パタヘネ輪読会

5.4 (途中) - 5.7

2021/06/25 (金)

5.4　キャッシュの性能の測定と改善

前回までのあらすじ

キャッシュの性能を改善する技法

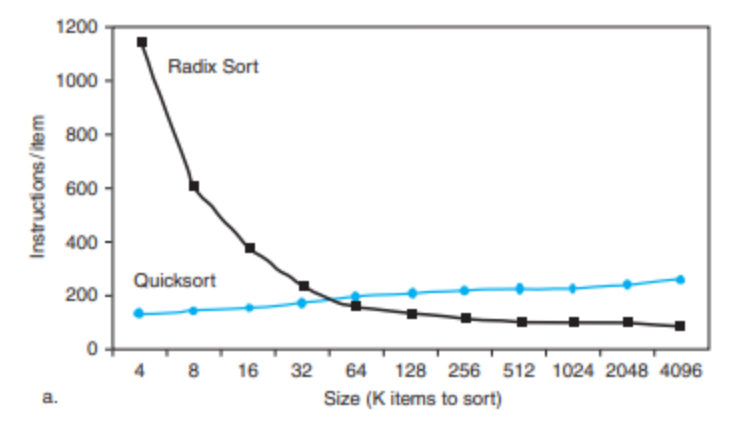
- アソシアティブ方式
- マルチレベル・キャッシュ

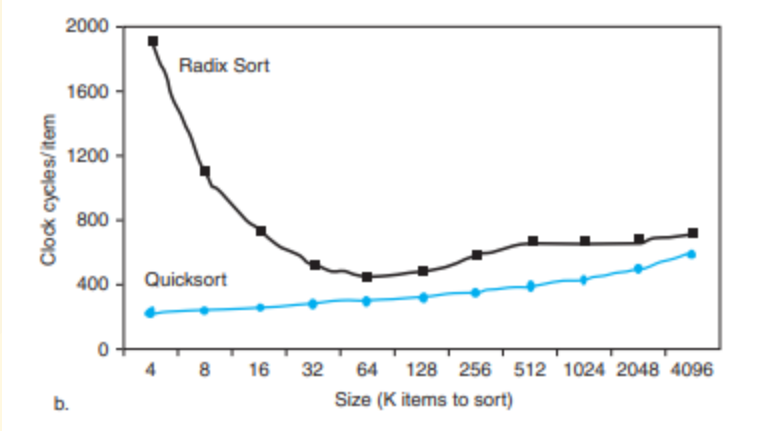
5.4　キャッシュの性能の測定と改善

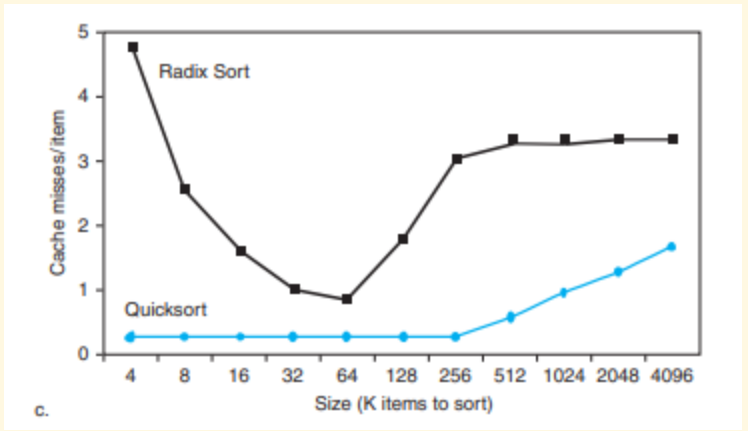
例: ソート

配列が大きいほど実行命令数は基数ソートの方が少なくなるが、キャッシュ・ミス率が跳ね上がり、実行時間としては不利になっていく。

図: 命令数 / クロック・サイクル数 / キャッシュ・ミス率

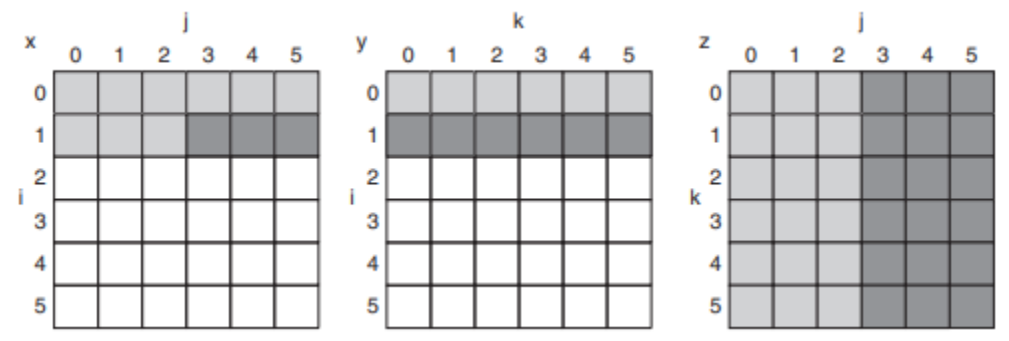






5.4　キャッシュの性能の測定と改善

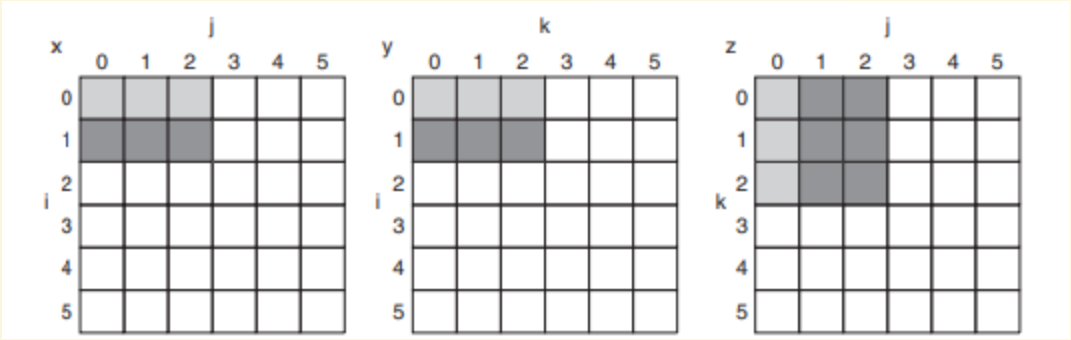
例: DGEMM (倍精度汎用行列乗算)



- C の N 要素、A の N 要素、B の $N \times N$ 要素に読み書きしている
- $N = 32$ 程度ならキャッシュに乗る
- キャッシュミスすると最悪 $2N^3 + N^2$ 程度のアクセスが発生

5.4　キャッシュの性能の測定と改善

例: DGEMM (倍精度汎用行列乗算)



- 大きさ `BLOCKSIZE` の部分行列に分割して確実にキャッシュに載せる
- アクセスされるメモリ語の合計は $2N^3/\texttt{BLOCKSIZE} + N^2$

5.4　キャッシュの性能の測定と改善

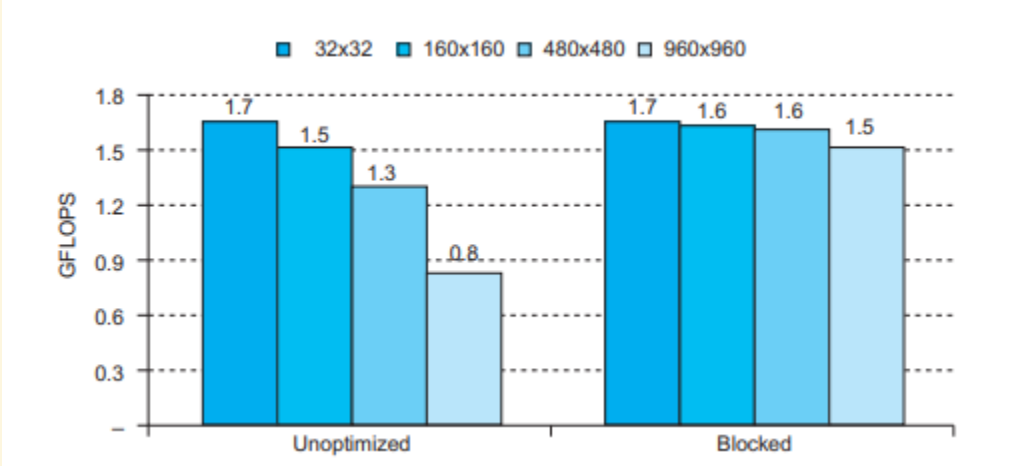
```
#define BLOCKSIZE 32

void do_block(int n, int si, int sj, int sk, double *A, double *B, double *C) {
    for (int i = si; i < si + BLOCKSIZE; ++i)
