

Operating Systems Three Easy Pieces (in Japanese)

Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau (University of Wisconsin-Madison)
日本語訳: syarochan

2021年1月20日

第 I 部

Virtualization

1. 本の対話

教授：この本にようこと！ この本は、Operating Systems in Three Easy Pieces と呼ばれていて、ここではオペレーティングシステムについて知つておくべきことを教えています。私は“教授”と呼ばれています。あなたは誰ですか？

学生：こんにちは教授！ 私はあなたが思ったとおり、「学生」と呼ばれています。さっそく学んでいきたいです！

教授：いいですね。質問はありますか？

学生：はい！ なぜ“Three Easy Pieces”と呼ばれているんですか？

教授：それは簡単なことです。ええと、ご存知の通り、リチャード・フェインマンの物理学に関する講義があつて…

学生：ああ！ 「Surely You’re Joking, Mr. Feynman」を書いた人のことですか？ 素晴らしい本ですよね！ その本と何か関係が？

教授：ええと… その本ではないですね。彼が書いたその本は素晴らしいですよ。よかつたらあなたも読んでみてください。彼が書いた本は物理学に関する基本である「Six Easy Pieces」という本でまとめられています。それを私たちは OS の話題について 3 つの簡単なステップで行うつもりです。OS は物理学の半分ほど難しさなので三つのステップにしたということがあります。

学生：私は物理学が好きなので、それはおそらく良い本なのでしょうね。3 つの簡単なステップというのは具体的には何ですか？

教授：それは、私たちが学ぶべき 3 つの重要なものである仮想化、並行性、永続性です。これらのアイデアを学ぶにあたり、OS の仕組み、つまり CPU 上でどのプログラムを次に実行するか、どのように仮想メモリシステムのメモリ過負荷を処理するか、仮想マシンモニタの動作方法、ディスクに関する情報を管理する方法、また少しではありますが部品が故障したときに動作する分散システムを構築する方法についても説明します。具体的に言つたらそういうものですね。

学生：私はあなたが何について話しているのか分かりません。

教授：大丈夫ですよ！ それはこれから学んでいけばいいんですよ。

学生：別の質問があります：これらのこと学ぶのに最も良い方法は何ですか？

教授：よい質問ですね！ まあ、それぞれの人が自分でこれらのことを理解する必要がありますが、ここであなたが何をすべきかということについては：教授が教材を紹介するのを聞くためにクラスに行く。そして、毎週末に、これらのメモを読んで、あなたの頭の中にある知識が少しでも入っていくようにしましょう。もちろん、しばらくしてから（ヒント：試験前に！）、メモをもう一度読んで知識を固めましょう。もちろん、あなたの教授はいくつかの宿題やプロジェクトを割り当てていくでしょう。特に、実際の問題を解決するために、プロジェクトを参加しているとき、これらのノート内のアイデアを基に行動していくことが最良の方法です。孔子が言ったように…

学生：ああ、私は知っています！ 「聞いて忘れる。見ると覚える。しては理解する」

教授：（驚いたことに）私が何を言おうとしていたのか知っていたのですか！？

学生：定型文のようなものでしたので…。また、私は孔子の大ファンであり、実際にこの引用文のより良い出典である荀子のさらに大ファンです。

教授：（驚いた） これからお互いにうまくやっていきそうですね！

学生：教授、もう一度質問させてください。これらの対話の目的は何ですか？ 私の考えではメインとしてやるのはこの本だけだと思いますが？ なぜ、これらの資料でだけで説明をしないのですか？

教授：良い質問だね！ 説明しますと、物語の外に身を引っ張って少し考えてみるのが時々実世界では役に立ちます。これらの対話はそのために必要なです。二人で、これらのかなり複雑な知識をすべて理解するため

に協力していくのです。あなたにその役割を任せてもいいですか？

学生：だから一緒に考える必要があるということですね？もちろんです。私は他に何をすればよいでしょうか？私はこの本から出たことないので人生というものがないですけど…

教授：私もそうですね、悲しいことに。まあ、とりあえずやっていきましょう！

2. オペレーティングシステムの概要

学部のオペレーティングシステムコースを受講している場合は、コンピュータプログラムが実行されたときに何が行われているかについての知識が必要です。そうでなければこの本（と対応するコース）を読むことは困難です。（Patt / Patel [PP03] と特に Bryant / O'Hallaron [BOH10] の両方がかなり素晴らしい本です。）そのため、わからない場合はこの文書を読んだり、最寄りの書店に行ったりしてください。

さっそく質問です。プログラムを実行するとどうなりますか？

実行中のプログラムは非常に簡単なことをしています。命令を実行します。それは毎秒何百万という（そして最近では数十億もの）何百もの時間にプロセッサはメモリから命令をフェッチし、それをデコードし（すなわち、これがどの命令であるかを調べる）、それを実行する（すなわち、2つの数値を加算する、メモリにアクセスする、条件をチェックする、関数にジャンプする、など）。この命令で処理が完了すると、プロセッサは次の命令に進みます。以下同様に、プログラムが最後に完了するまで続きます [1]。

このように、我々はちょうどコンピュータのフォンノイマンモデルの基礎を述べた [2]。シンプルに聞こえるでしょう？ しかし、このクラスでは、プログラムの実行中に、システムを使いやすくすることを第一の目標として進めていきます。

[1] もちろん、現代のプロセッサは、プログラムをより速く実行させるために多くの不思議なことを行います。複数の命令を一度に実行するだけでなく、命令の発行や完了の順番を入れ替えてしまうのです！ しかし、ここでそのことを心配するのはやめましょう。我々は今、シンプルなモデルだけに興味があります。外から見る限り、ほとんどのプログラムの命令は一度に1つずつ、規則正しく順番通りに実行されます。[2] フォンノイマンは、コンピューティングシステムの初期のパイオニアの一人でした。彼はまた、ゲーム理論と原爆に関する先駆的な仕事をし、NBAで6年間プレーしました。OK、そのうちの1つは真実ではありません。

問題：リソースを仮想化する方法

この本で答える中心的な質問の1つは、オペレーティングシステムがリソースをどのように仮想化するのかという単純なものです。これが問題の要点です。OSはなぜ仮想化をするのでしょうか？ 答えはシステムを使いやすくするためです。したがって、これらを理解するために、どのようにメカニズムに焦点を当てていますか？ OSはどのように効率的に機能しますか？ どのハードウェアサポートが必要ですか？

実際には、プログラムを実行しやすく（一度に多くのことを実行できるようにしても）、プログラムがメモリを共有できるようにし、プログラムとデバイスとのやりとりや他の作業を可能にする責任があります。ソフトウェアの本体は、システムが使いやすく簡単かつ正確に動作することを確認する責任を負っているため、オペレーティングシステム(OS)[3]と呼ばれています。

OSが主に行う主な方法は、仮想化と呼ばれる一般的な手法です。つまり、OSは物理リソース（プロセッサ、メモリ、ディスクなど）を取り出し、より一般的で強力で使いやすい仮想形式に変換します。したがって、オペレーティングシステムを仮想マシンと呼ぶことがあります。

もちろん、ユーザーがOSに何をすべきかを伝え、仮想マシンの機能（プログラムの実行、メモリの割り当て、ファイルへのアクセスなど）を利用できるようにするため、OSはいくつかのインターフェースも提供します（API）。また呼び出すことができます。実際、典型的なOSは、アプリケーションで利用できる数百のシステムコールをエクスポートします。OSはプログラムを実行したり、メモリやデバイスにアクセスしたり、その他の関連するアクションを呼び出すため、OSはアプリケーションに標準ライブラリを提供することもあります。

最後に、仮想化は多くのプログラムを実行し（CPUを共有する）、多くのプログラムが同時に自分の命令とデータ（つまり共有メモリ）にアクセスし、デバイスにアクセスするための多くのプログラム（したがってディスクなどを共有する）では、リソースマネージャーと呼ばれることもあります。CPU、メモリ、およびディス

クはそれぞれ、システムのリソースです。したがって、これらのリソースを管理するオペレーティングシステムの役割は、効率的に、公正に、または他の多くの可能な目標を念頭に置きながら行うことです。OS の役割を少しづつ理解するために、いくつかの例を見てみましょう。

[3] OS の初期の別の名前は、スーパーバイザーまたはマスター・コントロール・プログラムでした。どうやら、後者はちょっと過激に聞こえました（詳細は Tron の映画を見てください）。だから、ありがたいことに、“オペレーティング・システム”が代わりました。

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/time.h>
4 #include <assert.h>
5 #include "common.h"
6
7 int
8 main(int argc, char *argv[])
9 {
10     if (argc != 2) {
11         fprintf(stderr, "usage: cpu <string>\n");
12         exit(1);
13     }
14     char *str = argv[1];
15     while (1) {
16         Spin(1);
17         printf("%s\n", str);
18     }
19     return 0;
20 }
```

Figure 2.1: Simple Example: Code That Loops and Prints (`cpu.c`)

2.1 CPU の仮想化

図 2.1 に最初のプログラムを示します。大したことはやっていません。実際には、`Spin()` は、一度実行されると時間と戻り値を繰り返しチェックする関数です。次に、ユーザーがコマンドラインで渡した文字列を出力し、永遠に繰り返します。

このファイルを `cpu.c` として保存し、シングルプロセッサ（CPU と呼ぶこともあります）を持つシステム上でコンパイルして実行することにします。

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu "A"
A
A
A
A
^C
prompt>
```

さほど面白い実行結果でもありません。システムはプログラムの実行を開始し、1 秒が経過するまで時間を繰り返し確認します。1 秒が経過すると、コードはユーザーが渡した入力文字列（この例では文字 “A”）を出力し、処理を続けます。プログラムは永遠に実行されることに注意してください。“Control-c”（UNIX ベースのシステムではフォアグラウンドで実行中のプログラムを終了させる）を押すだけでプログラムを停止できます。

さて、同じことをやろうが、今度はこの同じプログラムの多くの異なるインスタンスを実行しましょう。図 2.2 に、このやや複雑な例の結果を示します。

```

prompt> ./cpu A & ; ./cpu B & ; ./cpu C & ; ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
C
B
D
...

```

Figure 2.2: Running Many Programs At Once

さて、今度は少し面白いことになっています。プロセッサは一つだけなのに、どういうわけか 4 つのプログラムすべてが同時に動いているように見えます！この魔法はどうやって起こるのでしょうか？ [4]

オペレーティングシステムは、ハードウェアの助けを借りて、この錯覚、すなわち、システムに非常に多数の仮想 CPU があるという錯覚を担当していることが分かります。単一の（または少数の）CPU を無限個の CPU に変換することで多くのプログラムを見かけ上一度に実行できるようにすることが CPU の仮想化であり、本書の第一の主題の焦点です。もちろん、プログラムを実行して停止させたり、実行するプログラムを OS に指示するには、OS に希望を伝えるために使用できるインターフェース（API）が必要です。この本では、これらの API について説明します。もちろんこれは、ほとんどのユーザーがオペレーティングシステムと対話する主な方法です

また、一度に複数のプログラムを実行できることにより、あらゆる種類の新しい質問が生まれます。たとえば、2 つのプログラムを特定の時間に実行したい場合、実行する必要がありますか？この質問は、OS のポリシーによって解決されます。ポリシーはこれらのタイプの質問に答えるために OS 内の多くの異なる場所で使用されるため、オペレーティングシステムが実装する基本的なメカニズム（複数のプログラムを一度に実行する能力など）について学びます。

[4] &記号を使用して、同時に 4 つのプロセスをどのように実行したかに注意してください。そうすることで、tcsd シェルのバックグラウンドでジョブが実行されます。つまり、ユーザーは次のコマンドをすぐに発行できます。この場合、実行する別のプログラムです。コマンド間のセミコロンで、tcsd で同時に複数のプログラムを実行することができます。別のシェル（bash など）を使用している場合は、動作が少し異なります。詳細については、オンラインでドキュメントを参照してください。

```

1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "common.h"
5
6  int
7  main(int argc, char *argv[])
8  {
9      int *p = malloc(sizeof(int));           // a1
10     assert(p != NULL);
11     printf("(%d) address pointed to by p: %p\n",
12            getpid(), p);                 // a2
13     *p = 0;                                // a3
14     while (1) {
15         Spin(1);
16         *p = *p + 1;
17         printf("(%d) p: %d\n", getpid(), *p); // a4
18     }
19     return 0;
20 }
```

Figure 2.3: A Program that Accesses Memory (`mem.c`)

2.2 メモリの仮想化

さあ、メモリを考えましょう。現代の機械によって提示される物理的記憶のモデルは非常に簡単です。メモリは単なるバイト配列です。メモリを読み取るには、そこに格納されているデータにアクセスできるようにアドレスを指定する必要があります。メモリを書き込む（または更新する）ためには、与えられたアドレスに書き込まれるデータも指定しなければなりません。

メモリは、プログラムが実行されているときは常にアクセスされます。プログラムは、すべてのデータ構造をメモリに保持し、ロードやストアなどのさまざまな命令や、作業中にメモリにアクセスする他の明示的な命令によってアクセスします。プログラムの各命令もメモリに記憶されていることを忘れないでください。したがって、メモリは各命令フェッチでアクセスされます。

`malloc()` を呼び出してメモリを割り当てるプログラム（図 2.3）を見てみましょう。このプログラムの出力はここにあります：

```

prompt> ./mem
(2134) address pointed to by p: 0x200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C
```

プログラムはいくつかのことを行います。まず、いくつかのメモリを割り当てます（行 a1）。次に、メモリ（a2）のアドレスをプリントアウトした後、新たに割り当てられたメモリ（a3）の最初のスロットに番号 0 を入れます。最後に、ループし、1 秒間遅延し、p に保持されているアドレスに格納されている値をインクリメントします。print 文ごとに、実行中のプログラムのプロセス識別子（PID）と呼ばれるものも表示されます。この PID は、実行中のプロセスごとに一意です。

```

prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) address pointed to by p: 0x200000
(24114) address pointed to by p: 0x200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...

```

Figure 2.4: Running The Memory Program Multiple Times

ここでも、この最初の結果はそれほど興味深いものではありません。新しく割り当てられたメモリはアドレス 0x200000 にあります。プログラムが実行されると、値がゆっくりと更新され、結果が output されます。

今度は、同じプログラムの複数のインスタンスを再度実行して、何が起こるかを確認します (図 2.4)。この例では、実行中の各プログラムが同じアドレス (0x200000) にメモリを割り当てていますが、それぞれが独立して 0x200000 の値を更新しているようです。これは、実行中の各プログラムが、他の実行中のプログラム [5] と同じ物理メモリを共有するのではなく、独自のプライベートメモリを持つかのようです。

確かに、OS がメモリを仮想化しているので、これがまさにここで起こっていることです。各プロセスは、独自のプライベート仮想アドレス空間 (アドレス空間と呼ばれることがある) にアクセスし、OS は何らかの形でマシンの物理メモリにマップします。実行中のプログラム内のメモリ参照は、他のプロセス (または OS 自体) のアドレス空間には影響しません。実行中のプログラムに関する限り、それ自身に物理的な記憶があります。しかし、実際には、物理メモリはオペレーティングシステムによって管理される共有リソースです。まさしくこのすべてがどのように達成されたかは、本書の最初の部分の主題であり、仮想化の話題です。

2.3 同時実行

この本の別の主なテーマは並行性です。この概念的な用語は、同じプログラム内で同時に (すなわち、同時に)多くのことを処理する際に発生する多数の問題を指すために使用され、対処されなければいけません。並行性の問題は、オペレーティングシステム自体の中で最初に生じました。上記の仮想化の例でもわかるように、OS は多くのことを一度に処理しています。最初に 1 つのプロセスを実行し、次に別のプロセスなどを実行しています。

[5] この例が機能するには、アドレス空間のランダム化が無効になっていることを確認する必要があります。ランダム化は、特定の種類のセキュリティ上の欠陥に対する優れた防御となり得ることが判明しました。スタックストミング攻撃を介してコンピュータシステムに侵入する方法を学びたい場合は、特に自分自身でそれについて詳しく読むことができます。

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "common.h"
4
5  volatile int counter = 0;
6  int loops;
7
8  void *worker(void *arg) {
9      int i;
10     for (i = 0; i < loops; i++) {
11         counter++;
12     }
13     return NULL;
14 }
15
16 int
17 main(int argc, char *argv[])
18 {
19     if (argc != 2) {
20         fprintf(stderr, "usage: threads <value>\n");
21         exit(1);
22     }
23     loops = atoi(argv[1]);
24     pthread_t p1, p2;
25     printf("Initial value : %d\n", counter);
26
27     Pthread_create(&p1, NULL, worker, NULL);
28     Pthread_create(&p2, NULL, worker, NULL);
29     Pthread_join(p1, NULL);
30     Pthread_join(p2, NULL);
31     printf("Final value : %d\n", counter);
32     return 0;
33 }

```

Figure 2.5: A Multi-threaded Program (*threads.c*)

残念ながら、並行性の問題はもはや OS 自体に制限されなくなりました。実際、現代のマルチスレッドプログラムは同じ問題を抱えています。マルチスレッドプログラムの例を示してみましょう（図 2.5）。

この例を今は完全に理解していないかもしれません（そして、後の章、同時実行の本の節でそれについてもっと学びます）、基本的な考え方は簡単です。メインプログラムは、`Pthread_create()`[6] を使用して 2 つのスレッドを作成します。スレッドは、他の関数と同じメモリ空間内で実行される関数を考えることができます。複数のスレッドを同時にアクティブにすることができます。この例では、各スレッドは `worker()` というルーチンで実行を開始します。このルーチンでは、ループ内のカウンタをループ回数だけインクリメントします。

[6] 実際の呼び出しは小文字にする必要があります。`pthread create()` 大文字のバージョンは独自のランナーで、`pthread create()` を呼び出し、戻りコードが呼び出しが成功したことを示します。詳細については、コードを参照してください。

問題のクラウド：正しい並行プログラムを作る方法

同じメモリ空間内に複数のスレッドを同時に実行すると、どうすれば正しく動作するプログラムを構築できますか？ どのプリミティブが OS から必要ですか？ どのような仕組みがハードウェアによって提供されるべきですか？ どのようにそれらを使用して並行性の問題を解決できますか？

以下は、変数ループの入力値を 1000 に設定してこのプログラムを実行すると何が起こるかを示すものです。ループの値によって、2 人の作業者のそれぞれがループ内の共有カウンタを増分する回数が決まります。ループの値を 1000 に設定してプログラムを実行すると、カウンタの最終値はどのようになるでしょうか？

```

prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value   : 2000

```

おそらく推測したように、2つのスレッドが終了すると、カウンタの最終値は2000になり、各スレッドはカウンタを1000回インクリメントします。実際、ループの入力値をNに設定すると、プログラムの最終出力は $2N$ になると予想されます。しかし、実際はそれほど単純ではありません。同じプログラムを実行しますが、ループの値が高いほど、何が起こるか見てみましょう。

```

prompt> ./thread 100000
Initial value : 0
Final value   : 143012      // huh??
prompt> ./thread 100000
Initial value : 0
Final value   : 137298      // what the??

```

この実行では、100,000の入力値を与えたときに最終値200,000を取得する代わりに、最初に143,012を取得します。次に、プログラムを2回実行すると、間違った値を取得するだけでなく、前回とは異なる値を取得します。実際には、高い値のループで繰り返しプログラムを実行すると、正しい答えが得られることがあります。それでなぜこれが起こっているのですか？

判明したように、これらの奇妙で珍しい結果の理由は、命令がどのように実行されるかに関連しており、これは一度に1つです。残念ながら、共有カウンタがインクリメントされる上記のプログラムの重要な部分は、カウンタの値をメモリからレジスタにロードする命令と、レジスタをインクリメントする命令と、メモリに格納する命令の3つの命令です。これらの3つの命令は原子的に(一度に)実行されないため、奇妙なことが起こる可能性があります。並行性のこの問題は、この本の第2部で詳しく説明します。

```

1 #include <stdio.h>
2 #include <unistd.h>
3 #include <assert.h>
4 #include <fcntl.h>
5 #include <sys/types.h>
6
7 int
8 main(int argc, char *argv[])
9 {
10     int fd = open("/tmp/file", O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU);
11     assert(fd > -1);
12     int rc = write(fd, "hello world\n", 13);
13     assert(rc == 13);
14     close(fd);
15     return 0;
16 }

```

Figure 2.6: A Program That Does I/O (`io.c`)

2.4 永続性

コースの3番目の主要テーマは持続性です。システムメモリでは、DRAMのようなデバイスが値を揮発性の方法で格納するため、データを簡単に失う可能性があります。電源が切れるかシステムがクラッシュすると、メモリ内のデータは失われます。したがって、データを永続的に保存できるようにするために、ハードウェアとソフトウェアが必要です。このようなストレージは、ユーザーがデータを大事にしているため、あらゆるシステムにとって重要です。

ハードウェアは、ある種の入力/出力またはI/Oデバイスの形で提供されます。現代のシステムでは、ハー

ドライブは長寿命の情報のための共通リポジトリですが、ソリッドステートドライブ (SSD) もこの分野で進んでいます。

通常、ディスクを管理するオペレーティングシステムのソフトウェアは、ファイルシステムと呼ばれます。したがって、ユーザがシステムのディスク上に信頼できる効率的な方法で作成したファイルを格納する責任があります。

CPU とメモリ用に OS が提供する抽象化とは異なり、OS はアプリケーションごとにプライベートの仮想化ディスクを作成しません。むしろ、ユーザーはよく、ファイル内の情報を共有したいと考えられています。例えば、C プログラムを書くときは、最初に Emacs^[7]などのエディタを使って C ファイル (emacs -nw main.c) を作成し編集することができます。いったん完了したら、コンパイラを使用してソースコードを実行可能ファイルにすることができます (gcc -o main main.c など)。作業が終了したら、新しい実行可能ファイル (./main など) を実行することができます。したがって、異なるプロセス間でファイルがどのように共有されているかを見ることができます。まず、Emacs はコンパイラへの入力として機能するファイルを作成します。コンパイラはその入力ファイルを使用して新しい実行可能ファイルを作成します (多くの手順で、詳細についてはコンパイラコースを参照してください)。最後に、新しい実行可能ファイルが実行されます。そして、新しいプログラムが生まれました！

これをよりよく理解するために、いくつかのコードを見てみましょう。図 2.6 に、文字列 “hello world” を含むファイル (/tmp/file) を作成するためのコードを示します。

[7] Emacs を使うべきです。vi を使用している場合、おそらく何か問題があります。実際のコードエディタではないものを使用している場合、それはさらに悪化します。

問題のクラウド：データを永続的に保存する方法

ファイルシステムは、永続データの管理を担当する OS の一部です。そうするためにはどのような技術が必要ですか？ 高性能でこれを実現するためにはどのような仕組みとポリシーが必要ですか？ ハードウェアとソフトウェアの不具合に直面して、信頼性はどのように達成されましたか？

このタスクを達成するために、プログラムはオペレーティングシステムに 3 回の呼び出しを行います。最初に、open() の呼び出しがファイルを開き、それを作成します。2 番目の write() はファイルに何らかのデータを書き込みます。3 番目の close() は、ファイルを閉じて、プログラムがこれ以上データを書き込んでいないことを示します。これらのシステムコールは、オペレーティングシステムのファイルシステムと呼ばれる部分にルーティングされます。ファイルシステムは要求を処理し、ユーザーに何らかのエラーコードを返します。

実際にディスクに書き込むために OS が何をしているのか疑問に思うかもしれません。私たちはあなたを見てくれるでしょうが、まず目を閉じることを約束しなければなりません。ファイルシステムはかなりの作業をしなければなりません。まず、ディスク上のどこにこの新しいデータが存在するかを把握し、ファイルシステムが維持しているさまざまな構造でそれを追跡します。そうするために、基盤となるストレージデバイスに I/O 要求を発行し、既存の構造を読み込んだり、更新 (書き込み) する必要があります。デバイスドライバ^[8]を書いた人なら誰もが知っているように、デバイスをあなたのために何かすることは、複雑で詳細なプロセスです。低レベルのデバイスインターフェースとその正確な意味についての深い知識が必要です。幸いにも、OS は、システムコールを通じてデバイスにアクセスする標準的でシンプルな方法を提供します。したがって、OS は標準ライブラリと見なされることがあります。

もちろん、デバイスがどのようにアクセスされるか、ファイルシステムがそのデバイス上でデータを永続的に管理する方法については、より多くの詳細があります。パフォーマンス上の理由から、ほとんどのファイルシステムでは、最初にそのような書き込みをしばらくの間遅延させて、より大きなグループにバッチすることを望んでいます。書き込み中のシステムクラッシュの問題を処理するために、ほとんどのファイルシステムには、ジャーナリングやコピーオンライトなどの複雑な書き込みプロトコルが組み込まれています。書き込み

シーケンス中にエラーが発生した場合、後で合理的な状態に回復することができます。異なる一般的な操作を効率的にするために、ファイルシステムは、単純なリストから複雑なツリーに至るまで、多くの異なるデータ構造とアクセス方法を採用しています。この本の第3部では、デバイスとI/O、ディスク、RAID、ファイルシステムについて詳しく説明します。

[8] デバイスドライバは、特定のデバイスをどう扱うかを知っているオペレーティングシステムのコードです。デバイスとデバイスドライバについては後で詳しく説明します。

2.5 デザイン目標

OSはCPU、メモリ、ディスクなどの物理的なリソースを取り、それらを仮想化するという考えを持っています。これは、並行性に関連する難しくて厄介な問題を処理します。また、ファイルを永続的に保存するため、長期的に安全になります。このようなシステムを構築したいと考えているので、設計と実装に集中し、必要に応じてトレードオフを行うためにいくつかの目標を念頭に置いていきたいと考えています。適切なトレードオフのセットを見つけることは、システム構築の鍵です。

最も基本的な目標の1つは、システムを便利で使いやすいものにするために、いくつかの抽象化を構築することです。抽象は、私たちがコンピュータサイエンスでやるすべてにとって基本的なものです。抽象化は、大規模なプログラムを小さく分かりやすく分け、アセンブリなどを考えずにC[9]のような高級言語で記述したり、論理ゲートを考えずにコードを書いたり、構築したりすることができますトランジスタについてあまり考えずにゲートから外に出るプロセッサー。抽象化は非常に基本的なもので、時にはその重要性を忘れることがありますが、ここでは取り上げません。したがって、各セクションでは、時間の経過とともに発展した主要な抽象化についていくつか議論し、OSの部分について考える方法を説明します。

オペレーティングシステムの設計と実装の1つの目標は、高性能を提供することです。これを言うもう1つの方法は、OSのオーバーヘッドを最小限に抑えることです。仮想化とシステムを使いやすくすることは価値がありますが、コストはかかりません。したがって、我々は過度のオーバーヘッドなしに仮想化や他のOS機能を提供するよう努めなければなりません。これらのオーバーヘッドは、余分な時間(より多くの命令)と余分なスペース(メモリ内またはディスク上)といういくつかの形式で発生します。可能であれば、どちらか一方または両方、または両方を最小化するソリューションを探します。しかし、完璧さは必ずしも達成可能なわけではなく、私たちが気づくことを学び、(適切な場合には)容認するものです。

もう1つの目標は、アプリケーション間、およびOSとアプリケーション間の保護を提供することです。同時に多くのプログラムを実行できるようにしたいので、悪意のある、または偶発的な悪い行為が、他の行為に悪影響を及ぼさないようにしたいです。私たちは確かに、アプリケーションが(それがシステム上で動いているすべてのプログラムに影響するように)OSそのものを傷つけることはできないようにしたい。保護は、オペレーティングシステムの根底にある主な原則の1つ、つまり隔離の原則の中心にあります。プロセスを互いに隔離することは、保護の鍵であり、したがってOSが何をしなければならないかの根底にあります。

オペレーティングシステムもノンストップで実行する必要があります。それが失敗すると、システム上で実行されているすべてのアプリケーションも失敗します。この依存性のために、オペレーティングシステムはしばしば高い信頼性を提供するように努めています。オペレーティングシステムがますます複雑になり(時には何百万行ものコードを含む)、信頼性の高いオペレーティングシステムを構築することは非常に難しい課題です。実際には、現場で行われている多くの研究(BS + 09, SS + 10])は、この正確な問題に焦点を当てています。

他の目標は理にかなっています。エネルギー効率はますます緑の世界で重要です。悪意のあるアプリケーションに対するセキュリティ(実際には保護の拡張)は、特に高度にネットワーク化されたこれらの時期には重要です。OSは小型で小型のデバイス上で動作するため、モビリティはますます重要になっています。システムの使用方法によっては、OSの目標が異なるため、少なくともわずかに異なる方法で実装される可能性がある

ります。しかし、わかるように、OS を構築する方法について私たちが提示する多くの原則は、さまざまなデバイスに役立ちます。

[9] C 言語を高級言語と呼ぶことに反対する人もいます。これは OS のコースであることを覚えておいてください。ここでは、いつもアセンブリでコードを書く必要がないのがうれしいです！

2.5 いくつかの歴史

この紹介を終える前に、オペレーティングシステムがどのように開発されたかを簡単に説明しましょう。人間によって構築されたシステムのように、エンジニアは設計上重要なことを学んだので、時間の経過とともにオペレーティングシステムに良いアイデアが蓄積されました。ここでは、いくつかの主要な開発について説明します。開発方法については、Brinch Hansen の優れたオペレーティングシステムの歴史をご覧ください [BH00]。

初期のオペレーティングシステム：ジャストライブラリ

当初は、オペレーティングシステムはあまり働きませんでした。基本的には、一般的に使用される関数の単なるライブラリのセットでした。例えば、システムの各プログラマーに低レベルの I/O 処理コードを書き込ませる代わりに、「OS」はそのような API を提供し、したがって開発をより容易にします。

通常、これらの古いメインフレームシステムでは、人間のオペレータによって制御されるように、1つのプログラムが一度に 1 つずつ実行されました。現代の OS が何をすると思いますか（たとえば、ジョブを実行する順序を決めるなど）は、このオペレータによって実行されました。あなたがスマートな開発者だったなら、あなたはこのオペレータに親切であり、あなたの仕事をキューの前に移動させるかもしれません。

この計算モードはバッチ処理と呼ばれ、多数のジョブが設定され、オペレータによって「バッチ」で実行されるためです。その時点では、コンピュータはコストのためにインタラクティブな方法で使用されていませんでした。コンピュータの前に座って使用することは、あまりにも高価でした。ほとんどの場合、1 時間に数十万ドルの費用がかかりました [BH00]。

ライブラリを超えて：保護 オペレーティングシステムは、一般的に使用されるサービスの単純なライブラリではなく、マシンを管理する上でより中心的な役割を果たしました。これの重要な側面の 1 つは、OS に代わって実行されるコードが特別であることの認識でした。デバイスの制御を持っていたので、通常のアプリケーションコードとは異なった扱いをするべきです。どうしてでしょうか？ 別の方法としてディスク上のどこからでもアプリケーションを読み込めるようにしたらどうでしょうか？ 任意のプログラムがファイルを読み取ることができるように、プライバシーの概念が窓から出てきます。したがって、ライブラリとしてファイルシステムを実装することは意味がありません。代わりに、何かが必要でした。

したがって、Atlas コンピューティングシステム [K + 61, L78] によって開発されたシステムコールのアイデアが発明されました。OS ルーチンをライブラリとして提供する代わりに（ここでは、アクセスするためのプロシージャコールを行うだけです）、ハードウェア命令とハードウェアの特別なペアを追加して、OS への移行をより正式で制御されたプロセスにすることでした。

システムコールとプロシージャコールとの間の主な相違点は、システムコールがハードウェア特権レベルを上げると同時に OS に制御（すなわち、ジャンプ）を転送することです。ユーザーアプリケーションは、ユーザー モードと呼ばれるもので実行されます。つまり、ハードウェアによってアプリケーションが実行できる処理が制限されます。

たとえば、ユーザー モードで実行されているアプリケーションは、ディスクへの I/O 要求を開始したり、物理メモリページにアクセスしたり、ネットワーク上でパケットを送信したりすることはできません。システムコールが開始されると（通常はトラップと呼ばれる特別なハードウェア命令によって）、ハードウェアは事前に指定されたトラップハンドラ（以前に設定した OS）に制御を移し、同時にカーネル モードに特権レベルを上げ

ます。

カーネルモードでは、OS はシステムのハードウェアに完全にアクセスできるため、I/O 要求の開始やプログラムで使用可能なメモリの増加などが可能です。OS が要求を処理すると、特別な return-from-trap 命令を介してユーザーに制御が戻され、ユーザーモードに戻ります。同時に、アプリケーションが中断した場所に制御を戻します。

マルチプログラミングの時代

オペレーティングシステムが本当に始まったのは、メインフレームを超えたコンピューティングの時代、ミニコンピュータの時代でした。デジタル機器の PDP ファミリのような古典的なマシンは、コンピュータを非常に手頃な価格にしました。したがって、大規模組織ごとに 1 つのメインフレームを持つ代わりに、組織内の少数の人が自分のコンピュータを持つ可能性があります。驚くことではないが、このコスト低下の大きな影響の 1 つは、開発者の活動の増加であった。よりスマートな人々はコンピュータを手に入れ、コンピュータシステムをより面白く美しいものにしました。

特に、マルチプログラミングは、マシンリソースをより有効に利用したいという要望のために普及しました。一度に 1 つのジョブを実行するのではなく、複数のジョブをメモリにロードし、それらのジョブ間で迅速に切り替えて CPU 使用率を向上させます。I/O デバイスが遅いため、この切り替えは特に重要でした。その I/O が処理されている間に CPU 上でプログラムが待機するのは、CPU 時間の無駄でした。代わりに、別の仕事に切り替えてしばらく運転してみてはどうですか？

I/O と割り込みの存在下でマルチプログラミングとオーバーラップをサポートしたいという要望は、多くの方向性に沿ったオペレーティングシステムの概念開発における革新を強いられました。メモリ保護などの問題が重要になりました。あるプログラムが別のプログラムのメモリにアクセスすることはできません。マルチプログラミングによって導入された並行性の問題に対処する方法を理解することも重要でした。割り込みがあつても OS が正しく動作していることを確認することは大きな課題です。本書の後半で、これらの問題と関連するトピックを検討します。

その実用的な進歩の 1 つは、UNIX オペレーティングシステムの導入でした。これは主にベル研究所の Ken Thompson(および Dennis Ritchie)(電話会社)のおかげです。UNIX は、さまざまなオペレーティングシステム(特に Multics [O72]、TENEX [B + 72] や Berkeley TimeSharing System [S + 68] など)から多くの優れたアイデアを得ましたが、まもなく、このチームは世界中の人々に UNIX のソースコードを含むテープを出荷していました。多くの人が関与してシステムに追加されました。詳細は Aside(次のページ) を参照してください。

現代

ミニコンピュータの向こうには、新しいタイプのマシンが安くて速く、そして大衆にとって、今日のパソコン・コンピュータまたは PC が登場しました。ワークグループごとにミニコンピュータを共有するのではなく、低コストでデスクトップごとに 1 台のマシンを使用できるようになり、アップルの初期のマシン(Apple II など)と IBM PC を中心に、新しいタイプのマシンがすぐにコンピューティングの支配的な役割を果たすようになりました。

残念なことに、オペレーティングシステムでは、初期のシステムがミニコンピュータの時代に学んだ教訓を忘れてしまった(または知らなかった)ので、最初は PC が大きく飛躍しました。例えば、DOS のような初期のオペレーティングシステム(マイクロソフトのディスクオペレーティングシステム)は、メモリ保護が重要ではないと考えていました。したがって、悪意のある(または、あまりにもプログラムの悪い) アプリケーションは、メモリ全体を書くことができます。Mac OS の第 1 世代(v9 以前)は、ジョブスケジューリングに協力的なアプローチを取りました。したがって、偶発的に無限ループに突き当たったスレッドがシステム全体を引き継ぎ、再起動を強制する可能性があります。この世代のシステムに欠けている OS 機能の痛ましいリストは長

いですが、ここでの議論のためには長すぎます。

幸運なことに、何年もの苦しみの後、ミニコンピュータのオペレーティングシステムの古い機能がデスクトップに乗り始めました。たとえば、Mac OS X/macOS には、そのような成熟したシステムから期待されるすべての機能を含む、UNIX が中心です。Windows は、特に Windows NT を始めとして、Microsoft OS テクノロジの飛躍的な飛躍を踏まえて、コンピューティングの歴史において多くの素晴らしいアイデアを採用しています。今日の携帯電話でも、Linux などのオペレーティングシステムが稼動していますが、これは 1980 年代に PC を走らせたもの (1970 年代のミニコンピュータに似ています) に似ています。OS 開発の全盛期に開発された優れたアイデアが現代の世界に浸透していることがわかりました。さらに優れた点は、これらのアイデアが引き続き発展し、より多くの機能を提供し、ユーザーやアプリケーションにとって現代のシステムをより良くすることです。

ASIDE : UNIX の重要性

オペレーティングシステムの歴史において UNIX の重要性を誇張することは困難です。以前のシステム (特に、MIT の有名な Multics システム) の影響を受けて、UNIX は多くの素晴らしいアイデアを集め、シンプルで強力なシステムを作りました。オリジナルの “Bell Labs” の基盤となつた UNIX は、より強力なワークフローを形成するために一緒に接続できる、強力で強力なプログラムを構築する統一的な原則でした。コマンドを入力するシェルは、このようなメタレベルプログラミングを可能にするパイプなどのプリミティブを提供しました。したがって、より大きなタスクを達成するためにプログラムをまとめするのが容易になりました。たとえば、単語 “foo” を含むテキストファイルの行を見つけて、そのような行がいくつ存在するかを調べるには、grep foo file.txt | wc -l と入力して、grep と wc を使用します。

UNIX 環境はプログラマや開発者にとってもフレンドリーで、新しい C プログラミング言語のコンパイラも提供していました。プログラマーが自分のプログラムを書いたり共有したりすることが容易になり、UNIX は非常に人気がありました。そして、おそらく、著者たちは、初期の形式のオープンソースソフトウェアである、誰かに尋ねた人に無料でコピーを寄贈したことを多分助けてくれたでしょう。

また、コードのアクセシビリティと可読性も非常に重要でした。C で書かれた美しく小さなカーネルを持っている人は、カーネルで遊ぶように他の人を招き、新しくてクールな機能を追加しました。たとえば、Bill Joy が率いる Berkeley の企業グループは、高度な仮想メモリ、ファイルシステム、およびネットワークサブシステムを備えた素晴らしいディストリビューション (Berkeley Systems Distribution、または BSD) を作りました。Joy は後に Sun Microsystems を共同設立しました。

残念ながら、UNIX の普及は、企業が所有権と利益を主張しようとするにつれて少し遅くなりました。しかし、弁護士の不幸な (しかし共通の) 結果が関与しています。多くの企業には Sun Microsystems の SunOS、IBM の AIX、HP の HPUX、SGI の IRIX などの独自の変種がありました。AT & T/Bell Labs と他のプレイヤーの間の法的争いは、UNIX 上で暗い雲を投げかけ、Windows が導入され PC 市場の多くを占めるように生き残るかどうか、多くの人が疑問に思っていました。

ASIDE : そしてその後の Linux

幸運なことに、UNIX の場合、Linus Torvalds という若いフィンランドのハッカーは、元のシステムの背後にある原則とアイデアに大いに借りて、コードベースではなく、合法性の問題を避けて独自の UNIX バージョンを作成することに決めました。彼は世界中の多くの人たちの助けを得て、まもなく Linux が生まれました (そして現代のオープンソースソフトウェアの動き) インターネット時代が到来すると、Google、Amazon、Facebook などの大部分の企業は無料で、ニーズに合わせて簡単に変更できるように、Linux を実行することを選択しました。確かに、これらの新会社の

成功がそのようなシステムが存在しなかったと想像するのは難しいです。

スマートフォンが支配的なユーザー向けプラットフォームになるにつれて、Linux は同じ理由の多くのために (Android 経由で) そこにも拠点を見つけました。そして Steve Jobs は彼の UNIX ベースの NeXTStep オペレーティング環境を Apple に任せ、デスクトップ上で UNIX を普及させました (Apple の技術の多くのユーザーはおそらくこの事実を知らないでしょう)。そして、UNIX は今まで以上に重要な存在です。コンピューティングの神は、あなたがそれらを信じているならば、この素晴らしい結果に感謝しなければなりません。

2.7 概要

したがって、私たちは OS について紹介しています。今日のオペレーティングシステムでは、システムを比較的使いやすくし、今日使用しているほとんどすべてのオペレーティングシステムは、本書で説明する開発の影響を受けています。

残念なことに、時間的な制約のため、本書でカバーしていない OS のいくつかの部分があります。たとえば、オペレーティングシステムには多くのネットワーキングコードがあります。私たちはそれについてもっと学ぶためにあなたにネットワーキングクラスを取ることを任せます。同様に、グラフィックスデバイスは特に重要です。最後に、オペレーティングシステムの書籍の中には、セキュリティに関する大きな話題があります。OS は実行中のプログラム間で保護を提供し、ユーザーにファイルを保護する能力を与えるなければならないという意味でそうするでしょうが、セキュリティコースで見つかる、より深刻なセキュリティ問題を掘り下げません。

しかし、CPU とメモリの仮想化の基本、並行性、デバイスやファイルシステムによる永続性など、重要なトピックがたくさんあります。心配しないでください！ カバーすべき多くの土地がありますが、そのほとんどは非常に涼しく、道路の終わりには、コンピュータシステムが実際にどのように機能するかについて考えます。

#参考文献

[BS+09] “Tolerating File-System Mistakes with EnvyFS”

Lakshmi N. Bairavasundaram, Swaminathan Sundararaman, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

USENIX '09, San Diego, CA, June 2009

A fun paper about using multiple file systems at once to tolerate a mistake in any one of them.

[BH00] “The Evolution of Operating Systems”

P. Brinch Hansen

In Classic Operating Systems: From Batch Processing to Distributed Systems Springer-Verlag, New York, 2000

This essay provides an intro to a wonderful collection of papers about historically significant systems.

[B+72] “TENEX, A Paged Time Sharing System for the PDP-10”

Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, Raymond S. Tomlinson

CACM, Volume 15, Number 3, March 1972

TENEX has much of the machinery found in modern operating systems; read more about it to see how much innovation was already in place in the early 1970's.

[B75] “The Mythical Man-Month”

Fred Brooks

Addison-Wesley, 1975

A classic text on software engineering; well worth the read.

[BOH10] “Computer Systems: A Programmer’s Perspective”

Randal E. Bryant and David R. O’Hallaron

Addison-Wesley, 2010

Another great intro to how computer systems work. Has a little bit of overlap with this book — so if you'd like, you can skip the last few chapters of that book, or simply read them to get a different perspective on some of the same material. After all, one good way to build up your own knowledge is to hear as many other perspectives as possible, and then develop your own opinion and thoughts on the matter. You know, by thinking!

[G85] “The GNU Manifesto”

Richard Stallman, 1985

Available: <https://www.gnu.org/gnu/manifesto.html>

A huge part of Linux's success was no doubt the presence of an excellent compiler, gcc, and other relevant pieces of open software, all thanks to the GNU effort headed by Richard Stallman. Stallman is quite a visionary when it comes to open source, and this manifesto lays out his thoughts as to why; well worth the read.

[K+61] “One-Level Storage System”

T. Kilburn, D.B.G. Edwards, M.J. Lanigan, F.H. Sumner

IRE Transactions on Electronic Computers, April 1962

The Atlas pioneered much of what you see in modern systems. However, this paper is not the best read. If you were to only read one, you might try the historical perspective below [L78].

[L78] “The Manchester Mark I and Atlas: A Historical Perspective”

S. H. Lavington

Communications of the ACM archive

Volume 21, Issue 1 (January 1978), pages 4-12

A nice piece of history on the early development of computer systems and the pioneering efforts of the Atlas. Of course, one could go back and read the Atlas papers themselves, but this paper provides a great overview and adds some historical perspective.

[O72] “The Multics System: An Examination of its Structure”

Elliott Organick, 1972

A great overview of Multics. So many good ideas, and yet it was an over-designed system, shooting for too much, and thus never really worked as expected. A classic example of what Fred Brooks would call the “second-system effect” [B75].

[PP03] “Introduction to Computing Systems:

From Bits and Gates to C and Beyond”

Yale N. Patt and Sanjay J. Patel

McGraw-Hill, 2003

One of our favorite intro to computing systems books. Starts at transistors and gets you all the way up to C; the early material is particularly great.

[RT74] “The UNIX Time-Sharing System”

Dennis M. Ritchie and Ken Thompson

CACM, Volume 17, Number 7, July 1974, pages 365-375

A great summary of UNIX written as it was taking over the world of computing, by the people who wrote it.

[S68] “SDS 940 Time-Sharing System”

Scientific Data Systems Inc.

TECHNICAL MANUAL, SDS 90 11168 August 1968

Available: <http://goo.gl/EN0Zrn>

Yes, a technical manual was the best we could find. But it is fascinating to read these old system documents, and see how much was already in place in the late 1960's. One of the minds behind the Berkeley Time-Sharing System (which eventually became the SDS system) was Butler Lampson, who later won a Turing award for his contributions in systems.

[SS+10] "Membrane: Operating System Support for Restartable File Systems"

Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Michael M. Swift

FAST '10, San Jose, CA, February 2010

The great thing about writing your own class notes: you can advertise your own research. But this paper is actually pretty neat — when a file system hits a bug and crashes, Membrane auto-magically restarts it, all without applications or the rest of the system being affected.

3. 仮想化に関する対話

教授：そして、オペレーティングシステム上の 3 つの主要な部分のうち、最初の部分である仮想化のお話になります。

学生：しかし、偉大なる先生、仮想化とは何ですか？

教授：桃があると想像してください。

学生：桃？（信用できない）

教授：はい、桃です。それを物理的な桃と呼ぶことにしよう。しかし、この桃を食べたい多くの人がいるとしましょう。それぞれ食べたい人全員に桃をあげることができたら全員幸せですよね。私たちは食べる桃を仮想化された桃と呼んでいます。私たちは何とか一つの物理的な桃からこれらの仮想化された桃を多く作り出します。そして、重要なことは：この錯覚では、彼らは物理的な桃を持っているように見えますが、実際にはそうではありません。

学生：桃を分けあっているように見えるけど、実際は平等に配られていないことを知らないのですか？

教授：そうです！

学生：しかし、たった一つの桃があります。

教授：はい。そして…？

学生：もし私が一つの桃しかもっていないことを知っていて、他の人と桃を分けていたら、私は桃が平等に配られないことに気づくと思う。

教授：そうです！いい視点ですね。しかし、桃を多く食べたい人がいるのです。ほとんどの時間彼らは昼寝や何か他のことをしているので、あなたはその桃を奪って、しばらくの間他の誰かに桃を与えることができます。それで、私たちは多くのバーチャル・ピーチの錯覚を作り出します。one peach for each person!

学生：悪いキャンペーンのスローガンのように聞こえます。あなたはコンピュータについて話していますが、本当にコンピュータ専攻の教授ですか？

教授：まあ待ちなさい若者よ、あなたにこれからより具体的な例を出しましょう。良いアイデアを思いついた！最も基本的なリソースである CPU を具体的な例として取りあげましょう。1つのシステムに 1 つの物理 CPU があるとします（現在は 2 つまたは 4 つ以上の場合があります）。仮想化は、単一の CPU を使用して、システム上で実行されているアプリケーションに対して多くの仮想 CPU のように見せかけるものです。したがって、各アプリケーションは使用する独自の CPU を持っていると考えていますが、実際には 1 つしかありません。したがって、OS は美しい錯覚を作り出しました。これが、CPU を仮想化です。

学生：うわー！それは魔法のように聞こえますね。もっと教えてください！それはどのように機能するのですか？

教授：そのような言葉が出てくるということは準備ができているようですね。

学生：もちろんです！私は教授を認めなければならないようですね…ただちょっと心配しているのは、教授が再び桃のことについて話を始めることなんですが…。

教授：心配しないでください。私は桃が好きではないので。それでは、始めましょう！

4. The Abstraction: The Process

この章では、OS がユーザに提供するもっとも基本的な抽象概念の 1 つ、すなわちプロセスについて説明します。プロセスの定義は、非公式には非常に単純です。実行中のプログラム [V+65, BH70] です。プログラム自体は特に意味のないものです。ディスク上に、一連の命令（例えば、いくつかの静的なデータ）があります。これらのバイトを取り込んで実行させるのは、オペレーティングシステムであり、プログラムを何か役に立つものに変えるものです。

一度に複数のプログラムを実行したいと思うことがあります。たとえば、Web ブラウザー、メールプログラム、ゲーム、音楽プレーヤーなどを実行したいデスクトップパソコンやノートパソコンを考えてみましょう。実際、典型的なシステムは、見かけ上、数十から数百のプロセスを同時に実行している可能性があります。そうすることで、システムが使いやすくなります。CPU が利用可能かどうか心配する必要は決してありません。単にプログラムを実行するだけです。

問題：

どのように多くの CPU があるように見せるのでしょうか？ 利用可能な物理 CPU はほんのわずかですが、OS はどのようにしてこの CPU をほぼ無限に供給しているような錯覚をさせるのでしょうか？

OS は CPU を仮想化してこの錯覚を作り出します。1 つのプロセスを実行し、それを停止し、別のプロセスを実行するなど、実際には 1 つの物理 CPU しか存在しない場合でも、多くの仮想 CPU が存在するという錯覚をつくりだすことがあります。CPU の time sharing と呼ばれるこの基本的な技術は、ユーザーが好きなだけ多くの並行プロセスを実行できるようにします。しかし、並行処理はパフォーマンスと trade off です。CPU を共有する必要がある場合は、それぞれがゆっくりと実行されるため、パフォーマンスが低下します。

CPU の仮想化を実装し、それをうまく実装するためには、低レベルのマシンといいくつかの高レベルのインテリジェンスが必要です。私たちは、低レベルの機構メカニズム (low-level machinery mechanisms) と呼んでいます。メカニズムとは、必要な部分を実装する低レベルのメソッドまたはプロトコルです。たとえば、コンテキストスイッチを実装する方法を後で学習します。これにより、OS にあるプログラムの実行を停止し、特定の CPU 上で別のプログラムを実行できるようになります。この time sharing mechanism はすべての最新の OS で採用されています。

これらのメカニズムの上には、OS のインテリジェンスの一部がポリシーの形で存在します。ポリシーとは、OS 内で何らかの決定を下すためのアルゴリズムです。たとえば、CPU 上で実行可能なプログラムの複数があれば、どのプログラムを OS で実行する必要がありますか？ OS のスケジューリング方針は、どのプログラムが実行されているかなどの仕事量に関する情報や、パフォーマンス指標など、履歴情報を使用してこの決定を行います。対話型パフォーマンスを最適化するシステム、またはスループット) を決定することができます。

TIP: USE TIME SHARING (AND SPACE SHARING)

Time sharing は、OS がリソースを共有するために使用する基本的な手法です。リソースをある法則に従って少しずつ、次に少しずつ使用することを許可することにより、問題のリソース（例えば、CPU またはネットワーククリンク）を多くの人が共有することができます。time sharing の対応は、リソースを使用したい人の間で（空間内で）分け合う空間共有です。たとえば、ディスクスペースは当然スペース共有されたリソースです。ディスクスペースの領域のブロックがファイルに割り当てられると、ユーザーが元のファイルを削除するまで、通常は別のファイルに割り当てられません。

4.1 The Abstraction: A Process

実行中のプログラムの OS によって提供される抽象化は、プロセスと呼ばれるものです。上記のように、プロセスは実行中のプログラムのことです。どの瞬間においても、実行中にアクセスまたは影響を与えるシステムのさまざまな部分のインベントリを取ることによって、プロセスを要約することができます。

どのようなプロセスを構成するのかを理解するには、マシンの状態 (machine state) を理解する必要があります。実行中にプログラムが読み書きできるものがあるのかを知る必要があります。

プロセスを構成するマシン状態の 1 つのとしてわかりやすいものは、メモリです。命令はメモリに置かれます。実行中のプログラムが読み書きするデータもメモリに格納されます。したがって、プロセスがアドレス指定できるメモリ (アドレス空間と呼ばれる) はプロセスの一部です。

また、プロセスのマシン状態の一部はレジスタです。多くの命令はレジスタを明示的に読み込んだり更新したりするので、プロセスの実行にとって重要です。

このマシン状態の一部を形成する特別なレジスタがいくつかあることに注意してください。例えば、プログラムカウンタ (PC)(命令ポインタまたは IP と呼ばれることがある) は、プログラムのどの命令が現在実行されているかを示します。同様に、スタックポインタと関連するフレームポインタを使用して、関数パラメータ、ローカル変数、およびリターンアドレスのスタックを管理します。

最後に、プログラムはよくストレージデバイスにもアクセスします。そのような I/O 情報は、プロセスが現在開いているファイルのリストも含みます。

TIP: SEPARATE POLICY AND MECHANISM

多くのオペレーティングシステムで共通の設計は、低レベルのメカニズム [L+75] から高レベルのポリシーを分離することです。メカニズムは、システムに関する質問への答えを提供するものと考えることができます。たとえば、オペレーティングシステムはコンテキストスイッチをどのように実行するでしょうか？

ポリシーは質問に対する回答を提供します。たとえば、オペレーティングシステムは現在どのプロセスを実行する必要があるでしょうか？これらの 2 つを分離することで、メカニズムを再考する必要ななしにポリシーを簡単に変更できるため、一般的なソフトウェア設計の原則であるモジュール性の一種です。

4.2 Process API

次の章まで実際のプロセス API の議論を延期しますが、ここではまずオペレーティングシステムのどのインターフェースに何が含まれなければならないかについていくつかの考えを示します。これらの API は、何らかの形で最新のオペレーティングシステムで使用できます。

- Create : オペレーティングシステムには、新しいプロセスを作成するためのメソッドが含まれている必要があります。シェルにコマンドを入力するか、アプリケーションアイコンをダブルクリックすると、OS が呼び出されて、指定したプログラムを実行するための新しいプロセスが作成されます。
- Destroy : プロセス作成用のインターフェースがあるため、システムはプロセスを強制的に破棄するインターフェースも提供します。もちろん、多くのプロセスが実行され、完了したばかりのプロセスは終了します。しかしながら、ユーザがそれらを止めることをしたいかもしれないので、暴走プロセスを止めるためのインターフェースは非常に有用です。
- Wait : プロセスが実行を停止するのを待つことが有用な場合があります。したがって、何らかの種類の待機インターフェースが提供されることが多いです。
- Miscellaneous Control : プロセスを強制終了または待機する以外に、可能な他のコントロールがあるこ

とがあります。たとえば、ほとんどのオペレーティングシステムでは、プロセスを中断させて（しばらく実行しないようにする）何らかの方法を提供してから、プロセスを再開します（実行し続けます）。

- Status : 通常、プロセスの実行時間や状態など、プロセスに関するステータス情報を得るためのインターフェースがあります。

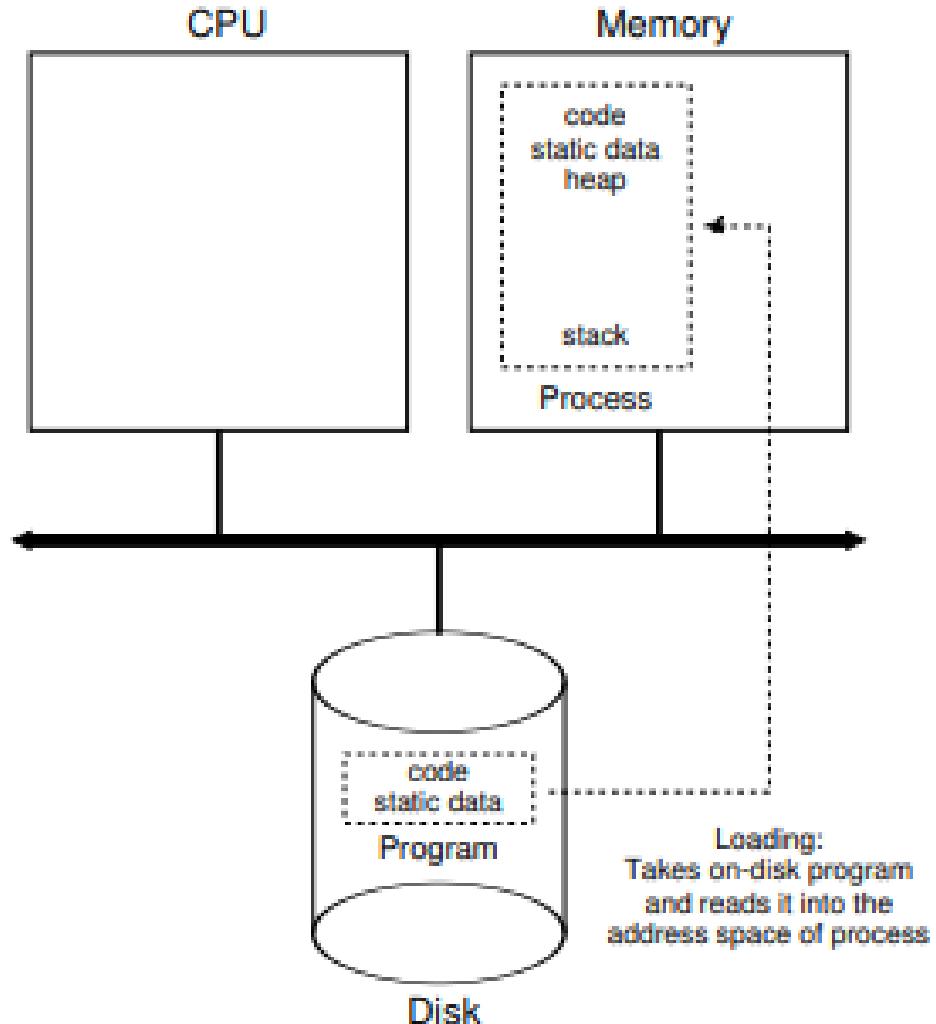


Figure 4.1: Loading: From Program To Process

4.3 Process Creation: A Little More Detail

私たちが少し疑問に思うことは、プログラムがプロセスにどのように変換されるかということです。具体的には、OSはどのようにプログラムを起動して実行しているのでしょうか？ プロセスの作成は実際にどのように機能するのでしょうか？

プログラムを実行するためにOSが最初にしなければならないことは、そのコードおよび任意の静的データ（例えば、初期化された変数）をメモリにプロセスのアドレス空間にロードすることです。プログラムは、ある種の実行可能形式で、ディスク（または現代のシステムでは、フラッシュベースのSSD）に最初は常駐しています。したがって、プログラムと静的データをメモリにロードするプロセスでは、OSがディスクからこれら

のバイトを読み込み、どこかのメモリに配置する必要があります(図 4.1 を参照)。

初期の(または単純な)オペレーティングシステムでは、ローディングプロセスは一回で完了させます。つまり、プログラムを実行する前に一度ロードをすべて終わらせます。現代の OS は、遅延的です。つまり、プログラムの実行中に必要とされるときにのみコードまたはデータの断片をロードすることによって、少しずつロードさせてプロセスを実行させます。コードやデータの遅延的な読み込みがどのように機能するかを本当に理解するには、ページングとスワッピングのメカニズムについてもっと理解しておく必要があります。今後はメモリの仮想化について議論します。今のところ、何かを実行する前に、重要なプログラムビットをディスクからメモリに取り込むためには、OS が何らかの作業を行う必要があることを覚えておいてください。

コードと静的データがメモリにロードされると、プロセスを実行する前に OS が行う必要のあることがあります。いくつかのメモリは、プログラムの runtime stack(または単に stack) に割り当てなければなりません。既に知っているはずのように、C プログラムはローカル変数、関数のパラメータ、および戻りアドレスにスタックを使用します。OS はこのメモリを割り当て、プロセスに渡します。また、OS は引数でスタックを初期化します:具体的には、`main()` 関数の引数、すなわち `argc` と `argv` 配列に値を代入します。

OS は、プログラムのヒープ用にいくつかのメモリを割り当てることもできます。C プログラムでは、明示的に要求された動的に割り当てられたデータにヒープが使用されます。プログラムは `malloc()` を呼び出して heap 領域を要求し、`free()` を呼び出して明示的に heap 領域を解放します。ヒープは、リンクリスト、ハッシュテーブル、ツリー、その他のデータ構造に必要です。最初のヒープ領域は小さいです。より多くのメモリを割り当てるときは、最初にプログラムが実行されるとき、また、`malloc()` ライブライ API を介してより多くのメモリを要求するときです。このとき、heap 領域を大きくするために OS が関与し、より多くのメモリをプロセスに割り当てるかもしれません。

OS は、特に入力/出力(I/O)に関連して、いくつかの他の初期化タスクも行います。たとえば、UNIX システムでは、各プロセスは標準で入力、出力、およびエラーの 3 つの open file descriptor をデフォルトで持っています。これらのディスクリプタは、プログラムが端末からの入力を容易に読み取ることができるとともに、出力を画面に出力することを可能にします。永続性(persistence)に関する本の第 3 部では、I/O やファイルディスクリプタなどについて学びます。

メモリにコードとスタティックデータをロードすることで、スタックを作成して初期化し、I/O 設定に関連する他の作業を行うことで、OS はプログラム実行のステージを(最終的に)設定します。したがって、エントリポイント(プログラムを起動するところ)、最後に `main()` ルーチン(次の章で説明する特殊なメカニズムによって)にジャンプすることによって、CPU は CPU の制御を移します。そこで、新しく作成されたプロセスに渡され、プログラムが実行を開始します。

4.4 Process States

プロセスが何であるかについていくつかのアイデアがあるので(私たちはこの概念を改良し続けるつもりですが)、プロセスがどのように作成されるかについて、ある時点でプロセスができるさまざまな状態についてお話ししましょう。プロセスがこれらの状態の 1 つになるという考えは、初期のコンピュータシステム [DV66、V+65] で生まれました。単純化されたビューでは、プロセスは次の 3 つの状態のいずれかになります。

- Running: 実行中の状態では、プロセスがプロセッサ上で実行されています。これは命令を実行していることを意味します。
- Ready: レディ状態では、プロセスは実行準備が整っていますが、何らかの理由で OS がこの瞬間にプロセスを実行しないことを選択しました。
- Blocked: ブロックされた状態では、プロセスが何らかの操作を実行したため、他のイベントが発生するまで実行できません。一般的な例: プロセスがディスクへの I/O 要求を開始すると、プロセスがブロックされるため、他のプロセスでプロセッサを使用することができます。

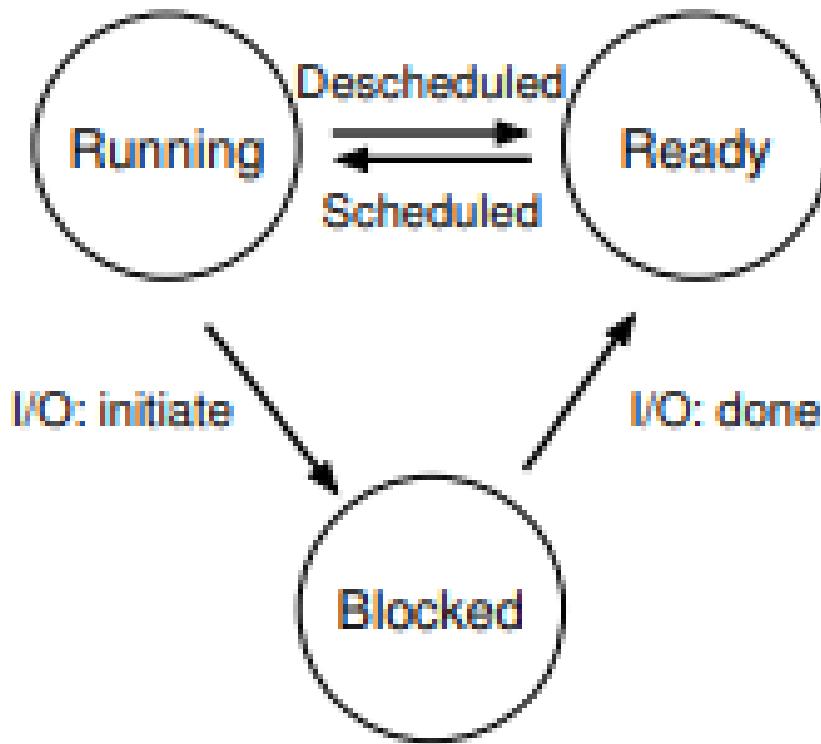


Figure 4.2: Process: State Transitions

これらの状態をグラフにマッピングすると、図 4.2 の図になります。この図でわかるように、プロセスは、OS の裁量で、実行可能状態と実行状態の間で移動できます。準備完了から実行中に移行することは、プロセスがスケジュールされていることを意味します。実行中から準備完了に移行すると、そのプロセスはスケジュールされていません。プロセスがブロックされると（例えば、I/O 動作を開始することによって）、OS は何らかのイベント（例えば、I/O 完了）が発生するまでその状態を維持します。その時点で、プロセスは再び準備完了状態に移行します（OS が決定した場合は、すぐに再実行する可能性があります）

2つのプロセスがこれらの状態のいくつかをどのように移行するかの例を見てみましょう。まず、2つのプロセスが稼働しているとします。それぞれのプロセスは CPU を使用しています（I/O はありません）。この場合、各プロセスの状態のトレースは、このようになります（図 4.3）。

Time	Process ₀	Process ₁	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process ₀ now done
5	—	Running	
6	—	Running	
7	—	Running	
8	—	Running	Process ₁ now done

Figure 4.3: Tracing Process State: CPU Only

この次の例では、最初のプロセスが実行した後に I/O を発行します。その時点で、プロセスはブロックされ、他のプロセスに実行の機会が与えられます。図 4.4 に、このシナリオのトレースを示します。

具体的には、Process0 は I/O を開始し、完了するのを待ってブロックされます。ディスクからの読み取りやネットワークからのパケットの待機など、プロセスはブロックされます。OS は Process0 が CPU を使用していないことを認識し、Process1 の実行を開始します。Process1 が実行されている間、I/O は完了し、Process0 を準備完了に戻します。最後に、Process1 が終了し、Process0 が実行されて終了します。

Time	Process ₀	Process ₁	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process ₀ initiates I/O
4	Blocked	Running	Process ₀ is blocked,
5	Blocked	Running	so Process ₁ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process ₁ now done
9	Running	—	
10	Running	—	Process ₀ now done

Figure 4.4: Tracing Process State: CPU and I/O

この簡単な例でも、OS が決定しなければならないことが、たくさんあることに注意してください。まず、Process0 が I/O を発行している間に、システムは Process1 を実行することを決定しなければなりませんでし

た。そうすることで、CPU をビジーに保つことによってリソースの使用率が向上します。第 2 に、システムは I/O が完了したときに Process0 に戻らないことにしました。これが良い判断かどうかは不明です。みなさんはどう思いますか？ これらのタイプの決定は、OS スケジューラによって行われます。これについては、今後いくつかの章で説明します。

4.5 Data Structures

OS はプログラムであり、どのプログラムでも、さまざまな関連情報を追跡するいくつかの重要なデータ構造を持っています。たとえば、各プロセスの状態を追跡するために、OS は、準備が整っているすべてのプロセスのプロセスリストと、現在実行中のプロセスを追跡するための追加情報を保持している可能性があります。OS はブロックされたプロセスを何らかの形で追跡しなければなりません。I/O イベントが完了したら、OS は正しいプロセスを起動して、再度実行する準備をしておく必要があります。

図 4.5 は、OS が xv6 カーネル [CK+08] の各プロセスについてどのようなタイプの情報を追跡する必要があるかを示しています。Linux、Mac OS X、Windows などの「実際の」オペレーティングシステムにも同様のプロセス構造が存在します。それらを見てどのくらい複雑であるかを見てください。

この図から、OS がプロセスについて追跡している重要な情報をいくつか確認できます。レジスタコンテキストは、停止されたプロセスに対して、そのレジスタの内容を保持する。プロセスが停止すると、そのレジスタはこのメモリ位置に保存されます。これらのレジスタを復元する（すなわち、それらの値を実際の物理レジスタに戻す）ことによって、OS はプロセスの実行を再開することができます。コンテキストスイッチと呼ばれるこの手法については、今後の章で詳しく説明します。

```

// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
    int eip;
    int esp;
    int ebx;
    int ecx;
    int edx;
    int esi;
    int edi;
    int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
    char *mem;                      // Start of process memory
    uint sz;                        // Size of process memory
    char *kstack;                   // Bottom of kernel stack
                                    // for this process
    enum proc_state state;          // Process state
    int pid;                        // Process ID
    struct proc *parent;            // Parent process
    void *chan;                     // If non-zero, sleeping on chan
    int killed;                     // If non-zero, have been killed
    struct file *ofile[NOFILE];    // Open files
    struct inode *cwd;              // Current directory
    struct context context;         // Switch here to run process
    struct trapframe *tf;           // Trap frame for the
                                    // current interrupt
};

```

Figure 4.5: The xv6 Proc Structure

図から、プロセスが実行可能、準備完了、ブロックされている以外の状態がいくつかあることもわかります。場合によっては、プロセスが作成されているときにシステムが初期状態になることがあります。また、プロセスは終了したが、まだクリーンアップされていない最終状態に置くことができます（UNIXベースのシステムでは、ゾンビ状態1と呼ばれます）。この最終状態は、他のプロセス（通常はプロセスを作成した親プロセス）がプロセスの戻りコードを調べ、正常終了したプロセスが正常に実行されたかどうかを確認するのに便利です（UNIXベースシステムでは、彼らは正常にタスクを達成しており、それ以外の場合は0ではない）。終了すると、親は、子の完了を待つために1回の最終コール（例えば、`wait()`）を行い、現在絶滅しているプロセスを参照する関連するデータ構造をクリーンアップできることをOSに示します。

ASIDE: DATA STRUCTURE — THE PROCESS LIST

オペレーティングシステムには、さまざまな重要なデータ構造があります。プロセスリストは、重要なデータ構造の一つです。これは単純なもののが一つですが、確かに複数のプログラムを同時に

実行する能力を持つ OS は、システム内のすべての実行中のプログラムを追跡するために、この構造に似たものを持っています。時には、プロセスに関する情報を格納する個々の構造体を Process Control Block(PCB) と呼ぶこともあります。Process Control Block(PCB) は、各プロセスに関する情報を含む C 構造体に関する素晴らしい方法です。

4.6 Summary

我々は OS の最も基本的な抽象化を紹介した。プロセスというのは、それは非常に単純に実行中のプログラムとみなされることです。この概念を念頭に置いて、プロセスを実装するために必要な低レベルのメカニズムと、インテリジェントな方法でそれらをスケジューリングするために必要とされるより高いレベルのポリシーを扱います。メカニズムとポリシーを組み合わせることによって、オペレーティングシステムが CPU をどのように仮想化するかを理解していきます。

参考文献

[BH70] “The Nucleus of a Multiprogramming System”

Per Brinch Hansen

Communications of the ACM, Volume 13, Number 4, April 1970

This paper introduces one of the first microkernels in operating systems history, called Nucleus. The idea of smaller, more minimal systems is a theme that rears its head repeatedly in OS history; it all began with Brinch Hansen’s work described herein.

[CK+08] “The xv6 Operating System”

Russ Cox, Frans Kaashoek, Robert Morris, Nickolai Zeldovich

From: <https://github.com/mit-pdos/xv6-public>

The coolest real and little OS in the world. Download and play with it to learn more about the details of how operating systems actually work. We have been using an older version (2012-01-30-1-g1c41342) and hence some examples in the book may not match the latest in the source.

[DV66] “Programming Semantics for Multiprogrammed Computations”

Jack B. Dennis and Earl C. Van Horn

Communications of the ACM, Volume 9, Number 3, March 1966

This paper defined many of the early terms and concepts around building multiprogrammed systems.

[L+75] “Policy/mechanism separation in Hydra”

R. Levin, E. Cohen, W. Corwin, F. Pollack, W. Wulf

SOSP 1975

An early paper about how to structure operating systems in a research OS known as Hydra. While Hydra never became a mainstream OS, some of its ideas influenced OS designers.

[V+65] “Structure of the Multics Supervisor”

V.A. Vyssotsky, F. J. Corbato, R. M. Graham

Fall Joint Computer Conference, 1965

An early paper on Multics, which described many of the basic ideas and terms that we find in modern systems. Some of the vision behind computing as a utility are finally being realized in modern cloud systems.

5. 間奏：プロセス API

ASIDE : INTERLUDES

Interludes では、オペレーティングシステム API に重点を置いて、それらを使用する方法など、システムのより実用的な側面について説明します。実用的なことが気に入らなければ、これらの間奏をスキップすることができます。しかし実用的なものが好きなのは、実は実生活では一般的に便利だからです。例えば、企業は、通常、あなたの非実用的なスキルのためにあなたを雇うことはありません。

この中で、UNIX システムでのプロセス作成について説明します。UNIX は、`fork()` と `exec()` の 2 つのシステムコールを使用して、新しいプロセスを作成する最も興味深い方法の 1 つを提供します。3 番目のルーチン `wait()` は、作成したプロセスが完了するのを待つプロセスによって使用できます。ここでは、これらのインターフェースをいくつかの簡単な例を挙げてより詳細に提示し、私たちの動機づけを示します。

CRUX : プロセスの作成と制御方法

プロセスの作成と制御のために OS が提示すべきインターフェースは何ですか？ これらのインターフェイスは、使いやすさとユーティリティ性を実現するためにどのように設計されるべきですか？

5.1 システムコール `fork()`

`fork()` システムコールは新しいプロセス [C63] の作成に使用されます。具体的には、図 5.1 に示すようなコードを持つ実行プログラムがあります。コードを調べるか、それを入力して実行してください。

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int
6  main(int argc, char *argv[])
7  {
8      printf("hello world (pid:%d)\n", (int) getpid());
9      int rc = fork();
10     if (rc < 0) {           // fork failed; exit
11         fprintf(stderr, "fork failed\n");
12         exit(1);
13     } else if (rc == 0) {    // child (new process)
14         printf("hello, I am child (pid:%d)\n", (int) getpid());
15     } else {                // parent goes down this path (main)
16         printf("hello, I am parent of %d (pid:%d)\n",
17                 rc, (int) getpid());
18     }
19     return 0;
20 }
```

Figure 5.1: Calling `fork()` (`p1.c`)

When you run this program (called `p1.c`), you'll see the following:

```

prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

p1.c. で何が起きたのかもっと詳しく知りましょう。最初に実行を開始すると、プロセスは hello world メッセージを出力します。そのメッセージにはプロセス識別子 (PID とも呼ばれます) が含まれています。このプロセスの PID は 29146 です。UNIX システムでは、PID は、(例えば) 実行を停止するなど、プロセスで何かをしたい場合にプロセスの名前を付けるために使用されます。

ここで面白い部分が始まります。このプロセスは、OS が提供する `fork()` システムコールを呼び出して、新しいプロセスを作成します。作成されるプロセスは、呼び出しプロセスの (ほぼ) 正確なコピーです。つまり、OS には、実行中のプログラム p1 のコピーが 2 つあり、両方とも `fork()` システムコールから復帰しようとしているようです。新たに作成されたプロセス (親とは対照的に、子と呼ばれる) は `main()` で動作を開始しません。(「hello、world」メッセージは一度だけ出力されます。むしろ、`fork()` 自体を呼び出したかのように、開始します。

気づいたかもしれません。子供は正確なコピーではありません。具体的には、アドレス空間 (すなわち、自身のプライベートメモリ) のコピー、自身のレジスタ、それ自身の PC などを持っていますが、`fork()` の呼び出し元に返す値は異なります。具体的には、親が新しく作成された子の PID を受け取っている間、子はゼロの戻りコードを受け取ります。この区別は、2 つの異なるケースを扱うコードを書くのが簡単なので (上記と同様に) 便利です。

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5
6 int
7 main(int argc, char *argv[])
8 {
9     printf("hello world (pid:%d)\n", (int) getpid());
10    int rc = fork();
11    if (rc < 0) {           // fork failed; exit
12        fprintf(stderr, "fork failed\n");
13        exit(1);
14    } else if (rc == 0) {   // child (new process)
15        printf("hello, I am child (pid:%d)\n", (int) getpid());
16    } else {                // parent goes down this path (main)
17        int wc = wait(NULL);
18        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
19               rc, wc, (int) getpid());
20    }
21    return 0;
22 }
```

Figure 5.2: Calling `fork()` And `wait()` (p2.c)

また気づいたかもしれません。出力 (p1.c の) は決定的ではありません。子プロセスが作成されると、システムには親プロセスと子プロセスの 2 つのアクティブなプロセスがあります。単一の CPU を持つシステムで実行していると仮定すると (単純化のため)、その時点で子プロセスまたは親プロセスが実行される可能性があります。上の例では、親がメッセージを最初に出力しました。それ以外の場合は、この出力トレースに示すように、逆の場合があります。

```

prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

CPU スケジューラは、すぐに詳細に議論するトピックで、特定の瞬間に実行されるプロセスを決定します。スケジューラは複雑であるため、通常は何を選択するのか、したがってどのプロセスが最初に実行されるのかについて前提はできません。この非決定性は、特にマルチスレッドプログラムでは、いくつかの興味深い

問題につながります。したがって、本書の第2部で並行性を学ぶと、非決定性がさらに増えます。

5.2 システムコール `wait()`

これまでのところ、メッセージを出力して終了する子供を作成しました。時には、親プロセスが子プロセスの処理を完了するのを待つことは非常に便利です。このタスクは、`wait()` システムコール（またはより完全な兄弟 `waitpid()`）で実行されます。詳細は図 5.2 を参照してください。

この例 (`p2.c`) では、親プロセスは `wait()` を呼び出して、子プロセスの実行が終了するまでその実行を遅らせます。子が終了すると、`wait()` は親に戻ります。

上記のコードに `wait()` を追加すると、出力が決定的になります。

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
prompt>
```

このコードでは、子が常に最初に表示されることがわかりました。なぜ我々はそれを知っていますか？ 前と同じように最初に走って、親の前にプリントするかもしれません。しかし、親が最初に実行されると、すぐに `wait()` が呼び出されます。このシステムコールは、子が実行されて終了するまで返されません。したがって、親が最初に実行されても、子が実行を終了するのを丁寧に待ってから `wait()` が戻り、親がそのメッセージを表示します。

5.3 最後、システムコール `exec()`

プロセス作成 API の最終的かつ重要な部分は、`exec()` システムコールです。このシステムコールは、呼び出し元のプログラムとは異なるプログラムを実行する場合に便利です。たとえば、`p2.c` の `fork()` の呼び出しは、同じプログラムのコピーを実行し続ける場合にのみ便利です。しかし、多くの場合、別のプログラムを実行する必要があります。`exec()` はそれだけを行います（図 5.3）。

この例では、子プロセスは `execvp()` を呼び出して、ワードカウントプログラムであるプログラム `wc` を実行します。実際には、ソースファイル `p3.c` 上で `wc` を実行するので、ファイル内にいくつの行、単語、およびバイトがあるかがわかります。

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
    29      107     1030 p3.c
hello, I am parent of 29384 (wc:29384) (pid:29383)
prompt>
```

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <sys/wait.h>
6
7  int
8  main(int argc, char *argv[])
9  {
10     printf("hello world (pid:%d)\n", (int) getpid());
11     int rc = fork();
12     if (rc < 0) {           // fork failed; exit
13         fprintf(stderr, "fork failed\n");
14         exit(1);
15     } else if (rc == 0) {    // child (new process)
16         printf("hello, I am child (pid:%d)\n", (int) getpid());
17         char *myargs[3];
18         myargs[0] = strdup("wc");    // program: "wc" (word count)
19         myargs[1] = strdup("p3.c");  // argument: file to count
20         myargs[2] = NULL;           // marks end of array
21         execvp(myargs[0], myargs); // runs word count
22         printf("this shouldn't print out");
23     } else {                // parent goes down this path (main)
24         int wc = wait(NULL);
25         printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
26                 rc, wc, (int) getpid());
27     }
28 }
29 }
```

Figure 5.3: Calling `fork()`, `wait()`, And `exec()` (`p3.c`)

`fork()` システムコールは異常です。犯罪のパートナーである `exec()` は普通ではありません。実行可能ファイルの名前 (`wc` など) といいくつかの引数 (`p3.c` など) を指定すると、実行可能ファイルからコード (および静的データ) をロードし、現在のコードセグメント (および現在の静的データ) を上書きします。プログラムのメモリ空間のヒープおよびスタックおよび他の部分が再初期化されます。その後、OS は単にそのプログラムを実行し、そのプロセスの `argv` として引数を渡します。したがって、新しいプロセスは作成されません。むしろ、現在実行中のプログラム (以前は `p3`) を別の実行中のプログラム (`wc`) に変換します。子の中で `exec()` の後では、`p3.c` が決して走っていないかのようです。`exec()` への呼び出しが成功すると、決して戻りません。

5.4 どうして？ API の動機付け

もちろん、あなたが持っているかもしれない 1 つの大きな疑問：なぜ新しいプロセスを作成する単純な行為でなければならないものに対して、このような奇妙なインターフェースを構築するのでしょうか？ `fork()` と `exec()` の分離は UNIX シェルを構築する上で不可欠です。なぜなら、シェルが `fork()` の呼び出しの後、`exec()` の呼び出しの前にコードを実行できるからです。このコードは実行しようとしているプログラムの環境を変更することができ、したがって様々な興味深い機能を容易に構築することができます。

ヒント：それを得る（ラムローンの法律）

Lampson 氏が彼の著名な「コンピュータシステム設計のヒント」[L83] で述べているように、「それを正しくしてください。抽象化もシンプルさも、それを正しくするための代替手段ではありません」。時には、正しいことをしなければならない時もあります。プロセス作成のための API を設計する方法はたくさんあります。しかし、`fork()` と `exec()` の組み合わせは単純で非常に強力です。ここでは、UNIX デザイナーは単にそれを正しく理解しました。そして Lampson はしばしば「正しい」と言いましたので、私たちはその名誉の中で法律を名づけました。

シェルは単なるユーザー プログラムです。それはプロンプトを表示し、何かを入力するのを待ちます。次

に、コマンド(実行可能プログラムの名前に加えて任意の引数)を入力します。ほとんどの場合、シェルはファイルシステム内で実行可能ファイルがどこにあるのかを確認し、`fork()`を呼び出してコマンドを実行する新しい子プロセスを作成し、`exec()`のいくつかの変種を呼び出してコマンドを実行します。`Wait()`を呼び出してコマンドを実行することで完了します。子が完了すると、シェルは`wait()`から戻り、プロンプトをもう一度出力して、次のコマンドの準備をします。

`fork()`と`exec()`を分離することで、シェルは便利なものを簡単に扱うことができます。

```
prompt> wc p3.c > newfile.txt
```

上記の例では、プログラム`wc`の出力が output ファイル`newfile.txt`にリダイレクトされます(より大きい記号はリダイレクトがどのように示されているかです)。シェルがこのタスクを達成する方法は非常に簡単です。子が作成されるとき、`exec()`を呼び出す前に、シェルは標準出力を閉じてファイル`newfile.txt`を開きます。これにより、すぐに実行されるプログラム`wc`からの出力は、画面ではなくファイルに送信されます。

図 5.4 は、これを正確に実行するプログラムを示しています。このリダイレクションが機能する理由は、オペレーティングシステムがファイル記述子をどのように管理するかという仮定によるものです。具体的には、UNIX システムはゼロでフリーファイル記述子を探し始めます。この場合、`STDOUT_FILENO`が最初に使用可能になり、したがって`open()`が呼び出されるときに割り当てられます。その後、子プロセスによる`printf()`などのルーチンによる標準出力ファイル記述子への後続の書き込みは、画面ではなく新しく開いたファイルに透過的にルーティングされます。

```
prompt> ./p4
prompt> cat p4.output
      32      109      846 p4.c
prompt>
```

