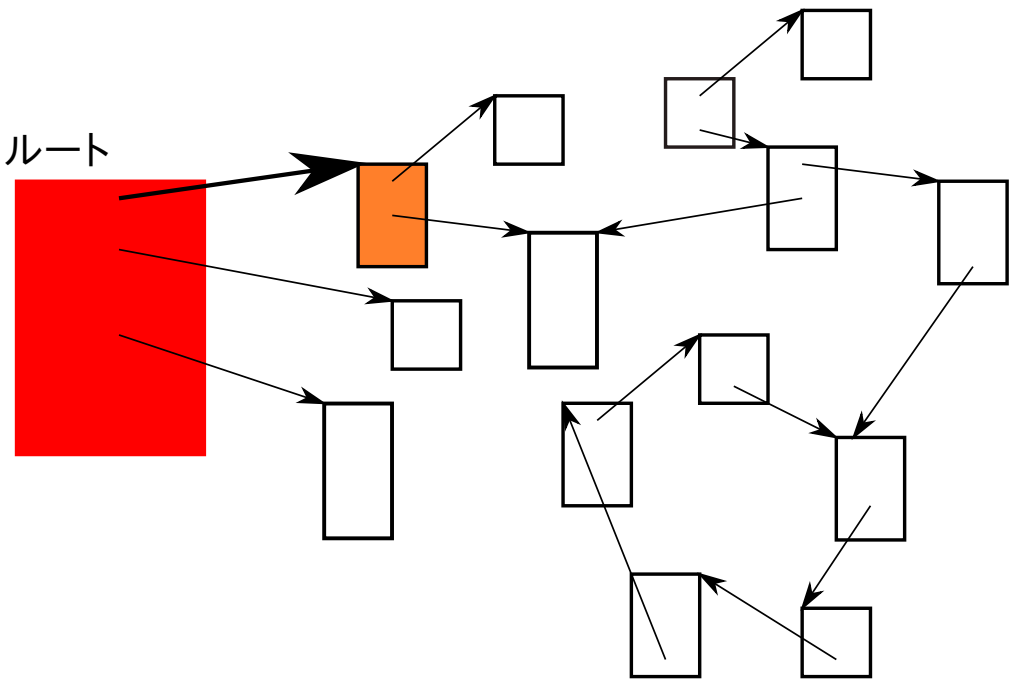
GCの動作 ≈ グラフのノード探索

- ノード：メモリを割り当てた単位 (C であれば malloc 1 回で割り当てた領域)
- 辺：ポインタ
- GC ≈ ルートから到達可能なノードを列挙し, それ以外を回収