        for (int j = sj; j < sj + BLOCKSIZE; ++j)
        {
            double cij = C[i + j * n];  // cij = C[i][j]
            for (int k = sk; k < sk + BLOCKSIZE; ++k)
                cij += A[i + k * n] * B[k + j * n];  // cij += A[i][k] * B[k][j]
            c[i + j * n] = cij;  // C[i][j] = cij
        }
}

void dgemm(int n, double *A, double *B, double *C) {
    for (int sj = 0; sj < n; sj += BLOCKSIZE)
        for (int si = 0; si < n; si += BLOCKSIZE)
            for (int sk = 0; sk < n; sk += BLOCKSIZE)
                do_block(n, si, sj, sk, A, B, C);
}
```

5.4　キャッシュの性能の測定と改善

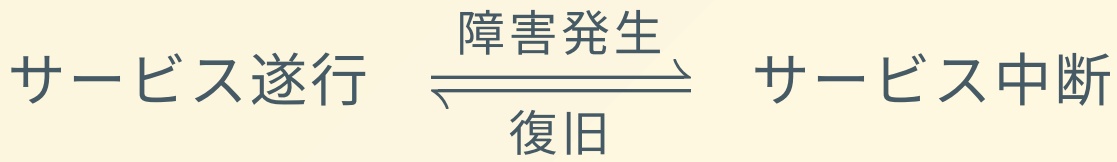
性能評価



(最近話題になってたやつ) [2のべき乗サイズの配列は危ないという話 via 行列積]

5.5　信頼性の高い記憶階層

1. **サービス遂行**: サービスが仕様どおりに提供されている
2. **サービス中断**: 提供されているサービスが仕様から外れている



「信頼性」と「可用性」を定量的に診断

5.5　信頼性の高い記憶階層

信頼性 (reliability) に関する用語

- **平均故障寿命** (**MTTF**: Mean Time To Failure)
- **年間故障率** (**AFR**: Average Failure Rate)