次に、p4.c プログラムを実行した結果を示します。

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <fcntl.h>
6 #include <sys/wait.h>
7
8 int
9 main(int argc, char *argv[])
10 {
11     int rc = fork();
12     if (rc < 0) {           // fork failed; exit
13         fprintf(stderr, "fork failed\n");
14         exit(1);
15     } else if (rc == 0) {    // child: redirect standard output to a file
16         close(STDOUT_FILENO);
17         open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
18
19         // now exec "wc"...
20         char *myargs[3];
21         myargs[0] = strdup("wc");    // program: "wc" (word count)
22         myargs[1] = strdup("p4.c"); // argument: file to count
23         myargs[2] = NULL;          // marks end of array
24         execvp(myargs[0], myargs); // runs word count
25     } else {                // parent goes down this path (main)
26         int wc = wait(NULL);
27     }
28     return 0;
29 }
```

Figure 5.4: All Of The Above With Redirection (p4.c)

あなたは、この出力に関して(少なくとも)2つの面白い小物を気づくでしょう。まず、p4 を実行すると、何も起こっていないかのように見えます。シェルはコマンドプロンプトを表示するだけで、すぐに次のコマン

ドの準備ができます。しかし、そうではありません。プログラム p4 は実際に `fork()` を呼び出して新しい子を作成し、`execvp()` の呼び出しによって `wc` プログラムを実行しました。出力がファイル `p4.output` にリダイレクトされたため、画面に出力が表示されません。次に、出力ファイルをキャッチすると、`wc` を実行したときに期待されるすべての出力が見つかることがわかります。

UNIX パイプも同様の方法で実装されますが、`pipe()` システムコールが実装されています。この場合、1つのプロセスの出力はカーネル・パイプ(すなわち、キュー)に接続され、別のプロセスの入力は同じパイプに接続されます。したがって、1つのプロセスの出力がシームレスに次の入力に使用され、長くて有効なコマンドのチェーンを一緒につなげることができます。簡単な例として、ファイル内の単語を探し、その単語が何回出現したかを数えます。パイプとユーティリティは `grep` と `wc` で簡単です。`grep -o foo file | wc -l` とプロンプトに入力します。

最後に、プロセス API を高レベルでスケッチしただけですが、これらの呼び出しが学習され、消化されることについての詳細がはるかに多くあります。本書の第3部でファイルシステムについて話すときに、たとえばファイル記述子についてもっと学びます。今のところ、`fork()`/`exec()` の組み合わせは、プロセスを作成して操作する強力な方法であると言うだけで十分です。

ASIDE : RTFM - 人のページを読む この本では、特定のシステムコールやライブラリ呼び出しを参照するときに、マニュアルページやマニュアルページを読むことを何度も指示します。マニュアルページは、UNIX システム上に存在するオリジナルのドキュメント形式です。Web と呼ばれるものが存在する前にそれらが作成されたことに気付きます。

マニュアルページを読むのに時間を費やすことは、システムプログラマの成長の鍵となる一歩です。それらのページにはたくさんの便利な小技が隠されています。特に便利なページには、使用しているシェル (`tcsh`, `bash` など) のマニュアルページと、プログラムが作るシステムコール (戻り値とエラー条件が存在するかどうかを確認するためのもの) があります。

最後に、マニュアルページを読むことで、いくつかの恥ずかしさを軽減できます。`fork()` の複雑さについて同僚に尋ねると、彼らは単に「RTFM」と返答するかもしれません。これは、あなたの同僚の読書の手引きを静かに促す方法です。

5.5 API のその他の部分

`fork()`, `exec()`, `wait()` 以外にも、UNIX システムのプロセスとやりとりするための多くのインターフェースがあります。たとえば、`kill()` システムコールは、スリープ状態に陥るか、死ぬか、その他の有用な命令を含む、シグナルをプロセスに送信するために使用されます。実際、シグナルサブシステム全体は、シグナルを受信して処理する方法を含め、プロセスに外部イベントを提供する豊富なインフラストラクチャを提供します。

便利な多くのコマンドラインツールがあります。たとえば、`ps` コマンドを使用すると、実行中のプロセスを確認できます。`ps` に渡す有用なフラグのマニュアルページを読んでください。ツールのトップは、システムのプロセスや CPU や他のリソースがどれくらい消費しているかを表示するので、非常に役立ちます。最後に、システムの負荷をすばやく把握するために使用できるさまざまな種類の CPU メーターがあります。たとえば、私たちは Macintosh Toolbars 上で実行されている Raging Menace ソフトウェアの MenuMeters を常に保持しているので、いつどの CPU が使用されているかを見ることができます。一般的に、何が起こっているかについてのより多くの情報は、より良いものです。

5.6 要約

`fork()`、`exec()`、`wait()` のような、UNIX プロセスの作成を扱ういくつかの API を紹介しました。しかし、私たちは表面を見ただけです。詳細については、Stevens と Rago [SR05]、特にプロセス制御、プロセス関係、および信号に関する章を読んでください。そこには学ぶべきことがたくさんあります。

ASIDE：コーディングの原点コーディングの宿題は、現代のオペレーティングシステムが提供しなければならない基本的な API のいくつかの経験を得るために、実際のマシンで実行するコードを書く小さな練習です。結局のところ、(おそらく) あなたはコンピュータ科学者なので、コーディングするのは正しいでしょうか？もちろん、本当に専門家になるためには、少し時間をかけてマシンをハッキングする必要があります。実際には、いくつかのコードを書いて、それがどのように動作するかを見るためのあらゆる言い訳を見つけてください。時間を費やし、あなたができるることを知っている賢いマスターになりなさい。

宿題（コード）

この課題では、読んだばかりのプロセス管理 API に精通する必要があります。心配する必要はありません - それは聞こえるよりもさらに楽しいです！一般的には、コードを書くことができるくらい多くの時間を見つけた方がずっと良いでしょう。

問題

1. `fork()` を呼び出すプログラムを記述します。`fork()` を呼び出す前に、メインプロセスに変数 (たとえば `x`) にアクセスさせ、その値を何か (例えば 100) に設定させます。子プロセスの変数はどのような値ですか？子と親の両方が `x` の値を変更すると、変数には何が起こりますか？
2. ファイルをオープンする (`open()` システムコールで) プログラムを作成し、`fork()` を呼び出して新しいプロセスを作成します。子と親の両方が `open()` によって返されたファイル記述子にアクセスできますか？同時にファイルに書き込むとき、つまり同時に実行するとどうなりますか？
3. `fork()` を使用して別のプログラムを作成します。子プロセスは “hello” を出力する必要があります。親プロセスは “さようなら” を印刷する必要があります。子プロセスが常に最初に印刷されるようにする必要があります。あなたは親で `wait()` を呼び出さずにこれを行うことができますか？
4. `fork()` を呼び出し、`exec()` の何らかの形式を呼び出してプログラム / bin / ls を実行するプログラムを記述します。`exec1()`、`execle()`、`execlp()`、`execv()`、`execvp()`、および `execvP()` を含む `exec()` のすべてのバリエーションを試すことができるかどうかを確認してください。なぜ同じ基本的な呼び出しの多くの変種があると思いますか？
5. 子プロセスが親プロセスで終了するのを待つために `wait()` を使うプログラムを書いてください。`wait()` は何を返しますか？子で `wait()` を使用するとどうなりますか？
6. `Wait()` の代わりに `waitpid()` を使用して、前のプログラムのわずかな変更を書き込んでください。`waitpid()` はいつ有用でしょうか？
7. 子プロセスを作成し、子プロセスで標準出力 (STDOUP FILENO) を閉じるプログラムを記述します。子が記述子を閉じた後に `printf()` を呼び出して出力をプリントするとどうなりますか？
8. 2 つの子を作成し、`pipe()` システムコールを使用して、標準出力をもう一方の標準入力に接続するプログラムを記述します。

参考文献

[C63] “A Multiprocessor System Design”

Melvin E. Conway

AFIPS '63 Fall Joint Computer Conference

New York, USA 1963

An early paper on how to design multiprocessing systems; may be the first place the term fork() was used in the discussion of spawning new processes.

[DV66] “Programming Semantics for Multiprogrammed Computations”

Jack B. Dennis and Earl C. Van Horn

Communications of the ACM, Volume 9, Number 3, March 1966

A classic paper that outlines the basics of multiprogrammed computer systems. Undoubtedly had great influence on Project MAC, Multics, and eventually UNIX.

[L83] “Hints for Computer Systems Design”

Butler Lampson

ACM Operating Systems Review, 15:5, October 1983

Lampson's famous hints on how to design computer systems. You should read it at some point in your life, and probably at many points in your life.

[SR05] “Advanced Programming in the UNIX Environment”

W. Richard Stevens and Stephen A. Rago

Addison-Wesley, 2005

All nuances and subtleties of using UNIX APIs are found herein. Buy this book! Read it! And most importantly, live it.

6. Mechanism: Limited Direct Execution

CPU を仮想化するためには、オペレーティングシステムは、一見同時に実行している多くのジョブの中で何らかの形で物理 CPU を共有する必要があります。基本的なアイデアは簡単です。プロセスを 1 つ実行してから別のプロセスを実行するなどです。このように CPU を time slice で共有することにより、仮想化が実現されます。

しかし、そのような仮想化機構を構築するには、いくつかの課題があります。1 つはパフォーマンスです。システムに過度のオーバーヘッドを加えずに、どのように仮想化を実装できるでしょうか？もう 1 つは資源管理です。CPU を制御できなくなることなしにプロセスを効率的に実行するにはどうすればよいでしょうか？OS にとっては特に資源管理が重要です。というのも、OS は資源に関しての担当者だからです。資源管理がなければ、プロセスは単に永遠に実行するだけでマシンを乗っ取ることができてしまい、アクセスを許可したくない情報にアクセスすることもできてしまいます。したがって、制御を失わずに高いパフォーマンスを得ることは、オペレーティングシステムを構築する上での中心的な課題の 1 つです。

THE CRUX: HOW TO EFFICIENTLY VIRTUALIZE THE CPU WITH CONTROL(CPU をコントロールと効率的に仮想化する方法)

OS は、システムの制御を維持しながら効率的に CPU を仮想化する必要があります。そのためには、ハードウェアとオペレーティングシステムの両方のサポートが必要になります。OS は、その作業を効果的に達成するために、しばしば賢明なハードウェアサポートを使用します。(優れているハードウェアサポートであるほど効率が上がる。OS サポートも同様)

6.1 Basic Technique: Limited Direct Execution

プログラムが期待どおりの速さで動作するようにするために、OS 開発者は限定された direct execution と呼ばれる技術を思いついたわけではありません。アイデアの「direct executin」の部分は単純です。プログラムを CPU 上で直接実行するだけです。したがって、OS がプログラムの実行を開始したいときには、プロセスリストにプロセスエントリを作成し、プロセスリストにいくつかのメモリを割り当て、プログラムコードをメモリにロードし（ディスクから）、エントリポイントを見つけます（`main()` ルーチンやそれに似たもの）、ジャンプしてユーザーのコードの実行を開始します。図 6.1 は、この基本的な direct execution プロトコルを示しています（まだ制限はありません）。通常の呼び出しを使用して、戻り先である、プログラムの `main()` にジャンプし、後でカーネルに戻ります。

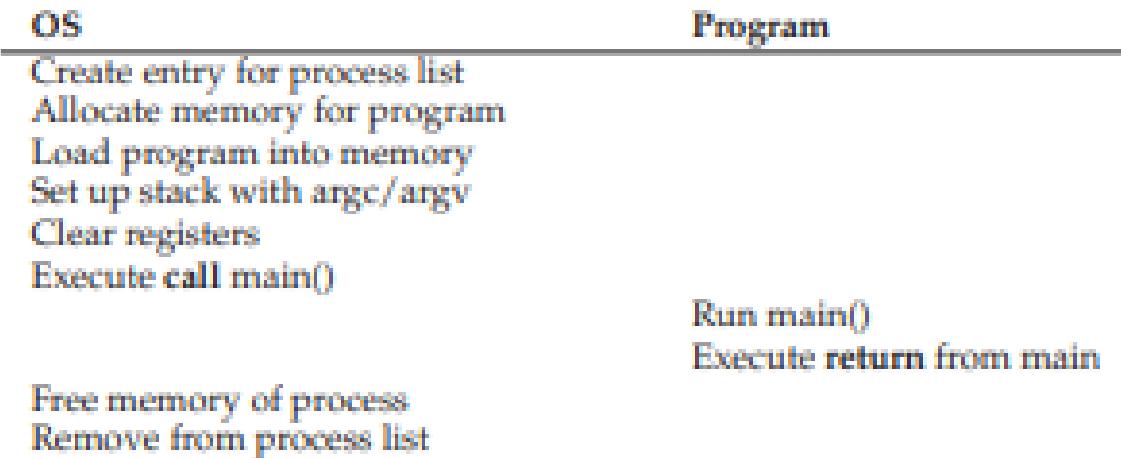


Figure 6.1: Direct Execution Protocol (Without Limits)

シンプルに聞こえるでしょう？しかし、このアプローチでは、CPU を仮想化するためのいくつかの問題が生じます。最初はの問題は簡単です。プログラムを実行するだけの場合、OS はプログラムを効率的に実行しながら、プログラムで実行したくないことを OS がどのように判断することができますか？2つ目は、プロセスを実行しているときに、オペレーティングシステムが実行を停止して別のプロセスに切り替える方法です。つまり、CPU を仮想化するために必要な time slice を実装しますか？

これらの質問に答えるにあたって、CPU を仮想化するために必要なことをもっとよく理解していきます。これらの技術を開発する際には、名前の「限定された」部分がどこから生じているのかもわかります。OS を実行しているプログラムに制限を加えることなく、OS は何の制御もされないまるで「単なるライブラリ」です。夢のあるオペレーティングシステム実現するための非常に悲しい現状の OS の状態です！

6.2 Problem #1: Restricted Operations

直接実行には高速という明白な利点があります。プログラムはハードウェア CPU 上でネイティブに実行されるため、期待どおりの速さで実行されます。しかし、CPU 上で実行すると、ある問題が生じてしまいます。例えば、ディスクに I/O 要求を発行したり、CPU やメモリなどのより多くのシステムリソースにアクセスするなど、何らかの制限された操作を実行したい場合はどうなるでしょうか？

THE CRUX: HOW TO PERFORM RESTRICTED OPERATIONS(制限付き操作を実行する方法)

プロセスは I/O などの制限付きの操作を実行できる必要がありますが、プロセスがシステム全体を完全に制御することはできません。OS とハードウェアはどのように連携して動作するのでしょうか？

ASIDE: WHY SYSTEM CALLS LOOK LIKE PROCEDURE CALLS(システムコールがプロシージャコールのように見える理由)

open() や read() のようなシステムコールへの呼び出しが C 言語の典型的な手続き呼び出しとまったく同じように見えるのはなぜかと思うでしょう。つまり、プロシージャコールのように見える場合、システムはシステムコールであるとはどのように判断できるでしょうか？実は、プロシージャコールですが、そのプロシージャコールの中に隠されているのは有名なトラップ命令なのです。

具体的には、`open()`(たとえば)を呼び出すと、C ライブラリにプロシージャコールを実行しています。そこでは、`open()` やその他のシステムコールのいずれにしても、ライブラリは、カーネルとの間で合意した呼び出し規約を使用して、引数を既知の場所(スタックや特定のレジスタ)、システムコール番号をよく知られた場所に(スタックまたはレジスタ上に)入れ、前述のトラップ命令を実行します。

トラップ後のライブラリのコードは、戻り値をアンパックし、システムコールを発行したプログラムに制御を返します。したがって、システムコールを実行する C ライブラリの部分は、引数を正しく処理して値を正しく返し、ハードウェア固有のトラップ命令を実行するために慣習に注意深く従う必要があるため、アセンブリで手作業でコーディングされます。では、なぜあなたが個人的に OS にトラップするアセンブリコードを書く必要がないのでしょうか？ それは、誰かがすでにあなたのためにそのアセンブリを書いてくれているからです。

1 つのアプローチは、I/O やその他の関連する操作の観点から、任意のプロセスに必要なものを実行させることです。しかしそうすることは、望ましい多くの種類のシステムの構築を妨げるでしょう。たとえば、ファイルへのアクセスを許可する前に、アクセス権をチェックするファイルシステムを構築するとしましょう。(アノニマスユーザの権限でアクセス権をチェックをするものを構築する) そうすると、ディスク全体の読み書きができなくなり、すべての保護が失われてしまいます。

したがって、我々が取るアプローチは、ユーザー mode として知られる新しいプロセッサモードを導入することです。ユーザー mode で実行されるコードは、できることで制限されています。たとえば、ユーザー mode で実行している場合、プロセスは I/O 要求を発行できません。そのようにすると、プロセッサは例外を発生させます。OS はプロセスを終了させる可能性があります。

ユーザー mode とは対照的に、オペレーティングシステム(またはカーネル)が実行されるカーネル mode は、I/O 要求の発行やすべてのタイプの制限付き命令の実行などの特権操作を含む実行可能なコードです。

ただし、ディスクからの読み取りなど、何らかの特権操作を実行する場合、ユーザーの処理はどうすればよいでしょうか？ これを可能にするために、ほぼすべての現代のハードウェアは、ユーザープログラムがシステムコールを実行する能力を提供します。Atlas [K + 61, L78] のような古代のマシンで開発されたシステムコールでは、カーネルは、ファイルシステムへのアクセス、プロセスの作成と破棄、他のプロセスとの通信など、特定の重要な機能をユーザープログラムに慎重に公開することができます。より多くのメモリを割り当てることができます。ほとんどのオペレーティングシステムは数百個の呼び出しを提供します(詳細は POSIX 標準を参照してください)[P10]。初期の Unix システムでは、約 20 コールのより簡潔なサブセットが公開されました。

TIP: USE PROTECTED CONTROL TRANSFER(保護された制御転送を使用する)

ハードウェアは、異なる実行モードを提供することによって OS を支援する。ユーザー mode では、アプリケーションはハードウェアリソースに完全にアクセスできません。カーネル mode では、OS はマシンの全リソースにアクセスできます。カーネルやトラップからユーザー mode のプログラムに戻すための特別な指示や、OS がトラップテーブルがメモリ上にあるハードウェアに指示するための指示も提供されています。

システムコールを実行するには、プログラムが特別なトラップ命令を実行する必要があります。この命令は同時にカーネルにジャンプし、特権レベルをカーネル mode に上げます。一度カーネル内で実行されると、システムは必要な特権操作(許可されている場合)を実行できるようになります。呼び出しプロセスに必要な作業を行うことができます。終了すると、OS は特別な `return from trap` 命令(トラップ帰還命令)を呼び出します。これは、呼び出し元のユーザープログラムに戻り、同時に特権レベルをユーザー mode に戻します命令です。

トラップを実行するときは、OS が `return from trap` 命令を発行したときに正しく戻るために、呼び出し元

のレジスタを十分に保存する必要があるという点で、ハードウェアは少し注意する必要があります。たとえば x86 では、プロセッサはプログラムカウンタ、フラグ、その他のいくつかのレジスタをプロセスごとのカーネルスタックにプッシュします。return from trap はこれらの値をスタックからポップし、ユーザー・モード・プログラムの実行を再開します（詳細については、インテルのシステム・マニュアル [I11] を参照してください）。他のハードウェアシステムでは異なる表記法が使用されていますが、基本コンセプトはプラットフォーム間で似ています。

ここで、詳細な説明を飛ばしていた重要なものが 1 つあります。トラップはどのコードを OS 内部でどのように実行するのかを知っているでしょうか？ ということです。答えは明白で、呼び出し元のプロセスはジャンプするアドレスを指定できません（プロセージャを呼び出すときと同じように）。もし、呼び出し元のプロセスがアドレスを指定できたとしましょう。そうすると、プログラムがカーネル内のどこにでもジャンプすることができますといった、非常に悪い考えになってしまふのです。したがって、カーネルは、トラップで実行されるコードを慎重に制御する必要があります。

カーネルは、起動時にトラップテーブルを設定することで、そのようにします。マシンが起動すると、特権（カーネル）モードで実行されるので、必要に応じてマシンのハードウェアを自由に設定することができます。OS が最初に行うことの 1 つは、例外的なイベントが発生したときに実行するコードをハードウェアに伝えることです。たとえば、ハードディスク割り込みが発生したとき、キーボード割り込みが発生したとき、またはプログラムがシステムコールを行うときに、どのコードを実行すべきでしょうか？ OS は、これらのトラップハンドラの場所をハードウェアに通知します。通常、何らかの特別な指示があります。ハードウェアに通知されると、マシンが次にリブートされるまで、これらのハンドラの位置が記憶されます、これにより、システムコールおよび他の例外的なイベントが発生したときに何をするか（すなわち、ジャンプするコード）を知ります。

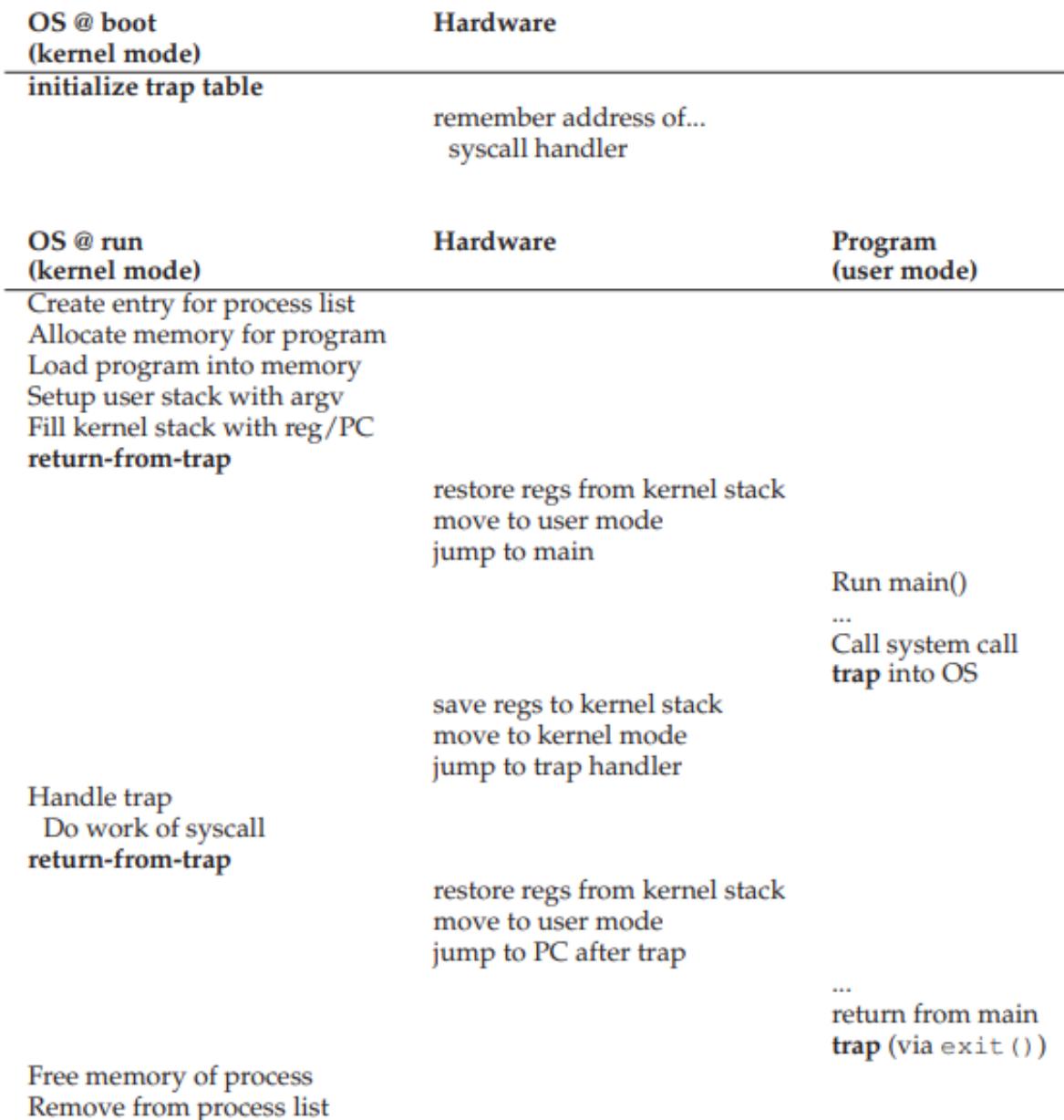


Figure 6.2: Limited Direct Execution Protocol

正確なシステムコールを指定するには、通常、各システムコールにシステムコール番号が割り当てられます。したがって、ユーザコードは、呼び出したいのシステムコール番号をレジスタまたはスタック上の指定された位置に配置する役割を担います。OS はトラップハンドラ内でシステムコールを処理するときにこの番号を調べ、有効であることを確認し、有効であれば対応するコードを実行します。このレベルの間接指定は、保護の一種として機能します。ユーザーコードは、ジャンプ先の正確なアドレスを指定することはできず、number を介して特定のサービスを要求する必要があります。

最後に、トラップテーブルがどこにあるかをハードウェアに伝える命令を実行できることは、非常に強力な機能です。つまり、特権操作もあります。この命令をユーザモードで実行しようとすると、ハードウェアはこれを許可しません。(ヒント：adios、問題のプログラム)。もし、あなた自身のトラップテーブルをインストールできたら、システムはどんな恐ろしいことができるでしょうか？ そんなマシンをあなたは引き継ぎたいですか？

タイムライン(図 6.2 の時間が下がるにつれて)は、プロトコルをまとめたものです。各プロセスには、カーネル

ネルに入りする際にレジスタ（汎用レジスタとプログラムカウンタを含む）が（ハードウェアによって）保存され、そこから復元されるカーネルスタックがあると仮定します。

LDE プロトコルには 2 つのフェーズがあります。最初の（ブート時に）カーネルはトラップテーブルを初期化し、CPU はその後の使用のためにその場所を覚えています。カーネルは特権命令（すべての特権命令は太字で強調表示されています）を介して行います。

2 番目のプロセス（プロセス実行時）では、カーネルは、プロセスの実行を開始するために return from trap 命令を使用する前に、いくつかのことを設定します（たとえば、プロセスリストにノードを割り当て、メモリを割り当てる）。CPU をユーザー モードに切り替え、プロセスの実行を開始します。プロセスがシステムコールを発行することを望む場合、プロセスはそれを処理する OS に再びトラップし、再びトラップからプロセスへの戻り値を介して制御を返します。プロセスはその作業を完了し、main() から戻ります。これは通常、プログラムを適切に終了するスタブコードに戻ります（たとえば、OS にトラップする exit() システムコールを呼び出します）。この時点では、OS はクリーンアップされ、実行されます。

6.3 Problem #2: Switching Between Processes

次の direct execution の問題は、プロセス間の切り替えを実現することです。プロセス間の切り替えは簡単ですね。OS は 1 つのプロセスを停止し、別のプロセスを開始するだけで済むはずです。ではいったい何が大きな問題だろうか？しかし、実際にはややこしいことです。具体的には、プロセスが CPU 上で実行されている場合、これは定義上、OS が実行されていないことを意味します。OS が稼働していない場合、どうすれば何ができるのですか？（ヒント：できない）これは哲学的に聞こえますが、実際の問題です。つまり、CPU で実行されていない場合、OS がアクションを実行する方法がないです。

THE CRUX: HOW TO REGAIN CONTROL OF THE CPU(CPU の制御を元に戻す方法) どのようにして、オペレーティングシステムは CPU を制御してプロセス間を切り替えることができますか？

A Cooperative Approach: Wait For System Calls

過去にいくつかのシステムが採用してきたアプローチ（例えば、Macintosh オペレーティングシステムの旧バージョン [M11] や古い Xerox Alto システム [A79]）は、協調的アプローチとして知られています。このスタイルでは、OS はシステムのプロセスが合理的に動作することを信頼します。長時間実行されるプロセスは、CPU が定期的に CPU を放棄して、OS が他のタスクを実行することを決定できると想定されます。

したがって、このユートピアの世界で友好的なプロセスが CPU を放棄するのはどのようなものでしょうか？ほとんどのプロセスは、例えば、ファイルを開いて読み込んだり、別のマシンにメッセージを送信したり、新しいプロセスを作成したりするなど、システムコールを頻繁に行うことによって、CPU の OS への制御をかなり頻繁に転送します。このようなシステムには、明示的なシステムコールが含まれていることが多く、OS に制御を渡す以外は何もないので、他のプロセスを実行できます。

アプリケーションは、何か違法行為をした場合にも、OS に制御を移します。たとえば、アプリケーションがゼロで割るか、アクセスできないメモリにアクセスしようとすると、OS にトラップが生成されます。その後、OS は CPU の制御を再度行います（問題のプロセスを終了させる可能性があります）。

したがって、協調スケジューリングシステムでは、システムコールや何らかの不正な操作が行われるのを待つ、OS が CPU の制御を取り戻します。また、あなたはこの受動的なアプローチが理想よりも劣っているのではないかと考えているかもしれません。たとえば、プロセス（悪意のあるバグや完全なバグなど）が無限ループで終了し、システムコールを実行しない場合などはどうなりますか？OS は何をすることができですか？

TIP: DEALING WITH APPLICATION MISBEHAVIOR(誤った動きをするアプリケーションの処理の仕方) オペレーティングシステムは、設計(悪意のある)または事故(バグ)のいずれかを行なうべきでない何かをしようとするプロセスに対処しなければならないことがあります。最新のシステムでは、OSがこのような不正行為を処理しようとする方法は、単に犯人を終了させることです。ワンストライク！バッターアウト！やり方としては残酷かもしれません...。ところで、違法にメモリにアクセスしたり、違法な命令を実行しようとすると、OSは何をすべきでしょうか？

A Non-Cooperative Approach: The OS Takes Control

ハードウェアの追加の助けがなければ、プロセスがシステムコール(またはミス)を拒否してOSに制御を戻すことができない場合、OSはまったく何もできません。実際、協調的なアプローチでは、プロセスが無限ループに陥ってしまったときの唯一の手段は、コンピュータシステムのすべての問題に対する古くからの解決策であるマシンを再起動することです。

THE CRUX: HOW TO GAIN CONTROL WITHOUT COOPERATION(協力なしに制御を得る方法) プロセスが協調的でなくとも、OSはどのようにしてCPUを制御できますか？不正なプロセスがマシンを占有しないようにするには、OSは何ができますか？

TIP: USE THE TIMER INTERRUPT TO REGAIN CONTROL(制御装置をリセットするためにタイマ割り込みを使用する) タイマー割り込みを追加することで、プロセスが非協調的な方法で動作しても、CPU上でOSを再実行することができます。したがって、このハードウェア機能は、OSがマシンの制御を維持するのを助ける上で不可欠です。

その答えは単純です。何年も前にコンピュータシステムを構築している多くの人々によって発見されました。それは、timer interrupt[M + 63]を使うことです。タイマー装置は非常に多くのミリ秒ごとに割り込みを発生させるようにプログラムすることができます。割り込みが発生すると、現在実行中のプロセスが停止し、OS内の事前設定された割り込みハンドラが実行されます。この時点で、OSはCPUの制御を取り戻しました。したがって、現在のプロセスを停止し、別のプロセスを開始することができます。

以前にシステムコールで説明したように、OSはタイマ割り込みが発生したときに実行するコードをハードウェアに通知する必要があります。したがって、ブート時には、OSはまさに実行するコードをハードウェアに通知することを行います。第2に、ブートシーケンス中にも、OSはタイマーを開始しなければいけません。これはもちろん特権操作です。タイマーが開始されると、OSは制御が最終的に特権操作で戻されるという点、OSが自由にユーザープログラムを実行できるという点で安全です。タイマーは、(特権操作でもある)電源をオフにすることができます。これについては、同時実行性のところでより詳しく説明します。

割り込みが発生したとき、特に割り込みが発生したときに実行されていたプログラムの状態を十分に保存して、その後のreturn from trap命令が実行中のプログラムを正常に再開できるようにするには、ハードウェアに若干の責任があることに注意してください。この一連のアクションは、カーネルへの明示的なシステムコールトラップ中のハードウェアの振る舞いと非常によく似ています。具体的には、return from trap命令を用いて、さまざまなレジスタが(カーネルスタックに)簡単に保存と復元が行われます。

Saving and Restoring Context

OSが制御を取り戻したので、システムコールを介して、またはタイマ割り込みを介してより強力に、現在実行中のプロセスを継続して実行するか、別のプロセスに切り替えるかを決定する必要があります。この決定はスケジューラと呼ばれるオペレーティングシステムの一部によって行われます。次のいくつかの章でスケジューリング方針について詳しく説明します。

切り替えが決定された場合、OS はコンテキストスイッチと呼ばれる低レベルのコードを実行します。コンテキストスイッチは概念的に単純です。すべての OS が実行する必要があるのは、現在実行中のプロセス（例えば、カーネルスタック上にある）にいくつかのレジスタ値を保存し、すぐに実行するプロセスのいくつかを復元することです（そのカーネルスタックから）。そうすることにより、OS は、return from trap 命令が最終的に実行されたときに、実行中のプロセスに戻るのではなく、別のプロセスの実行を再開するようにします。

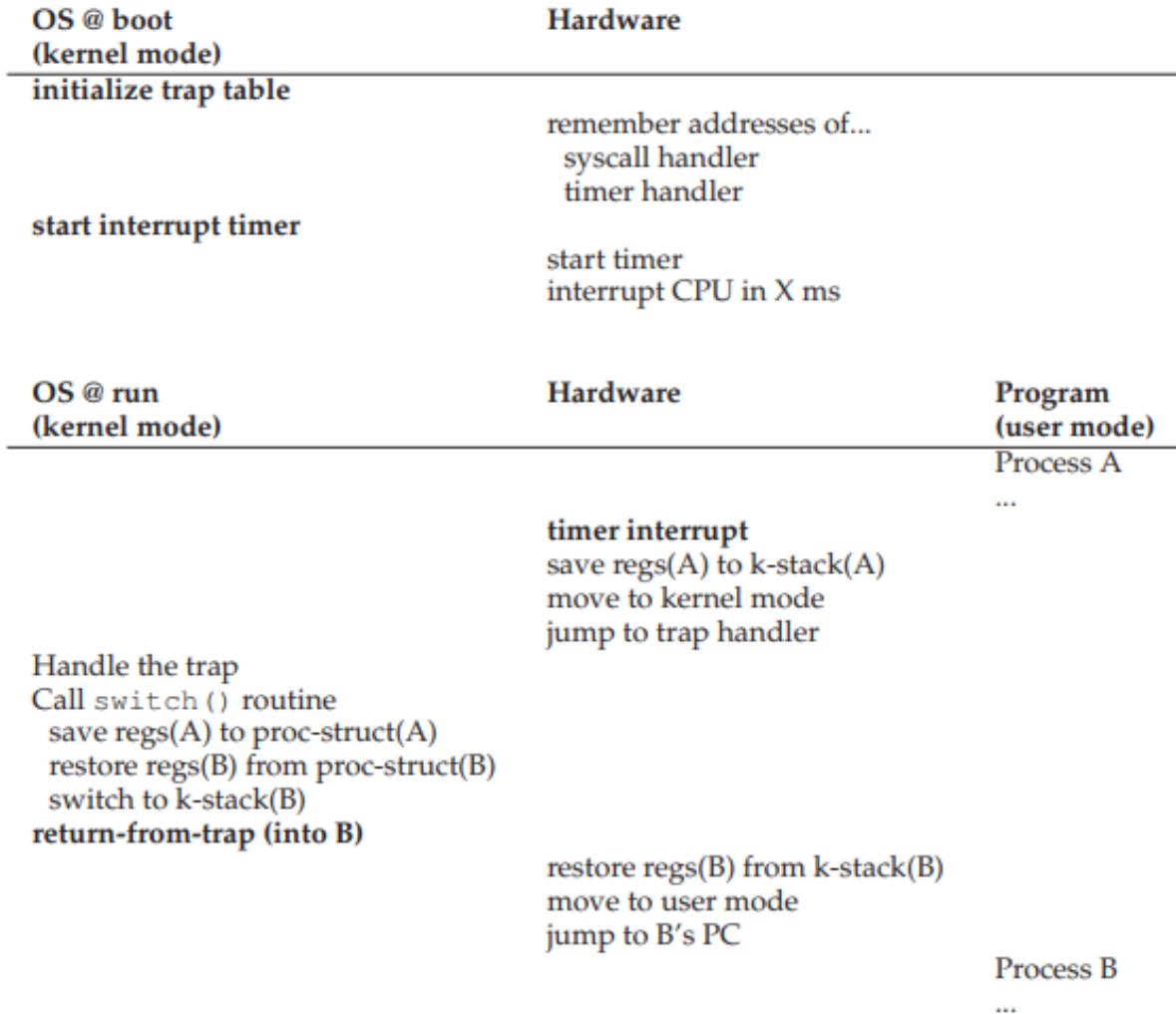


Figure 6.3: Limited Direct Execution Protocol (Timer Interrupt)

現在実行中のプロセスのコンテキストを保存するために、OS は、汎用レジスタ、PC、および現在実行中のプロセスのカーネルスタックポインタを保存するために、いくつかの低レベルのアセンブリコードを実行し、PC を起動し、すぐに実行されるプロセスのためにカーネルスタックに切り替えます。スタックを切り替えることによって、カーネルは 1 つのプロセス（中断されたプロセス）のコンテキストでスイッチコードへの呼び出しに入り、別のプロセス（直ちに実行されるもの）のコンテキストで戻ります。そして、OS が最終的に return from trap 命令を実行すると、直ちに実行されるプロセスが現在実行中のプロセスになります。したがって、コンテキストスイッチは完了です。

プロセス全体のタイムラインを図 6.3 に示します。この例では、プロセス A が実行されていて、タイマ割り込みによって中断されています。ハードウェアはレジスタをカーネルスタックに保存し、カーネルに入れます（カーネルモードに切り替える）。タイマ割り込みハンドラでは、OS は実行中のプロセス A からプロセス B に切り替えることを決定します。その時点で、現在のレジスタ値を（A のプロセス構造に）慎重に保存する `switch()` ルーチンを呼び出し、（そのプロセス構造体のエントリから）B を処理し、具体的には B のカーネル

スタックを使用するようにスタックポインタを変更することによってコンテキストを切り替えます。最後に、OS は return from trap 命令を使用して、B のレジスタを復元して実行します。

このプロトコル中に起こるレジスタセーブ/リストアには 2 種類のタイプがあることに注意してください。1つは、タイマ割り込みが発生したときです。この場合、実行中のプロセスのユーザー・レジスタは、そのプロセスのカーネル・スタックを使用して、ハードウェアによって暗黙的に保管されます。2つめは、OS が A から B に切り替えることを決定したときです。この場合、カーネルレジスタはソフトウェア(すなわち OS)によって明示的に保存されますが、今回はプロセスのプロセス構造内のメモリに保存されます。後者の動作では、A がカーネルにトラップされた後、B がカーネルにトラップされ、システムが実行します。

そのようなスイッチがどのように制定されているかを理解するために、図 6.4 に xv6 のコンテキストスイッチコードを示します。あなたがそれを理解できるかどうかを確認してください(そうするためには、xv6 といくつかの xv6 について知っておく必要があります)。コンテキスト構造 old と new は、それぞれ古いプロセスのプロセス構造と新しいプロセスのプロセス構造にあります。

```

1  # void swtch(struct context **old, struct context *new);
2  #
3  # Save current register context in old
4  # and then load register context from new.
5  .globl swtch
6  swtch:
7      # Save old registers
8      movl 4(%esp), %eax    # put old ptr into eax
9      popl 0(%eax)         # save the old IP
10     movl %esp, 4(%eax)   # and stack
11     movl %ebx, 8(%eax)   # and other registers
12     movl %ecx, 12(%eax)
13     movl %edx, 16(%eax)
14     movl %esi, 20(%eax)
15     movl %edi, 24(%eax)
16     movl %ebp, 28(%eax)
17
18     # Load new registers
19     movl 4(%esp), %eax    # put new ptr into eax
20     movl 28(%eax), %ebp  # restore other registers
21     movl 24(%eax), %edi
22     movl 20(%eax), %esi
23     movl 16(%eax), %edx
24     movl 12(%eax), %ecx
25     movl 8(%eax), %ebx
26     movl 4(%eax), %esp   # stack is switched here
27     pushl 0(%eax)        # return addr put in place
28     ret                  # finally return into new ctxt

```

Figure 6.4: The xv6 Context Switch Code

6.4 Worried About Concurrency?

「システムコール中にタイマー割り込みが発生するとどうなるのですか？」または「1つの割り込みを処理して別のものが起きたときにどうなりますか？」と思っている人もいます。カーネルで扱うのが難しくないのでありますか？

答えは YES!です。OS は、割り込みやトラップの処理中に別の割り込みが発生した場合に何が起こるかについて、実際に懸念する必要があります。これは、実際にはこの本の第 2 部全体の正確な話題であり、並行性に関するものです。それまでは詳細な議論を延期する予定です。

あなたの知識欲求を搔き立てるために、OS がどのようにこれらのトリッキーな状況を処理するかの基本をいくつか紹介します。OS が行う簡単なことの 1 つは、割り込み処理中に割り込みを無効にすることです。これにより、1 つの割り込みが処理されるときに、他の割り込みが CPU に渡されることはありません。もちろん、OS はそうすることに注意する必要があります。なぜなら、割り込みを長時間無効にすると割り込みが失われる可能性があるからです（つまり、技術的には悪い）。

また、オペレーティングシステムは、内部データ構造への同時アクセスを保護するために、多くの洗練されたロック方式を開発しています。これにより、複数のアクティビティをカーネル内で同時に実行することができます。特に、マルチプロセッサで便利です。しかし、この本の次の記事では並行性について説明しますが、このようなロックは複雑になり、さまざまな興味深い見つけにくいバグにつながります。

ASIDE: HOW LONG CONTEXT SWITCHES TAKE(コンテキストスイッチにどのくらい時間がかかるか)

とある好奇心がある人が質問をしたとしましょう。たとえば、「コンテキストスイッチのようなものがどれくらい時間がかかりますか？」または、「あるいはシステムコールはどのくらい時間がかかりますか？」という質問のために、lmbench [MS96] というツールがあります。

上記の質問のことを正確に測定するだけでなく、関連性のあるいくつかのパフォーマンス指標も測定します。結果は、時間の経過と共にかなり改善され、プロセッサの性能を大まかに追跡しています。たとえば、1996 年に Linux 1.3.37 を 200 MHz P6 CPU 上で実行すると、システムコールは約 4 マイクロ秒かかり、コンテキストスイッチは約 6 マイクロ秒でした [MS96]

現代のシステムは、1~2GHz または 3GHz のプロセッサを搭載したシステムでは、1 マイクロ秒未満の結果でほぼ一桁の性能を発揮します。すべてのオペレーティングシステムの動作が CPU のパフォーマンスを追跡するわけではないことに注意してください。Ousterhout が観察しているように、多くの OS 操作はメモリを大量に消費しており、メモリ帯域幅はプロセッサ速度のように大幅に向上していません [O90]。

したがって、あなたの仕事量によっては、最新かつ最高のプロセッサーを購入しても、希望通りの速度で OS が動作するわけではありません。

6.5 Summary

我々は、限定された direct execution と総称する CPU 仮想化を実装するためのいくつかの重要な低レベルのメカニズムについて説明しました。基本的な考え方は簡単です。CPU 上で実行したいプログラムを実行するだけですが、ハードウェアをセットアップして、OS の支援なしにプロセスができるのを制限するようにしてください。この一般的なアプローチは実生活でも行われます。たとえば、子供がいると考えてみましょう。危険なものを入れたキャビネットをロックし、電気ソケットをカバーします。このように部屋が準備されたら、部屋の最も危険な面が制限されていることを知って、赤ちゃんは自由に歩き回ること（ベビープルーフ）ができます。

同様の方法で、OS も考えることができます。OS は最初に(ブート時に)トランプハンドラを設定し、割り込みタイマーを起動し、制限されたモードでプロセスを実行するだけで、CPU を「ベビープルーフ」します。そうすることで、OS がプロセスを効率的に実行でき、特権操作を実行するために OS の介入を必要とするだけでなく、CPU を長時間独占してスイッチアウトする必要がある場合もあります。

したがって、CPU を仮想化するための基本的なメカニズムがあります。しかし、重要な質問は答えられていません。それは、「どのプロセスが所定の時間に実行すべきですか?」という質問です。これは、スケジューラが答えなければならない質問です。したがって、私たちの次の調査トピックはスケジューラになります。

TIP: REBOOT IS USEFUL

以前は、協調的なプリエンプションのもとでの無限ループ(および類似の動作)に対する唯一の解決策は、マシンを再起動することでした。このハックを嘲笑するかもしれません、研究者は再起動(または一般的にソフトウェアの一部を起動)は堅牢なシステムを構築する上で非常に有用なツールになることを示しています [C+04]

具体的には、再起動はソフトウェアを既知のテスト済みの状態に戻すので便利です。また、再起動すると、そうでなければ扱いにくい古いまたは漏れたりソース(例えば、メモリ)を再利用する。最後に、再起動は簡単に自動化できます。これらの理由から、システム管理ソフトウェアの大規模クラスタ・インターネット・サービスでは、マシンをリセットして上記の利点を得るために、一連のマシンを定期的に再起動することは珍しいことではありません。

したがって、次に再起動すると、ちょっと醜いハックを制定するだけではありません。むしろ、コンピュータシステムの動作を改善するために時間をかけてテストされたアプローチを使用しています。やったぜ!

参考文献

- [A79] “Alto User’s Handbook”
Xerox Palo Alto Research Center, September 1979
Available: <http://history-computer.com/Library/AltoUsersHandbook.pdf>
An amazing system, way ahead of its time. Became famous because Steve Jobs visited, took notes, and built Lisa and eventually Mac.
- [C+04] “Microreboot — A Technique for Cheap Recovery”
George Canea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, Armando Fox
OSDI ’04, San Francisco, CA, December 2004
An excellent paper pointing out how far one can go with reboot in building more robust systems.
- [I11] “Intel 64 and IA-32 Architectures Software Developer’s Manual”
Volume 3A and 3B: System Programming Guide
Intel Corporation, January 2011
- [K+61] “One-Level Storage System”
T. Kilburn, D.B.G. Edwards, M.J. Lanigan, F.H. Sumner
IRE Transactions on Electronic Computers, April 1962
The Atlas pioneered much of what you see in modern systems. However, this paper is not the best one to read. If you were to only read one, you might try the historical perspective below [L78].
- [L78] “The Manchester Mark I and Atlas: A Historical Perspective”
S. H. Lavington
Communications of the ACM, 21:1, January 1978

A history of the early development of computers and the pioneering efforts of Atlas.

[M+63] “A Time-Sharing Debugging System for a Small Computer”

J. McCarthy, S. Boilen, E. Fredkin, J. C. R. Licklider

AFIPS '63 (Spring), May, 1963, New York, USA

An early paper about time-sharing that refers to using a timer interrupt; the quote that discusses it:
“The basic task of the channel 17 clock routine is to decide whether to remove the current user from core
and if so to decide which user program to swap in as he goes out.”

[MS96] “lmbench: Portable tools for performance analysis”

Larry McVoy and Carl Staelin

USENIX Annual Technical Conference, January 1996

A fun paper about how to measure a number of different things about your OS and its performance.
Download lmbench and give it a try.

[M11] “Mac OS 9”

January 2011

Available: http://en.wikipedia.org/wiki/Mac_OS_9

[O90] “Why Aren’t Operating Systems Getting Faster as Fast as Hardware?”

J. Ousterhout

USENIX Summer Conference, June 1990

A classic paper on the nature of operating system performance.

[P10] “The Single UNIX Specification, Version 3”

The Open Group, May 2010

Available: <http://www.unix.org/version3/>

This is hard and painful to read, so probably avoid it if you can.

[S07] “The Geometry of Innocent Flesh on the Bone:

Return-into-libc without Function Calls (on the x86)”

Hovav Shacham

CCS '07, October 2007

One of those awesome, mind-blowing ideas that you’ll see in research from time to time. The author shows that if you can jump into code arbitrarily, you can essentially stitch together any code sequence you like (given a large code base); read the paper for the details. The technique makes it even harder to defend against malicious attacks, alas.

7. Scheduling: Introduction

現時点で、実行中のプロセスの低レベルのメカニズム（例えば、コンテキストの切り替え）は明確にわかっていないといけません。そうでない場合は、1つまたは2つの章に戻って、低レベルのメカニズムがどのように機能するかの説明を読んで理解してください。しかし、OSスケジューラが採用している高度なポリシーについてはまだ理解していません。これからは、さまざまな人々が長年にわたって開発してきた一連のスケジューリング方針（時には専門分野と呼ばれる）を提示します。

スケジューリングの起点は、実際には、コンピュータシステム以前からあります。初期のアプローチは運用管理の分野から取り上げられ、コンピュータに適用されました。このことは少しも不思議でないでしょう、流れ作業での組み立てやその他多くのことがらにはスケジューリングが必要であり、同様の問題の多くはそこにも存在するのですから。したがって、私たちの問題は以下のようになります。

THE CRUX: HOW TO DEVELOP SCHEDULING POLICY(スケジューリング方針を開発する方法)

スケジューリング方針について考えるための基本的なフレームワークをどのように開発する必要があるでしょうか？ 主要な前提は何でしょうか？ どのような指標が重要になるでしょうか？ コンピュータシステムの初期段階ではどのような基本的なアプローチが使用されているのでしょうか？

7.1 Workload Assumptions

適応可能なポリシーの範囲に入る前に、まずシステム内で実行されているプロセスについて簡単に仮定を立てましょう。時にはまとめて仕事量と呼ばれます。仕事量を決定することは、ポリシー作成の重要な部分であり、仕事量について知るほどポリシーが細かく調整されます。

私たちがここで作った仕事量の仮定はほとんど非現実的ですが、それは問題ありません。私たちが行くにつれてリラックスし、最終的には…(思考停止) すべてに対応できる運用スケジューリング規律になるでしょう。システムで実行されているプロセス（ジョブとも呼ばれる）について、次の前提を設定します。

1. 各ジョブは同じ時間実行されます。
2. すべてのジョブが同時に到着します。
3. 開始されると、各ジョブは完了まで実行されます。
4. すべてのジョブはCPUのみを使用する（すなわち、I/Oを実行しない）
5. 各ジョブの実行時間は既知である。

これらの前提の多くは非現実的であると言いましたが、オーウェルの動物園ではいくつかの動物が他の動物よりも同等であるように [O45]、いくつかの仮定はこの章の他の仮定よりも非現実的です。特に、各ジョブの実行時間はわかっているかもしれません。スケジューラはすべてを把握しているでしょう。しかし、スケジューラはすべてを把握できるようになりますが、すぐにはうまくいかないでしょう。

7.2 Scheduling Metrics

仕事量の前提をする以外にも、スケジューリング・メトリックという異なるスケジューリング・ポリシーを比較できるようにするには、もう1つ必要です。メトリックは、何かを測定するために使用するものであり、スケジューリングに意味があるさまざまなメトリックがあります。

しかし今のところ、単一の指標、つまりターンアラウンド・タイムを単に取るだけで、私たちの人生を単純

化しましょう。ジョブの所要時間は、ジョブが完了した時刻からジョブがシステムに到着した時刻を引いた時刻として定義されます。より正式には、ターンアラウンドタイム $T_{turnaround}$ は次の通りです:

$$T_{turnaround} = T_{completion} - T_{arrival} \quad (7.1)$$

すべての仕事が同時に到着すると仮定しているので、今のところ $T_{arrival} = 0$ 、したがって $T_{turnaround} = T_{completion}$ です。この事実は、前述の前提条件を緩和するにつれて変化します。ターンアラウンドタイムはパフォーマンスマトリックであることに注意してください。もう一つの測定基準は、Jain の公平指数 (Jain's Fairness Index [J91]) によって測定された公平さです。パフォーマンスと公平さは、スケジューリングにおいてよく不安定にさせるものです。例えば、スケジューラは性能を最適化することができるが、いくつかのジョブが実行されるのを防止し、公平性を低下させるといったことが考えられます。この難解は「人生はいつも完璧というわけではない」ことを示しています。

7.3 First In, First Out (FIFO)

実装可能な最も基本的なアルゴリズムは、FIFO(First In, First Out) スケジューリング、または First Come, First Served FCFS) と呼ばれます。

FIFO にはいくつかの肯定的な特性があります。これは明らかに単純で実装が容易です。そして、私たちの前提を考えれば、それはかなりうまくいくでしょう。簡単な例を一緒に考えましょう。ほぼ同時に ($T_{arrival} = 0$)、3 つのジョブがシステム A、B、C に到着したとします。FIFO は三つのうちからどれかをとって仕事を最初にしなければいけないので、A が同時に到着したときに、A は C の前に僅差で到着したばかりの B の前に僅差で到着したと仮定しましょう。これらのジョブの平均所要時間はどのくらいですか？

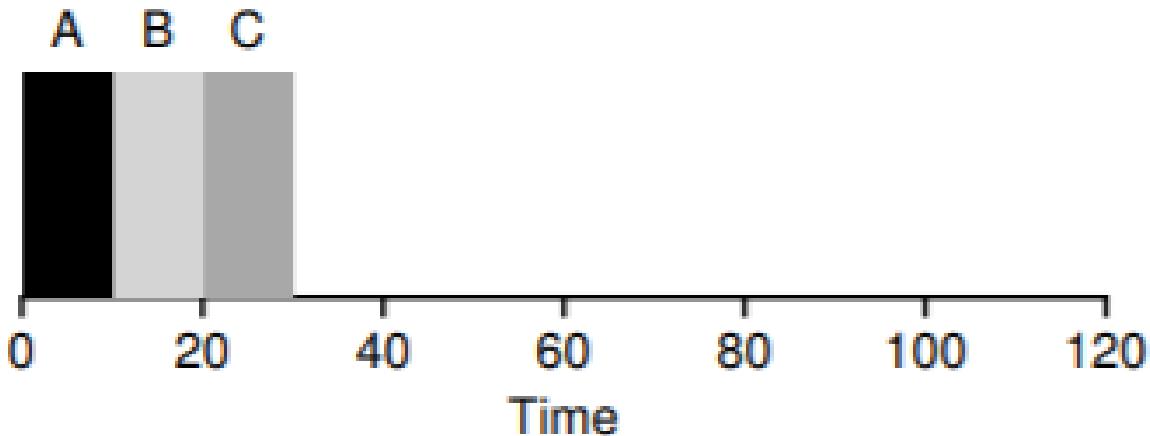


Figure 7.1: FIFO Simple Example

図 7.1 から、A は 10 で終了し、B は 20 で、C は 30 で終了したことがわかります。したがって、3 つのジョブの平均所要時間は単純に $\frac{10+20+30}{3} = 20$ 。ターンアラウンドタイムの計算は簡単です。

5 つの前提条件のうちの 1 つを緩和させましょう。特に、仮説 1 を緩和して、それぞれのジョブが同じ時間実行されると仮定しないようにしてください。FIFO は今どのように機能しますか？ FIFO のパフォーマンスを落とさせるためには、どのような仕事量を構築できますか？

とりあえず、異なる長さのジョブがどのように FIFO スケジューリングの問題を引き起こすかを示す例を見てみましょう。特に 3 つのジョブ (A, B, C) を仮定しますが、この時間 A は 100 秒、B と C はそれぞれ 10 回実行されます。

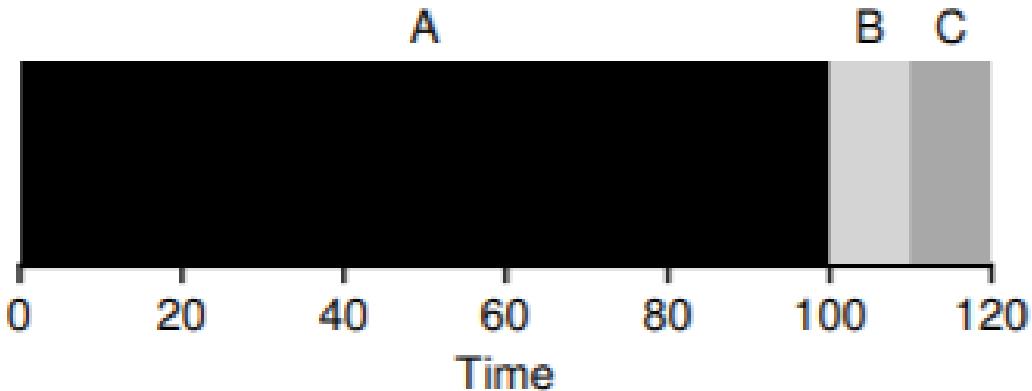


Figure 7.2: Why FIFO Is Not That Great

図 7.2 に示すように、ジョブ A は 100 秒間完全に実行されてから、B または C が実行されます。したがって、システムの平均所要時間は高くなります。 $\frac{100+110+120}{3} = 110$ 秒です。

この問題は、一般的に、コンボイエフェクト [B+79] と呼ばれ、リソースの比較的短い潜在的なコンシューマーの数が、重量のあるリソースコンシューマーの後ろにキューイングされます。このスケジューリングのシナリオでは、食料雑貨品店での 1 コマを思い出させるかもしれません。レジに並んでいるときにすでに、あなたの目の前に人がいて、食料品などがいっぱいのカートが 3 つあります。しばらくレジ精算に時間がかかるでしょう。だから何をすべきでしょうか？ さまざまな時間に実行される新しいジョブの現実に対処するために、より良いアルゴリズムを開発するにはどうすればよいでしょうか？

TIP: THE PRINCIPLE OF SJF(SJF の原則)

SJF は、顧客（または、この場合は職種）ごとに認識されるターンアラウンドタイムが重要なシステムに適用できる一般的なスケジューリングの原則を表します。問題の施設が顧客満足を気にしている場合、SJF を考慮した可能性があります。たとえば、食料雑貨品店には、買う数少ない買い物客が、品数が多い買い物をする家族の後ろにつかないようにするために、「10 項目以下」のレジがあります。

7.4 Shortest Job First (SJF)

非常に単純なアプローチがこの問題を解決することが判明しました。実際には、オペレーションリサーチ [C54, PV56] から持ってきたアイデアであり、コンピュータシステムのジョブのスケジューリングに適用されます。この新しいスケジューリング規律は Shortest Job First(SJF) と呼ばれています。この名前はポリシーを完全に記述しているため、覚えやすいはずです。まず最短のジョブを実行し、次に最短のジョブを実行します。

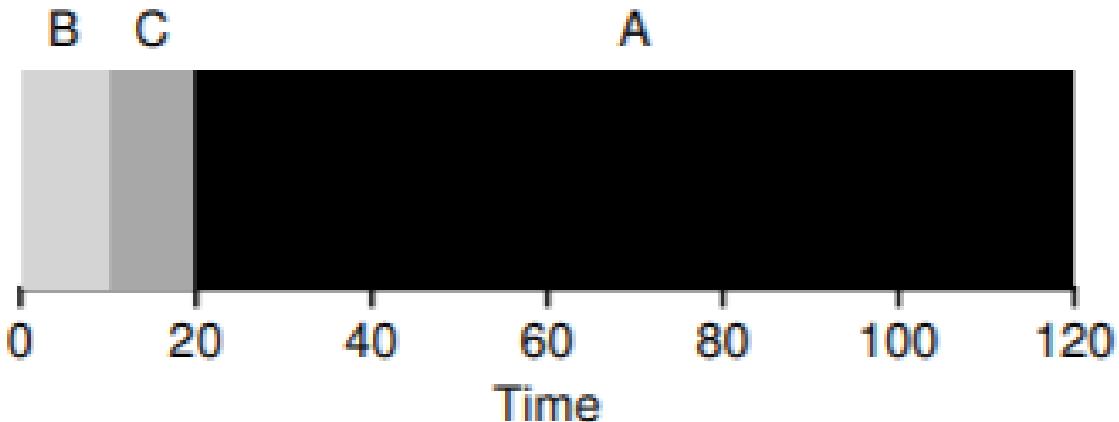


Figure 7.3: SJF Simple Example

上記の例を SJF のスケジューリング方針として考えてみましょう。図 7.3 は、A、B、C を実行した結果を示しています。この図から、SJF が平均ターンアラウンド時間に関してより優れたパフォーマンスを発揮する理由が分かりやすくなつたといいでしょう。A の前で B と C を実行するだけで、SJF は $\frac{10+20+120}{3} = 50$ に短縮し、2 倍以上の改善をもたらします。

事実、同時に到着するジョブについての仮定を仮定すると、SJF が実際に最適なスケジューリングアルゴリズムであることを証明することができます。しかし、理論や運用研究ではなく、システムに関する研究ですので証明することはしないです。

ASIDE: PREEMPTIVE SCHEDULERS

バッチ・コンピューティングの昔は、多くの非プリエンプティブ・スケジューラーが開発されました。このようなシステムでは、新しいジョブを実行するかどうかを検討する前に、各ジョブを完了するまで実行します。事実上すべての現代のスケジューラはプリエンティブです、つまり、別のプロセスを実行するために 1 つのプロセスを停止させます。これは、スケジューラが以前に学んだメカニズムを採用していることを意味します。具体的には、スケジューラはコンテキストスイッチを実行して、実行中のプロセスを一時停止し、別のプロセスを再開（または開始）することです。

この結果から、我々は SJF でスケジューリングするための良いアプローチを考え付きますが、私たちの仮定はまだかなり非現実的です。もう一度リラックスして考えてみましょう。具体的には、仮定 2 を対象とすることができます、現在ではジョブをすべてではなくいつでも到着できると仮定しています。これはどのような問題につながるでしょうか？ ここでは、例を使って問題を再び説明しましょう。今度は、A が $t = 0$ に到着し、100 秒間実行する必要があると仮定します。B と C は $t = 10$ に到着し、それぞれ 10 秒間実行する必要があります。純粋な SJF では、図 7.4 のようなスケジュールになります。

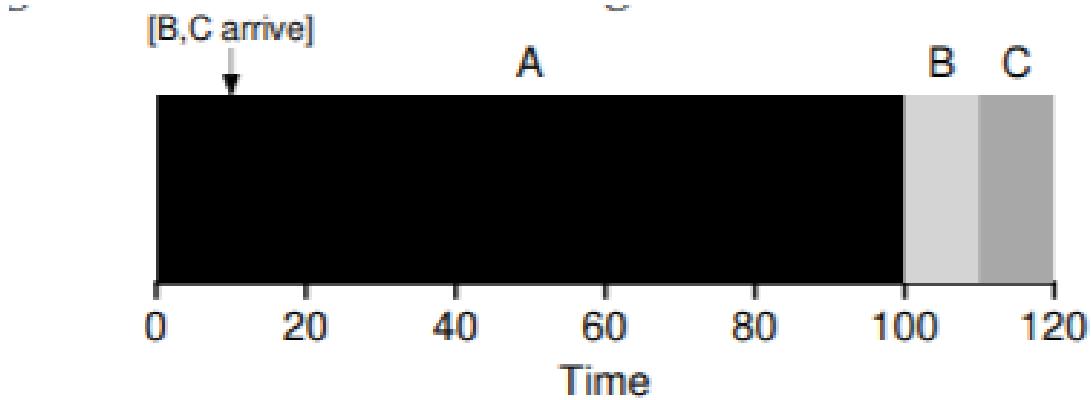


Figure 7.4: SJF With Late Arrivals From B and C

この図から分かるように、A の直後に B と C が到着したにもかかわらず、A が完了するまで待たされ、同じ護送船の問題を抱えています。これら 3 つのジョブの平均所要時間は $\frac{100+(110-10)+(120-10)}{3}$ です。スケジューラは何をするべきでしょうか？

7.5 Shortest Time-to-Completion First (STCF)

この懸念に対処するためには、想定を緩和する必要があります（ジョブは完了するまで実行する必要があります）。また、スケジューラ自体の中にいくつかの仕組みが必要です。思い出してみましょう。前回の議論で timer interrupts とコンテクストスイッチの話題がありました。それらを当てはめて考えてみましょう。B と C が到着すると、スケジューラは何か他のことを行うことができます。つまり、ジョブ A をプリエンティブして別のジョブを実行すると A は後で実行されます。

我々の定義による SJF は、非プリエンティブなスケジューラであり、上述の問題を抱えています。幸いにも、STJF(最短完了までの最短完了) または PSJF(Preemptive Shortest Job First) スケジューラ [CK68] と呼ばれる SJF にプリエンプションを追加するスケジューラがあります。新しいジョブがシステムに入るたびに、STCF スケジューラーは残りのジョブ（新しいジョブを含む）の中で最も時間が残っていないものを判別し、そのジョブをスケジュールします。したがって、この例では、STCF は A を優先して B と C を完了させます。終了したときにのみ、A の残りの時間がスケジュールされます。図 7.5 に例を示します。

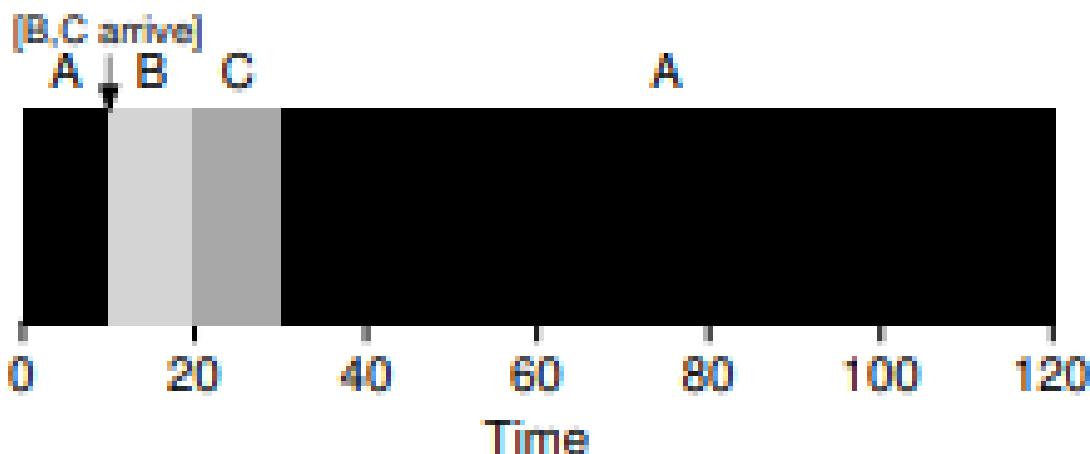


Figure 7.5: STCF Simple Example

その結果、50 秒 ($\frac{(120-0)+(20-10)+(30-10)}{3}$) の大幅に改善された平均所要時間が得られます。私たちの新し

い前提を考慮すると、STCF は確かに最適です。すべての仕事が同時に到着した場合に SJF が最適であるとすれば、おそらく STCF の最適性を見ることができます。

7.6 A New Metric: Response Time

したがって、ジョブの長さを知り、ジョブが CPU のみを使用し、唯一のメトリックがターンアラウンドタイムだった場合、STCF は素晴らしいポリシーになります。実際、多くの初期のバッチ・コンピューティング・システムでは、これらのタイプのスケジューリング・アルゴリズムが採用されていました。しかし、time slice のマシンの導入はそれらの考えをすべて変えました。ユーザーは端末に座り、システムからのインタラクティブなパフォーマンスを要求するようになりました。そして、新しいメトリックが生まれました。それは応答時間です。応答時間は、ジョブがシステムに到着してからスケジュールされた最初の時間までの時間として定義されます。より正式には以下の式になります。

$$T_{\text{response}} = T_{\text{firstrun}} - T_{\text{arrival}} \quad (7.2)$$

たとえば、上記のスケジュール (A が時刻 0 に到着し、B と C が時刻 10 に到着した場合) では、各ジョブの応答時間は、ジョブ A の場合は 0、B の場合は 0、C の場合は 10 平均 : 3.33) になります。STCF と関連する分野は、特にレスポンスタイムには適していません。たとえば、3 つのジョブが同時に到着した場合、3 つ目のジョブは、前の 2 つのジョブが完全に実行されるのを待ってから、1 回だけスケジュールされます。ターンアラウンドタイムには優れていますが、このアプローチは応答時間とインタラクティビティにとって非常に悪いものです。実際には、ターミナルに座って入力し、システムからの応答を見るのに 10 秒待たなければならぬと想像してください。したがって、まだ別の問題が残っています。「応答時間に敏感なスケジューラを構築するにはどうすればよいのか」ということです。

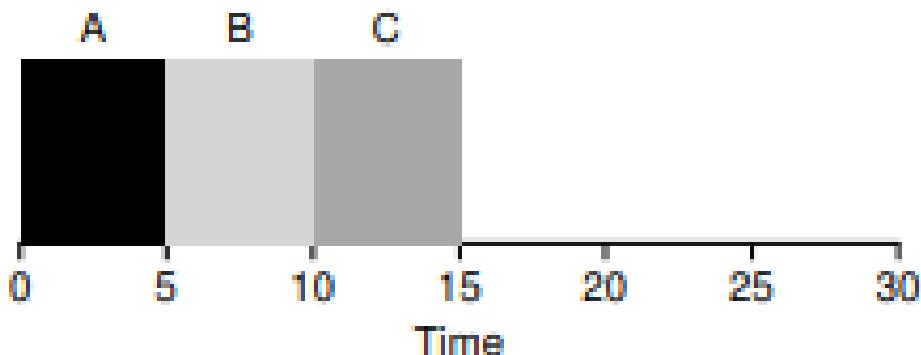


Figure 7.6: SJF Again (Bad for Response Time)

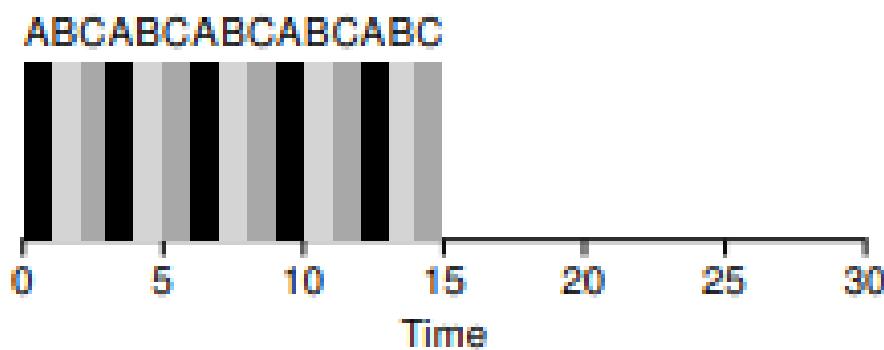


Figure 7.7: Round Robin (Good for Response Time)

7.7 Round Robin

この問題を解決するために、我々は古典的にラウンドロビン (RR) スケジューリング [K64] と呼ばれる新しいスケジューリングアルゴリズムを導入します。基本的な考え方は簡単です。ジョブを完了して実行する代わりに、RR は time slice で (時にはスケジューリングクォンタムと呼ばれます) ジョブを実行し、実行キューの次のジョブに切り替えます。ジョブが終了するまで繰り返し実行します。このため、RR は時々 time slicing と呼ばれます。タイムスライスの長さはタイマ割り込み期間の倍数でなければならないことに注意してください。したがって、タイマーが 10 ミリ秒ごとに割り込みする場合、タイムスライスは 10,20、または 10 ミリ秒の任意の他の倍数になる可能性があります。

RR をより詳細に理解するために、例を見てみましょう。3 つのジョブ A、B、C が同時にシステムに到着し、それぞれが 5 秒間実行したいとします。SJF スケジューラは、各ジョブを実行してから別のジョブを実行します (図 7.6)。対照的に、タイムスライスが 1 秒の RR はジョブをすばやく循環します (図 7.7)。RR の平均応答時間は、SJF の場合、 $\frac{0+1+2}{3} = 1$ で、平均応答時間は $\frac{0+5+10}{3} = 5$ です。

ご覧のとおり、タイムスライスの長さは RR にとって重要です。応答時間メトリックの下で RR のパフォーマンスが良好なほど短くなればなります。ただし、タイムスライスを短くすることは時には問題になります。例えば、突然コンテキスト切り替えのコストが全体のパフォーマンスにかかることがあります。したがって、タイムスライスの長さを決定することは、システム設計者とのトレードオフをもたらします。つまりシステムが応答しなくなるまでタイムスライスを長くするというのは、コンテキストスイッチのコストをなくす長さにもなります。

TIP: AMORTIZATION CAN REDUCE COSTS

コストカットの一般的な手法は、あるオペレーションに固定費がかかる場合にシステムで一般的に使用されます。そのコストをより少なくする (すなわち、より少ない回数を実行することによって)、システムへの総コストが低減されます。例えば、タイムスライスが 10ms に設定され、コンテキストスイッチコストが 1ms である場合、約 10 % の時間がコンテキスト切り替えにかかり、浪費されます。このコストをカットしたい場合、タイムスライスを例えば 100ms に増加させます。この場合、1 % 未満の時間がコンテキスト切り替えに費やされるため、タイムスライスのコストがカットされます。

コンテキスト切替えのコストは、ただ単にいくつかのレジスタを保存して復元する OS の動作からは生じないことに注意してください。ここでいうコストというのは、プログラムが実行されると、CPU キャッシュ、TLB、分岐予測子、およびその他のオンチップハードウェアに大量の状態が構築されます。別のジョブに切り替えると、この状態がフラッシュされ、現在実行中のジョブに関連する新しい状態が持ち込まれます。これは、パフォーマンスコスト [MB91] を顕著にする可能性があります。

応答時間が私たちの唯一のメトリックであるなら、妥当なタイムスライスを持つ RR は優れたスケジューラです。しかし、ターンアラウンドタイムはどうですか？ 上記の例をもう一度見てみましょう。それぞれ 5 秒の実行時間有する A、B、および C は同時に到着し、RR は (長い)1 秒タイムスライスを有するスケジューラである。上記の画像から、A は 13 で終了し、B は 14 で終了し、C は 15 で終了し、平均 14 であることがわかります。これはかなりひどい！

それはターンアラウンドタイム基準であるならば、RR は確かに最悪なポリシーの一つです。RR がやっていることは、できるだけ各ジョブを伸ばすことです。各ジョブを短い時間だけ実行してから、次のジョブに移動します。ターンアラウンドタイムはジョブが終了したときだけ気にするので、RR はほとんどの場合 pessimal であり、多くの場合単純な FIFO よりも悪いです。

より一般的には、公正である (すなわち、小さな時間スケールでアクティブプロセス間で CPU を均等に分

割する)RRなどのポリシーは、ターンアラウンドタイムなどでは、ほとんど機能しません。不公平の場合は、より短い仕事を完了まで実行することができますが、応答時間を犠牲にするしかありません。代わりに公正さを評価すると、応答時間は短縮されますが、ターンアラウンドタイムを犠牲にしています。この種のトレードオフはシステムでは一般的です。

我々は2種類のスケジューラを開発しました。最初のタイプ(SJF、STCF)はターンアラウンドタイムを最適化しますが、応答時間には悪いです。第2のタイプ(RR)は応答時間を最適化するが、ターンアラウンドには悪いです。また、仮定4(そのジョブはI/Oを実行しない)と仮定5(各ジョブの実行時間が分かっていること)という2つの仮定が緩和される必要があります。次に、これらの仮定に取り組んでみましょう。

TIP: OVERLAP ENABLES HIGHER UTILIZATION(オーバーラップはより高い活用を可能にする)

可能であれば、システムを最大限に活用するために操作を重複させてみてください。オーバーラップは、ディスクI/Oを実行するときやリモートマシンにメッセージを送信するときなど、多くの異なるドメインで役立ちます。いずれの場合でも、操作を開始してから他の作業に切り替えることは良いアイデアであり、システムの全体的な利用率と効率を向上させます。

7.8 Incorporating I/O

まず、仮定4を緩和します。もちろん、すべてのプログラムがI/Oを実行します。何も入力しなかったプログラムを想像してみましょう。毎回同じ出力を生成します。アウトプットのないものを想像してみましょう。そのプログラムは実行していても問題ありません。

スケジューラは、現在実行中のジョブがI/O中にCPUを使用しないため、ジョブがI/O要求を開始したときに決定することを明確に示しています。I/O完了を待ってブロックされます。I/Oがハード・ディスク・ドライブに送信されると、ドライブの現在の入出力負荷に応じて、プロセスが数ミリ秒以上ブロックされることがあります。したがって、スケジューラは、その時点でCPU上に別のジョブをスケジュールする必要があります。

スケジューラは、I/Oが完了したときにも決定を下す必要があります。これが発生すると割り込みが発生し、OSが実行されI/Oを発行したプロセスがブロックされて元の準備状態に戻ります。もちろんその時点でジョブを実行することもできます。OSはどのように各仕事を扱うべきでしょうか？

この問題をよりよく理解するには、それぞれ50ミリ秒のCPU時間が必要な2つのジョブAとBがあるとします。しかし、1つの明らかな違いがあります。Aは、10ミリ秒間実行され、その後、Bは単純に50ミリ秒のCPUを使用しないI/Oを実行しないのに対し、I/O要求(I/Oは、それぞれが10ミリ秒を取ることがここで仮定)を発行します。スケジューラはまずAを実行し、次にBを実行します(図7.8)。

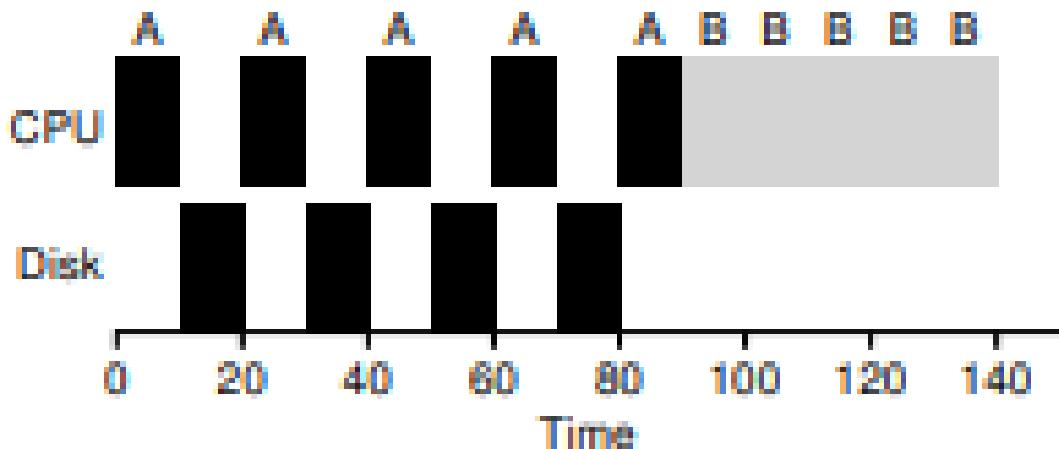


Figure 7.8: Poor Use of Resources

STCF スケジューラーを構築しようとしているします。このようなスケジューラは、A が 5 つの 10ms サブジョブに分割されているのに対し、B は単なる 50ms の CPU 時間が必要であるという事実をどのように説明するのでしょうか？明らかに、I/O を考慮する方法を考慮せずに 1 つのジョブを実行し、次に他のジョブを実行するだけでは意味がありません。

一般的なアプローチは、A の各 10ms サブジョブを独立したジョブとして扱うことです。従って、システムが始動するときその選択は、10msA または 50msB をスケジュールするかどうかです。STCF では選択肢が明確です。この場合、より短いものを選択します。次に、A のサブジョブが完了し、B のみが残され実行が開始されます。次に、A の新しいサブジョブが提出され、B をプリエンプトして 10 ミリ秒間実行されます。そうすることで、オーバーラップが可能になり、別のプロセスの I/O が完了するのを待っている間に CPU があるプロセスによって使用されます。このようにシステムをより有効に利用することができます。(図 7.9 参照)

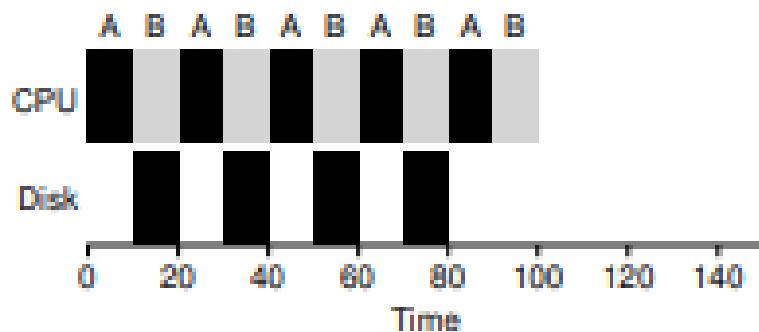


Figure 7.9: Overlap Allows Better Use of Resources

スケジューラーが I/O をどのように組み込むのかを見ていきます。スケジューラは、各 CPU バーストをジョブとして扱うことにより、「インタラクティブ」なプロセスが頻繁に実行されるようにします。これらの対話式ジョブは I/O を実行していますが、CPU を大量に使用する他のジョブが実行されるため、プロセッサーをうまく活用できます。

7.9 No More Oracle

I/O の基本的なアプローチでは、スケジューラは各ジョブの長さを知っていることを前提にしています。前にも述べたように、これはおそらく最悪の仮定である可能性があります。実際、一般的な汎用 OS(私たちが気にする OS のような)では、OS は通常、各ジョブの長さについてほとんど知りません。なので、SJF/STCF のような振る舞いを各ジョブの長さを知らずに構築するにはどうすればよいでしょうか？さらに、RR スケジューラで見たアイディアのいくつかをどのように組み込んで、応答時間も非常に良いものにすればいいでしょうか？

7.10 Summary

スケジューリングの背後にある基本的なアイデアを紹介し、2つのアプローチを開発しました。最初のジョブは最短のジョブを実行し、したがってターンアラウンドタイムを最適化します。2番目のジョブはすべてのジョブを交互に実行し、応答時間を最適化します。両方が悪いのは、一方だけが良いことです。残念なことに、システムでは一般的なトレードオフです。

私たちはまた、I/O をどのように組み込むかを図で見てきましたが、まだ問題を解決できていません。次に、最近の過去を使用して将来を予測するスケジューラを構築することによって、この問題を解決する方法を見てていきます。このスケジューラは multi-level feedback queue と呼ばれ、次の章のトピックです。

#参考文献

[B+79] “The Convoy Phenomenon”

M. Blasgen, J. Gray, M. Mitoma, T. Price

ACM Operating Systems Review, 13:2, April 1979

Perhaps the first reference to convoys, which occurs in databases as well as the OS.

[C54] “Priority Assignment in Waiting Line Problems”

A. Cobham

Journal of Operations Research, 2:70, pages 70–76, 1954

The pioneering paper on using an SJF approach in scheduling the repair of machines.

[K64] “Analysis of a Time-Shared Processor”

Leonard Kleinrock

Naval Research Logistics Quarterly, 11:1, pages 59–73, March 1964

May be the first reference to the round-robin scheduling algorithm; certainly one of the first analyses of said approach to scheduling a time-shared system.

[CK68] “Computer Scheduling Methods and their Countermeasures”

Edward G. Coffman and Leonard Kleinrock

AFIPS '68 (Spring), April 1968

An excellent early introduction to and analysis of a number of basic scheduling disciplines.

[J91] “The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling”

R. Jain

Interscience, New York, April 1991

The standard text on computer systems measurement. A great reference for your library, for sure.

[O45] “Animal Farm”

George Orwell

Secker and Warburg (London), 1945

A great but depressing allegorical book about power and its corruptions. Some say it is a critique of Stalin and the pre-WWII Stalin era in the U.S.S.R; we say it's a critique of pigs.

[PV56] "Machine Repair as a Priority Waiting-Line Problem"

Thomas E. Phipps Jr. and W. R. Van Voorhis

Operations Research, 4:1, pages 76–86, February 1956

Follow-on work that generalizes the SJF approach to machine repair from Cobham's original work; also postulates the utility of an STCF approach in such an environment. Specifically, "There are certain types of repair work, ... involving much dismantling and covering the floor with nuts and bolts, which certainly should not be interrupted once undertaken; in other cases it would be inadvisable to continue work on a long job if one or more short ones became available (p.81)."

[MB91] "The effect of context switches on cache performance"

Jeffrey C. Mogul and Anita Borg

ASPLOS, 1991

A nice study on how cache performance can be affected by context switching; less of an issue in today's systems where processors issue billions of instructions per second but context-switches still happen in the millisecond time range.

[W15] "You can't have your cake and eat it"

http://en.wikipedia.org/wiki/You_can't_have_your_cake_and_eat_it

Wikipedia, as of December 2015

The best part of this page is reading all the similar idioms from other languages. In Tamil, you can't "have both the moustache and drink the soup."

8. Scheduling: The Multi-Level Feedback Queue

この章では、マルチレベルフィードバックキュー (MLFQ) と呼ばれるスケジューリングの最もよく知られたアプローチの 1 つを開発するという課題に取り組んでいきます。マルチレベルフィードバックキュー (MLFQ) スケジューラは、Corbato et al によって生み出されました。互換性のあるタイムシェアリングシステム (CTSS) として知られているシステムで 1962 年に [C+62]、Multics と言われる OS に適応させた。後に、Corbato に最高の名誉である Turing Award を授与されました。そのスケジューラはその後、いくつかの最新のシステムでの実装に、長年にわたり改良を重ねてきました。

MLFQ が対処しようとしている基本的な問題は 2 つです。まず、ターンアラウンドタイムを最適化したいと思います。これは、前のメモで見たように、短いジョブを先に実行することによって行われます。残念ながら、OS は、SJF(または STCF) のようなアルゴリズムが必要とするジョブが実行される時間を一般に知りません。

第 2 に、MLFQ は、対話型ユーザ (すなわち、スクリーンに座って、プロセスが終了するのを待っているユーザ) に応答するシステムを、応答時間を最小限にすることをしたい。残念ながら、Round Robin のようなアルゴリズムは応答時間を短縮しますが、ターンアラウンド時間は最悪です。したがって、私たちの問題は、「一般的にプロセスについて何も知らないので、これらの目標を達成するためにスケジューラをどのように構築できますか?」ということです。スケジューラは、システムの実行中に、実行中のジョブの特性をどのように学習して、スケジューリングをより適切に行うことができるのでしょうか?

THE CRUX: HOW TO SCHEDULE WITHOUT PERFECT KNOWLEDGE?(完全な知識なしにスケジュールを設定するにはどうすればよいでしょうか?)

> インタラクティブなジョブの応答時間を最小限に抑えながら、ジョブの長さを前もって知らなくてもターンアラウンドタイムを最小限に抑えるスケジューラを設計するにはどうすればよいでしょうか?

TIP: LEARN FROM HISTORY(歴史から学ぶ)

マルチレベルフィードバックキューは、将来を予測するために過去から学習するシステムの優れた例です。このようなアプローチは、オペレーティングシステム (およびハードウェアブランチ予測子やキャッシングアルゴリズムを含む、コンピュータサイエンスの他の多くの場所) で一般的です。このようなアプローチは、ジョブが行動の段階を持ち、予測可能な場合に機能します。もちろん、それらは簡単に間違っている可能性があり、システムが全く知識なしで持っているよりも悪い決定を下さないように、そのような技術には注意を払わなければなりません。

8.1 MLFQ: Basic Rules

このようなスケジューラを構築するために、この章ではマルチレベルフィードバックキューの基本的なアルゴリズムについて説明します。多くの実装された MLFQ の詳細は異なりますが [E95]、ほとんどのアプローチは似ています。

私たちの処理では、MLFQ にはいくつかの異なるキューがあり、それぞれ異なる優先順位が割り当てられています。任意の時点で、実行準備が整っているジョブは单一のキューにいます。MLFQ は優先度を使用して、所定の時間に実行するジョブを決定します。優先度の高いジョブ (つまり、上位キューのジョブ) が選択されて実行されます。

もちろん、複数のジョブがキューに存在し、同じ優先度を持つ可能性があります。この場合、それらのジョブの間でラウンドロビンスケジューリングを使用します。

したがって、MLFQ の最初の 2 つの基本的なルールに到達します。

- ルール 1：優先度 (A) > 優先度 (B) の場合、A が実行されます (B は実行されません)。
- ルール 2：優先順位 (A) = 優先順位 (B) の場合、A と B は RR で実行されます。

したがって、MLFQ スケジューリングのカギとなる考えは、スケジューラが優先順位を設定する方法にあります。MLFQ は、各ジョブに固定の優先順位を与えるのではなく、観察された動作に基づいてジョブの優先順位を変更します。例えば、キーボードからの入力を待っている間にジョブが繰り返し CPU を放棄した場合、MLFQ は対話型プロセスがどのように動作するのかというと、優先度を高く保ちます。その代わりに、ジョブが CPU を集中的に長時間使用すると、MLFQ は優先順位を下げます。このようにして、MLFQ はプロセスが実行されるにつれてプロセスを学習しようとし、ジョブの履歴を使用して将来の動作を予測します。

与えられた瞬間に待ち行列がどのように見えるかを描写すると、次のようなものが見えます (図 8.1)。図では、2 つのジョブ (A と B) が最も高い優先順位にあり、ジョブ C は中間にあり、ジョブ D は最も低い優先順位にあります。MLFQ がどのように動作するかについての現在の知識があれば、スケジューラは A と B の間でタイムスライスを交互に切り替えるだけです。poor jobs である C と D は決して実行されないでしょう！

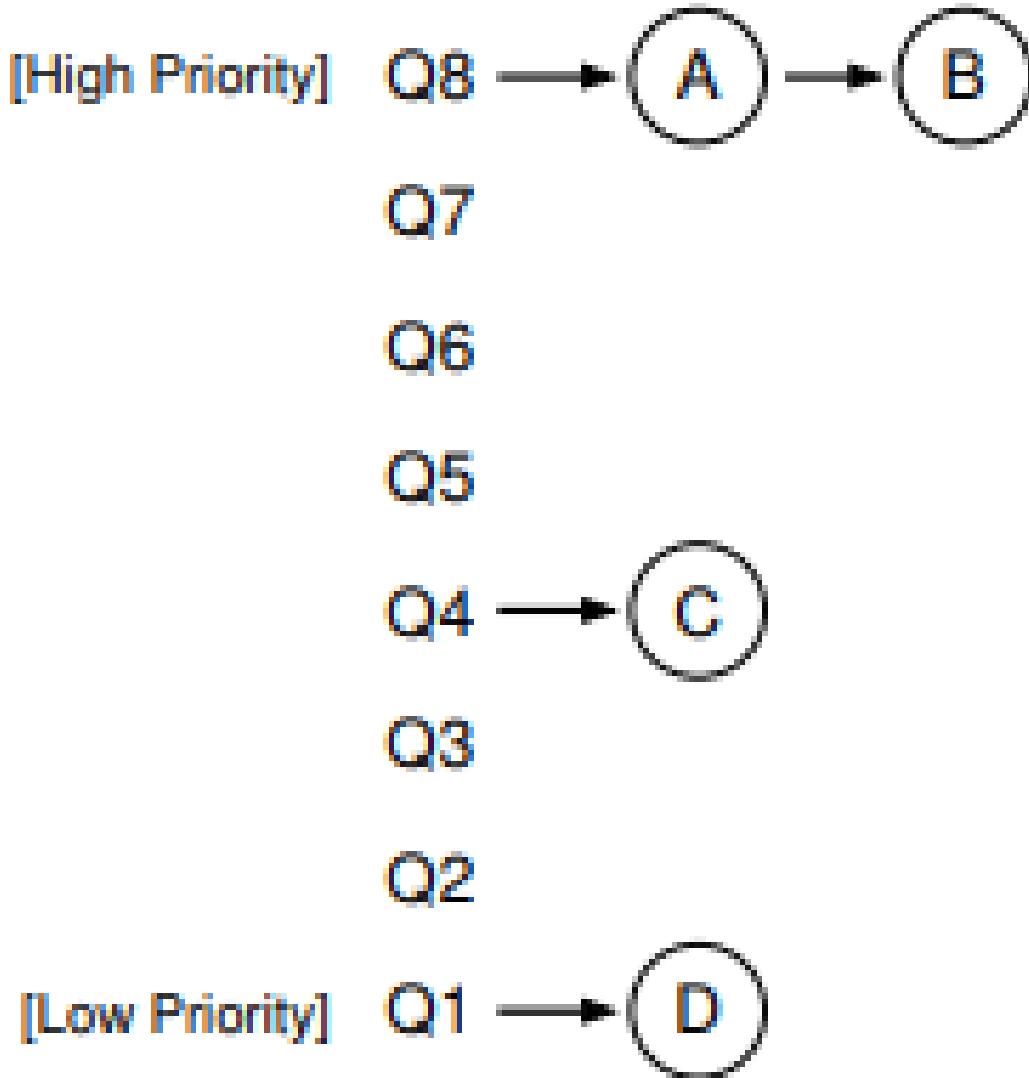


Figure 8.1: MLFQ Example

もちろん、いくつかのキューの静的なスナップショットを表示しても、MLFQ の仕組みは分かりません。私

たちが必要とするのは、時間の経過とともにどのようにジョブ優先度が変化するかを理解することです。そして、この本の章をはじめて読んでいる人にとっては驚いたことでしょう。実はこれが、次にやることなのです。

8.2 Attempt #1: How To Change Priority

我々は今、MLFQ がジョブの存続時間に応じて優先順位をどのように変更するのか（したがって、それがどのキューにあるか）を決定する必要があります。これを実行するには、短期間実行している（頻繁に CPU を解放するかもしれない）インタラクティブなジョブと、CPU 時間のかかる実行時間の長い「CPU バウンド」のジョブ応答時間は重要ではありません。優先度調整アルゴリズムでの最初の試みは次のとおりです。

- ルール 3：ジョブがシステムに入ると、ジョブは最高優先順位（一番上のキュー）に配置されます。
- ルール 4a：ジョブが実行中にタイムスライス全体を使い切った場合、ジョブの優先順位が下げられます（つまり、1つ下のキューに移動します）。
- ルール 4b：タイムスライスが立ち上がる前にジョブが CPU を放棄した場合、ジョブは同じ優先順位のままでです。

Example 1: A Single Long-Running Job

いくつかの例を見てみましょう。まず、システム内で長時間実行されているジョブがあった場合にどうなるかを見ていきます。図 8.2 は、3 つのキュースケジューラで時間の経過と共にこのジョブに何が起こるかを示しています。

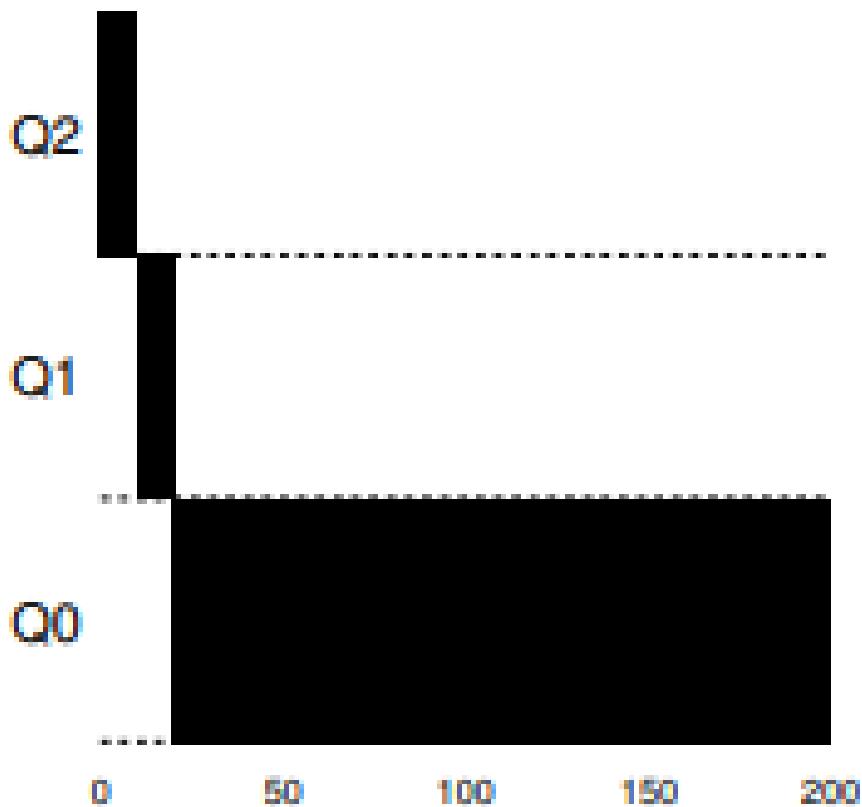


Figure 8.2: Long-running Job Over Time

この例でわかるように、ジョブは最高優先順位 (Q2) で入力します。10ms の单一のタイムスライスの後、ス

ケジューラはジョブの優先度を1つ下げる所以、ジョブはQ1上にあります。Q1でタイムスライスを実行した後、ジョブは最終的にシステム内の最も低い優先順位(Q0)に下げられ、残りはそのまま残ります。かなりシンプルでしょう？

Example 2: Along Came A Short Job

もっと複雑な例を見て、MLFQがSJFに近づこうとしていることを願ってみましょう。この例では、長時間CPUを集中的に使用するジョブであるAと短時間実行する対話式ジョブであるBの2つのジョブがあります。Aがしばらく実行されていて、Bが到着したとします。何が起こるでしょうか？MLFQはBを実行する時にSJFと似ているでしょうか？図8.3は、このシナリオの結果を示しています。A(黒で表示)は、最も優先度の低いキューで実行されています(長時間実行されるCPUインテンシブジョブと同様)。B(灰色で示されている)は時刻T=100に到着し、したがって最高の待ち行列に挿入されます。実行時間が短い(わずか20ms)ので、Bは2つのタイムスライスでボトムキューに達する前に完了します。そして、Aは実行を再開します。(低い優先度で)

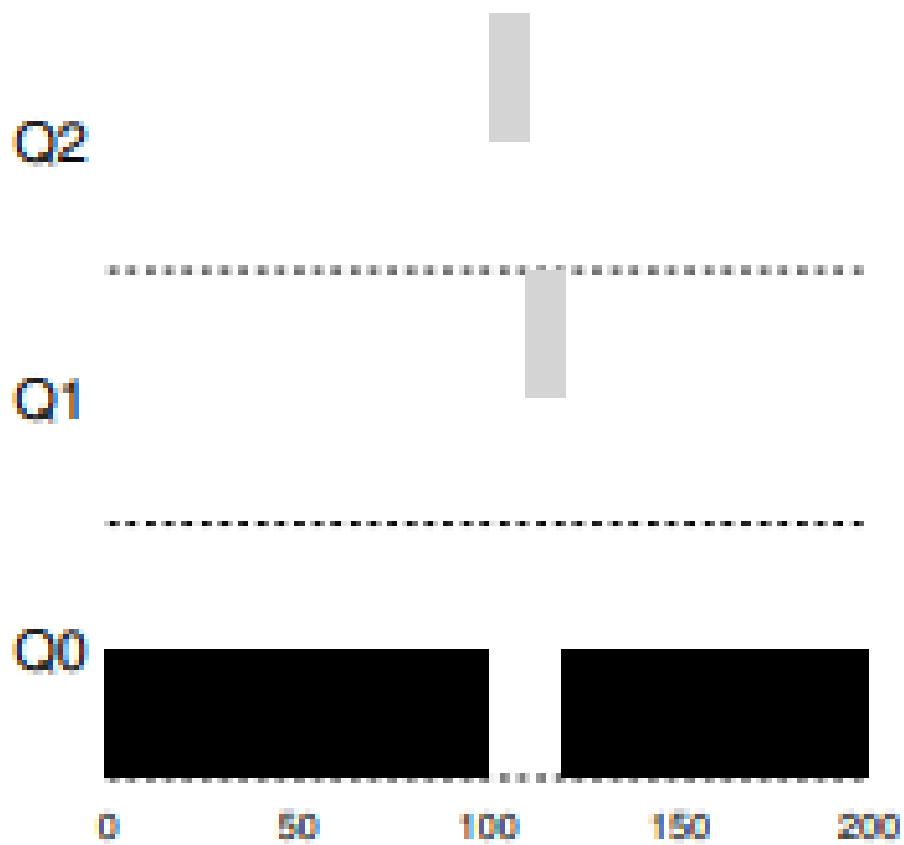


Figure 8.3: Along Came An Interactive Job

この例から、アルゴリズムの主要な目的の1つを理解できます。短期間の仕事であろうと長期間の仕事であろうと、それは短い仕事であろうと仮定しているため、ジョブの優先順位は高く与えられます。実際に短い仕事であれば、すぐに実行され完了します。短い仕事でなければ、ゆっくりとキューの下に移動していくため、すぐに、より長いバッチのようなプロセスであることがすぐに証明されます。このようにして、MLFQはSJFに似ています。

Example 3: What About I/O?

ここで、いくつかの I/O を持つ例を見てみましょう。ルール 4b が上で述べたように、プロセスがタイムスライスを使い切る前にプロセッサを放棄した場合、そのプロセスと同じ優先度に保ちます。このルールの目的は簡単です。たとえば、インタラクティブジョブが（キーボードやマウスからのユーザー入力を待つなどして）多くの I/O を実行している場合、タイムスライスが完了する前に CPU を放棄します。そのような場合、私たちはその仕事にペナルティを課すことはせず、単にジョブを同じレベルに保ちます。

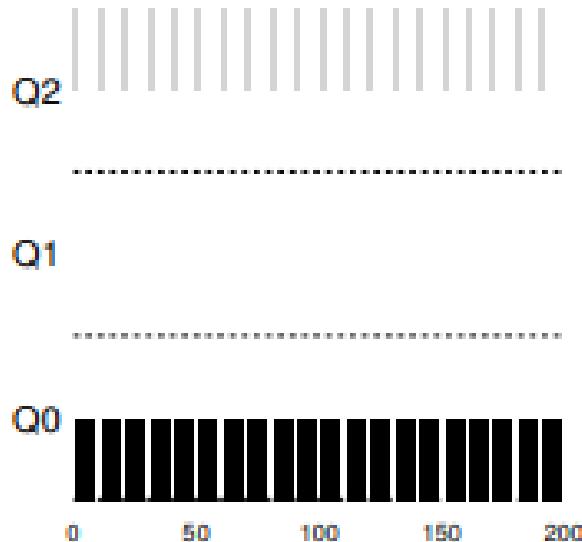


Figure 8.4: A Mixed I/O-intensive and CPU-intensive Workload

図 8.4 は、これがどのように動作するかの例を示しています。長時間実行されるバッチジョブ A(黒で表示)、I/O を実行する前に CPU を 1ms だけ必要とするインタラクティブジョブ B(灰色で表示) があります。つまり、ジョブ A とジョブ B は CPU の競合を起こす要因があります。MLFQ のアプローチは、B が CPU を解放し続けるため、B を最優先度に保ちます。B がインタラクティブジョブの場合、MLFQ はインタラクティブジョブをすばやく実行するという目標を達成します。

Problems With Our Current MLFQ

このように、我々は基本的な MLFQ を持っています。長時間実行されるジョブ間で CPU を公平に共有し、短時間または I/O 集約型のインタラクティブなジョブをすばやく実行できるように、かなり良い仕事をしているようです。しかし残念ながら、これまで開発してきたアプローチには重大な欠陥があります。あなたはどんなことを思いつくことができますか？（ちょっと立ち止まって考えてみましょう）

まず最初に、飢餓問題があります。システムにインタラクティブなジョブが多すぎると、すべての CPU 時間を消費するため、長時間実行されるジョブは決して CPU 時間が与えられません。このシナリオでも、これらの仕事について少し考えていきたいと思います。

2 番目に、頭のいいユーザは自分のプログラムをスケジューラのゲームに書き直すことができます。スケジューラーのゲームは、スケジューラーを騙して多くの利益を得ようとするずるい行為のことを一般的にさします。私たちが記述したアルゴリズムは、次の攻撃の影響を受けやすいです。タイムスライスが終了する前に、I/O 操作を発行して CPU を解放します。そうすることで、同じキューに留まることができ、CPU 時間の割合が高くなります。（CPU を放棄する前に 99 % のタイムスライスを実行するなどして）正しく実行された場合、ジョブは CPU をほぼ独占することができます。

最後に、プログラムは時間の経過とともに行動を変えるかもしれません。CPU バウンド（実行時間の中が

ジョブ) は、インタラクティブ(実行時間の短いジョブ)に移行する可能性があります。現在のアプローチでは、このようなジョブに対しては考慮されていないため、システム内の他の対話式ジョブと同様に扱われません。

8.3 Attempt #2: The Priority Boost

ルールを変更して、飢餓の問題を回避できるかどうかを見てみましょう。CPU バウンドジョブが公平に扱われることを保証するために、どうすればよいのでしょうか？ここで簡単な考え方とは、システム内のすべてのジョブの優先度を定期的に高めることです。これを達成するには多くの方法がありますが、単純なことをしましょう。一番上のキューにすべてを投げるようにします。したがって、新しいルールは以下の通りになります。

- ルール 5：ある期間 S の後、システム内のすべてのジョブを一番上のキューに移動します。

この新しいルールは、1 度に 2 つの問題を解決します。まず、プロセスが枯渢しないことが保証されます。トップキューに座ると、ジョブは CPU を他の優先順位の高いジョブとラウンドロビン方式で共有し、最終的にサービスを受け取ります。第 2 に、CPU バウンドジョブがインタラクティブになった場合、スケジューラは優先順位ブーストを受信すると、それを適切に処理します。

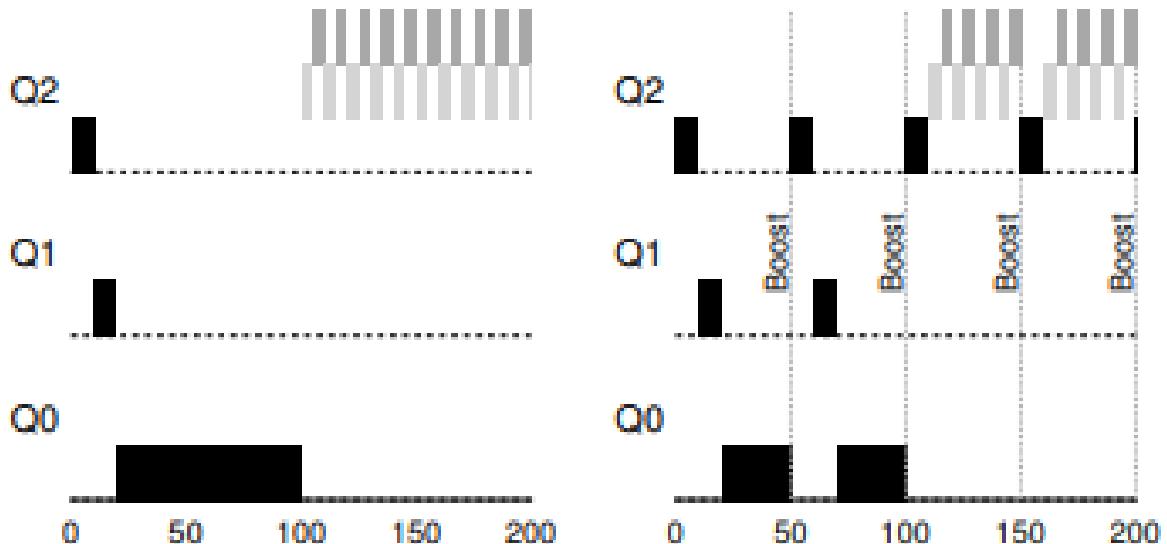


Figure 8.5: Without (Left) and With (Right) Priority Boost

例を見てみましょう。このシナリオでは、短時間実行される 2 つのインタラクティブなジョブ、長時間実行されるジョブが CPU 競合する場合の動作を示します。図 8.5 に 2 つのグラフを示します。左には優先順位の上昇はないので、2 つの短い仕事が到着すると長時間実行されるジョブは飢餓状態に陥ってしまいます。右側には、50 ミリ秒ごとに優先順位が上がります(値が小さすぎる可能性がありますが、ここでは例として使用しています)。したがって、長時間実行されるジョブが進歩し、50 ms ごとに最高の優先度を持ち、したがって定期的に実行されます。

もちろん、期間 S の追加について質問したいことがあるでしょう。それは S はどのように設定すべきですか？ということです。よく研究されているシステム研究者である John Ousterhout [O11] は、それらを正しく設定するためにある種の黒魔術を必要としていたため、システムの voo-doo 定数でそのような値を呼んでいました。残念ながら、 S にはその加減を考える必要があります。 S が高すぎると、長く実行されるジョブは飢餓状態になってしまう可能性があります。逆に低すぎると、短時間実行されるインタラクティブなジョブが CPU の適切なシェアを得られない可能性があります。

8.4 Attempt #3: Better Accounting

解決すべきもう一つの問題があります。私たちのスケジューラーのズルを防ぐ方法はどうするでしょう？あなたが推測したように、実際の原因はルール 4a と 4b であり、タイムスライスの有効期限が切れる前に CPU を放棄することでジョブが優先順位を保持できるようになります。つまりこれらに対して何をするべきでしょうか？

ここでの解決策は、MLFQ の各レベルでの CPU 時間のより良いアカウンティングを実行することです。つまり、あるレベルで使用されているプロセスをどれだけ time slice するかを忘れる代わりに、スケジューラは追跡しておく必要があります。具体的には、プロセスが CPU 割り当てを使用すると、次の優先順位のキューに降格されます。1 つの長いバーストでタイムスライスを使用するのか、小さなバーストで使用するのかは関係ありません。したがって、ルール 4a と 4b を次の單一ルールに書き換えます。

- ルール 4：ジョブが所定のレベルで time slice を使い切れば (CPU を何回諦めたかにかかわらず)、その優先順位は低下します (つまり、1 つのキューに移動します)。

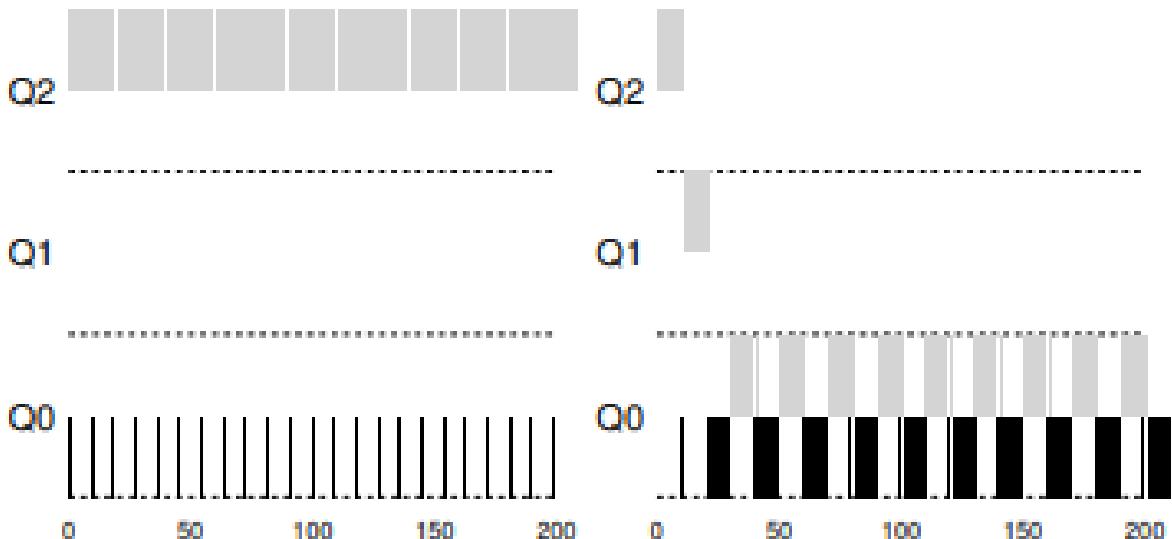


Figure 8.6: Without (Left) and With (Right) Gaming Tolerance

例を見てみましょう。図 8.6 は、古いルール 4a と 4b(左側)とズル対策をした新しいルール 4 を使用して、スケジューラを試してみると仕事量がどうなるかを示しています。タイムスライスの直前に I/O が終了し、CPU 時間を支配します。このような保護機能を使用すると、プロセスの I/O 動作に関係なく、キューレベルをゆっくりと移動させることができ、CPU の不公平な分担をになることはできません。

8.5 Tuning MLFQ And Other Issues

MLFQ スケジューリングでは、他にもいくつかの問題があります。1 つの大きな問題は、そのようなスケジューラをどのようにパラメータ化するかです。たとえば、いくつのキューが必要になるでしょうか？ タイムスライスはキューごとにどのくらいの大きさにする必要があるでしょうか？ 飢餓状態を回避し、行動変化(長いジョブから短いジョブ)があったとき、どのくらいの頻度で優先順位を上げるべきでしょうか？ これらの質問には簡単な答えはないので、仕事量の経験とそれに続くスケジューラのチューニングだけが満足できるバランスにつながっていきます。

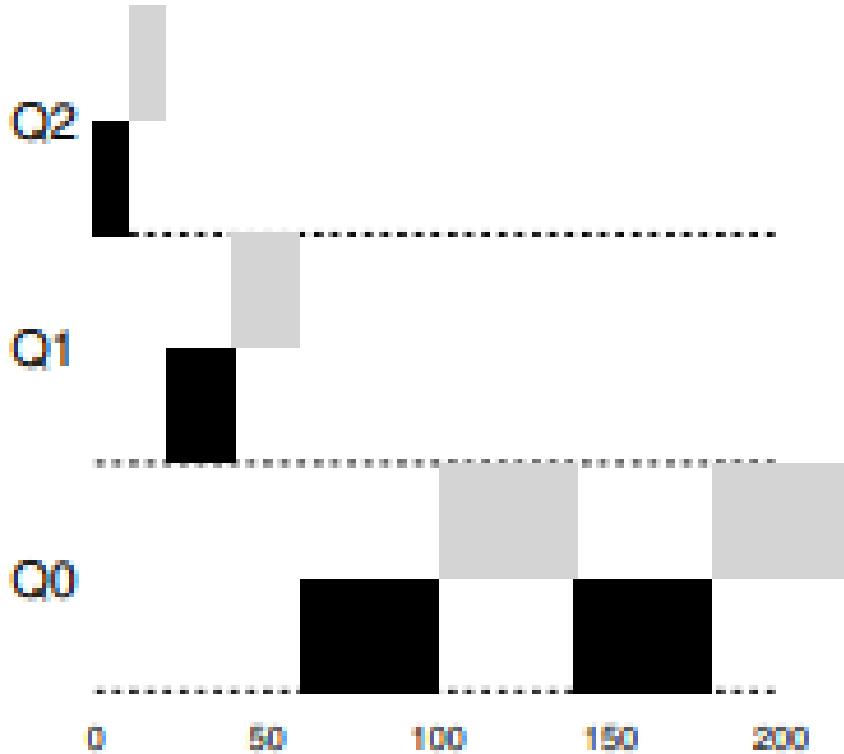


Figure 8.7: Lower Priority, Longer Quanta

例えば、ほとんどの MLFQ の変形では、異なるキュー間でタイムスライスの長さを変えることができます。高優先度キューには通常、短い時間スライスが与えられます。結局、対話型ジョブから構成され、それらの間で迅速に交互になることが合理的です。(例えば、10 ミリ秒以下)一方、低優先度キューには、CPU にバインドされた長時間実行ジョブが含まれています。したがって、より長い時間スライスがうまく機能します(例えば、100ms)。図 8.7 は、2 つの長時間実行ジョブが最高キューで 10 ミリ秒、中に 20 個、最低で 40 個実行される例を示しています。

TIP: AVOID VOO-DOO CONSTANTS (OUSTERHOUT'S LAW)

可能な限り、voo-doo 定数を避けることは良いアイデアです。残念ながら、上記の例のように困難です。システムが良い価値を学ばせるようにすることもできますが、それは簡単ではありません。そのため、設定ファイルにデフォルトのパラメータ値が入力されています。一応、何かが正常に動作していないときに熟練した管理者が微調整できます。

しかし、一般的にこれらは変更されないままにされることが多いので、デフォルトがフィールドでうまくいくでしょう。このヒントは、私たちの古い OS の教授である John Ousterhout によってもたらされました。それで、私たちはそれを Ousterhout の法則と呼んでいます。

Solaris MLFQ の実装では、Time Sharing scheduling class(TS) が特に簡単に構成できます。プロセスの優先度がライフタイム全体でどのように変更され、どのタイムスライスがどのくらいの期間、どのくらいの頻度でジョブの優先度を上昇させるかを正確に決定する一連のテーブルを提供します [AD00]。管理者はスケジューラを異なる方法で動作させるために、このテーブルを使用することができます。テーブルのデフォルト値は 60 キューであり、タイムスライスの長さは 20 ミリ秒(最優先)から数百ミリ秒(最低)までゆっくりと増加し、プライオリティは 1 秒程度でキューレベルが上昇していきます。

他の MLFQ スケジューラでは、この章で説明しているテーブルやルールは使用しません。むしろ数式を

使って優先順位を調整します。たとえば、FreeBSD スケジューラ (バージョン 4.3) は、ジョブの現在の優先度を計算するために式を使用し、プロセスが使用した CPU の量 [LM+89]に基づいています。さらに、使用時間の経過とともに減衰し、ここで記載された方法とは異なる方法で優先順位ブーストを提供します。そのような減衰利用アルゴリズムとそのプロパティ [E95] の優れた概要については、Epema の論文を参照してください。

最後に、多くのスケジューラには、あなたが今後、知るかもしれない他の機能がいくつかあります。たとえば、一部のスケジューラでは、オペレーティングシステムの作業に最も高い優先順位が設定されています。したがって、典型的なユーザジョブは、システム内で最高レベルの優先順位を得ることができません。一部のシステムでは、ユーザーのアドバイスによって優先順位を設定することもできます。たとえば、コマンドラインユーティリティ nice を使用すると、ジョブの優先度を (多少) 増減させることができます。したがって、任意の時点で実行する可能性を増減できます。詳細については、man ページを参照してください。

8.6 MLFQ: Summary

マルチレベルフィードバックキュー (MLFQ) と呼ばれるスケジューリング手法について説明しました。複数のレベルのキューがあり、フィードバックを使用して特定のジョブの優先順位を判断する理由がわかりました。歴史はそのガイドです。つまり、時間の経過とともにどのように行動するかに注意を払い、それに応じて対応していきます。

TIP: USE ADVICE WHERE POSSIBLE(可能性のあるアドバイスを使用する)

オペレーティングシステムは、システムのあらゆるプロセスに対して最適な方法は知らないため、ユーザーや管理者が OS にいくつかのヒントを提供するためのインターフェイスを提供すると便利なことがあります。OS は必ずしもそれに注意を払う必要はないので、そのようなヒントのアドバイスと呼ぶことが多いです。

しかし、より良い決定をするためにアドバイスを考慮することが必要でしょう。このようなヒントは、スケジューラ (例えば、nice)、メモリマネージャ (例えば、madvise)、およびファイルシステム (例えば、通知プリフェッчおよびキャッシング [P+95]) を含む、OS の多くの部分において有用である。

MLFQ ルールの洗練されたセットは、この章全体に広がっています。

- ルール 1：優先度 (A)> 優先度 (B) の場合、A が実行されます (B は実行されません)。
- ルール 2：優先順位 (A)= 優先順位 (B) の場合、A & B は RR で実行されます。
- ルール 3：ジョブがシステムに入ると、ジョブは最高優先順位 (一番上のキュー) に配置されます。
- ルール 4：ジョブが所定のレベルで時間割り当てを使い切れば (CPU を何回譲めたかにかかわらず)、その優先順位が下げられます (つまり、1 つのキューに移動します)。
- ルール 5：ある期間 S の後、システム内のすべてのジョブを一番上のキューに移動します。

MLFQ は次の理由から面白いです。ジョブの性質に関する前情報を要求するのではなく、ジョブの実行を観察し、それに応じて優先順位を付けます。このようにして、短期間のインタラクティブなジョブのための優れた全体的なパフォーマンス (SJF/STCF に似ています) を提供することができ、長期にわたる CPU 集約的なジョブのために公平な割り当てをします。このため、BSD UNIX 派生物 [LM+89, B86]、Solaris [M06]、および Windows NT 以降の Windows オペレーティングシステム [CS97] を含む多くのシステムが、基本スケジューラとして MLFQ の形式を使用しています。

参考文献

[AD00] “Multilevel Feedback Queue Scheduling in Solaris”

Andrea Arpac-Dusseau

Available: <http://www.ostep.org/Citations/notes-solaris.pdf>

A great short set of notes by one of the authors on the details of the Solaris scheduler. OK, we are probably biased in this description, but the notes are pretty darn good.

[B86] “The Design of the UNIX Operating System”

M.J. Bach

Prentice-Hall, 1986

One of the classic old books on how a real UNIX operating system is built; a definite must-read for kernel hackers.

[C+62] “An Experimental Time-Sharing System”

F. J. Corbato, M. M. Daggett, R. C. Daley

IFIPS 1962

A bit hard to read, but the source of many of the first ideas in multi-level feedback scheduling. Much of this later went into Multics, which one could argue was the most influential operating system of all time.

[CS97] “Inside Windows NT”

Helen Custer and David A. Solomon

Microsoft Press, 1997

The NT book, if you want to learn about something other than UNIX. Of course, why would you? OK, we’re kidding; you might actually work for Microsoft some day you know.

[E95] “An Analysis of Decay-Usage Scheduling in Multiprocessors”

D.H.J. Epema

SIGMETRICS ’95

A nice paper on the state of the art of scheduling back in the mid 1990s, including a good overview of the basic approach behind decay-usage schedulers.

[LM+89] “The Design and Implementation of the 4.3BSD UNIX Operating System”

S.J. Leffler, M.K. McKusick, M.J. Karels, J.S. Quarterman

Addison-Wesley, 1989

Another OS classic, written by four of the main people behind BSD. The later versions of this book, while more up to date, don’t quite match the beauty of this one.

[M06] “Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture”

Richard McDougall

Prentice-Hall, 2006

A good book about Solaris and how it works.

[O11] “John Ousterhout’s Home Page” John Ousterhout Available: <http://www.stanford.edu/~ouster/>
The home page of the famous Professor Ousterhout. The two co-authors of this book had the pleasure of taking graduate operating systems from Ousterhout while in graduate school; indeed, this is where the two co-authors got to know each other, eventually leading to marriage, kids, and even this book. Thus, you really can blame Ousterhout for this entire mess you’re in.

[P+95] “Informed Prefetching and Caching” R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, J. Zelenka SOSP ’95 A fun paper about some very cool ideas in file systems, including how applications

can give the OS advice about what files it is accessing and how it plans to access them

9. Scheduling: Proportional Share

この章では、proportional share scheduler(比例共有スケジューラ)と呼ばれる別のタイプのスケジューラを検討します。これは、時々 fair share schedulerとも呼ばれています。比例共有は、単純な概念に基づいています。つまり、処理時間や応答時間を最適化する代わりに、スケジューラは各ジョブがCPU時間の一定割合を取得することを保証します。

比例配分スケジューリングの優れた現代的な例は、Waldspurger and Weihl [WW94] の研究で発見され、宝くじスケジューリングと呼ばれています。しかし、この考えは確かに古いです [KL88]。基本的な考え方非常に単純です。たいていの場合、宝くじを開催して、次に実行するプロセスを決定します。より頻繁に実行されるプロセスには、宝くじに勝つチャンスが与えられます。

CRUX: HOW TO SHARE THE CPU PROPORTIONALLY(CPUを比例的に共有する方法)

比例的にCPUを共有するスケジューラを設計するにはどうすればよいでしょうか？ そうするための鍵となる仕組みは何でしょうか？ スケジューラはどれくらい効果的でしょうか？

9.1 Basic Concept: Tickets Represent Your Share

基本的な宝くじスケジューリングはの基本概念の1つです。具体的には、プロセス(またはユーザーなど)が受け取るべきリソースのシェアを表すために使用されるチケットです。プロセスが持つチケットの割合は、問題のシステムリソースのシェアを表します。

例を見てみましょう。AとBの2つのプロセスを想像してみましょう。さらにAには75のチケットがあり、Bには25しかありません。したがって、AはCPUの75%、Bは残りの25%を受け取ります。

宝くじのスケジューリングは、頻繁に(例えば、毎回のタイムスライス)毎回宝くじを開催することによって、確率論的に(決定論的ではない)決まっていきます。宝くじを開催することは簡単です。スケジューラはチケットがいくつあるかを知る必要があります(この例では100です)。その後、スケジューラは0から99までの番号の勝ちのチケットを選びます。Aが0から74、Bが75から99までのチケットを保留していると仮定すると、勝ちのチケットは単にAかBのどちらが実行されたかを判断するだけです。スケジューラは、その勝利したプロセスの状態をロードし、実行します。

TIP: USE RANDOMNESS

宝くじスケジューリングの最も美しい面の1つは、ランダム性の使用です。決定を下さなければならぬときは、ランダム化されたアプローチを使用することは堅牢で簡単な方法です。

ランダムアプローチには、従来の決定よりも少なくとも3つの利点があります。まず、ランダムはより伝統的なアルゴリズムが扱いにくいかもしれない変わったケースの振る舞いを避けます。たとえば、LRU置換ポリシー(仮想メモリの章で詳細に説明する)を考えてみましょう。多くの場合、適切な置換アルゴリズムであるにもかかわらず、LRUは、いくつかの循環して連続的に起こること(同じ事象)に対して弱いです。一方、ランダムは、そのような最悪の場合はありません。

二つ目はランダムも軽量であり、状態を追跡するためのものは必要ありません。伝統的な公平スケジューリングアルゴリズムでは、各プロセスがどれだけのCPUを受け取ったかを追跡するには、プロセスごとのアカウンティングが必要です。プロセスごとのアカウンティングは、各プロセスの実行後に更新する必要があります。これをランダムに行うと、プロセスごとの状態の最小限度(例えば、それぞれが有するチケットの数)だけが必要となります。

最後に、ランダム化はかなり速くなる可能性があります。乱数の生成が迅速であれば、決定を下すこともでき、したがって、速度が必要な多くの場所でランダムを使用することができます。ただ

し、速いほど、ランダムは疑似乱数に向かう傾向が強くなります。

宝くじスケジューラーの勝利チケットの出力例: 63 85 70 39 76 17 29 41 36 39 10 99 68 83 63 62 43 0 49
49

スケジュールの結果:

A	A A	A A A A A A	A	A A A A A A
B	B		B	B

この例からわかるように、宝くじスケジューリングでのランダム化の使用は、望ましい比率に合致する確率的正確性につながりますが、保証はありません。上記の例では、B は、望ましい 25 % の割り当てではなく、20 のタイムスライス (20 %) のうちの 4 つのみを実行します。しかし、これらの 2 つのジョブ (長い時間が必要なジョブ) が競合するほど、望ましい割合を達成する可能性が高くなります。

TIP: USE TICKETS TO REPRESENT SHARES(代表者に株式を提示するためのチケットを使用する)

抽選 (およびストライド) スケジューリングの設計における最も強力な (そして基本的な) メカニズムの 1 つは、チケットのスケジューリングです。これらの例では、チケットは CPU のプロセスシェアを表すために使用されていますが、より広範囲に適用できます。たとえば、ハイパーバイザの仮想メモリ管理に関する最近の作業では、Waldspurger はチケットを使用してゲスト OS のメモリシェア [W02] を表す方法を示しています。なので、所有権の割合を表すためのメカニズムを必要としている場合、このコンセプトはピッタリかもしれません。

9.2 Ticket Mechanisms

宝くじスケジューリングでは、チケットを異なる方法で、時には有用な方法で操作するためのメカニズムもいくつか用意されています。1 つの方法はチケット通貨のコンセプトです。通貨では、チケットのセットを持つユーザーは、自分の好きな通貨で自分の仕事にチケットを割り当てるすることができます。システムはその通貨を正しいグローバル値に自動的に変換します。

たとえば、ユーザー A と B にそれぞれ 100 個のチケットが与えられているとします。ユーザー A は 2 つのジョブ A1 と A2 を実行しており、ユーザー A の通貨でそれぞれ 500 チケット (合計 1000 通) を提供します。ユーザー B は 1 ジョブのみを実行しており、10 チケット (合計 10 件) を提供します。システムは A1 と A2 の配分を A の通貨でそれぞれ 500 からグローバル通貨でそれぞれ 50 に変換します。同様に B1 の 10 枚のチケットは 100 枚のチケットに変換されます。抽選はグローバルチケット通貨 (200 トータル) で行われ、どのジョブが実行されているかが決まります。