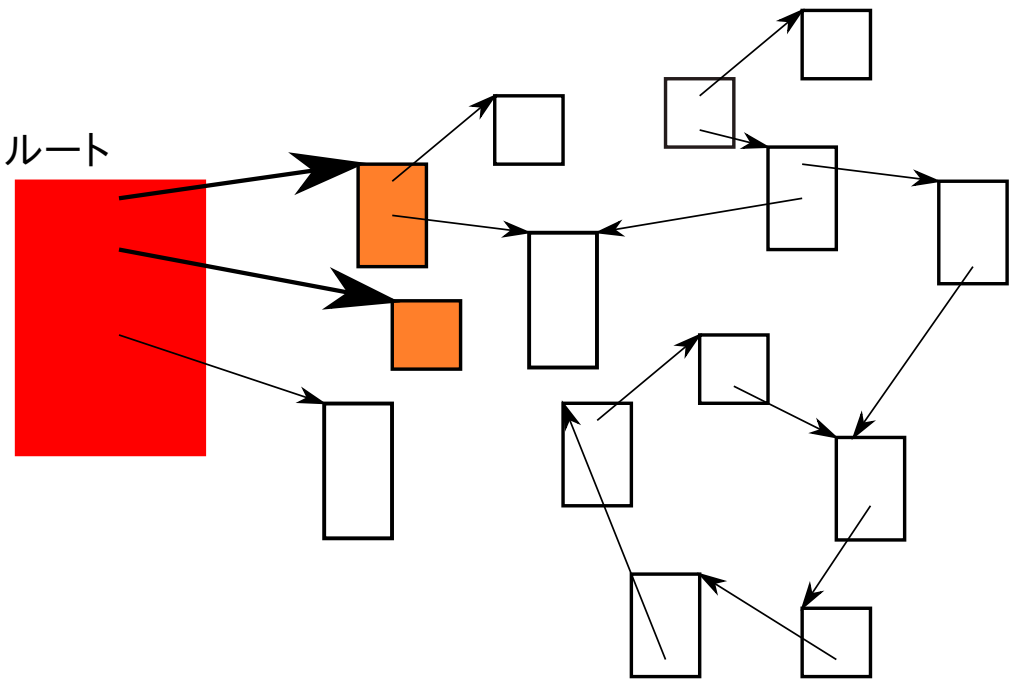
GCの動作 ≈ グラフのノード探索

- ノード：メモリを割り当てた単位 (C であれば malloc 1 回で割り当てた領域)
- 辺：ポインタ
- GC ≈ ルートから到達可能なノードを列挙し, それ以外を回収



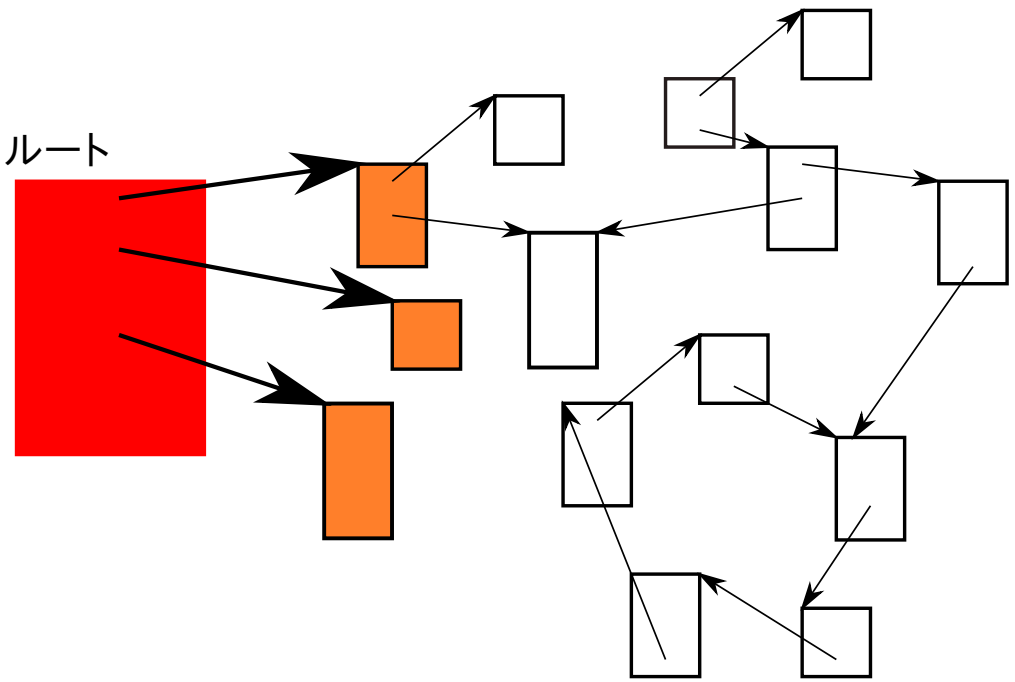
GCの動作 ≈ グラフのノード探索

- ノード：メモリを割り当てた単位 (C であれば malloc 1 回で割り当てた領域)
- 辺：ポインタ
- GC ≈ ルートから到達可能なノードを列挙し, それ以外を回収