例: MTTF が 1,000,000 時間 (≒ 114 年) のディスクは故障しなさそうに思えるが、
　このディスクを 2 台搭載するサーバーを 50,000 台用意したときの AFR は？

解答: 1 年 = 8760 時間。
　MTTF が 1,000,000 時間のとき AFR は 8760 / 1,000,000 = 0.876 %。
　10,000 台のディスクがあれば 1 年間に 876 台、1 日に 2 台以上の
　ディスクが故障する。

5.5　信頼性の高い記憶階層

可用性 (availability) に関する用語

- **平均修復時間** (**MTTR**: Mean Time To Repair)
- **平均故障間隔** (**MTBF**: Mean Time Between Failure) = MTTF + MTTR

$$\text{可用性} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

- MTTF の改善方法
  - 故障回避: 故障の発生原因を元から断つ
  - 故障許容: 冗長性を持たせて、故障が発生してもよくする
  - 故障予測: 故障の発生箇所と発生時期を予想して、故障する前に交換

5.5　信頼性の高い記憶階層

Hamming の 1 ビット誤り検出

- パリティ・コード: 語中の 1 の数が奇数なら 1、偶数なら 0 を付加
- パリティ・コードによって 1 ビットの誤り検出が可能
  - 2 ビットは無理。3 ビットはできるけど可能性は低い

例題: 値が 31_{10} のバイトの右端にパリティ・ビットをつけよ。
　最上位ビットを反転したとき、誤りが検出されるか。

解答: 31_{10} は 00011111_2。
　パリティが奇数なので 1 をつけて、$000111111\mathbf{1}_2$ となる。
　最上位ビットが反転されると $\mathbf{1}00111111_2$。
　これはパリティが奇数なので誤っている。

5.5　信頼性の高い記憶階層

Hamming の 1 ビット誤り訂正

Bit position		1	2	3	4	5	6	7	8	9	10	11	12
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Parity bit coverage	p1	X		X		X		X		X		X	
	p2		X	X			X	X			X	X	
	p4				X	X	X	X					X
	p8								X	X	X	X	X

- 左から順に 1 から番号を振り、2 のべき乗の位置をパリティ・ビット、その他をデータ・ビットとする
- パリティ・ビットは以下のチェック対象を偶数パリティにするように決まる
  - ビット 1 はビット (1, 3, 5, 7, ...) をチェック
  - ビット 2 はビット (2, 3, 6, 7, 10, 11, ...) をチェック
  - ビット 4 はビット (4-7, 12-15, 20-23, ...) をチェック

5.5　信頼性の高い記憶階層

Hamming の 1 ビット誤り訂正

例題: データ 10011010_2 に関する Hamming ECC コードは？

解答: パリティ・ビットを入れると $\_\_1\_001\_1010$。
位置 1 のパリティ・ビットは $\_\_\mathbf{1}\_\mathbf{0}01\_\mathbf{1}0\mathbf{1}0$ をチェック → 0 となる。
位置 2 のパリティ・ビットは $0\_\mathbf{1}\_0\mathbf{01}\_\mathbf{1}0\mathbf{1}0$ をチェック → 1 となる。
位置 4 のパリティ・ビットは $011\_\mathbf{001}\_10\mathbf{1}0$ をチェック → 1 となる。
位置 8 のパリティ・ビットは $0111001\_\mathbf{1010}$ をチェック → 0 となる。
最終的に 011100101010 となる。

5.5　信頼性の高い記憶階層

Hamming の 1 ビット誤り訂正

例題: ビット 10 を反転させて、誤りを訂正せよ。

解答: ビット 10 を反転させると $01110010\mathbf{1}10$。
パリティ・ビット 1 は $\mathbf{0}1\mathbf{1}10\mathbf{0}10\mathbf{1}1\mathbf{1}0$ であり OK。
パリティ・ビット 2 は $0\mathbf{1}1\mathbf{0}0\mathbf{1}01\mathbf{1}0$ でありこの中に誤りがある。
パリティ・ビット 4 は $011\mathbf{00}1011\mathbf{0}$ であり OK。
パリティ・ビット 8 は $01110\mathbf{01110}$ でありこの中に誤りがある。
パリティ・ビットの 2 と 8 が正しくないので、2 + 8 = 10 よりビット 10 が誤り。
ビット 10 を訂正して $01110010\mathbf{0}10$ を得る。

5.5　信頼性の高い記憶階層

Hamming の 1 ビット誤り訂正 / 2 ビット誤り検出 (SEC/DED)

- 1 ビット誤り訂正コードにもう 1 ビット、全体に対するパリティ・ビットをつけると 2 ビットの誤り検出が可能
- 今日のサーバーのメモリでは SEC/DED が当たり前になっている
- 8 バイトのデータ・ブロックに 1 バイトを追加すれば実現可能

5.6　仮想マシン

仮想マシン (VM: Virtual Machine)

- 分離とセキュリティの重要性
- 標準的な OS ではセキュリティと信頼性に問題がある
- クラウドでは 1 台のコンピュータを大勢のユーザーが共有
- プロセッサの速度の向上によりオーバーヘッドは許容できる

- 広義には、Java VM のようにエミュレーションするものを含む
- ISA レベルでハードウエアと合致するものを **システム仮想マシン** と呼ぶ

5.6　仮想マシン

仮想マシン (**VM**: Virtual Machine)