```
User A -> 500 (A's currency) to A1 -> 50 (global currency)
-> 500 (A's currency) to A2 -> 50 (global currency)
User B -> 10 (B's currency) to B1 -> 100 (global currency)
```

もう 1 つの便利なメカニズムはチケットの転送です。転送によって、プロセスはそのチケットを別のプロセスに一時的に渡すことができます。この機能は、クライアントプロセスがクライアントに代わって何らかの作業を要求するメッセージをサーバーに送信するクライアント/サーバー設定で特に便利です。作業をスピードアップするために、クライアントはチケットをサーバーに渡すことができるため、サーバーがクライアントの要求を処理している間にサーバーのパフォーマンスを最大化しようとします。終了すると、サーバーはチケットをクライアントに転送し、以前と同じ状態になります。

最後に、チケットのインフレは時には有用なテクニックです。インフレでは、プロセスが所有するチケット

の数を一時的に増減できます。もちろん、互いを信頼しないプロセスとの競合を起こすときは、これはほとんど意味がありません。1つの貪欲なプロセスが、膨大な数のチケットを手に入れてマシンを引き継ぐ可能性があります。どちらかというと、プロセスのグループがお互いを信頼する環境でインフレを適用することができます。このような場合、あるプロセスが CPU 時間を必要としていることが分かっている場合、そのプロセスの必要性をシステムに反映させる方法としてチケットの価値を高めることができます。

9.3 Implementation

おそらく、宝くじスケジューリングに関する最も驚くべきことは、その実装の単純さです。必要なのは、勝ちのチケットを選ぶための良い乱数ジェネレータ、システムのプロセス(例えば、リスト)を追跡するためのデータ構造、およびチケットの総数です。プロセスをリストにしておくとしましょう。ここでは、A、B、Cの3つのプロセスで構成された例を示します。各プロセスにはいくつかのチケットがあります。



スケジューリングの決定をするには、最初に、チケットの総数(400)から乱数(勝者)を選択する必要があります。次に、300という数字を選びましょう。次に、簡単なカウンターを使ってリストをトラバースすることで勝者が確定します(図 9.1)

```

1 // counter: used to track if we've found the winner yet
2 int counter = 0;
3
4 // winner: use some call to a random number generator to
5 //           get a value, between 0 and the total # of tickets
6 int winner = getrandom(0, totaltickets);
7
8 // current: use this to walk through the list of jobs
9 node_t *current = head;
10
11 // loop until the sum of ticket values is > the winner
12 while (current) {
13     counter = counter + current->tickets;
14     if (counter > winner)
15         break; // found the winner
16     current = current->next;
17 }
18 // 'current' is the winner: schedule it...
  
```

Figure 9.1: Lottery Scheduling Decision Code

コードはプロセスのリストを調べ、値が勝者を超えるまでカウンタに各チケットの値を追加します。そのようになると、現在のリスト要素が勝者になります。勝利チケットの例が300の場合、次のことが行われます。まず、Aのチケットを計上するためにカウンタを100にインクリメントします。100が300未満であるため、ループは継続します。その後、カウンターは150(Bのチケット)に更新され、まだ300未満です。最後に、カウンタは400(明らかに300を超える)に更新され、したがって、現在のC(勝者)を指してループから抜け出します。

ます。

このプロセスを最も効率的にするには、一般的に、チケットの最高数から最低数までソート順にリストを整理することが最善の方法です。順序付けはアルゴリズムの正確さには影響しません。しかし、特にほとんどのチケットを所有するプロセスがいくつかある場合は、リストの反復回数を最小限に抑えることができます。

9.4 An Example

宝くじスケジューリングの動きを理解しやすくするために、チケット数が同じ(100回)、ランタイムが同じ(R は変化します)の2つのジョブの完了時間について簡単に検討します。

このシナリオでは、各ジョブがほぼ同時に終了するようにしたいと思いますが、抽選スケジュールがランダムであるため、あるジョブが他のジョブより先に終了することがあります。この差を定量化するために、簡単な不公平度メトリック U を定義します。 U は、最初のジョブが完了した時刻を2番目のジョブが完了した時刻で割ったものです。例えば、 $R = 10$ で、第1のジョブが時間10(および第2のジョブが20)で終了する場合、 $\$ U = \text{frac}\{10\}{20} = 0.5$ 両方のジョブがほぼ同時に終了すると、 U は1に非常に近くなります。このシナリオでは、それが目標です。完全に公平なスケジューラは $U = 1$ を達成します。

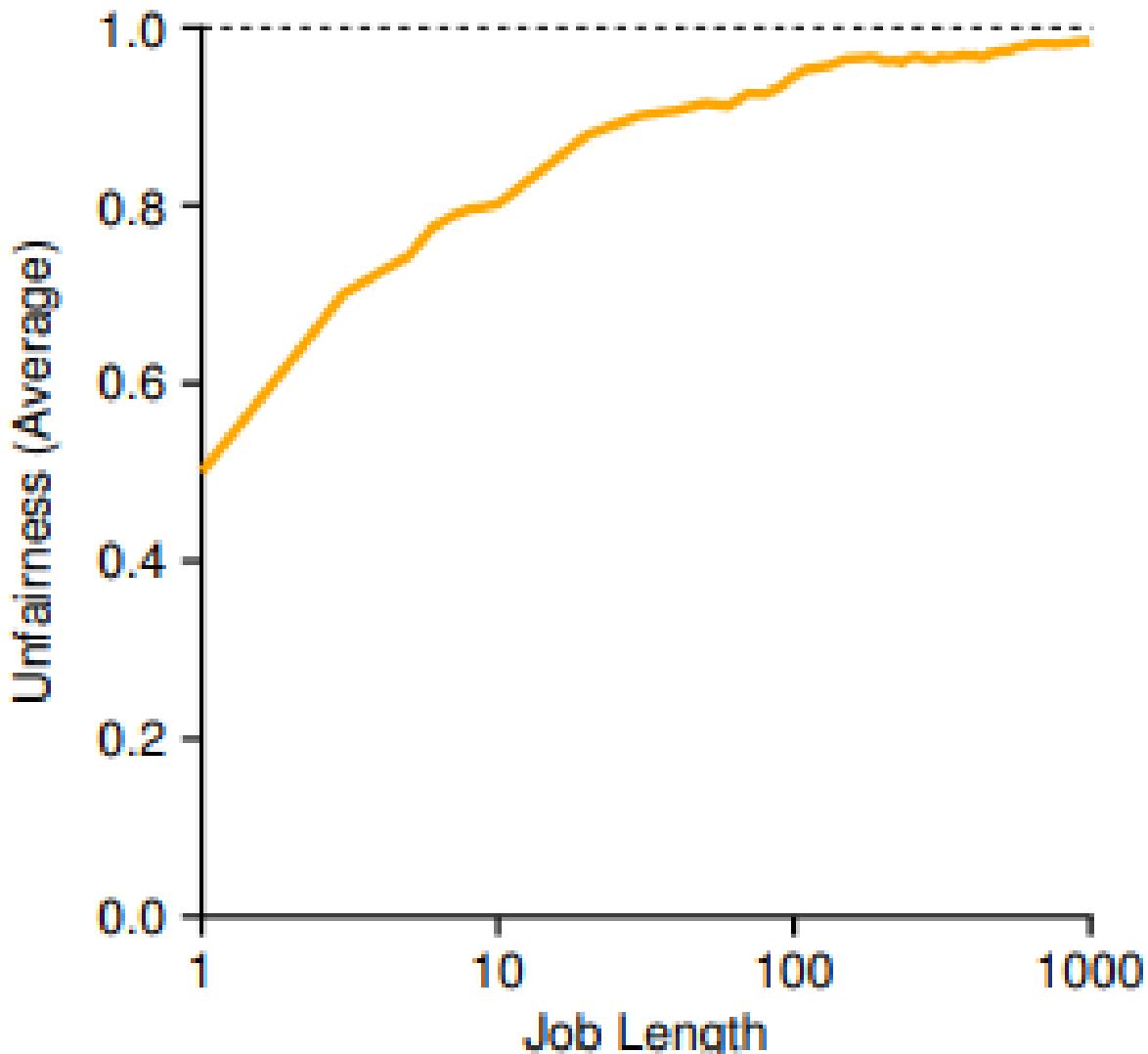


Figure 9.2: Lottery Fairness Study

図 9.2 は、2 つのジョブ (R) の長さが 30 回の試行にわたって 1 から 1000 まで変化したときの平均不公平をプロットしたものです (結果は章の最後に用意されているシミュレータを介して生成されます)。グラフからわかるように、ジョブの長さがそれほど長くない場合、平均的な不公平さが最悪です。長いタイムスライスのジョブが実行される場合にのみ、宝くじスケジューラは望んだ結果に近づきます。

9.5 How To Assign Tickets?

宝くじスケジューリングで取り上げなかった問題の 1 つは、チケットをジョブに割り当てる方法です。この問題は難しいです。もちろん、システムの動作はチケットの割り当て方法に大きく依存します。1 つのアプローチは、ユーザーが最もよく知っていると仮定することです。このような場合、各ユーザーにはいくつかのチケットが渡され、ユーザーは希望どおりに実行する任意のジョブにチケットを割り当てるすることができます。しかし、このソリューションは非効率です。実際はユーザーが何をすべきかを教えてくれないことがほとんどでしょう。そのため、「チケット割り当て問題」はいまだに残っています。

9.6 Why Not Deterministic?

疑問に思うかもしれません。なぜ乱数を使うのでしょうか？ 上の説明で見たように、ランダム化はシンプルな（そしてほぼ正しい）スケジューラをもたらしますが、時には特に短い時間を超えたスケールで正確な正しい比率を提供しないことがあります。この理由から、Waldspurger は stride scheduling を発明しました。これは確定的な fair share scheduler です [W95]。

stride scheduling も簡単です。システム内の各ジョブには stride があり、チケットの数に比例して逆になります。上記の例では、ジョブ A、B、C(それぞれ 100,50,250 枚) を使用して、各プロセスが割り当てられたチケットの数で大きな数を除算して、それぞれの歩数を計算することができます。たとえば、チケットの値ごとで 10,000 を割ると、A、B、C の stride 値は 100、200、40 になります。この値を各プロセスの stride と呼びます。プロセスが実行されるたびに、グローバルな進捗状況を追跡するために、stride によってカウンタをインクリメントします（パス値と呼ばれます）。

スケジューラは次に、ストライドとパスを使用して、次に実行するプロセスを決定します。基本的なアイデアは簡単です。いつでも、パスの値が最も低い実行プロセスを選択します。プロセスを実行すると、そのストライドによってパスカウンタがインクリメントされます。擬似コードの実装は、Waldspurger [W95] によって提供されています。

```
current = remove_min(queue); // クライアントの最小パスを取り出す
schedule(current); // リソースを使用する
current->pass += current->stride; // 現在のストライド値をパスに加算する
insert(queue, current); // キューの中に戻す
```

この例では、ストライド値が 100,200、40 の 3 つのプロセス (A、B、C) から始まり、パスの値はすべて最初は 0 です。したがって、最初はいずれかのプロセスが実行され、それらの合格値も同様に低いからです。A を選択すると仮定します（任意に、ローパス値が等しいプロセスのいずれかを選択できます）A が実行されます。タイムスライスを終了すると、パスの値は 100 に更新されます。次に、パスの値が 200 に設定された B を実行します。最後に、パスの値が 40 にインクリメントされた C を実行します。C の最小パス値を選んで実行し、パスを 80 に更新します（C のストライドは 40 です）。その後、C は再び実行され（最低のパス値）、パスは 120 に上がります。次に A が実行され、パスは 200 に更新されます（現在は B に等しくなります）。次に、C は 2 回以上実行され、パスは 160、次に 200 に更新されます。この時点で、すべてのパス値は再び等しくなり、プロセスは無期限に繰り返されます。図 9.3 は、時間の経過によるスケジューラの動作を示しています。

図からわかるように、C は 5 回、A は 2 回、B は 1 回実行された。正確には、250、100、50 というチケットの値が比例していった。ストライドスケジューリングは各スケジューリングサイクルの終わりにそれらを正

確に得ます。

ストライドスケジューリングの正確さを考えれば、どうして宝くじスケジューリングを使うのでしょうか？ 実は、宝くじのスケジューリングは、ストライドスケジューリングがないという素晴らしい特性を持っています。グローバルな状態がありません。上記のストライドスケジューリングの例の真ん中に新しい仕事が入ったとします。その合格値はどう設定しますか？ 0に設定します？ もし0に設定したとしたら、CPUを独占してしまいます。宝くじスケジューリングでは、プロセスごとにグローバルな状態はありません。どのようなチケットを持っていても新しいプロセスを追加し、一つしかないグローバル変数を更新して、合計チケット数を追跡し、そこから移動するだけです。このようにして、宝くじは新しいプロセスを合理的な方法で組み込むことをはるかに容易にします。

9.7 Summary

比例配分スケジューリングの概念を紹介し、宝くじとストライドスケジューリングという2つの実装について簡単に説明しました。宝くじは比例的なシェアを達成するための巧妙な方法でランダム化を使用します。ストライドは決定的にそうです。両方とも概念的に興味深いものですが、さまざまな理由でCPUスケジューラとして広く普及していません。

1つは、そのようなアプローチがI/O[AC97]と特によく合わないということです。もう1つは、チケットの割り当てという難しい問題を残すことです。つまり、ブラウザに割り当てられるチケットの数はどのように判断できますか？ 汎用スケジューラ（以前に議論したMLFQや他の同様のLinuxスケジューラなど）は、より優雅に機能し、より広く展開されています。

結果として、比例配分スケジューラは、これらの問題のいくつか（例えば、株式の割当）が比較的解決しやすい領域において、より有用です。たとえば、仮想化されたデータセンターでは、CPUサイクルの4分の1をWindows VMに割り当て、残りを基本Linuxインストールに割り当てることができます。比例的な共有はシンプルで効果的です。VMWareのESX Serverで今回紹介したような比例的にメモリを共有する方法については、Waldspurger[W02]を参照してください。

参考文献

- [AC97] “Extending Proportional-Share Scheduling to a Network of Workstations”
Andrea C. Arpaci-Dusseau and David E. Culler
PDPTA’97, June 1997
A paper by one of the authors on how to extend proportional-share scheduling to work better in a clustered environment.
- [D82] “Why Numbering Should Start At Zero”
Edsger Dijkstra, August 1982
<http://www.cs.utexas.edu/users/EWD/ewd08xx/EWD831.PDF>
A short note from E. Dijkstra, one of the pioneers of computer science. We’ll be hearing much more on this guy in the section on Concurrency. In the meanwhile, enjoy this note, which includes this motivating quote: “One of my colleagues — not a computing scientist — accused a number of younger computing scientists of ‘pedantry’ because they started numbering at zero.” The note explains why doing so is logical.
- [KL88] “A Fair Share Scheduler”
J. Kay and P. Lauder
CACM, Volume 31 Issue 1, January 1988

An early reference to a fair-share scheduler.

[WW94] “Lottery Scheduling: Flexible Proportional-Share Resource Management”

Carl A. Waldspurger and William E. Weihl

OSDI ’94, November 1994

The landmark paper on lottery scheduling that got the systems community re-energized about scheduling, fair sharing, and the power of simple randomized algorithms.

[W95] “Lottery and Stride Scheduling: Flexible

Proportional-Share Resource Management”

Carl A. Waldspurger

Ph.D. Thesis, MIT, 1995

The award-winning thesis of Waldspurger’s that outlines lottery and stride scheduling. If you’re thinking of writing a Ph.D. dissertation at some point, you should always have a good example around, to give you something to strive for: this is such a good one.

[W02] “Memory Resource Management in VMware ESX Server”

Carl A. Waldspurger

OSDI ’02, Boston, Massachusetts

The paper to read about memory management in VMMs (a.k.a., hypervisors). In addition to being relatively easy to read, the paper contains numerous cool ideas about this new type of VMM-level memory management.

10. Multiprocessor Scheduling (Advanced)

この章では、マルチプロセッサスケジューリングの基本について説明します。このトピックは比較的後の章で学ぶようなものが含んでいるので、並行性のトピック（つまり、本の第2の主要な「簡単な部分」）を勉強した後に、もう一度戻ってくるのが最善の方法です。

マルチプロセッサシステムは、コンピューティング分野のハイエンドでのみ長年存在していますが、ますます普及しており、デスクトップマシン、ラップトップ、さらにはモバイルデバイスにまで浸透しています。複数のCPUコアが1つのチップに集積されたマルチコアプロセッサの登場は、この普及の源です。これらのチップは、作るのは難しいですが「あまり多くの電力を使用することなく、単一のCPUをより速くすること」が現在コンピューターアーキテクトとして求められています。そして、現在私たちは皆、いくつかのCPU（優れたCPU達）をベンダーは私たちに提供してくれています。これはいいことですよね？

もちろん、1つ以上のCPUになると多くの問題が発生します。主なものは、典型的なアプリケーション（つまり、あなたが書いたCプログラム）は単一のCPUしか使用しないということです。より多くのCPUを追加しても、その単一のアプリケーションをより速く実行することはできません。この問題を解決するには、スレッドを使用して並列実行するようにアプリケーションを書き直す必要があります（詳細は第2章で詳しく説明しています）。マルチスレッド・アプリケーションは、複数のCPUにまたがって作業を分散させることができます。したがって、より多くのCPUリソースが与えられた場合、より速く実行されます。

ASIDE: ADVANCED CHAPTERS

この章では、書籍の幅広い部分を真に理解するために必要な材料があります。たとえば、マルチプロセッサスケジューリングに関するこの章では、並行処理の中で最初に読んだ後のほうが意味があります。しかし、それは論理的には本の部分（一般的に）とCPUスケジューリング（具体的には）の部分ではこの順番で合っています。従って、そのような章は順不同になるため、後でもう一度読むことをおすすめします。今回の場合は、並行性の部分の後にもう一度ここを読むことをおすすめします。

アプリケーション以外にも、オペレーティングシステムで発生する新たな問題は、マルチプロセッサスケジューリングの問題です（驚くことではありません）。ここまででは、シングルプロセッサスケジューリングの原則についていくつか議論しました。これらのアイデアをどのように拡張して複数のCPUで動作させることができますか？どんな新しい問題を克服する必要がありますか？したがって、私たちの問題は以下のようになります。

CRUX: HOW TO SCHEDULE JOBS ON MULTIPLE CPUS

OSは複数のCPU上でジョブをどのようにスケジュールするべきですか？新しい問題は何でしょうか？同じ古い技法が機能するのですか？新しいアイデアが必要ですか？

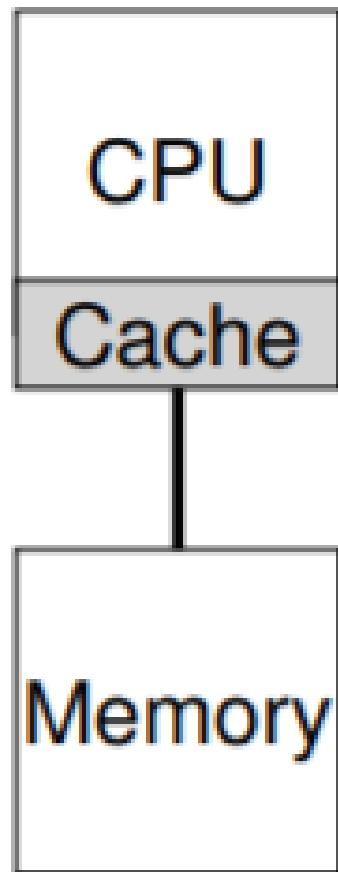


Figure 10.1: Single CPU With Cache

10.1 Background: Multiprocessor Architecture

マルチプロセッサスケジューリングを取り巻く新たな問題を理解するためには、シングル CPU ハードウェアとマルチ CPU ハードウェアの新しく根本的な違いを理解する必要があります。この違いは、ハードウェアキャッシュ（図 10.1 など）の使用と、データが複数のプロセッサでどのように共有されるかを中心にしています。今度は、この問題についてより高いレベルで議論します。詳細は他の場所 [CSG99] で利用可能であり、特に上級またはおそらく卒業生のコンピューターアーキテクチャコースです。

単一の CPU を持つシステムでは、ハードウェアキャッシュの階層があり、一般に、プロセッサがプログラムをより速く実行するのに役立ちます。キャッシュは、（一般的に）システムのメインメモリにある一般的なデータのコピーを保持する小さくて速いメモリです。対照的に、メインメモリはすべてのデータを保持しますが、この大きなメモリへのアクセスは遅くなります。頻繁にアクセスされるデータをキャッシュに保存することで、システムは大規模で低速なメモリを高速に見ることができます。

一例として、明示的なロード命令を発行してメモリから値をフェッチするプログラムと、単一の CPU のみを持つシンプルなシステムを考えます。CPU には小さなキャッシュ（たとえば 64 KB）と大きなメインメモリがあります。

プログラムが初めてこの負荷を発生させると、データはメインメモリに格納されます。したがって、フェッ

チに時間がかかります（おそらく数十ナノ秒、または数百ナノ秒）。プロセッサは、データが再使用されることが予想される場合、ロードされたデータのコピーを CPU キャッシュに格納します。プログラムがこの同じデータ項目を後で再び取り出す場合、CPU は最初にキャッシュ内のデータ項目をチェックします。もしそれが見つかると、データははるかに速く（例えば数ナノ秒）フェッチされるので、プログラムはより速く実行されます。

したがって、キャッシュは局所性の概念に基づいており、その中には時間的局所性と空間的局所性という 2 つの種類があります。時間的局所性の背後にあるアイデアは、あるデータにアクセスすると、近い将来再びアクセスされる可能性があるということです。変数や命令自体がループ内で繰り返しアクセスされることを想像してください。空間的局所性の背後にある考え方は、プログラムがアドレス x のデータ項目にアクセスすると、 x の近くのデータ項目にもアクセスする可能性があるということです。ここでは、配列を介してストリーミングされたプログラム、または順次実行される命令を考える。これらのタイプの局所性は多くのプログラムに存在するため、ハードウェアシステムは、どのデータをキャッシュに入れるかをよく推測し、うまく機能します。

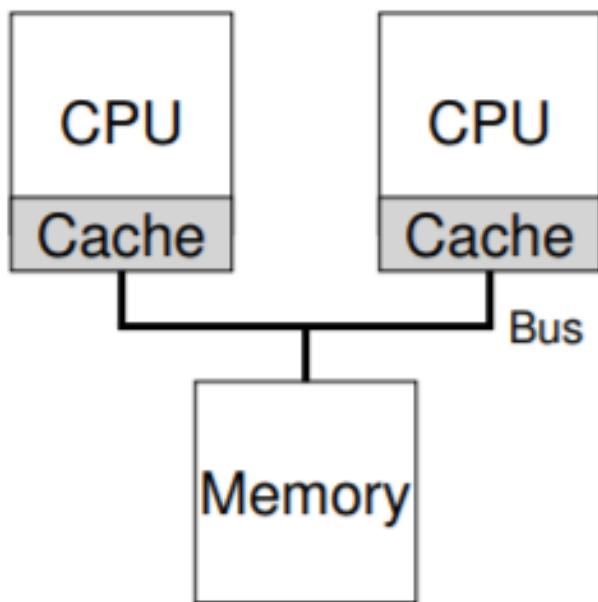


Figure 10.2: Two CPUs With Caches Sharing Memory

複雑な部分については、図 10.2 に示すように、1 つのシステムに複数のプロセッサを搭載し、1 つの共有メインメモリを使用するとどうなりますか？判明したように、複数の CPU を使ったキャッシュははるかに複雑です。

例えば、CPU1 上で動作するプログラムがアドレス A でデータ項目（値 D を持つ）を読み込むとします。データは CPU1 上のキャッシュにないので、システムはそれを主メモリから取り出し、値 D を得ます。プログラムは次にアドレス A の値を変更し、キャッシュを新しい値 D' で更新するだけです。メインメモリにデータを書き込むのが遅いため、システムは（通常）後でそれを行います。次に、OS がプログラムの実行を停止して CPU 2 に移動することを決定したと仮定します。プログラムはアドレス A の値を再読み取りします。そのようなデータ CPU2 のキャッシュは存在しないので、システムは主メモリから値を取り出し、正しい値 D' の代わりに古い値 D を得ます。…あれれ～おかしいぞ～？

この一般的な問題はキャッシュ一貫性の問題と呼ばれ、問題の解決に関わるさまざまことを記述した膨大な研究文献があります [SHW11]。ここでは、すべてのニュアンスをスキップしていくつかの大きなポイントを立てます。コンピュータ・アーキテクチャー・クラス（または 3 つ）で詳細は確認してください。

基本的な解決策は、ハードウェアによって提供されます。メモリアクセスを監視することによって、ハードウェアは基本的に「正しいこと」が起こり、单一の共有メモリのビューが保持されることを保証できます。バースペースのシステムでこれを行う方法の1つは、バススヌーピング [G83] と呼ばれる古い技術を使用することです。各キャッシュは、それらを主メモリに接続するバスを観察することによってメモリ更新に注意を払います。

CPU がキャッシュ内に保持しているデータ項目の更新を見ると、その変更を認識してそのコピーを無効にするか(つまり、それを自分のキャッシュから削除する)、または更新する(すなわち、キャッシュに新しい値を入れるあまりにも)。上記で暗示されているように、ライトバックキャッシュはこれをもっと複雑にしますが(メインメモリへの書き込みは後で見ることができないため)、基本的な仕組みがどのように機能するか想像することができます。

10.2 Don't Forget Synchronization

キャッシュが一貫性を提供するために、このすべての作業を行うとすれば、プログラム(またはOS自体)は共有データにアクセスする際に何かを心配する必要があるでしょうか? 残念なことに、答えは「はい」です。この本の第2部では、並行処理のトピックについて詳しく説明しています。ここでは詳しく説明しませんが、ここでは基本的なアイデアの一部をスケッチ/レビューします(同時実行性に慣れていると仮定します)。

CPU間で共有データ項目や構造体にアクセスする場合(特に更新する場合)、相互排他プリミティブ(ロックなど)を使用して正確性を保証する必要があります(ロックフリーのデータ構造を構築するなどの他のアプローチは複雑で、時には詳細については、並行処理に関するデッドロックの章を参照してください)。たとえば、複数のCPUで同時に共有キューにアクセスしているとします。ロックがなければ、キューから要素を同時に追加または削除することは、基本的な一貫性の定義があっても、期待どおりに機能しません。データ構造を新しい状態に原子的に更新するためのロックが必要です。