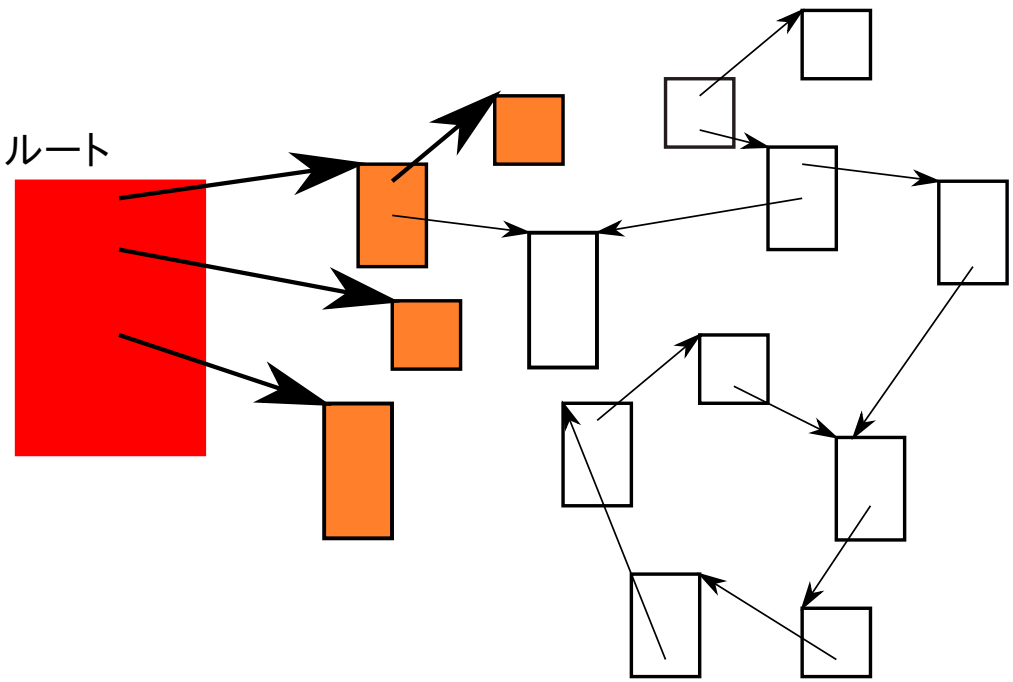
GCの動作 ≈ グラフのノード探索

- ノード：メモリを割り当てた単位 (C であれば malloc 1 回で割り当てた領域)
- 辺：ポインタ
- GC ≈ ルートから到達可能なノードを列挙し, それ以外を回収



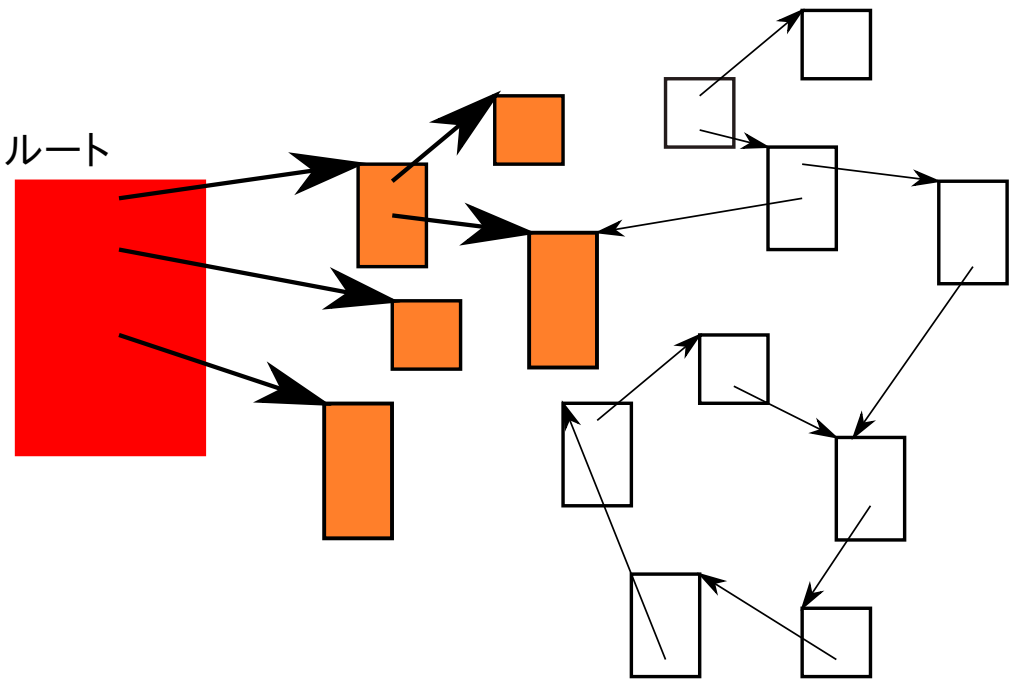
GCの動作 ≈ グラフのノード探索

- ノード：メモリを割り当てた単位 (C であれば malloc 1 回で割り当てた領域)
- 辺：ポインタ
- GC ≈ ルートから到達可能なノードを列挙し, それ以外を回収