- **仮想マシン・モニター** (**VMM**) あるいは **ハイパーバイザ** と呼ばれる ソフトウエアによって支援される
- **ホスト** のリソースを、複数の **ゲスト VM** の間で管理

- 例: AWS EC2 で使われる理由
  - 同じサーバーで各ユーザーを相互に保護
  - インスタンスに対するソフトウエアの配布が楽
  - ユーザーによる終了が楽
  - ハードウエアの詳細を隠して新しいサーバーを導入できる
  - プロセッサ、ネットワーク、ディスク・スペースの使用を制御できる

5.6　仮想マシン

VMM の要件

- 本来のハードウエア上での実行とまったく同様に動作すること
- ゲストは実際のシステム・リソースの割当てを直接変更できてはならない

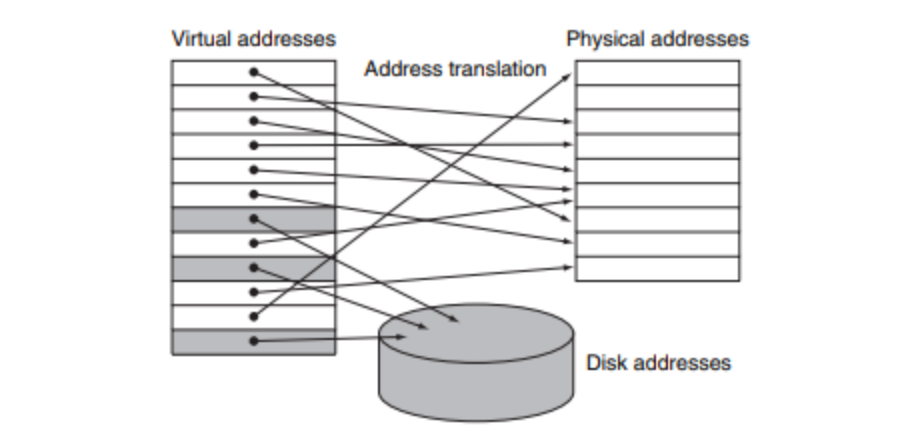
よって仮想化するには

- システムとユーザーという 2 つのプロセッサ・モードがあること
- システム・モードでのみ利用可能な特権命令があり、
  これがユーザー・モードで実行されたらトラップする

アーキテクチャ側で仮想化が考慮されていないと大変
x86 や ARMv7 や MIPS では考慮されていない

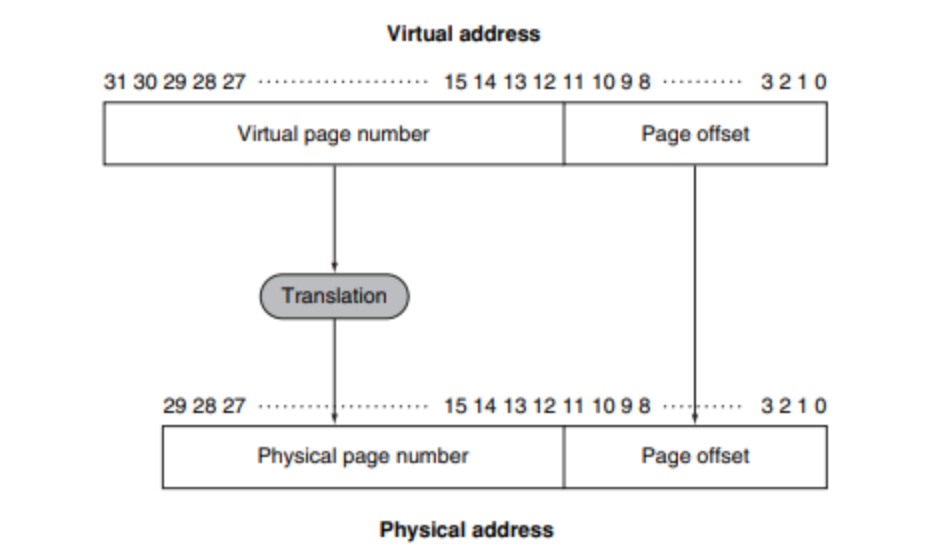
5.7　仮想記憶

- 主記憶を 2 次記憶にとっての「キャッシュ」のように使う
- 各プログラムの **仮想アドレス** を **物理アドレス** に変換 (**アドレス変換**)
- 仮想マシン間のアドレス空間の保護が可能
- 主記憶の容量を超えるサイズのプログラムを動かせる
  - かつてはプログラマが工夫しなければならなかった
  - 仮想記憶を使うと再配置 (relocation) によっていい感じにできる