```

1  typedef struct __Node_t {
2      int             value;
3      struct __Node_t *next;
4  } Node_t;
5
6  int List_Pop() {
7      Node_t *tmp = head;           // remember old head ...
8      int value   = head->value;  // ... and its value
9      head       = head->next;    // advance head to next pointer
10     free(tmp);                // free old head
11     return value;              // return value at head
12 }
```

Figure 10.3: Simple List Delete Code

これをより具体的にするには、図10.3に示すように、共有リンクリストから要素を削除するために使用されるこのコードシーケンスを想像してください。2つのCPU上のスレッドが同時にこのルーチンに入る場合を想像してください。スレッド1が最初の行を実行すると、tmp変数にheadの現在の値が格納されます。スレッド2が最初の行を実行しても、それ自身のプライベートtmp変数に格納されているheadの値と同じになります(スタックにはtmpが割り当てられ、各スレッドに専用のプライベートストレージが割り当てられます)。したがって、各スレッドがリストの先頭から要素を削除するのではなく、各スレッドは同じヘッド要素を削除しようとします。この場合、4行目のhead要素のダブルフリー化が試行されます同じデータ値を2回返す可能性があります)。

もちろん、解決策は、ロックによって正しいルーチンを作成することです。この場合、単純なミューテックス(例えば、pthread mutex tm;)を割り当ててから、ルーチンの始めにロック(&m)を追加し、最後に

`unlock(& m)` を追加すると、問題が解決され、コードが実行されます望んだ通りに。残念なことに、わかるように、このようなアプローチは、特にパフォーマンスに関して問題がないわけではありません。具体的には、CPU の数が増えると、同期共有データ構造へのアクセスが非常に遅くなります。

10.3 One Final Issue: Cache Affinity

最後に、キャッシュアフィニティとして知られるマルチプロセッサキャッシュスケジューラを構築する際に、1つの問題が発生します。この概念は単純です。プロセスは特定の CPU 上で実行されると、CPU のキャッシュ (および TLB) に多くの状態ビットを構築します。次のプロセス実行時には、その CPU 上のキャッシュに状態の一部がすでに存在する場合に、より速く実行されるため、同じ CPU 上で実行することがかなり有効です。その代わりに毎回異なる CPU でプロセスを実行すると、プロセスが実行されたたびに状態をリロードする必要があるため、プロセスのパフォーマンスは悪化します (ハードウェアのキャッシュ一貫性プロトコルのおかげで異なる CPU 上で正しく動作することに注意してください) したがって、マルチプロセッサスケジューラは、スケジューリングの決定を行う際にキャッシュの親和性を考慮する必要があります。可能であれば、同じ CPU にプロセスを保持する方が望ましいでしょう。

10.4 Single-Queue Scheduling

この背景に基づいて、マルチプロセッサシステム用のスケジューラを構築する方法について説明します。最も基本的なアプローチは、スケジューリングが必要なすべてのジョブを1つのキューに入れることによって、単一プロセッサスケジューリングの基本フレームワークを単純に再利用することです。この singlequeue multiprocessor scheduling または SQMS を簡潔に呼んでいます。このアプローチには単純さの利点があります。次に実行する最適なジョブを選択し、複数の CPU(2つの CPU がある場合など、実行するには最適な2つのジョブを選ぶ可能性があります)で動作するような既存のポリシーが必要ありません。

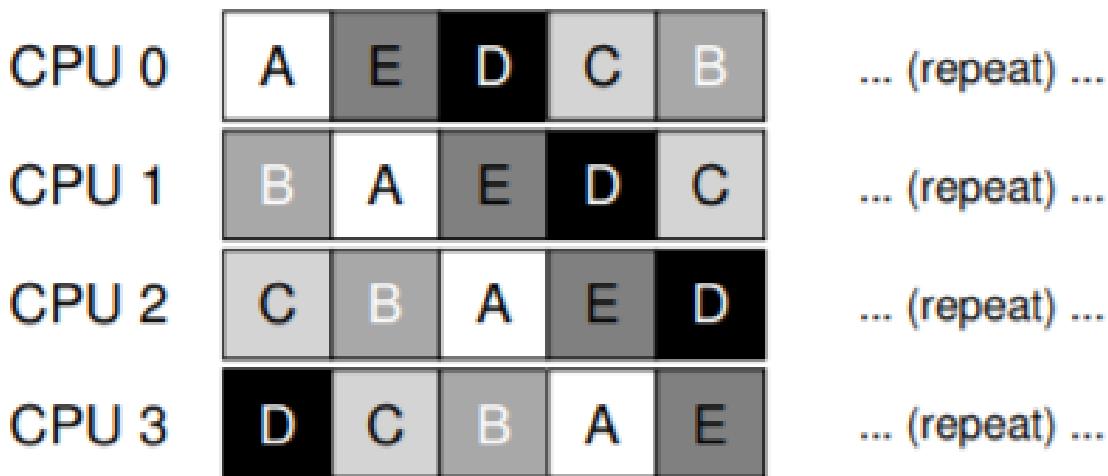
しかし、SQMS には明らかな欠点があります。第1の問題はスケーラビリティの欠如です。スケジューラが複数の CPU で正しく動作するように、開発者は上記のようにコードに何らかのロックを挿入します。ロックは、SQMS コードが单一のキューにアクセスして (たとえば、実行する次のジョブを見つけるために) 正しい結果が得られるようにします。

残念なことに、ロックはパフォーマンスを大幅に低下させる可能性があります。特に、システム内の CPU の数が増えると、パフォーマンスが低下する可能性があります [A91]。このような単一のロックの競合が増えるにつれて、システムはロックオーバーヘッドに多くの時間を費やし、システムが実行すべき作業を短時間で済ませることができます (注: 将来的にはこれを実際に測定するのは素晴らしいことです)。

SQMS の第2の主な問題は、キャッシュ親和性です。たとえば、実行する5つのジョブ (A、B、C、D、E) と4つのプロセッサがあるとします。スケジューリングキューは次のようにになります。

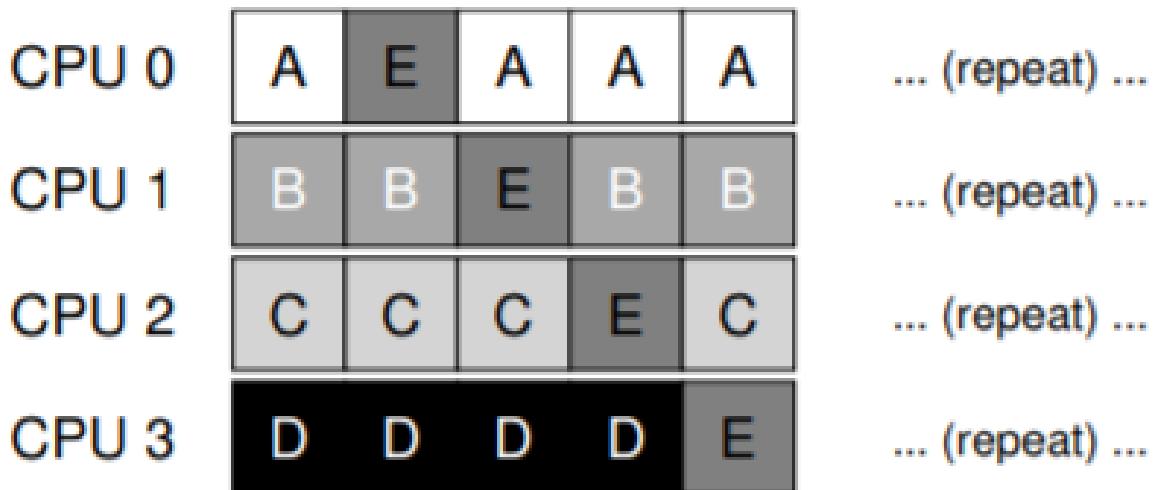


時間の経過とともに、各ジョブがタイムスライスで実行され、次に別のジョブが選択されたと仮定すると、ここでは CPU 間で可能なジョブスケジュールがあります:



各 CPU は単にグローバルに共有されたキューから実行する次のジョブを選択するため、各ジョブは CPU から CPU にバウンドして終了し、キャッシングの親和性の観点からは正反対の動作をします。

この問題を処理するために、ほとんどの SQMS スケジューラには、可能であればプロセスが同じ CPU 上で実行し続ける可能性を高めるために、ある種の親和性メカニズムが組み込まれています。具体的には、一部のジョブに対して親和性を提供するかもしれません、負荷を均衡させるために他の人を動かします。たとえば、次のようにスケジュールされた同じ 5 つのジョブを想像してみてください。



この構成では、ジョブ A から D はプロセッサ間で移動されず、ジョブ E のみが CPU から CPU に移行し、ほとんどの場合親和性が維持されます。次回に別のジョブを移行することもできます。これにより、ある種の親和性の公平性も実現します。しかしながら、そのようなスキームを実装することは複雑になります。したがって、SQMS のアプローチには長所と短所があることがわかります。既存のシングル CPU スケジューラを実装するのは簡単ですが、定義上は单一のキューしかありません。ただし、(同期のオーバーヘッドのために) 規模が大きくならず、キャッシングの親和性が容易に保持されません。

10.5 Multi-Queue Scheduling

シングルキュースケジューラで生じる問題のため、いくつかのシステムでは、複数のキューが選択されます（たとえば、CPU ごとに 1 つ）。この手法を multi queue multiprocessor scheduling（または MQMS）と呼びます。MQMS では、基本スケジューリングフレームワークは複数のスケジューリングキューで構成されています。

ます。各キューは、ラウンドロビンなどの特定のスケジューリングルールに従う可能性がありますが、もちろん他のアルゴリズムでも使用できます。ジョブがシステムに入ると、あるヒューリスティック（例えば、ランダムまたは他のジョブよりも少ないジョブを選択する）に従って、正確に1つのスケジューリング・キューに置かれる。したがって、本質的に独立してスケジュールされるため、单一キュー方式で見られる情報の共有と同期の問題は回避されます。

たとえば、CPU が2つしかないシステム（CPU 0 と CPU 1 というラベルが付いているシステム）があり、いくつかのジョブがシステムに入っているとします（A、B、C、Dなど）。各 CPU にスケジューリング・キューがある場合、OS は各ジョブを配置するキューを決定する必要があります。それは次のようなことをするかもしれません：



キューのスケジューリング方針に応じて、各 CPU は何を実行すべきかを決定する際に、2つのジョブを選択できるようになりました。たとえば、ラウンドロビンを使用すると、次のようなスケジュールが生成されることがあります。

CPU 0	A	A	C	C	A	A	C	C	A	A	C	C	...
CPU 1	B	B	D	D	B	B	D	D	B	B	D	D	...

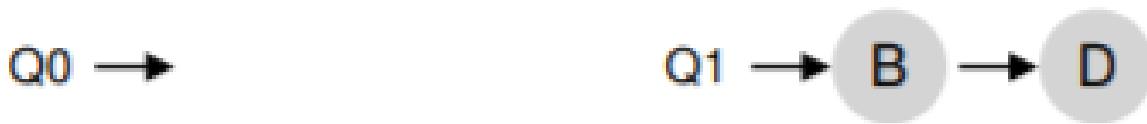
MQMS には本質的にスケーラビリティがあるという点で SQMS の明確な利点があります。CPU の数が増えるにつれて、キューの数も増えるので、ロックとキャッシュの競合が中心的な問題ではありません。さらに、MQMS は本質的にキャッシュ親和性を提供します。ジョブは同じ CPU 上に留まり、キャッシュされたコンテンツをその中に再利用する利点を得ます。しかし、注意を払っていれば、マルチ・キュー・ベースのアプローチの基本である新しい問題があることがわかります。負荷の不均衡です。上記と同じ設定（4つのジョブ、2つの CPU）があるとしますが、ジョブの1つ（Cなど）が終了したとします。これで、次のスケジューリングキューが作成されたとします。



次に、システムの各キューでラウンドロビンポリシーを実行すると、結果として得られるスケジュールが表示されます。

CPU 0	A	A	A	A	A	A	A	A	A	A	A	A	...
CPU 1	B	B	D	D	B	B	D	D	B	B	D	D	...

この図から分かるように、A は B と D の2倍の CPU を取得しますが、これは望ましい結果ではありません。さらに悪いことに、A と C の両方が終了し、ジョブ B と D だけがシステムに残っているとしましょう。スケジューリングキューは次のようにになります。



その結果、CPU 0 はアイドル状態のままでです。CPU 使用のタイムラインは悲惨です。

CPU 0



なので、貧弱な MQMS は何をすべきでしょうか？負荷の不均衡という問題をどのように克服すればよいでしょうか？

CRUX: HOW TO DEAL WITH LOAD IMBALANCE

MQMS は、ロード・インバランスをどのように処理して、望ましいスケジューリング目標をより良く達成する必要がありますか？

このクエリーに対する明白な答えは、ジョブを移動することです。これは、マイグレーションとも呼ばれる手法です。1つのCPUから別のCPUへジョブを移行することにより、真のロードバランスを達成できます。明快さを加えるためにいくつかの例を見てみましょう。もう一度、1つのCPUがアイドル状態で、もう1つが何らかのジョブを持っている状況があります。



この場合、必要な移行は容易に理解できます。つまり、OS は B または D のいずれかを CPU 0 に単に移動する必要があります。この単一のジョブの移行結果は均等に均等に分散され、誰もが満足しています。以前の例では、A が CPU 0 に、B と D が CPU 1 に交互に置かれていたより複雑なケースが発生します。



もちろん、可能な他の多くの移行パターンが存在します。しかし、難しい部分が存在します。それは、システムはどのようにしてそのような移行を制定するべきですか？という問題です。1つの基本的なアプローチは、ワーク・スタイル [FLR98] として知られる技法を使用することです。ワーク・スタイルのアプローチでは、ジョブが少ない（ソース）キューは、別の（ターゲット）キューを時々見て、そのキューがどれくらい完全であるかを確認します。ターゲットキューが（特に）ソースキューよりもいっぱいの場合、ソースはターゲットキューからの1つ以上のジョブを「盗み」、負荷のバランスをとるのに役立ちます。

もちろん、そのようなアプローチには自然な負荷があります。あまりにも頻繁に他のキューを見回すと、オーバーヘッドが大きくなり、スケーリングに問題が発生します。これは、最初に複数のキューを実装する目的全体です。一方、他のキューを非常に頻繁に見ないと、重大な負荷の不均衡に苦しむ危険があります。システムポリシーの設計でよく見られるように、正しい閾値を見つけることは、黒魔術です。

10.6 Linux Multiprocessor Schedulers

興味深いことに、Linux コミュニティでは、マルチプロセッサスケジューラを構築するための共通の解決方法はありませんでした。時間の経過とともに、O(1) スケジューラ、CFS(Complete Fair Scheduler)、BFS スケジューラ (BFS) という 3 つの異なるスケジューラが発生しました。スケジューラの長所と短所の優れた概要については、Meehean の論文を参照してください [M11]。ここでは基本のいくつかを要約します。

O(1) と CFS は複数のキューを使用しますが、BFS は単一のキューを使用し、両方のアプローチが成功することを示しています。もちろん、これらのスケジューラを分ける他の多くの詳細があります。例えば、O(1) スケジューラは、様々なスケジューリング目標を達成するために、時間の経過とともにプロセスの優先度を変更し、次に優先度の高いスケジューラ (前述の MLFQ と同様) である。インタラクティビティは特に重要です。これとは対照的に、CFS は決定論的な比例的なアプローチ (前述のように、ストライドスケジューリングに似ています) です。BFS は、3 つのキュー間の唯一のキュー方式でもあり、比例配分ですが、EEVDF(EEVDVD)[SA96] というより複雑な方式に基づいています。これらの近代的なアルゴリズムの詳細については、お読みください。あなたは彼らが今やっていることを理解することができるはずです！

10.7 Summary

マルチプロセッサスケジューリングにはさまざまなアプローチがあります。シングルキュアアプローチ (SQMS) はロードバランスを取るように構築するのは簡単ですが、本質的には多くのプロセッサとキャッシュの親和性にスケーリングすることが難しいです。マルチキュー方式 (MQMS) は、スケーラビリティとキャッシュアフィニティをうまく処理しますが、負荷の不均衡に問題があり、より複雑です。どのようなアプローチをとっても、簡単な答えはありません。一般的なスケジューラを構築するのは難しい作業です。小さなコード変更は大きな動作上の違いにつながります。

#参考文献

[A90] “The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors”

Thomas E. Anderson

IEEE TPDS Volume 1:1, January 1990

A classic paper on how different locking alternatives do and don't scale. By Tom Anderson, very well known researcher in both systems and networking. And author of a very fine OS textbook, we must say.

[B+10] “An Analysis of Linux Scalability to Many Cores Abstract”

Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, Nickolai Zeldovich

OSDI '10, Vancouver, Canada, October 2010

A terrific modern paper on the difficulties of scaling Linux to many cores.

[CSG99] “Parallel Computer Architecture: A Hardware/Software Approach”

David E. Culler, Jaswinder Pal Singh, and Anoop Gupta Morgan Kaufmann, 1999

A treasure filled with details about parallel machines and algorithms. As Mark Hill humorously observes on the jacket, the book contains more information than most research papers.

[FLR98] “The Implementation of the Cilk-5 Multithreaded Language”

Matteo Frigo, Charles E. Leiserson, Keith Randall

PLDI '98, Montreal, Canada, June 1998

Cilk is a lightweight language and runtime for writing parallel programs, and an excellent example of the work-stealing paradigm.

[G83] “Using Cache Memory To Reduce Processor-Memory Traffic”

James R. Goodman

ISCA '83, Stockholm, Sweden, June 1983

The pioneering paper on how to use bus snooping, i.e., paying attention to requests you see on the bus, to build a cache coherence protocol. Goodman's research over many years at Wisconsin is full of cleverness, this being but one example.

[M11] "Towards Transparent CPU Scheduling"

Joseph T. Meehean

Doctoral Dissertation at University of Wisconsin—Madison, 2011

A dissertation that covers a lot of the details of how modern Linux multiprocessor scheduling works. Pretty awesome! But, as co-advisors of Joe's, we may be a bit biased here.

[SHW11] "A Primer on Memory Consistency and Cache Coherence"

Daniel J. Sorin, Mark D. Hill, and David A. Wood

Synthesis Lectures in Computer Architecture

Morgan and Claypool Publishers, May 2011

A definitive overview of memory consistency and multiprocessor caching. Required reading for anyone who likes to know way too much about a given topic.

[SA96] "Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation"

Ion Stoica and Hussein Abdel-Wahab

Technical Report TR-95-22, Old Dominion University, 1996

A tech report on this cool scheduling idea, from Ion Stoica, now a professor at U.C. Berkeley and world expert in networking, distributed systems, and many other things.

13 The Abstraction: Address Spaces

初期のコンピュータシステムの構築は簡単でした。なぜでしょうか？ 実はユーザーはあまり期待していませんでした。「使いやすさ」、「高性能」、「信頼性」など、これらの頭痛に本当に悩まされているのは、期待しているユーザーです。

13.1 Early Systems

メモリの観点から、初期のマシンはユーザーにあまり抽象的なものを提供しませんでした。基本的には、マシンの物理メモリは図 13.1 のようになりました。

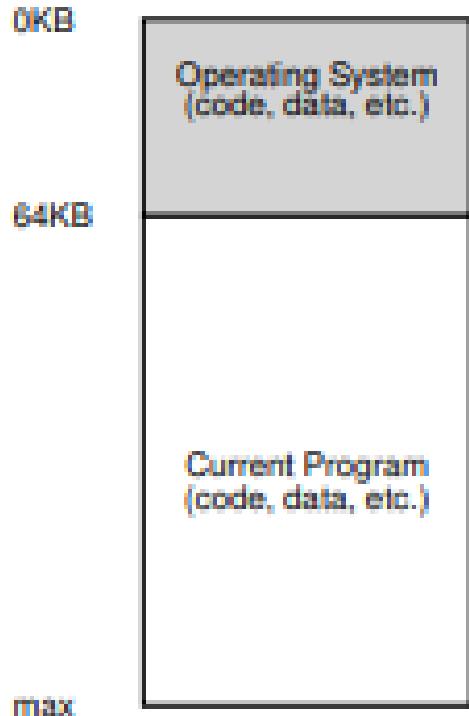


Figure 13.1: Operating Systems: The Early Days

OS はメモリにあるルーチン (実際にはライブラリである) であり、物理メモリにある実行中のプログラム (プロセス) が 1 つあります。この例では 64k)、残りのメモリを使用しました。ここには錯覚はほとんどなく、ユーザーは OS からあまり期待していませんでした。当時の OS 開発者にとって、設計は簡単だったでしょうか？

13.2 Multiprogramming and Time Sharing

しばらくすると、マシンが高価だったため、人々はマシンをより効果的に共有し始めました。このように、マルチプログラミングの時代は、複数のプロセスがある時点で動作する準備が整った状態で生まれました [DV66]。そして、例えば、I/O を実行することが決定されたときなど、OS はそれらの間で切り替わります。そうすることで、CPU の有効利用が増えました。このような効率の向上は、各マシンのコストが数十万ドルから数百万ドルに達したときに特に重要でした (Mac が高価だと思っていました)。

まもなく、人々はより多くの機械を要求し始め、time sharing の時代が生まれました [S59, L60, M62,

M83]。具体的には、バッチ・コンピューティングの限界を認識していました。特に、プログラム・デバッグ・サイクルに長い時間がかかっていたため、プログラム・デバッグ・サイクルに疲れてしまったプログラマー自身にとっては限界がありました [CV65]。多くのユーザーが現在実行中のタスクからタイムリーな応答を待っている（または望んでいる）マシンを同時に使用している可能性があるため、対話性の概念が重要になりました。

タイムシェアリングを実装する 1 つの方法は、すべてのメモリ（図 13.1、ページ 113）に完全にアクセスできるようにしながら、1 つのプロセスを実行してから停止し、すべての状態を何らかの種類のディスク（すべての物理メモリ）を読み込み、他のプロセスの状態をロードし、しばらく実行して、マシンの粗い共有を実行します。[M+63]

残念ながら、このアプローチには大きな問題があります。それは遅すぎます。特にメモリが増えるほどです。レジスタレベルの状態（PC、汎用レジスタなど）の保存と復元は比較的高速ですが、メモリの内容全体をディスクに保存することは残念ながら効果がありません。したがって、私たちがやりたいことは、プロセスをメモリに置き換えながらプロセスを残し、OS が効率的に時間を共有できるようにすることです（図 13.2）。

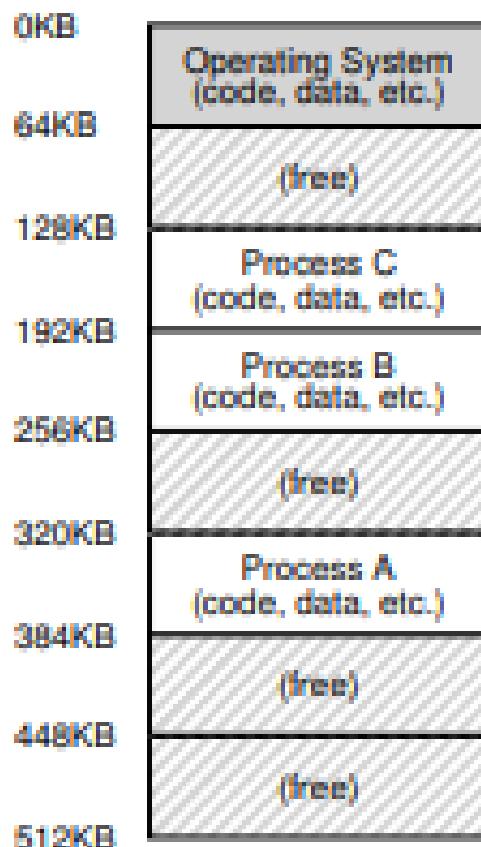


Figure 13.2: Three Processes: Sharing Memory

この図には 3 つのプロセス（A、B、C）があり、それぞれに 512KB の物理メモリの一部が割り当てられています。単一の CPU を想定すると、OS はプロセスの 1 つ（たとえば A）を実行することを選択し、他のプロセス（B および C）は実行待ちのレディキューに入ります。

time slice が普及するにつれて、おそらくオペレーティングシステムに新しい要求が加えられたと推測できます。特に、複数のプログラムを同時にメモリに常駐させることで、重要な問題が保護されます。プロセスが他のプロセスのメモリを読み書きできるようにしたり、悪化させたりすることは望ましくありません。

13.3 The Address Space

しかし、私たちはこれらの厄介なユーザーを念頭に置いておく必要があります。そのためには、OS が物理メモリーの使いやすい抽象化を作成する必要があります。この抽象概念はアドレス空間と呼ばれ、実行中のプログラムのシステム内のメモリのビューです。メモリのこの基本的な OS 抽象化を理解することは、メモリがどのように仮想化されているかを理解するうえで重要です。

プロセスのアドレス空間には、実行中のプログラムのすべてのメモリ状態が含まれます。例えば、プログラムのコード（命令）はどこかのメモリになければならないため、アドレス空間にあります。プログラムが実行されている間、スタックを使用して関数呼び出しチェーン内のどこにあるかを追跡し、ローカル変数を割り当て、ルーチンとの間でパラメータと戻り値を渡します。最後に、ヒープは、C で `malloc()` を呼び出したときや C++ や Java などのオブジェクト指向の言語で新しいときなど、動的に割り当てられたユーザー管理のメモリーに使用されます。もちろん、他にも静的に初期化された変数などがありますが、ここではコード、スタック、ヒープの 3 つのコンポーネントを想定します。

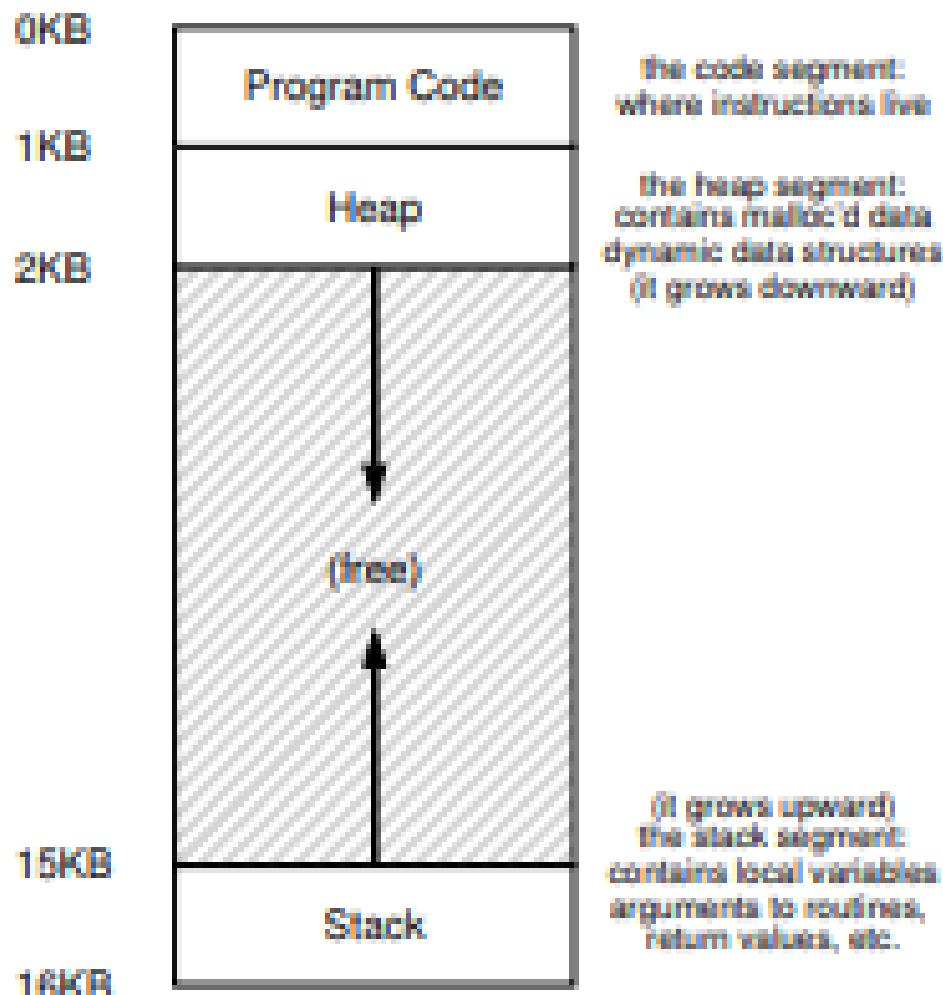


Figure 13.3: An Example Address Space

図 13.3(115 ページ) の例では、小さなアドレス空間 (16KB のみ) があります。プログラムコードは、アドレ

ス空間の先頭にあります(この例では0から始まり、アドレス空間の最初の1Kにパックされています)。コードは静的であり(したがってメモリに配置するのが簡単なので)、アドレス空間の先頭に配置することができ、プログラムが実行されるにつれてそれ以上のスペースは必要ないことがわかります。

次に、プログラムが実行されている間に拡大(縮小)するアドレス空間の2つの領域があります。それらはヒープ(上部)とスタック(下部)です。それぞれが成長したいと思っているため、アドレス空間の両端に配置することで、このような成長を可能にします。逆方向に成長するだけです。したがって、ヒープはコードの直後(1KB)から始まり、下向きに成長します(ユーザーがmalloc()により多くのメモリを要求したとき)。スタックは16KBから始まり、上に向かって(ユーザが手続き呼び出しを行ったときなどに)成長します。ただし、このスタックとヒープの配置は単なる規則です。もし望むのであれば、アドレス空間を別の方法で整理することができます(後で、出てきますが、複数のスレッドがアドレス空間に共存するとき、アドレス空間を分ける良い方法はもうありません)。

もちろん、アドレス空間について説明するとき、OSが実行中のプログラムに提供している抽象概念が記述されています。プログラムは実際には物理アドレス0~16KBのメモリにありません。むしろ、任意の物理アドレス(複数可)にロードされます。図13.2のプロセスA、B、Cを調べます。そこでは、各プロセスが異なるアドレスのメモリにどのようにロードされているかを見ることができます。したがって、問題は以下のようになります。

THE CRUX: HOW TO VIRTUALIZE MEMORY

OSは、単一の物理メモリの上に複数の実行中のプロセス(すべての共有メモリ)のプライベートな、潜在的に大きなアドレス空間のこの抽象化をどのように構築できますか?

OSがこれを実行すると、実行中のプログラムは特定のアドレス(例えば0)でメモリにロードされ、潜在的に非常に大きなアドレス空間(32ビットまたは64ビットなど)を持つと考えているため、現実は全く異なっています。

たとえば、図13.2のプロセスAがアドレス0(仮想アドレスと呼ぶ)で負荷を実行しようとすると、何らかのハードウェアサポートと並行して、OSは実際には負荷が実際には発生しないことを確認する必要があります。物理アドレス0に行くのではなく、物理アドレス320KB(Aはメモリにロードされます)に行きます。これは、世界のあらゆる最新のコンピュータシステムの基礎をなすメモリの仮想化の鍵です。

TIP: THE PRINCIPLE OF ISOLATION

信頼性の高いシステムを構築する上で、分離は重要な原則です。2つのエンティティが互いに適切に分離されている場合、これは一方が他方に影響を与えずに失敗することができます。オペレーティングシステムは、プロセスをお互いに分離するように努力し、このようにして、他のプロセスを害するのを防止します。メモリ分離を使用することにより、OSは、実行中のプログラムが基礎となるOSの動作に影響を与えないことをさらに保証します。最近のOSの中には、OSの他の部分からOSの一部を切り離すことによって、さらに孤立しているものもあります。したがって、このようなマイクロカーネル[BH70, R+89, S+03]は、典型的なモノリシックカーネル設計よりも高い信頼性を提供することができます。

13.4 Goals

このように、私たちはOSの仕事に、メモリを仮想化するというこの一連のノートに着きます。OSはメモリを仮想化するだけでなく、それはスタイルでそうするでしょう。OSがそうしていることを確認するには、私たちを導くいくつかの目標が必要です。私たちは前にこれらの目標を見てきましたが(序論を考えてください)、我々はそれらをもう一度見ていきますが、確かに繰り返す価値があります。

仮想メモリ(VM)システムの1つの主な目標は透明性です。OSは実行中のプログラムには見えない方法で仮想メモリを実装する必要があります。したがって、プログラムはメモリが仮想化されているという事実を認識すべきではありません。むしろ、プログラムはあたかも独自の物理メモリを持つかのように動作します。背後では、OS(およびハードウェア)は、さまざまなジョブ間でメモリを多重化するためのすべての作業を行い、したがって錯覚を実装します。

VMのもう1つの目標は効率です。OSは、時間の点(すなわち、プログラムをはるかにゆっくりと実行させない)、およびスペース(すなわち、仮想化をサポートするために必要な構造に対してあまりにも多くのメモリを使用しない)の両方において、仮想化ができるだけ効率的にするよう努力すべきです。時間効率の高い仮想化を実現するには、TLBなどのハードウェア機能を含むハードウェアサポートにOSを依存させる必要があります(当然のことながらこれについて学びます)。

最後に、第3のVM目標は保護です。OSはプロセス同士を保護し、OS自体をプロセスから保護する必要があります。1つのプロセスがロード、ストア、または命令フェッチを実行する場合、他のプロセスまたはOS自体のメモリ内容(つまり、アドレス空間外のもの)には何らかの形でアクセスまたは影響を与えるべきではありません。したがって、保護はプロセス間の分離の性質を提供することを可能にします。各プロセスは、他の障害のあるプロセスや悪意のあるプロセスから安全な、独自の隔離された繭で実行されている必要があります。

ASIDE: EVERY ADDRESS YOU SEE IS VIRTUAL

ポインタを出力するCプログラムを書いたことがありますか？表示されている値(大きい値、16進数で表示されることが多い)は、仮想アドレスです。あなたのプログラムのコードがどこにあるのだろうか？あなたもそれを印刷することができます。はい、print文で出力することができれば、それは仮想アドレスです。実際、ユーザーレベルのプログラムのプログラマーとして見えるアドレスはすべて仮想アドレスです。メモリを仮想化するトリッキーなテクニックによって、これらの命令とデータ値がマシンの物理メモリのどこにあるのかを知るのは、OSだけです。プログラムのアドレスをプリントアウトすると、それは仮想的なものです。物事がメモリにどのようにレイアウトされているかの錯覚です。OS(およびハードウェア)だけが本当の真実を知っています。

`main()`ルーチン(コードが存在する場所)、`malloc()`から返されたヒープ割り当て値の値、スタック上の整数の位置を出力する小さなプログラムです。

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    printf("location of code : %p\n", (void *) main);
    printf("location of heap : %p\n", (void *) malloc(1));
    int x = 3;
    printf("location of stack : %p\n", (void *) &x);
    return x;
}
```

64bit Macで動かしたときは以下のようになります。

```
location of code : 0x1095afe50
location of heap : 0x1096008c0
location of stack : 0x7fff691aea64
```

このことから、コードがアドレス空間で最初に来てからヒープになり、スタックがこの大きな仮想空間の反対側にあることが分かります。これらのアドレスはすべて仮想アドレスであり、実際の物理ロケーションから

値を取得するために OS とハードウェアによって変換されます。

次の章では、ハードウェアとオペレーティングシステムのサポートを含め、メモリを仮想化するために必要な基本的なメカニズムについて説明します。また、空き容量を管理する方法や、空き容量が少なくなったときにメモリから解放されるページなど、オペレーティングシステムで発生するより関連性の高いポリシーのいくつかについても調査します。そうすることで、現代の仮想メモリシステムが実際にどのように機能するかを理解していきます。

13.5 Summary

私たちは主要な OS サブシステム、仮想メモリの導入を見てきました。VM システムは、その命令およびデータをすべてその中に保持する、プログラムに大きな、まばらな、プライベートなアドレス空間の錯覚を提供する責任があります。いくつかの重要なハードウェアの助けを借りて、OS はこれらの仮想メモリ参照をそれぞれ取り出し、それらを物理アドレスに変換して、物理メモリに提示して必要な情報を取り出すことができます。OS はこれを、多くのプロセスで同時に実行し、プログラムを互いに保護し、OS を保護するようにします。全体的なアプローチには、多くの仕組み（低レベルマシンがたくさんあります）と、いくつかの重要なポリシーが必要です。重要なメカニズムを最初に説明して、下から上へと学んでいきます。

参考文献

- [BH70] “The Nucleus of a Multiprogramming System”
Per Brinch Hansen
Communications of the ACM, 13:4, April 1970
The first paper to suggest that the OS, or kernel, should be a minimal and flexible substrate for building customized operating systems; this theme is revisited throughout OS research history.
- [CV65] “Introduction and Overview of the Multics System”
F. J. Corbato and V. A. Vyssotsky
Fall Joint Computer Conference, 1965
A great early Multics paper. Here is the great quote about time sharing: “The impetus for time sharing first arose from professional programmers because of their constant frustration in debugging programs at batch processing installations. Thus, the original goal was to time-share computers to allow simultaneous access by several persons while giving to each of them the illusion of having the whole machine at his disposal.”
- [DV66] “Programming Semantics for Multiprogrammed Computations”
Jack B. Dennis and Earl C. Van Horn
Communications of the ACM, Volume 9, Number 3, March 1966
An early paper (but not the first) on multiprogramming.
- [L60] “Man-Computer Symbiosis”
J. C. R. Licklider
IRE Transactions on Human Factors in Electronics, HFE-1:1, March 1960
A funky paper about how computers and people are going to enter into a symbiotic age; clearly well ahead of its time but a fascinating read nonetheless.
- [M62] “Time-Sharing Computer Systems”
J. McCarthy
Management and the Computer of the Future, MIT Press, Cambridge, Mass, 1962

Probably McCarthy's earliest recorded paper on time sharing. However, in another paper [M83], he claims to have been thinking of the idea since 1957. McCarthy left the systems area and went on to become a giant in Artificial Intelligence at Stanford, including the creation of the LISP programming language. See McCarthy's home page for more info: <http://www-formal.stanford.edu/jmc/>

[M+63] "A Time-Sharing Debugging System for a Small Computer"

J. McCarthy, S. Boilen, E. Fredkin, J. C. R. Licklider

AFIPS '63 (Spring), New York, NY, May 1963

A great early example of a system that swapped program memory to the "drum" when the program wasn't running, and then back into "core" memory when it was about to be run.

[M83] "Reminiscences on the History of Time Sharing"

John McCarthy

Winter or Spring of 1983

Available: <http://www-formal.stanford.edu/jmc/history/timesharing/timesharing.html>

A terrific historical note on where the idea of time-sharing might have come from, including some doubts towards those who cite Strachey's work [S59] as the pioneering work in this area.

[NS07] "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation"

Nicholas Nethercote and Julian Seward

PLDI 2007, San Diego, California, June 2007

Valgrind is a lifesaver of a program for those who use unsafe languages like C. Read this paper to learn about its very cool binary instrumentation techniques – it's really quite impressive.

[R+89] "Mach: A System Software kernel"

Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alessandro Forin, David Golub, Michael Jones

COMPON 89, February 1989

Although not the first project on microkernels per se, the Mach project at CMU was well known and influential; it still lives today deep in the bowels of Mac OS X.

[S59] "Time Sharing in Large Fast Computers"

C. Strachey

Proceedings of the International Conference on Information Processing, UNESCO, June 1959

One of the earliest references on time sharing.

[S+03] "Improving the Reliability of Commodity Operating Systems"

Michael M. Swift, Brian N. Bershad, Henry M. Levy

SOSP 2003

The first paper to show how microkernel-like thinking can improve operating system reliability.

14 Interlude: Memory API

この章で、UNIX システムでのメモリ割り当てインターフェースについて説明します。提供されるインターフェイスは非常にシンプルです。

CRUX: HOW TO ALLOCATE AND MANAGE MEMORY

UNIX/C プログラムでは、堅牢で信頼性の高いソフトウェアを構築する上で、メモリの割り当てと管理の方法を理解することが重要です。よく使われるインターフェースは何ですか？ どのような間違いを避けるべきですか？

14.1 Types of Memory

C プログラムの実行には、2 種類のメモリが割り当てられています。最初のものはスタックメモリと呼ばれ、その割り当てと割り当て解除はコンパイラによって暗黙的に管理されます。このため、自動メモリと呼ばれることもあります。

C でスタック上のメモリを宣言するのは簡単です。たとえば、`x` と呼ばれる整数の場合、関数 `func()` にあるスペースが必要であるとします。そのような記憶を宣言するには、次のようなことをしてください。

```
void func() {
    int x; // declares an integer on the stack
    ...
}
```

コンパイラは残りを行い、`func()` を呼び出すときにスタックにスペースを確保します。関数から戻ると、コンパイラはメモリを解放します。したがって、呼び出しの呼び出しを超えて生きる情報が必要な場合は、その情報をスタックに残さないほうがよいでしょう。

ヒープメモリと呼ばれる 2 種類目のメモリが長持ちするメモリが必要性です。すべての割り当てと割り当て解除は、プログラマによって明示的に処理されます。これを使うことは間違いなく重い責任です！ そして確かに多くのバグの原因にもなっています。しかし、注意を払うと、このようなインターフェースを問題なく使用できます。ヒープに整数を割り当てる方法の例を次に示します。

```
void func() {
    int *x = (int *) malloc(sizeof(int));
    ...
}
```

この小さなコードスニペットに関するいくつかの注意をこれから述べます。最初に、スタックとヒープの両方の割り当てがこの行で行われることに気付くかもしれません。まず、コンパイラは、ポインタ (`int * x`) の宣言を見たときに整数へのポインタのためのスペースを確保することを知っています。その後、プログラムが `malloc()` を呼び出すと、ヒープ上の整数のためのスペースを要求します。ルーチンはそのような整数のアドレスを返します (成功した場合はアドレス、失敗した場合は `NULL` を返します)。その後、プログラムで使用するためにスタックに格納されます。

明示的な性質のために、そしてヒープメモリは、より多様な使用法のために、ユーザーとシステムの両方にとつてより多くの課題が存在します。

14.2 The malloc() Call

malloc()呼び出しは非常に単純です。ヒープ上の空き容量を求めるサイズを渡し、それが成功し、新たに割り当てられた領域へのポインタを返すか、失敗し NULL を返します。マニュアルページには、malloc を使用するために必要なことが示されています。コマンドラインで man malloc と入力すると、次のように表示されます。

```
#include <stdlib.h>
...
void *malloc(size_t size);
```

この情報から、malloc を使用するためのヘッダファイル stdlib.h をインクルードするだけで済みます。実際には、これを行う必要はありません。すべての C プログラムがデフォルトでリンクする C ライブラリには、その内部に malloc() のコードがあります。ヘッダを追加するだけで、コンパイラは malloc() を正しく呼び出すかどうかを確認できます(たとえば、適切な数の引数を適切な型に渡すなど)

単一のパラメータ malloc() は、必要なバイト数を単純に記述するタイプ t の型です。しかし、ほとんどのプログラマーはここに数字を直接入力しません(10 など)。代わりに、様々なルーチンおよびマクロが利用されます。たとえば、倍精度浮動小数点値にスペースを割り当てるには、次のようにします。

```
double *d = (double *) malloc(sizeof(double));
```

うわー、二重にたくさんあります！ この malloc() の呼び出しは、sizeof() 演算子を使用して適切な量のスペースを要求します。これは一般にコンパイル時の演算子とみなされます。つまり、コンパイル時に実際のサイズが分かり、したがって malloc の引数として数値(この場合は 8、倍精度の場合は 8)が代入されます。このため、sizeof() は関数呼び出しではなく演算子として正しく考えられます(関数呼び出しは実行時に行われます)。

TIP: WHEN IN DOUBT, TRY IT OUT

使用しているルーチンや演算子がどのように動作しているかわからない場合は、単純に試してみて、期待どおりに動作するかどうかを確認することに代わるものはありません。マニュアルページやその他のドキュメントを読むのは便利ですが、実際にどのように動作するかは重要です。いくつかのコードを書いてテストしてください。それは間違いなくあなたのコードがあなたの望むように動作することを確認する最善の方法です。

また、変数の名前(型だけでなく)を sizeof() に渡すこともできますが、場合によっては結果が得られない場合もありますので注意してください。たとえば、次のコードスニペットを見てみましょう。

```
int *x = malloc(10 * sizeof(int));
printf("%d\n", sizeof(x));
```

最初の行では、10 個の整数の配列のためのスペースを宣言しました。これはきれいで素敵です。しかし、次の行で sizeof() を使用すると、4(32 ビットマシンの場合) や 8(64 ビットマシンの場合)などの小さな値が返されます。その理由は、sizeof() は、動的に割り当てられたメモリ量ではなく、整数へのポインタの大きさを単に求めていると考えているからです。しかし、sizeof() が期待通りに動作することもあります。

```
int x[10];
printf("%d\n", sizeof(x));
```

この場合、40 バイトが割り当てられていることをコンパイラが知るための十分な静的情報があります。注意すべき別の場所は文字列です。文字列のスペースを宣言するときは、関数 `strlen()` を使用して文字列の長さを取得し、その末尾のスペースを確保するために 1 を追加する次のイディオムを使用します。`malloc(strlen(s)+1)sizeof()` を使用すると、ここで問題が発生する可能性があります。

`malloc()` は `void` 型へのポインタを返します。こうしている理由は、C 言語でアドレスを渡して、プログラマがそれをどうするかを決定させるためです。プログラマーは、キャストと呼ばれるものを使用することによってさらに助けます。上記の例では、プログラマーは `malloc()` の戻り値の型を `double` へのポインタにキャストします。`malloc()` の結果をキャストすることは、コンパイラやあなたのコードを読んでいる他のプログラマーには、「ええ、私がやっていることは分かっています」と言っている以外に、キャストは本当に何もやってくれません。正確に使うため以外にキャストは必要ありません。

14.3 The `free()` Call

結論として、メモリの割り当ては簡単な方程式の一部です。いつ、どのように、そしてメモリを解放するかを知ることは難しい部分です。使用されていないヒープメモリを解放するために、プログラマは単に `free()` を呼び出します。

```
int *x = malloc(10 * sizeof(int));
...
free(x);
```

このルーチンは `malloc()` によって返された 1 つの引数をとります。したがって、割り当てられた領域のサイズはユーザーによって渡されず、メモリ割り当てライブラリ自体によって追跡される必要があります。

14.4 Common Errors

`malloc()` と `free()` の使用には多くの一般的なエラーがあります。ここでは、学部のオペレーティングシステムコースを教える上で何度も見てきたことがあります。これらの例はすべて、コンパイラからの指令でコンパイルして実行します。正しい C プログラムを構築するには C プログラムをコンパイルする必要がありますが、(難しい方法で) よく学びたいのであれば十分に知る必要があります。

正確なメモリ管理は、多くの新しい言語が自動メモリ管理をサポートしているという問題がありました。そのような言語では、メモリを割り当てるために `malloc()` に似たもの (通常は新しいものや新しいオブジェクトを割り当てるのに似たもの) を呼び出している間に、空き領域に何かを呼び出す必要はありません。例えば、ガベージコレクタでは、参照していないメモリを見つけ出し、解放します。

Forgetting To Allocate Memory

多くのルーチンは、呼び出す前にメモリを割り当てる 것을期待しています。たとえば、ルーチン `strcpy(dst, src)` は、ソースポインタからデスティネーションポインタに文字列をコピーします。しかし、あなたが慎重でない場合、あなたはこれを行うかもしれません。

```
char *src = "hello";
char *dst; // oops! unallocated
```

```
strcpy(dst, src); // segfault and die
```

このコードを実行すると、セグメンテーションフォルトにつながる可能性があります。つまり、「メモリに関して何か間違っているよ！ もしかしてアホなプログラマ？？ やめたら、プログラマ」というコードです。

TIP: IT COMPILED OR IT RAN ! = IT IS CORRECT

プログラムがコンパイルされたか、正しく1回または複数回実行されただけであっても、プログラムが正しいとは限りません。多くのイベントは、あなたはそれがうまくいくと思うかもしれません、何かの拍子で止まります。一般的な生徒の反応は、「ちょっと待って！ プログラムは以前は動いた！」と言い、コンパイラ、オペレーティングシステム、ハードウェア、または教授の責任を(たとえそれを言い表しても)責めます。しかし、問題は通常あなたのコード内にあると思うのが正しいです。あなたがそれらの他のコンポーネントを責める前に、デバッグしてください。

この場合、適切なコードは次のようにになります。

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src) + 1);
strcpy(dst, src); // work properly
```

代わりに、`strdup()`を使用して、あなたの人生をさらに楽にすることができます。詳細は `strdup` の man ページを参照してください。

Not Allocating Enough Memory

関連するエラーで十分なメモリが割り当てられていない場合があります。バッファオーバーフローと呼ばれることもあります。上記の例では、一般的なエラーは、宛先バッファに十分な余裕を持たせることです。

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src)); // too small!
strcpy(dst, src); // work properly
```

面白いことに、`malloc` の実装方法やその他の詳細によっては、このプログラムはよく正しく動作します。場合によっては、文字列のコピーが実行されるときに、割り当てられたスペースの最後を過ぎて1バイト分先に書き込まれますが、これは無害で、おそらくは使用されない変数を上書きするだけかもしれません。しかし、場合によっては、これらのオーバーフローは非常に有害であり、実際にはシステム [W06] の多くのセキュリティ脆弱性の原因です。他のケースでは、`malloc` ライブラリは少し余分なスペースを割り当てていますので、あなたのプログラムは実際には他の変数の値を書き換えず、うまく動作します。他の場合では、プログラムは本当に故障しクラッシュするかもしれません。たとえそれが正しく実行されたとしても、正しいことを意味するわけではありません。

Forgetting to Initialize Allocated Memory

このエラーでは、`malloc()` を適切に呼び出しますが、新しく割り当てられたデータ型にいくつかの値を入力するのを忘れてしまいます。絶対にこれはしないでください！ あなたが忘れてしまった場合、プログラムは最終的に、未知の値のデータをヒープから読み出す初期化されていない読み込みに遭遇します。そこに何があるのか誰が知っていますか？ あなたが運が良ければ、プログラムがまだ機能するような価値があります(たとえば、ゼロ)。不幸であれば、ランダムなデータや害があるデータに遭遇するでしょう。

Forgetting To Free Memory

別の一般的なエラーはメモリリークと呼ばれ、メモリを解放するのを忘れたときに発生します。長時間実行するアプリケーションやシステム (OS 自体など) では、これは大きな問題です。したがって、一般的に、あなたがメモリのチャンクで完了したら、あなたはそれを解放することを確認する必要があります。注意してほしいことはガベージコレクションを使っている言語では、こここの話は役に立たないことに注意してください。しかし、ガベージコレクションでも、まだメモリの一部を参照している場合、ガベージコレクタはそれを解放しないため、現代の言語でもメモリリークが問題になります。

場合によっては、`free()` を呼び出すのが合理的であるように見えるかもしれません。たとえば、あなたのプログラムは短命であり、すぐに終了します。この場合、プロセスが終了すると、OS は割り当てられたすべてのページをクリーンアップし、したがってメモリリーク自体は発生しません。これは確かに機能しますが、開発するときはおそらく戦略が悪いので、そのような戦略を選ぶことには注意してください。長期的には、プログラマーとしてのあなたの目標の 1 つは良い戦略を開発することです。そのような戦略の 1 つは、あなたがどのようにメモリを管理しているのか (C のような言語で)、あなたが割り当てたブロックを解放することです。あなたがそうしないで逃げることができたとしても、あなたが明示的に割り当てる各バイトを解放する習慣を得るのは良いでしょう。

Freeing Memory Before You Are Done With It

場合によっては、プログラムの使用が終了する前にメモリが解放されることもあります。そのような間違いは、ぶら下がりポインタと呼ばれ、あなたが推測できるように、それはまた悪いことです。後で使用すると、プログラムがクラッシュしたり、有効なメモリを上書きすることができます (たとえば、`free()` を呼び出した後、`malloc()` をもう一度呼び出すと、余分に解放されたメモリがリサイクルされます)。

Freeing Memory Repeatedly

プログラムでは、メモリを複数回解放することもあります。これはダブルフリーとして知られています。その結果は未定義です。あなたが想像しているように、メモリ割り当てライブラリは混乱し、あらゆる種類の奇妙なことをするかもしれません。クラッシュは共通の結果です。

Calling `free()` Incorrectly

私たちが議論している最後の問題は、`free()` の呼び出しが間違っていることです。結局、`free()` はあなたが以前に `malloc()` から受け取ったポインタの 1 つだけを渡すことを期待しています。あなたが他の値を渡すと、悪いことが起きる可能性があります。したがって、このような無効な自由は危険であり、もちろん避けなければなりません。

ASIDE: WHY NO MEMORY IS LEAKED ONCE YOUR PROCESS EXITS

短命のプログラムを書くときには、`malloc()` を使っていくらかの領域を割り当てるかもしれません。プログラムは実行され、完了しようとしています。終了する直前に `free()` を呼び出す必要がありますか？それが間違っているように見えますが、実際の意味では記憶が失われません。理由は簡単です。実際には、システムには 2 つのレベルのメモリ管理があります。第 1 レベルのメモリ管理は、実行時にプロセスにメモリを引き渡す OS によって実行され、プロセスが終了する (または終了する) ときにメモリを取り戻します。第 2 レベルの管理は、`malloc()` と `free()` を呼び出すときにヒープ内など、各プロセス内にあります。`free()` を呼び出すことに失敗してヒープ内のメモリがリークしても、オペレーティングシステムは、プログラムのすべてのメモリ (コード、スタック、ここではヒープに関連するページを含む) を実行が終了したら再利用します。アドレス空

間内のヒープの状態が何であっても、OS はプロセスが終了したときにそれらのページをすべて取り戻すので、解放していないにもかかわらずメモリが失われます。

したがって、短命のプログラムでは、メモリが漏れても操作上の問題が発生しないことがよくあります(ただし、悪い形であると考えられます)。長時間実行しているサーバー(Web サーバーやデータベース管理システムなど、決して終了しないサーバーなど)を作成すると、漏れたメモリははるかに大きな問題になり、メモリが不足するとクラッシュする可能性があります。もちろん、メモリのリークは、オペレーティングシステムそのものの 1 つの特定のプログラム内でのさらに大きな問題です。カーネルコードを書く人は、すべての中で最も厳しい仕事をしています…

Summary

ご覧のように、メモリを乱用する方法はたくさんあります。メモリのエラーが頻繁に発生するため、コード内でこのような問題を見つけるのに役立つエコスペースのツールが開発されました。[\[HJ92\]](#) と [valgrind](#) [\[SN05\]](#) の両方を調べてください。どちらもメモリ関連の問題の原因を突き止めるのに役立ちます。これらの強力なツールの使用に慣れたら、そのツールを使わずに生き残った方法が不思議に思うでしょう。

14.5 Underlying OS Support

`malloc()` と `free()` について議論するときにシステムコールについて話していないことに気づいたかもしれません。その理由はシンプルです。システムコールではなく、ライブラリ呼び出しえです。したがって、`malloc` ライブラリは仮想アドレス空間内の空間を管理しますが、それ自体は OS に呼び出されるいくつかのシステムコールの上に構築され、より多くのメモリを要求したり、システムに戻っていくことがあります。

そのようなシステムコールの 1 つは `brk` と呼ばれ、プログラムのブレークの位置を変更するために使用されます。ヒープの終わりの位置です。1 つの引数(新しいブレークのアドレス)が必要なので、新しいブレークが現在のブレークより大きいか小さいかに基づいてヒープのサイズを増減します。追加の呼び出し `sbrk` はインクリメントされますが、そうでなければ同様の目的を果たします。

`brk` または `sbrk` のいずれかを直接呼び出すことはできません。それらはメモリ割り当てライブラリによって使用されます。あなたがそれらを使用しようとすると、おそらく何かが(ひどく)間違ってしまうことになります。代わりに `malloc()` と `free()` を使用してください。

最後に、`mmap()` 呼び出しによってオペレーティングシステムからメモリを取得することもできます。正しい引数を渡すことで、`mmap()` はプログラム内で匿名のメモリ領域を作成することができます。この領域は特定のファイルに関連付けられず、スワップ空間に関連付けられます。このメモリは、ヒープのように扱うことができ、ヒープのように管理することもできます。詳細は、`mmap()` のマニュアルページを参照してください。

14.6 Other Calls

メモリ割り当てライブラリがサポートするその他の呼び出しがいくつあります。たとえば、`calloc()` はメモリを割り当て、返す前にメモリをゼロにします。これは、メモリがゼロになっていると仮定して、それを自分で初期化することを忘れるところで、いくつかのエラーを防ぎます(上記の「初期化されていない読み取り」の段落を参照してください)。ルーチン `realloc()` は、何か(配列など)にスペースを割り当ててから、何かを追加する必要があるときにも便利です。`realloc()` は新しいメモリ領域を作り、古い領域をポインタを新しい領域に返します。

14.7 Summary

メモリ割り当てを扱う API のいくつかを紹介しました。いつものように、私たちは基本的な話をしました。詳細は他の場所でも入手可能です。詳細は、C ブック [KR88] とスティーブンス SR05 を参照してください。これらの問題の多くを自動的に検出して修正する方法に関する現代的な最新の論文については、Novark et al. [N+07] というものがあります。このペーパーには、一般的な問題の素敵な要約と、それらを見つけて修正するためのいくつかのすばらしいアイデアも含まれています。

参考文献

- [HJ92] Purify: Fast Detection of Memory Leaks and Access Errors
R. Hastings and B. Joyce
USENIX Winter '92
The paper behind the cool Purify tool, now a commercial product.
- [KR88] “The C Programming Language”
Brian Kernighan and Dennis Ritchie
Prentice-Hall 1988
The C book, by the developers of C. Read it once, do some programming, then read it again, and then keep it near your desk or wherever you program.
- [N+07] “Exterminator: Automatically Correcting Memory Errors with High Probability”
Gene Novark, Emery D. Berger, and Benjamin G. Zorn
PLDI 2007
A cool paper on finding and correcting memory errors automatically, and a great overview of many common errors in C and C++ programs.
- [SN05] “Using Valgrind to Detect Undefined Value Errors with Bit-precision”
J. Seward and N. Nethercote
USENIX '05
How to use valgrind to find certain types of errors.
- [SR05] “Advanced Programming in the UNIX Environment”
W. Richard Stevens and Stephen A. Rago
Addison-Wesley, 2005
We've said it before, we'll say it again: read this book many times and use it as a reference whenever you are in doubt. The authors are always surprised at how each time they read something in this book, they learn something new, even after many years of C programming.
- [W06] “Survey on Buffer Overflow Attacks and Countermeasures”
Tim Werthman
Available: www.nds.rub.de/lehre/seminar/SS06/Werthmann_BufferOverflow.pdf
A nice survey of buffer overflows and some of the security problems they cause. Refers to many of the famous exploits.

15 Mechanism: Address Translation

CPU の仮想化の開発では、限定直接実行 (LDE) と呼ばれる一般的なメカニズムに焦点を当てました。LDE の背後にある考え方は単純です。ほとんどの場合、プログラムをハードウェア上で直接実行させたいです。しかし、特定のキーポイント (プロセスがシステムコールを発行するときやタイマー割り込みが発生したときなど) では、OS が関与し、「正しい」ことが起こるように調整します。したがって、OS は、少しハードウェアをサポートして、効率的な仮想化を実現するために、実行中のプログラムから抜け出すために最善を尽くします。しかし、これらの重要な時点で介在することにより、OS はハードウェアの制御を確実に維持します。効率性と制御性は、最新のオペレーティングシステムの主な目標の 2 つです。

メモリの仮想化では、同様の戦略を追求し、効率と制御の両面を達成しながら、望ましい仮想化を実現します。効率性は、最初は非常に初步的である (例えば、ほんの数個のレジスタ) ハードウェアサポートを利用しますが、かなり複雑になっています (例えば、TLB、ページテーブルサポートなど) 制御は、OS がアプリケーションがメモリにアクセスすることを許可されていないことを保証することを意味します。したがって、アプリケーションをお互いから保護し、アプリケーションから OS を保護するために、ここでもハードウェアの助けが必要です。最後に、柔軟性の点で、VM システムからもう少し必要なものがあります。具体的には、プログラムがアドレス空間をどのような方法でも使用できるようにしたいので、システムのプログラミングを容易にしたいと考えています。それで、私たちはとある要点にたどり着きます。

THE CRUX: HOW TO EFFICIENTLY AND FLEXIBLY VIRTUALIZE MEMORY

どのように効率的な仮想化を構築できますか？ アプリケーションに必要な柔軟性はどのように提供しますか？ アプリケーションがアクセスできるメモリの場所をどのように制御し、アプリケーションのメモリアクセスが適切に制限されているかを確実に管理するにはどうすればよいですか？ どのようにしてこのすべてを効率的に行うのですか？

限定直接実行の一般的なアプローチに加えて、使用する一般的な手法は、ハードウェアベースのアドレス変換、または単にアドレス変換と呼ばれるものです。アドレス変換では、ハードウェアは、各メモリアクセス (命令フェッチ、ロード、またはストア) を変換し、命令によって提供される仮想アドレスを、必要な情報が実際に位置する物理アドレスに変更します。したがって、各メモリ参照ごとに、ハードウェアによってアドレス変換が実行され、アプリケーションメモリ参照がメモリ内の実際の位置にリダイレクトされます。

もちろん、ハードウェアだけでは、メモリを仮想化することはできません。効率的に行うための低レベルのメカニズムを提供するだけです。正しい変換が行われるように、ハードウェアをセットアップするためには、OS が重要なポイントに関わる必要があります。したがって、メモリを管理し、空き領域と使用中の領域を追跡し、メモリの使用方法を制御するために適切に介入する必要があります。

もう一度この仕事の目標は、美しい錯覚を作り出すことです。プログラムには、独自のコードとデータが存在する独自のプライベートメモリがあります。その仮想現実の背後には醜い物理的真理があります。つまり、CPU(または CPU 達) が 1 つのプログラムと次のプログラムを実行する間に、多くのプログラムが実際に同時にメモリを共有しています。仮想化を通じて、OS は (ハードウェアの助けを借りて) 醜いマシンの現実を、有用で、強力で、使いやすい抽象であるものに変えます。

15.1 Assumptions

メモリを仮想化しようとする私たちの最初の試みは、非常に簡単です。TLB、マルチレベルのページテーブル、その他の技術的な不思議な点を理解しようとすると、すぐに OS があなたを馬鹿にするでしょう。ここでは OS がどのように動くかをお話しするだけです。

具体的には、ユーザーのアドレス空間を物理メモリに連続して配置する必要があると想定します。わかりやすくするために、アドレス空間のサイズがそれほど大きくないと仮定します。具体的には、物理メモリのサイズよりも小さいことを示します。最後に、各アドレス空間がまったく同じサイズであると仮定します。これらの前提が非現実的であるとは心配しないでください。そのようにして現実的な仮想化を実現します。

15.2 An Example

アドレス変換を実装するために必要なことと、そのようなメカニズムが必要な理由を理解するために、簡単な例を見てみましょう。アドレス空間が図 15.1 のようなプロセスがあるとします。ここで検討するのは、メモリから値をロードし、3 ずつインクリメントした後、その値をメモリに戻す短いコードシーケンスです。このコードの C 言語表現では次のようになるかもしれません。

```
void func() {
    int x = 3000; // thanks, Perry.
    x = x + 3; // this is the line of code we are interested in
```

TIP: INTERPOSITION IS POWERFUL

Interposition は、コンピュータシステムにおいて大きな効果を発揮するためによく使用される一般的かつ強力な技術です。仮想化メモリでは、ハードウェアが各メモリアクセスに介入し、プロセスによって発行された各仮想アドレスを、必要な情報が実際に格納されている物理アドレスに変換します。しかしながら、Interposition の一般的な技術は、より広範に適用可能です。実際には、ほとんどすべての明確なインターフェースを Interposition させたり、新しい機能を追加したり、システムの他の側面を改善したりすることができます。このようなアプローチの通常のメリットの1つは透明性です。インターフェースのクライアントを変更する必要がなく Interposition が行われることが多いです。

コンパイラは、このコード行をアセンブリに変換します。これは、(x86 アセンブリ内で) このように見えます。Linux の場合は objdump、Mac の場合は otool を使って逆アセンブルしてください：

```
128: movl 0x0(%ebx), %eax ;load 0+ebx into eax
132: addl $0x03, %eax ;add 3 to eax register
135: movl %eax, 0x0(%ebx) ;store eax back to mem
```

このコードスニペットは比較的簡単です。x のアドレスがレジスタ ebx に置かれていると想定し、movl 命令 (“ロングワード” 移動の場合) を使用してそのアドレスの値を汎用レジスタ eax にロードします。次の命令は eax に 3 を加え、最後の命令はその同じ位置のメモリに eax の値を戻します。

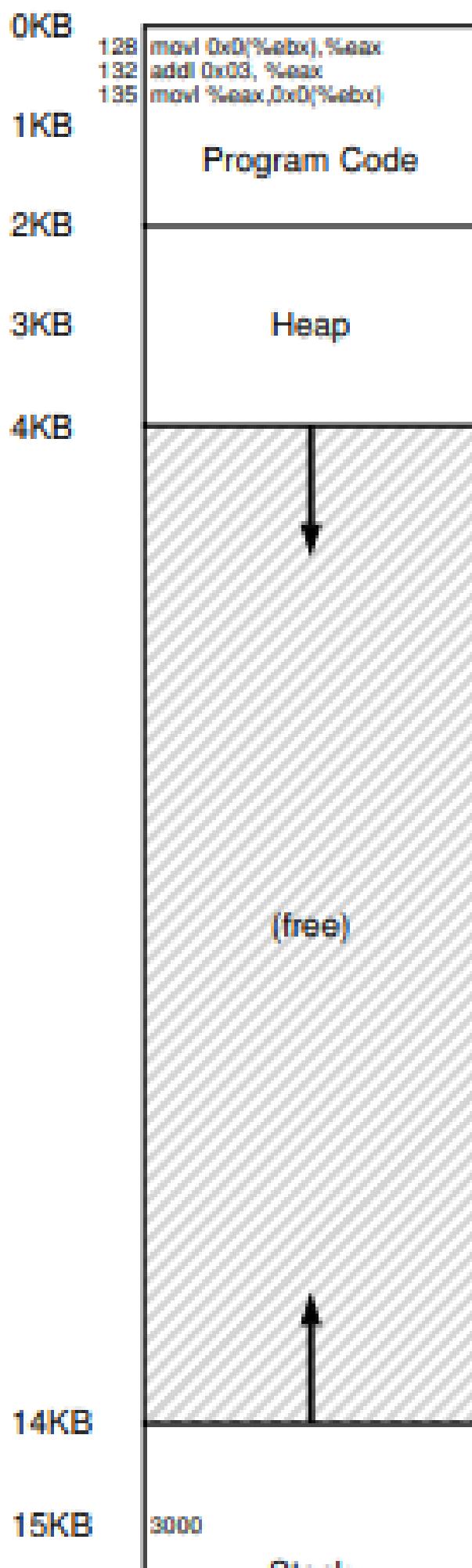


図 15.1(137 ページ) では、コードとデータの両方がプロセスのアドレス空間にどのように配置されているかを見ることができます。3 命令コードシーケンスは 128 番地(最上部付近のコードセクション)に配置され、変数 *x* の値はアドレス 15KB に配置されます(最下部にスタックされる)。図では、*x* の初期値はスタック上の位置に示されているように 3000 です。

これらの命令が実行されると、プロセスの観点から、次のメモリアクセスが行われます。- 128 番地の命令をフェッチする - この命令を実行する(アドレス 15 KB からロードする) - アドレス 132 で命令をフェッチする - この命令を実行する(メモリ参照なし) - 135 番地に命令をフェッチする - この命令を実行する(15KB のアドレスにストアする)

プログラムの観点からは、アドレス空間はアドレス 0 から始まり、最大 16 KB まで増加します。それが生成するすべてのメモリ参照は、これらの範囲内になければなりません。しかし、メモリを仮想化するために、OS は必ずしもアドレス 0 でなくても、物理メモリのどこかにプロセスを配置する必要があります。したがって、プロセスに透過的な方法でこのプロセスをどのようにメモリ内に再配置できますか？ 実際にはアドレス空間が他の物理アドレスに位置しているときに、0 から始まる仮想アドレス空間の錯覚をどのように提供することができますか？

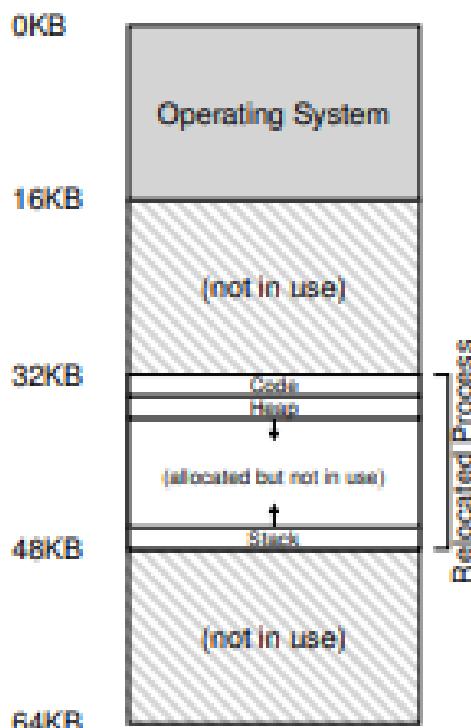


Figure 15.2: Physical Memory with a Single Relocated Process

このプロセスのアドレス空間がメモリに配置された後の物理メモリの例を図 15.2 に示します。この図では、物理メモリの最初のスロットを使用して OS を認識し、上の例のプロセスを物理メモリアドレス 32 KB から始まるスロットに再配置しました。他の 2 つのスロットは空いています(16 KB-32 KB と 48 KB-64 KB)

15.3 Dynamic (Hardware-based) Relocation

ハードウェアベースのアドレス変換の理解を得るために、まず最初のアプローチについて説明します。1950 年代後半の初めてのタイムシェアリングマシンでは、基本(base)と境界(bounds)と呼ばれる簡単なアイデアが導入されました。この技術は動的再配置(dynamic relocation)とも呼ばれます。両方の用語を同じ意味で使用します [SS74]。

具体的には、各 CPU 内に 2 つのハードウェアレジスタが必要です。1 つは基本レジスタ、もう 1 つは境界レジスタです（制限レジスタとも呼ばれます）。この基本と境界のペアは、私たちが物理メモリ内のどこにでもアドレス空間を置くことを可能にし、プロセスがそれ自身のアドレス空間にしかアクセスできないようにします。

この設定では、各プログラムはアドレスゼロでロードされているかのように記述され、コンパイルされます。しかし、プログラムが実行を開始すると、OS は物理メモリ内のどこにロードすべきかを決定し、基本レジスタをその値に設定する。上記の例では、OS は物理アドレス 32 KB にプロセスをロードすることを決定し、基本レジスタをこの値に設定します。

プロセスが実行されているときに面白いことが起こります。ここで、プロセスによってメモリ参照が生成されると、プロセッサによって次のように変換されます。

$$\text{physical address} = \text{virtual address} + \text{base}$$

ASIDE: SOFTWARE-BASED RELOCATION

ハードウェアサポートが始まる前の初期の段階では、純粋にソフトウェアメソッドを使用して粗い形式の再配置を実行したシステムもありました。基本的な手法は静的再配置 (static relocation) と呼ばれ、ローダーと呼ばれるソフトウェアが実行しようとしている実行ファイルを取り込み、そのアドレスを物理メモリの望ましいオフセットに書き換えます。

例えば、命令がアドレス 1000 からレジスタ（例えば、`movl 1000, %eax`）へのロードであり、プログラムのアドレス空間が 3000 番地から読み込まれた場合（プログラムが考えるよう 0 ではなく）ローダーは、各アドレスを 3000 でオフセットするように命令を書き換えます（たとえば、`movl 4000, %eax`）。このようにして、プロセスのアドレス空間の単純な静的再配置がされます。

しかし、静的再配置には多くの問題があります。プロセスが不正なアドレスを生成し、他のプロセスや OS メモリに不正にアクセスする可能性があります。そして最も重要なのは、保護を提供しないことです。真の保護 [WL+93] のためには、一般的にハードウェアのサポートが必要になります。別の否定的な点は、いったん配置されると、後でアドレス空間を別の場所に再配置することが難しいことです [M65]。

プロセスによって生成された各メモリ参照は仮想アドレスです。ハードウェアはベースレジスタの内容をこのアドレスに加算し、結果はメモリシステムに発行できる物理アドレスです。これをよりよく理解するために、1 つの命令が実行されたときの動作をトレースしてみましょう。具体的には、前のシーケンスからの 1 つの命令を見てみましょう：

```
128: movl 0x0(%ebx), %eax
```

プログラムカウンタ (PC) は 128 に設定されています。ハードウェアがこの命令をフェッチする必要があるときには、最初に 32KB の基本レジスタ値 (32768) に値を加算して 32896 の物理アドレスを取得します。ハードウェアはその物理アドレスから命令をフェッチします。次に、プロセッサは命令の実行を開始します。ある時点では、プロセスは仮想アドレス 15 KB からプロセッサに命令を発行し、再び基本レジスタ (32 KB) に加算して、最終物理アドレス 47 KB を取得し、その結果、必要な内容を取得します。

仮想アドレスを物理アドレスに変換することは、まさにアドレス変換と呼ばれる手法です。すなわち、ハードウェアは、プロセスが参照していると考える仮想アドレスを取り、データが実際に存在する物理アドレスに変換します。このアドレスの再配置は実行時に行われるため、プロセスの実行が開始された後でもアドレス空間を移動できるため、技術はよく動的再配置 (dynamic relocation) [M65] と呼ばれます。

TIP: HARDWARE-BASED DYNAMIC RELOCATION

動的再配置では、少しハードウェアが大きくなります。具体的には、ベースレジスタは、（プログラムによって生成された）仮想アドレスを物理アドレスに変換するために使用されます。境界（ま

たは制限) レジスタは、そのようなアドレスがアドレス空間の範囲内にあることを保証します。これらは一緒になって、シンプルで効率的なメモリの仮想化を実現します。

今あなたは疑問に思っているかもしれません。その境界(限界)登録はどうなりましたか? ということです。結局のところ、これはベースと境界アプローチではありませんか? 確かにそうです。あなたが推測したように、境界レジスタは保護を助けるためのものです。具体的には、プロセッサはまずメモリ参照が境界内にあることを確認して、メモリ参照が合法であることを確認する。上記の単純な例では、境界レジスタは常に 16 KB に設定されます。プロセスが境界よりも大きい仮想アドレスまたは負のアドレスを生成した場合、CPU は例外を発生させ、プロセスは終了する可能性があります。境界のポイントは、プロセスによって生成されたすべてのアドレスが合法でプロセスの「境界」内にあることを確認することです。

基本レジスタと境界レジスタは、チップ上に保持されているハードウェア構造(CPUごとに1組)であることに注意してください。メモリ管理ユニット(MMU)のアドレス変換に役立つプロセッサの一部を人々が呼ぶこともあります。より洗練されたメモリ管理技術を開発するにつれて、MMU にさらに多くの回路を追加するはずです。

バインドされたレジスタについての小さなことで、2つの方法のいずれかで定義できます。上記のように、アドレス空間のサイズを保持するため、ハードウェアは、ベース値を加算する前に、仮想アドレスを最初にチェックします。2番目の方法では、アドレス空間の最後の物理アドレスが保持されます。したがって、ハードウェアはまずベース値を加算し、アドレスが境界内にあることを確認します。どちらの方法も論理的に同等です。簡単にするために、通常は前者の方法を仮定します。

Example Translations

ベースとバウンドによるアドレス変換をより詳細に理解するために、例を見てみましょう。物理アドレス 16 KB にサイズ 4 KB(非現実的小さいです) のアドレス空間がロードされたプロセスを想像してください。以下に、いくつかのアドレス変換の結果を示します。

Virtual Address		Physical Address
0	→	16 KB
1 KB	→	17 KB
3000	→	19384
4400	→	Fault (out of bounds)

この例からわかるように、仮想アドレス(アドレス空間へのオフセットと見なすことができます)に基本アドレスを加算するだけで簡単に物理アドレスを取得できます。仮想アドレスが「大きすぎる」または負の場合のみ、結果がフォルトとなり、例外が発生します。

ASIDE: DATA STRUCTURE — THE FREE LIST

OS は、空きメモリのどの部分が使用されていないかを追跡して、プロセスにメモリを割り当てることができるようになります。もちろん、このようなタスクには多くの異なるデータ構造を使用できます。最も単純なもの(ここではこれを仮定します)は、現在使用されていない物理メモリの範囲のリストであるフリーリストです。

Hardware Requirements	Notes
Privileged mode	<i>Needed to prevent user-mode processes from executing privileged operations</i>
Base/bounds registers	<i>Need pair of registers per CPU to support address translation and bounds checks</i>
Ability to translate virtual addresses and check if within bounds	<i>Circuitry to do translations and check limits; in this case, quite simple</i>
Privileged instruction(s) to update base/bounds	<i>OS must be able to set these values before letting a user program run</i>
Privileged instruction(s) to register exception handlers	<i>OS must be able to tell hardware what code to run if exception occurs</i>
Ability to raise exceptions	<i>When processes try to access privileged instructions or out-of-bounds memory</i>

Figure 15.3: Dynamic Relocation: Hardware Requirements

15.4 Hardware Support: A Summary

ハードウェアから必要なサポートを要約しましょう(図 15.3 参照)まず、CPU 仮想化の章で説明したように、2つの異なる CPU モードが必要です。OS は特権モード(またはカーネルモード)で実行され、マシン全体にアクセスできます。アプリケーションはユーザー モードで実行されます。ユーザー モードでは、ユーザーの操作が制限されています。ある種のプロセッサステータスワードに格納されている単一のビットは、CPU が現在どのモードで動作しているかを示します。特定の特別な機会(例えば、システムコールまたは他の種類の例外または割り込み)が発生すると、CPU はモードを切り替えます。

ハードウェアは、ベースレジスタと境界レジスタ自体も提供する必要があります。したがって、各 CPU は、CPU のメモリ管理ユニット(MMU)の一部である一組のレジスタを追加します。ユーザー プログラムが実行されると、ハードウェアは、ユーザー プログラムによって生成された仮想アドレスにベース値を加算することによって、各アドレスを変換します。また、ハードウェアは、アドレスが有効であるかどうかを確認することができなければなりません。これは、境界レジスタと CPU 内のいくつかの回路を使用して行います。

ハードウェアは、ベースレジスタと境界レジスタを変更するための特別な命令を提供し、異なるプロセスが実行されたときに OS がそれらを変更できるようにする必要があります。これらの命令には特権があります。カーネル(または特権)モードでのみ、レジスタを変更できます。実行中にベースレジスタを任意に変更することができれば、ユーザ プロセスが壊れる可能性があると想像してください。

最後に、CPU は、ユーザ プログラムが不法にメモリにアクセスしようとする状況(「範囲外」のアドレス)で例外を生成できなければいけません。この場合、CPU はユーザ プログラムの実行を停止し、OS の「範囲外」例外ハンドラを実行するように調整する必要があります。OS ハンドラは、どのように反応するかを知ることができます。この場合、プロセスを終了する可能性があります。同様に、ユーザ プログラムが(特権の)基本レジスタと境界レジスタの値を変更しようとすると、CPU は例外を発生させ、「ユーザ モードで特権動作を実行しようとした」というハンドラを実行する必要があります。CPU は、これらのハンドラの位置を通知する方法も提供する必要があります。このようにして特権命令をいくつか追加する必要があります。

Hardware Requirements	Notes
Privileged mode	<i>Needed to prevent user-mode processes from executing privileged operations</i>
Base/bounds registers	<i>Need pair of registers per CPU to support address translation and bounds checks</i>
Ability to translate virtual addresses and check if within bounds	<i>Circuitry to do translations and check limits; in this case, quite simple</i>
Privileged instruction(s) to update base/bounds	<i>OS must be able to set these values before letting a user program run</i>
Privileged instruction(s) to register exception handlers	<i>OS must be able to tell hardware what code to run if exception occurs</i>
Ability to raise exceptions	<i>When processes try to access privileged instructions or out-of-bounds memory</i>

Figure 15.3: Dynamic Relocation: Hardware Requirements

15.5 Operating System Issues

ハードウェアが動的再配置をサポートするための新しい機能を提供するのと同じように、OSには新たな問題があります。ハードウェアサポートとOS管理の組み合わせにより、シンプルな仮想メモリの実装が可能になります。具体的には、仮想メモリのベース・バウンド・バージョンを実装するためにOSが関与しなければならないいくつかの重要な接合点があります。

まず、OSは、プロセスが作成されたときにアクションを実行し、メモリ内のアドレス空間のための領域を見つけなければなりません。幸いにも、各アドレス空間が(a)物理メモリのサイズより小さく、(b)同じサイズであるという前提を考えると、これはOSにとって非常に簡単です。物理メモリーをスロットの配列として表示し、各スロットがフリーであるか使用中であるかを追跡できます。新しいプロセスが作成されると、OSはデータ構造(フリーリストと呼ばれることが多い)を検索して、新しいアドレス空間のためのスペースを見つけて使用することをマークする必要があります。可変サイズのアドレス空間では、より複雑になりますが、今後の章ではその懸念を残していきます。

例を見てみましょう。図15.2(139ページ)では、OS自体が物理メモリの最初のスロットを使用していること、および上記の例のプロセスを物理メモリアドレス32 KBから始まるスロットに再配置したことがわかります。他の2つのスロットは空いています(16 KB-32 KBと48 KB-64 KB)。したがって、空きリストはこれらの2つのエントリで構成されます。

第2に、OSは、プロセスが終了したとき(すなわち、正常に終了したとき、または誤って実行されたために強制終了したとき)、他のプロセスまたはOSで使用するためにすべてのメモリを再利用するときに何らかの作業を行わなければいけません。プロセスが終了すると、OSはそのメモリを空きリストに戻し、必要に応じて関連するデータ構造をクリーンアップします。

第3に、コンテキストスイッチが発生した場合、OSはさらにいくつかのステップを実行する必要があります。実際には、各プログラムはメモリ内の異なる物理アドレスにロードされるため、実行中のプログラムごとにその値が異なります。したがって、OSは、プロセス間で切り替えるときに、ベースと境界のペアを保存して復元する必要があります。具体的には、OSがプロセスの実行を停止すると決定した場合、プロセスストラクチャやプロセス制御ブロック(PCB)などのプロセスごとの構造によって、ベースレジスタと境界レジスタの値をメモリに保存する必要があります。同様に、OSが実行中のプロセスを再開する(または最初に実行する)場合、CPU上のベースと境界の値をこのプロセスの正しい値に設定する必要があります。

OS Requirements	Notes
Memory management	<i>Need to allocate memory for new processes; Reclaim memory from terminated processes; Generally manage memory via free list</i>
Base/bounds management	<i>Must set base/bounds properly upon context switch</i>
Exception handling	<i>Code to run when exceptions arise; likely action is to terminate offending process</i>

Figure 15.4: Dynamic Relocation: Operating System Responsibilities

プロセスが停止された（すなわち、実行されていない）ときに、OSはメモリ内のある場所から別の場所へアドレス空間を移動させることができます。プロセスのアドレス空間を移動するために、OSは最初にプロセスをディスクエューリングします。つまり、OSは現在の場所から新しい場所にアドレス空間をコピーします。最後にOSは、（プロセス構造内の）保存されたベースレジスタを更新して、新しい位置を指します。プロセスが再開されると、その（新しい）ベースレジスタが復元され、命令とデータがメモリ内の全く新しい場所にあることを知らずに、再び実行を開始します。

第4に、OSは上記のように例外ハンドラまたは呼び出される関数を提供しなければいけません。OSはブート時に（特権命令によって）これらのハンドラをインストールします。たとえば、プロセスが境界外のメモリにアクセスしようとすると、CPUは例外を送出します。そのような例外が発生した場合、OSは対応する必要があります。OSの一般的な反応は敵意の1つとして、攻撃プロセスを終了させる可能性があります。OSは実行中のマシンを高度に保護する必要があるため、メモリアクセス違反をするプロセスは実行させない必要があります。

OS @ boot (kernel mode)	Hardware	
initialize trap table	remember addresses of... system call handler timer handler illegal mem-access handler illegal instruction handler	
OS @ run (kernel mode)	Hardware	Program (user mode)
To start process A: allocate entry in process table allocate memory for process set base/bounds registers return-from-trap (into A)	start timer; interrupt after X ms	Process A runs Fetch instruction
	restore registers of A move to user mode jump to A's (initial) PC	Execute instruction
	Translate virtual address and perform fetch	
	If explicit load/store: Ensure address is in-bounds; Translate virtual address and perform load/store	
	...	
	Timer interrupt move to kernel mode Jump to interrupt handler	
Handle the trap Call switch() routine save regs(A) to proc-struct(A) (including base/bounds) restore regs(B) from proc-struct(B) (including base/bounds) return-from-trap (into B)	restore registers of B move to user mode jump to B's PC	Process B runs Execute bad load
	Load is out-of-bounds; move to kernel mode jump to trap handler	
Handle the trap Decide to terminate process B de-allocate B's memory free B's entry in process table		

Figure 15.5: Limited Direct Execution Protocol (Dynamic Relocation)

図 15.5(145 ページ) は、タイムライン内のハードウェア/OS の相互作用の大部分を示しています。この図

は、起動時に OS を使用してマシンを使用できる状態にしてから、プロセス（プロセス A）の実行が開始されたときの状態を示しています。OS の介入なしにハードウェアによってメモリ変換がどのように処理されるかを注意してください。ある時点でタイマ割り込みが発生し、OS はプロセス B に切り替わり、プロセス B は（不正なメモリアドレスに対して）「不良ロード」を実行します。その時点では、OS は関与し、プロセスの終了と B のメモリを解放し、プロセステーブルからそのエントリを削除することによってクリーンアップする必要があります。この図からわかるように、我々は依然として制限付き直接実行という基本的なアプローチに従っています。ほとんどの場合、OS はハードウェアを適切に設定し、プロセスを CPU 上で直接実行させます。プロセスが誤動作した場合にのみ OS が関与しなければいけません。

15.6 Summary

この章では、アドレス変換と呼ばれる仮想メモリで使用される特定のメカニズムを使用した限定直接実行の概念を拡張しました。アドレス変換により、OS はプロセスからのすべてのメモリアクセスを制御し、アクセスがアドレス空間の範囲内に収まるようにします。この手法の効率性の鍵はハードウェアのサポートであり、仮想アドレス（プロセスのメモリビュー）を物理的なもの（実際のビュー）に変換するアクセスごとに迅速に変換を実行します。このすべては、再配置されたプロセスに対して透過的な方法で実行されます。そのプロセスはメモリ参照が変換されているということを知らず、素晴らしい錯覚を作り出します。

また、ベースと境界または動的再配置と呼ばれる 1 つの特定の仮想化形態も見てきました。仮想アドレスにベース・レジスタを加算し、プロセスによって生成されたアドレスが境界内にあるかどうかをチェックするには、少しだけハードウェア・ロジックを必要とするため、ベースと境界の仮想化は非常に効率的です。ベースと境界は保護も提供します。OS とハードウェアが組み合わざって、プロセスが独自のアドレス空間外でメモリ参照を生成できないようにします。確かに保護は OS の最も重要な目標の 1 つです。それがなければ、OS はマシンを制御することができません（プロセスがメモリを上書きすることができれば、トラップテーブルを上書きしてシステムを引き継ぐような厄介なことを簡単に行うことができます）

残念なことに、動的再配置のこの単純な手法は非効率的です。たとえば、図 15.2(139 ページ) に示されているように、再配置されたプロセスは 32KB から 48KB の物理メモリを使用しています。ただし、プロセススタッカとヒープがあまりにも大きくなないので、2 つの間のスペースのすべてが単に無駄になります。このタイプの廃棄物は、通常、内部断片化と呼ばれ、割り当てられた単位内の空間がすべて使用されていない（すなわち、断片化されている）ので、空いている部分は無駄になります。現在のアプローチでは、より多くのプロセスに十分な物理メモリがあるかもしれません、現在は固定サイズのスロットにアドレス空間を配置することに制限されているため、内部断片化が発生する可能性があります。したがって、物理メモリをより有効に活用し、内部断片化を回避するために、より洗練されたメカニズムが必要になります。私たちの最初の試みは、セグメント化と呼ばれるベースと境界のわずかな一般化です。次にこれについて説明します。

参考文献

[M65] “On Dynamic Program Relocation”
W.C. McGee
IBM Systems Journal
Volume 4, Number 3, 1965, pages 184–199
This paper is a nice summary of early work on dynamic relocation, as well as some basics on static relocation.

[P90] “Relocating loader for MS-DOS .EXE executable files”
Kenneth D. A. Pillay

Microprocessors & Microsystems archive

Volume 14, Issue 7 (September 1990)

An example of a relocating loader for MS-DOS. Not the first one, but just a relatively modern example of how such a system works.

[SS74] “The Protection of Information in Computer Systems”

J. Saltzer and M. Schroeder

CACM, July 1974

From this paper: “The concepts of base-and-bound register and hardware-interpreted descriptors appeared, apparently independently, between 1957 and 1959 on three projects with diverse goals. At M.I.T., McCarthy suggested the base-and-bound idea as part of the memory protection system necessary to make time-sharing feasible. IBM independently developed the base-and-bound register as a mechanism to permit reliable multiprogramming of the Stretch (7030) computer system. At Burroughs, R. Barton suggested that hardware-interpreted descriptors would provide direct support for the naming scope rules of higher level languages in the B5000 computer system.” We found this quote on Mark Smotherman’s cool history pages [S04]; see them for more information.

[S04] “System Call Support”

Mark Smotherman, May 2004

<http://people.cs.clemson.edu/~mark/syscall.html>

A neat history of system call support. Smotherman has also collected some early history on items like interrupts and other fun aspects of computing history. See his web pages for more details.

[WL+93] “Efficient Software-based Fault Isolation”

Robert Wahbe, Steven Lucco, Thomas E. Anderson, Susan L. Graham

SOSP ’93

A terrific paper about how you can use compiler support to bound memory references from a program, without hardware support. The paper sparked renewed interest in software techniques for isolation of memory references.

[W17] Answer to footnote: “Is there anything other than havoc that can be wreaked?”

Waciuma Wanjohi, October 2017

Amazingly, this enterprising reader found the answer via google’s Ngram viewing tool (available at the following URL: <http://books.google.com/ngrams>). The answer, thanks to Mr. Wanjohi: “It’s only since about 1970 that ‘wreak havoc’ has been more popular than ‘wreak vengeance’. In the 1800s, the word wreak was almost always followed by ‘his/their vengeance.’” Apparently, when you wreak, you are up to no good, but at least breakers have some options now.