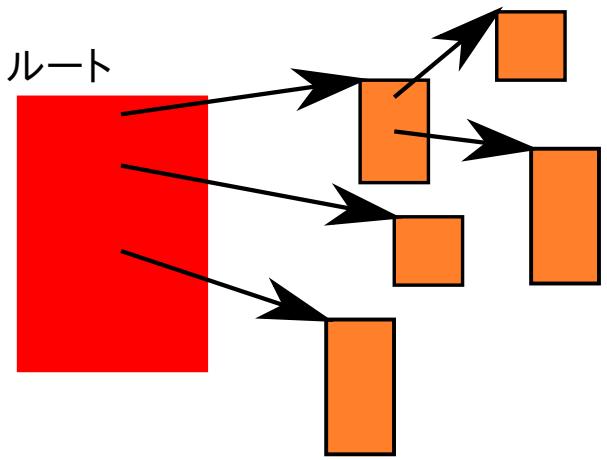
GCの動作 ≈ グラフのノード探索

- ノード：メモリを割り当てた単位 (C であれば malloc 1 回で割り当てた領域)
- 辺：ポインタ
- GC ≈ ルートから到達可能なノードを列挙し, それ以外を回収



GCの動作 ≈ グラフのノード探索

- ノード：メモリを割り当てた単位 (C であれば malloc 1 回で割り当てた領域)
- 辺：ポインタ
- GC ≈ ルートから到達可能なノードを列挙し, それ以外を回収



GC とプログラミング言語

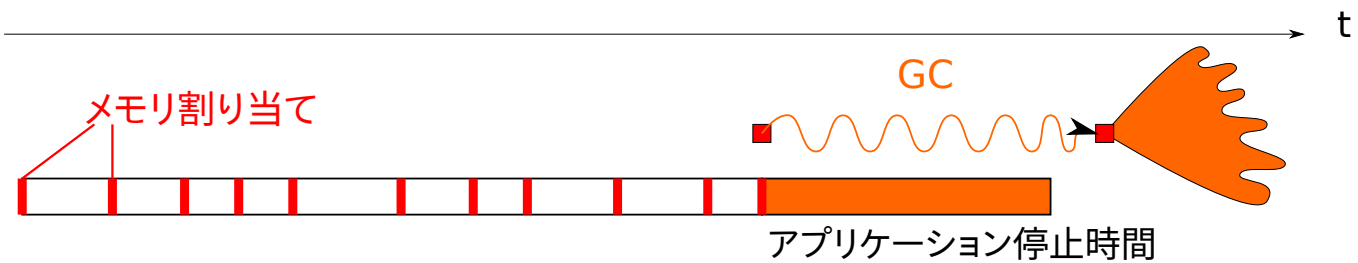
- GC はプログラミング言語に不可欠の機能
  - 詳しくは 4 年生の授業「プログラミング言語」にて
  - インクリメンタル GC のために仮想記憶 API が役に立つのでここで取り上げます
- C/C++/Fortran 以外のほとんどのプログラミング言語 (Python, Ruby, Julia, etc.) は GC を持つ
  - おかげでプログラマは自分でメモリを回収する (free/delete などを呼ぶ) 必要がない (安全)
- C/C++ からライブラリとして使える GC がある (Boehm GC)
- 本講義の残りでもこれを使って実験

Boehm GC

- `malloc` の代わりに `GC_MALLOC` という関数を呼ぶ (だけ)
  - このとき, 必要なタイミングで (≈ OS から取得した領域を使い果たしたら) GC が起動され, メモリを回収
  - OS からどれだけメモリを取得するかは調節可能 (詳細省略)
- `free` は呼ばなくて良い (使われない領域を勝手に回収・再利用してくれる)