5.7　仮想記憶

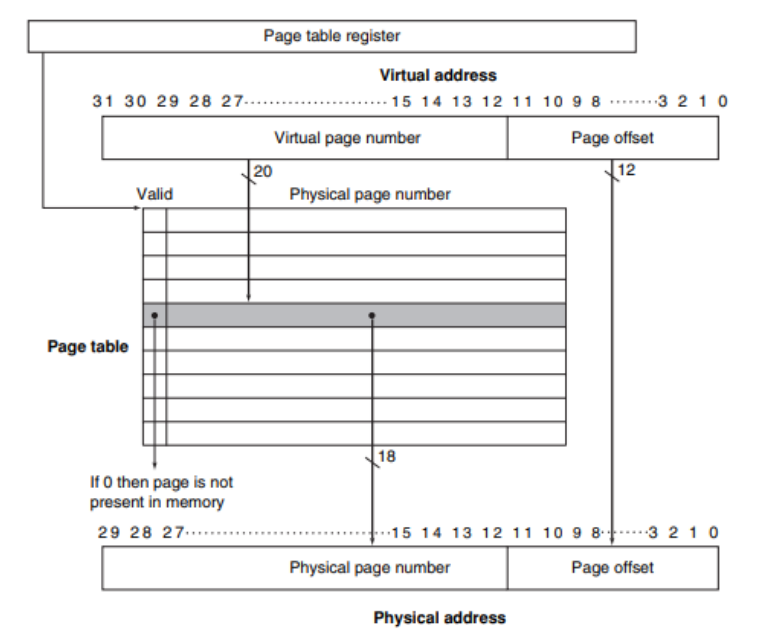
- 仮想記憶のブロックを **ページ** と呼ぶ
  - (仮想ページ番号, ページ内オフセット) を (物理ページ番号, ページ内オフセット) に変換
  - ページサイズは典型的には 4 KiB から 16 KiB ほど、組み込みでは 1 KiB ほど



ブロックが固定長であるページ方式に対し、可変長であるセグメント方式も存在する。
結局はページ方式の方が扱いやすく、性能が良い。

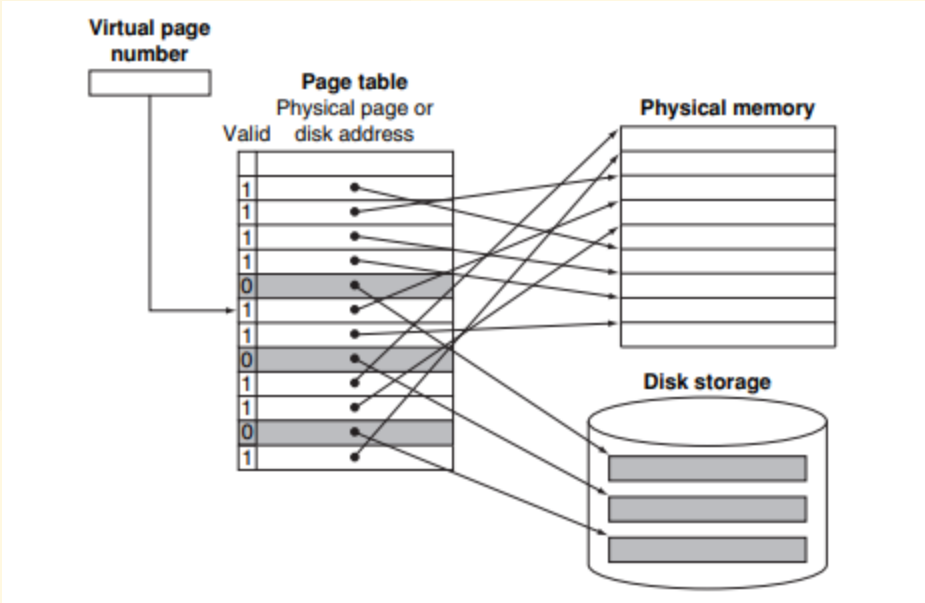
5.7　仮想記憶

- 仮想記憶のミスを **ページ・フォルト** と呼ぶ
  - ミス・ペナルティがかなり大きいので、極力ページ・フォルトを減らしたい
  - うまくページ配置・選択できるならフル・アソシアティブ方式が望ましい
- エントリを検索する機構: **ページ・テーブル**



5.7　仮想記憶

- エントリの有効ビットが 0 のときページ・フォルト
- ページ・テーブルはプロセスに基づく
- プロセスのすべてのページ用のスペースを予約しておく (**スワップ空間**)
- 物理メモリやディスクとの対応は OS が管理
  - LRU (Least Recently Used) 法



5.7　仮想記憶

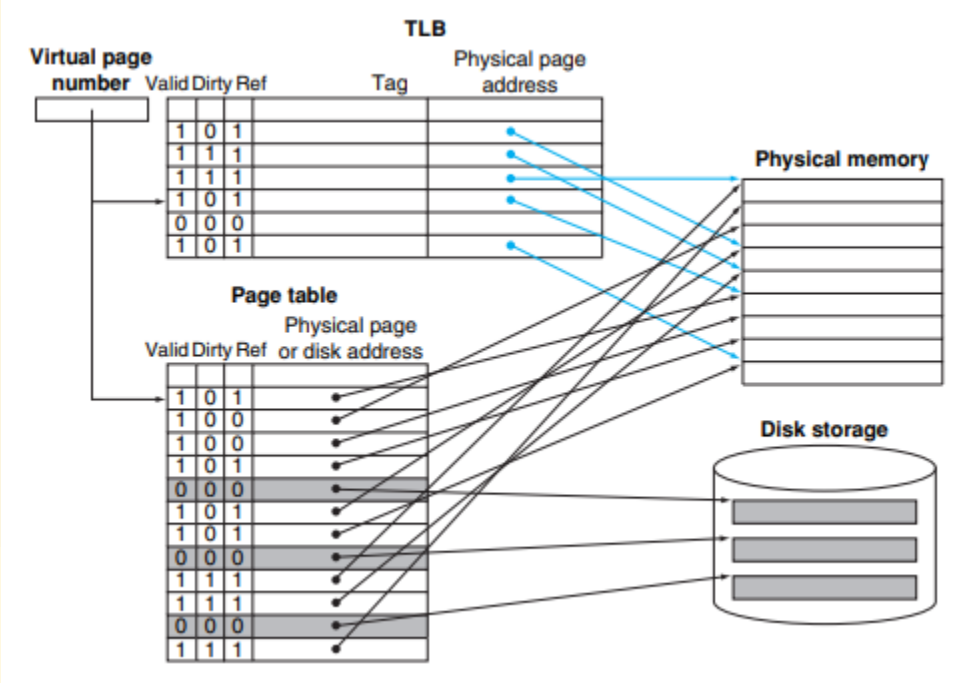
- 仮想アドレス 32 ビット、ページ・サイズ 4 KiB、エントリ 4 バイトとすると
  - エントリ数 = $2^{32} / 2^{12} = 2^{20}$
  - ページ・テーブルのサイズ = $2^{20} \times 4 = 4$ MiB
- プロセスごとに 4 MiB 必要でしんどい
  - プロセスが数百走っていたら？
- 対処法
  - 仮想アドレスに対してページ・テーブルのサイズを動的に合わせる
  - スタック領域とヒープ領域があるので、上の方法を 2 方向に対応する
  - ハッシング (hashing) でサイズを減らす
  - ページ・テーブルをマルチレベル化
  - ページ・テーブルをページング