16 Segmentation

これまで、各プロセスのアドレス空間全体をメモリに入れていました。ベースレジスタと境界レジスタを使用すると、OSは物理メモリの異なる部分にプロセスを簡単に再配置できます。しかし、あなたはこれらのアドレス空間について興味深いことに気づいたかもしれません。スタックとヒープの間であり中央に大きな空き領域があります。

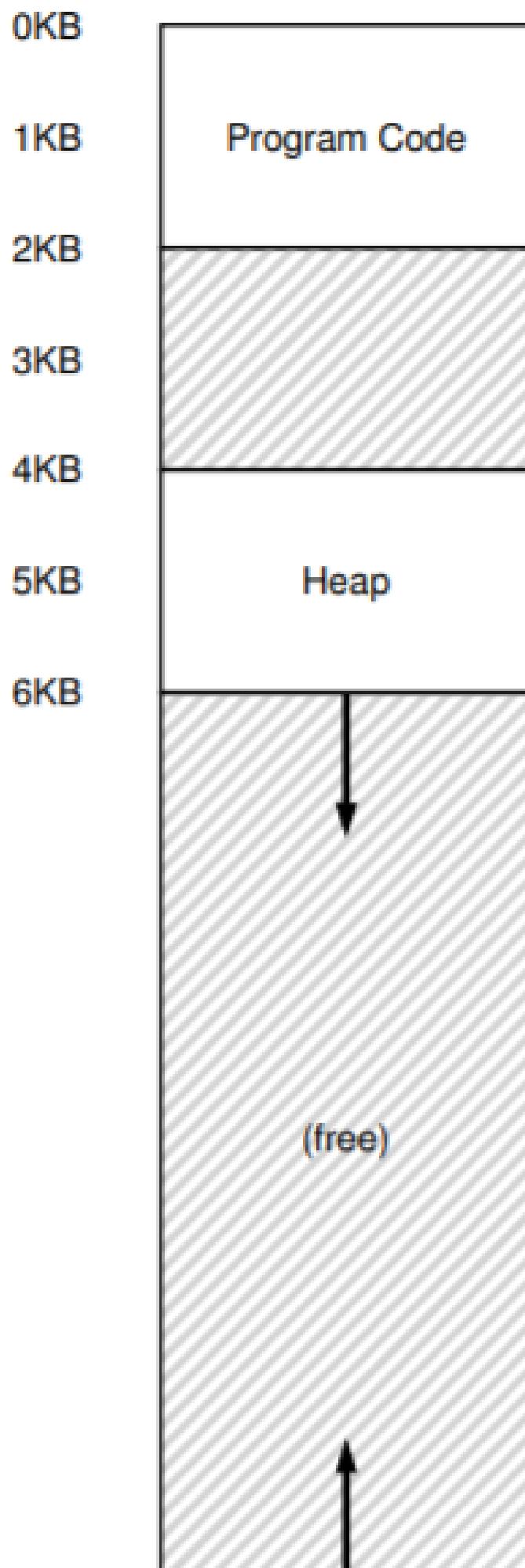


図 16.1 から分かるように、スタックとヒープの間のスペースはプロセスによって使用されていませんが、物理メモリのどこかにアドレス空間全体を再配置すると、物理メモリを占有しています。したがって、メモリを仮想化するためにベースレジスタと境界レジスタのペアを使用する単純なアプローチは無駄です。また、アドレス空間全体がメモリに収まらないときは、プログラムを実行するのが非常に難しくなります。したがって、ベースと境界は、私たちが思っているほど柔軟ではありません。

THE CRUX: HOW TO SUPPORT A LARGE ADDRESS SPACE

スタックとヒープの間に（潜在的に）空き領域が多い大きなアドレス空間をどのようにサポートしますか？我々の例では、小さな（ふりがな）アドレススペースで、無駄がそこまでないように見えることに注意してください。もし、32 ビットのアドレス空間（4 GB のサイズ）であつたらという状況を想像してみてください。一般的なプログラムは MB のメモリしか使用しませんが、アドレス空間全体がメモリに常駐することを要求するので大きな無駄になってしまいます。

16.1 Segmentation: Generalized Base/Bounds

この問題を解決するために、アイディアが生まれました。これをセグメンテーションと呼びます。少なくとも 1960 年代初頭と同じくらい古いものであることは、かなり古い考えです [H61, G62]。アイディアは簡単です。私たちの MMU にはベースと境界のペアが 1 つずつあるのではなく、アドレス空間の論理セグメントごとにベースと境界のペアがないのはなぜですか？セグメントは、特定の長さのアドレス空間のちょうど連続した部分であり、正規アドレス空間には、コード、スタック、およびヒープの 3 つの論理的に異なるセグメントがあります。セグメンテーションでは、物理メモリの異なる部分にこれらのセグメントを配置し、物理メモリに未使用の仮想アドレス空間を埋め込まないようにします。

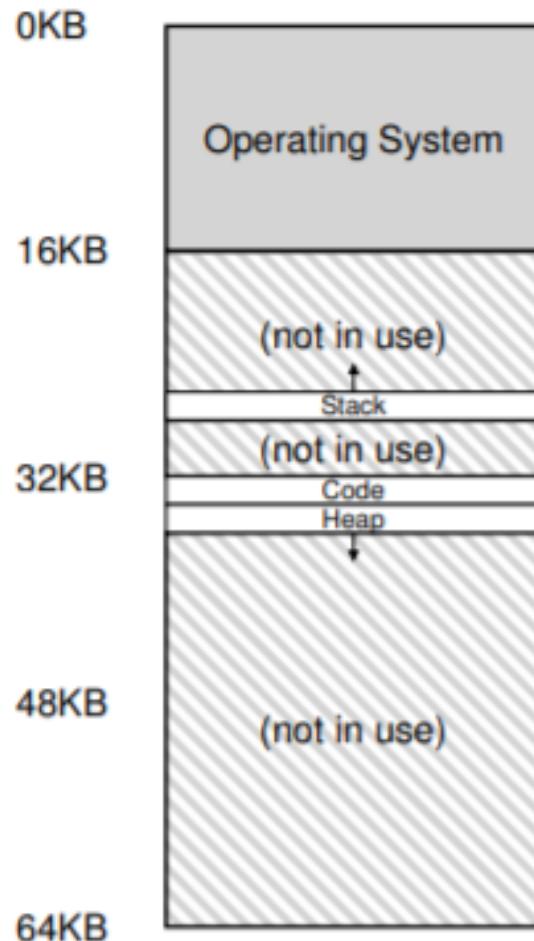


Figure 16.2: Placing Segments In Physical Memory

例を見てみましょう。図 16.1 のアドレス空間を物理メモリに配置したいとします。セグメントごとにベースと境界のペアを使用して、各セグメントを物理メモリに独立して配置することができます。たとえば、図 16.2 を参照してください。そこには 3 つのセグメントがあります。64KB の物理メモリー (および OS 用に 16KB が予約されています) が表示されます。

図でわかるように、使用されているメモリだけが物理メモリの領域を割り当てられているため、大量の未使用アドレス空間 (疎なアドレス空間と呼ばれることもあります) を持つ大きなアドレス空間に対応できます。セグメンテーションをサポートするために必要な私たちの MMU のハードウェア構造は、すでに知っているものです。この場合は、3 つのベースとバウンドのレジスタペアのセットです。下の図 16.3 に、上記の例のレジスタ値を示します。各境界レジスタはセグメントのサイズを保持します。

Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

Figure 16.3: Segment Register Values

図から分かるように、コードセグメントは物理アドレス 32KB に配置され、サイズは 2KB、ヒープセグメントは 34KB に配置され、サイズは 2KB です。

図 16.1 のアドレス空間を使って変換例を見てみましょう。仮想アドレス 100(コードセグメント内にある)への参照が行われたと仮定します。参照が行われると(命令フェッチなど)、ハードウェアはこのセグメント(この場合は 100)のオフセットにベース値を加算して、100 + 32KB または 32868 の物理アドレスに到達します。アドレスが境界内にあることを確認し(100 が 2KB 未満)、それが存在することを確認し、物理メモリアドレス 32868 への参照を発行します。

ASIDE: THE SEGMENTATION FAULT

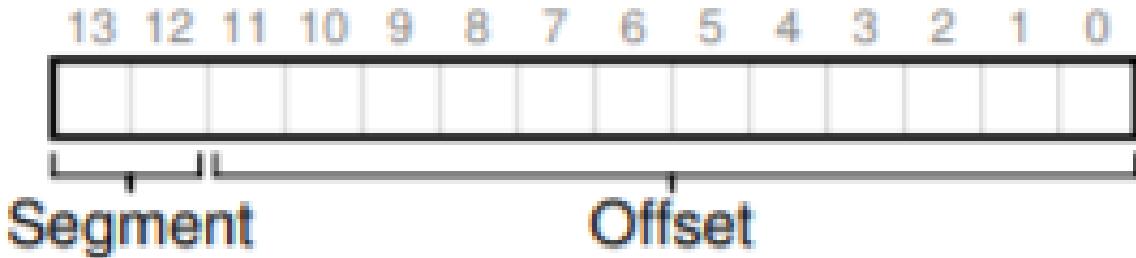
セグメンテーション違反または違反という用語は、セグメント化されたマシン上での不正なアドレスへのメモリアクセスによって発生します。ユーモラスな言い方をすれば、セグメンテーションをまったくサポートしていないマシンであっても、この言葉はそのままです。あなたのコードがフォールトを起こしている理由を理解できなければ、ユーモラスなこともあります。

次に、ヒープ内のアドレス、仮想アドレス 4200 を見てみましょう(図 16.1 を参照)。最初に行うべきことは、ヒープへのオフセット、すなわちアドレスが参照するこのセグメントのどのバイトを抽出するかです。ヒープは仮想アドレス 4KB(4096) から開始するので、4200 のオフセットは実際に 4200 マイナス 4096 つまり 104 です。次に、このオフセット(104)を取得し、ベースレジスタの物理アドレス(34K)に追加して、望ましい結果を得ます: 34920。

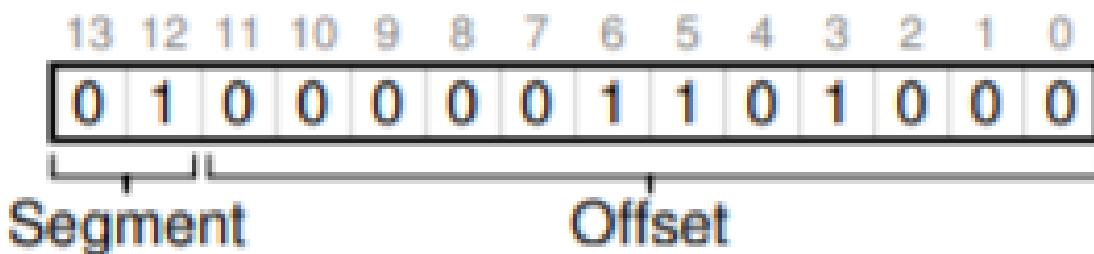
ヒープの終わりを超えて 7KB などの違法アドレスを参照しようとするとどうなりますか? 何が起こるか想像することができます。ハードウェアはアドレスが範囲外であることを検出し、OS にトラップし、問題のプロセスが終了する可能性があります。そして、すべての C プログラマーが恐怖を覚える有名な用語の起源、つまりセグメンテーション違反を知りました。

16.2 Which Segment Are We Referring To?

ハードウェアは、変換中にセグメントレジスタを使用します。セグメントへのオフセット、およびアドレスが参照するセグメントをどのように知っていますか? 明示的アプローチと呼ばれることがある一般的なアプローチの 1 つは、アドレス空間を仮想アドレスの上位数ビットに基づいてセグメントに分割することです。この技法は VAX/VMS システム [LL82] で使用されていました。上記の例では、3 つのセグメントがあります。したがって、私たちの仕事を達成するには 2 ビットが必要です。14 ビット仮想アドレスの上位 2 ビットを使用してセグメントを選択すると、仮想アドレスは次のようにになります。



この例では、上位 2 ビットが 00 の場合、ハードウェアは仮想アドレスがコードセグメント内にあることを認識し、コードベースと境界のペアを使用してアドレスを正しい物理位置に再配置します。上位 2 ビットが 01 の場合、ハードウェアはアドレスがヒープにあることを認識し、ヒープのベースと境界を使用します。これを明確にするために、ヒープ仮想アドレスを上から (4200) 取り出して変換してみましょう。バイナリ形式の仮想アドレス 4200 がここに表示されます。



画像からわかるように、上位 2 ビット (01) は、どのセグメントを参照しているかをハードウェアに伝えます。下位 12 ビットはセグメントへのオフセットです : 0000 0110 1000、または 16 進数は 0x068 または 104 です。したがって、ハードウェアは、どのセグメントレジスタを使用するかを決定するために最初の 2 ビットを取り、次にセグメントへのオフセットとして次の 12 ビットを取ります。ベースレジスタをオフセットに加えることによって、ハードウェアは最終的な物理アドレスになります。オフセットは境界チェックを容易にします。オフセットが境界よりも小さいかどうかを簡単に確認できます。もしそうでなければ、アドレスは不正です。したがって、ベースと境界が配列 (セグメントごとに 1 つのエントリ) であれば、ハードウェアは次のようにして目的の物理アドレスを取得します。

```
// get top 2 bits of 14-bit VA
Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
// now get offset
Offset = VirtualAddress & OFFSET_MASK
if (Offset >= Bounds[Segment])
    RaiseException(PROTECTION_FAULT)
else
    PhysAddr = Base[Segment] + Offset
    Register = AccessMemory(PhysAddr)
```

実行中の例では、上の定数の値を記入することができます。具体的には、SEG MASK は 0x3000、SEG SHIFT は 12、OFFSET MASK は 0xFFFF に設定されます。

また、上位 2 ビットを使用し、3 つのセグメント (コード、ヒープ、スタック) しか持たない場合、アドレス空間の 1 つのセグメントが使用されなくなることに気付くかもしれません。したがって、一部のシステムでは、ヒープと同じセグメントにコードを配置し、したがって、使用するセグメントを選択するために 1 ビット

のみを使用します [LL82]。

特定のアドレスがどのセグメントにあるかをハードウェアが判断する他の方法があります。暗黙的なアプローチでは、ハードウェアはアドレスがどのように形成されたかを知ることによってセグメントを決定します。例えば、アドレスがプログラムカウンタから生成された（すなわち命令フェッチであった）場合、アドレスはコードセグメント内にあります。アドレスがスタックポインタまたはベースポインタに基づいている場合は、スタックセグメント内になければなりません。他のアドレスはすべてヒープ内になければなりません。

16.3 What About The Stack?

ここまででは、アドレス空間の重要な要素の 1 つ、スタックを除外しました。スタックは上の図の物理アドレス 28KB に再配置されていますが、重要な違いが 1 つあります。スタックは後退します。物理メモリでは、28KB から始まり、仮想アドレス 16KB から 14KB に対応する 26KB にまで拡大します。変換は異なる方法で進めなければなりません。

Segment	Base	Size	Grows Positive?
Code	32K	2K	1
Heap	34K	2K	1
Stack	28K	2K	0

Figure 16.4: Segment Registers (With Negative-Growth Support)

まず必要なのは、ハードウェアの追加サポートです。ハードウェアは、ベースと境界の値の代わりに、セグメントがどのように成長するかを知る必要があります（たとえば、セグメントが正の方向に成長すると 1 に設定され、負の場合は 0 に設定されます）。ハードウェアが追跡している更新されたビューを図 16.4 に示します。

セグメントが負の方向に成長することをハードウェアが理解すると、ハードウェアはこのような仮想アドレスをわずかに異なる形で変換する必要があります。スタック仮想アドレスの例を取り上げ、それを翻訳してプロセスを理解してみましょう。

この例では、仮想アドレス 15KB にアクセスすることを想定しています。物理アドレス 27KB にマップする必要があります。私たちの仮想アドレスはバイナリ形式で、11 1100 0000 0000(hex 0x3C00) のようになります。ハードウェアは上位 2 ビット (11) を使用してセグメントを指定しますが、オフセットは 3KB です。正しい負のオフセットを得るには、3KB から最大セグメントサイズを減算する必要があります。この例では、セグメントは 4KB になりますので、正しい負のオフセットは 3KB マイナス 4KB -1KB です。正しい物理アドレス (27KB) に達するように、マイナスオフセット (-1KB) をベース (28KB) に追加するだけです。境界チェックは、負のオフセットの絶対値がセグメントのサイズより小さいことを確認することによって計算できます。

16.4 Support for Sharing

セグメンテーションのサポートが増えるにつれて、システム設計者はもう少しハードウェアをサポートして新しいタイプの効率を実現できることをすぐに認識しました。具体的には、メモリを節約するために、アドレス空間間で特定のメモリセグメントを共有すると便利なことがあります。特に、今日のシステムではコード共有が一般的であり、依然として使用されています。

共有をサポートするためには、保護ビットの形でハードウェアから少し余分なサポートが必要です。基本的なサポートは、セグメントごとに数ビットを追加し、プログラムがセグメントを読み書きできるかどうか、またはセグメント内にあるコードを実行するかどうかを示します。コードセグメントを読み取り専用に設定することにより、分離を害することなく、同じコードを複数のプロセスにわたって共有することができます。各プロセスはそれ自身のプライベートメモリにアクセスしていると考えていますが、OS は秘密にプロセ

スによって変更できないメモリを共有しているので、錯覚は保持されます。

Segment	Base	Size	Grows Positive?	Protection
Code	32K	2K	1	Read-Execute
Heap	34K	2K	1	Read-Write
Stack	28K	2K	0	Read-Write

Figure 16.5: Segment Register Values (with Protection)

ハードウェア (および OS) によって追跡される追加情報の例を図 16.5 に示します。ご覧のように、コードセグメントは読み込みと実行が設定されているため、メモリ内の同じ物理セグメントを複数の仮想アドレス空間にマップできます。

保護ビットを使用すると、前述のハードウェアアルゴリズムも変更する必要があります。仮想アドレスが境界内にあるかどうかを検査することに加えて、ハードウェアはまた、特定のアクセスが許可されるかどうかをチェックしなければいけません。ユーザープロセスが読み取り専用セグメントに書き込もうとするか、実行不可能なセグメントから実行しようとすると、ハードウェアは例外を発生させ、OS に問題のプロセスを処理させる必要があります。

16.5 Fine-grained vs. Coarse-grained Segmentation

これまでの例のほとんどは、ほんのわずかのセグメント (コード、スタック、ヒープ) を持つシステムに焦点を当てていました。このセグメンテーションは、アドレス空間を比較的大きく粗い塊に分割するので、粗い粒度を考えることができます。しかしながら、いくつかの初期のシステム (例えば、Multics [CV65、DD68]) は、より柔軟であり、アドレス空間が細かいセグメント化と呼ばれる多数のより小さいセグメントから構成されることが可能でした。

多くのセグメントをサポートするには、ハードウェアのサポートをさらに必要とし、ある種のセグメントテーブルをメモリに格納します。このようなセグメントテーブルは、通常、非常に多数のセグメントの作成をサポートし、したがって、システムが、これまで説明したよりも柔軟性の高い方法でセグメントを使用できるようになります。たとえば、Burroughs B5000 のような初期のマシンでは何千ものセグメントがサポートされていたため、コンパイラはコードとデータを別々のセグメントに分割して OS とハードウェアがサポートすることを期待していました [RK68]。当時の考えは、セグメントを細かく分割することによって、どのセグメントが使用されているか、どのセグメントが使用されていないかをより良く知ることができ、主メモリをより効率的に利用できるようになりました。

16.6 OS Support

セグメンテーションの仕組みについての基本的な考え方が必要です。システムが動作するにつれてアドレス空間の一部が物理メモリに再配置されるため、アドレス空間全体に対して単一のベース/境界ペアを使用した簡単なアプローチと比較して、物理メモリの大幅な節約が達成されます。具体的には、スタックとヒープの間の未使用領域をすべて物理メモリに割り当てる必要はなく、より多くのアドレス空間を物理メモリに収めることができます。

しかし、セグメンテーションはいくつかの新しい問題を引き起こします。最初に、対処しなければならない新しい OS の問題について説明します。今まで述べてきたものは古いものです。コンテキストスイッチで OS は何をすべきですか？ それは、セグメントレジスタを保存して復元する必要があります。明らかに、各プロセスには独自の仮想アドレス空間があり、OS はプロセスを再実行する前にこれらのレジスタを正しく設定する必要があります。

第2の重要な問題は、物理メモリの空き領域を管理することです。新しいアドレス空間が作成されると、OSはそのセグメントの物理メモリ内の領域を見つけることができなければなりません。以前は、各アドレス空間が同じサイズであると仮定していたため、物理メモリはプロセスが収まるスロットの束と考えることができました。プロセスごとに多数のセグメントがあり、各セグメントは異なるサイズになっています。

一般的な問題は、物理メモリがすぐに空き領域の小さな穴で満たされ、新しいセグメントを割り当てるごとに既存のセグメントを拡張することが困難になることです。この問題を外部断片化と呼んでいます[R69]。図16.6を参照してください。

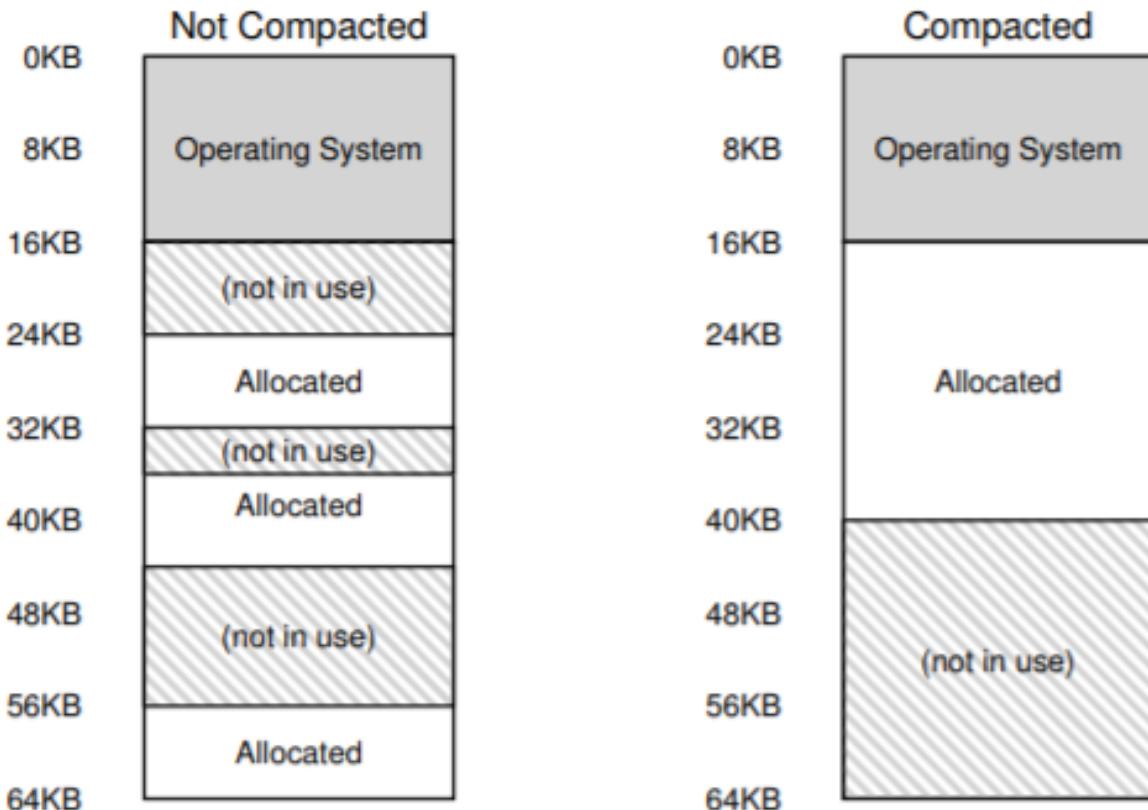


Figure 16.6: Non-compacted and Compacted Memory

この例では、プロセスが20KBのセグメントを割り当てようとしています。この例では、24KBの空きがありますが、1つの連続したセグメントではありません（むしろ3つの連続していないチャンク）。したがって、OSは20KB要求を満たすことができません。

この問題に対する1つの解決策は、既存のセグメントを再配置することによって物理メモリをコンパクト化することです。例えば、OSは実行中のプロセスを停止し、それらのデータを1つの連続したメモリ領域にコピーし、それらのセグメントレジスタ値を新しい物理位置を指すように変更することができます。したがって、これにより、OSは新しい割り当て要求を成功させることができます。しかし、コピー・セグメントはメモリーを消費し、一般的にかなりの量のプロセッサー時間を使用するため、圧縮はコストがかかります。圧縮された物理メモリの図については、図16.6(右)を参照してください。

よりシンプルな方法は、割り当てのために利用可能な大量のメモリを保持しようとするフリーリスト管理アルゴリズムを使用することです。ベストフィット（空きスペースのリストを保持し、リクエストへの望ましい割り当てを満たすサイズに最も近いものを返す）、ワーストフィット、ファーストフィットなどの古典的なアルゴリズムを含む、人々が取ったアプローチは文字通り何百もあります。バディアルゴリズム[K68]のようなより複雑なスキームもあります。

Wilson らによる優れた調査によると、このようなアルゴリズム [W+95] の詳細を知りたい場合や、後の章で基本のいくつかをカバーするまでお待ちください。残念なことに、たとえどんなにスマートなアルゴリズムであっても、外部の断片化は依然として存在します。従って、良いアルゴリズムはそれを最小化しようとするだけです。

TIP: IF 1000 SOLUTIONS EXIST, NO GREAT ONE DOES

外部の断片化を最小限に抑えるために非常に多くの異なるアルゴリズムが存在するという事実は、より根底にある真実を示唆しています。問題を解決する最良の方法はありません。したがって、我々は合理的な何かのために解決し、十分に良くなることを願っています。唯一の実際の解決策(次の章で説明します)は、可変サイズのチャンクにメモリを決して割り当てないことによって、問題を完全に回避することです。

16.7 Summary

セグメンテーションは多くの問題を解決し、メモリのより効果的な仮想化を構築するのに役立ちます。ダイナミックリロケーションだけでなく、セグメント化は、アドレス空間の論理セグメント間の巨大な潜在的なメモリの浪費を避けることによって、無駄にあいているアドレス空間をよりよくサポートすることができます。

算術セグメント化が容易で、ハードウェアにも適しているため、高速です。変換のオーバーヘッドは最小限に抑えられます。フリンジの利益も発生します。それはコード共有です。コードが別のセグメント内に配置されている場合、そのセグメントは実行中の複数のプログラム間で共有される可能性があります。

しかし、私たちが学んだように、可変サイズのセグメントをメモリに割り当てるには、克服したいいくつかの問題につながります。最初に述べたように、外部の断片化です。セグメントは変数化されているため、空きメモリが奇数の部分に細分化されるため、メモリ割り当て要求を満たすことは困難です。スマートアルゴリズム [W+95] を使用することも、周期的にコンパクトなメモリを使用することもできますが、この問題は基本的に避けがたいものです。

第2の、そしておそらくより重要な問題は、セグメンテーションが、完全に一般化された無駄にあいているアドレス空間をサポートするのに十分柔軟でないことです。たとえば、1つの論理セグメント内に大規模だが、まばらに使用されるヒープがある場合、ヒープ全体がアクセスされるためにはまだメモリに常駐する必要があります。言い換れば、アドレス空間がどのように使用されているかのモデルが、基礎となるセグメンテーションの設計方法と正確に一致しない場合、セグメンテーションはうまく機能しません。したがって、新しいソリューションを見つける必要があります。それらを見つける準備ができていますか？

参考文献

- [CV65] “Introduction and Overview of the Multics System”
F. J. Corbato and V. A. Vyssotsky
Fall Joint Computer Conference, 1965
One of five papers presented on Multics at the Fall Joint Computer Conference; oh to be a fly on the wall in that room that day!
- [DD68] “Virtual Memory, Processes, and Sharing in Multics”
Robert C. Daley and Jack B. Dennis
Communications of the ACM, Volume 11, Issue 5, May 1968
An early paper on how to perform dynamic linking in Multics, which was way ahead of its time. Dynamic linking finally found its way back into systems about 20 years later, as the large X-windows libraries

demanded it. Some say that these large X11 libraries were MIT's revenge for removing support for dynamic linking in early versions of UNIX!

[G62] "Fact Segmentation"

M. N. Greenfield

Proceedings of the SJCC, Volume 21, May 1962

Another early paper on segmentation; so early that it has no references to other work.

[H61] "Program Organization and Record Keeping for Dynamic Storage"

A. W. Holt

Communications of the ACM, Volume 4, Issue 10, October 1961

An incredibly early and difficult to read paper about segmentation and some of its uses.

[I09] "Intel 64 and IA-32 Architectures Software Developer's Manuals"

Intel, 2009

Available: <http://www.intel.com/products/processor/manuals>

Try reading about segmentation in here (Chapter 3 in Volume 3a); it'll hurt your head, at least a little bit.

[K68] "The Art of Computer Programming: Volume I"

Donald Knuth

Addison-Wesley, 1968

Knuth is famous not only for his early books on the Art of Computer Programming but for his typesetting system TeX which is still a powerhouse typesetting tool used by professionals today, and indeed to typeset this very book. His tomes on algorithms are a great early reference to many of the algorithms that underly computing systems today.

[L83] "Hints for Computer Systems Design"

Butler Lampson

ACM Operating Systems Review, 15:5, October 1983

A treasure-trove of sage advice on how to build systems. Hard to read in one sitting; take it in a little at a time, like a fine wine, or a reference manual.

[LL82] "Virtual Memory Management in the VAX/VMS Operating System"

Henry M. Levy and Peter H. Lipman

IEEE Computer, Volume 15, Number 3 (March 1982)

A classic memory management system, with lots of common sense in its design. We'll study it in more detail in a later chapter.

[RK68] "Dynamic Storage Allocation Systems"

B. Randell and C.J. Kuehner

Communications of the ACM

Volume 11(5), pages 297-306, May 1968

A nice overview of the differences between paging and segmentation, with some historical discussion of various machines.

[R69] "A note on storage fragmentation and program segmentation"

Brian Randell

Communications of the ACM

Volume 12(7), pages 365-372, July 1969

One of the earliest papers to discuss fragmentation.

[W+95] "Dynamic Storage Allocation: A Survey and Critical Review"

Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles
In International Workshop on Memory Management
Scotland, United Kingdom, September 1995
A great survey paper on memory allocators.

17 Free-Space Management

この章では、malloc ライブラリ (プロセスのヒープのページを管理する) であろうと、OS 自体 (アドレスの一部を管理するものであろうと) を問わず、メモリ管理システムの基本的な側面について議論するために、プロセスのスペース)。具体的には、free space management を取り巻く課題について議論していきます。

問題をより具体的にしましょう。空き領域を管理することは確実に簡単です。ページングの概念について議論するときにわかります。管理しているスペースが固定サイズのユニットに分割されている場合は簡単です。このような場合、これらの固定サイズのユニットのリストを保持するだけです。クライアントがそのうちの1つを要求すると、最初のエントリを返します。

free space management がより難しくなる (興味深い) のは、管理している空き領域が可変サイズのユニットで構成されている場合です。これは、ユーザレベルのメモリ割り当てライブラリ (malloc() および free()) や、セグメンテーションを使用して仮想メモリを実装する際に物理メモリを管理する OS で発生します。どちらの場合でも、存在する問題は外部断片化として知られています。空き領域はサイズの小さな断片に細断され、断片化されます。空き領域の合計量が要求のサイズを超えていても、要求を満たすことができる連続した1つの領域がないため、後続の要求が失敗する可能性があります。



図はこの問題の例を示しています。この場合、使用可能な空き領域の合計は 20 バイトです。残念ながら、サイズ 10 の 2 つのチャンクに分割されています。その結果、20 バイトの空きがあっても 15 バイトの要求は失敗します。そこで、この章で取り上げる問題に到達します。

CRUX: HOW TO MANAGE FREE SPACE

可変サイズの要求を満たす場合、空き領域をどのように管理するべきか？ 断片化を最小限に抑えるためにどのような戦略を使用できますか？ 代替アプローチの時間と空間のオーバーヘッドはどのくらいですか？

17.1 Assumptions

この議論の大部分は、ユーザレベルのメモリ割り当てライブラリにあるアロケータの偉大な歴史に焦点を当てています。私たちはウィルソンの優れた調査 [W+95] を参考にしていますが、興味を持った読者がソースドキュメント自体に行くことを勧めます。

malloc() と free() が提供するような基本的なインターフェースを想定しています。具体的には、void * malloc(size t size) は、アプリケーションによって要求されたバイト数である size という単一のパラメータをとります。そのサイズの領域 (またはそれ以上の大きさ) にポインタ (特定の型のないポインタ、または C 言語の void ポインタ) を返します。void free(void * ptr) はポインタをとり、対応するチャンクを解放します。インターフェイスの意味に注意してください。ユーザーはスペースを解放するときにライブラリにサイズを通知しません。したがって、ライブラリは、メモリへのポインタが渡されたときに、メモリのチャンクがどれだけ大きいかを把握できる必要があります。この章の後半でこれを行う方法について説明します。

このライブラリが管理する領域は歴史的にヒープとして知られており、ヒープの空き領域を管理するために

使用される汎用データ構造はどちらかのフリーリストです。この構造体には、メモリの管理対象領域内の空きチャunkのすべてへの参照が含まれます。もちろん、このデータ構造はリストそのものでなくとも、空き領域を追跡するためのデータ構造の一種である必要はありません。

さらに、主に、前述のように外部断片化に関心があると仮定します。アロケータは、もちろん、内部断片化の問題を抱えている可能性もあります。割り振り者が要求された量よりも多くのメモリを渡すと、そのようなチャunk内の未使用の（したがって未使用の）スペースは内部断片化とみなされます（割り当てられたユニットの内部で発生するため）。しかし、単純化のため、主に外部フラグメンテーションに焦点を当てます。

また、クライアントにメモリを渡すと、メモリ内の別の場所に移動することはできません。たとえば、プログラムが `malloc()` を呼び出し、ヒープ内のいくつかの領域へのポインタが与えられた場合、そのメモリ領域は、プログラムが対応するメモリ領域を介して返すまで、プログラムによって“所有”されます（ライブラリによっては移動できません）`free()` を呼び出します。したがって、断片化に対抗するのに有用な空き領域の圧縮は不可能です。しかし、セグメンテーションを実装する際に、断片化に対処するために、空き領域の圧縮をOSで使用することができます（セグメンテーションの章で説明したように）。

最後に、アロケータが連続したバイトの領域を管理していると仮定します。場合によっては、アロケータがその領域の拡張を要求することもできます。例えば、ユーザレベルのメモリ割り当てライブラリは、スペースを使い果たしたときにカーネルを呼び出して（`sbrk` などのシステムコールを介して）ヒープを拡張することができます。しかし、わかりやすくするために、領域は一生を通して固定された单一のサイズであると仮定します。

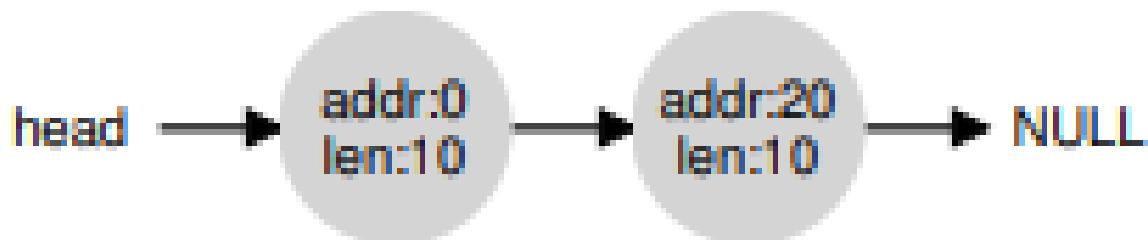
17.2 Low-level Mechanisms

いくつかのポリシーの詳細を調べる前に、まず大部分のアロケータで使用されるいくつかの一般的なメカニズムについて説明します。まず、ほとんどのアロケータで一般的な手法である分割と融合の基本について説明します。次に、割り当てられた領域のサイズをどのように素早く簡単に追跡できるかを示します。最後に、空き領域内に単純なリストを作成し、空き領域と空でない領域を追跡する方法について説明します。

Splitting and Coalescing 空きリストには、まだヒープに残っている空き領域を記述する要素のセットが含まれています。したがって、次の 30 バイトのヒープを想定します。



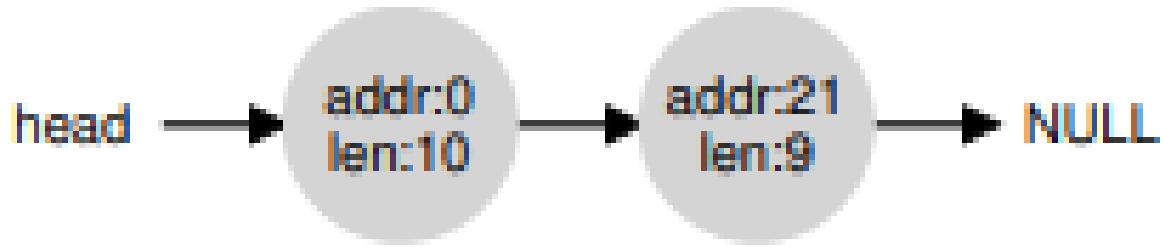
このヒープの空きリストには 2 つの要素があります。1 つのエントリは最初の 10 バイトの空きセグメント（バイト 0~9）を記述し、1 つのエントリは他の空きセグメント（バイト 20~29）を記述します。



上記のように、10 バイトを超えるリクエストは失敗します（NULL を返します）。利用可能なサイズのメモリが連続して 1 つだけはありません。いずれかのフリーチャunkによって、そのサイズ（10 バイト）を正確に満たすことができます。しかし、要求が 10 バイトより小さいものの場合はどうなりますか？

ただ 1 バイトのメモリが要求されているとします。この場合、アロケータは分割と呼ばれるアクションを実行します。要求を満たすことができ、2 つに分割する空きメモリを見つけます。

最初のチャンクは呼び出し元に戻ります。2 番目のチャンクはリストに残ります。したがって、上記の例では、1 バイトの要求が行われ、アロケータが 2 つの要素のうち 2 番目の要素を使用して要求を満たすことにした場合、`malloc()` の呼び出しは 20 を返します 1 バイトの割り当て領域)、リストは次のようにになります。



上記のように、10 バイトを超えるリクエストは失敗します (NULL を返します)。利用可能なサイズのメモリが連続して 1 つだけはありません。いずれかのフリーチャンクによって、そのサイズ (10 バイト) を正確に満たすことができます。しかし、要求が 10 バイトより小さいものの場合はどうなりますか？

ただ 1 バイトのメモリが要求されているとします。この場合、アロケータは分割と呼ばれるアクションを実行します。要求を満たすことができ、2 つに分割する空きメモリを見つけます。

最初のチャンクは呼び出し元に戻ります。2 番目のチャンクはリストに残ります。したがって、上記の例では、1 バイトの要求が行われ、アロケータが 2 つの要素のうち 2 番目の要素を使用して要求を満たすことにした場合、`malloc()` の呼び出しは 20 を返します。1 バイトの割り当て領域)、リストは次のようにになります。



この問題に注意してください。ヒープ全体が空いている間は、見かけ上 10 バイトの 3 つのチャンクに分割されています。したがって、ユーザーが 20 バイトを要求した場合、単純なリストの探索ではそのような空きチャンクを見つけず、失敗を返します。

この問題を回避するためにアロケータが行うことは、メモリのチャンクが解放されたときの空き領域の合体です。アイデアは簡単です：メモリに空きチャンクを返すときは、空き領域の近くのチャンクだけでなく、返すチャンクのアドレスも注意深く見てください。新しく作成されたスペースが既存の空きチャンクの 1 つ（またはこの例では 2 つ）のすぐ隣にある場合は、それらを 1 つの大きな空きチャンクにマージします。したがって、合体によって、最終的なリストは次のようにになります。



確かに、これは割り当てが行われる前のヒープリストの最初のようなものです。合体により、アロケータは、

アプリケーションに大きな空き領域があることを確認できます。

Tracking The Size Of Allocated Regions

`free(void * ptr)` のインターフェースが `size` パラメータを取っていないことに気づいたかもしれません。`malloc` ライブライアリは解放されたメモリ領域のサイズを素早く決定し、空きリストにスペースを組み込むことができると仮定されています。

このタスクを達成するために、大部分のアロケータは、メモリ内に保持されたヘッダブロックに、通常はメモリのチャンクの直前に少しの余分な情報を格納します。もう一度例を見てみましょう(図 17.1)。この例では、`ptr` が指す、20 バイトの割り当てブロックを調べています。ユーザが `malloc()` と呼ばれ、その結果を `ptr` に格納すると想像してください。たとえば、`ptr = malloc(20);` となります。

ヘッダーには、割り当てられた領域のサイズが最小限含まれます。(この場合は 20) それは、割り当て解除を高速化するための追加ポインタ、追加の完全性チェックを提供するためのマジックナンバー、および他の情報を含むこともできます。領域のサイズとマジックナンバーを含む単純なヘッダを仮定しましょう。

```
typedef struct __header_t {
    int size;
    int magic;
} header_t;
```

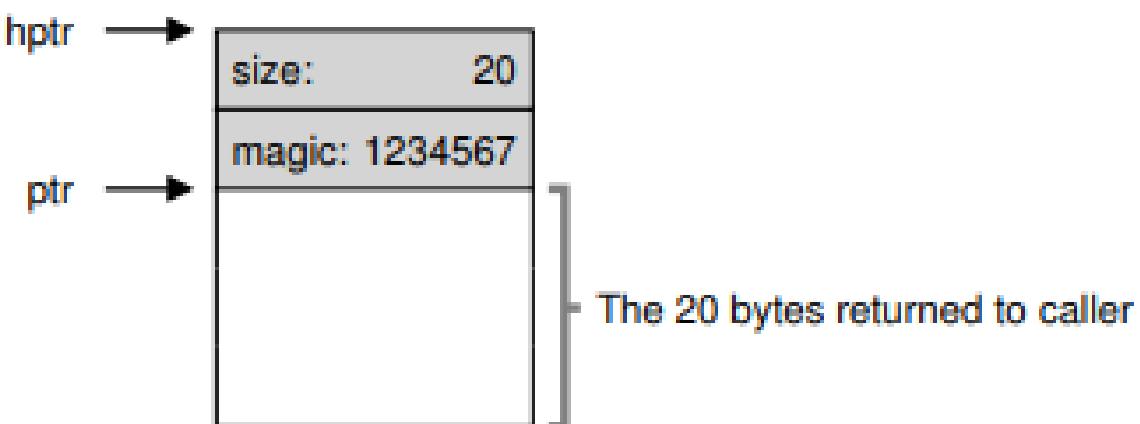


Figure 17.2: Specific Contents Of The Header

上記の例は、図 17.2 のようになります。ユーザーが `free(ptr)` を呼び出すと、ライブラリは単純なポインタ演算を使用して、ヘッダーの開始位置を特定します。

```
void free(void *ptr) {
    header_t *hptr = (void *)ptr - sizeof(header_t);
    ...
}
```

このようなヘッダへのポインタを取得した後、ライブラリはマジックナンバーがサニティチェック(`assert(hptr-> magic == 1234567)`)として期待値と一致するかどうかを簡単に判断し、新たに解放された領域の合計サイズを単純な数学(すなわち、領域のサイズにヘッダのサイズを加える)。最後の文では小さいながらも重要なことに注意してください。空き領域のサイズは、ヘッダーのサイズにユーザーに割り当てられたスペースのサイズを加えたものです。したがって、ユーザーが N バイトのメモリーを要求すると、ライブラリはサイズ N の空きチャンクを検索しません。むしろ、サイズ N のフリー チャンクとヘッダーのサイズを

検索します。

Embedding A Free List

これまでのところ、私たちは単純なフリーリストを概念実体として扱ってきました。それはヒープ内のメモリの空きチャンクを記述する単なるリストです。しかし、空き領域の中にこのようなリストをどうやって作りますか？

より典型的なリストでは、新しいノードを割り当てるときに、ノードのためのスペースが必要なときに `malloc()` を呼び出すだけです。残念ながら、メモリ割り当てライブラリ内では、これを行うことはできません。代わりに、空き領域内にリストを構築する必要があります。

4096 バイトのメモリを管理する（つまり、ヒープは 4KB）と仮定します。これをフリーリストとして管理するには、最初に前記リストを初期化する必要があります。最初は、リストにはサイズ 4096（ヘッダサイズを引いたもの）のエントリが 1 つあります。リストのノードの説明は次のとおりです。

```
typedef struct __node_t {
    int size;
    struct __node_t *next;
} node_t;
```

次に、ヒープを初期化し、空きリストの最初の要素をそのスペース内に配置するコードを見てみましょう。ヒープは、システムコール `mmap` の呼び出しによって取得された空き領域内に構築されていると仮定しています。このようなヒープを作成する唯一の方法ではありませんが、この例ではうまく機能します。

```
// mmap() returns a pointer to a chunk of free space
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_ANON|MAP_PRIVATE, -1, 0);
head->size = 4096 - sizeof(node_t);
head->next = NULL;
```

このコードを実行すると、リストのステータスは、サイズが 4088 の単一のエントリを持つことになります。はい、これは小さなヒープですが、ここではわかりやすい例です。ヘッドポインタはこの範囲の先頭アドレスを含みます。それが 16KB であると仮定しましょう（仮想アドレスは問題ありません）。視覚的には、ヒープは図 17.3 のようになります。

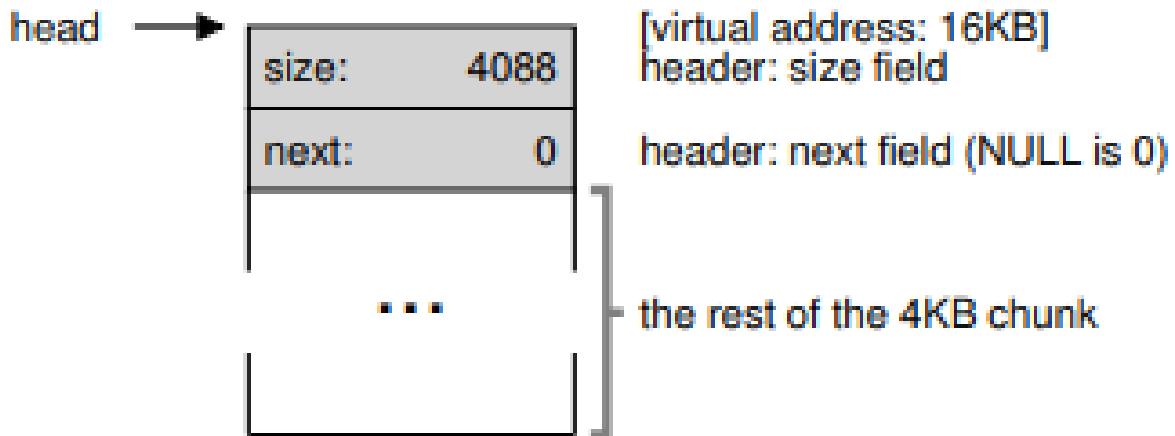


Figure 17.3: A Heap With One Free Chunk

さて、メモリのチャックが要求されたとしましょう。たとえば、100 バイトのサイズです。この要求を処理するために、ライブラリは最初に要求を収容するのに十分な大きさを見つけます。空きチャック（サイズ：4088）が1つしかないため、このチャックが選択されます。次に、チャックは2つに分割されます。1つのチャックは要求（および前述のヘッダー）を処理するのに十分な大きさで、空きチャックは残りのチャックです。8 バイトのヘッダ（整数サイズと整数のマジックナンバー）を仮定すると、ヒープ内のスペースは図 17.4 のようになります。

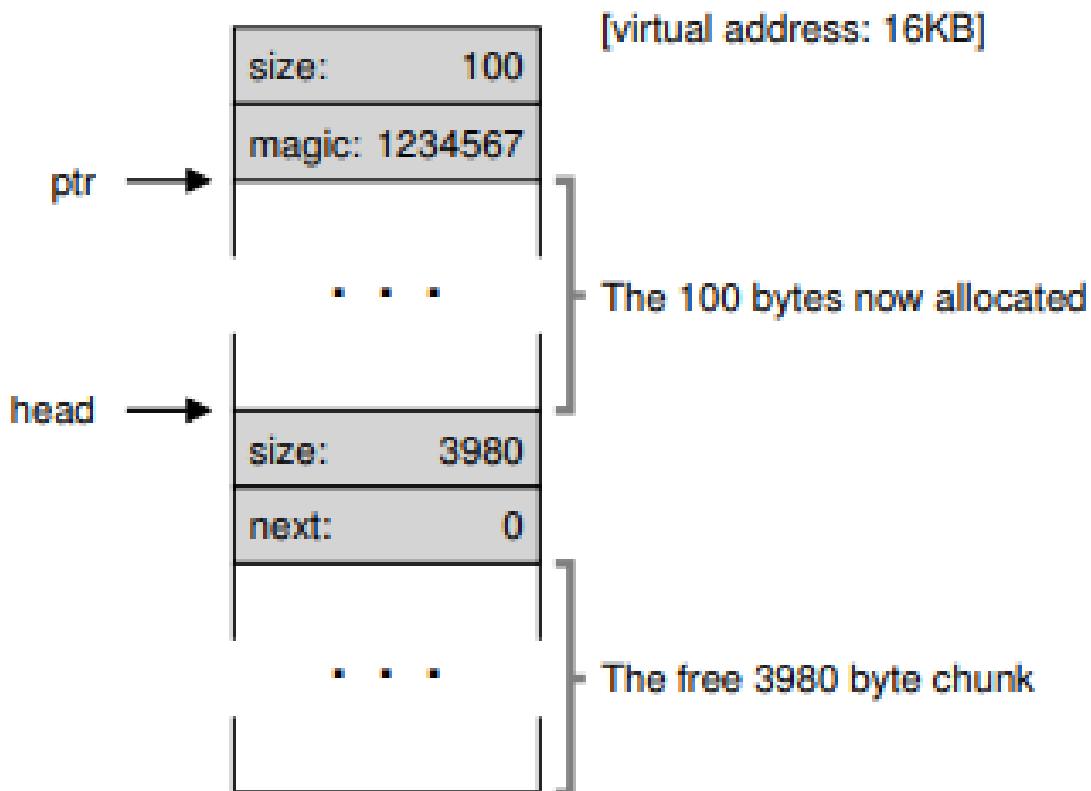


Figure 17.4: A Heap: After One Allocation

したがって、100 バイトの要求時に、ライブラリーは既存の1つの空きチャックから 108 バイトを割り当

て、(上の図で `ptr` とマークされた) ポインターを戻し、割り当てられたスペースの直前のヘッダー情報を後で使用できるように `free()` を呼び出し、リスト内の 1 つの空きノードを 3980 バイト (4088 から 108) に縮小します。

次に、割り当てられた 3 つの領域がある場合のヒープを見てみましょう。各領域は 100 バイト (またはヘッダーを含む 108) です。このヒープの可視化を図 17.5 に示します。

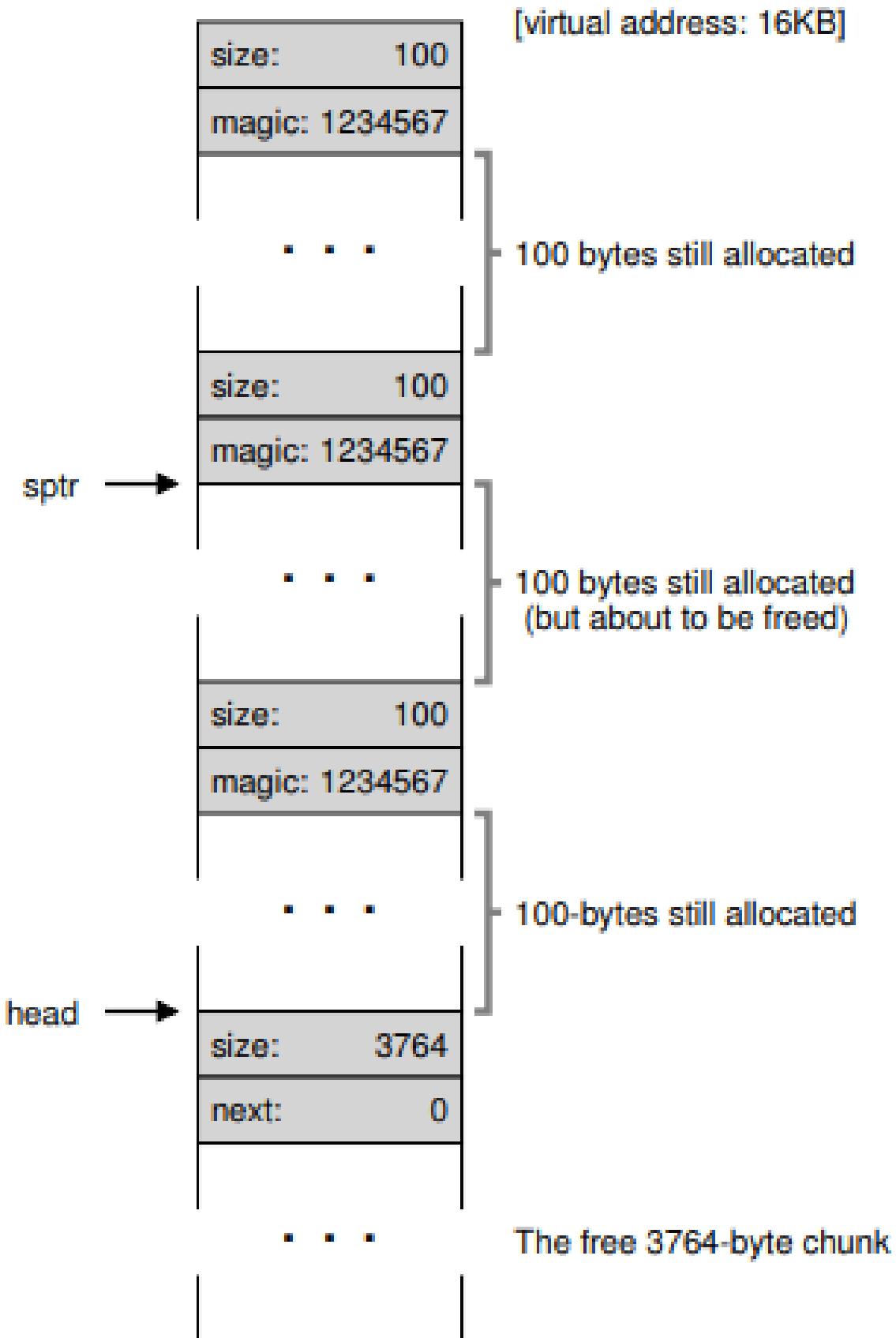


Figure 17.5: Free Space With Three Chunks Allocated

そこから見ることができるように、最初のヒープの 324 バイトが割り当てられているので、その領域に 3 つのヘッダーがあり、呼び出し元のプログラムで 3 つの 100 バイトの領域が使用されています。フリーリストは興味深いもので、ただ 1 つのノード (頭が指している) ですが、今は 3 つの分割後のサイズが 3764 バイトです。しかし、呼び出し元のプログラムが `free()` でメモリを返すとどうなりますか？

この例では、アプリケーションは `free(16500)` を呼び出すことによって割り当てられたメモリの中央のチャンクを返します (値 16500 は、メモリ領域の先頭 16384 を前のチャンクの 108 に追加することによって到着します。このチャンクのヘッダー) この値は前の図にポインタ `sptr` で示されています。

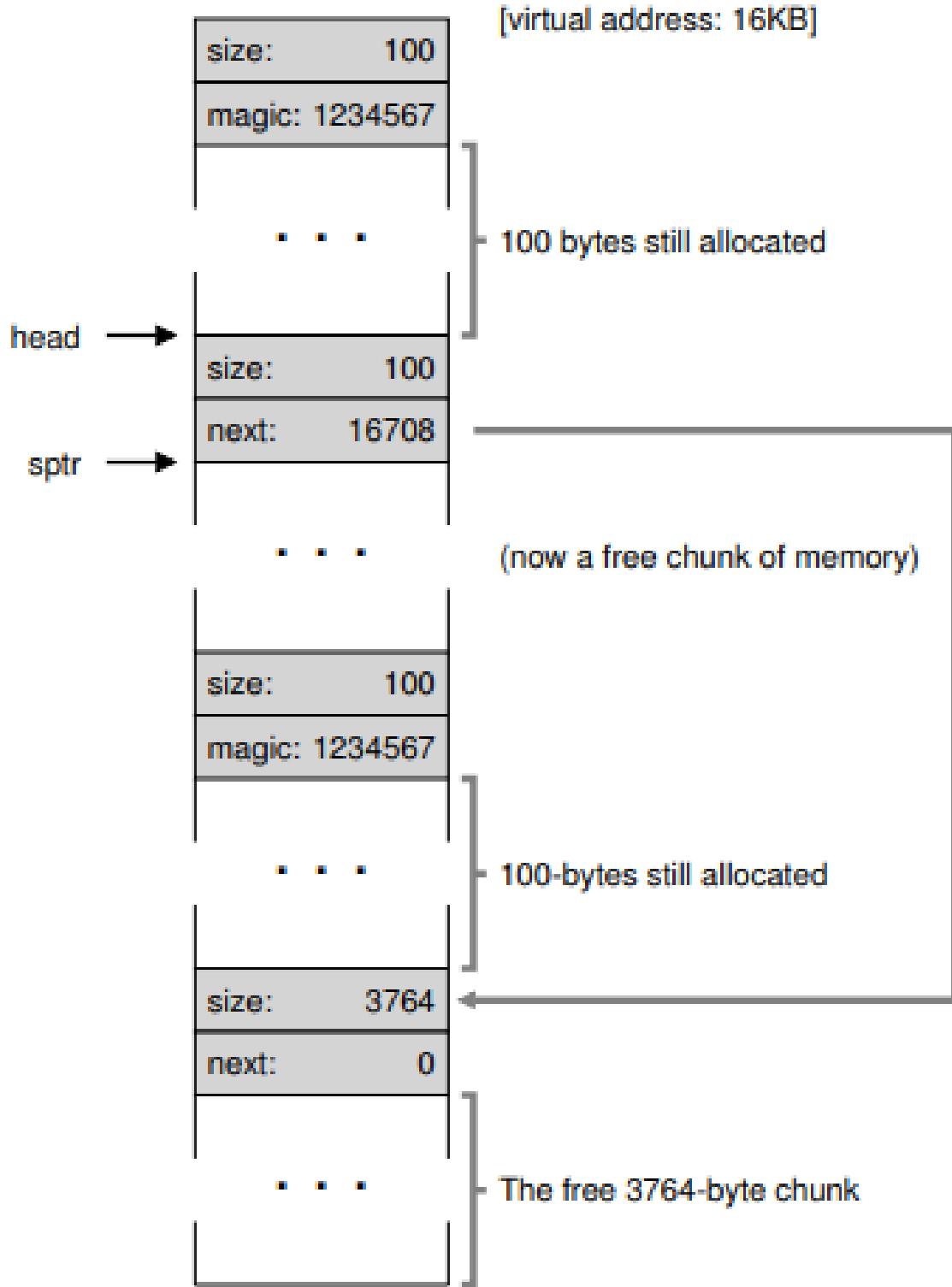


Figure 17.6: Free Space With Two Chunks Allocated

ライブラリは空き領域のサイズをすぐに把握し、空きチャunkを空きリストに戻します。空きリストの先頭に挿入すると、スペースは次のようにになります(図 17.6)。そして今、私たちは小さなフリーチャunk(100 バイト、リストの先頭を指す)と大きな空きチャunk(3764 バイト)で始まるリストを持っています。

私たちのリストには、最終的に1つ以上の要素があります！そして、はい、free space は断片化しています

が、残念ながら一般的なことです。

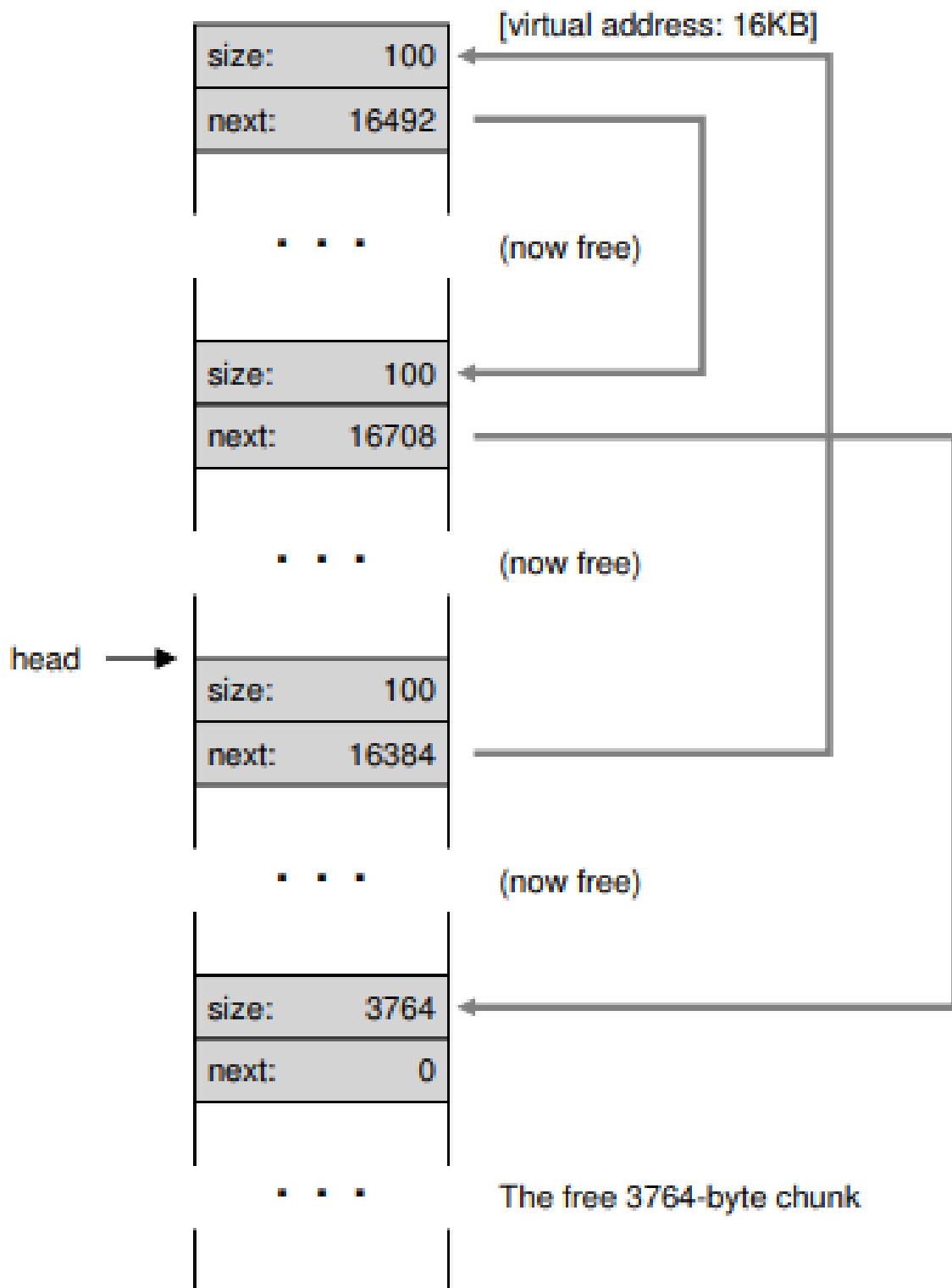


Figure 17.7: A Non-Coalesced Free List

1つの最後の例：最後の 2 つの使用中チャンクが解放されたと仮定しましょう。合体しないと、断片化しているフリーリストになる可能性があります（図 17.7 参照）

図からわかるように、今は大きな混乱があります！ どうしてでしょう？ シンプルなので、リストをまとめ

るのを忘れてしまいました。メモリはすべて free されていますが、断片化したメモリとして見えます。解決策は簡単です：リストを通って隣接チャンクをマージします。終了すると、ヒープは再び全体になります。

Growing The Heap

私たちは、多くの割り当てライブラリ内で見つかった 1 つの最後のメカニズムについて議論すべきです。具体的には、ヒープにスペースがなくなったらどうしたらいいですか？ 最も簡単なアプローチは失敗することです。場合によってはこれが唯一のオプションであるため、NULL を返すことは名誉あるアプローチです。

従来のほとんどのアロケータは、小さなヒープから始めて、ヒープがなくなったときに、OS からより多くのメモリを要求します。通常、これはヒープを成長させてそこから新しいチャンクを割り当てるために何らかの種類のシステムコール（例えば、ほとんどの UNIX システムでは sbrk）を行うことを意味します。sbrk 要求を処理するために、OS は空きの物理ページを見つけ、要求元のプロセスのアドレス空間にマップしてから、新しいヒープの終わりの値を返します。その時点で、より大きいヒープが利用可能であり、要求を正常に処理することができます。

17.3 Basic Strategies

今ではいくつかの機械を手に入れましたので、空き領域を管理するための基本的な戦略について説明しましょう。これらのアプローチは、ほとんどあなたが自分自身を考えることができる非常にシンプルなポリシーに基づいています。

理想的なアロケータは高速であり、断片化を最小限に抑えます。残念なことに、割り振りと空き要求の流れは任意です（結局、プログラマーによって決定されます）。間違った入力を考えると、どのような戦略もかなり悪いことがあります。したがって、私たちは「最良の」アプローチについては説明しませんが、むしろいくつかの基本について話し、賛否両論について議論します。

Best Fit

ベストフィット戦略は非常に単純です。まず、フリーリストを検索し、要求されたサイズよりも大きいか大きいサイズの空きメモリを探します。そのグループの中で最も小さいグループを返します。これはいわゆるベストフィットチャンクです（最小適合とも呼ぶことができます）。空きリストを 1 回通過するだけで正しいブロックが返されます。

ベストフィットの背後にある直感は簡単です。ユーザーが求めるものに近いブロックを返すことで、ベストフィットは無駄なスペースを減らそうとします。しかし、コストがあります。素朴な実装では、正しい空きブロックの網羅的な検索を実行すると、パフォーマンスが大幅に低下します。

Worst Fit

Worst Fit アプローチはベストフィットの逆です。最大のチャンクを見つけて要求された量を返します。残りの（大）チャンクを空きリストに残しておきます。Worst Fit は、ベストフィット手法から生じる可能性のある小さなチャンクの代わりに、大きなチャンクを自由に残そうとします。しかし、もう一度、空き領域を完全に検索する必要があり、この方法はコストがかかる可能性があります。さらに悪いことに、ほとんどの調査ではパフォーマンスが悪く、余分な断片化が発生していますが、依然として高いオーバーヘッドが発生しています。

First Fit

First Fit 方法は、単に十分に大きい最初のブロックを見つけ、要求された量をユーザーに返します。前と同様に、残りの空き領域は、後続の要求で空きにされます。

First Fit はスピードの長所を持っています。空き領域を網羅的に検索する必要はありませんが、小さなオブジェクトで空きリストの先頭を汚染することがあります。したがって、アロケータが空きリストの順序をどのように管理するかが問題になります。1つのアプローチは、アドレスベースの順序付けを使用することです。リストを空き領域のアドレス順に並べることにより、合体が容易になり、断片化が減少する傾向があります。

Next Fit

リストの始めに最初に適合する検索を常に開始するのではなく、次の適合アルゴリズムは、リスト内で最後に探していた場所への余分なポインタを保持します。このアイデアは、リスト全体に空きスペースの検索をより均一に広げることで、リストの先頭が崩れないようにすることです。このようなアプローチのパフォーマンスは、完全な検索が再び回避されるため、first fit と非常に似ています。

Examples

上記の戦略のいくつかの例を示します。サイズが 10, 30、および 20 の 3 つの要素を持つフリーリストを想像してください。(ヘッダーやその他の詳細は無視されます)



ベストフィット手法はリスト全体を検索し、要求に対応できる最小の空き領域であるため、20 が最適であることがわかります。



この例のように、最もよく合ったアプローチでよく起こりますが、小さな free チャンクが残ります。worst fit アプローチも同様ですが、この例では最大のチャンクが見つかります。



ファーストフィット戦略は、この例では、worst fit と同じことを行い、要求を満たすことができる最初のフリーブロックを見つけます。違いは検索コストです。リスト全体の見方がベストフィットとワーストフィットは違います。ファーストフィットは、適合するものが見つかるまでフリーチャンクを調べるだけで検索コストを削減します。

これらの例は、割り当て方針の表面を触っているだけです。より深い理解のためには、実際の仕事量とより複雑なアロケータの振る舞い(例えば、合体)によるより詳細な分析が必要です。

17.4 Other Approaches

上記の基本的なアプローチ以外にも、何らかの方法でメモリ割り当てを改善するための多くの提案されたテクニックとアルゴリズムがあります。

Segregated Lists

1つの興味深いアプローチは、分離されたリストの使用です。基本的な考え方は単純です。特定のアプリケーションが一般的なサイズの要求を1つ（またはいくつか）持っている場合は、そのサイズのオブジェクトを管理するためだけに別のリストを保持します。他のすべての要求は、一般的なメモリアロケータに転送されます。

そのようなアプローチの利点は明らかです。1つの特定のサイズの要求専用のメモリチャンクを持つことで、断片化の懸念が大幅に減ります。さらに、リストの複雑な検索が不要であるため、割り振り要求および空き要求は、適切なサイズのものであれば、すばやく処理できます。

ちょうどいいアイデアのように、このアプローチはシステムにも新しい複雑さをもたらします。たとえば、一般プールとは異なり、特定のサイズの特別なリクエストを処理するメモリプールにどれだけのメモリを割り当てる必要がありますか？1つの特定のアロケータである、Uber engineer の Jeff Bonwick(Solaris カーネル用に設計された)のスラブアロケータは、この問題をかなりうまく処理します [B94]。

具体的には、カーネルが起動すると、頻繁に要求される可能性が高いカーネルオブジェクト（ロック、ファイルシステムの inode など）に多数のオブジェクトキャッシュを割り当てます。このようにして、オブジェクトキャッシュは、与えるのサイズの空きリストをそれぞれ分離し、メモリ割り当ておよび空き要求を迅速に提供します。与えられたキャッシュが空き容量が少なくなったときには、より一般的なメモリアロケータから要求された量のスラブ（要求された合計量がページサイズの倍数と問題のオブジェクトです）を要求します。逆に、与えられたスラブ内のオブジェクトの参照カウントがすべてゼロになると、汎用アロケータは、VM システムがより多くのメモリを必要とするときによく行われる特別なアロケータからメモリを再要求することができます。

ASIDE: GREAT ENGINEERS ARE REALLY GREAT

Jeff Bonwick(ここで言及したスラブアロケータを書いただけでなく、すばらしいファイルシステムの先駆けでもあった ZFS)のようなエンジニアは、シリコンバレーの中心です。ほぼすべての偉大な製品や技術の背後には、才能、能力、献身が平均以上の人物（または少人数のグループ）があります。Mark Zuckerberg(Facebook の)は次のように述べています。「自分の役割において例外的な人は、少しだけ良い人ではありません。かなり良い人です。彼らは 100 倍も優れています。」これは、今日でも、世界の顔を永遠に変える会社 (Google、Apple、Facebook) を 1~2 人が立ち上げることができる理由です。懸命に働いて、あなたはそのような“100x”人になるかもしれません。それに失敗するかもしれません。しかし、そのような人と働くとあなたは 1 ヶ月以内に多くを学ぶよりも、1 日で多くを学ぶでしょう。

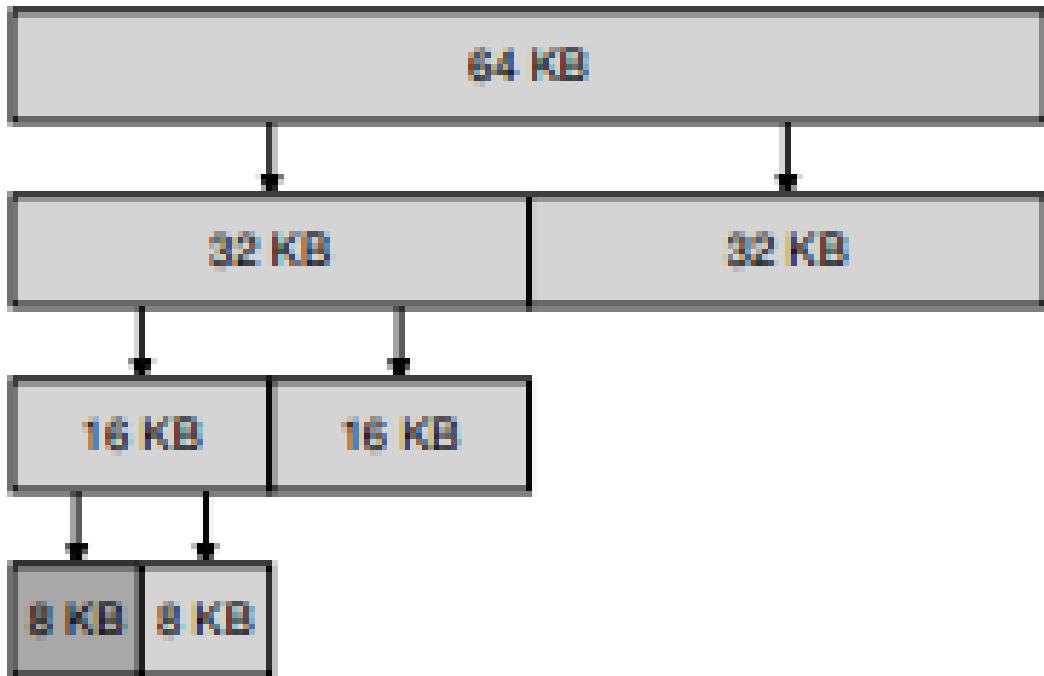
スラブアロケータはまた、リスト上の空きオブジェクトを事前初期化状態に保つことによって、ほとんどの分離リスト手法を超えています。Bonwick は、データ構造の初期化と破壊はコストがかかることを示しています [B94]。しかし、スラブアロケータは、解放されたオブジェクトを初期化された状態の特定のリストに保持することにより、オブジェクトごとの頻繁な初期化および破棄サイクルを回避し、オーバーヘッドを顕著に低減することができます。

Buddy Allocation

合体はアロケータにとって重要なことで、いくつかのアプローチは合体を簡単にすることを目的として設計されています。1つの良い例がバイナリバディアロケータ [K65] にあります。

このようなシステムでは、空きメモリは、概念的には、サイズ 2^N の1つの大きな空間と考えます。メモリの要求が行われると、空き領域の探索は、要求を収容するのに十分な大きさのブロックが見つかるまで、空き領域を2つに再帰的に分割します（さらに2つに分割すると、スペースが小さくなります）。この時点では、要求

されたブロックがユーザーに返されます。ここでは、7KB ブロックの検索で 64KB の空き領域を分けた例を示します。



この例では、左端の 8KB ブロックが割り当てられ（暗い影の灰色で示されるように）、ユーザーに返されます。このスキームでは、2 つのサイズのブロックを出力することだけが許可されているため、内部の断片化に悩まされる可能性があることに注意してください。

バディ割り当ての美しさは、そのブロックが解放されたときに何が起こるかで見られます。8KB のブロックを空きリストに戻すとき、アロケータは「バディ」8KB が空いているかどうかをチェックします。そうであれば、2 つのブロックを合併して 16KB のブロックにする。次に、アロケータは、16KB ブロックのバディがまだフリーであるかどうかをチェックします。そうであれば、それらの 2 つのブロックを結合する。この再帰的な結合プロセスは、ツリー全体を再構築し、空き領域全体を復元するか、バディが使用中であることが判明したときに停止します。

バディ割り当てがうまく機能する理由は、特定のブロックのバディを決定するのが簡単だからです。どうやって調べるでしょうか？ 上記の空き領域にあるブロックのアドレスについて考えてみましょう。よく注意深く考えてみると、各バディーペアのアドレスは 1 ビットだけ異なることがわかります。このビットはバディツリーのレベルによって決まります。したがって、バイナリバディ割り当て方式がどのように機能するかという基本的な考え方があります。詳細については、いつものように、Wilson 調査 [W+95] を参照してください。

Other Ideas

上記の多くのアプローチの 1 つの大きな問題は、スケーリングの欠如です。具体的には、リストの検索は非常に遅くなる可能性があります。したがって、高度なアロケータは、これらのコストに対処するために、より複雑なデータ構造を使用し、パフォーマンスを単純化します。例としては、バランスのとれたバイナリツリー、スプレイツリー、または部分的に順序付けられたツリー [W+95] があります。

現代のシステムでは複数のプロセッサがあり、マルチスレッドの仕事量を実行することが多いため（詳細については、同時実行に関する本のセクションで詳しく説明します）、マルチプロセッサベースのシステムでアロケータをうまく動作させるために多くの努力が費やされたことは驚くことではありません。2 つの素晴らしい

例が Berger et al[B+00] とエバンス [E06]。詳細を確認してください。

これらは、メモリアロケータについて人々が時間をかけて持っている何千ものアイデアのうちの 2 つに過ぎません。好奇心が強い場合は、あなた自身で読んでください。それに失敗した場合、glibc アロケータがどのように動作するかを読んで [S15]、実際の世界がどのようなものかを理解してください。

17.5 Summary

この章では、最も基本的な形のメモリアロケータについて説明しました。そのようなアロケータはどこにでも存在し、あなたが書いたすべての C プログラムにリンクされているだけでなく、独自のデータ構造のためにメモリを管理している基礎となる OS にも存在します。多くのシステムと同様に、そのようなシステムを構築するには多くのトレードオフがあります。アロケータに与えられる正確な仕事量について知るほど、その仕事量に対してよりうまく動作するよう調整することができます。広範囲の仕事量でうまく動作する、空間効率の良い高速でスケーラブルなアロケータを作成することは、現代のコンピュータシステムにおいて進行中の課題です。

参考文献

- [B+00] “Hoard: A Scalable Memory Allocator for Multithreaded Applications”
Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson
ASPLOS-IX, November 2000
Berger and company’s excellent allocator for multiprocessor systems. Beyond just being a fun paper, also used in practice!
- [B94] “The Slab Allocator: An Object-Caching Kernel Memory Allocator”
Jeff Bonwick
USENIX ’94
A cool paper about how to build an allocator for an operating system kernel, and a great example of how to specialize for particular common object sizes.
- [E06] “A Scalable Concurrent malloc(3) Implementation for FreeBSD”
Jason Evans
<http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf>
April 2006
A detailed look at how to build a real modern allocator for use in multiprocessors. The “jemalloc” allocator is in widespread use today, within FreeBSD, NetBSD, Mozilla Firefox, and within Facebook.
- [K65] “A Fast Storage Allocator”
Kenneth C. Knowlton
Communications of the ACM, Volume 8, Number 10, October 1965
The common reference for buddy allocation. Random strange fact: Knuth gives credit for the idea not to Knowlton but to Harry Markowitz, a Nobel-prize winning economist. Another strange fact: Knuth communicates all of his emails via a secretary; he doesn’t send email himself, rather he tells his secretary what email to send and then the secretary does the work of emailing. Last Knuth fact: he created TeX, the tool used to typeset this book. It is an amazing piece of software.
- [S15] “Understanding glibc malloc”
Sploitfun
February, 2015

<https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/>

A deep dive into how glibc malloc works. Amazingly detailed and a very cool read.

[W+95] “Dynamic Storage Allocation: A Survey and Critical Review”

Paul R. Wilson, Mark S. Johnstone, Michael Neely, David Boles

International Workshop on Memory Management

Kinross, Scotland, September 1995

An excellent and far-reaching survey of many facets of memory allocation. Far too much detail to go into in this tiny chapter!