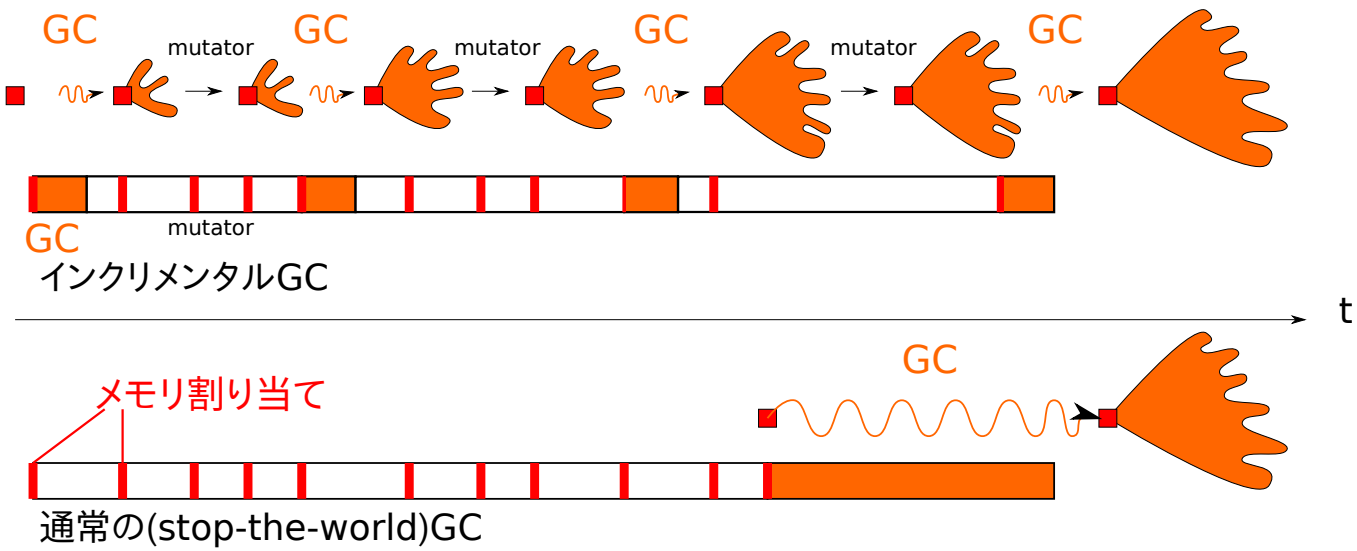
普通の (インクリメンタルでない) GC の問題点

- 到達可能なノードを全て発見し終えるまで, 一切メモリが回収できない
  - 例えば 10 GB のデータを使っているプログラムで GC がおきると, メモリを 10 GB 分触り終えるまで, (1 バイトも) メモリが回収できない
- その間, ユーザプログラムは動けない
  - GC がグラフをたどっている間にグラフが変化すると, 面倒なことになりそう
- ⇒ プログラムが時折経験する「停止時間」が (プログラムが使っているデータ量に応じて) 長くなる



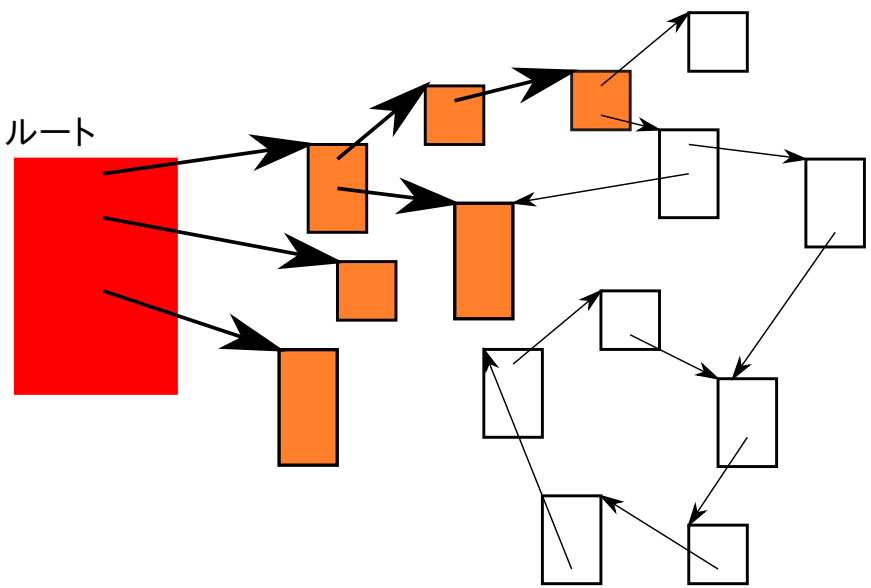
インクリメンタルGC

- GC の「停止時間 ($\neq$ 総時間)」を短くする GC
  - 実時間性や応答性を必要とするアプリケーション
- 停止時間 $\approx$ 到達可能なノード全てを走査する時間
- インクリメンタル GC はその走査を，少しずつ「細切れ」に行って，停止時間を短くする
  - e.g., 10GB「一気に」走査せずに，10MB ずつ 1000 回に分けて走査