5.7　仮想記憶

- 書き込みは？
- ディスクへの書き込みはめっちゃ遅いのでライト・スルーではなく
  ライト・バック
- ページを置き換えるときに、そのページを書き戻す必要があるかどうかを
  判定する **ダーティ・ビット** を持つ

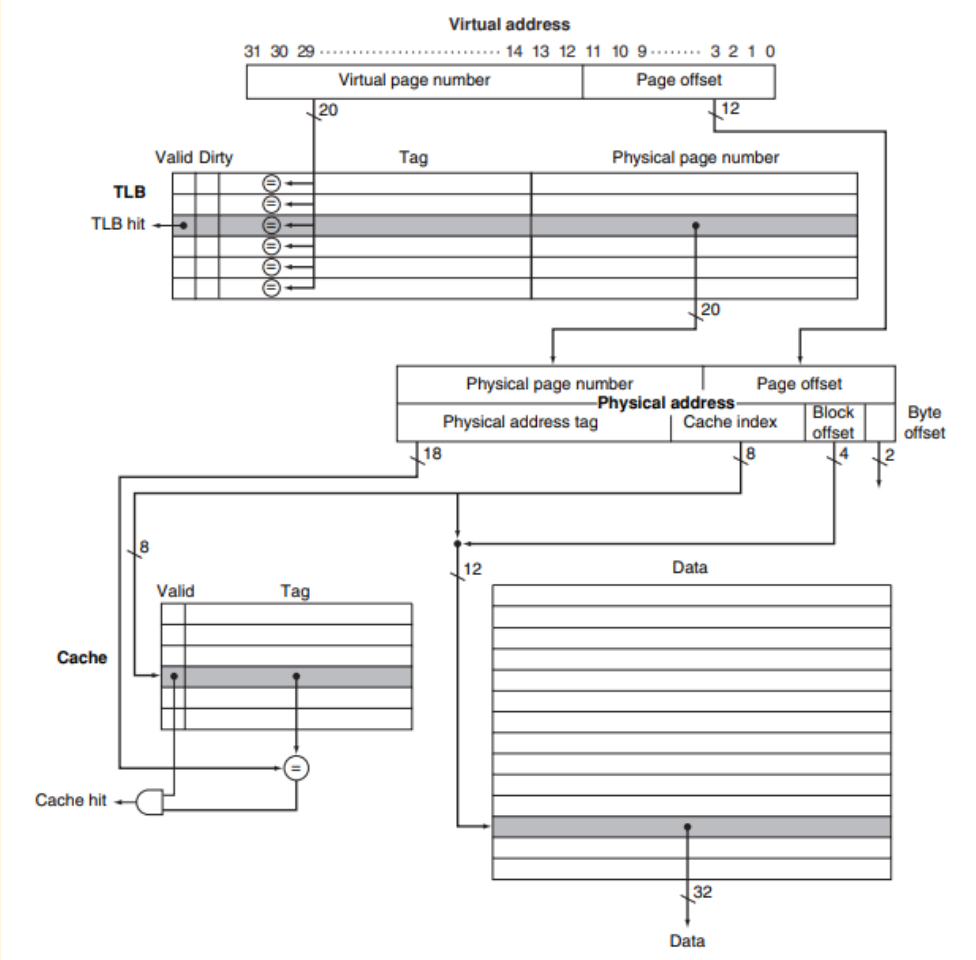
5.7　仮想記憶

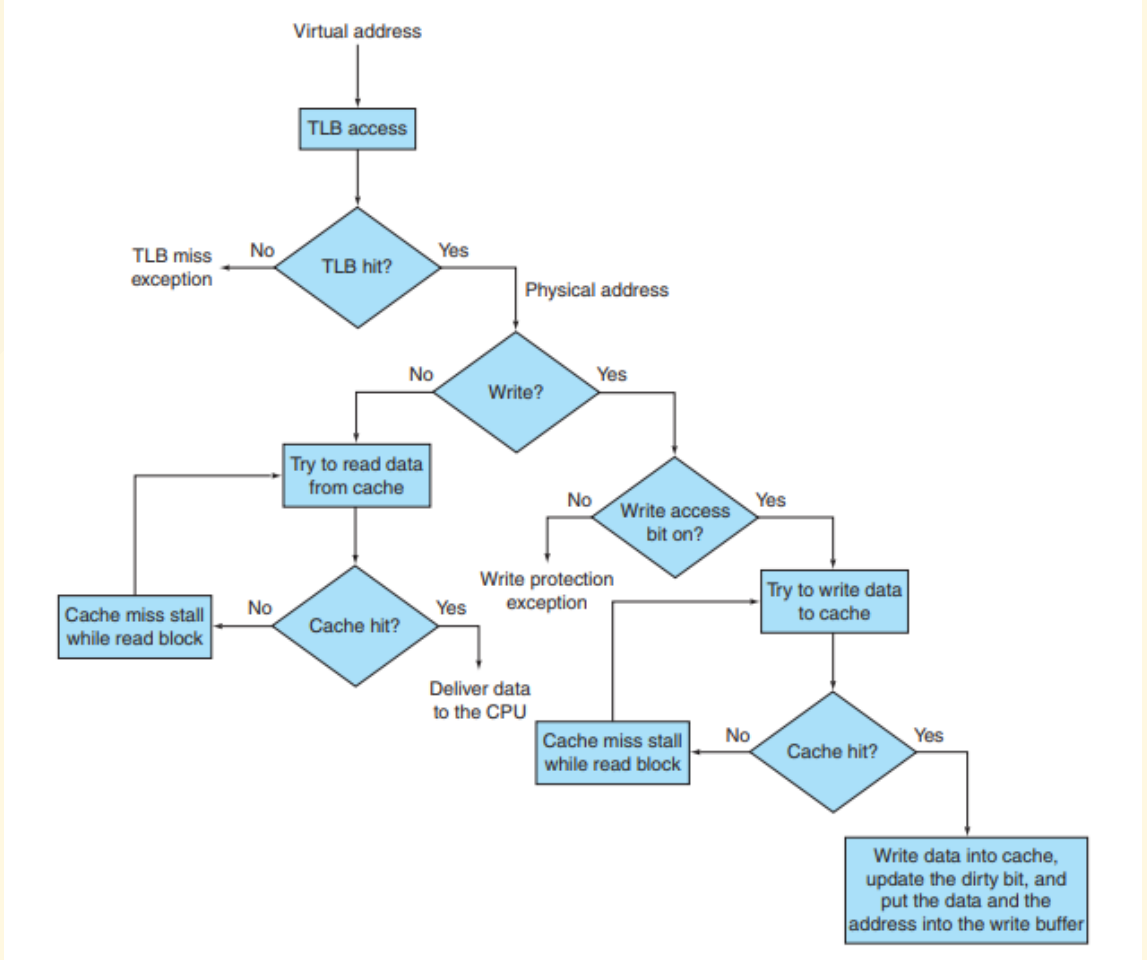
- ページの参照にも局所性がある
- **アドレス変換バッファ (TLB**: Translation-Lookaside Buffer)
  - 最近のアドレス変換の内容を記録
  - TLB ミスとページ・フォルトは見分ける必要がある



5.7　仮想記憶

実例: Intrinsity FastMATH





5.7　仮想記憶

仮想記憶と TLB とキャッシュの関係って何

TLB	ページ・テーブル	キャッシュ	状況
ヒット	ヒット	ミス	TLB でヒット
ミス	ヒット	ヒット	TLB ミス → ページ・テーブルでヒット
ミス	ヒット	ミス	TLB ミス → ページ・テーブルでヒット
ミス	ミス	ミス	TLB ミス → ページ・フォルト
ヒット	ミス	ミス	発生しない
ヒット	ミス	ヒット	発生しない
ミス	ミス	ヒット	発生しない

5.7　仮想記憶