18 Paging: Introduction

スペース管理問題を解決するには、オペレーティングシステムが 2 つのアプローチのうちの 1 つをとると言われることがあります。第 1 のアプローチは、仮想メモリのセグメンテーションで見たように、可変サイズの断片に分割することです。残念なことに、この解決策には固有の難点があります。特に、スペースを異なるサイズのチャンクに分割すると、スペース自体が断片化する可能性があり、時間の経過とともに割り当てがより困難になります。

したがって、第 2 のアプローチを検討する価値があるかもしれません。スペースを固定サイズの断片に切り詰めることです。仮想メモリでは、このアイディアをページングと呼びますが、これは初期の重要なシステムである Atlas [KE+62, L78] に戻ります。プロセスのアドレス空間をいくつかの可変長論理セグメント（例えば、コード、ヒープ、スタック）に分割するのではなく、固定サイズの単位に分割します。それぞれはページと呼ばれます。これに対応して、物理メモリは、ページ・フレームと呼ばれる固定サイズのスロットの配列と見なします。これらの各フレームには単一の仮想メモリページを含めることができます。

THE CRUX: HOW TO VIRTUALIZE MEMORY WITH PAGES

セグメント化の問題を避けるために、どのようにしてページを使ってメモリを仮想化できますか？

基本的なテクニックは何ですか？ 最小限のスペースと時間のオーバーヘッドで、これらのテクニックをどのようにうまく機能させるにはどうすればよいでしょう？

18.1 A Simple Example And Overview

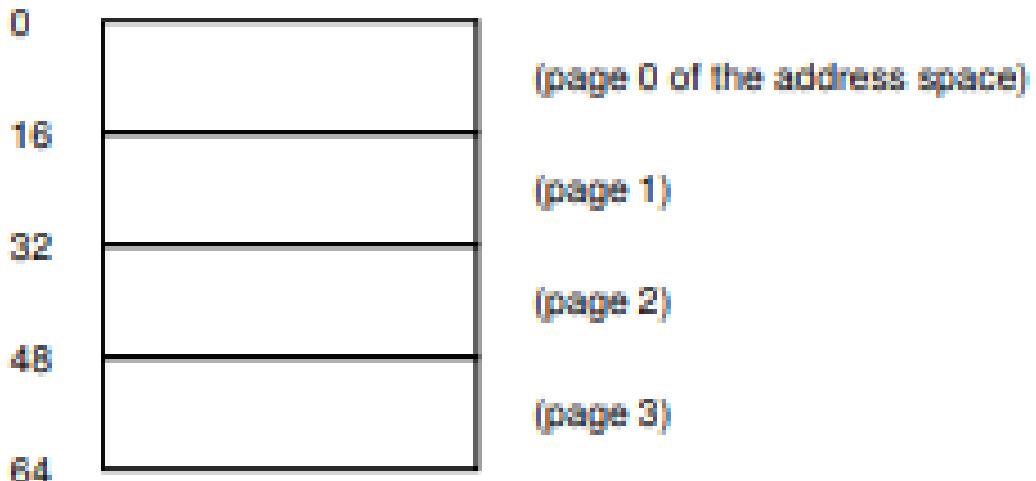


Figure 18.1: A Simple 64-byte Address Space

このアプローチをより明確にするために、簡単な例をあげて説明しましょう。図 18.1 は、4 つの 16 バイトページ（仮想ページ 0,1,2、および 3）を持つ 64 バイトのサイズの小さなアドレス空間の例を示しています。実際のアドレス空間は、もちろん 32 ビットであり、したがって 4GB のアドレス空間、さらには 64 ビットも同様です。この本では、小さな例を使ってダイジェストを簡単にすることがよくあります。

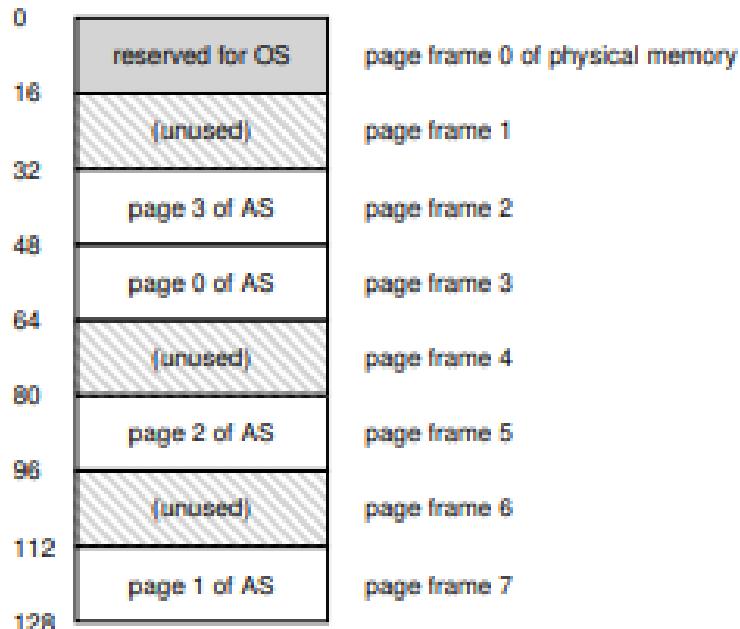


Figure 18.2: A 64-Byte Address Space In A 128-Byte Physical Memory

図 18.2 に示すように、物理メモリも固定サイズのスロット（この場合は 128 バイトの物理メモリを作成しますが、限りなく小さくします）が 8 ページのフレームで構成されています。図でわかるように、仮想アドレス空間のページは物理メモリ全体の異なる場所に配置されています。このダイアグラムは、物理メモリの一部を使用している OS も示しています。

ページングには、これまでのアプローチよりも多くの利点があります。おそらく、最も重要な改善は柔軟性です。完全に開発されたページング手法では、システムがアドレス空間をどのように使用するかにかかわらず、アドレス空間の抽象化を効果的にサポートできます。たとえば、ヒープとスタックの成長方向と使用方法については想定しません。

もう 1 つの利点は、ページングが提供する free space management の単純さです。たとえば、OS が 8 ページの物理メモリに小さな 64 バイトのアドレス空間を配置したい場合、OS は単に 4 つの空きページを見つけてます。おそらく OS はこのためのすべての空きページのフリーリストを保持しており、このリストから最初の 4 つの空きページだけを取得します。この例では、OS は、物理フレーム 3 のアドレス空間 (AS) の仮想ページ 0、物理フレーム 7 の AS の仮想ページ 1、フレーム 5 のページ 2、フレーム 2 のページ 3 を配置しています。ページフレーム 1、4、および 6 は現在フリーです。

アドレス空間の各仮想ページが物理メモリに配置される場所を記録するために、オペレーティングシステムは通常、ページテーブルと呼ばれるプロセス単位のデータ構造を保持します。ページテーブルの主な役割は、アドレス空間の各仮想ページのアドレス変換を格納することで、各ページが物理メモリのどこにあるのかを知ることができます。（仮想ページ 0 → 物理フレーム 3）、（VP1 → PF7）、（VP2 → PF5）、（VP3 → PF2）の 4 つのエントリがページテーブルにあります。

このページテーブルはプロセスごとのデータ構造であることを覚えておくことが重要です（ほとんどのページテーブル構造についてはプロセスごとの構造ですが、ここで触れる例外は逆ページテーブルです）。上記の例で別のプロセスを実行する場合、仮想ページは明らかに別の物理ページにマップされるため（別のページテーブルを管理する必要があります）

さて、アドレス変換の例を実行するのに十分なことは分かっています。その小さなアドレススペース (64 バイト) を持つプロセスがメモリアクセスを実行しているとします。

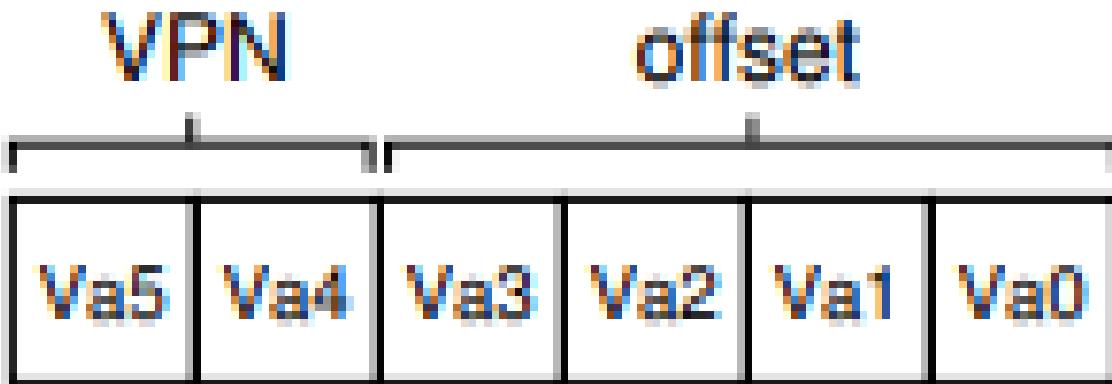
```
movl <virtual address>, %eax
```

具体的には、アドレス<virtual address>からレジスタ eax へのデータの明示的なロードに注意を払いましょう（したがって、先に起こっていなければならない命令フェッチを無視します）。

プロセスが生成したこの仮想アドレスを変換するには、仮想ページ番号 (VPN) とページ内のオフセットの 2 つのコンポーネントに分割する必要があります。この例では、プロセスの仮想アドレス空間が 64 バイトであるため、仮想アドレス ($2^6 = 64$) に合計 6 ビット必要です。したがって、仮想アドレスは次のように概念化できます。



この図において、Va5 は仮想アドレスの最上位ビットであり、Va0 は最下位ビットです。ページサイズ (16 バイト) を知っているので、仮想アドレスをさらに次のように分割することができます。

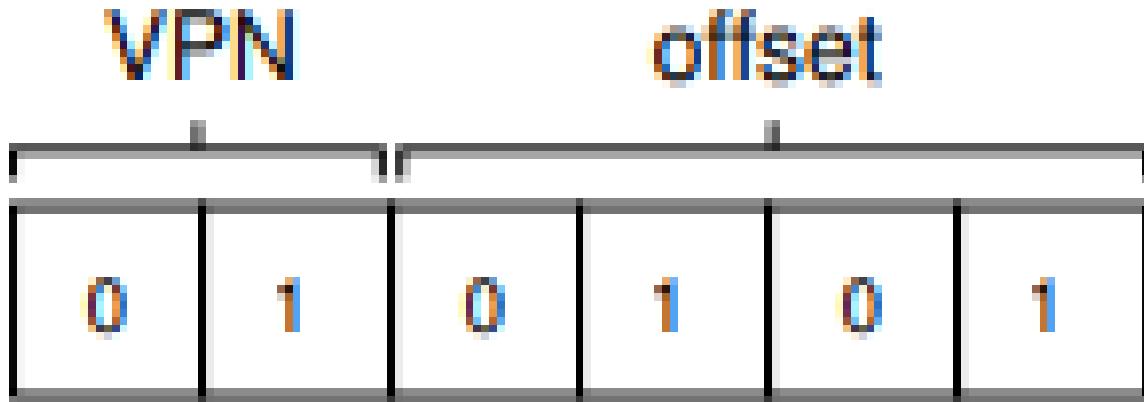


ページサイズは、64 バイトのアドレス空間では 16 バイトごとです。したがって、4 ページを選択できる必要があります。アドレスの上位 2 ビットはそれを選択するためにあります。したがって、2 ビットの仮想ページ番号 (VPN) があります。残りのビットは、ページのどのバイトにいるのかを示します。この場合は 4 ビットです。これをオフセットと呼びます。

プロセスが仮想アドレスを生成する場合、OS とハードウェアを結合して意味のある物理アドレスに変換する必要があります。たとえば、上記の負荷が仮想アドレス 21 であると仮定します。

```
movl 21, %eax
```

「21」をバイナリ形式にすると、「010101」が得られます。したがって、この仮想アドレスを調べて、仮想ページ番号 (VPN) とオフセットに分解する方法を確認できます。



したがって、仮想アドレス「21」は、仮想ページ「01」(または「1」)の5番目(「0101」番目)のバイト上にある。仮想ページ番号を使用して、ページテーブルをインデックス化し、仮想ページ1がどの物理フレーム内に存在するかを見つけることができます。上記のページテーブルでは、物理フレーム番号(PFN)(物理ページ番号またはPPNとも呼ばれます)は7(バイナリ111)です。したがって、VPNをPFNに置き換えてこの仮想アドレスを変換し、物理メモリにロードします。(図18.3)

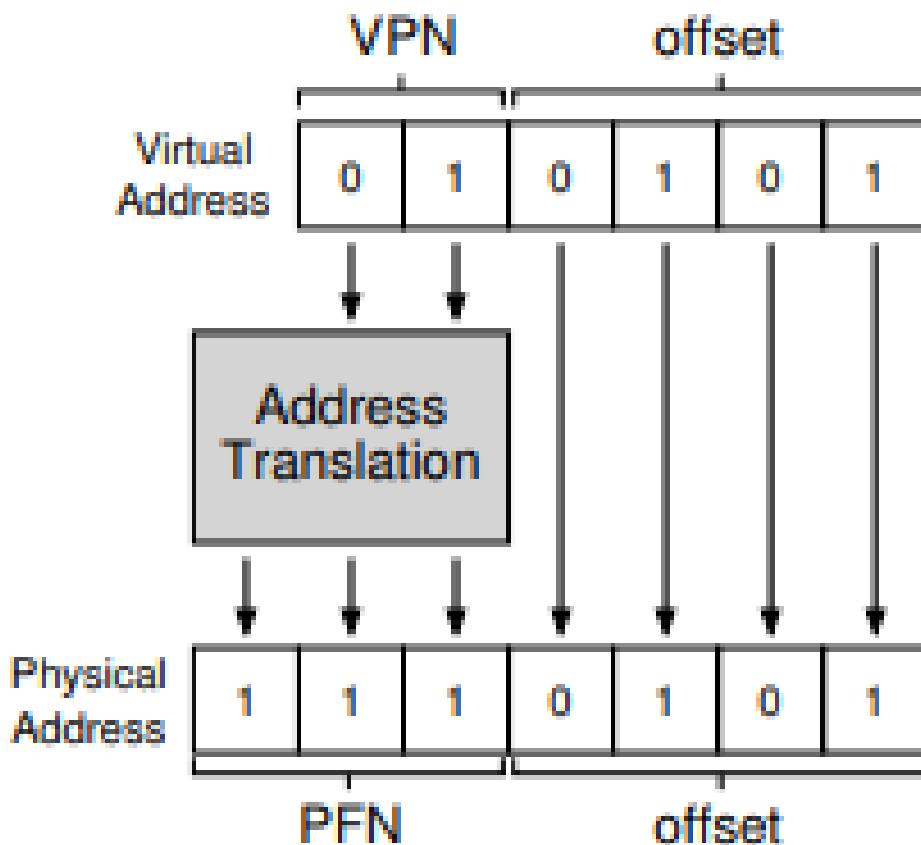


Figure 18.3: The Address Translation Process

オフセットは、ページ内のどのバイトを必要としているかを示すだけなので、オフセットは同じである(つまり、変換されない)ことに注意してください。私たちの最終的な物理アドレスは1110101(10進数では117)であり、正確に必要なデータをロードして取得します。(図18.2)

この基本的な概要を念頭に置いて、ページングに関するいくつかの基本的な質問をすることができます。たとえば、これらのページテーブルはどこに格納されていますか？ ページテーブルの典型的な内容とそのテーブルの大きさは何ですか？ ページングはシステムをかなり遅くしますか？ これらの疑問やその他の疑問に答える質問は、少なくとも部分的には以下の本文で回答しています。さあ、読んでいきましょう！

18.2 Where Are Page Tables Stored?

ページ・テーブルは、以前に議論した小さなセグメント・テーブルまたはベース/境界ペアよりもはるかに大きくなることがあります。たとえば、典型的な 32 ビットのアドレス空間で、4KB のページがあるとします。この仮想アドレスは、20 ビットの VPN と 12 ビットのオフセットに分割されます (1KB のページサイズには 10 ビットが必要で、4KB になるにはさらに 2 つ追加する必要があります)。

20 ビットの VPN は、各プロセスで OS が管理しなければならない 2^{20} の変換があることを示しています (これはおよそ 100 万です)。実際の変換に加えてページテーブルエントリ (PTE) あたり 4 バイトが必要であれば、各ページテーブルに必要なメモリは 4MB になります。それはかなり大きいです。100 個のプロセスが実行されているとしましょう。つまり、これらのアドレス変換だけで 400 MB のメモリが必要です。機械のギガバイトの記憶がある現代でさえ、変換のためだけに大きな塊を使用するのは少しおかしいですね。そのようなページテーブルが 64 ビットのアドレス空間にどれだけ大きなものになるかについても考えたくもないですね。

ページテーブルは非常に大きいので、現在実行中のプロセスのページテーブルを格納するための特別なオンチップハードウェアを MMU に保持しません。代わりに、各プロセスのページテーブルをどこかのメモリに格納します。ページテーブルが OS が管理する物理メモリに存在すると仮定してみましょう。後でわかるように、OS のメモリそのものを仮想化することができるので、ページテーブルを OS の仮想メモリに格納することができます (そしてディスクにスワップすることもできます)。しかし、これは今のところ混乱するでしょう。図 18.4 に、OS メモリ内のページテーブルの写真を示します。そこには小さな変換がありますか？

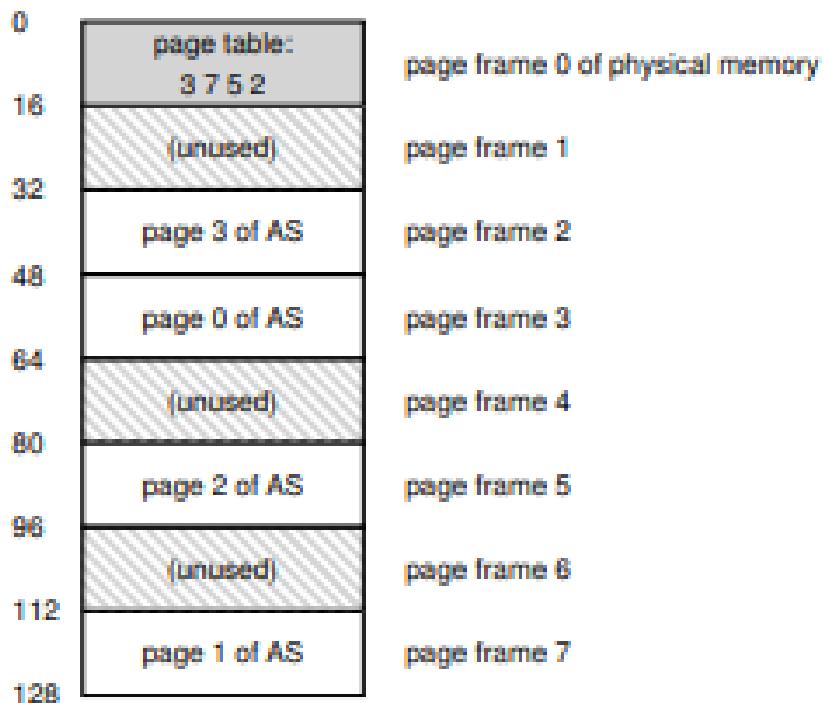


Figure 18.4: Example: Page Table in Kernel Physical Memory

18.3 What's Actually In The Page Table?

ページテーブルの構成について少し話をしましょう。ページテーブルは、仮想アドレス（または実際には仮想ページ番号）を物理アドレス（物理フレーム番号）にマッピングするために使用される単なるデータ構造です。したがって、どのようなデータ構造でも機能します。最も簡単な形式は線形ページテーブルと呼ばれ、単なる配列です。OSは配列を仮想ページ番号（VPN）で索引付けし、その索引でページ表エントリ（PTE）を検索して、必要な物理フレーム番号（PFN）を見つけます。今のところ、この単純な線形構造を仮定します。後の章では、より高度なデータ構造を使用して、ページングに関するいくつかの問題を解決していきます。

各PTEの内容については、いくつかのレベルで理解する価値のあるさまざまなビットがあります。有効なビットは、特定の変換が有効かどうかを示すために一般的です。たとえば、プログラムの実行が開始されると、コードとヒープ、そのアドレス空間の一方の端にスタック、その他があります。その間の未使用スペースはすべて無効とマークされ、プロセスがこのようなメモリにアクセスしようとすると、OSにトラップが生成され、プロセスが終了する可能性があります。従って、有効ビットは、未割り当てアドレス空間をサポートするために重要です。単にアドレス空間内の未使用ページをすべて無効にするだけで、それらのページに物理フレームを割り当てる必要がなくなり、メモリを大幅に節約することができます。

また、ページの読み込み、書き込み、実行が可能かどうかを示す保護ビットがあります。ここでも、これらのビットによって許可されていない方法でページにアクセスすると、OSにトラップが生成されます。

重要ないくつかのビットがありますが、今はあまり話しません。presentビットは、このページが物理メモリ内にあるかディスク上にあるか（すなわち、スワップアウトされたか）を示します。物理メモリよりも大きいアドレス空間をサポートするために、アドレス空間の一部をディスクにスワップする方法を調べるときに必要になります。スワップは、ほとんど使用されないページをディスクに移動することによって、OSが物理メモリを解放することができます。dirtyビットも一般的であり、ページがメモリに格納されてから変更されたかどうかを示します。

referenceビット（a.k.a アクセスピット）は、ページがアクセスされたかどうかを追跡するために使用されることがあり、どのページが頻繁にアクセスされて必要であり、メモリに保持されるべきかを決定するのに有用です。このような知識は、ページの置換時に重要です。トピックについては、以降の章で詳しく説明します。

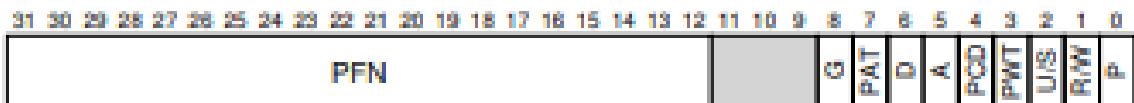


Figure 18.5: An x86 Page Table Entry (PTE)

図18.5に、x86アーキテクチャ[I09]のページテーブルエントリの例を示します。presentビット(P)、このページに書き込みが許可されているかどうかを判定するread/writeビット(R/W)、ユーザモードプロセスがページにアクセスできるかどうかを決定するuser/supervisorビット(U/S)、これらのページのハードウェアキャッシングがどのように機能するかを決定する数ビット(PWT、PCD、PAT、およびG)、accessedビット(A)、dirtyビット(D)、最後に、ページ・フレーム番号(PFN)を含みます。

x86ページングのサポートの詳細については、インテルアーキテクチャマニュアル[I09]を参照してください。しかし、あらかじめ注意してください。これらのマニュアルを読むことは、かなり有益ですが（もちろん、OSでそのようなページテーブルを使用するコードを書く人にとっては必要です）、最初は挑戦的かもしれません。ちょっとした忍耐と多くの好奇心が必要です。

18.4 Paging: Also Too Slow

メモリ内のページテーブルでは、大きすぎる可能性があることを既に認識しています。それが判明したので、落ち着いてページテーブルを考えることができます。例題として簡単なものを考えていきましょう：

```
movl 21, %eax
```

繰り返しますが、アドレス 21 への明示的な参照を調べて、命令フェッチについて心配することはありません。この例では、ハードウェアが変換を実行すると仮定します。必要なデータをフェッチするために、システムは最初に仮想アドレス (21) を正しい物理アドレス (117) に変換しなければいけません。従って、アドレス 117 からデータをフェッチする前に、システムはまず、プロセスのページテーブルから適切なページテーブルエントリをフェッチし、変換を実行し、次に物理メモリからデータをロードしなければいけません。

そのためには、ハードウェアはページテーブルが現在実行中のプロセスのどこにあるのかを知る必要があります。ここでは、単一のページテーブルベースレジスタに、ページテーブルの開始位置の物理アドレスが含まれていると仮定します。目的のページテーブルエントリー (PTE) の場所を見つけるために、ハードウェアは次の機能を実行します。

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
PTEAddr = PageTableBaseRegister + (VPN * sizeof(PTE))
```

この例では、VPN Mask は 0x30(16 進数、または 110000) に設定され、フル仮想アドレスから VPN ビットが選択されます。SHIFT は 4(オフセットのビット数) に設定されているため、正しい整数仮想ページ番号を形成するために VPN ビットを下に移動します。たとえば、仮想アドレス 21(010101) で、マスクするとこの値は 010000 になります。それをシフトさせると 01 になります。それが仮想ページ 1 になります。次にこの値を、ページテーブルベースレジスタが指示する PTE の配列へのインデックスとして使用します。

この物理アドレスがわかれば、ハードウェアはメモリから PTE を取り出し、PFN を抽出し、仮想アドレスからのオフセットと連結して必要な物理アドレスを形成することができます。具体的には、SHIFT によって左シフトされた PFN を考えることができます。次に、オフセットと OR を取って最終アドレスを次のようにします。

```
offset = VirtualAddress & OFFSET_MASK
PhysAddr = (PFN << SHIFT) | offset

// Extract the VPN from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT

// Form the address of the page-table entry (PTE)
PTEAddr = PTBR + (VPN * sizeof(PTE))

// Fetch the PTE
PTE = AccessMemory(PTEAddr)

// Check if process can access the page
if (PTE.Valid == False)
    RaiseException(SEGMENTATIONFAULT)
```

```

else if (CanAccess(PTE.ProtectBits) == False)
RaiseException(PROTECTION_FAULT)
else
// Access is OK: form physical address and fetch it
offset = VirtualAddress & OFFSET_MASK
PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
Register = AccessMemory(PhysAddr)

```

// Figure 18.6: Accessing Memory With Paging

最後に、ハードウェアはメモリから必要なデータを取り出し、それをレジスタ eax に入れることができます。プログラムはメモリから値をロードするのに成功しました！

要約すると、各メモリ参照で何が起きるかについての初期プロトコルを説明します。図 18.6 に基本的なアプローチを示します。すべてのメモリ参照（命令フェッチまたは明示的なロードまたはストア）にかかるわらず、ページングでは、最初にページテーブルから変換をフェッチするために 1 つの余分なメモリ参照を実行する必要があります。それはたくさんの仕事です！ 余分なメモリ参照はコストがかかり、この場合、プロセスが 2 倍以上遅くなる可能性があります。

そして今、解決しなければならない 2 つの本当の問題があります。ハードウェアとソフトウェアの両方を慎重に設計しなければ、ページテーブルはシステムの動作が遅くなり過ぎるだけでなく、あまりにも多くのメモリを占有します。メモリ仮想化ニーズのための一見すばらしい解決策ですが、これらの 2 つの重大な問題をまず解決する必要があります。

18.5 A Memory Trace

終了する前に、単純なメモリアクセスの例をトレースして、ページングを使用したときに発生するすべてのメモリアクセスを実証します。興味のあるコードスニペット（C 言語の array.c というファイル）は次のようになります。

```

int array[1000];
...
for (i = 0; i < 1000; i++)
array[i] = 0;

```

ASIDE: DATA STRUCTURE — THE PAGE TABLE

現代の OS のメモリ管理サブシステムにおける最も重要なデータ構造の 1 つは、ページテーブルです。一般に、ページテーブルは仮想アドレスから物理アドレスへの変換を格納しているので、アドレス空間の各ページが実際に物理メモリ内に存在する場所をシステムに知らせることができます。各アドレス空間はそのような変換を必要とするため、一般に、システムにはプロセスごとに 1 つのページテーブルがあります。ページテーブルの正確な構造は、ハードウェア（古いシステム）によって決定されるか、または OS（現代システム）によってより柔軟に管理されます。

We compile array.c and run it with the following commands:

```

prompt> gcc -o array array.c -Wall -O
prompt> ./array

```

もちろん、このコードスニペット(単純に配列を初期化する)にアクセスするメモリが本当に理解できるようになるには、さらにいくつかのことを知っておく必要があります。まず、結果のバイナリを(Linuxではobjdump、Macではotoolを使用して)逆アセンブルして、どのアセンブリ命令を使ってループ内の配列を初期化するかを調べる必要があります。結果のアセンブリコードは次のとおりです。

```
1024 movl $0x0,(%edi,%eax,4)
1028 incl %eax
1032 cmpl $0x03e8,%eax
1036 jne 0x1024
```

コードは、あなたが少し x86 を知っていれば、実際にはかなり理解しやすいです。最初の命令は、値のゼロ(\$0x0として表示)を配列の場所の仮想メモリアドレスに移動します。このアドレスは、%ediに%eaxを加算して%eaxに4を掛け合わせて計算します。したがって、%ediは配列のベースアドレスを保持しますが、%eaxは配列 index(i)を保持します。配列は4バイトの整数の配列であるため、4を掛けます。

2番目の命令は、%eaxに保持されている配列インデックスをインクリメントし、3番目の命令は、そのレジスタの内容を16進値0x03e8または1000に比較します。比較によって、2つの値が等しくないのであれば、4番目の命令はループの先頭にジャンプします。

この命令シーケンスがどのようなメモリアクセスを行うか(仮想レベルと物理レベルの両方で)を理解するためには、仮想メモリ内のどこにコードスニペットと配列があるのかとページテーブルの内容と場所を知る必要があります。

この例では、サイズが64KB(非現実的に小さい)の仮想アドレス空間を想定しています。また、1KBのページサイズを想定しています。

ここで知る必要があるのは、ページテーブルの内容とその物理メモリ内の場所です。線形(配列ベース)ページテーブルを持ち、それが物理アドレス1KB(1024)にあると仮定しましょう。その内容に関しては、この例のためにマップしたことを心配する必要のある仮想ページがほんの少しあります。まず、コードが存在する仮想ページがあります。ページサイズは1KBなので、仮想アドレス1024は仮想アドレス空間の2番目のページにあります(VPN = 1、最初のページはVPN = 1)。この仮想ページが物理フレーム4(VPN 1 → PFN 4)にマップされているとします。

次に、配列自体があります。そのサイズは4000バイト(1000個の整数)で、仮想アドレス40000～44000(最後のバイトは含まない)に存在すると仮定します。この小数点範囲の仮想ページは、VPN = 39…VPN = 42です。したがって、これらのページのマッピングが必要です。例(VPN 39 → PFN 7)、(VPN 40 → PFN 8)、(VPN 41 → PFN 9)、(VPN 42 → PFN 10)の仮想-物理マッピングを想定してみましょう。

これで、プログラムのメモリ参照をトレースする準備ができました。実行されると、各命令フェッチは2つのメモリ参照を生成します。1つはページテーブルに命令が存在する物理フレームを見つけるためのもの、もう1つは処理のためにCPUにフェッチする命令です。さらに、1つの明示的なメモリ参照がmov命令の形式で存在します。これは最初に別のページテーブルアクセスを追加し(配列の仮想アドレスを正しい物理アドレスに変換します)、次に配列アクセス自体を追加します。

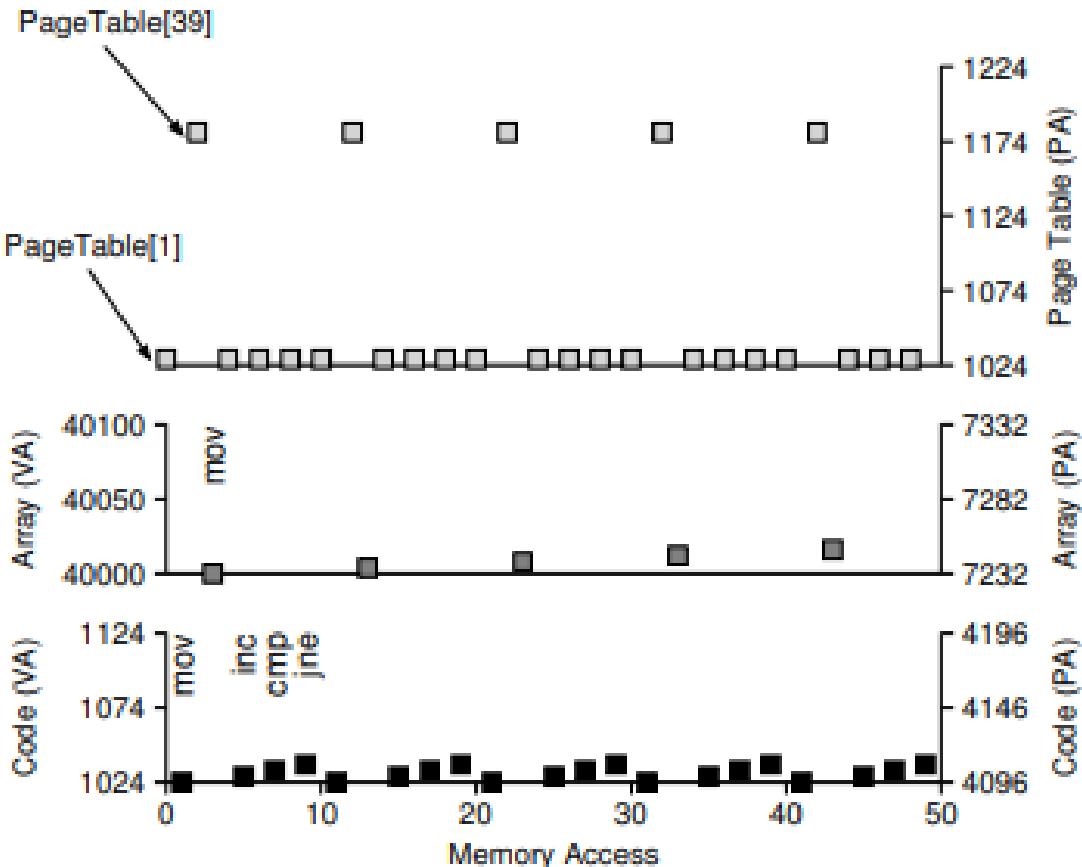


Figure 18.7: A Virtual (And Physical) Memory Trace

最初の 5 回のループ反復のプロセス全体が、図 18.7 に示されています。一番下のグラフは y 軸の命令メモリ参照を黒で示しています (仮想アドレスは左に、実際の物理アドレスは右にあります)。真ん中のグラフは、暗い灰色の配列アクセスを示しています (再び左上の仮想と右の物理を示します) 最後に、一番上のグラフは、ページテーブルのメモリアクセスを明るい灰色で示しています (この例のページテーブルは物理メモリに存在します) トレース全体の x 軸は、ループの最初の 5 回の反復でのメモリアクセスを示します。ループ当たり 10 回のメモリアクセスがあります。これには、4 回の命令フェッチ、1 回の明示的なメモリ更新、これらの 4 回のフェッチと 1 回の明示的な更新を変換する 5 回のページテーブルアクセスが含まれます。

この視覚化に現れるパターンを理解できるかどうかを確認してください。特に、ループが最初の 5 回の反復を超えて実行されると、何が変わるでしょうか？どの新しいメモリ位置にアクセスするのでしょうか？

これはちょうど例のはんの一例です (C コードのはんの数行だけです)。しかし、実際のアプリケーションの実際のメモリ動作を理解することの複雑さをすでに感じ取っているかもしれません。これ以上はメカニズムを複雑にするだけなので、ここまで終わりです。ごめんなさい！

18.6 Summary

私たちは、メモリを仮想化するという課題に対する解決策として、ページングの概念を導入しました。ページングには、以前のアプローチ (セグメンテーションなど) に比べて多くの利点があります。第 1 に、ページング (設計による) はメモリを固定サイズの単位に分割するため、外部の断片化につながることはありません。第 2 に、これは非常に柔軟であり、仮想アドレス空間の未使用部分を使用可能、使用不可の両方に対応してくれます。

ただし、ページングのサポートを気にせずに実装すると、マシンが遅くなり（ページテーブルにアクセスするために余分なメモリアクセスが増えます）、メモリが無駄になります（有用なアプリケーションデータではなくページテーブルでメモリがいっぱいになります）。私たちは、ページングシステムを思いつくのはちょっと難しいと思うでしょう。次の2つの章では、幸いにも、マシンを速くする方法、メモリの無駄をなくす方法を教えてくれるでしょう。

参考文献

[KE+62] “One-level Storage System”

T. Kilburn, and D.B.G. Edwards and M.J. Lanigan and F.H. Sumner

IRE Trans. EC-11, 2 (1962), pp. 223-235

(Reprinted in Bell and Newell, “Computer Structures: Readings and Examples” McGraw-Hill, New York, 1971). The Atlas pioneered the idea of dividing memory into fixed-sized pages and in many senses was an early form of the memory-management ideas we see in modern computer systems.

[I09] “Intel 64 and IA-32 Architectures Software Developer’s Manuals”

Intel, 2009

Available: <http://www.intel.com/products/processor/manuals>

In particular, pay attention to “Volume 3A: System Programming Guide Part 1” and “Volume 3B: System Programming Guide Part 2”

[L78] “The Manchester Mark I and atlas: a historical perspective”

S. H. Lavington

Communications of the ACM archive

Volume 21, Issue 1 (January 1978), pp. 4-12

Special issue on computer architecture

This paper is a great retrospective of some of the history of the development of some important computer systems. As we sometimes forget in the US, many of these new ideas came from overseas.

19 Paging: Faster Translations (TLBs)

仮想メモリをサポートするためのコアメカニズムとしてページングを使用すると、パフォーマンスのオーバーヘッドが発生する可能性があります。アドレス空間を小さい固定サイズの単位(すなわち、ページ)にすることにより、ページングは大量のマッピング情報を必要とします。そのマッピング情報は一般的に物理メモリに格納されるため、ページングは論理的にはプログラムによって生成された各仮想アドレスに対して余分なメモリルックアップを必要とします。命令のフェッチや明示的なロードやストアの前に、変換情報のためのメモリへの移動は非常に遅いです。

THE CRUX: HOW TO SPEED UP ADDRESS TRANSLATION

どのようにしてアドレス変換を高速化し、ページングに必要な余分なメモリ参照を避けることができますか？どんなハードウェアサポートが必要ですか？どのようなOSの関与が必要ですか？

物事を速くしたいとき、OSは通常何らかの助けを必要とします。また、ハードウェアから助けを得ていることがよくあります。アドレス変換を高速化するために、(歴史的な理由で[CP78]と呼ばれる)変換ルックアサイドバッファ、すなわちTLB[CG68、C95]を追加します。TLBは、チップのメモリ管理ユニット(MMU)の一部であり、一般的な仮想から物理へのアドレス変換のハードウェアキャッシュです。したがって、より良い名前はアドレス変換キャッシュになります。各仮想メモリ参照時に、ハードウェアは最初にTLBをチェックして、その中に所望の変換が保持されているかどうかを調べます。そうであれば、ページテーブル(すべての変換が含まれています)を参照することなく、変換が(迅速に)実行されます。パフォーマンスに大きな影響を与えるため、実際の意味でのTLBは仮想メモリを可能にします[C95]。

19.1 TLB Basic Algorithm

```

1   VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2   (Success, TlbEntry) = TLB_Lookup(VPN)
3   if (Success == True) // TLB Hit
4       if (CanAccess(TlbEntry.ProtectBits) == True)
5           Offset = VirtualAddress & OFFSET_MASK
6           PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7           Register = AccessMemory(PhysAddr)
8       else
9           RaiseException(PROTECTION_FAULT)
10  else // TLB Miss
11      PTEAddr = PTBR + (VPN * sizeof(PTE))
12      PTE = AccessMemory(PTEAddr)
13      if (PTE.Valid == False)
14          RaiseException(SEGMENTATION_FAULT)
15      else if (CanAccess(PTE.ProtectBits) == False)
16          RaiseException(PROTECTION_FAULT)
17      else
18          TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
19          RetryInstruction()

```

Figure 19.1: TLB Control Flow Algorithm

図19.1は、単純な線形ページテーブル(すなわち、ページテーブルが配列)とハードウェア管理TLBです。

ハードウェアがページ変換の多くの処理すると仮定して、ハードウェアが仮想アドレス変換を処理する方法の概略を示しています。テーブルのアクセスについては、後で詳しく説明します。

ハードウェアが従うアルゴリズムは次のとおりです。まず、仮想アドレス(図19.1の1行目)から仮想ページ番号(VPN)を抽出し、TLBがこのVPN(2行目)の変換を保持しているかどうかを確認します。もしそうなら、私たちはTLBヒットを持っています。これは、TLBが変換を保持していることを意味します。ここで、関連するTLBエントリからページフレーム番号(PFN)を抽出し、元の仮想アドレスからのオフセットに連結し、望んだ物理アドレス(PA)を形成し、メモリを保護することができます(5-7行目)。チェックは失敗しません(4行目)。

CPUがTLB(TLBミス)で変換を見つけられない場合は、さらに処理する必要があります。この例では、ハードウェアがページテーブルにアクセスして変換を検索し(11~12行目)、プロセスによって生成された仮想メモリ参照が有効でアクセス可能であると仮定すると(13行目、15行目)TLBの更新(18行目)。これらの一連のアクションは、主にページテーブルにアクセスするために必要な余分なメモリ参照が原因でコストがかかります(12行目)。最後に、TLBが更新されると、ハードウェアは命令を再試行します。今回はTLBに変換があり、メモリ参照は素早く処理されます。

TLBは、すべてのキャッシュと同様に、一般的なケースでは、キャッシュ内に変換がある(すなわち、ヒットしている)という前提で構築されています。TLBが処理コアの近くにあり、非常に高速になるように設計されているため、オーバーヘッドはほとんどありません。ミスが発生すると、ページングのコストが高くなります。変換を見つけるためにページテーブルにアクセスしなければならず、余分なメモリ参照(またはより複雑なページテーブルを含む)が結果として生じます。これが頻繁に発生する場合、プログラムは著しく遅く実行される可能性があります。ほとんどのCPU命令と比較してメモリアクセスは非常にコストがかかり、TLBミスはより多くのメモリアクセスにつながります。したがって、可能な限りTLBミスを避けることが重要です。

19.2 Example: Accessing An Array

TLBの動作を明確にするために、単純な仮想アドレストレースを調べて、TLBがどのように性能を向上させるかを見てみましょう。この例では、仮想アドレス100から始まる、メモリ内に10個の4バイト整数の配列があるとしましょう。さらに、16バイトページを持つ小さな8ビット仮想アドレス空間があるとします。したがって、仮想アドレスは4ビットのVPN(16の仮想ページがあります)と4ビットのオフセット(各ページに16バイトあります)に分割されます。

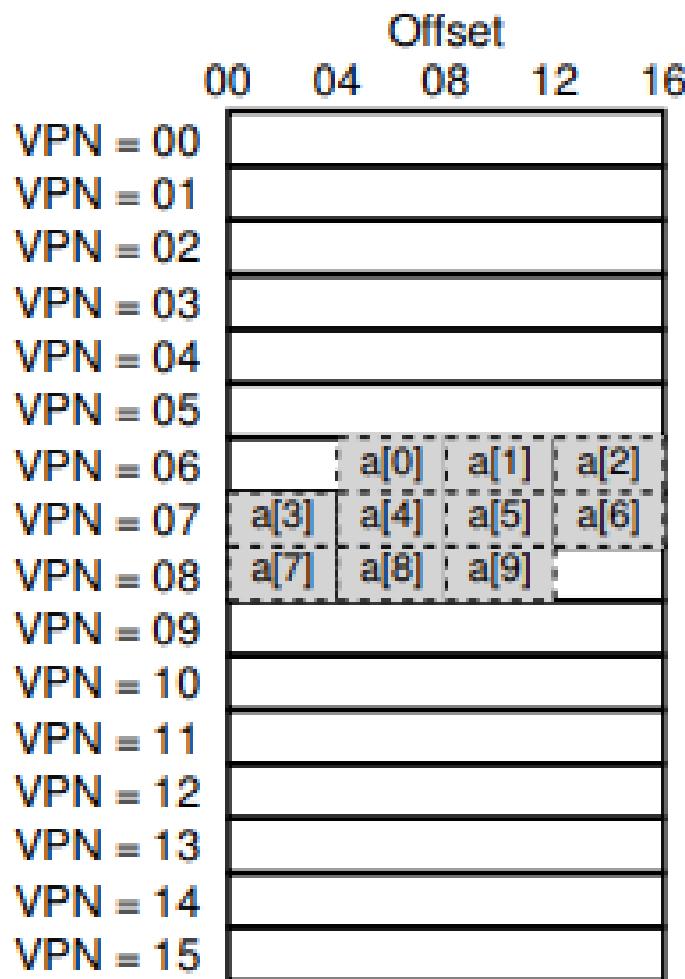


Figure 19.2: Example: An Array In A Tiny Address Space

図 19.2 に、16 バイトページにレイアウトされた配列を示します。ご覧のとおり、配列の最初のエントリ ($a[0]$) は (VPN = 06、offset = 04) から始まります。そのページには 3 つの 4 バイト整数しか収まりません。配列は次のページ (VPN = 07) に進み、次の 4 つのエントリ ($a[3] \dots a[6]$) が見つかります。最後に、10 エントリ配列 ($a[7] \dots a[9]$) の最後の 3 つのエントリは、アドレス空間の次のページ (VPN = 08) に配置されます。次に、各配列要素にアクセスする単純なループを考えてみましょう。C 言語では次のようにになります。

```
int sum = 0;
for (i = 0; i < 10; i++) {
    sum += a[i];
}
```

簡単にするために、私たちは、ループに対して生成される唯一のメモリアクセスが配列に対するものであると推測します (変数 i と sum 、および命令自体は無視します)。最初の配列要素 ($a[0]$) がアクセスされると、CPU は仮想アドレス 100 へのロードを認識します。ハードウェアはこの (VPN = 06) から VPN を抽出し、それを使用して TLB の有効な変換をチェックします。プログラムが配列に初めてアクセスすると仮定すると、

結果は TLB ミスになります。

次のアクセスは [1] へのアクセスです。TLB ヒット！ 配列の 2 番目の要素は最初の要素の隣にいるため、同じページに存在します。配列の最初の要素にアクセスするときにすでにこのページにアクセスしているので、変換はすでに TLB にロードされています。[2] へのアクセスは、[0] と [1] と同じページにも存在するため、同様の成功（別のヒット）に遭遇します。

残念ながら、プログラムが [3] にアクセスすると、別の TLB ミスが発生します。しかし、TLB 内のすべてのエントリがメモリ内の同じページに存在するため、次のエントリ（a [4] … a [6]）が再びヒットします。

最後に [7] にアクセスすると、最後の TLB ミスが 1 つ発生します。ハードウェアは、再び、物理メモリ内のこの仮想ページの位置を把握するためにページテーブルを参照し、それに応じて TLB を更新します。TLB 内のすべてのエントリがメモリ内の同じページに存在するため、最後の 2 回のアクセス（a [8] と a [9]）には、再びヒットします。

ヒット、ヒット、ミス、ヒット、ヒット、ヒット、ミス、ヒット、ヒット、配列への 10 回のアクセス中に TLB アクティビティを要約しましょう。したがって、ヒット数をアクセス総数で割った TLB ヒット率は 70 % です。これはあまり高くありませんが（確かに、ヒット率は 100 % に近づいています）、これは非ゼロです。これは驚くかもしれません。プログラムが配列にアクセスするのはこれが初めてですが、TLB は空間的局所性のためにパフォーマンスを向上させます。配列の要素は、ページに密接にパックされている（すなわち、それらは互いに空間的に近い）ので、ページ上の要素への最初のアクセスのみが TLB ミスをします。

この例では、ページサイズが果たす役割にも注意してください。ページ・サイズが単に 2 倍の大きさ（16 バイトではなく 32 バイト）であれば、配列アクセスにはより少ないミスしか生じません。一般的なページサイズは 4KB に似ていますので、これらのタイプの密な配列ベースのアクセスは優れた TLB パフォーマンスを実現し、1 ページのアクセス当たり 1 回のミスに遭遇します。

TLB のパフォーマンスに関する最後の 1 つは、このループが完了してすぐにプログラムが再び配列にアクセスすると、必要な変換をキャッシュするのに十分な TLB があると仮定すると、ヒット、ヒット、ヒット、ヒット、ヒット、ヒット、ヒットになります。この場合、時間的局所性、すなわち時間的にメモリ項目の迅速な再参照のために TLB ヒット率が高くなります。キャッシュと同様、TLB はプログラムのプロパティである成功のために空間的（キャッシュの大きさ）および時間的局所性（再びアクセスされる可能性）に依存しています。関心のあるプログラムがそのような局所性（および多くのプログラム）を示す場合、TLB のヒット率は高くなる可能性が高いです。

TIP: USE CACHING WHEN POSSIBLE

キャッシングは、コンピュータシステムで最も基本的なパフォーマンス手法の 1 つです。それは、「共通の高速化」[HP06] を何度も繰り返すことです。ハードウェアキャッシングの背後にあるアイデアは、命令とデータ参照における局所性を利用することです。通常、時間的局所性と空間的局所性の 2 つのタイプがあります。時間的局所性を考慮すると、最近アクセスされた命令またはデータ項目は、将来すぐに再アクセスされる可能性が高いという考えがあります。ループ変数や命令をループ内で考えてみましょう。それらは時間の経過と共に繰り返しアクセスされます。空間的局所性では、プログラムがアドレス x のメモリにアクセスすると、 x の近くのメモリにすぐにアクセスする可能性があるという考えがあります。1 つの要素にアクセスしてから次の要素にアクセスする何らかの配列をストリーミングすることを想像してください。もちろん、これらの特性はプログラムの正確な性質に依存するため、厳しい法則ではなく、より大雑把なルールです。

ハードウェアキャッシングは、命令、データ、アドレス変換（TLB のように）を問わず、メモリのコピーを小型で高速なオンチップメモリに保存することでローカリティを利用します。要求を満たすために（遅い）メモリに移動する代わりに、プロセッサは、まず、近くのコピーがキャッシング内に存在するかどうかをチェックすることができます。そうであれば、プロセッサは迅速に（すなわち、

数 CPU サイクルで) アクセスし、メモリにアクセスするのに費やす時間(多くのナノ秒)を費やすことを回避することができます。

あなたは疑問に思うかもしれません。(TLB のような) キャッシュがすばらしいのであれば、もつと大きなキャッシュを作り、その中にすべてのデータを保存してみてください。残念ながら、これは物理学のようなより基本的な法則に踏み込んでいます。高速キャッシュが必要な場合は、光速や他の物理的な制約などの問題が関連するため、キャッシュを小さくする必要があります。定義どおりの大規模なキャッシュはすべて遅いため、目的が破綻してしまいます。したがって、私たちは小さくて高速なキャッシュが必要なのです。残っている問題は、パフォーマンスを向上させるためにそれらを最適に使用する方法です。

19.3 Who Handles The TLB Miss?

私たちは答えなければならない 1 つの質問があります。誰が TLB ミスを処理するのですか? ハードウェアまたはソフトウェア(OS)の 2 つの答えが可能です。昔、ハードウェアには複雑な命令セット(複雑な命令セットのコンピュータでは CISC と呼ばれることがあります)がありました。ハードウェアを構築した人々は、OS の人たちをあまり信頼しませんでした。したがって、ハードウェアは TLB ミスを完全に処理します。これを行うには、ハードウェアはページテーブルがメモリ内のどこにあるのかを(図 19.1 の 11 行目で使用されているページテーブルベースレジスタを使用して)正確に知る必要があります。欠落していると、ハードウェアはページテーブルを「歩いて」、正しいページテーブルエントリを見つけて、必要な変換を抽出し、TLB を更新し、命令を再試行します。ハードウェア管理 TLB を持つ「古い」アーキテクチャの例は、固定マルチレベルページテーブルを使用する Intel x86 アーキテクチャです(詳細は次の章を参照してください)。現在のページテーブルは CR3 レジスタ[I09]によって指示されます。

```

VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True) // TLB Hit
if (CanAccess(TlbEntry.ProtectBits) == True)
    Offset = VirtualAddress & OFFSET_MASK
    PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
    Register = AccessMemory(PhysAddr)
else
    RaiseException(PROTECTION_FAULT)
else // TLB Miss
    RaiseException(TLB_MISS)
// Figure 19.3: TLB Control Flow Algorithm (OS Handled)

```

より近代的なアーキテクチャ(例えば、MIPS R10k [H93] または Sun の SPARC v9 [WG00]、RISC または縮小命令セットコンピュータ)は、ソフトウェア管理 TLB として知られているものを持っています。TLB ミスでは、ハードウェアは単に現在の命令ストリームを一時停止し、特権レベルをカーネルモードに持ち上げ、トラップハンドラにジャンプする例外(図 19.3 の 11 行目)を生成するだけです。ご想像のとおり、このトラップハンドラは、TLB ミスを処理する明白な目的で書かれた OS 内のコードです。実行されると、コードはページテーブルの変換を検索し、特殊な“特権”命令を使用して TLB を更新し、トラップから戻ります。この時点で、ハードウェアは命令をリトライします(結果として TLB ヒットとなります)。

重要な詳細をいくつか議論しましょう。まず、return from trap 命令は、システムコールを処理する前に見

たトラップからの復帰とは少し異なる必要があります。後者のソフトウェアの場合、プロシージャコールからのリターンがプロシージャへのコールの直後の命令に戻るのと同様に、return from trap は OS へのトラップ後の命令で実行を再開する必要があります。

前者のハードウェアの場合、TLB ミスハンドリングトラップから復帰するとき、ハードウェアはトラップを引き起こした命令で実行を再開しなければいけません。この再試行により、命令が再び実行され、今回は TLB ヒットとなります。したがって、トラップや例外がどのように発生したかによって、ハードウェアは OS にトラップするときに別の PC(プログラムカウンタ) を保存しなければならず、その時間が到来したときに適切に再開する必要があります。

第 2 に、TLB ミス・ハンドリング・コードを実行するとき、OS は無限の TLB ミス・チェインが発生しないように注意する必要があります。その問題に対しては多くの解決策が存在します。たとえば、TLB ミスハンドラを物理メモリに保存しておくことができます(アドレス変換が行われていない場合)。または、ハンドラコード自体を永続的に有効な変換の TLB にエントリとして予約します。これらの有線変換は常に TLB でヒットします。

ソフトウェア管理アプローチの主な利点は柔軟性です。OS は、ハードウェアの変更を必要とせずに、ページテーブルを実装したい任意のデータ構造を使用できます。別の利点は単純さです。TLB の制御フロー(図 19.3 の 11 行目と、図 19.1 の 11 行目とは対照的に)で見られるように、ハードウェアはミスで多くを行う必要はありません。例外が発生し、OS の TLB ミスハンドラが残りの処理を行います。

ASIDE: RISC VS. CISC

1980 年代には、コンピューターアーキテクチャのコミュニティで大きな戦いが起こりました。一方は複雑な命令セットコンピューティングのための CISC でした。もう一方の側では、RISC が Reduced Instruction Set Computing [PS81] でした。RISC 側は Berkeley の David Patterson と Stanford の John Hennessy(有名な著書 [HP06] の共著者でもある)が主導していましたが、後に John Cocke は RISC に関する彼の初期の研究で Turing 賞を受賞しました [CM00]。CISC 命令セットは多くの命令を持つ傾向があり、各命令は比較的強力です。たとえば、2 つのポインタと長さを取り、ソースからデスティネーションにバイトをコピーする文字列コピーが表示されます。CISC の背後にある考え方とは、命令はハイレベルのプリミティブでなければならず、アセンブリ言語そのものを使いやすくし、コードをよりコンパクトにする必要があるということでした。

RISC 命令セットはまったく逆です。RISC の背後にある重要なことは、命令セットは実際にコンパイラターゲットであり、実際にはすべてのコンパイラが高性能コードを生成するために使用できる単純なプリミティブであることです。したがって、RISC の主張者は、可能な限りハードウェア(特にマイクロコード)から多くのものを取り除き、単純なもの、均一で速いものを残しておきたいと主張しました。

初期段階では、RISC チップは著しく高速であったため、大きなインパクトを与えました [BC91]。多くの論文が書かれました。いくつかの企業が形成された(例えば、MIPS および Sun)。しかし、インテルなどの CISC メーカは、複雑な命令をマイクロ命令に変換したパイプラインステージを早期に追加するなど、RISC のような処理が可能な、多くの RISC 技術をプロセッサのコアに組み込んでいました。これらのイノベーションに加え、各チップ上のトランジスタ数の増加により、CISC は競争力を維持することができました。最終的な結果は、議論が崩れ落ちたことです。今日、両方のタイプのプロセッサを高速に動作させることができます。

19.4 TLB Contents: What's In There?

ハードウェア TLB の内容をより詳しく見てみましょう。一般的な TLB は、32,64、または 128 のエントリを持ち、fully associative(完全連想型) と呼ばれるものです。基本的には、TLB 内の任意の変換が可能であり、TLB 全体を並行して検索して目的の変換を見つけることを意味します。TLB エントリは次のようになります。



変換がこれらの場所のいずれかに終わる可能性があるため、VPN と PFN の両方が各エントリに存在することに注意してください (TLB は完全連想型キャッシュと呼ばれます)。ハードウェアはエントリを並行して検索し、一致するエントリがあるかどうかを確認します。

ASIDE: TLB VALID BIT 6= PAGE TABLE VALID BIT

よくある間違いは、TLB で見つかった有効なビットをページテーブルで見つかったものと混同することです。ページテーブルでは、ページテーブルエントリ (PTE) が無効とマークされている場合、ページがプロセスによって割り当てられていないことを意味し、正常に動作するプログラムによってアクセスされるべきではありません。無効なページがアクセスされたときの通常の応答は、プロセスを強制終了することによって応答する OS にトラップすることです。

TLB 有効ビットは、対照的に、単に TLB エントリ内に有効な変換があるかどうかを示します。例えば、システムがブートするとき、アドレス変換がそこにキャッシュされていないので、各 TLB エントリの共通初期状態は無効に設定されます。仮想メモリが有効になり、プログラムの実行が開始されて仮想アドレス空間にアクセスすると、TLB にはゆっくりとデータが読み込まれ、有効なエントリがすぐに TLB を満たします。

TLB の有効ビットはコンテキストスイッチを実行するときにも非常に便利です。これについては後で詳しく説明します。すべての TLB エントリを無効に設定することにより、システムは実行しようとしているプロセスが前のプロセスからの仮想から物理への変換を誤って使用しないようにすることができます。

もっと興味深いのは「その他のビット」です。たとえば、TLB には通常、エントリが有効な変換を持つかどうかを示す有効ビットがあります。また、ページにアクセスする方法を決定するプロテクションビットもあります (ページテーブルのように)。たとえば、コード・ページは読み取りと実行のマークが付けられ、ヒープ・ページは読み取りと書き込みのマークが付けられます。また、アドレス空間識別子、ダーティビットなど、いくつかの他のフィールドがあるかもしれません。詳細は以下を参照してください。

19.5 TLB Issue: Context Switches

TLB では、プロセス (したがってアドレス空間) を切り替えるときにいくつかの新しい問題が発生します。具体的には、TLB には、現在実行中のプロセスに対してのみ有効な仮想から物理への変換が含まれています。これらの変換は、他のプロセスにとって意味がありません。結果として、あるプロセスから別のプロセスに切り替えるときに、ハードウェアまたは OS(またはその両方) は、実行しようとしているプロセスが以前に実行されたプロセスの変換を誤って使用しないように注意する必要があります。

この状況をよりよく理解するために、例を見てみましょう。1つのプロセス (P1) が実行されているときには、TLB が有効な変換、すなわち P1 のページテーブルから来た変換をキャッシュしていると仮定します。この例では、P1 の 10 番目の仮想ページが物理フレーム 100 にマッピングされていると仮定します。

この例では、別のプロセス (P2) が存在し、OS がすぐにコンテキストスイッチを実行して実行することを決定する場合があります。ここでは、P2 の 10 番目の仮想ページが物理フレーム 170 にマッピングされているものとします。両方のプロセスのエントリが TLB にある場合、TLB の内容は次のようになります。

VPN	PFN	valid	prot
10	100	1	rwx
10	170	1	rwx

上記の TLB では、VPN 10 は PFN 100(P1) または PFN 170(P2) のいずれかに変換されますが、ハードウェアはどのプロセスがどのエントリを意味するのかを区別できません。したがって、TLB が複数のプロセス間で仮想化を正確かつ効率的にサポートするためには、さらにいくつかの作業を行う必要があります。なので、以下が問題になります。

THE CRUX: HOW TO MANAGE TLB CONTENTS ON A CONTEXT SWITCH

プロセス間のコンテキスト切り替え時に、最後のプロセスの TLB 内の変換は、実行しようとしているプロセスにとって意味がありません。この問題を解決するには、ハードウェアや OS は何をすべきですか？

この問題にはいくつかの解決策があります。1つのアプローチは、単にコンテキストスイッチ上で TLB をフラッシュし、次のプロセスを実行する前に TLB を空にすることです。ソフトウェアベースのシステムでは、これは明示的な（および特権の）ハードウェア命令で実現できます。ハードウェア管理の TLB では、ページテーブルのベースレジスタが変更されたときにフラッシュすることができます（OS はコンテキストスイッチ上で PTBR を変更する必要があります）。いずれの場合も、フラッシュ動作はすべての有効ビットを単に 0 に設定し、本質的に TLB の内容をクリアします。

各コンテキストスイッチで TLB をフラッシュすることにより、TLB 内の誤った変換にプロセスが偶然に遭遇することはないので、今や実用的な解決策があります。しかし、コストがかかります。プロセスが実行されるたびに、TLB ミスがデータ・ページやコード・ページに接觸すると、TLB ミスが発生する必要があります。OS がプロセス間で頻繁に切り替わると、このコストが高くなる可能性があります。

このオーバーヘッドを減らすために、コンテキストスイッチ間で TLB を共有できるようにするハードウェアサポートが追加されているシステムもあります。特に、一部のハードウェアシステムでは、TLB にアドレス空間識別子 (ASID) フィールドが用意されています。ASID はプロセス識別子 (PID) と考えることができますが、通常はビット数が少なくなります（たとえば、ASID の場合は 8 ビット、PID の場合は 32 ビット）。

上記の例の TLB を上から取り、ASID を追加すると、プロセスが TLB を容易に共有できることがわかります。それ以外は同一の変換を区別するためには、ASID フィールドだけが必要です。ここに、追加された ASID フィールドを含む TLB の描写があります。

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
—	—	0	—	—
10	170	1	rwx	2
—	—	0	—	—

したがって、アドレス空間識別子を使用すると、TLB は混乱を起こさずに異なるプロセスからの変換を同時に保持できます。もちろん、ハードウェアは変換を実行するために現在どのプロセスが実行されているかを知る必要があるため、OS はコンテクストスイッチ上で現在のプロセスの ASID にいくつかの特権レジスタを設定する必要があります。別として、TLB の 2 つのエントリが非常に似ている別のケースも考えているかもしれません。この例では、2 つの異なるプロセスに 2 つのエントリがあり、2 つの異なる VPN が同じ物理ページを指しています。

VPN	PFN	valid	prot	ASID
10	101	1	r-x	1
—	—	0	—	—
50	101	1	r-x	2
—	—	0	—	—

この状況は、たとえば、2 つのプロセスがページ（コードページなど）を共有する場合に発生する可能性があります。上記の例では、プロセス 1 が物理ページ 101 をプロセス 2 と共有しています。P1 はこのページをアドレス空間の 10 番目のページにマッピングし、P2 はアドレス空間の 50 番目のページにマッピングします。コードページ（バイナリまたは共有ライブラリ）の共有は、使用されている物理ページの数を減らし、メモリのオーバーヘッドを削減するので便利です。

19.6 Issue: Replacement Policy

他のキャッシングと同様に、TLB も同様に、考慮すべきもう一つの問題はキャッシングの置き換えです。具体的には、TLB に新しいエントリをインストールするときに、古いエントリを置き換える必要があります。

THE CRUX: HOW TO DESIGN TLB REPLACEMENT POLICY

新しい TLB エントリを追加する際に、どの TLB エントリを置き換える必要がありますか？ もち

ろん、目標はミス率を最小限に抑える（またはヒット率を高める）ことで、パフォーマンスを向上させることです。

ページをディスクにスワップする問題に取り組む際に、このようなポリシーを詳細に検討します。ここでは、いくつかの典型的なポリシーを強調します。最も一般的な方法の1つは、最も最近に使用されたエントリ（LRUエントリ）を削除することです。LRUは、最近使用されていないエントリが追い出される可能性が高いと仮定して、メモリ参照ストリームの局所性を利用しようとしています。別の典型的な手法は、無作為にTLBマッピングを退去させるランダムポリシーを使用することです。このようなポリシーは、その単純さとコーナーケースの振る舞いを回避する能力のために有用です。たとえば、LRUなどの「合理的な」ポリシーは、プログラムがサイズnのTLBを使用してn+1ページにわたってループするとき、非常に不合理な振る舞いをします。この場合、LRUはすべてのアクセス時にミスします。一方、ランダムはn+1ページあるうち、ランダムに1つを選ぶだけなので、はるかに優れています。

19.7 A Real TLB Entry

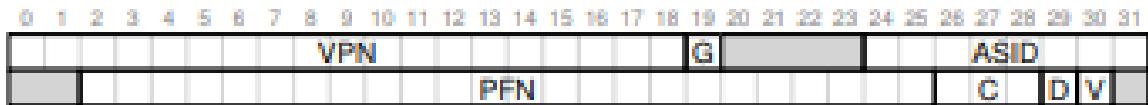


Figure 19.4: A MIPS TLB Entry

最後に、実際のTLBを簡単に見てみましょう。この例は、ソフトウェア管理のTLBを使用する最新のシステムであるMIPS R4000 [H93]の例です。わずかに単純化されたMIPSのTLBエントリを図19.4に示します。MIPS R4000は、4KBページの32ビットアドレス空間をサポートしています。したがって、一般的な仮想アドレスでは20ビットのVPNと12ビットのオフセットが予想されます。ただし、TLBに表示されているように、VPNには19ビットしかありません。結果として、ユーザーアドレスはアドレス空間の半分（カーネル用に予約されている残り）からしか得られないため、19ビットのVPNしか必要としません。VPNは最大24ビットの物理フレーム番号(PFN)に変換されるため、最大64GBの（物理的な）メインメモリ(2^{24} 4KBページ)を持つシステムをサポートできます。

MIPS TLBには他にも興味深いビットがいくつかあります。プロセス間でグローバルに共有されるページに使用されるグローバルビット(G)があります。従って、グローバルビットがセットされている場合、ASIDは無視されます。また、OSがアドレス空間を区別するために使用できる8ビットのASIDも表示されています(前述)。1つの質問： $256(2^8)$ を超えるプロセスが同時に実行されている場合、OSはどうすべきですか？最後に、ページがハードウェアによってどのようにキャッシュされるかを決定する3つのCoherence(C)ビットがあります(これらの注釈の範囲を少し超えてます)。ページが書き込まれたときにマークされるdirtyビット(これは後で使用します)。エントリに有効な変換が存在するかどうかをハードウェアに知らせるvaidビット。複数のページサイズをサポートするpage mask field(図示せず)もあります。なぜ大きなページを持つことが有用なのかを後で見ていきます。最後に、64ビットのうちのいくつかは未使用(図では網掛けのグレー)です。

MIPS TLBには通常32または64のエントリがあり、そのほとんどはユーザプロセスが実行されるときに使用されます。しかし、いくつかはOS用に予約されています。wired registerは、OSによって予約されるTLBのスロット数をハードウェアに伝えるためにOSによって設定できます。OSは、(例えば、TLBミスハンドラ内で)TLBミスが問題となる重要な時間にアクセスしたいコードおよびデータに対して、これらの予約されたマッピングを使用します。

MIPS TLBはソフトウェアで管理されているため、TLBを更新する手順が必要です。MIPSは、以下の4

つの命令を提供しています。TLBP : TLB を調べて、特定の変換がそこにあるかどうかを調べます。TLBR エントリの内容をレジスタに読み込む TLBR。特定の TLB エントリを置き換える TLBWI。ランダムな TLB エントリを置き換える TLBWR。OS はこれらの命令を使用して、TLB の内容を管理します。もちろん、これらの命令は特権です。ユーザプロセスが TLB の内容を変更する可能性がある場合(ヒント: マシンの引き継ぎ、悪意のある「OS」の実行、または Sun の消滅を含む)について何ができるか想像してください。

TIP: RAM ISN'T ALWAYS RAM (CULLER'S LAW)

ランダムアクセスメモリ(RAM)という用語は、言葉の通り、ランダムに別の RAM に速くアクセスできます。TLB などのハードウェア/OS 機能のために、RAM をこのように考えるのは一般的には良いですが、特にそのページが現在 TLB によってマップされていない場合は、メモリの特定のページにアクセスするとコストがかかる可能性があります。したがって、実装のヒントを覚えておくとよいでしょう。RAM は常に RAM であるとは限りません。場合によっては、アクセスされたページ数が TLB カバレッジを超えた場合など、アドレス空間にランダムにアクセスすると、パフォーマンスが著しく低下する可能性があります。当社の顧問の一人、David Culler は、多くの業績上の問題の原因として常に TLB を指していたため、この法律の名前を「Culler's Law」と名付けました。

19.8 Summary

私たちは、ハードウェアがアドレス変換を高速化するためにどのように役立つかを見てきました。アドレス変換キャッシュとして小型の専用オンチップ TLB を提供することにより、主メモリ内のページテーブルにアクセスすることなく、ほとんどのメモリ参照がうまく処理されます。したがって、一般的なケースでは、プログラムのパフォーマンスは、メモリがまったく仮想化されていないか、オペレーティングシステムにとって優れた成果であり、現代システムでのページングの使用に不可欠です。

しかし、TLB は存在するすべてのプログラムのために、その性能を提供できません。特に、プログラムが短期間にアクセスするページ数が TLB に収まるページ数を超える場合、プログラムは多数の TLB ミスを生成し、したがってかなり遅く実行されます。この現象は TLB の適用範囲を超えてると言い、特定のプログラムではかなり問題になる可能性があります。次の章で説明するように、1つのソリューションは、より大きなページサイズのサポートを含めることです。キーデータ構造を、より大きなページによってマッピングされるプログラムのアドレス空間の領域にマッピングすることによって、TLB の有効範囲を拡大することができる。ラージ・ページのサポートは、大規模でランダムにアクセスされる特定のデータ構造を持つデータベース管理システム(DBMS)などのプログラムによって利用されることがよくあります。

言及に値する他の TLB の問題の1つは、TLB アクセスが CPU パイプラインのボトルネックになりやすいことです。特に、physical index キャッシュと呼ばれます。このようなキャッシュでは、キャッシュにアクセスする前にアドレス変換が行われなければなりません。そのため、かなり遅くなる可能性があります。この潜在的な問題のために、人々は仮想アドレスを持つキャッシュにアクセスするためのあらゆる種類の賢明な方法を検討しているため、キャッシュヒットの場合には高価な変換手順を避けています。このような virtual index キャッシュは、パフォーマンス上の問題を解決しますが、ハードウェア設計にも新たな問題をもたらします。詳細については Wiggins の詳細な調査を参照してください [W03]。

参考文献

[BC91] “Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization”

D. Bhandarkar and Douglas W. Clark

Communications of the ACM, September 1991

A great and fair comparison between RISC and CISC. The bottom line: on similar hardware, RISC was about a factor of three better in performance.

[CM00] “The evolution of RISC technology at IBM”

John Cocke and V. Markstein

IBM Journal of Research and Development, 44:1/2

A summary of the ideas and work behind the IBM 801, which many consider the first true RISC microprocessor.

[C95] “The Core of the Black Canyon Computer Corporation”

John Couleur

IEEE Annals of History of Computing, 17:4, 1995

In this fascinating historical note, Couleur talks about how he invented the TLB in 1964 while working for GE, and the fortuitous collaboration that thus ensued with the Project MAC folks at MIT.

[CG68] “Shared-access Data Processing System”

John F. Couleur and Edward L. Glaser

Patent 3412382, November 1968

The patent that contains the idea for an associative memory to store address translations. The idea, according to Couleur, came in 1964.

[CP78] “The architecture of the IBM System/370”

R.P. Case and A. Padegs

Communications of the ACM. 21:1, 73-96, January 1978

Perhaps the first paper to use the term translation lookaside buffer. The name arises from the historical name for a cache, which was a lookaside buffer as called by those developing the Atlas system at the University of Manchester; a cache of address translations thus became a translation lookaside buffer. Even though the term lookaside buffer fell out of favor, TLB seems to have stuck, for whatever reason.

[H93] “MIPS R4000 Microprocessor User’s Manual”.

Joe Heinrich, Prentice-Hall, June 1993

Available: http://cag.csail.mit.edu/raw/documents/R4400_Uman_book_Ed2.pdf

[HP06] “Computer Architecture: A Quantitative Approach”

John Hennessy and David Patterson

Morgan-Kaufmann, 2006

A great book about computer architecture. We have a particular attachment to the classic first edition.

[I09] “Intel 64 and IA-32 Architectures Software Developer’s Manuals”

Intel, 2009

Available: <http://www.intel.com/products/processor/manuals>

In particular, pay attention to “Volume 3A: System Programming Guide Part 1” and “Volume 3B: System Programming Guide Part 2”

[PS81] “RISC-I: A Reduced Instruction Set VLSI Computer”

D.A. Patterson and C.H. Sequin

ISCA ’81, Minneapolis, May 1981

The paper that introduced the term RISC, and started the avalanche of research into simplifying computer chips for performance.

[SB92] “CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking”

Rafael H. Saavedra-Barrera

EECS Department, University of California, Berkeley

Technical Report No. UCB/CSD-92-684, February 1992

www.eecs.berkeley.edu/Pubs/TechRpts/1992/CSD-92-684.pdf

A great dissertation about how to predict execution time of applications by breaking them down into constituent pieces and knowing the cost of each piece. Probably the most interesting part that comes out of this work is the tool to measure details of the cache hierarchy (described in Chapter 5). Make sure to check out the wonderful diagrams therein.

[W03] “A Survey on the Interaction Between Caching, Translation and Protection”

Adam Wiggins

University of New South Wales TR UNSW-CSE-TR-0321, August, 2003

An excellent survey of how TLBs interact with other parts of the CPU pipeline, namely hardware caches.

[WG00] “The SPARC Architecture Manual: Version 9”

David L. Weaver and Tom Germond, September 2000

SPARC International, San Jose, California

Available: <http://www.sparc.org/standards/SPARCV9.pdf>

20 Paging: Smaller Tables

ここでは、ページングが導入する第 2 の問題に取り組んでいきます。ページテーブルが大きすぎるため、メモリを多く消費してしまいます。まずは、線形ページテーブルから始めましょう。線形ページテーブルはメモリ規模がかなり大きくなります。ここでも、4KB(2^{12} バイト) ページと 4 バイトのページテーブルエントリを持つ 32 ビットアドレス空間 (2^{32} バイト) を仮定します。したがって、アドレス空間には約 100 万の仮想ページがあります ($2^{32}/2^{12}$)。ページテーブルエントリサイズを掛け合わせると、ページテーブルのサイズが 4MB であることがわかります。また、システム内のすべてのプロセスについて 1 ページテーブルがあります。100 個のアクティブなプロセス (現代のシステムでは一般的ではない) で、ページテーブルのためだけに数百メガバイトのメモリを割り当てます。その結果、我々はこの重い負担を軽減するための技術を模索しています。

CRUX: HOW TO MAKE PAGE TABLES SMALLER?

単純な配列ベースのページテーブル (通常は線形ページテーブル) は大きすぎるため、一般的なシステムではあまりにも多くのメモリを占有します。どのようにしてページテーブルを小さくすることができますか？重要なアイデアは何ですか？これらの新しいデータ構造の結果、どのような非効率性が生じますか？

20.1 Simple Solution: Bigger Pages

ページテーブルのサイズを 1 つの簡単な方法で減らすことができます。32 ビットアドレス空間でもう一度考えてみましょう、しかし今回は 16KB のページを仮定します。したがって、18 ビットの VPN と 14 ビットのオフセットがあります。各 PTE(4 バイト) に同じサイズを仮定すると、線形ページテーブルには 2^{18} のエントリがあり、ページテーブルの合計サイズは 1MB となり、ページテーブルのサイズは 4 倍に縮小されます (この縮小はページサイズの 4 倍の増加と反比例です)

しかし、このアプローチの主な問題は、大きなページが各ページ内で無駄になるということです。内部断片化として知られている問題 (割り当て単位内のゴミ) です。アプリケーションはこうしてページの割り当てを終わらせますが、それぞれの小さなビットまたはある部分だけを使用するため、これらの大きすぎるページせいで、メモリはがいっぱいになります。したがって、ほとんどのシステムでは、通常の場合、4KB(x86 の場合) または 8KB(SPARCv9 の場合) の比較的小さいページサイズを使用します。しかし、ページテーブルの問題は単純には解決できません。

ASIDE: MULTIPLE PAGE SIZES

さて、MIPS、SPARC、x86-64 などの多くのアーキテクチャで複数のページサイズがサポートされていることに注意してください。通常、小さな (4KB または 8KB) ページサイズが使用されます。しかし、スマートなアプリケーションがそれを要求すると、アドレス空間の特定の部分に 1 つの大きなページ (たとえばサイズ 4MB) を使用することができ、頻繁に使用される (そして大きな) データ構造をそのようなスペースは单一の TLB エントリのみを消費します。このような大きなページの使用は、データベース管理システムやその他のハイエンドの商用アプリケーションでは一般的です。しかし、複数のページサイズを使用する主な理由は、ページテーブルのスペースを節約することではありません。TLB への負担を軽減し、プログラムが TLB ミスを多すぎることなく多くのアドレス空間にアクセスできるようにします。しかし、[N+02] のように複数のページサイズを使用すると、OS の仮想メモリマネージャがより複雑になるため、大規模なページを直接要求するアプリケーションに新しいインターフェイスをエクスポートするだけで大きなページが簡単に使

用されることがあります。

20.2 Hybrid Approach: Paging and Segments

あなたが人生の中で何かに合理的だが異なるアプローチを2つ持っているときは、常に両方の世界の最高を得ることができるかどうかを確認するために2つの組み合わせを調べる必要があります。このような組み合わせをハイブリッドと呼んでいます [M28]。

数年前、Multics の作成者 (特に Jack Dennis) は、Multics 仮想メモリシステム [M07] の構築においてこのような考えにチャレンジしました。具体的には、Dennis はページテーブルのメモリオーバーヘッドを減らすためにページングとセグメンテーションを組み合わせる考えを持っていました。典型的な線形ページテーブルをより詳細に調べることによって、これがなぜ機能するのかが分かります。ヒープとスタックの使用部分が小さいアドレス空間があるとします。この例では、1KB ページの小さな 16KB アドレス空間を使用します (図 20.1)。このアドレス空間のページテーブルは図 20.2 にあります。

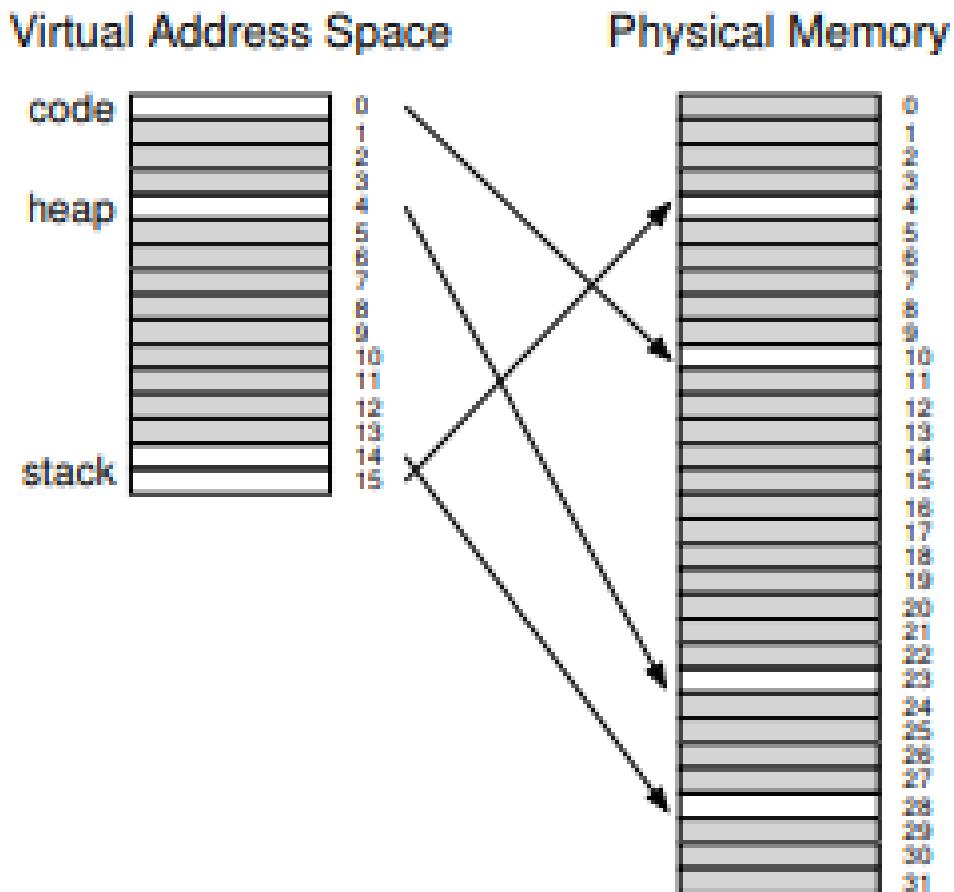


Figure 20.1: A 16KB Address Space With 1KB Pages

PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
23	1	rw-	1	1
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
28	1	rw-	1	1
4	1	rw-	1	1

Figure 20.2: A Page Table For 16KB Address Space

この例では、單一コードページ (VPN 0) が物理ページ 10、單一ヒープページ (VPN 4) から物理ページ 23、およびアドレススペースの他端の 2 つのスタックページ (VPN 14 および 15) にマップされ、物理ページ 28 および 4 にそれぞれマッピングされる。画像からわかるように、ページテーブルのほとんどは使用されず、無効なエントリがいっぱいです。これは、16KB の小さなアドレス空間用です。32 ビットのアドレス空間のページテーブルとそこに潜在するすべての無駄なスペースを想像してみてください！

したがって、私たちのハイブリッドアプローチとしては、プロセスのアドレス空間全体に单一のページテーブルを持たせる代わりに、論理セグメントごとに 1 つのページテーブルを作成するはどうですか？ この例では、3 つのページテーブルを用意しています。1 つはアドレススペースのコード、ヒープ、スタック部分用です。

ここで、セグメンテーションを覚えていれば、各セグメントが物理メモリにどこにあるのかを教えてくれるベースレジスタと、そのセグメントのサイズを教えてくれる境界レジスタまたは限界レジスタがありました。ハイブリッドでは、MMU にはそれらの構造があります。ここでは、ベースを使用してセグメント自体を指すのではなく、そのセグメントのページテーブルの物理アドレスを保持します。境界レジスタは、ページテーブルの終わり（すなわち、有効ページ数）を示すために使用されます。

明確な例を挙げてみましょう。4KB ページの 32 ビット仮想アドレス空間と、4 つのセグメントに分割されたアドレス空間を想定します。この例では、コード用、ヒープ用、スタック用の 3 つのセグメントのみを使用します。

アドレスが参照するセグメントを特定するために、アドレス空間の上位 2 ビットを使用します。00 が何も使われていないセグメント、コードの場合は 10、ヒープの場合は 10、スタックの場合は 11 が使用されていると

仮定しましょう。したがって、仮想アドレスは次のようにになります。



ハードウェアでは、コード、ヒープ、スタックごとに1つずつ、3つのベース/境界のペアがあるとします。プロセスが実行されているとき、これらの各セグメントのベース・レジスタには、そのセグメントの線形ページテーブルの物理アドレスが格納されます。したがって、システム内の各プロセスには、3つのページテーブルが関連付けられています。コンテキストスイッチでは、これらのレジスタは、新しく実行されているプロセスのページテーブルの位置を反映するように変更する必要があります。

TLB ミス (ハードウェア管理 TLB、すなわちハードウェアが TLB ミスを処理することを前提とする) では、ハードウェアはセグメントビット (SN) を使用して、使用するベースおよび境界のペアを決定する。次に、ハードウェアは、物理アドレスを取り込み、VPN と組み合わせてページテーブルエントリ (PTE) のアドレスを形成します。

```
SN = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
VPN = (VirtualAddress & VPN_MASK) >> VPN_SHIFT
AddressOfPTE = Base[SN] + (VPN * sizeof(PTE))
```

このシーケンスはよく知られているはずです。従来の線形ページテーブルとほぼ同じです。もちろん、唯一の違いは、単一のページテーブルベースレジスタの代わりに3つのセグメントベースレジスタの1つを使用することです。

ハイブリッド方式の重要な違いは、セグメントごとに境界レジスタが存在することです。各境界レジスタはセグメント内の最大有効ページの値を保持する。たとえば、コードセグメントが最初の3つのページ (0,1,2) を使用している場合、コードセグメントページテーブルには3つのエントリが割り当てられ、境界レジスタは3に設定されます。セグメントの終わりを超えたメモリアクセスは例外を生成し、プロセスの終了につながる可能性があります。このように、我々のハイブリッド手法は、線形ページテーブルと比較して大幅なメモリ節約を実現する。スタックとヒープの間の未割り当てページは、ページテーブル内のスペースを占有しなくなりました (有効でないものとしてマークする)。

しかし、気付いているように、このアプローチは問題がないわけではありません。まず、セグメント化を使用する必要があります。以前に議論したように、セグメンテーションは、アドレス空間の特定の使用パターンを想定しているので、我々が望むほどフレキシブルではありません。たとえば、大規模ではあるがまばらなヒープがあると、ページテーブルの無駄が多くなります。第二に、このハイブリッドは外部断片化を再び引き起こします。ほとんどのメモリはページサイズの単位で管理されますが、ページテーブルは任意のサイズ (PTE の倍数) になります。したがって、メモリ内の空き領域を見つけることはより複雑です。これらの理由から、人々は小さなページテーブルを実装するためのより良い方法を模索し続けました。

TIP: USE HYBRIDS

2つの良いアイデアと反対のアイデアがある場合は、両方の世界のベストを達成するためにハイブリッドに組み込むことができるかどうかを常に確認する必要があります。ハイブリッドトウモロコシ種は、例えば、任意の天然に存在する種よりも頑強であることが知られています。もちろん、すべてのハイブリッドが良い考えではありません。

20.3 Multi-level Page Tables

別のアプローチではセグメンテーションに頼るのではなく、同じ問題を攻撃します。つまり、ページテーブル内のすべての無効な領域をメモリ内に保存するのではなく、どのように取り除くのでしょうか？このアプローチは、線形ページテーブルをツリーのようなものに変えるため、マルチレベルのページテーブルと呼ばれています。このアプローチは、現代の多くのシステム（例えば、x86 [BOH10]）を採用するほど効果的です。ここで、このアプローチについて詳しく説明します。

マルチレベルのページテーブルの背後にある基本的な考え方は簡単です。まず、ページ・テーブルをページ・サイズの単位に切ります。ページテーブルエントリ（PTE）のページ全体が無効である場合、ページテーブルのそのページに対してはまったく割り当てません。ページテーブルのページが有効かどうか（有効な場合はメモリ内のどこにあるか）を追跡するには、ページディレクトリと呼ばれる新しい構造を使用します。したがって、ページディレクトリは、ページテーブルのページがどこにあるか、またはページテーブルのページ全体に有効なページが含まれていないことを通知するために使用できます。

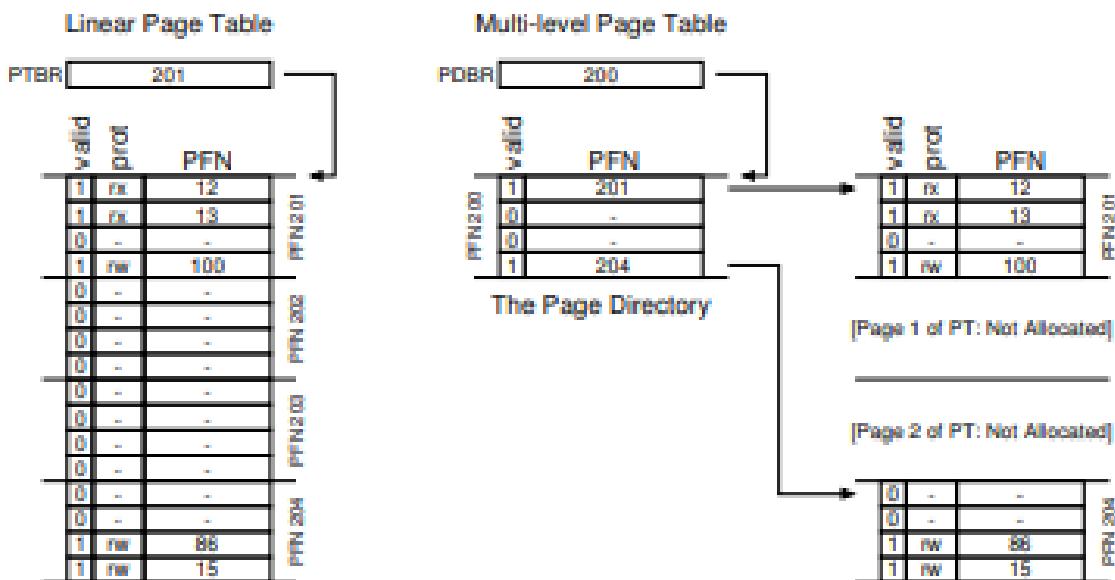


Figure 20.3: Linear (Left) And Multi-Level (Right) Page Tables

図 20.3 に例を示します。図の左側には古典的な線形ページテーブルがあります。アドレス空間の中間領域の大部分が有効ではないにもかかわらず、これらの領域に割り当てられたページテーブルスペース（すなわち、ページテーブルの中央の 2 ページ）が依然として必要です。右側には、複数レベルのページテーブルがあります。ページディレクトリは、ページテーブルの 2 ページだけを有効（最初と最後）としてマークします。したがって、ページテーブルの 2 つのページだけがメモリに存在します。したがって、マルチレベルテーブルが何をしているのかを視覚化する方法の 1 つを見ることができます。線形ページテーブルの一部を消して（他の用途ではそれらのフレームを解放して）、ページテーブルのどのページを割り当てるかを追跡します。

単純な 2 レベルの表のページディレクトリには、ページテーブルのページごとに 1 つのエントリが含まれています。これは、多数の page directory entries (PDE) で構成されています。PDE(最小限) は有効ビットと page frame number(PFN) を持ち、PTE に似ています。しかし、上記で示したように、この有効ビットの意味はわずかに異なっています。PDE エントリが有効であれば、(PFN を介して) エントリがポイントするページテーブルの少なくとも 1 つのページが有効であり、この PDE が指示するそのページ上の少なくとも 1 つ

の PTEにおいて、その PTE内の有効ビットは 1にセットされる。PDE エントリの有効ビットが 0の場合、PDE は定義されません。

多レベルのページテーブルはこれまで見てきたアプローチに比べて明らかな利点がいくつかあります。最初に、おそらく最も明白なことに、マルチレベル・テーブルは、使用しているアドレス空間の量に比例したページ・テーブル・スペースのみを割り当てます。したがって、一般にコンパクトで、省メモリなアドレス空間をサポートします。

第 2に、慎重に構築された場合、ページテーブルの各部分はページ内にきれいに収まるため、メモリの管理が容易になります。OS は、ページテーブルを割り当てたり、拡張したりする必要がある場合、次の空きページを簡単に取得できます。これを単純な(ページングされていない)線形ページテーブルと比較してください。これは VPN によってインデックスされた PTE の配列です。このような構造では、線形ページテーブル全体が物理メモリに連続して存在しなければいけません。大きなページテーブル(例えば 4MB)では、未使用の連続した空き物理メモリのような大きなチャンクを見つけることは非常に困難です。マルチレベル構造では、ページテーブルの部分を指すページディレクトリを使用して間接レベルを追加します。そのため、物理メモリに必要な場所にページテーブルページを置くことを可能にします。

TIP: UNDERSTAND TIME-SPACE TRADE-OFFS

データ構造を構築する際には、構築時に時間空間のトレードオフを常に考慮する必要があります。

通常、特定のデータ構造へのアクセスを高速化したい場合は、その構造体に対してメモリを多く使用するペナルティを支払わなければなりません。

マルチレベルテーブルには省メモリの代わりに高速化を犠牲にするコストがかかります。TLB ミスでは、メモリからの 2つのロードが、ページテーブル(ページディレクトリ用と、PTE 用)から適切な変換情報を得るために必要となります。したがって、マルチレベルテーブルは、時間空間のトレードオフの小さな例です。私たちはもっと小さなテーブルを望みそれを手に入れましたが、トレードオフがあります。一般的なケース(TLB ヒット)ではパフォーマンスは明らかに同一ですが、TLB ミスはこの小さなテーブルではコストが高くなります。別の問題点は、複雑さです。ページテーブル参照(TLB ミス時)を処理するのがハードウェアであれ、OS であれ、間違いなく単純な線形ページテーブル参照よりも複雑です。多くの場合、パフォーマンスを向上させたり、メモリを削減したりするために、複雑さを増やすことになります。マルチレベルテーブルの場合、貴重なメモリを節約するために、ページテーブルのルックアップがより複雑になっています。

A Detailed Multi-Level Example

マルチレベルのページテーブルの背後にある考え方をよりよく理解するために、例を挙げてみましょう。64KB のページで、サイズが 16KB の小さなアドレス空間を想像してみてください。したがって、VPN 用に 8ビット、オフセット用に 6ビットの 14ビットの仮想アドレス空間があります。線形ページテーブルは、たとえアドレス空間のごく一部が使用されているとしても、28(256)のエントリを持ちます。図 20.4 にそのようなアドレス空間の一例を示します。

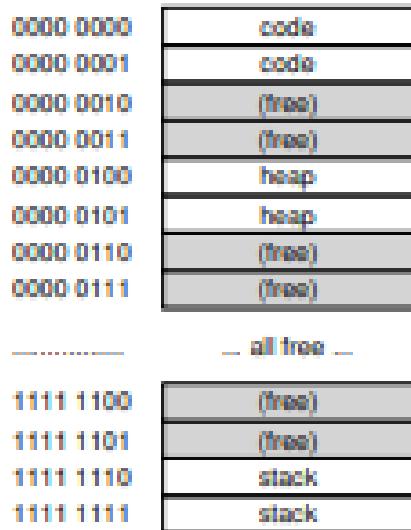


Figure 20.4: A 16KB Address Space With 64-byte Pages

TIP: BE WARY OF COMPLEXITY

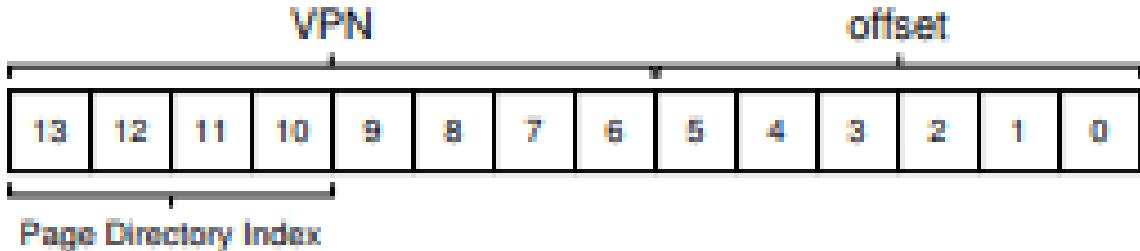
システム設計者は、システムに複雑さを加えることに注意する必要があります。優れたシステム構築者は、手元にあるタスクを達成する最も簡単なシステムを実装しています。たとえば、ディスク容量が豊富な場合は、できるだけ数バイトで使用しにくいファイルシステムを設計しません。同様に、プロセッサが高速であれば、CPU 内に最適化された手作業で作成された作業用のコードよりも、クリーンでわかりやすいモジュールを OS に書き込むほうがよいでしょう。早期に最適化されたコードや他の形式で、不必要的複雑さに注意してください。このようなアプローチにより、システムの理解、維持、デバッグが難しくなります。Antoine de Saint-Exupery は次のように書いています。「完璧は、もはや何も追加する必要がなくなったときではなく、もはや取り去るものがなくなったときでもありません」彼は「実際に目的を達成するよりも完璧と言うのは簡単である。」と書いていませんでした。

この例では、仮想ページ 0 と 1 はコード用、仮想ページ 4 と 5 はヒープ用、仮想ページ 254 と 255 はスタック用です。アドレス空間の残りのページは使用されません。

このアドレス空間の 2 レベルのページテーブルを作成するには、完全な線形ページテーブルから始め、ページサイズの単位に分割します。私たちの全テーブル（この例では）には 256 個のエントリがあります。各 PTE が 4 バイトのサイズであると仮定します。したがって、私たちのページテーブルは 1KB(256×4 バイト) のサイズです。64 バイトのページがある場合、1KB のページテーブルは 16 の 64 バイトページに分割できます。各ページは 16 個の PTE を保持できます。

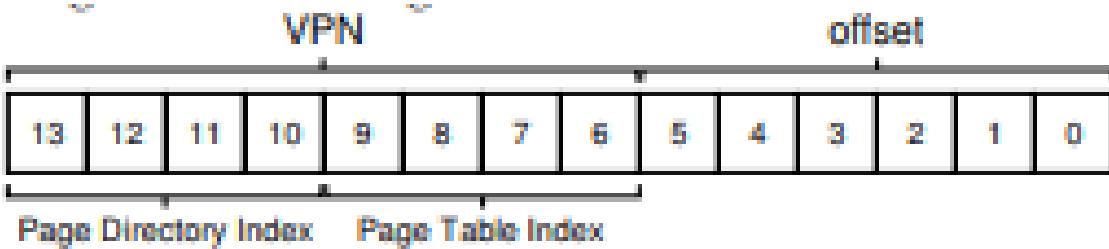
ここで理解する必要があるのは、VPN を使って、まずページディレクトリに、次にページテーブルのページにインデックスを付ける方法です。それぞれがエントリの配列であることを忘れないでください。したがって、私たちが把握する必要があるのは、それぞれの VPN からインデックスを作成する方法だけです。

最初にページディレクトリにインデックスを作成しましょう。この例のページテーブルは小さく、16 ページにわたる 256 エントリです。ページディレクトリには、ページテーブルのページごとに 1 つのエントリが必要です。従って、それは 16 のエントリーを有する。その結果、ディレクトリにインデックスを作成するには、VPN の 4 ビットが必要です。次のように、VPN の上位 4 ビットを使用します。



page directory index(略して PDIIndex) は、VPN から page directory entry (PDE) を取り出すために使われます。使われる計算式は、 $PDEAddr = PageDirBase + (PDIIndex * sizeof(PDE))$ となります。これにより、ページディレクトリが作成されます。ここでは、変換の進捗状況を確認します。

ページディレクトリのエントリが無効とマークされている場合、アクセスが無効であることがわかり、例外が発生します。しかし、PDE が有効な場合は、さらに多くの作業が必要です。具体的には、このページエントリエントリが指すページテーブルのページからページテーブルエントリ (PTE) をフェッチする必要があります。この PTE を見つけるには、VPN の残りのビットを使用してページテーブルの部分にインデックスを設定する必要があります。



このページテーブルインデックス (略して PTIndex) を使用して、ページテーブル自体にインデックスを付けて、PTE のアドレスを指定できます。