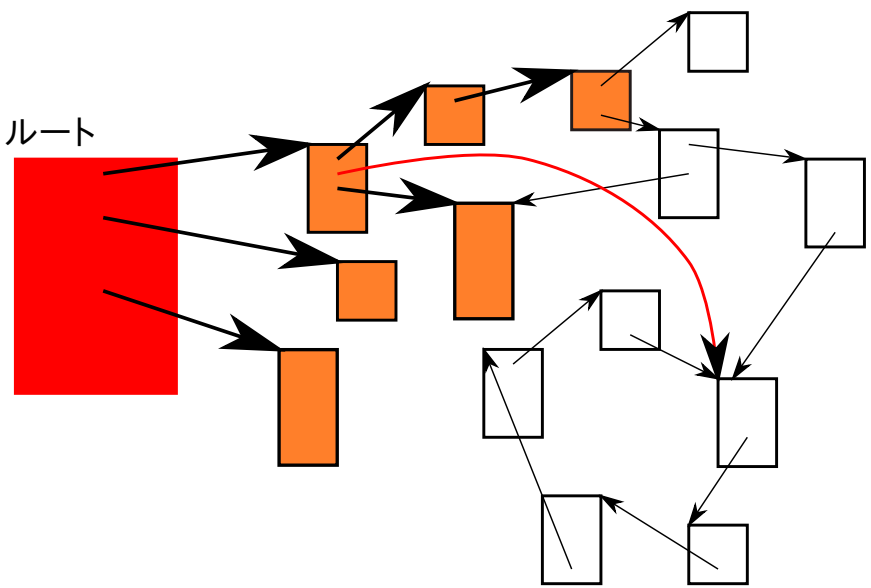
インクリメンタルGCの困難さと解決策

- グラフを走査している間にユーザプログラムが動作
- ⇒ 走査中にグラフが変化する (書き換わる)



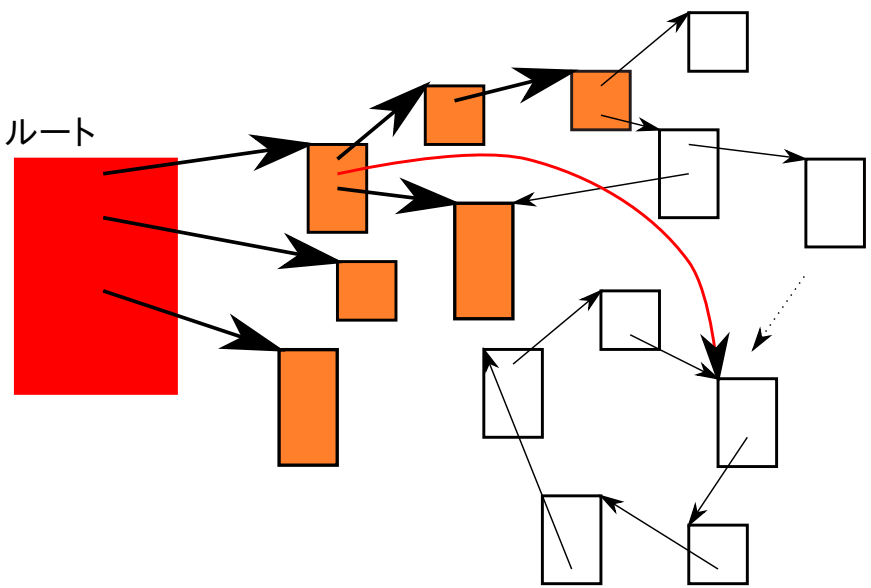
インクリメンタルGCの困難さと解決策

- グラフを走査している間にユーザプログラムが動作
- ⇒ 走査中にグラフが変化する (書き換わる)