仮想記憶の保護機構

- ユーザー・プロセス用と OS のプロセス用の 2 つのモードを指定できる
  - 後者はカーネルプロセス、スーパーバイザプロセス、エグゼクティブプロセスなど呼ばれる
- ユーザー・プロセスは CPU 情報の一部を読み出せるが書き込めない
- ユーザー・モードとスーパーバイザ・モードの相互のやりとり
  - システム・コール (MIPS では `syscall` 命令)

- プロセス間の読み書きの保護
- プロセス間の情報の共有
- プロセスの切り替え (**コンテキスト・スイッチ**)

5.7　仮想記憶

- TLB ミスしたとき
- ページがメモリにあるとき
  - 主記憶からページ・テーブルのエントリを取り込む
  - TLB ミスを起こした命令を再実行
- ページがメモリにないとき
  - 例外機構を利用して稼働中のプロセスに割り込み
  - 該当プロセスの状態を退避し OS に処理を移す
  - ページ・テーブルを参照してディスク上の位置を見つけ出す
  - 置き換え対象の物理ページを選ぶ。ダーティ・ビットが立っていたらそのページをディスクに書き戻さなければならない
  - ページをディスクから読み出して物理ページに納める
  - 状態を復元して、例外処理から復帰
  - とても面倒くさいし、数百万ステップかかるので、この間に別のプロセスを処理しておく