```
PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
```

page directory entry (PDE) から取得した page frame number(PFN) は、PTE のアドレスを形成するためにページ・テーブル索引と組み合わせる前に、左シフトして配置する必要があることに注意してください。

これがすべて意味をなさないかどうかを確認するために、いくつかの実際の値を持つ複数レベルのページテーブルを記入し、1つの仮想アドレスを変換します。この例のページ・ディレクトリから始めましょう (図 20.5 の左側)。この図では、各 page directory entry (PDE) がアドレス・スペースのページ・テーブルのページについて何かを記述していることが分かります。この例では、アドレス空間に (開始時と終了時に) 2 つの有効な領域と、その間にいくつかの無効なマッピングがあります。

物理ページ 100(ページテーブルの 0 番目のページの物理フレーム番号) には、アドレス空間内の最初の 16 個の VPN 用の 16 個のページテーブルエントリの最初のページがあります。ページテーブルのこの部分の内容については、図 20.5(中央部) を参照してください。

Page Directory		Page of PT (@PFN:100)			Page of PT (@PFN:101)		
PFN	valid?	PFN	valid	prot	PFN	valid	prot
100	1	10	1	r-x	—	0	—
—	0	23	1	r-x	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	80	1	rW-	—	0	—
—	0	59	1	rW-	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
101	1	—	0	—	55	1	rW-
					45	1	rW-

Figure 20.5: A Page Directory, And Pieces Of Page Table

ページテーブルのこのページには、最初の 16 個の VPN のマッピングが含まれています。この例では、VPN 0 と 1 が有効であり (コードセグメント)、4 と 5(ヒープ) です。したがって、テーブルは、これらのページのそれぞれについてのマッピング情報を持っています。残りのエントリは無効とマークされます。ページテーブルの他の有効なページは、PFN 101 内にある。このページは、アドレス空間の最後の 16 個の VPN のマッピングを含みます。詳細は図 20.5(右) を参照してください。

この例では、VPN 254 と 255(スタック) は有効なマッピングを持っています。うまくいけば、この例からわかるように、マルチレベルの索引構造でどれくらいのスペースを節約できるかということです。この例では、線形ページテーブルに 16 ページすべてを割り当てるのではなく、ページディレクトリに 1 つ、ページテーブルのチャンクに有効なマッピングが 2 つずつ割り当てます。大きな (32 ビットまたは 64 ビット) アドレス空間の節約は大きく働くでしょう。

最後に、この情報を使用して変換を実行してみましょう。ここには、VPN 254 の 0 番目のバイト : 0x3F80、または 11 11000 1000 0000 のバイナリを参照するアドレスがあります。

VPN の上位 4 ビットを使用してページディレクトリにインデックスを作成することを思い出してください。したがって、1111 は、上記のページディレクトリの最後のエントリ (0 番目から 15 番目のエントリ) を選択します。これは、アドレス 101 に位置するページテーブルの有効なページを示します。次に、VPN の次の 4 ビット (1110) を使用して、ページテーブルのそのページにインデックスを付け、欲しい PTE を見つけます。1110 は、ページ上の次の最後 (14 番目) のエントリであり、仮想アドレス空間のページ 254 が物理ページ 55 にマップされていることを示しています。PFN = 55(または 16 進数 0x37) を offset = 000000 に連結すると、欲しい物理アドレスを形成し、その要求をメモリシステムに発行することができます。

```
PhysAddr = (PTE.PFN << SHIFT) + offset
= 00 1101 1100 0000 = 0x0DC0.
```

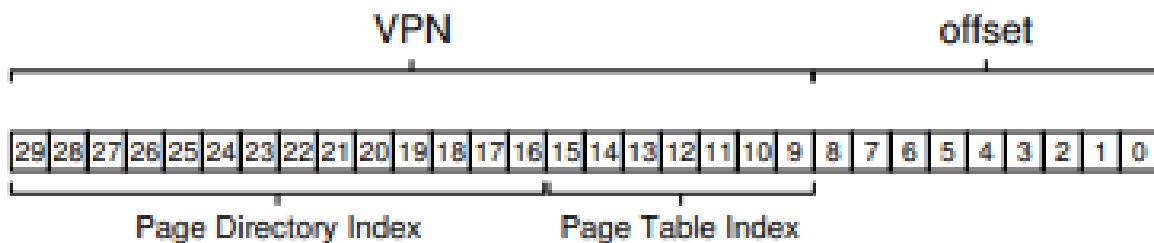
ページテーブルのページを指すページディレクトリを使用して、2 レベルのページテーブルを構築する方法についていくつか考えてください。残念なことに、ここで説明するように、ページテーブルの 2 つのレベルでは不十分な場合があります。

More Than Two Levels

これまでの例では、複数レベルのページテーブルはページディレクトリとページテーブルの 2 つのレベルしか持たないと仮定しています。場合によっては、より深いツリーが可能です（実際には必要です）。簡単な例を取り上げて、より深いマルチレベルテーブルが役立つ理由を説明しましょう。この例では、30 ビットの仮想アドレス空間と小さな（512 バイト）ページがあるとします。したがって、仮想アドレスには 21 ビットの仮想ページ番号コンポーネントと 9 ビットのオフセットがあります。

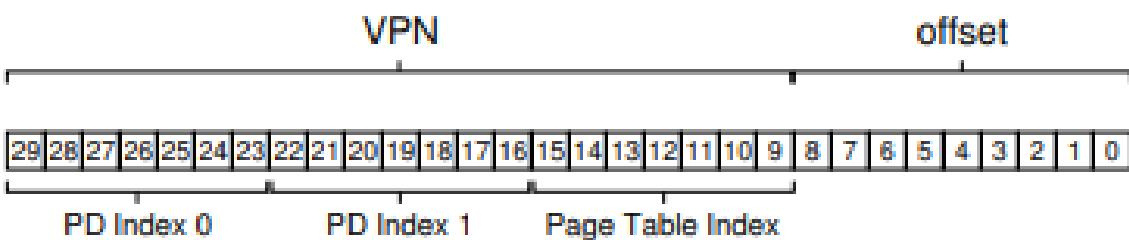
マルチレベルのページテーブルを構築することの目標を忘れないでください。ページテーブルの各部分を 1 ページに収めるようにします。これまでには、ページテーブル自体についてのみ検討していました。ただし、ページディレクトリが大きすぎるとどうなりますか？

ページテーブルのすべての部分をページ内に収めるために、複数レベルのテーブルに必要なレベルの数を決定するには、ページ内にいくつのページテーブルエントリが取まるかを判断することから始めます。ページサイズが 512 バイトで、PTE サイズが 4 バイトであると仮定すると、1 ページに 128 個の PTE を収めることができます。ページテーブルのページにインデックスを付けると、VPN の最下位 7 ビット ($\log_2 128$) がインデックスとして必要であると判断できます。



上記の図から気づくかもしれないのは、（大きな）ページディレクトリに残っているビット数です。ページディレクトリのエントリが 2^{14} であれば、ページは 1 ページではなく 128 ページになります。ページに収まる複数レベルのページテーブルが消えます。

この問題を解決するために、ページディレクトリ自体を複数のページに分割し、その上に別のページディレクトリを追加して、ページディレクトリのページを指すようにして、ツリーのレベルをさらに高めます。したがって、仮想アドレスを次のように分割することができます。



現在、上位レベルのページディレクトリのインデックスを作成するときには、仮想アドレスの最上位ビット（図の PD インデックス 0）を使用します。この索引を使用して、top level page directory から page directory entry をフェッチすることができます。有効な場合、トップレベル PDE からの物理フレーム番号と VPN の次の部分（PD インデックス 1）を組み合わせて、ページディレクトリの第 2 レベルを調べます。最後に、有効であれば、PTE アドレスは、第 2 レベルの PDE からのアドレスと組み合わされたページテーブルインデックスを使用することによって形成することができます。

The Translation Process: Remember the TLB

2 レベルのページテーブルを使用してアドレス変換のプロセス全体を要約すると、アルゴリズムフロー形式で制御フローを示します(図 20.6)。この図は、すべてのメモリ参照時にハードウェアで何が起こるかを示しています(ハードウェア管理 TLB を前提としています)。

図からわかるように、複雑なマルチレベル・ページ・テーブル・アクセスが発生する前に、ハードウェアはまず TLB をチェックします。ヒットすると、物理アドレスは、以前と同様にページテーブルに全くアクセスせずに直接形成される。TLB ミス時にのみ、ハードウェアは完全なマルチレベルルックアップを実行する必要があります。このパスでは、従来の 2 レベルページテーブルのコストを確認できます。有効な変換を検索するための 2 つの追加メモリアクセスです。

```

1   VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2   (Success, TlbEntry) = TLB_Lookup(VPN)
3   if (Success == True) // TLB Hit
4       if (CanAccess(TlbEntry.ProtectBits) == True)
5           Offset = VirtualAddress & OFFSET_MASK
6           PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7           Register = AccessMemory(PhysAddr)
8       else
9           RaiseException(PROTECTION_FAULT)
10  else // TLB Miss
11      // first, get page directory entry
12      PDIndex = (VPN & PD_MASK) >> PD_SHIFT
13      PDEAddr = PDBR + (PDIndex * sizeof(PDE))
14      PDE = AccessMemory(PDEAddr)
15      if (PDE.Valid == False)
16          RaiseException(SEGMENTATION_FAULT)
17      else
18          // PDE is valid: now fetch PTE from page table
19          PTIndex = (VPN & PT_MASK) >> PT_SHIFT
20          PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
21          PTE = AccessMemory(PTEAddr)
22          if (PTE.Valid == False)
23              RaiseException(SEGMENTATION_FAULT)
24          else if (CanAccess(PTE.ProtectBits) == False)
25              RaiseException(PROTECTION_FAULT)
26          else
27              TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
28          RetryInstruction()

```

Figure 20.6: Multi-level Page Table Control Flow

20.4 Inverted Page Tables

反転したページテーブルを使用すると、ページテーブルの世界でさらに極端なスペース節減が見られます。ここでは、多くのページテーブル(システムのプロセスごとに 1 つ)を持つ代わりに、システムの物理ページごとに 1 つのページテーブルを保持しています。このエントリは、どのプロセスがこのページを使用しているか、およびそのプロセスのどの仮想ページがこの物理ページにマッピングされているかを示します。

正しいエントリを見つけることは、このデータ構造を通して検索することになります。線形スキヤンは高価であり、検索を高速化するためにハッシュテーブルが基本構造上に構築されることが多いです。PowerPC はそのようなアーキテクチャ [JM98] の一例です。

より一般的には、逆ページテーブルは、最初から述べたことを示しています。ページテーブルは単なるデータ構造です。あなたは、データ構造を使ってたくさん面白いことをすることができます。小さくても大きくても、遅くとも速くしても構いません。多段ページテーブルと逆ページテーブルは、できることの多くの2つの例に過ぎません。

20.5 Swapping the Page Tables to Disk

最後に、1つの最終的な仮定の緩和について議論する。ここまででは、カーネルが所有する物理メモリにページテーブルが存在すると仮定しています。しかし、ページテーブルのサイズを減らすための多くのトリックがあっても、メモリに入るには大きすぎる可能性があります。したがって、一部のシステムでは、このようなページテーブルをカーネル仮想メモリに配置するため、少しでもメモリが圧迫がされた場合に、システムがこれらのページテーブルの一部をディスクにスワップすることができます。これについては、今後の章(VAX/VMSのケーススタディ)で詳しく説明します。一度、ページをメモリ内外に移動する方法を理解したら、さらに詳しく説明します。

##20.6 Summary 実際のページテーブルの構築方法を見てきました。必ずしも線形配列ではなく、より複雑なデータ構造である必要があります。このようなテーブルは、時間と空間のトレードオフであり、テーブルが大きくなればなるほどTLBミスを高速に処理できるだけになります。しかし、テーブルが小さくなればなるほど省メモリになりますがTLBミスが発生したら遅くなります。したがって、正しい構造の選択は、指定された環境の制約に強く依存します。

メモリが制約されたシステム(多くの古いシステムのように)では、小さな構造が理にかなっています。妥当な量のメモリを持ち、多数のページを積極的に使用する仕事量では、TLBミスの速度を上げる大きなテーブルが適切な選択肢になる可能性があります。ソフトウェア管理されたTLBを使用すると、データ構造全体がオペレーティングシステム革新者の喜びにつながります。どのような新しい構造を思いつくことができますか?どのような問題を解決しますか?これらの質問を考えて、オペレーティングシステム開発者だけが夢を見ることができる大きな夢を描いてみてください。

参考文献

- [BOH10] “Computer Systems: A Programmer’s Perspective”
Randal E. Bryant and David R. O’Hallaron
Addison-Wesley, 2010
We have yet to find a good first reference to the multi-level page table. However, this great textbook by Bryant and O’Hallaron dives into the details of x86, which at least is an early system that used such structures. It’s also just a great book to have.
- [JM98] “Virtual Memory: Issues of Implementation”
Bruce Jacob and Trevor Mudge
IEEE Computer, June 1998
An excellent survey of a number of different systems and their approach to virtualizing memory. Plenty of details on x86, PowerPC, MIPS, and other architectures.
- [LL82] “Virtual Memory Management in the VAX/VMS Operating System”
Hank Levy and P. Lipman
IEEE Computer, Vol. 15, No. 3, March 1982
A terrific paper about a real virtual memory manager in a classic operating system, VMS. So terrific, in fact, that we’ll use it to review everything we’ve learned about virtual memory thus far a few chapters

from now.

[M28] “Reese’s Peanut Butter Cups”

Mars Candy Corporation.

Apparently these fine confections were invented in 1928 by Harry Burnett Reese, a former dairy farmer and shipping foreman for one Milton S. Hershey. At least, that is what it says on Wikipedia. If true, Hershey and Reese probably hated each other’s guts, as any two chocolate barons should.

[N+02] “Practical, Transparent Operating System Support for Superpages”

Juan Navarro, Sitaram Iyer, Peter Druschel, Alan Cox

OSDI ’02, Boston, Massachusetts, October 2002

A nice paper showing all the details you have to get right to incorporate large pages, or superpages, into a modern OS. Not as easy as you might think, alas.

[M07] “Multics: History”

Available: <http://www.multicians.org/history.html>

This amazing web site provides a huge amount of history on the Multics system, certainly one of the most influential systems in OS history. The quote from therein: “Jack Dennis of MIT contributed influential architectural ideas to the beginning of Multics, especially the idea of combining paging and segmentation.”
(from Section 1.2.1)

21 Beyond Physical Memory: Mechanisms

ここまででは、アドレス空間は非現実的に小さく、物理メモリに収まると仮定しました。実際には、実行中のすべてのプロセスのアドレス空間がすべてメモリに収まると仮定しています。我々は今、これらの大きな仮定を緩和し、多数の並行して動作する大規模なアドレス空間をサポートしたいと仮定します。

これを行うには、メモリ階層に追加のレベルが必要です。ここまででは、すべてのページが物理メモリに存在すると仮定しています。しかし、大規模なアドレス空間をサポートするために、OSは現在大きな需要がないものはアドレス空間の一部を移動する場所が必要です。一般に、そのような場所の特徴は、メモリよりも容量が大きいことです。その結果、一般的に速度が遅くなります（速度が速ければメモリとして使用します）。現代のシステムでは、この役割は通常、ハードディスクドライブによって提供されます。したがって、私たちのメモリ階層では、容量が大きくて速度が遅いハードドライブが一番上にあり、容量が小さくて速いメモリは一番下にあります。

THE CRUX: HOW TO GO BEYOND PHYSICAL MEMORY

OSはどのようにして、より大きい、より遅いデバイスを使って、大きな仮想アドレス空間の錯覚を透過的に提供することができますか？

1つ疑問を持っているかもしれません。それは、プロセスのために单一の大きなアドレス空間をサポートしたいのはなぜ？ ということです。その答えは利便性と使いやすさです。アドレス空間が大きいと、プログラムのデータ構造に十分な余裕があれば心配する必要はありません。むしろ、必要に応じてメモリを割り当て、自然にプログラムを書くだけです。OSが提供する強力な錯覚です。メモリオーバーレイを使用していた旧式のシステムでは、プログラマは必要に応じてコードやデータを手動でメモリ内外に移動する必要がありました [D97]。関数の呼び出しやデータへのアクセスの前に、まずコードやデータをメモリに配置する必要があります。

ASIDE: STORAGE TECHNOLOGIES

I/Oデバイスが実際にどのように後で動作するかについてもっと深く掘り下げて調べます（I/Oデバイスの章を参照）。もちろん、低速デバイスはハードディスクである必要はありませんが、FlashベースのSSDなど、よりモダンなものになる可能性があります。それらについても話します。現時点では、物理メモリよりも大きな非常に大きな仮想メモリの錯覚を構築するのに役立つ大きなデバイスと比較的遅いデバイスがあると仮定します。

单一のプロセスだけではなく、スワップ空間を追加することで、複数の同時に実行されているプロセスに対して、大きな仮想メモリの錯覚をOSがサポートできるようになります。初期のマシンはすべてのプロセスが必要とするすべてのページを一度に保持することができないため、マルチプログラミング（複数のプログラムを「すぐに」実行して、マシンをより良く利用する）は、ほとんどのページを交換する能力を要求しました。したがって、マルチプログラミングと使いやすさの組み合わせにより、物理的に利用可能なメモリよりも多くのメモリを使用することがサポートされるようになります。これは、現代のすべてのVMシステムが行うことです。

21.1 Swap Space

最初に行う必要のある作業は、ページを前後に移動するためにディスク上にスペースを確保することです。オペレーティングシステムでは、スワップスペースと呼ばれるスペースは、メモリの外にページをスワップして、ページをメモリ内にスワップするためです。したがって、OSはページサイズ単位でスワップ領域の読み

書きを行うことができます。これを行うには、OS は特定のページのディスクアドレスを覚えておく必要があります。

スワップスペースのサイズは重要です。最終的には、特定の時間にシステムが使用できるメモリページの最大数を決定します。わかりやすくするために、今のところそれは非常に大きいものとしましょう。

ちょっとした例(図 21.1)では、4 ページの物理メモリと 8 ページのスワップ領域の小さな例を見ることがあります。この例では、3 つのプロセス(Proc 0、Proc 1、および Proc 2)が物理メモリを積極的に共有しています。しかし、3 つのそれぞれは有効なページの一部のみをメモリに持ち、残りの部分はディスクのスワップ領域に配置します。4 番目のプロセス(Proc 3)は、すべてのページをディスクにスワップアウトしているため、現在実行中ではありません。スワップのブロックは空きのままであります。この小さな例からも、スワップスペースを使用すると、システムが実際よりも大きなメモリをまとめることができます。

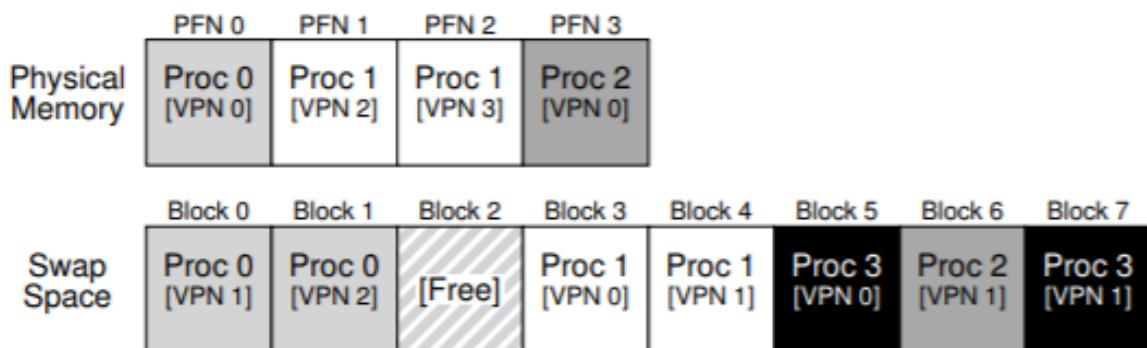


Figure 21.1: Physical Memory and Swap Space

スワップスペースは、トラフィックをスワップするためのディスク上の唯一の場所ではありません。たとえば、プログラムバイナリ(たとえば、ls、または独自のコンパイルされたメインプログラム)を実行しているとします。このバイナリのコードページは、ディスク上に最初に発見され、プログラムが実行されるとメモリにロードされます(プログラムの実行開始時に一度に、現代システムでは一度に 1 ページずつ必要に応じてロードされます)。ただし、システムが他のニーズに合わせて物理メモリに空き領域を確保する必要がある場合は、後からでも、ファイルシステム内のディスク上にあるバイナリを再度スワップすることができるため、これらのコードページのメモリ領域を安全に再利用できます。

21.2 The Present Bit

ディスク上にスペースを確保したので、ディスクとのスワップページをサポートするために、いくつかのマシンをシステムに追加する必要があります。わかりやすくするために、ハードウェア管理 TLB を備えたシステムを想定してみましょう。

まず、メモリ参照で何が起こるかを思い出してください。実行中のプロセスは、仮想メモリ参照(命令フェッチまたはデータアクセス用)を生成します。この場合、ハードウェアはメモリから目的のデータをフェッチする前に、それらを物理アドレスに変換します。

ハードウェアは最初に仮想アドレスから VPN を抽出し、TLB に一致(TLB ヒット)をチェックし、ヒットした場合に結果の物理アドレスを生成してメモリからフェッチします。これは速い(追加のメモリアクセスを必要としない)ので、これは一般的なケースです。

VPN が TLB に見つからない場合(TLB ミス)、ハードウェアは(ページテーブルベースレジスタを使用して)ページテーブルをメモリに配置し、VPN を使用してインデックスを特定し、このページのページテーブルエントリ(PTE)を検索します。ページが有効で物理メモリに存在する場合、ハードウェアは PTE から PFN

を抽出し、それを TLB にインストールし命令を再試行します。今回は TLB ヒットを生成します。ここまで順調ですね。

しかし、ページをディスクにスワップさせたい場合は、さらにメカニズムを追加する必要があります。具体的には、ハードウェアが PTE を調べると、ページが物理メモリに存在しないことがあります。ハードウェア（またはソフトウェア管理の TLB アプローチにおける OS）がこれを判断する方法は、現在のビットと呼ばれる各ページテーブルエントリ内の新しい情報によって行われます。現在のビットが 1 に設定されている場合、ページが物理メモリに存在し、すべてが上記のように進行することを意味します。ゼロに設定されている場合、ページはメモリにではなく、ディスクのどこかにあります。物理メモリにないページにアクセスする行為は、通常、ページフォルトと呼ばれます。

ASIDE: SWAPPING TERMINOLOGY AND OTHER THINGS

仮想メモリシステムの用語は、マシンやオペレーティングシステムでは多少混乱し、変わることもあります。例えば、ページフォルトは、より一般的には、何らかの種類のフォルトを生成するページテーブルへの参照にアクセスすることができます。これは、ここで議論しているフォルトのタイプ、すなわちページが存在しないフォルトを含むことができます。しかし、不正なメモリアクセスを参照した場合はどうでしょう。確かに、プロセスの仮想アドレス空間にマップされたページへの法的なアクセス（当然のことながら物理メモリではない）を「障害」と呼ぶことは間違いません。より正確には、それをページミスと呼ばれるべきです。しかし、プログラムが「ページフォルト」であると言うと、OS がディスクにスワップアウトした仮想アドレス空間の一部にアクセスしていることをよく意味します。私たちは、この動作が「障害」として知られるようになった理由は、それを処理するオペレーティングシステムのメカニズムに関係していると考えています。めったに起きない何かが起こったとき、すなわち、ハードウェアが何らかの処理方法を知らないとき、ハードウェアは制御を OS に移し、処理してくれることを望むでしょう。この場合、プロセスがアクセスしたいページがメモリから欠落しています。ハードウェアが可能なのは例外を発生させ、そこから OS が引き継ぎます。これは、プロセスが何らかの違法行為を行った場合と同じであるため、アクティビティを「障害」と呼ぶのは驚くことではありません。

ページフォルトが発生すると、OS はページフォルトを処理するために呼び出されます。ここで説明するように、ページフォルトハンドラと呼ばれる特定のコードが実行され、ページフォルトを処理する必要があります。

21.3 The Page Fault

TLB ミスでは、ハードウェア管理された TLB（ハードウェアがページテーブルを参照して目的の変換を検索する）とソフトウェア管理 TLB（OS が実行する場所）の 2 種類のシステムがあります。いずれのタイプのシステムにおいても、ページが存在しない場合、ページフォルトを処理するために OS が担当します。適切に指定された OS ページフォルトハンドラが実行され、何をすべきかが決定されます。事実上、すべてのシステムがソフトウェアのページフォルトを処理します。ハードウェア管理の TLB を使用しても、ハードウェアはこの重要な義務を管理するために OS を信頼します。

ページが存在せず、ディスクにスワップされている場合、OS はページフォルトを処理するためにページをメモリにスワップする必要があります。ここに疑問が生じます。OS は、どのようにして目的のページを見つけるかを知っていますか？多くのシステムでは、ページテーブルはそのような情報を格納するための自然な場所です。一般的な OS の場合は、ページの PFN などのデータに通常使用される PTE のビットをディスクアドレスとして使用できます。OS がページのページフォルトを受信すると、OS は PTE を調べてアドレスを見つけ、ディスクに要求を発行してページをメモリにフェッチします。

ASIDE: WHY HARDWARE DOESN'T HANDLE PAGE FAULTS

TLB を学んだ経験から、ハードウェア設計者は、OS に任せるのを嫌っています。では、なぜ彼らは OS にページ違反を処理するのを信頼しているのでしょうか？ 主な理由はいくつかあります。まず、ディスクへのページフォルトは遅いです。OS が大量の命令を実行してフォールトを処理するのに時間がかかる場合でも、ディスク操作そのものは伝統的に遅すぎて、逆に実行中のソフトウェアの様々なオーバーヘッドを最小限に抑えられます。次に、ページフォルトを処理できるようにするために、ハードウェアはスワップ領域、I/O をディスクに発行する方法、および現在はあまり知られていないその他多くの詳細を理解する必要があります。したがって、パフォーマンスとシンプルさの両方の理由から、OS はページフォルトを処理し、ハードウェアタイプも満足できるものになります。

ディスク I/O が完了すると、OS はページテーブルを更新してページを現在のものとしてマークし、ページテーブルエントリ (PTE) の PFN フィールドを更新して、新たにフェッチされたページのメモリ内の位置を記録し、命令を再試行します。この次の試行は、TLB ミスを生成し、サービスされ、TLB 変換を更新します（このステップを回避するためにページフォルトを処理するときに TLB を交互に更新することができます）。最後に、最後の再起動は TLB 内の変換を見つけ、変換された物理アドレスのメモリから望んだデータまたは命令をフェッチすることに進みます。

I/O が実行されている間は、プロセスはブロックされた状態になります。従って、ページフォルトが処理されている間に、OS は他の準備完了プロセスを自由に実行することができる。I/O は高価なので、あるプロセスの I/O(ページフォルト) と他のプロセスの実行とのオーバーラップは、マルチプログラムされたシステムがそのハードウェアを最も効果的に使用できるもう 1 つの方法です。

21.4 What If Memory Is Full?

上記のプロセスでは、スワップ領域からページをページインするための空きメモリが十分にあると想定していることがわかります。もちろん、これは当てはまらないかもしれません。つまり、メモリがいっぱい（またはそれに近い）かもしれません。したがって、OS は、最初に 1 つまたは複数のページをページアウトして、その OS が投入しようとしている新しいページのためのスペースを作りたいと思うかもしれません。プロセスを選んでキックアウトまたは置換することをページ置換ポリシーといいます。

上記のように、間違ったページを見つけキックアウトや置換をすると、プログラムのパフォーマンスに大きなコストをかけることになります。そのため、良いページ置換ポリシーを作成することに多くの考えがあります。間違った決定をすると、プログラムはメモリのようなスピードではなく、ディスクのようなスピードで動作する可能性があります。現在の技術では、プログラムが 1 万分の 1 倍または 10 万分の 1 倍遅く実行される可能性があります。したがって、そのようなポリシーは我々がある程度詳細に研究すべきでしょう。それはまさに次の章でやることです。今のところ、ここに記載されているメカニズムの上に構築されたこのようなポリシーが存在することを理解するだけで十分です。

21.5 Page Fault Control Flow

この知識がすべて整ったら、メモリアクセスの完全な制御フローを概略的にスケッチすることができます。言い換えれば、誰かが「プログラムがメモリからデータを取り込むときにどうなるか」と尋ねるとき、あなたはすべての異なる可能性についてかなり良い考えを持っているはずです。詳細は、図 21.2 および図 21.3 の制御フローを参照してください。最初の図は、変換中にハードウェアが何をするかを示し、2 番目はページフォルト時に OS が行うことと示しています。

```

1   VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2   (Success, TlbEntry) = TLB_Lookup(VPN)
3   if (Success == True) // TLB Hit
4       if (CanAccess(TlbEntry.ProtectBits) == True)
5           Offset = VirtualAddress & OFFSET_MASK
6           PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7           Register = AccessMemory(PhysAddr)
8       else
9           RaiseException(PROTECTION_FAULT)
10  else // TLB Miss
11      PTEAddr = PTBR + (VPN * sizeof(PTE))
12      PTE = AccessMemory(PTEAddr)
13      if (PTE.Valid == False)
14          RaiseException(SEGMENTATION_FAULT)
15      else
16          if (CanAccess(PTE.ProtectBits) == False)
17              RaiseException(PROTECTION_FAULT)
18          else if (PTE.Present == True)
19              // assuming hardware-managed TLB
20              TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
21              RetryInstruction()
22          else if (PTE.Present == False)
23              RaiseException(PAGE_FAULT)

```

Figure 21.2: Page-Fault Control Flow Algorithm (Hardware)

図 21.2 のハードウェア制御のフロー図から、TLB ミスが発生したときを理解する重要な 3 つのケースがあることに注意してください。まず、ページが存在し、有効であったこと (18-21 行目)。この場合、TLB ミスハンドラは、単に PTE から PFN を取り込み、命令を再試行し (今度は TLB ヒットを生じる)、前に説明したように (何度も) 続行することができます。2 番目のケース (22-23 行目) では、ページフォルトハンドラを実行する必要があります。これはアクセスするプロセスの正当なページでしたが (結局のところ有効です)、物理メモリには存在しません。3 番目 (最後に)、プログラムのバグなどの理由で、無効なページにアクセスする可能性があります (行 13-14)。この場合、PTE の他のビットは実際には重要ではありません。ハードウェアがこの無効なアクセスをトラップし、OS トラップハンドラが実行され、問題のプロセスが終了する可能性があります。

```

1   PFN = FindFreePhysicalPage()
2   if (PFN == -1) // no free page found
3       PFN = EvictPage() // run replacement algorithm
4   DiskRead(PTE.DiskAddr, pfn) // sleep (waiting for I/O)
5   PTE.present = True // update page table with present
6   PTE.PFN = PFN // bit and translation (PFN)
7   RetryInstruction() // retry instruction

```

Figure 21.3: Page-Fault Control Flow Algorithm (Software)

図 21.3 のソフトウェア制御フローから、ページフォルトを処理するために OS が大まかに何をしなければならないかを見ることができます。第 1 に、OS は、間もなくフォールトインされるページが存在する物理フレームを見つける必要があります。そのようなページがない場合は、置換アルゴリズムが実行され、メモリからいくつかのページが蹴られるまで、メモリを解放するまで待たなければなりません。物理的なフレームがあれば、ハンドラはスワップ領域からページを読み込むための I/O 要求を発行します。最後に、その低速動作が完了すると、OS はページテーブルを更新して命令を再試行します。再試行により TLB ミスが発生し、次に別の再試行時に TLB ヒットが発生し、その時点でハードウェアは所望のアイテムにアクセスすることができる。

21.6 When Replacements Really Occur

これまでのところ、置き換えがどのように行われるかを記述したところでは、OSはメモリが完全にいっぱいになるまで待ってから、他のページのためのスペースを作るためにページを置き換える(追い出す)だけです。しかし、これは少し現実的ではありません。OSがメモリの一部をもっと積極的に解放する理由はたくさんあります。

少量のメモリを確保するために、ほとんどのオペレーティングシステムは、メモリからページを取り出す開始時期を決定するために、ある種の上限値(HW)と下限値(LW)を持っています。これがどのように機能するかは次のとおりです。利用可能なLWページ数が少ないとOSが認識すると、メモリの解放を担当するバックグラウンドスレッドが実行されます。スレッドは、利用可能なHWページがあるところまでページを退去させます。つまり、あるHWで設定している上限値を超えていたり、HWに移動させます。また、あるLWで設定している下限値を下回っているページをLWに移動させます。スワップデーモンやページデーモンと呼ばれることがあるバックグラウンドスレッドは、実行中のプロセスやOSが使用するメモリを解放してくれたことをうかがってスリープ状態になります。

一度に多数の置換を実行することにより、新しいパフォーマンスの最適化が可能になります。たとえば、多くのシステムでは、いくつかのページをクラスタ化またはグループ化し、スワップパーティションに一度に書き出し、ディスクの効率を高めます[LL82]。ディスクをより詳細に説明するときに後で説明するように、このようなクラスタリングはディスクのシークおよびローテーションオーバーヘッドを減らし、パフォーマンスを大幅に向上させます。

バックグラウンドページングスレッドを処理するには、図21.3の制御フローを少し変更が必要があります。直接置換を実行するのではなく、代わりに、使用可能な空きページがあるかどうかを単純にチェックします。そうでない場合は、空きページが必要であることをバックグラウンドページングスレッドに通知します。スレッドがいくつかのページを解放すると、元のスレッドが再び呼び起こされ、それが目的のページにページングされ、その作業に進むことができます。

TIP: DO WORK IN THE BACKGROUND

いくつかの作業がある場合、効率を高め、操作のグループ化を可能にするために、バックグラウンドで実行することをお勧めします。オペレーティングシステムはよくバックグラウンドで動作します。たとえば、実際にデータをディスクに書き込む前に、多くのシステムがファイル書き込みをメモリにバッファリングします。そうすることで多くの利点が得られます。ディスク効率が向上したディスクは一度に多くの書き込みを受け取ることができます。書き込みが決してディスクに行く必要がない場合(すなわち、ファイルが削除された場合)、作業の削減ができる可能性があります。システムがアイドル状態のときにバックグラウンド作業が行われる可能性があるため、ハードウェア[G+95]をより有効に活用できるため、アイドル時間の有効活用が可能になります。

21.7 Summary

この短い章では、システム内に物理的に存在するより多くのメモリにアクセスするという概念を導入しました。これを行うには、ページがメモリー内に存在するかどうかを知らせるために、現在のビット(ある種のもの)を含める必要があります。ページテーブル構造の複雑になります。そうでない場合は、オペレーティングシステムのページフォルトハンドラがページフォールトを処理するために実行され、ディスクからメモリへの望んだページの移動を行います。おそらくメモリ内のいくつかのページを先に交換してすぐに、スワップインするためのスペースを確保します。

重要なことに、これらのアクションはすべてプロセスに対して透過的に行われます。プロセスに関する限り、それは自身のプライベートで連続した仮想メモリにアクセスしているだけです。背後では、ページは物理メモリ内の任意の(連続していない)場所に配置され、時にはメモリにも存在せず、ディスクからのフェッチが必要になることがあります。一般的なケースでは、メモリアクセスが高速であることが重要ですが、場合によってはそれを処理するために複数のディスク操作が必要になることもあります。最悪の場合、1つの命令を実行するように簡単な作業ですが、最悪の場合、その作業が完了するまでに数ミリ秒かかることがあります。

参考文献

[CS94] “Take Our Word For It”

F. Corbato and R. Steinberg

Available: <http://www.takeourword.com/TOW146/page4.html>

Richard Steinberg writes: “Someone has asked me the origin of the word daemon as it applies to computing. Best I can tell based on my research, the word was first used by people on your team at Project MAC using the IBM 7094 in 1963.” Professor Corbato replies: “Our use of the word daemon was inspired by the Maxwell’s daemon of physics and thermodynamics (my background is in physics). Maxwell’s daemon was an imaginary agent which helped sort molecules of different speeds and worked tirelessly in the background. We fancifully began to use the word daemon to describe background processes which worked tirelessly to perform system chores.”

[D97] “Before Memory Was Virtual”

Peter Denning

From In the Beginning: Recollections of Software Pioneers, Wiley, November 1997

An excellent historical piece by one of the pioneers of virtual memory and working sets.

[G+95] “Idleness is not sloth”

Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, John Wilkes

USENIX ATC ’95, New Orleans, Louisiana

A fun and easy-to-read discussion of how idle time can be better used in systems, with lots of good examples.

[LL82] “Virtual Memory Management in the VAX/VMS Operating System”

Hank Levy and P. Lipman

IEEE Computer, Vol. 15, No. 3, March 1982

Not the first place where such clustering was used, but a clear and simple explanation of how such a mechanism works.

22 Beyond Physical Memory: Policies

仮想メモリマネージャでは、空きメモリが大量にある場合は簡単です。ページフォルトが発生した場合、フリーページリストに空きページがあり、それをフォールトページに割り当てます。

ほとんどのメモリが解放されれば、少し面白いことになります。このような場合、このメモリ圧迫により、OSは強制的に使用されるページのためのスペースを作るためにページをページアウトすることを開始します。追放するページ（またはページ）を決定することは、OSの置換ポリシー内にカプセル化されています。歴史的には、古いシステムでは物理メモリがほとんどなかったため、初期の仮想メモリシステムで最も重要な決定の1つでした。最小限にとどめておくと、これはもう少し知つておく価値がある面白いポリシーです。

THE CRUX: HOW TO DECIDE WHICH PAGE TO EVICT

OSは、どのページ（またはページ）をメモリから退去させるかをOSはどのように決定できますか？この決定はシステムの置き換え方針によって行われます。システムの置き換え方針は、一般的な原則（下記参照）に従いますが、偏った場合の振る舞いを避けるための調整も含まれています。

22.1 Cache Management

ポリシーに入る前に、我々がまず解決しようとしている問題についてより詳しく説明します。メインメモリには、システム内のすべてのページのサブセットが格納されているため、システム内の仮想メモリページのキャッシュと見なすことができます。従って、このキャッシュのための置換ポリシーを選ぶことにおける我々の目標は、キャッシュミスの数を最小限にすること、すなわち、ディスクからページをフェッチする回数を最小にすることです。あるいは、キャッシュヒットの数、すなわちアクセスされたページがメモリ内に見つかった回数を最大にするという目標になります。

キャッシュヒット数とミス数を知ることで、プログラムの average memory access time(AMAT) を計算できます（メトリックコンピューターアーキテクトはハードウェアキャッシュを計算します [HP06]）具体的には、これらの値を考慮して、次のようにプログラムの AMAT を計算することができます。

$$AMAT = TM + (PM_{\text{Miss}} \cdot TD) \quad (22.1)$$

ここで、TMはメモリにアクセスするコスト、TDはディスクにアクセスするコスト、PMはキャッシュ内のデータを見つからない確率（ミス）を示します。PMissは0.0から1.0まで変化し、確率（例えば、10%のミス率はPMiss = 0.10を意味する）ではなく、パーセントミス率を参照することもあります。常にメモリ内のデータにアクセスするコストを支払うことに注意してください。さらに、見逃したときには、ディスクからデータを取り出すためのコストをさらに支払わなければなりません。

たとえば、（小さな）アドレス空間を持つマシン（4KB、256バイトページ）を想像してみましょう。したがって、仮想アドレスには、4ビットのVPN（最上位ビット）と8ビットのオフセット（最下位ビット）の2つのコンポーネントがあります。したがって、この例のプロセスは、24または16の合計仮想ページにアクセスできます。この例では、プロセスは、0x000、0x100、0x200、0x300、0x400、0x500、0x600、0x700、0x800、0x900という、次のメモリ参照（すなわち、仮想アドレス）を生成します。これらの仮想アドレスは、アドレス空間の最初の10個のページのそれぞれの最初のバイトを指します（ページ番号は各仮想アドレスの最初の16進数です）

さらに、仮想ページ3を除くすべてのページが既にメモリ内にあると仮定します。したがって、メモリ参照のシーケンスは、ヒット、ヒット、ヒット、ミス、ヒット、ヒット、ヒット、ヒット、ヒットになります。ヒット率（メモリ内の参照の割合）を計算することができます。このとき、90%の参照がメモリに格納さ

れているため、90 %です。従って、ミス率は 10 % ($P_{Miss} = 0.1$) です。一般に、 $P_{Hit} + P_{Miss} = 1.0$; ヒット率 + ミス率合計を 100 %とします。

AMAT を計算するには、メモリにアクセスするコストとディスクにアクセスするコストを知る必要があります。メモリ (TM) にアクセスするコストが約 100 ナノ秒であり、ディスク (TD) にアクセスするコストが約 10 ミリ秒であると仮定すると、 $100\text{ns} + 1\text{ms}$ 、すなわち 1.0001ms である $100\text{ns} + 0.1 \cdot 10\text{ms}$ 、約 1 ミリ秒です。ヒット率が 99.9 % ($P_{miss} = 0.001$) だった場合、AMAT は 10.1 マイクロ秒、つまり約 100 倍高速です。ヒット率が 100 %に近づくと、AMAT は 100 ナノ秒に近づきます。

残念ながら、この例で分かるように、ディスクアクセスのコストは現代のシステムでは非常に高く、わずかなミス率でも実行中のプログラムの AMAT 全体をすぐに支配します。できるだけ多くのミスを避けるか、ディスクの速度でゆっくりと実行する必要があります。これを手助けする 1 つの方法は、現在行っているように、優れたポリシーを慎重に開発することです。

22.2 The Optimal Replacement Policy

特定の置換ポリシーがどのように機能するかをよりよく理解するには、可能な限り最良の置換ポリシーと比較することをお勧めします。結局のところ、このような最適なポリシーは、何年も前に Belady によって開発されました B66 最適な置換ポリシーは、全体として最小のミス数につながります。Belady は、将来最も速くにアクセスされるページを置き換えるシンプルな（しかし、残念なことに実装が難しい）アプローチが、最適なポリシーであり、結果としてキャッシュミスが最小限に抑えられることを示しました。

TIP: COMPARING AGAINST OPTIMAL IS USEFUL

最適な方針はあまり実用的ではありませんが、シミュレーションや他の研究の比較点としては非常に有用です。あなたが何も比較せずに派手な新しいアルゴリズムが 80 %のヒット率を持っているとしても意味がないということです。最適は 82 %のヒット率を達成すると言います（あなたの新しいアプローチは最適に非常に近いです）そのため、最適なものが何であるかを知ることで、より良い比較を実行し、改善の可能性がどれくらいあるのか、理想的なポリシーの改善に近づくことができます [AD03]。

うまくいけば、最適なポリシーの背後にあるものは理にかなっています。それについて考えてみましょう。あなたがいくつかのページを投げ捨てなければならない場合、最もアクセスする将来が遠いページを投げ捨てませんか？ そうすることで、本質的に、キャッシュ内の他のすべてのページが、最も遠いページよりも重要であるということになります。これが当てはまる理由は簡単です。最も遠いものを参照する前に、他のページを参照します。

最適なポリシーがもたらす決定を理解するための簡単な例をたどってみましょう。プログラムは、 $0,1,2,0,1,3,0,3,1,2,1$ の仮想ページの次のストリームにアクセスすると仮定します。図 22.1 は、3 ページに収まるキャッシュを想定した最適な動作を示しています。

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	3	0, 1, 2
1	Hit		0, 1, 2

Figure 22.1: Tracing The Optimal Policy

この図では、次の操作を確認できます。最初の 3 回のアクセスは、キャッシュが空の状態で開始するため、ミスをしています。このようなミスはコールドスタートミス（または強制ミス）と呼ばれることがあります。次に、キャッシュ内でヒットしたページ 0 と 1 を再度参照します。最後に、別のミス（3 ページ目）に達しますが、今回はキャッシュがいっぱいです。つまり、交換が行われなければいけません。どちらのページを置き換えなければならないのか疑問に思うでしょう。最適なポリシーでは、現在キャッシュ内にある各ページ（0,1,2）を調べ、0 がほぼ同時にアクセスされ、1 が少し後にアクセスされ、2 が将来最も後にアクセスされることを確認します。つまり、ページ 2 を退かせて、キャッシュ内のページ 0,1,3 を生成します。次の 3 つの参考文献はヒットですが、退去したページ 2 のミスで苦しんでいます。ここで、最適なポリシーは、キャッシュ内の各ページ（0,1、および 3）を調べ、ページ 1（アクセスしようとしている）を追い出さない限り、問題ありません。この例では、ページ 3 が削除されていることが示されていますが、0 でも問題ありません。最後に、1 ページ目でヒットし、トレースが完了しました。

ASIDE: TYPES OF CACHE MISSES

コンピューターアーキテクチャーの世界では、設計者はタイプごとにミスを 3 つのカテゴリの 1 つに特徴付けることが有用であることを示しています。compulsory、capacity、conflict のスリー C [H87] と呼ばれることがあります。一つ目の強制的なミス（またはコールドスタートミス [EF78]）が発生するのは、キャッシュが最初から空であり、これがアイテムへの最初の参照であるためです。二つ目の容量不足は、キャッシュの容量が足りなくなり、新しいアイテムをキャッシュに持ち込むためにアイテムを追い出す必要があるために発生します。三つ目の競合ミスは、ハードウェアキャッシュ内にアイテムを置くことができる限界があるため、ハードウェアで発生します。そのよ

うなキャッシュは常に完全に連想的です。つまり、メモリ内のどこにページを置くことができるかに制限がないので、OS ページ・キャッシュ内では発生しません。詳細は H & P を参照してください [HP06]。

キャッシュのヒット率も計算できます。ヒット率は 6 ヒットと 5 ヒットで、ヒット率は $(\text{ヒット}) / (\text{ヒット} + \text{ミス})$ で、 $(6) / (6 + 5)$ または 54.5 % です。ヒット率を法とする強制ミスを計算することもできます(つまり、特定のページへの最初のミスを無視する)。その結果、ヒット率は 85.7 % になります。

残念ながら、以前にスケジューリング方針の策定において見た時と同じように、ページの将来は一般的にわかりません。汎用オペレーティングシステムの最適なポリシーを構築することはできません。したがって、実際の展開可能なポリシーを開発する際に、どのページを退去させるかを決める他の方法を見つけるアプローチに焦点を当てます。

22.3 A Simple Policy: FIFO

多くの初期のシステムでは、最適かつ雇用された非常に単純な代替ポリシーに近づくことの複雑さが回避されました。たとえば、一部のシステムでは、FIFO(先入れ先出し)置換が使用されました。ここでは、ページはシステムに入るときに単にキューに入れられます。置換が行われると、キューの末尾のページ(「ファーストイン」ページ)が追い出されます。FIFO には大きな強みがあります。実装が非常に簡単です。

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		First-in → 0
1	Miss		First-in → 0, 1
2	Miss		First-in → 0, 1, 2
0	Hit		First-in → 0, 1, 2
1	Hit		First-in → 0, 1, 2
3	Miss	0	First-in → 1, 2, 3
0	Miss	1	First-in → 2, 3, 0
3	Hit		First-in → 2, 3, 0
1	Miss	2	First-in → 3, 0, 1
2	Miss	3	First-in → 0, 1, 2
1	Hit		First-in → 0, 1, 2

Figure 22.2: Tracing The FIFO Policy

サンプル参照ストリームで FIFO がどのように動作するかを調べてみましょう(図 22.2)。ページ 0、1、2 への 3 回の強制ミスでトレースを開始し、0 と 1 の両方でヒットします。次に、ページ 3 が参照され、ミスが発生します。置換の決定は FIFO で簡単です。「最初のもの」であったページを選択します(図のキャッシュ状態は FIFO 順で、最初のページは左にあります)。これはページ 0 です。残念ながら、次のアクセスはページ 0 へのものであり、別のミスと置換(ページ 1 の)が発生します。その後、3 ページ目にヒットしましたが、1 と 2 でミスして、最終的に 3 になりました。

FIFO を最適値と比較すると、FIFO は著しく悪化します。つまり、ヒット率は 36.4 % (強制ミスを除くと 57.1 %) です。FIFO は単にブロックの重要性を判断することはできません。たとえページ 0 が何度もアクセスされたとしても、FIFO はメモリに取り込まれた最初のものだったからです。

ASIDE: BELADY'S ANOMALY

Belady(最適政策の)と同僚たちは、予想外に行動した興味深い参照ストリームを見つけました [BNS69]。メモリ参照ストリームが 1,2,3,4,1,2,5,1,2,3,4,5 だったとしましょう。彼らが研究していたのは、キャッシング・サイズが 3 から 4 ページに変更されたときに、キャッシング・ヒット率がどのように変化したかです。一般に、キャッシングが大きくなると、キャッシング・ヒット率が向上する(向上する)ことが期待されます。しかし、この場合、FIFO では、悪化します！この行動は、一般的に Belady's Anomaly(彼の共著者の賛辞)と呼ばれています。

LRU などの他のポリシーは、この問題を抱えていません。なぜでしょうか？結論として、LRU にはスタックプロパティ [M+70] があります。このプロパティを持つアルゴリズムの場合、サイズ $N + 1$ のキャッシングには当然サイズ N のキャッシングの内容が含まれます。したがって、キャッシングサイズを増やすと、ヒット率は変わらないか向上します。FIFO とランダム(とりわけ)は明らかにスタックのプロパティに従わず、したがって異常な動作の影響を受けやすいです。

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	0	1, 2, 3
0	Miss	1	2, 3, 0
3	Hit		2, 3, 0
1	Miss	3	2, 0, 1
2	Hit		2, 0, 1
1	Hit		2, 0, 1

Figure 22.3: Tracing The Random Policy

22.4 Another Simple Policy: Random

もう 1 つの同様の置換ポリシーは Random です。これはメモリ不足のもとで置換するランダムなページを選択するだけです。ランダムは FIFO に似た性質を持っています。実装するのは簡単ですが、どのブロックを取り除くことを考えると最適ではありません。私たちの有名な例のリファレンストリームで Random がど

うなるかを見てみましょう（図 22.3 を参照）

もちろん、ランダムはどのように幸運な（または不運な）ランダムがその選択肢に入るかに完全に依存します。上記の例では、Random は FIFO より少し良く、最適より少し劣っています。実際には、無作為実験を何千回も実行し、それがどのように一般的に行うかを決定することができます。図 22.4 は、無作為のシード値が異なる 10,000 件の試行に対して、無作為に達成したヒット数を示しています。あなたが見ることができるよう、時には（時間のわずか 40 % を過ぎて）、ランダムは最適なほど良く、サンプルのトレースで 6 ヒットを達成します。時にはそれは 2 ヒット以下を達成するなど、さらに悪化する場合もあります。ランダムは抽選の運勢によって決まります。

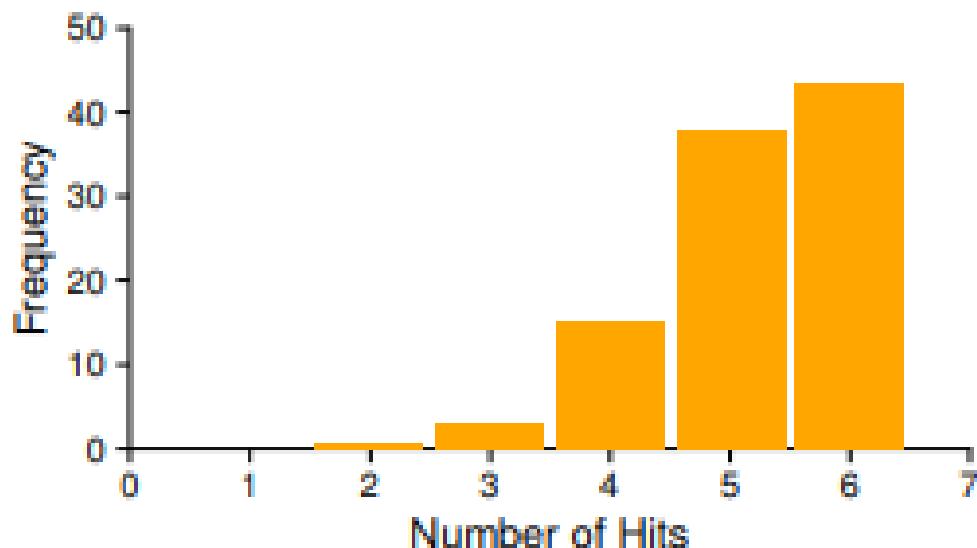


Figure 22.4: Random Performance Over 10,000 Trials

22.5 Using History: LRU

残念なことに、FIFO やランダムなどの単純なポリシーは共通の問題を抱えている可能性があります。重要なページが再度参照される可能性があります。FIFO は、最初に持ち込まれたページをキックアウトします。これが重要なコードやデータ構造を持つページである場合、そのページはまもなくページングされますが、追い出されます。したがって、FIFO、ランダムなどのポリシーは最適に近づきそうにありません。よりスマートなものが必要です。

スケジューリング方針で行ったように、将来の推測を改善するために、履歴を見てみましょう。たとえば、プログラムが近い過去にページにアクセスした場合、近い将来にもう一度そのページにアクセスする可能性があります。

ページ置換ポリシーが使用できる履歴情報の 1 つのタイプは頻度です。ページが何度もアクセスされている場合は、明らかに何らかの価値があるので、置き換えてはいけません。もう一つは、アクセスの最新性です。より最近ページにアクセスした場合、おそらくそれが再びアクセスされる可能性が高くなります。

この一連のポリシーは、人々が地域の原則 [D70] として言及しているものに基づいており、基本的にはプログラムとその行動についての単なる見解です。この原理は、プログラムがあるコードシーケンス（例えば、ループ内）およびデータ構造（例えば、ループによってアクセスされる配列）にかなり頻繁にアクセスする傾向があることを簡単に示しています。したがって、どのページが重要であるかを把握するために履歴を使用して、そ

のページを追い出し時にメモリに保存しておく必要があります。

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		LRU → 0
1	Miss		LRU → 0, 1
2	Miss		LRU → 0, 1, 2
0	Hit		LRU → 1, 2, 0
1	Hit		LRU → 2, 0, 1
3	Miss	2	LRU → 0, 1, 3
0	Hit		LRU → 1, 3, 0
3	Hit		LRU → 1, 0, 3
1	Hit		LRU → 0, 3, 1
2	Miss	0	LRU → 3, 1, 2
1	Hit		LRU → 3, 2, 1

Figure 22.5: Tracing The LRU Policy

そして、歴史的に単純なアルゴリズムのファミリーが生まれました。最小使用頻度の高い (LFU) ポリシーは、退去が発生しなければならないときに最も頻繁に使用されないページを置き換えます。同様に、LRU(Least Recently Used) ポリシーは、最も最近使用されたページを置き換えます。これらのアルゴリズムは名前を知ると、それが何をするのか正確に分かります。これは名前にとて優れた特性です。LRU をよりよく理解するために、LRU がサンプルの参照ストリームでどのように動作するかを調べてみましょう。図 22.5 に結果を示します。この図から、LRU がランダムまたは FIFO などの履歴のような状態がないポリシーよりも優れた処理を行うために、履歴をどのように使用できるかがわかります。この例では、0 と 1 が最近アクセスされたため、LRU は最初にページを置換する必要があるときにページ 2 を退去させます。1 と 3 が最近アクセスされたため、ページ 0 が置き換えられます。どちらの場合も、履歴に基づく LRU の決定は正しいと判明し、次の参照はヒットします。したがって、単純な例では、LRU はパフォーマンスを最適にするためにできるだけ多くのことを行います。我々はまた、これらのアルゴリズムの対立するものが存在します。それは、Most Frequently Used(MFU) および Most Recently Used(MRU) です。ほとんどの場合 (すべてではありません)、これらのポリシーは、ほとんどのプログラムがそれを採用するのではなく局所性 (キャッシュの状態) を無視するため、うまく機能しません。

ASIDE: TYPES OF LOCALITY

プログラムが出現する傾向がある地域には 2 つのタイプがあります。一つは空間的局所性 (spatial locality) として知られており、ページ P がアクセスされた場合、その周辺のページ (例えば、P-1 または P + 1) もアクセスされる可能性が高いです。二つ目は、時間的局所性です。アクセスされたページは、近い将来再びアクセスされる可能性があります。このようなローカル性が存在すると仮定すると、ハードウェアシステムのキャッシュ階層で大きな役割を果たします。命令、データ、アドレス変換キャッシュのレベルをさまざまに配備して、局所性が存在する場合にプログラムを高

速に実行できます。もちろん、局所性の原則は、すべてのプログラムが従わなければならない厳しい規則ではありません。実際、一部のプログラムは、メモリ（またはディスク）にむしろランダムにアクセスするため、アクセスストリームに少しも局所性がありません。したがって、あらゆる種類のキャッシュ（ハードウェアまたはソフトウェア）を設計する際に局所性を覚えておくことは良いことですが、成功を保証するものではありません。むしろ、それはコンピュータシステムの設計において有用であることがよく証明されるのがヒューリスティックです。

22.6 Workload Examples

これらのポリシーのいくつかの動作をよりよく理解するために、例を見てみましょう。ここでは、小さなトレースではなく、より複雑な仕事量を調べます。しかし、これらの仕事量さえも大幅に単純化されます。より良い研究にはアプリケーショントレースが含まれます。

私たちの最初の仕事量には局所性がありません。つまり、各参照はアクセスされたページのセット内のランダムなページになります。この単純な例では、仕事量は時間の経過とともに 100 のユニークなページにアクセスし、ランダムに参照する次のページを選択します。全体的に 10,000 ページがアクセスされます。実験では、各ポリシーがどのキャッシュサイズの範囲でどのように動作するかを確認するために、キャッシュサイズを非常に小さい（1 ページ）からすべての固有ページ（100 ページ）を保持するのに十分に変更しています。

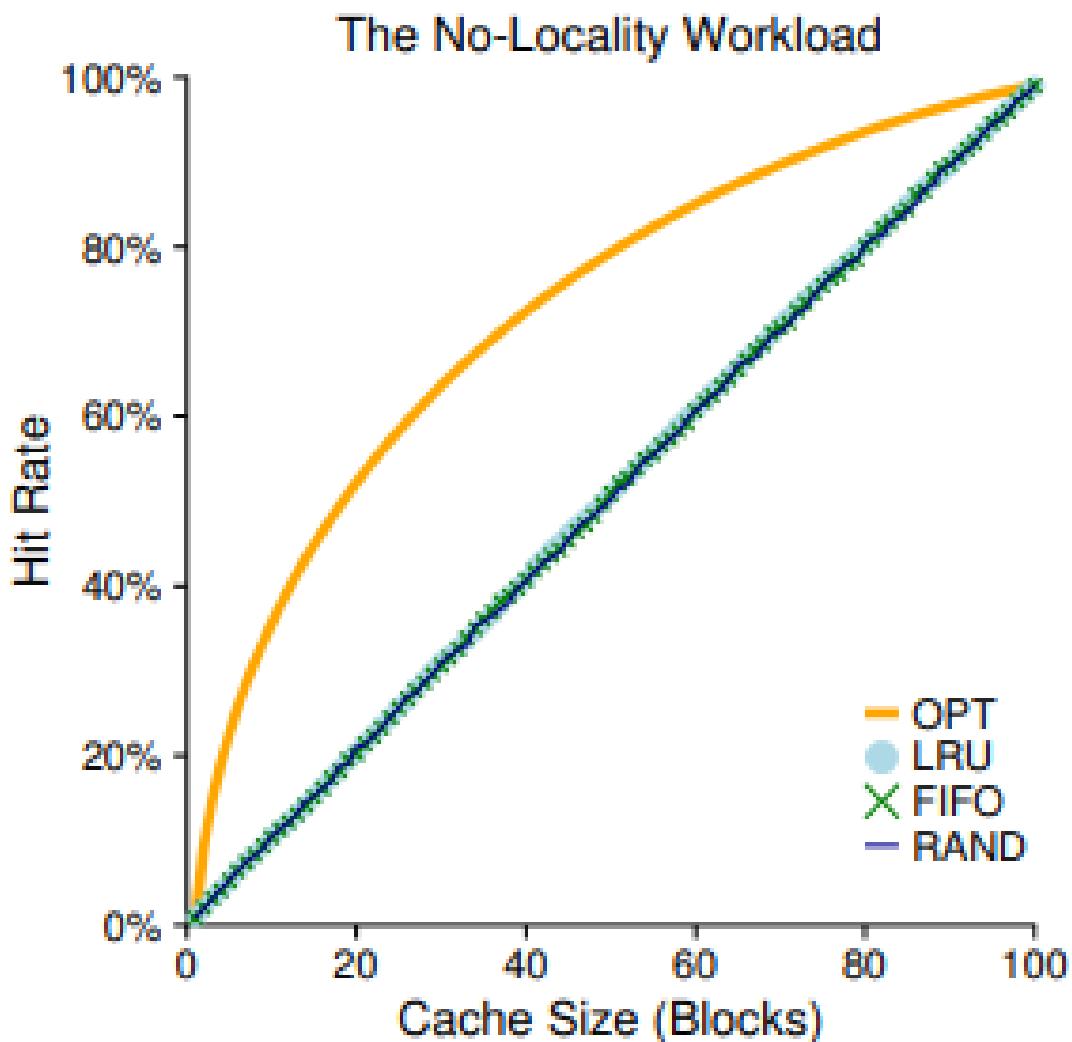


Figure 22.6: The No-Locality Workload

図 22.6 は、最適、LRU、ランダム、および FIFO の実験結果をプロットしたものです。図の y 軸は、各ポリシーが達成するヒット率を示しています。x 軸はキャッシュサイズを変更します。

グラフからいくつかの結論を導くことができます。まず、仕事量に局所性がない場合、どの現実的なポリシーを使用しているかは重要ではありません。LRU、FIFO、およびランダムはすべて、キャッシュのサイズによって正確に決定されるヒット率で同じ処理を行います。第 2 に、キャッシュが仕事量全体に適合するよう十分な大きさであれば、どのポリシーを使用するかは重要ではありません。参照されたすべてのブロックがキャッシュに取ると、すべてのポリシー（ランダム除く）は 100 % のヒット率に収束します。最後に、最適化が現実的なポリシーよりも顕著に優れていることがわかります。可能であれば、将来を見て、より良い仕事を置き換えます。

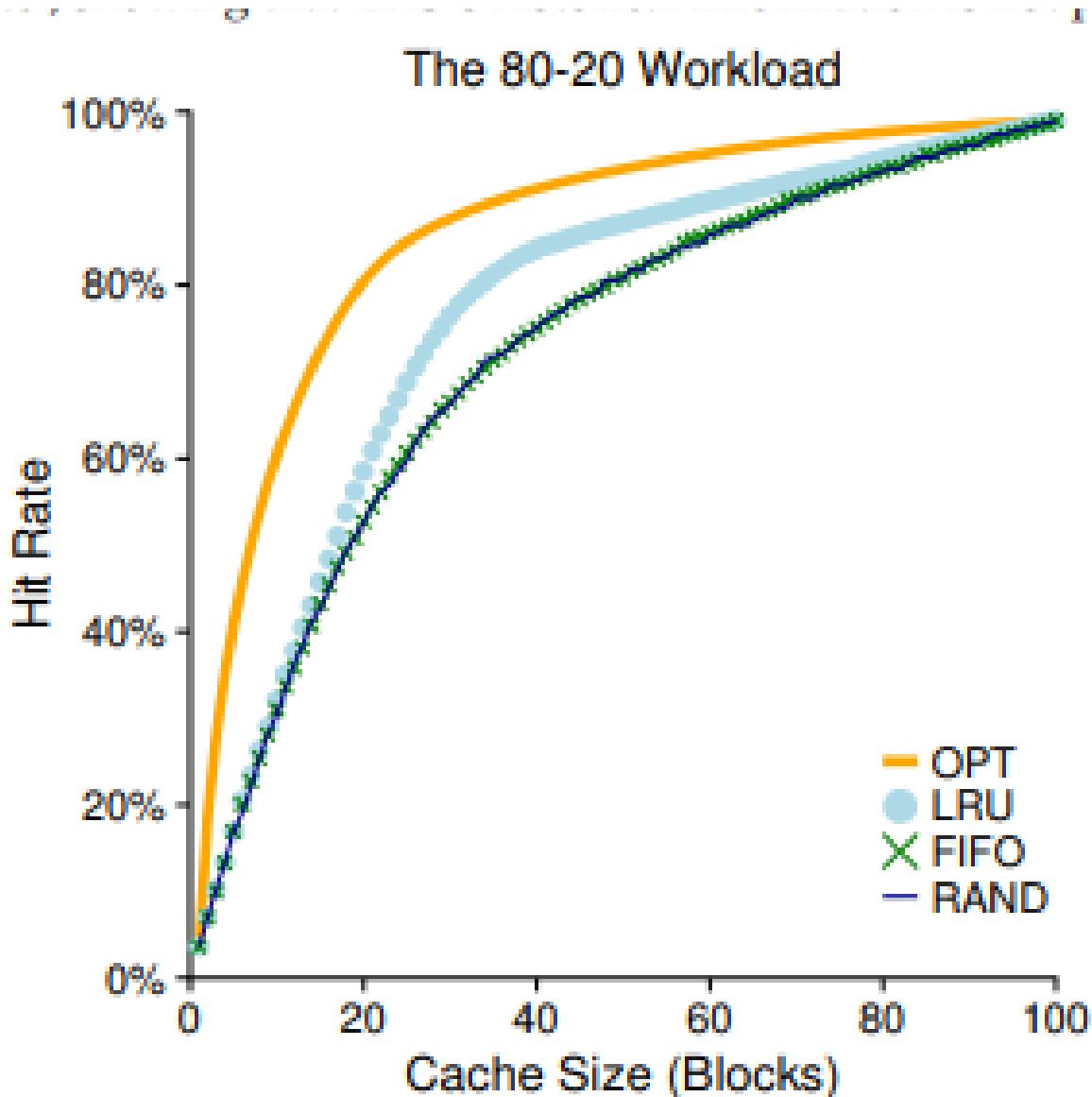


Figure 22.7: The 80-20 Workload

次の仕事量は「80-20」仕事量と呼ばれ、局所性を示します。参照の 80 % はページの 20 % (「ホット」ページ) に作成されます。参照の 20 % はページの 80 % (「コールド」ページ) に作成されます。私たちの仕事量には、ユニークなページが合計 100 個あります。したがって、「ホット」ページはほとんどの時間に参照され、「コールド」ページは残りのページに参照されます。図 22.7 は、この仕事量でポリシーがどのように機能するかを示しています。図からわかるように、ランダムと FIFO の両方が合理的にうまくいく一方で、LRU はホットなページを保持する可能性が高いため、より良い結果を示します。それらのページは過去に頻繁に参照されているため、近い将来再び参照される可能性があります。LRU の履歴情報が完璧ではないことを示しています。

ここで疑問に思うかもしれません。ランダムと FIFO と LRU は本当にそれほど大きなトレードオフでしょうか？ 答えはいつものように「それは依存している」です。ミスが非常にコストかかる場合、ヒット率のわずかな増加（ミス率の低下）でさえパフォーマンスに大きな違いをもたらす可能性があります。ミスがそれほどコストがかからない場合、もちろん LRU のメリットはそれほどありません。

最終的な仕事量を見てみましょう。これを「順序ループ」仕事量と呼びます。これは、50 ページを順番に参照していきます。つまり、0 から 1、...、49 ページまで順番に参照します。ループを繰り返して 50 ページ

へ合計 10,000 回のアクセスをします。図 22.8 の最後のグラフは、この仕事量でのポリシーの動作を示しています。

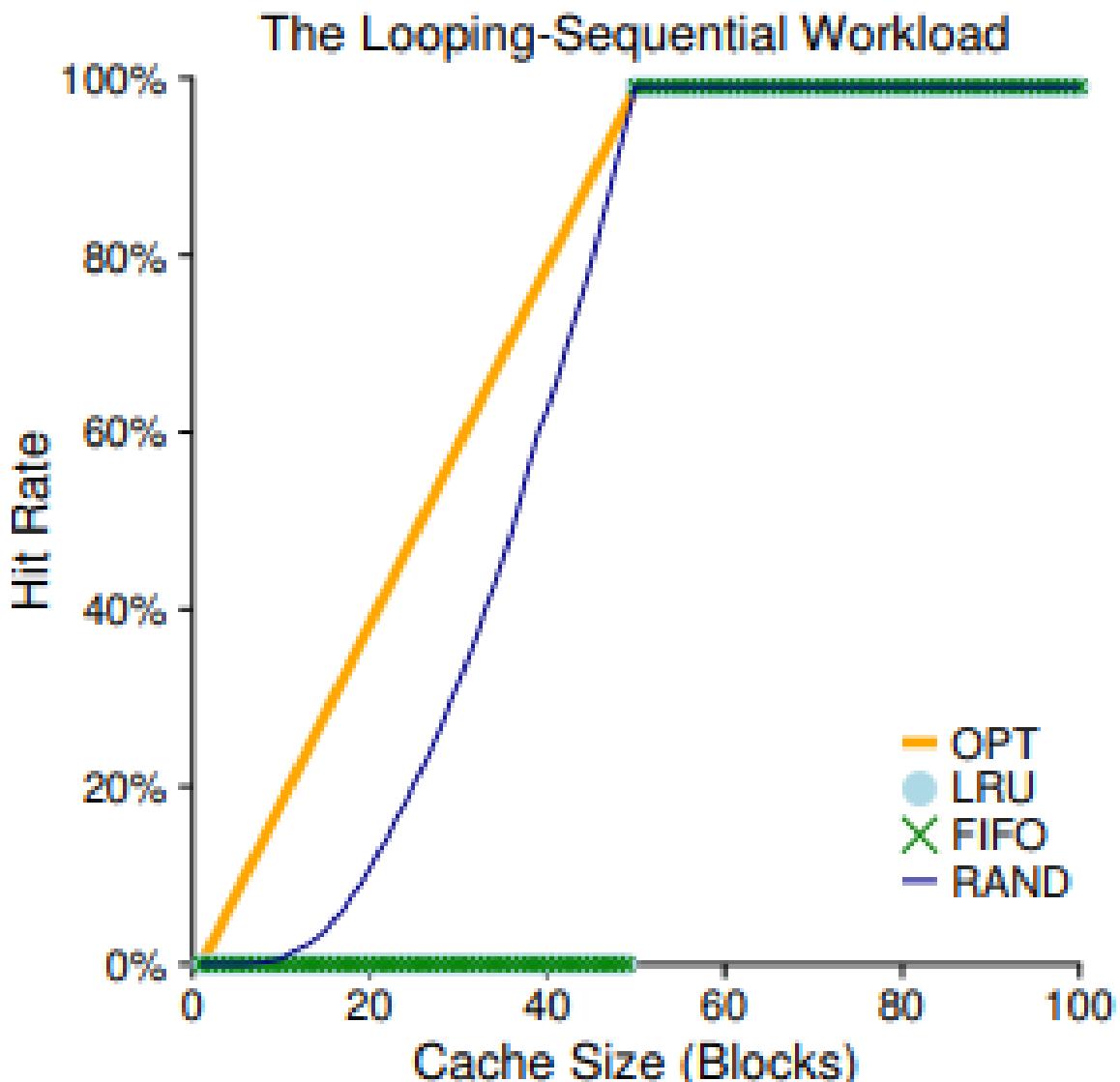


Figure 22.8: The Looping Workload

この仕事量は、多くのアプリケーション（データベース [CD85] などの重要な商用アプリケーションを含む）で一般的ですが、LRU と FIFO の両方で最悪のケースです。これらのアルゴリズムは、古いページを追い出します。そのため、仕事量がループする性質のため、これらの古いページは、将来使われるとしてもポリシーがキャッシュに保持しません。実際、サイズ 49 のキャッシュを使用しても、50 ページのループ順の仕事量ではヒット率は 0 %になります。興味深いことに、ランダムなポリシーは著しく優れており、最適に近づいていませんが、少なくともゼロ以外のヒット率を達成しています。ランダムには素晴らしい性質があることがわかります。

22.7 Implementing Historical Algorithms

ご覧のように、LRUなどのアルゴリズムは、一般的に、FIFOやランダムなどの単純なポリシーよりも優れた処理を行いますが、重要なページを捨てる可能性があります。残念なことに、履歴のポリシーは私たちに新たな挑戦を提示します。

たとえば、LRUを取ってみましょう。完全に実装するには、多くの作業が必要です。具体的には、各ページアクセス(すなわち、各メモリアクセス、命令フェッチまたはロードまたはストア)に応じて、このページをリストの前部(すなわち、MRU側)に移動させるためにいくつかのデータ構造を更新しなければいけません。これをFIFOに対比すると、ページのFIFOリストは、ページが取り除かれたとき(最初のページを取り除くことによって)にアクセスされるとき、または新しいページがリストに追加されたとき(最後の側に)どのページが最も最近に使用されたのかを把握するために、システムはすべてのメモリ参照に対していくつかのアカウンティング作業を行う必要があります。明らかに、細心の注意を払うことがありません。しかし、そのような一連の処理はパフォーマンスを大幅に低下させる可能があります。

これをスピードアップするのに役立つ方法の1つは、ハードウェアのサポートを少し追加することです。例えば、マシンは、各ページのアクセス時にメモリ内のtime fieldsを更新することができます(例えば、これはプロセス毎のページテーブル内にあってもよいし、メモリ内の別個の配列内にあってもよく、システムの物理ページ毎に1エントリ)。つまり、ページがアクセスされるとき、time fieldsはハードウェアによって現在の時間に設定されます。次に、ページを置換するとき、OSはシステム内のすべてのtime fieldsを単に走査して、最も最近に使用されたページを見つけることができます。

残念なことに、システム内のページ数が増えるにつれて、最も最近使用されていないページを見つけるために膨大な数のtime fieldsをスキャンするのは非常に高価です。4GBのメモリを搭載した最新のマシンを4KBのページに分けたと想像してください。このマシンには100万ページがあるため、最新のCPU速度であっても、LRUページの検索には長い時間がかかります。本当に交換する最も古いページを見つける必要があるのでしょうか?

CRUX: HOW TO IMPLEMENT AN LRU REPLACEMENT POLICY

完璧なLRUを実装するのにはコストがかかることを考へるのであれば、何らかの方法で近似させることができますか?

22.8 Approximating LRU

結論としては、答えは「はい」です。計算上のオーバーヘッドの観点から、LRUを近似する方がより現実的であり、現代の多くのシステムではそうです。アイデアは、使用ビット(リファレンスビットと呼ばれることもあります)の形でハードウェアサポートを必要とします。最初のものは、ページング付きの最初のシステムで実装されたアトラス onelevel store [KE+62]です。システムの1ページあたり1ビットの使用ビットがあり、その使用ビットはどこかのメモリに存在します(たとえば、プロセスごとのページテーブル内、または配列のどこかにある可能性があります)。ページが参照される(すなわち、読み書きされる)ときはいつも、使用ビットはハードウェアによって1にセットされます。ハードウェアはビットを決してクリアしません(すなわち0にセットする行為)それはクリアを行うのはOSの責任です。

OSはLRUを近似するために使用ビットをどのように使用しますか?まあ、たくさんある方法があるかもしれません、clock algorithm[C69]では単純なapproachが1つ提案されました。システムのすべてのページが循環リストに配置されているとします。時計の針は、最初にいくつかの特定のページを指しています(本当に問題ありません)。置換が行われなければならない場合、OSは、現在指示されたページPに1または0の使用ビットがあるかどうかをチェックします。1ならば、これはページPが最近使用されたことを意味し、し

たがって置換のための良好な候補ではありません。したがって、 P の使用ビットは 0(クリア) に設定され、クロック・ハンドは次のページ ($P + 1$) にインクリメントされます。アルゴリズムは、このページが最近使用されていないことを意味する 0 に設定された使用ビットが見つかるまで続きます (または、最悪の場合、すべてのページの検索を終了しすべてのクリアビットになっている)

このアプローチは、LRU を近似するために使用ビットを使用する唯一の方法ではないことに注意してください。実際には、使用ビットを定期的にクリアして、どちらのページが 1 と 0 の使用ビットを持っているかを区別して、どちらを置き換えるかを決めるアプローチは問題ありません。Corbato のクロックアルゴリズムは、成功を収めた初期のアプローチの 1 つで、未使用的ページを探しているすべてのメモリを繰り返しスキャンしないという素晴らしい特性を持っていました。

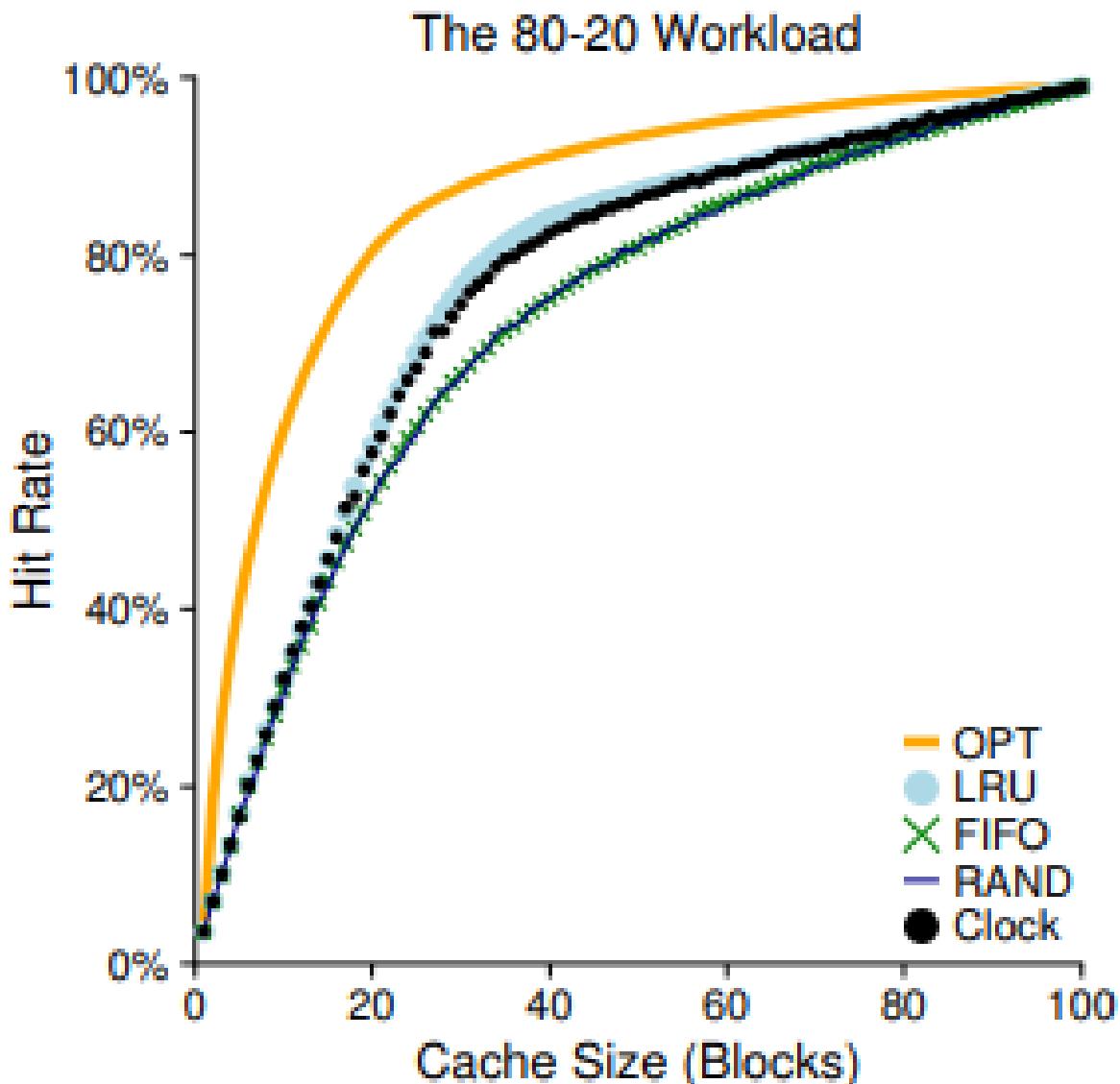


Figure 22.9: The 80-20 Workload With Clock

図 22.9 にクロックアルゴリズムの変形例の動作を示します。この変形は、置換を行うときにランダムにページをスキャンします。基準ビットが 1 にセットされたページに遭遇すると、ビットをクリアする (すなわち、それを 0 にセットする)。参照ビットが 0 に設定されたページが検出されると、参照ビットがその犠牲者として選択されます。ご覧のように、完璧な LRU とは言えませんが、履歴を全く考慮していないアプローチよりも優れています。

22.9 Considering Dirty Pages

一般的に行われているクロックアルゴリズム (Corbato [C69] が最初に提案したもの) を少し変更したのは、メモリ内でページが変更されたかどうかをさらに考慮することです。この理由は、ページが変更されて汚れている場合、ページを追い出すためにディスクに書き戻さなければならず、これは高価です。変更されていない (したがってクリーンな) 場合は、削除は容易です。物理的なフレームは、追加の I/O なしで他の目的のために単純に再利用することができます。したがって、一部の VM システムでは、ダーティページでクリーンページを削除することができます。

この動作をサポートするために、ハードウェアは変更されたビット (a.k.a. ダーティビット) を含むべきである。このビットは、ページが書き込まれるたびに設定されるため、ページ置換アルゴリズムに組み込むことができます。たとえば、クロックアルゴリズムを変更して、使用されていないページとクリーンなページの両方をスキャンして最初に削除することができます。それらを見つけることができず、次いで、未使用のページが汚れているかどうか、等々です。

22.10 Other VM Policies

ページ置換は、VM サブシステムが採用している唯一のポリシーではありません (ただし、最も重要です)。例えば、OS はページをメモリにいつ持ち込むかを決定しなければいけません。このポリシーは (Denning [D70] によって呼び出されたように) ページ選択ポリシーと呼ばれることもありますが、OS にはいくつかのオプションがあります。

ほとんどのページでは、OS は単純にデマンドページングを使用します。つまり、OS はページがアクセスされたときにメモリを「オンデマンドで」オンにします。もちろん、OS はページが使用されようとしていることを推測することができます。この動作はプリフェッチと呼ばれ、合理的な成功の可能性がある場合にのみ実行する必要があります。例えば、あるシステムは、コードページ P がメモリに持ち込まれると、そのコードページ P +1 がまもなくアクセスされる可能性が高いので、メモリに持ち込むべきであると仮定します。

別のポリシーは、OS がどのようにページをディスクに書き出すかを決定します。もちろん、それらは一度に 1 つずつ書き出すことができます。しかし、多くのシステムでは、多数のペンドイグ書き込みをメモリにまとめて 1 つの (より効率的な) 書き込みでディスクに書き込みます。この動作は、通常、クラスタリングと呼ばれ、単純に書き込みのグループ化と呼ばれ、多数の小さなものよりも効率的に单一の大きな書き込みを実行するディスクドライブの性質のために有効です。

22.11 Thrashing

閉鎖する前に、最終的な質問に答えます。メモリが単純に過多になったときに OS が行うべきことは、実行中のプロセスのメモリ要求が単に利用可能な物理メモリを上回るだけですか？ この場合、システムは絶えずページングを行い、時にはスラッシング [D70] と呼ばれる状態になります。

以前のオペレーティングシステムの中には、発生時にスラッシングを検出し対処するためのかなり洗練されたメカニズムがありました。例えば、一連のプロセスがある場合、システムは、プロセスの作業セット (積極的に使用しているページ) の縮小されたセットがメモリに収まり、改善されることを期待して、プロセスのサブセットを実行しないことを決定できます。このアプローチは、一般にアドミッションコントロールとして知られていますが、現実の生活だけでなく現代のコンピュータシステム (悲しいことに) で頻繁に遭遇する状況を、一度にすべてうまくやろうとするよりも、うまく動作しない方が良いと述べています。

現在のシステムの中には、メモリ過負荷に対するより厳しいアプローチをとっているものがあります。たとえば、Linux のバージョンによっては、メモリが過剰登録されたときにメモリ不足のキラー (OOM killer) を

実行するものがあります。このデーモンは大量のメモリを必要とするプロセスを選択して終了させるので、メモリーをあまりにも微妙な方法で減らすことができます。メモリ使用量を削減するのに成功しているが、このアプローチは、例えば X サーバを殺して、ディスプレイを必要とするアプリケーションを使用できなくすると、問題を引き起こす可能性があります。

22.12 Summary

我々は、すべての最新のオペレーティングシステムの VM サブシステムの一部であるいくつかのページ置換(およびその他の)ポリシーの導入を見てきました。現代のシステムでは、時計のような簡単な LRU 近似にいくつかの微調整が追加されています。例えば、走査抵抗は、ARC [MM03] のような多くの最近のアルゴリズムの重要な部分です。スキヤン耐性アルゴリズムは通常 LRU のようなものですが、LRU のワーストケースの動作を回避しようとしています。LRU はループ順の仕事量で見ました。したがって、ページ置換アルゴリズムの進化が続いていきます。

しかし、多くの場合、メモリアクセスとディスクアクセス時間との間の相違が増大するにつれて、前記アルゴリズムの重要性が減少しています。ディスクへのページングは非常に高価なので、頻繁なページングのコストは非常に高くなります。したがって、過度のページングに対する最善の解決策は、よく単純(知的に不満な場合)です。

参考文献

- [AD03] “Run-Time Adaptation in River”
Remzi H. Arpacı-Dusseau
ACM TOCS, 21:1, February 2003
A summary of one of the authors’ dissertation work on a system named River. Certainly one place where he learned that comparison against the ideal is an important technique for system designers.
- [B66] “A Study of Replacement Algorithms for Virtual-Storage Computer”
Laszlo A. Belady
IBM Systems Journal 5(2): 78-101, 1966
The paper that introduces the simple way to compute the optimal behavior of a policy (the MIN algorithm).
- [BNS69] “An Anomaly in Space-time Characteristics of Certain Programs Running in a Paging Machine”
L. A. Belady and R. A. Nelson and G. S. Shedler
Communications of the ACM, 12:6, June 1969
Introduction of the little sequence of memory references known as Belady’s Anomaly. How do Nelson and Shedler feel about this name, we wonder?
- [CD85] “An Evaluation of Buffer Management Strategies for Relational Database Systems”
Hong-Tai Chou and David J. DeWitt
VLDB ’85, Stockholm, Sweden, August 1985
A famous database paper on the different buffering strategies you should use under a number of common database access patterns. The more general lesson: if you know something about a workload, you can tailor policies to do better than the general-purpose ones usually found in the OS.
- [C69] “A Paging Experiment with the Multics System”
F.J. Corbató

Included in a Festschrift published in honor of Prof. P.M. Morse

MIT Press, Cambridge, MA, 1969

The original (and hard to find!) reference to the clock algorithm, though not the first usage of a use bit.

Thanks to H. Balakrishnan of MIT for digging up this paper for us.

[D70] “Virtual Memory”

Peter J. Denning

Computing Surveys, Vol. 2, No. 3, September 1970

Denning’s early and famous survey on virtual memory systems.

[EF78] “Cold-start vs. Warm-start Miss Ratios”

Malcolm C. Easton and Ronald Fagin

Communications of the ACM, 21:10, October 1978

A good discussion of cold-start vs. warm-start misses.

[FP89] “Electrochemically Induced Nuclear Fusion of Deuterium”

Martin Fleischmann and Stanley Pons

Journal of Electroanalytical Chemistry, Volume 26, Number 2, Part 1, April, 1989

The famous paper that would have revolutionized the world in providing an easy way to generate nearly infinite power from jars of water with a little metal in them. Unfortunately, the results published (and widely publicized) by Pons and Fleischmann turned out to be impossible to reproduce, and thus these two well-meaning scientists were discredited (and certainly, mocked). The only guy really happy about this result was Marvin Hawkins, whose name was left off this paper even though he participated in the work; he thus avoided having his name associated with one of the biggest scientific goofs of the 20th century.

[HP06] “Computer Architecture: A Quantitative Approach”

John Hennessy and David Patterson

Morgan-Kaufmann, 2006

A great and marvelous book about computer architecture. Read it!

[H87] “Aspects of Cache Memory and Instruction Buffer Performance”

Mark D. Hill

Ph.D. Dissertation, U.C. Berkeley, 1987

Mark Hill, in his dissertation work, introduced the Three C’s, which later gained wide popularity with its inclusion in H&P [HP06]. The quote from therein: “I have found it useful to partition misses . . . into three components intuitively based on the cause of the misses (page 49).”

[KE+62] “One-level Storage System”

T. Kilburn, and D.B.G. Edwards and M.J. Lanigan and F.H. Sumner

IRE Trans. EC-11:2, 1962

Although Atlas had a use bit, it only had a very small number of pages, and thus the scanning of the use bits in large memories was not a problem the authors solved.

[M+70] “Evaluation Techniques for Storage Hierarchies”

R. L. Mattson, J. Gecsei, D. R. Slutz, I. L. Traiger

IBM Systems Journal, Volume 9:2, 1970

A paper that is mostly about how to simulate cache hierarchies efficiently; certainly a classic in that regard, as well for its excellent discussion of some of the properties of various replacement algorithms. Can you figure out why the stack property might be useful for simulating a lot of different-sized caches at once?

[MM03] “ARC: A Self-Tuning, Low Overhead Replacement Cache”

Nimrod Megiddo and Dharmendra S. Modha

FAST 2003, February 2003, San Jose, California

An excellent modern paper about replacement algorithms, which includes a new policy, ARC, that is now used in some systems. Recognized in 2014 as a “Test of Time” award winner by the storage systems community at the FAST ’14 conference.

23 The VAX/VMS Virtual Memory System

仮想メモリの調査を終了する前に、VAX/VMS オペレーティング・システム [LL82] にある、きれいに整った仮想メモリ・マネージャの 1 つを詳しく見てみましょう。ここでは、以前の章でもたらされた概念のいくつかが完全なメモリマネージャにどのように集まっているかを示すためのシステムについて説明します。

23.1 Background

VAX-11 ミニコンピュータのアーキテクチャは、Digital Equipment Corporation(DEC) によって 1970 年代後半に導入されました。DEC は、ミニコンピュータの時代にコンピュータ業界で大規模な市場を持った企業でした。残念なことに一連の悪い決定と PC の出現はゆっくりと崩壊につながりました [C03]。このアーキテクチャは、VAX-11/780 やそれほど強力ではない VAX-11/750 など、いくつかの実装で実現されました。

このシステムの OS は VAX/VMS(または単純な VMS) として知られていましたが、主なアーキテクトは Dave Cutler でした。後で Microsoft の Windows NT [C93] を開発しようと努力しました。非常に安価な VAXen を含む広範囲のマシンで、同じアーキテクチャファミリーの非常にハイエンドでパワフルなマシンです。したがって、OS には、この巨大なシステム全体で機能し、うまく機能する仕組みとポリシーが必要でした。

THE CRUX: HOW TO AVOID THE CURSE OF GENERALITY オペレーティングシステムはしばしば「一般性の呪い」として知られている問題を抱えています。これらは幅広い種類のアプリケーションやシステムの一般的なサポートが行われています。この呪いの基本的な結果は、OS がいざれかのインストールを非常にうまくサポートしない可能性があるということです。VMS の場合、VAX-11 アーキテクチャーはさまざまな実装で実現されていたため、呪いは非常にリアルでした。したがって、幅広いシステムで効果的に動作するように OS を構築するにはどうすればよいですか？

追加の問題として、VMS はアーキテクチャの固有の欠陥のいくつかを隠すために使用されるソフトウェア革新の優れた例です。OS は効率的な抽象化と錯覚を構築するためにハードウェアに依存していることがよくありますが、ハードウェア設計者はすべてのことを正しく行うことができません。VAX ハードウェアにはいくつかの例があり、これらのハードウェアの欠陥にもかかわらず、VMS オペレーティングシステムが効果的で実用的なシステムを構築するために何をしているのかがわかります。

23.2 Memory Management Hardware

VAX-11 は、1 プロセス当たり 32 ビットの仮想アドレス空間を 512 バイトのページに分割して提供しました。したがって、仮想アドレスは 23 ビットの VPN と 9 ビットのオフセットで構成されています。さらに、VPN の上位 2 ビットを使用して、ページがどのセグメントに存在するかを区別しました。したがって、システムは以前に見たようにページングとセグメンテーションのハイブリッドでした。アドレス空間の下半分は「プロセス空間」と呼ばれ、各プロセスに固有のものでした。プロセス空間の前半 (P0 として知られている) では、ユーザープログラムが見つかっただけでなく、下向きに成長するヒープも検出されます。プロセス空間 (P1) の後半では、上向きに成長するスタックを見つけます。アドレススペースの上半分はシステムスペース (S) として知られていますが、半分しか使用されていません。保護された OS コードとデータはここにあり、OS はこのようにしてプロセス間で共有されます。

VMS デザイナの主な関心事の 1 つは、VAX ハードウェア (512 バイト) のページのサイズが非常に小さいことでした。歴史的な理由から選択されたこのサイズは、単純な線形ページテーブルを過度に大きくするという基本的な問題があります。したがって、VMS デザイナの最初の目標の 1 つは、VMS がページテーブルを使

用してメモリを圧迫しないようにすることでした。

システムは、2つの方法でメモリ上の圧迫するページテーブルの場所を減らしました。まず、ユーザー・アドレス空間を2つに分割することにより、VAX-11はプロセスごとにこれらのリージョン(P0とP1)のそれぞれにページ・テーブルを提供します。従って、スタックとヒープとの間のアドレス空間の未使用部分にはページテーブルスペースは必要ありません。baseとboundsレジスタは期待どおりに使用されます。ベースレジスタはそのセグメントのページテーブルのアドレスを保持し、境界はそのサイズ(すなわちページテーブルエントリの数)を保持します。

第2に、OSは、カーネル仮想メモリにユーザページテーブル(P0とP1のために1プロセスあたり2つ)を配置することにより、さらにメモリ圧迫を減らします。このように、ページテーブルを割り振ったり、成長させたりすると、カーネルは、セグメントS内に、それ自身の仮想メモリから空間を割り当てます。メモリが厳しい状況になると、カーネルはこれらのページテーブルのページをディスクにスワップして、他の用途のために使用します。

ページテーブルをカーネル仮想メモリに置くと、アドレス変換がさらに複雑になります。たとえば、P0またはP1の仮想アドレスを変換するには、まずハードウェアがページテーブル(そのプロセスのP0またはP1ページテーブル)でそのページのページテーブルエントリを検索する必要があります。ただし、ハードウェアはまずシステムページテーブル(物理メモリに存在する)を参照する必要があります。その変換が完了すると、ハードウェアはページテーブルのページのアドレスを学習し、最後に望んだメモリアクセスのアドレスを知ることができます。幸いにも、これはVAXのハードウェア管理のTLBによって高速化されています。このTLBは通常、この厄介な検索を回避します(うまくいけばの話ですが…)

23.3 A Real Address Space

VMSを研究するうえでの真面目なところは、実際のアドレス空間がどのように構築されているかを見ることができます(図23.1)。ここまででは、ユーザコード、ユーザデータ、ユーザヒープだけの単純なアドレス空間を想定していましたが、実アドレス空間は明らかに複雑です。

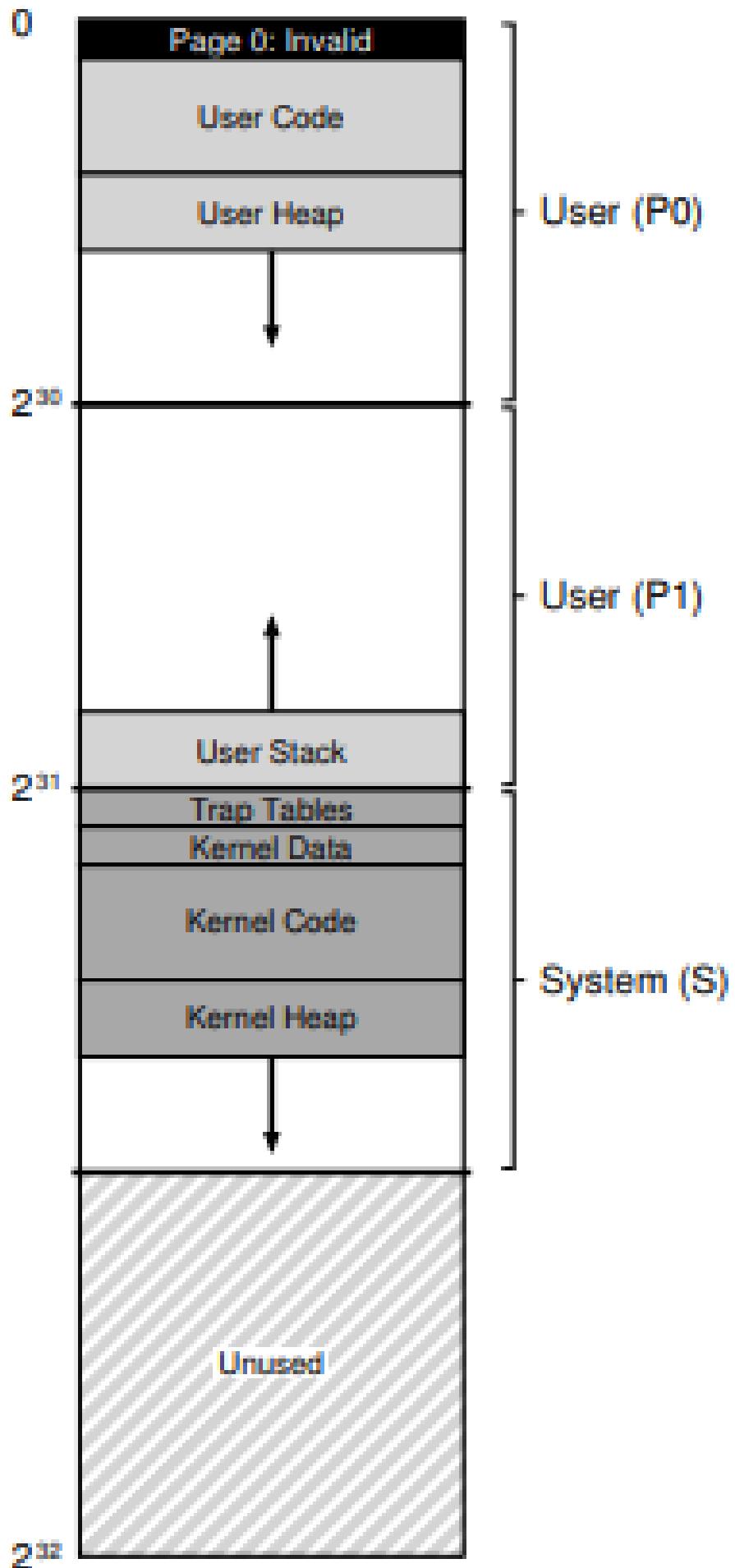


Figure 23.1: The VAX/VMS Address Space

ASIDE: WHY NULL POINTER ACCESSES CAUSE SEG FAULTS

NULL ポインタ逆参照で何が起こるかを正確に理解する必要があります。プロセスは、次のようにすることによって仮想アドレス 0 を生成します。