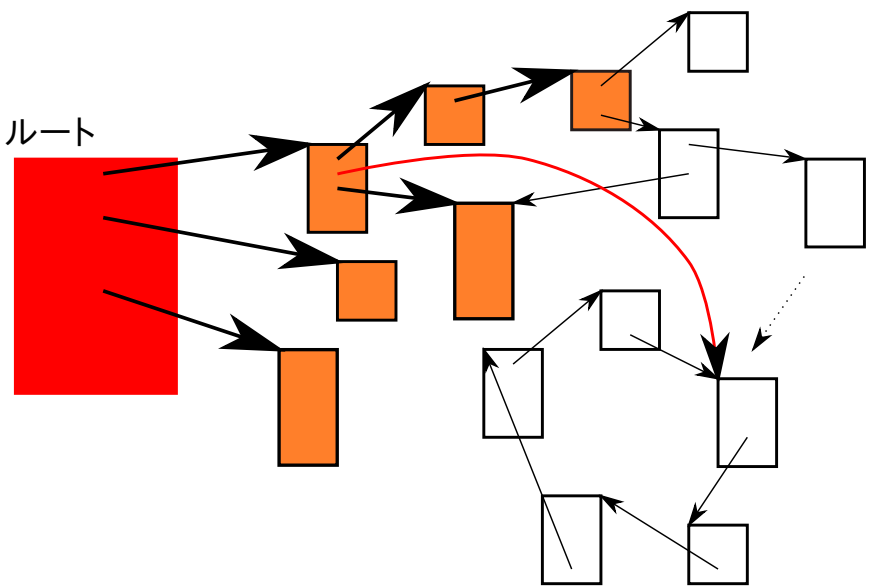
インクリメンタルGCの困難さと解決策

- グラフを走査している間にユーザプログラムが動作
- ⇒ 走査中にグラフが変化する (書き換わる)



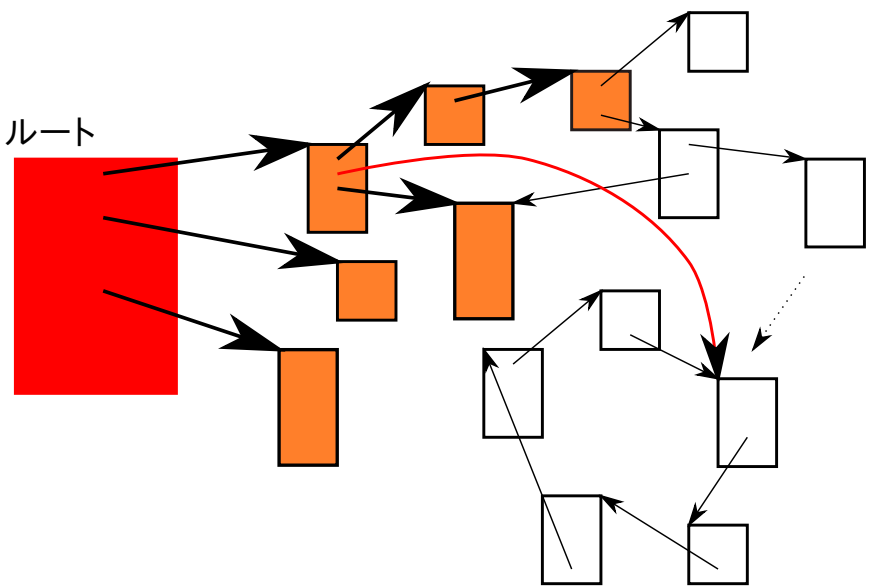
インクリメンタルGCの困難さと解決策

- グラフを走査している間にユーザプログラムが動作
- ⇒ 走査中にグラフが変化する (書き換わる)
- いくつか方法があるが, Boehm GC では, 書き換わったノード (ページ) から, 後でもう一度たどり直す
  - これで解決する事をきちんと説明すると長くなるので省略 (宣伝: 来学期「プログラミング言語」)



インクリメンタルGCの困難さと解決策

- グラフを走査している間にユーザプログラムが動作
- ⇒ 走査中にグラフが変化する (書き換わる)
- いくつか方法があるが, Boehm GC では, 書き換わったノード (ページ) から, 後でもう一度たどり直す
  - これで解決する事をきちんと説明すると長くなるので省略 (宣伝: 来学期「プログラミング言語」)
- 書き換わったページを検出するのに, 仮想記憶 API (mprotect) を使う



Boehm GCのインクリメンタルGC

1. GC_MALLOCが呼ばれた際, あるタイミングでGCを開始; 全領域を「書き込み不可 (PROT_READ)」に設定
2. その後GC_MALLOCが呼ばれた際, 到達可能な領域を「少しずつ」☒る (合間にユーザプログラムも動く)
3. ユーザが書き込み不可のページに書き込みを行ったら, そのページを記録しておく
4. グラフをたどり終えたら改めて, 書き込みまれたページから☒れるオブジェクトを☒る