5.7　仮想記憶

MIPS の場合

- TLB ミスが発生すると、ハードウエアは参照されていたページ番号を `BadVAddr` という特殊レジスタに退避し、例外を発生
- 例外処理ルーチンは OS を呼び出して TLB ミスを処理
  - TLB ミスのハンドラは $8000\,0000_{16}$ にある
- ページ・テーブルを検索
  - 専用のレジスタで高速化

```
TLBmiss:
  mfc0 $k1, Context   # ページ・テーブル・エントリのアドレスを一時レジスタ $k1 にコピー
  lw   $k1, 0($k1)    # ページ・テーブル・エントリを一時レジスタ $k1 にロード
  mtc0 $k1, EntryLo   # ページ・テーブル・エントリを特殊レジスタ EntryLo にコピー
  tlbwr               # EntryLo を Random に従い TLB エントリに書き込む
  eret                # TLB ミス例外から戻る
```

5.7　仮想記憶

MIPS の場合

- ページ・フォルトした場合
  - アクティブなプロセスの全状態を退避
  - 長いので省略……

x86 の場合

- 命令を再開可能にするのが難しい
  - 複雑な命令 (例: 何千語をも対象とするブロック転送命令) のせいで
    命令を最初から実行し直すことができない場合がある

5.7　仮想記憶

まとめ

- 仮想記憶の役割
  - 主記憶容量の制限を越える
  - プロセス間での主記憶共有の保護機構
- ミス率の削減
  - ページのサイズを大きくする
  - ページ・テーブルによるフル・アソシアティブ方式
  - OS による LRU や参照ビットなどの技法
- ライト・バック、ダーティ・ビット
- アドレス変換機構は OS のみが扱えてユーザー・プログラムは扱えない
- TLB によるキャッシュ

演習問題

パタヘネの問題、解きづらくない？

- 2019 年度　電子情報学専攻　専門　第 2 問
- 2019 年度　コンピュータ科学専攻　専門科目 I　問題 2

演習問題

2019 年度　電子情報学専攻　専門　第 2 問

仮想記憶により、プログラマは計算機が物理記憶よりも巨大な主記憶を持つかのように扱うことができる。
以下の問いに答えよ。

(2) ページテーブルが図 1 に示すように単一の配列で実装されているとする。仮想アドレス空間のサイズが 256 TBytes (= 2^{48} Bytes)、ページサイズが 4 KBytes (= 2^{12} Bytes)、各ページテーブルエントリのサイズが 8 Bytes の時、ページテーブルを格納するのに必要となる主記憶のサイズを答えよ。

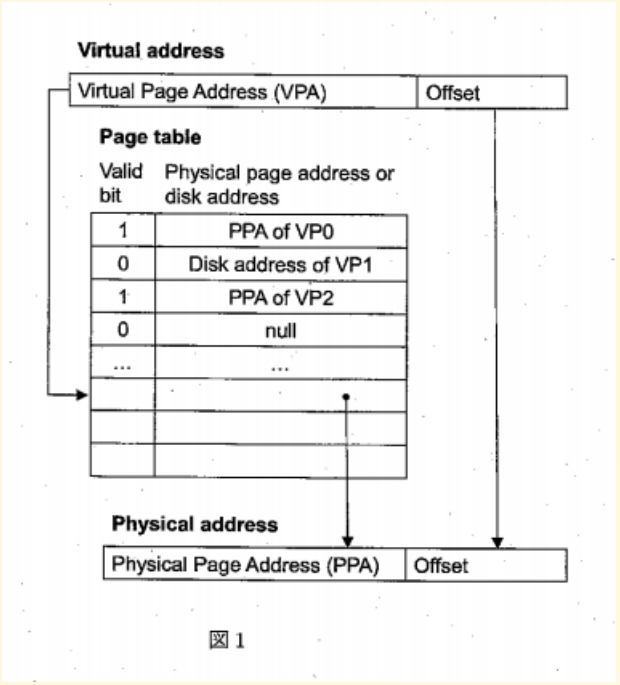


図 1

解答

$$
\begin{aligned}
2^{48} \div 2^{12} \times 2^{3} &= 2^{39}\ \text{Bytes} \\
&= 512\ \text{GBytes}
\end{aligned}
$$

演習問題

2019 年度　コンピュータ科学専攻　専門科目 I　問題 2

32 KB の物理メモリを有する 32 ビットのマシン上でオペレーティングシステムがページング機能を提供する場合を考える。ページサイズは 4 KB、仮想メモリのサイズは 4 GB であり、キャッシュメモリは存在しない。以下の問いに答えよ。なお、1 KB は 1024 バイトであるとする。

(2) 以下のページテーブルが与えられた際の仮想アドレス 2A0F (16 進数) に対応する物理アドレスを 16 進数で求めよ。

ページ番号（10 進数）	フレーム番号（2 進数）	有効ビット
0	111	1
1	000	0
2	110	1
3	000	0
4	101	1
5	000	0
6	000	0
7	000	0
8	000	0
9	001	1
10	100	1
11	000	1
12	011	1
13	000	0
14	000	0
15	010	1

解答

仮想アドレスは $2\mathrm{A0F}_{16} = 0010\ 1010\ 0000\ 1111_2$

↓

仮想ページ番号は $0010_2 = 2_{10}$

ページオフセットは $1010\ 0000\ 1111_2$

↓

物理ページ番号は 110_2

有効ビットが 1 を確認

↓

物理アドレスは $110\ 1010\ 0000\ 1111_2 = 6\mathrm{A0F}_{16}$