```
int *p = NULL; // set p = 0
*p = 10; // try to store value 10 to virtual address 0
```

ハードウェアは、TLB 内の VPN(ここでも 0) をルックアップしようとし、TLB ミスを起こします。ページテーブルが参照され、VPN 0 のエントリが無効とマークされていることがわかります。したがって、私たちは無効なアクセス権だったため、OS に制御を移し、プロセスを終了させる可能性があります (UNIX システムでは、プロセスにそのようなフォールトに反応するシグナルが送信されますが、キャッチされないとプロセスは強制終了されます)

例えば、コードセグメントはページ 0 から始まりません。代わりに、NULL ポインタアクセスを検出するためのサポートを提供するために、このページはアクセス不能とマークされます。したがって、アドレス空間を設計する際の懸案事項の 1 つはデバッグのサポートであり、アクセスできないゼロページはここで何らかの形で提供されます。

おそらくもっと重要なことに、カーネル仮想アドレス空間 (すなわち、そのデータ構造およびコード) は各ユーザアドレス空間の一部です。コンテキストスイッチでは、OS は、すぐに実行されるプロセスの適切なページテーブルを指すように P0 および P1 レジスタを変更します。しかし、S のベースレジスタと境界レジスタは変更されず、その結果、「同じ」カーネル構造が各ユーザアドレス空間にマッピングされます。

カーネルは、いくつかの理由から各アドレス空間にマップされます。このような構成により、カーネルの作業が容易になります。例えば、OS にポインタを渡すと (例えば、`write()` システムコールで)、そのポインタからのデータをそれ自身の構造にコピーすることは容易です。OS は、アクセスしているデータがどこから来るのか心配することなく、自然に書かれコンパイルされます。これとは対照的に、カーネルが完全に物理メモリに置かれていた場合、ページテーブルのページをディスクにスワップするなどの作業は非常に難しくなります。カーネルに独自のアドレス空間が与えられていれば、ユーザー-application とカーネル間でデータを移動することは、かなり複雑になってしまいます。この構造 (現在広く使われている) では、カーネルは、保護されているものの、アプリケーションのライブラリのように見えます。

このアドレス空間に関する最後の 1 つのポイントは、保護に関するものです。明らかに、OS はユーザアプリケーションが OS のデータやコードを読み書きすることを望みません。従って、ハードウェアは、これを可能にするためにページに対して異なる保護レベルをサポートしなければいけません。VAX は、ページテーブルの保護ビットに、特定のページにアクセスするために CPU が必要とする特権レベルを指定することによってそうしました。したがって、システムデータおよびコードは、ユーザデータおよびコードよりも高い保護レベルに設定されます。そのような情報にユーザーコードからアクセスしようとすると、OS にトラップが生成され、問題のプロセスが終了する可能性があります。

23.4 Page Replacement

VAX のページテーブルエントリ (PTE) には、有効ビット、保護フィールド (4 ビット)、変更 (またはダーティ) ビット、OS 使用のために予約されたフィールド (5 ビット)、および最後に物理メモリにページの場所を格納するための物理フレーム番号 (PFN) があります。しかし、参照ビットがありません。したがって、VMS 置換アルゴリズムは、どのページがアクティブであるかを決定するためのハードウェアサポートなしで行う必要があります。

開発者はまた、メモリを多くのメモリを使用するプログラムがあり、他のプログラムを実行するのを困難にする場合について懸念していました。これまで見てきたポリシーのほとんどは、このような騒ぎに敏感です。

たとえば、LRU はプロセス間でメモリを公平に共有しないグローバルポリシーです。

Segmented FIFO

この 2 つの問題に対処するために、開発者はセグメント化された FIFO 置換ポリシー [RL81] を考え出しました。アイデアは単純です。各プロセスには、resident set size(RSS) と呼ばれる、メモリに保存できる最大ページ数があります。これらの各ページは FIFO リストに保持されます。プロセスがその RSS を超えると、「先入れ先出し (first-in)」ページが追い出されます。FIFO はハードウェアからのサポートを必要とせず、実装が容易です。

当然のことながら、純粋な FIFO は、以前のように、特にうまく機能しません。FIFO のパフォーマンスを向上させるために、VMS はメモリから削除される前にページが配置される 2 つのセカンドチャンスリストを考えました。それはグローバルクリーンページフリーリストとダーティページリストです。プロセス P がその RSS を超えると、ページはプロセスごとの FIFO から削除されます。クリーン (変更されていない) の場合はクリーンページリストの最後に配置されます。汚れている (変更されている) 場合は、ダーティページリストの最後に配置されます。

別のプロセス Q に空きページが必要な場合は、グローバルクリーンリストから最初の空きページが削除されます。ただし、元のプロセス P がページフォルトが発生した場合、元のプロセス P が再利用される前に、P はフリー (またはダーティ) リストからそのページを再要求し、コストのかかるディスクアクセスを回避します。これらのセカンドチャンスリストが大きいほど、セグメント化された FIFO アルゴリズムは LRU [RL81] に近づきます。

Page Clustering

VMS で使用される別の最適化は、VMS の小さなページサイズを克服するのにも役立ちます。具体的には、このような小さいページでは、スワップ時のディスク I/O は、ディスクが大規模な転送をもっているものでよりうまくいくので、非常に非効率的である可能性があります。VMS は、I/O のスワッピングをより効率的にするために、いくつかの最適化を追加しますが、最も重要なのはクラスタリングです。クラスタリングでは、VMS は大規模なダーティー・リストから大量のバッチ・ページをグループ化し、それらを一気にディスクに書き込むことで、クリーンになります。スワップスペース内のどこにでもページを配置できるため、OS グループのページを作成したり、書き込み回数を減らしたりすることができます。パフォーマンスが向上するため、現代のシステムではクラスタリングが使用されています。

ASIDE: EMULATING REFERENCE BITS

分かっているように、システムでどのページが使用されているかという概念を得るために、ハードウェア参照ビットは必要ありません。実際、1980 年代初めに、Babaoglu と Joy は、VAX の保護ビットを使用して参照ビットをエミュレートすることができることを示しました [BJ81]。基本的な考え方として、システムでどのページが積極的に使用されているかを理解したい場合は、ページテーブルのすべてのページをアクセス不可能とマークします (しかし、プロセスによって実際にアクセス可能なページ、おそらくページテーブルエントリの「予約された OS フィールド」部分にあるかもしれません)。

プロセスがページにアクセスすると、OS にトラップが生成されます。OS はページが実際にアクセス可能であるかどうかをチェックし、そうであればページを通常の保護 (例えば、読み取り専用または読み書き) に戻します。その置換時に、OS はどのページがアクセス不能とマークされているかを確認することができます。

そのため、どのページが最近使用されていないのかを知ることができます。参照ビットのこの「エミュレーション」の鍵は、オーバーヘッドを削減しながら、ページ使用の良いアイデアを得ること

です。OSは、アクセス不可能なページをマーキングするにはあまりにも攻撃的であるため、やつてはいけません。また、オーバーヘッドが高すぎます。OSはまた、そのようなマーキングではあまりにも受動的であってはなりません。そうでないと、すべてのページが参照されます。また、OSは、どのページを退去させるべきかについては、まったく考えていません。

23.5 Other Neat VM Tricks

VMSには、デマンドゼロ(demand zeroing)とコピーオン・ライトの2つの標準的なトリックがあります。ここで、これらの遅延最適化について説明します。

VMS(およびほとんどの現代システム)における怠惰の1つの形態は、ページの要求のゼロ化(デマンドゼロ)です。これをよりよく理解するために、アドレス空間にヒープ内のページを追加する例を考えてみましょう。単純な実装では、OSはヒープにページを追加する要求に応答し、物理メモリ内のページを見つけてゼロにします(セキュリティが必要な場合は、他のプロセスが発生したときのページの内容を確認できます)それをあなたのアドレス空間にマッピングします(つまり、物理ページを望むようにページテーブルを設定する)。しかし、特にページがプロセスによって使用されない場合は、単純な実装にはコストがかかる可能性があります。

デマンドゼロは、アドレス空間にページが追加されてもOSはほとんど機能しません。ページ・テーブルには、そのページにアクセスできないとマークするエントリが挿入されます。プロセスがページを読み書きすると、OSへのトラップが発生します。トラップを処理するとき、OSは(実際にはページテーブルエントリの「OSのために予約された」部分に記されたビットを通して)これが実際にはデマンドゼロのページであることに気づきます。この時点で、OSは、物理ページを見つけ出し、ゼロにし、プロセスのアドレス空間にマッピングするために必要な作業を行います。プロセスがページにアクセスしない場合、この作業はすべて回避されます。

TIP: BE LAZY 遅延であることは、オペレーティングシステムに利点をもたらします。遅延は後で作業を延期することができます。これは、いくつかの理由でOS内で有益です。まず、作業を中止すると、現在の操作の待ち時間が短縮され、応答性が向上します。たとえば、オペレーティングシステムでは、ファイルへの書き込みがすぐに成功したことを報告し、バックグラウンドで後でディスクに書き込むことがよくあります。第二に、さらに重要なことに、遅延は時にはその作業をやる必要性をなくします。例えば、ファイルが削除されるまで書き込みを遅延させると、書き込みを一切行う必要がなくなります。

VMSで見いだされたもうひとつのクールな最適化(やはり、現代のあらゆるOSで)は、コピーオンライト(略してCOW)です。アイデアとしてはTENEXオペレーティングシステム[BB+72]というものがあります。そのアイデアは簡単です。OSがコピーする代わりに、あるアドレス空間から別のアドレス空間にページをコピーする必要があるとき、それをターゲットアドレスにマップすることができます。このとき両方のアドレス空間で読み取り専用にマークします。両方のアドレス空間がページだけを読み込んだ場合、OSは実際にデータを移動することなく高速コピーを実現します。ただし、アドレススペースの1つが実際にページに書き込もうとすると、OSにトラップされます。OSはそのページがCOWページであることに気づき、新しいページを割り当て、データで埋めて、この新しいページをフォールトを起こしたプロセスのアドレス空間にマッピングします。その後、プロセスは続行され、そのプロセス専用のコピーが作成されます。

COWは多くの理由で有用です。確かにどんな種類の共有ライブラリも、多くのプロセスのアドレス空間にコピーオンライトでマッピングすることができ、貴重なメモリースペースを節約できます。UNIXシステムでは、COWは`fork()`と`exec()`の正確さのためにさらに重要です。呼び出すことができるよう、`fork()`は呼び出し元のアドレス空間の正確なコピーを作成します。大きなアドレス空間では、そのようなコピーを作成するのが遅く、データ集約的です。さらに悪いことに、ほとんどのアドレス空間は`exec()`の後続の呼び出し

によって直ちに上書きされ、呼び出し元プロセスのアドレス空間とすぐに実行されるプログラムのアドレス空間が上書きされます。代わりに、コピーオンライトの `fork()` を実行することにより、OS は不必要的コピーをほとんど回避し、したがってパフォーマンスを向上させながら正しいデータを保持します。

23.6 Summary

仮想メモリシステム全体のトップからボトムのレビューを見たことがあります。基本的な仕組みやポリシーの大部分をすでによく理解したので、細部のほとんどは簡単に理解できるはずです。詳細は Levy and Lipman [LL82] の優れた（そして短い）論文に掲載されています。これらの章の背後にある原資料がどのようなものかを見るための素晴らしい方法です。

可能であれば、Linux やその他の最新のシステムについて読むことで、最先端技術についてさらに学ぶ必要があります。そこにはいくつかの合理的な書籍 [BC05] を含む多くのソース資料があります。VAX/VMS 上の古い文書に見られる古典的なアイデアが、現代のオペレーティングシステムの構築方法にどのように影響しているのか、あなたを驚かせるでしょう。

参考文献

- [BB+72] “TENEX, A Paged Time Sharing System for the PDP-10”
Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, Raymond S. Tomlinson
Communications of the ACM, Volume 15, March 1972
An early time-sharing OS where a number of good ideas came from. Copy-on-write was just one of those; inspiration for many other aspects of modern systems, including process management, virtual memory, and file systems are found herein.
- [BJ81] “Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Reference Bits”
Ozalp Babaoglu and William N. Joy
SOSP '81, Pacific Grove, California, December 1981
A clever idea paper on how to exploit existing protection machinery within a machine in order to emulate reference bits. The idea came from the group at Berkeley working on their own version of UNIX, known as the Berkeley Systems Distribution, or BSD. The group was heavily influential in the development of UNIX, in virtual memory, file systems, and networking.
- [BC05] “Understanding the Linux Kernel (Third Edition)”
Daniel P. Bovet and Marco Cesati
O'Reilly Media, November 2005
One of the many books you can find on Linux. They go out of date quickly, but many of the basics remain and are worth reading about.
- [C03] “The Innovator's Dilemma”
Clayton M. Christensen
Harper Paperbacks, January 2003
A fantastic book about the disk-drive industry and how new innovations disrupt existing ones. A good read for business majors and computer scientists alike. Provides insight on how large and successful companies completely fail.
- [C93] “Inside Windows NT”
Helen Custer and David Solomon

Microsoft Press, 1993

The book about Windows NT that explains the system top to bottom, in more detail than you might like. But seriously, a pretty good book.

[LL82] "Virtual Memory Management in the VAX/VMS Operating System"

Henry M. Levy, Peter H. Lipman

IEEE Computer, Volume 15, Number 3 (March 1982) Read the original source of most of this material; it is a concise and easy read. Particularly important if you wish to go to graduate school, where all you do is read papers, work, read some more papers, work more, eventually write a paper, and then work some more. But it is fun!

[RL81] "Segmented FIFO Page Replacement"

Rollins Turner and Henry Levy

SIGMETRICS '81, Las Vegas, Nevada, September 1981

A short paper that shows for some workloads, segmented FIFO can approach the performance of LRU

第 II 部

Concurrency

26 Concurrency: An Introduction

ここまででは、OS が実行する基本的な抽象概念の開発を見てきました。単一の物理 CPU を複数の仮想 CPU に変換する方法を見てきました。これにより、複数のプログラムが同時に実行されているように見えます。また、プロセスごとに大きな仮想プライベート仮想メモリを作成する方法を見てきました。このアドレス空間の抽象化によって、OS が実際に物理メモリ（および場合によってはディスク）上のアドレス空間を秘密に多重化しているときに、各プログラムが独自のメモリを持つかのように動作することができます。

ここでは、スレッドの 1 つの実行プロセスに対する新しい抽象概念を紹介します。プログラム内の単一の実行ポイント（すなわち、命令がフェッチされて実行される 1 つの PC（プログラムカウンタ））の古典的なビューの代わりに、マルチスレッドプログラムは複数の実行ポイントを持っています（つまり、複数の PC、それはフェッチされ実行されている）おそらく、これを考へるもう一つの方法は、各スレッドが 1 つの違いを除いて、別々のプロセスに非常に似ていることです。つまり、同じアドレス空間を共有し、同じデータにアクセスできます。

したがって、単一のスレッドの状態は、プロセスの状態と非常に似ています。それは、プログラムが命令をフェッチする場所を追跡するプログラムカウンタ（PC）を持っています。各スレッドには、計算に使用する専用のレジスタセットがあります。したがって、1 つのプロセッサ上で実行されている 2 つのスレッドがある場合、1 つを実行する (T1) からもう一方のスレッドを実行する (T2) に切り替えるときは、コンテキスト切り替えが行われなければなりません。スレッド間のコンテキスト切り替えは、プロセス間のコンテキスト切り替えと非常によく似ています。これは、T1 のレジスタ状態を保存し、T2 を実行する前に T2 のレジスタ状態を復元する必要があります。プロセスでは、状態をプロセス制御ブロック（PCB）に保存しました。現在、プロセスの各スレッドの状態を格納するために、1 つ以上のスレッド制御ブロック（TCB）が必要になります。ただし、プロセスと比較してスレッド間で実行するコンテキスト切り替えには大きな違いが 1 つあります。アドレス空間は同じままで（つまり、使用しているページテーブルを切り替える必要はありません）

スレッドとプロセスのもう 1 つの大きな違いは、スタックに関するものです。従来のプロセス（单一スレッドプロセスと呼ぶこともできる）のアドレス空間の単純なモデルでは、通常はアドレス空間の一番下に单一のスタックがあります（図 26.1、左）

しかし、マルチスレッドプロセスでは、各スレッドは独立して実行され、もちろんどのような作業をしていてもさまざまなルーチンを呼び出すことができます。アドレス空間には 1 つのスタックの代わりにスレッドごとに 1 つのスタックが存在します。たとえば、2 つのスレッドを持つマルチスレッドプロセスがあるとします。結果のアドレス空間は異なって見えます（図 26.1、右）。

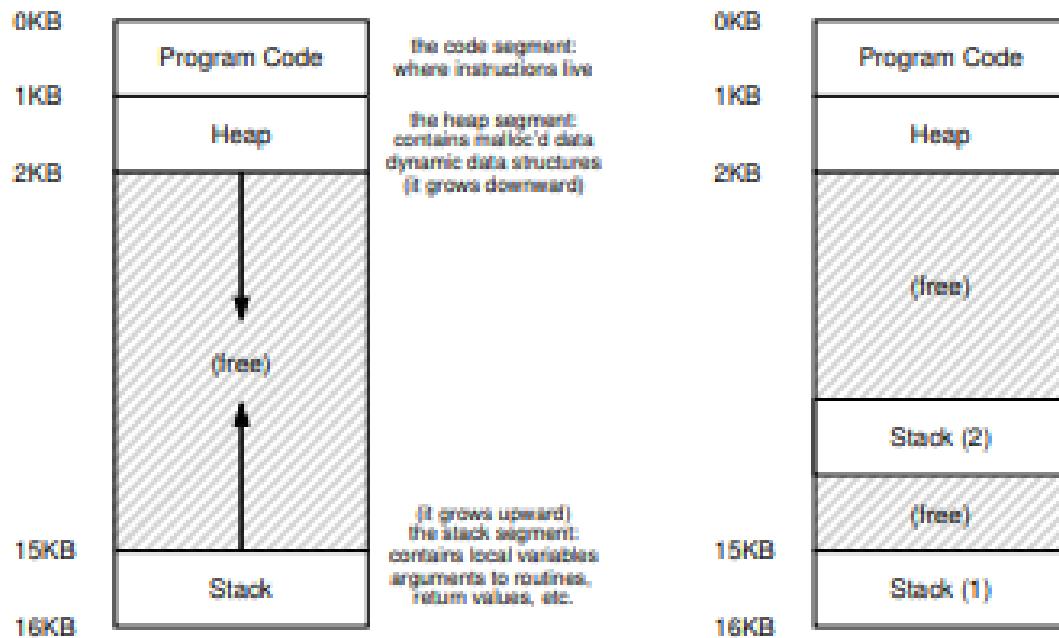


Figure 26.1: Single-Threaded And Multi-Threaded Address Spaces

この図では、プロセスのアドレス空間全体に 2 つのスタックが広がっていることがわかります。したがって、スレッドに割り当てられた変数、パラメータ、戻り値、およびスタック上に置かれた他のものは、thread local storage と呼ばれることがあるスタックに格納されます。

これがどのように私たちの美しいアドレス空間のレイアウトを崩すのか気づくかもしれません。以前は、スタックとヒープが独立して成長する可能性があり、アドレス空間の空き領域を使い果たしたときにのみ問題が発生しました。ここでは、もはやこのような素晴らしい状況はありません。幸運なことに、スタックは一般に非常に大きくする必要はありません（例外は再帰を頻繁に使用するプログラムです）

26.1 Why Use Threads?

スレッドの詳細とマルチスレッドプログラムの作成に伴う問題のいくつかに入る前に、もっと簡単な質問にまず答えてみましょう。なぜスレッドを使うべきなのでしょうか？

スレッドを使用する主な理由は少なくとも 2 つあります。最初のものは単純です。それは並列性です。たとえば、2 つの大きな配列と一緒に追加するか、配列内の各要素の値をある量だけインクリメントするなど、非常に大きな配列で操作を実行するプログラムを作成しているとします。单一のプロセッサで実行している場合、タスクは簡単です。それぞれの操作を実行するだけです。ただし、複数のプロセッサを搭載したシステム上でプログラムを実行している場合は、各プロセッサを使用して作業の一部を実行することで、このプロセスを大幅に高速化する可能性があります。標準のシングルスレッドプログラムを複数の CPU でこの種の作業を行うプログラムに変換する作業は並列化と呼ばれ、この作業を行う CPU ごとのスレッドを使用するのは、現代のハードウェアでプログラムをより高速に実行できる自然で一般的な方法です。

2 つ目の理由は、I/O が遅いことによる、ブロッキングされているプログラムの進行を避けるためです。さまざまなタイプの I/O を実行するプログラムを記述しているとします。メッセージの送信または受信の待機、明示的なディスク I/O の完了、またはページ・フォールトの完了（暗黙的）です。待機する代わりに、プログラムを使用して CPU を使用して計算を実行します、さらに I/O 要求を発行するなど、何か他の処理を実行することもできます。スレッドを使用するのは自然な方法です。プログラム内の 1 つのスレッドが待機している

(つまり、I/O を待ってブロックされている) 場合、CPU スケジューラは実行準備が整っていて有用なことをする他のスレッドに切り替えることができます。スレッディングは、プログラム間のプロセスのマルチプログラミングのように、単一のプログラム内の他のアクティビティとの I/O のオーバーラップを可能にします。その結果、多くの現代のサーバベースのアプリケーション（ウェブサーバ、データベース管理システムなど）は、その実装においてスレッドを利用します。

もちろん、上記のいずれの場合でも、スレッドの代わりに複数のプロセスを使用できます。しかし、スレッドはアドレス空間を共有するため、データの共有が容易になり、したがって、これらのタイプのプログラムを構築する際には自然な選択です。プロセスは、メモリ内のデータ構造をほとんど共有する必要のない、論理的に分離したタスクのためのより健全な選択です。

26.2 An Example: Thread Creation

詳細をいくつか取り上げましょう。私たちは 2 つのスレッドを作成するプログラムを実行したいとします。それぞれのスレッドは独立した作業を行います。この場合、「A」または「B」を印刷します。コードは図 26.2(278 ページ) に示されています。メインプログラムは 2 つのスレッドを作成し、それぞれは異なる引数（文字列 A または B）を使用しても、関数 `mythread()` を実行します。スレッドが作成されると、すぐに実行を開始することができます（スケジューラーの気まぐれに応じて）。あるいは、「実行可能」状態であるが、まだ実行状態ではなく、実行されていないと置いてみましょう。もちろん、マルチプロセッサ上では、スレッドは同時に実行することもできますが、この可能性についてはまだ心配しないでください。

```

1  #include <stdio.h>
2  #include <assert.h>
3  #include <pthread.h>
4
5  void *mythread(void *arg) {
6      printf("%s\n", (char *) arg);
7      return NULL;
8  }
9
10 int
11 main(int argc, char *argv[]) {
12     pthread_t p1, p2;
13     int rc;
14     printf("main: begin\n");
15     rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
16     rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
17     // join waits for the threads to finish
18     rc = pthread_join(p1, NULL); assert(rc == 0);
19     rc = pthread_join(p2, NULL); assert(rc == 0);
20     printf("main: end\n");
21     return 0;
22 }
```

Figure 26.2: Simple Thread Creation Code (t0.c)

2 つのスレッド（T1 と T2 と呼ぶ）を作成した後、メインスレッドは `pthread_join()` を呼び出し、特定のスレッドが完了するのを待ちます。このように 2 回実行すると、T1 と T2 が確実に実行され、完了してからメインスレッドが再び実行できるようになります。終了すると、「main : end」と表示され、終了します。全体的に、この実行中に 3 つのスレッド、すなわちメインスレッド、T1、および T2 が使用されました。

この小さなプログラムの実行順序を調べてみましょう。実行ダイアグラム（図 26.3）では、時間が下方向に増加し、各列は異なるスレッド（メインスレッド、スレッド 1、またはスレッド 2）が実行されていることを示

します。

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1	runs	
creates Thread 2	prints "A"	
waits for T1	returns	
waits for T2		runs
		prints "B"
		returns
prints "main: end"		

Figure 26.3: Thread Trace (1)

ただし、この順序は唯一の可能な順序ではないことに注意してください。実際には、一連の命令があれば、スケジューラが特定のポイントで実行するスレッドに応じてかなりの数があります。たとえば、スレッドが作成されるとすぐにスレッドが実行され、図 26.4 に示す実行につながります。

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
<i>returns immediately; T1 is done</i>		
waits for T2		
<i>returns immediately; T2 is done</i>		
prints "main: end"		

Figure 26.4: Thread Trace (2)

スレッド 1 が先に作成されたとしても、スケジューラがスレッド 2 を最初に実行することに決めた場合、「A」の前に「B」が印刷されていることもあります。最初に作成されたスレッドが最初に実行されると仮定する理由はありません。図 26.5 にこの最終実行順序を示します。スレッド 2 はスレッド 1 より前にスレッドを処理します。

main	Thread 1	Thread 2
starts running		
prints "main: begin"		
creates Thread 1		runs
creates Thread 2		prints "B"
		returns
waits for T1	runs	
	prints "A"	
	returns	
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Figure 26.5: Thread Trace (3)

あなたが見ることができるように、スレッドの作成について考える方法の1つは、関数呼び出しのようなものです。しかし、関数を最初に実行してから呼び出し元に戻る代わりに、呼び出されているルーチンの新しい実行スレッドが作成され、呼び出し元から独立して実行されます(おそらく作成から戻る前に)次に実行されるのはOSスケジューラによって決定され、スケジューラはいくつかの実用的なアルゴリズムを実装する可能性がありますが、特定の瞬間に何が実行されるのかを知ることは困難です。

この例からも分かるように、スレッドを使用すると複雑な作業になります。いつ実行するのかはすでに分かりません！コンピュータは、並行性なしでは理解するのに十分なほど難しい。残念ながら、同時実行性では、単に悪化します。ずっと悪いです。

#26.3 Why It Gets Worse: Shared Data 上に示した単純なスレッドの例は、スケジューラーがそれらを実行する方法に応じて、スレッドがどのように作成され、どのように異なる順序で実行されるかを示すのに役立ちました。しかし、共有データにアクセスするときにスレッドがどのようにやりとりするかは、あなたには分かりません。2つのスレッドがグローバル共有変数を更新したいという単純な例を想像してみましょう。調査するコードは、図26.6(280ページ)にあります。

```

1 #include <stdio.h>
2 #include <pthread.h>
3 #include "mythreads.h"
4
5 static volatile int counter = 0;
6
7 //
8 // mythread()
9 //
10 // Simply adds 1 to counter repeatedly, in a loop
11 // No, this is not how you would add 10,000,000 to
12 // a counter, but it shows the problem nicely.
13 //
14 void *
15 mythread(void *arg)
16 {
17     printf("%s: begin\n", (char *) arg);
18     int i;
19     for (i = 0; i < 1e7; i++) {
20         counter = counter + 1;
21     }
22     printf("%s: done\n", (char *) arg);
23     return NULL;
24 }
25
26 //
27 // main()
28 //
29 // Just launches two threads (pthread_create)
30 // and then waits for them (pthread_join)
31 //
32 int
33 main(int argc, char *argv[])
34 {
35     pthread_t p1, p2;
36     printf("main: begin (counter = %d)\n", counter);
37     Pthread_create(&p1, NULL, mythread, "A");
38     Pthread_create(&p2, NULL, mythread, "B");
39
40     // join waits for the threads to finish
41     Pthread_join(p1, NULL);
42     Pthread_join(p2, NULL);
43     printf("main: done with both (counter = %d)\n", counter);
44     return 0;
45 }

```

Figure 26.6: Sharing Data: Uh Oh (t1.c)

コードについてのいくつかの注意があります。まず、スティーブンスが [SR05] を提案しているように、スレッドの作成と結合ルーチンをラップして失敗時に終了するだけです。このようなシンプルなプログラムでは、少なくともエラーが発生したことに気づきたいですが、それについては非常に賢明なことはしません (exit など)。したがって、`pthread_create()` は単に `pthread_create()` を呼び出し、戻りコードが 0 であること

を確認します。そうでない場合、`pthread_create()` は単にメッセージを出力して終了します。

第 2 に、ワーカースレッドに 2 つの別々の関数本体を使用する代わりに、単一のコードを使用してスレッドに引数(この場合は文字列)を渡すだけで、各スレッドがメッセージの前に別の文字を表示できるようになります。

最後に、最も重要なのは、各作業者が何をしようとしているのかを見てみましょう。共有変数カウンタに数值を追加し、ループで 1000 万回($1e7$)を実行します。従って、望む最終結果は、20,000,000 です。

プログラムのコンパイルと実行を行い、プログラムの動作を確認します。時には、すべてが期待通りに働くこともあります。

```
prompt> gcc -o main main.c -Wall -pthread
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

残念ながら、このコードを実行すると、単一のプロセッサであっても、必ずしも望ましい結果が得られることは限りません。

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

結局のところ、コンピュータは決定論的な結果を生み出すとは思われませんか？

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19221041)
```

それぞれが正しく実行されるだけでなく、別の結果が得られます。大きな疑問が残っています。なぜこれが起こりますか？

TIP: KNOW AND USE YOUR TOOLS

コンピュータシステムの作成、デバッグ、および理解に役立つ新しいツールを常に学習する必要があります。ここでは、ディスアセンブラーと呼ばれる素敵なツールを使用します。実行ファイルに対して逆アセンブラーを実行すると、どのアセンブリ命令がプログラムを構成しているかが表示されます。たとえば、カウンタを更新するための低レベルのコードを理解したい場合(この例のように)、

objdump(Linux) を実行してアセンブリコードを表示します。

```
prompt> objdump -d main
```

そうすることで、(特に-g フラグを付けてコンパイルした場合に) きれいに表示され、プログラム内のシンボル情報を含む、プログラム内のすべての命令の長いリストが生成されます。objdump プログラムは、使い方を学ぶべき多くのツールの 1 つに過ぎません。gdb のようなデバッガ、valgrind や purify のようなメモリプロファイラ、もちろんコンパイラ自体はもっと学ぶために時間を費やすべきものです。ツールを使用している方が優れているほど、優れたシステムを構築できます。

26.4 The Heart Of The Problem: Uncontrolled Scheduling

なぜこのようなことが起こるかを理解するためには、コンパイラがカウンタの更新のために生成するコードシーケンスを理解する必要があります。この場合、カウンタに数値 (1) を追加するだけです。したがって、そうするためのコードシーケンスは、(x86 では) このように見えるかもしれません。

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

この例では、変数カウンタがアドレス 0x8049a1c にあると仮定しています。この 3 命令シーケンスでは、最初に x86 mov 命令が使用され、アドレスのメモリ値を取得してレジスタ eax に格納します。次に、add が実行され、eax レジスタの内容に 1(0x1) を加え、最後に eax の内容が同じアドレスのメモリに戻されます。

私たちの 2 つのスレッド (スレッド 1) のうちの 1 つがこのコード領域に入り、カウンタを 1 つ増やすことを想像してみましょう。それは、カウンタの値を読み込みます (最初は 50 としましょう)。その値はレジスタ eax にロードされます。したがって、スレッド 1 の場合は eax = 50 となり、レジスタに 1 が加算されます。従って eax = 51 になります。そして今、何か不幸なことが起こります。ここでタイマー割り込みがオフになります。したがって、OS は現在実行中のスレッド (その PC、eax を含むレジスタなど) の状態をスレッドの TCB に保存します。

スレッド 2 が実行されるように選択され、同じコードが入力されます。また、最初の命令を実行してカウンタの値を取得し、それを eax に入れます (実行時にはスレッドごとに専用のレジスタがあり、レジスタはそれらを保存および復元するコンテキストスイッチコードによって仮想化されます)。この時点での counter の値はまだ 50 であるため、スレッド 2 は eax = 50 です。スレッド 2 が次の 2 つの命令を実行し、eax を 1 つ増やして (つまり eax = 51)、eax の内容をカウンタ (アドレス 0x8049a1c) に保存すると仮定しましょう。したがって、グローバル変数カウンタは今や値 51 を持っています。

最後に、別のコンテキスト切り替えが発生し、スレッド 1 が実行を再開します。それが mov を実行して追加したこと思い出しても、最終的な mov 命令を実行しようとしています。eax = 51 でした。したがって、最終 mov 命令が実行され、その値がメモリに保存されます。カウンタは再び 51 に設定されます。

簡単に言えば、インクリメントカウンタのコードは 2 回実行されていますが、50 で開始されたカウンタは 51 にしかなりません。このプログラムの「正しい」バージョンでは、変数カウンタは 52 と等しくなるはずです。

問題をよりよく理解するための詳細な実行トレースを見てみましょう。この例では、上記のコードが次のシーケンスのようにメモリ上のアドレス 100 にロードされているものと仮定します (RISC 風の命令セットに使用されていたものの、x86 には可変長命令があります。この mov の命令は 5 バイトのメモリ、および add は 3 バイトのみ)

```
100 mov 0x8049a1c, %eax
```

```
105 add $0x1, %eax
108 mov %eax, 0x8049a1c
```

これらの前提で、図 26.7 に何が起こるかを示します。カウンタが値 50 で始まると仮定し、この例をトレースして、何が起こっているかを理解してください。

OS	Thread 1	Thread 2	(after instruction)		
			PC	%eax	counter
	before critical section		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add \$0x1, %eax		108	51	50
interrupt	save T1's state				
	restore T2's state		100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add \$0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt	save T2's state				
	restore T1's state		108	51	51
		mov %eax, 0x8049a1c	113	51	51

Figure 26.7: The Problem: Up Close and Personal

ここで示したことは競合条件と呼ばれ、結果はコードのタイミング実行に依存します。いくつかの不運(実行中のタイミングの悪い時点で発生するコンテキストスイッチ)により、間違った結果が得られます。実際、私たちは毎回異なる結果を得るかもしれません。したがって、私たちはコンピュータから慣れ親しんだ素敵な決定論的な計算の代わりに、この結果を不確定と呼びます。ここでは、出力がどのようになるか分からず、実行中に実際に異なる可能性があります。

このコードを実行する複数のスレッドが競合状態になる可能性があるため、このコードをクリティカルセクションと呼びます。クリティカルセクションは、共有変数(より一般的には共有リソース)にアクセスするコードであり、複数のスレッドで同時に実行してはいけません。

このコードで本当に欲しいのは、私たちが相互排除と呼ぶものです。このプロパティーは、あるスレッドがクリティカルセクション内で実行している場合、他のスレッドがクリティカルセクション内で実行していないことを保証します。

ところで、これらの用語のほとんどは、この分野のパイオニアであった Edsger Dijkstra によって造られたもので、実際にこの作業や他の作業のために Turing Award を受賞しました。問題の驚くほど明確な説明については、1968 年の「一連のプロセスの協力」[D68] の論文を参照してください。本書のこのセクションでは、Dijkstra について詳しく聞いていきます。

26.5 The Wish For Atomicity

この問題を解決する 1 つの方法は、单一のステップで、必要な処理を正確に行い、不意に中断する可能性をなくした、より強力な命令を持つことです。たとえば、このようなスーパーインストラクションがあればどうでしょうか?

```
memory-add 0x8049a1c, $0x1
```

この命令がメモリ位置に値を追加し、ハードウェアが原子的に実行することを保証すると仮定します。命令が実行されると、必要に応じて更新が実行されます。途中で命令を中断することはできません。これは、ハー

ドウェアから受け取った保証のためです。割り込みが発生した場合、命令がまったく実行されていないか、完了まで実行されています。中間の状態は存在しません。

原子的には、この文脈では、「単位として」を意味し、時には「すべてかどうか」とみなします。私たちが望むものは、3つの命令シーケンスを原子的に実行することです。

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

もしこれを行う一つの指示があれば、私たちはその指示を出すだけで済みます。しかし、一般的なケースでは、そのような指示はありません。私たちが並行してBツリーを構築していて、それを更新したいと思ったとします。ハードウェアが「Bツリーのアトミック更新」命令をサポートすることを本当に望んでいますか？おそらく、少なくとも純粋な命令セットではないでしょう。

したがって、代わりに、ハードウェアにいくつかの有用な命令を頼んで、同期プリミティブと呼ばれる一般的なセットを構築することができます。これらのハードウェア同期プリミティブを使用することで、オペレーティングシステムの助けを得て、クリティカルセクションに同期して制御された方法でアクセスするマルチスレッドコードを構築することができます。同時実行の困難な性質にもかかわらず、正確に正しい結果を生成します。同時実行、かなり素晴らしいですよね？

TIP: USE ATOMIC OPERATIONS 原子操作は、コンピュータ・アーキテクチャからコンカレント・コード（ここで学んでいるもの）、ファイル・システム（すぐに研究する予定です）、データベース管理システム、さらには分散システム [L + 93] といったシステムに使われている強力な技術です。

一連のアクションをアトミックにするという背後にあるアイデアは、「すべてかどうか」というフレーズで単純に表現されます。グループ化するすべてのアクションが発生したか、いずれも発生していない状態で、中間の状態が表示されていないかのように表示されます。時には、多くのアクションを1つのアトミックアクションにグループ化することをトランザクションと呼びます。これはデータベースとトランザクション処理の世界で非常に詳細に開発されたアイデアです [GR92]。

並行処理のテーマでは、短いシーケンスの命令を実行のアトミックブロックに変換するために同期プリミティブを使用しますが、アトミック性のアイデアはそれよりもはるかに大きくなります。たとえば、ファイルシステムでは、ディスク障害時に正しく動作するために不可欠なディスク上の状態をアトミックに移行するために、ジャーナリングやコピー・オン・ライトなどの技術を使用します。それが意味をなさないのであれば、心配しないでください。この後の章で学んでいきます。

これは本のこのセクションで研究する問題です。それは素晴らしい、難しい問題であり、あなたの心が傷つくはずです（少し）。それがなければ、あなたは理解できません！あなたの頭が痛むまで働き続けてください。あなたの頭が正しい方向に向かっていることを知っています。その時点では休憩をとります。私たちはあなたの頭があまりにも傷ついて欲しくない。

THE CRUX: HOW TO PROVIDE SUPPORT FOR SYNCHRONIZATION 有用な同期プリミティブを構築するために、ハードウェアから何をサポートする必要がありますか？OSから何をサポートする必要がありますか？これらのプリミティブを正確かつ効率的に構築するにはどうすればよいですか？プログラムはどのようにして目的の結果を得ることができますか？

26.6 One More Problem: Waiting For Another

この章では、共有変数にアクセスするスレッドとクリティカルセクションのアトミック性をサポートする必要があるスレッド間で、1つのタイプの対話しか発生しないように、並行性の問題を設定しています。それが判明すると、別の一般的なやりとりが起こります。あるスレッドは、別のスレッドが続行する前に何らかのアクションを完了するのを待たなければなりません。この相互作用は、たとえば、プロセスがディスク I/O を実行しスリープ状態になったときに発生します。I/O が完了すると、プロセスを継続して実行できるように、プロセスをその眠りから呼び出す必要があります。

したがって、今後の章では、アトミック性をサポートするための同期プリミティブのサポートを構築する方法だけでなく、マルチスレッドプログラムでよく見られるこの種のスリープ/スリープ状態の相互作用をサポートするメカニズムについても検討します。これが今や意味がわからないのであれば、それは大丈夫です！その章を何回も読み返すことです。そうでなければ、うまくいきません。

26.7 Summary: Why in OS Class?

ラップアップする前に、あなたが持っているかもしれない1つの質問です。なぜ私たちはこれをOSクラスで勉強していますか？「歴史」は1語の答えです。OSは最初の並行プログラムであり、多くの技術がOS内で使用するために作成されました。その後、マルチスレッドプロセスでは、アプリケーションプログラマもそのようなことを考慮する必要がありました。

たとえば、2つのプロセスが実行されている場合を考えてみましょう。ファイルに書き込むために `write()` を呼び出し、両方ともファイルにデータを追加する（つまり、データをファイルの最後に追加して長さを増やす）とします。これを行うには、両方とも新しいブロックを割り当てる、このブロックが存在するファイルの i ノードに記録し、新しい大きなサイズを反映するようにファイルのサイズを変更する必要があります。（そのほかのことについては本の第3部で詳しく説明します）割り込みはいつでも発生する可能性があるので、これらの共有構造を更新するコード（たとえば、割り当て用のビットマップまたはファイルの inode）はクリティカルセクションです。したがって、OS 設計者は、割り込みの導入の当初から、OS が内部構造をどのように更新するかについて心配する必要がありました。タイミングの悪い割り込みが上記のすべての問題を引き起こします。当然のことながら、ページテーブル、プロセスリスト、ファイルシステム構造、およびほぼすべてのカーネルデータ構造には、適切な同期プリミティブを使用して、正しく動作するように慎重にアクセスする必要があります。

ASIDE: KEY CONCURRENCY TERMS CRITICAL SECTION, RACE CONDITION, IN-DETERMINATE, MUTUAL EXCLUSION

これらの4つの用語は、並行コードの中核をなすものであり、明示的に呼び出す際には価値があると考えていました。詳細については、Dijkstra の初期の作品 [D65, D68] を参照してください。

1. クリティカルセクションは、共有リソース（通常は変数またはデータ構造）にアクセスするコードです。

2. 競合状態は、実行の複数のスレッドがほぼ同じ時間にクリティカルセクションに入る場合に発生します。両方とも共用データ構造を更新しようとし、驚くべき（そしておそらく望ましくない）結果につながります。

3. 不確定プログラムは、1つまたは複数の競合条件で構成されます。実行時に実行されるスレッドの種類によって、プログラムの出力は実行ごとに異なります。その結果は決定論的ではなく、通常コンピュータシステムから期待されるものです。

4. これらの問題を回避するために、スレッドは何らかの相互排他プリミティブを使用する必要があります。

ります。これにより、1つのスレッドだけがクリティカルセクションに入り、レースを回避し、確定的なプログラム出力を得ることが保証されます。

#参考文献

[D65] “Solution of a problem in concurrent programming control”

E. W. Dijkstra

Communications of the ACM, 8(9):569, September 1965

Pointed to as the first paper of Dijkstra's where he outlines the mutual exclusion problem and a solution. The solution, however, is not widely used; advanced hardware and OS support is needed, as we will see in the coming chapters.

[D68] “Cooperating sequential processes”

Edsger W. Dijkstra, 1968

Available: <http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF>

Dijkstra has an amazing number of his old papers, notes, and thoughts recorded (for posterity) on this website at the last place he worked, the University of Texas. Much of his foundational work, however, was done years earlier while he was at the Technische Hochschule of Eindhoven (THE), including this famous paper on “cooperating sequential processes”, which basically outlines all of the thinking that has to go into writing multi-threaded programs. Dijkstra discovered much of this while working on an operating system named after his school: the “THE” operating system (said “T”, “H”, “E”, and not like the word “the”).

[GR92] “Transaction Processing: Concepts and Techniques”

Jim Gray and Andreas Reuter

Morgan Kaufmann, September 1992

This book is the bible of transaction processing, written by one of the legends of the field, Jim Gray. It is, for this reason, also considered Jim Gray's “brain dump”, in which he wrote down everything he knows about how database management systems work. Sadly, Gray passed away tragically a few years back, and many of us lost a friend and great mentor, including the co-authors of said book, who were lucky enough to interact with Gray during their graduate school years.

[L+93] “Atomic Transactions”

Nancy Lynch, Michael Merritt, William Weihl, Alan Fekete

Morgan Kaufmann, August 1993

A nice text on some of the theory and practice of atomic transactions for distributed systems. Perhaps a bit formal for some, but lots of good material is found herein.

[SR05] “Advanced Programming in the UNIX Environment”

W. Richard Stevens and Stephen A. Rago

Addison-Wesley, 2005

As we've said many times, buy this book, and read it, in little chunks, preferably before going to bed. This way, you will actually fall asleep more quickly; more importantly, you learn a little more about how to become a serious UNIX programmer.

27 Interlude: Thread API

この章では、スレッド API の主要部分について簡単に説明します。各部分については、API の使い方を示すので、以降の章ではさらに詳しく説明します。詳細は、様々な書籍やオンライン情報源 [B89、B97、B+96、K+96] で見つけることができます。以降の章では、ロックや条件変数の概念をよりゆっくりと紹介しています。これらには多くの例があります。したがって、この章は参考文献のように使用してください。

CRUX: HOW TO CREATE AND CONTROL THREADS

スレッドの作成と制御のために OS が提示すべきインターフェースは何ですか？ これらのインターフェイスは、使いやすさとユーティリティ性を実現するためにどのように設計されるべきですか？

27.1 Thread Creation マルチスレッドプログラムを作成するには、最初に新しいスレッドを作成する必要があります。したがって、ある種のスレッド作成インターフェイスが存在する必要があります。POSIX では、簡単です。

```
#include <pthread.h>
int
pthread_create( pthread_t * thread,
                const pthread_attr_t * attr,
                void * (*start_routine)(void*),
                void * arg);
```

この宣言は少し複雑に見えるかもしれません (特に C で関数ポインタを使用していない場合)、実際にはそれほど悪くはありません。スレッド、attr、開始ルーチン、arg の 4 つの引数があります。最初のスレッドは、pthread 型の構造体へのポインタです。この構造体を使ってこのスレッドとやりとりするので、初期化するために `pthread_create()` に渡す必要があります。

2 番目の引数 attr は、このスレッドが持つ可能性のある属性を指定するために使用されます。いくつかの例には、スタックサイズの設定や、おそらくスレッドのスケジューリング優先順位に関する情報が含まれます。属性は、`pthread_attr_init()` を個別に呼び出して初期化されます。詳細については、マニュアルページを参照してください。しかし、ほとんどの場合、デフォルトは正常に動作します。この場合、単に NULL という値を渡します。

3 番目の引数は最も複雑ですが、実際には尋ねています。このスレッドはどの関数で実行されるべきですか？ C では、これを関数ポインタと呼びます。これは、関数名 (開始ルーチン) が void 型の 1 つの引数を渡していることを示しています (開始ルーチンの後にかっこで示されています) void 型の値を返します (つまり、void ポインタ) このルーチンが void ポインタの代わりに整数の引数を必要とする場合、宣言は次のようにになります。

```
int pthread_create(..., // first two args are the same
                  void * (*start_routine)(int),
                  int arg);
```

代わりに、ルーチンが void ポインタを引数として取りますが、整数を返した場合は、次のようにになります。

```
int pthread_create(..., // first two args are the same
int (*start_routine)(void *),
```

```
void * arg);
```

最後に、第 4 引数 arg は、スレッドが実行を開始する関数に渡す引数とまったく同じです。あなたは質問するかもしれません。なぜこれらの void ポインタが必要ですか？さて、答えは非常に簡単です。関数の開始ルーチンの引数として void ポインターを使用すると、任意の型の引数を渡すことができます。それを戻り値として持つと、スレッドはあらゆるタイプの結果を返すことができます。

図 27.1 の例を見てみましょう。ここでは、自分自身を定義する单一の型 (myarg_t) にパッケージ化された 2 つの引数を渡すスレッドを作成します。作成されたスレッドは、その引数を予期した型に単にキャストすることができます。

```

1 #include <pthread.h>
2
3 typedef struct __myarg_t {
4     int a;
5     int b;
6 } myarg_t;
7
8 void *mythread(void *arg) {
9     myarg_t *m = (myarg_t *) arg;
10    printf("%d %d\n", m->a, m->b);
11    return NULL;
12 }
13
14 int
15 main(int argc, char *argv[]) {
16     pthread_t p;
17     int rc;
18
19     myarg_t args;
20     args.a = 10;
21     args.b = 20;
22     rc = pthread_create(&p, NULL, mythread, &args);
23     ...
24 }
```

Figure 27.1: Creating a Thread

それがそこにあります！スレッドを作成すると、プログラム内に現在存在するすべてのスレッドと同じアドレス空間内で実行される独自の呼び出しstackoverflowを持つ別の実行中のエンティティが実際に存在します。こうして楽しいことが始まります！

27.2 Thread Completion

上記の例は、スレッドを作成する方法を示しています。しかし、スレッドが完了するのを待つ場合はどうなりますか？あなたは完了を待つために特別なことをする必要があります。特に、ルーチン `pthread_join()` を呼び出す必要があります。

```
int pthread_join(pthread_t thread, void **value_ptr);
```

このルーチンには 2 つの引数があります。最初の型は `pthread` 型のもので、どのスレッドを待つかを指定す

るために使われます。この変数は、スレッド作成ルーチンによって初期化されます (pthread_create() への引数としてポインタを渡すとき)。あなたがそれを保持しているなら、それを使ってそのスレッドが終了するのを待つことができます。

2番目の引数は、返される戻り値へのポインタです。ルーチンは何も返すことができないので、void へのポインタを返すように定義されています。pthread_join() ルーチンは渡された引数の値を変更するため、値そのものだけでなく、その値へのポインタを渡す必要があります。

別の例を見てみましょう (図 27.2) コードでは、単一のスレッドが再び作成され、myarg 構造体を介して 2つの引数が渡されます。値を返すには、myret_t 型が使用されます。スレッドの実行が終了すると、pthread_join() ルーチン 1 の内部で待機していたメインスレッドが戻り、スレッドから返された値、つまり myret_t の値にアクセスできます。

```

1  #include <stdio.h>
2  #include <pthread.h>
3  #include <assert.h>
4  #include <stdlib.h>
5
6  typedef struct __myarg_t {
7      int a;
8      int b;
9  } myarg_t;
10
11 typedef struct __myret_t {
12     int x;
13     int y;
14 } myret_t;
15
16 void *mythread(void *arg) {
17     myarg_t *m = (myarg_t *) arg;
18     printf("%d %d\n", m->a, m->b);
19     myret_t *r = Malloc(sizeof(myret_t));
20     r->x = 1;
21     r->y = 2;
22     return (void *) r;
23 }
24
25 int
26 main(int argc, char *argv[]) {
27     pthread_t p;
28     myret_t *m;
29
30     myarg_t args = {10, 20};
31     Pthread_create(&p, NULL, mythread, &args);
32     Pthread_join(p, (void **) &m);
33     printf("returned %d %d\n", m->x, m->y);
34     free(m);
35     return 0;
36 }
```

Figure 27.2: Waiting for Thread Completion

この例についていくつか注意してください。まず、この苦しいパッキングとアンパックのすべてを行う必要はありません。たとえば、引数を持たないスレッドを作成するだけであれば、スレッドを作成するときに引数として NULL を渡すことができます。同様に、戻り値を気にしなければ、`pthread_join()` に NULL を渡すことができます。次に、単一の値 (int など) を渡すだけの場合、引数としてパッケージ化する必要はありません。図 27.3 に例を示します。この場合、構造体の中に引数と戻り値をパッケージ化する必要がないので、少しシンプルです。

```

void *mythread(void *arg) {
    int m = (int) arg;
    printf("%d\n", m);
    return (void *) (arg + 1);
}

int main(int argc, char *argv[]) {
    pthread_t p;
    int rc, m;
    Pthread_create(&p, NULL, mythread, (void *) 100);
    Pthread_join(p, (void **) &m);
    printf("returned %d\n", m);
    return 0;
}

```

Figure 27.3: Simpler Argument Passing to a Thread

第 3 に、値がスレッドから返される方法に非常に注意する必要があることに注意してください。特に、スレッドの呼び出しstackoverflow に割り当てられたものを参照するポインタを返すことはありません。そうすれば、どうなると思いますか？ 危険なコードの例を図 27.3 の例から変更しました。

```

1 void *mythread(void *arg) {
2     myarg_t *m = (myarg_t *) arg;
3     printf("%d %d\n", m->a, m->b);
4     myret_t r; // ALLOCATED ON STACK: BAD!
5     r.x = 1;
6     r.y = 2;
7     return (void *) &r;
8 }

```

この場合、変数 `r` は `mythread` のスタックに割り当てられます。しかし、それが返ってくると、値は自動的に割り当てが解除されます (そのため、スタックは使いやすくなります) ので、現在割り当てられていない変数にポインタを戻すと、あらゆる種類の悪い結果につながります。確かに、あなたが返すと思った値をプリントアウトすると、おそらく (必ずしもそうではありませんが) 驚くことでしょう。それを試してみてください！

最後に、`pthread_create()` を使用してスレッドを作成し、それに続いて `pthread_join()` をすぐに呼び出すと、スレッドを作成するのは非常に奇妙な方法です。実際、この正確なタスクを達成するためのより簡単な方法があります。プロシージャコールと呼ばれます。明らかに、私たちは通常、1 つ以上のスレッドを作成し、それが完了するのを待っています。そうでなければ、スレッドをまったく使用する目的はありません。

マルチスレッドのすべてのコードが結合ルーチンを使用するわけではないことに注意してください。たとえば、マルチスレッド Web サーバーでは、多数のワーカースレッドが作成され、メインスレッドを使用して要

求を受け入れ、それらをワーカーに無期限に渡すことがあります。このように長寿命のプログラムは必要がないかかもしれません。しかし、特定のタスクを(並列に)実行するスレッドを作成する並列プログラムは、joinを使用して、そのようなすべての作業が終了して次の計算ステージに移る前に完了するようになります。

27.3 Locks

スレッドの作成と結合以外にも、おそらくPOSIXスレッドライブラリによって提供される次の最も有用な関数群は、ロックを介してクリティカルセクションに相互排除を提供するものです。この目的のために使用する最も基本的なルーチンのペアは、次のものによって提供されます。

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

ルーチンは理解しやすく使いやすいものでなければなりません。クリティカルセクションであるコード領域があり、正しい操作を保証するために保護する必要がある場合、ロックは非常に便利です。

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

コードの目的は次のとおりです。`pthread_mutex_lock()`が呼び出されたときに他のスレッドがロックを保持しない場合、スレッドはロックを取得してクリティカルセクションに入ります。別のスレッドが実際にロックを保持している場合、ロックを取得しようとするスレッドは、ロックを取得するまで(ロックを保持しているスレッドがロック解除呼び出しによって解除したことを意味します)もちろん、多くのスレッドは、ロック取得関数内で所定の時間待機している可能性があります。ただし、ロックを取得したスレッドのみがunlockを呼び出す必要があります。

残念ながら、このコードは2つの重要な問題があります。最初の問題は、適切な初期化の欠如です。すべてのロックは、適切な値を持っていることを保証するために、適切に初期化されなければなりません。

POSIXスレッドでは、ロックを初期化する2つの方法があります。これを行う1つの方法は、`PTHREAD_MUTEX_INITIALIZER`を次のように使用することです。

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

これを行うと、ロックがデフォルト値に設定され、ロックが使用可能になります。二つ目の方法は、これを実行する動的な方法(実行時)は、次のように`pthread_mutex_init()`を呼び出すことです。

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

このルーチンの最初の引数はロック自体のアドレスですが、2番目の引数はオプションの属性のセットです。単にNULLを渡すと、デフォルトが使用されます。いずれの方法でも動作しますが、通常は動的(後者)の方法を使用します。`pthread_mutex_destroy()`への対応する呼び出しへは、ロックが完了したときにも行われることに注意してください。すべての詳細については、マニュアルページを参照してください。

上記のコードの2番目の問題は、ロックとロック解除を呼び出すときにエラーコードをチェックできないことです。UNIXシステムで呼び出すほとんどすべてのライブラリルーチンと同様に、これらのルーチンも失

敗する可能性があります。コードでエラーコードが正しくチェックされないと、エラーが発生します。この場合、複数のスレッドがクリティカルセクションに入る可能性があります。最小限には、ルーチンが成功したことと主張するラッパーを使用します(たとえば、図 27.4 のように)。何かがうまくいかないときに、洗練されたプログラムの場合は、失敗をチェックして、ロックまたはロック解除が成功しないときに適切な何かを行うべきです。

```
// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_lock(mutex);
    assert(rc == 0);
}
```

Figure 27.4: An Example Wrapper

ロックとアンロックルーチンは、`pthreads` ライブラリ内のロックと対話する唯一のルーチンではありません。特に興味のあるルーチンが 2 つあります。

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                           struct timespec *abs_timeout);
```

これら 2 つの呼び出しは、ロック取得に使用されます。ロックがすでに保持されている場合、`trylock` は失敗を返します。一方、`timelock` は、指定した時間までのロックの試行を試みます。したがって、タイムアウトが 0 の `timelock` は、`trylock` と同じになってしまいます。これらのバージョンはどちらも一般的に避けなければなりません。しかし、将来の章では(例えば、デッドロックを調べるときなど)、ロック獲得ルーチンに突っ込まれることを避けること(おそらく無期限に)を避けることが有用な場合があります。

27.4 Condition Variables

スレッドライブラリのもう 1 つの主要なコンポーネントであり、確かに POSIX スレッドの場合は、条件変数が存在します。条件変数は、あるスレッドが別のスレッドが続行する前に何かを実行するのを待っている場合に、何らかの種類のシグナルがスレッド間で行われなければならない場合に便利です。このように対話したいプログラムでは、2 つの主要ルーチンが使用されます。

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

条件変数を使用するには、この条件に関連付けられたロックがさらに必要です。上記のいずれかのルーチンを呼び出すときは、このロックを保持する必要があります。

最初のルーチンである `pthread_cond_wait()` は、呼び出し元のスレッドをスリープ状態にして、通常はスリープ中のスレッドが気にするプログラムの何かが変更されたときに、他のスレッドがそれを通知するのを待ちます。典型的な使用法は次のようになります。

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
Pthread_mutex_lock(&lock);
while (ready == 0)
```

```
Pthread_cond_wait(&cond, &lock);
Pthread_mutex_unlock(&lock);
```

このコードでは、関連するロックおよび条件の初期化後、スレッドは、変数 ready がまだゼロ以外の値に設定されているかどうかを確認します。そうでなければ、スレッドは他のスレッドが起動するまでスリープするために単に wait ルーチンを呼び出します。他のスレッドで実行されるスレッドを起動するコードは次のようにになります。

```
Pthread_mutex_lock(&lock);
ready = 1;
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&lock);
```

このコードシーケンスについていくつか注意してください。まず、シグナリング（グローバル変数の準備を変更する場合と同様）時には、常にロックを保持するようにします。これにより、誤ってコードに競合状態が導入されることはありません。

第 2 に、待機コールが第 2 パラメータとしてロックを取ることに気付くかもしれないが、信号コールは条件を取るだけです。この違いの理由は、呼び出し元のスレッドをスリープ状態にすることに加えて、待機呼び出しが、呼び出し元をスリープ状態にするときにロックを解放するためです。もしそうでなければ、他のスレッドがロックを取得して目覚めるように通知する方法はありますか？ しかし、ウォッチした後に戻る前に、`pthread_cond_wait()` はロックを再取得します。したがって、待機中のスレッドが待機シーケンスの開始時に獲得されたロックと最後のロック解除の間で実行されている間、それはロックを保持します。

待機中のスレッドは、単純な `if` 文ではなく、`while` ループで条件を再チェックします。この章では、将来の章で条件変数を学習するときにこの問題について詳しく説明しますが、一般的に `while` ループを使うのは簡単で安全な方法です。条件を再チェックしますが（おそらく少しのオーバーヘッドを追加します）、待っているスレッドを擬似的に起動させるいくつかの `pthread` 実装があります。このような場合、再チェックを行わずに、待機中のスレッドは、条件が変更されていないにもかかわらず変更されたとみなし続けます。したがって、絶対的な事実ではなく、何かが変化したかもしれないというヒントとして目を覚ますことは、より安全です。

条件変数と関連するロックの代わりに、単純なフラグを使用して 2 つのスレッド間で信号を送ることが魅力的であることに注意してください。たとえば、待機コードでは、上記の待機コードを次のように書き換えることができます。

```
while (ready == 0)
; // spin
```

関連するシグナリングコードは次のようにになります。

```
ready = 1;
```

次のような理由からこれをしないでください。まず、多くの場合、パフォーマンスが低下します（CPU サイクルを浪費するだけの長時間の回転）。第二に、エラーが起こりやすい。最近の研究で [X+10] と表示されているように、フラグを使って（上記のように）スレッド間の同期をとると間違いを犯すのは驚くほど簡単です。その調査では、これらのアドホックな同期の使用の約半分がバグでした！ たとえ、そうしなくてもできると思うときでさえ、条件変数を使用してください。

条件変数が混乱して聞こえる場合は、あまり心配する必要はありません（まだ）ので、後の章で詳しく説明し

ます。それまでは、それらが存在することを知り、どのように、なぜそれらが使用されているかを知ることで十分です。

27.5 Compiling and Running

この章のすべてのコード例は、起動して実行するのが比較的簡単です。それらをコンパイルするには、コードにヘッダ pthread.h を含める必要があります。リンクの行では、-pthread フラグを追加して、pthreads ライブライバーと明示的にリンクする必要があります。たとえば、単純なマルチスレッドプログラムをコンパイルするには、次の操作が必要です。

```
prompt> gcc -o main main.c -Wall -pthread
```

main.c に pthreads ヘッダーが含まれている限り、並行プログラムを正常にコンパイルしました。ただし、この方法でいつものように、コンパイルがうまくいくかどうかは、まったく別の問題です。

27.6 Summary

スレッドの作成、ロックによる相互排他の構築、条件変数によるシグナル伝達と待機を含む、pthread ライブライバーの基礎を紹介しました。あなたは、忍耐と大切なケアを除いて、堅牢で効率的なマルチスレッドコードを書くのに他に多くのものを必要としません！

この章では、マルチスレッドコードを記述するときに役立つヒントをまとめています（詳細は次のページを参照してください）興味深い API の他の側面があります。より多くの情報が必要な場合は、Linux システムで man -k pthread と入力して、インターフェース全体を構成する 100 以上の API を確認してください。ただし、ここで説明する基本的な機能は、洗練された（うまくいけば、正しい、パフォーマンスの高い）マルチスレッドプログラムを構築できるようにする必要があります。スレッドを持つ難しい部分は API ではなく、並行プログラムをどのように構築するかの難しい論理です。詳細は以下をお読みください。

ASIDE: THREAD API GUIDELINES

POSIX スレッドライブラリ（または実際には任意のスレッドライブラリ）を使用してマルチスレッドプログラムを構築する際には、覚えておくべき重要なことはいくつかあります。

- 単純にする。何よりも、スレッド間のロックやシグナルのコードはできるだけシンプルでなければなりません。トリッキーなスレッドのやり取りはバグにつながります。
- スレッドのやりとりを最小限に抑えます。スレッドが相互作用する方法の数を最小限に保つようにしてください。それぞれの相互作用は、慎重に考察し、試練された真のアプローチで構築されるべきです（その多くは、今後の章で学ぶ）。
- ロックと条件変数を初期化する。そうしないと、時にはうまく動作しないことがあります、時には非常に奇妙な方法で失敗することがあります。
- リターンコードを確認します。もちろん、どのような C や UNIX プログラミングでも、それぞれのリターンコードをチェックする必要があります。ここでもそうです。そうでなければ行動が分かりにくくなり、(a) 悲鳴を上げる、(b) 髪の毛を抜く、(c) 両方をする可能性が高くなります。
- スレッドに引数を渡したり、スレッドから値を返す方法には注意してください。特に、スタックに割り当てられた変数への参照を渡すときは、おそらく何か間違っているでしょう。
- 各スレッドには独自のスタックがあります。上記の点に関連して、各スレッドには独自のスタックがあることに注意してください。したがって、あるスレッドが実行している関数の中にローカルに割り当てられた変数がある場合、それはそのスレッドにとって本質的にプライベートです。他のスレッドはそれに（簡単に）アクセスできません。スレッド間でデータを共有するには、値がヒー

アにあるか、またはグローバルにアクセス可能なロケールでなければなりません。

- 常にスレッド間で信号を送るには、条件変数を使用します。それはしばしば単純なフラグを使用することを魅力的ですが、しないでください。
- マニュアルページを使用してください。Linux では、特に、pthread の man ページは非常に有益であり、ここで紹介したニュアンスの多くについて、さらに詳細に議論します。注意深く読んでください！

参考文献

[B89] “An Introduction to Programming with Threads”

Andrew D. Birrell

DEC Technical Report, January, 1989

Available: <https://birrell.org/andrew/papers/035-Threads.pdf>

A classic but older introduction to threaded programming. Still a worthwhile read, and freely available.

[B97] “Programming with POSIX Threads”

David R. Butenhof

Addison-Wesley, May 1997

Another one of these books on threads.

[B+96] “PThreads Programming: A POSIX Standard for Better Multiprocessing”

Dick Buttlar, Jacqueline Farrell, Bradford Nichols

O'Reilly, September 1996

A reasonable book from the excellent, practical publishing house O'Reilly. Our bookshelves certainly contain a great deal of books from this company, including some excellent offerings on Perl, Python, and Javascript (particularly Crockford's “Javascript: The Good Parts”).

[K+96] “Programming With Threads”

Steve Kleiman, Devang Shah, Bart Smaalders

Prentice Hall, January 1996

Probably one of the better books in this space. Get it at your local library. Or steal it from your mother.

More seriously, just ask your mother for it – she'll let you borrow it, don't worry.

[X+10] “Ad Hoc Synchronization Considered Harmful”

Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, Zhiqiang Ma

OSDI 2010, Vancouver, Canada

This paper shows how seemingly simple synchronization code can lead to a surprising number of bugs. Use condition variables and do the signaling correctly!

28 Locks

並行処理の導入から同時プログラミングの基本的な問題の1つがわかりました。一連の命令を原子性で実行したいのですが、単一プロセッサ(または複数のプロセッサで同時に実行される複数のスレッド)に割り込みがあるため、私たちはできませんでした。この章では、ロックと呼ばれるものを導入して、この問題を直接攻撃しています。プログラマは、ソースコードにロックを付けてクリティカルセクションの周りに置くことで、そのようなクリティカルセクションが単一のアトミック命令であるかのように実行するようにします。

28.1 Locks: The Basic Idea

例として、クリティカルセクションが共有変数の標準的な更新であるこのように見えると仮定します。

```
balance = balance + 1;
```

もちろん、リンクリストに要素を追加するなど、他の重要なセクションも可能ですが、ここではこの簡単な例を続けます。ロックを使用するには、次のようにクリティカルセクションの周りにいくつかのコードを追加します。

```
1 lock_t mutex; // some globally-allocated lock 'mutex'  
2 ...  
3 lock(&mutex);  
4 balance = balance + 1;  
5 unlock(&mutex);
```

ロックは单なる変数であるため、ロックを使用するには、何らかのロック変数(上記の mutex など)を宣言する必要があります。このロック変数(または単に「ロック」と略記)は、任意の瞬間にロックの状態を保持します。これは使用可能(またはロック解除またはフリー)なので、スレッドがロックを保持していないか、または取得(またはロックまたは保持)されていないため、ちょうど1つのスレッドがロックを保持しており、ロックを保持しているスレッドや、ロック獲得を注文するためのキューなど、他の情報もデータ型に格納できますが、そのような情報はロックのユーザーからは隠されています。

`lock()` と `unlock()` ルーチンの中身は単純です。ルーチン `lock()` を呼び出すと、ロックを取得しようとします。他のスレッドがロックを保持していない(すなわち解放している)場合、スレッドはロックを取得してクリティカルセクションに入ります。このスレッドはロックの所有者と呼ばれることがあります。別のスレッドが、同じロック変数(この例では mutex)の `lock()` を呼び出すと、ロックが別のスレッドによって保持されている間は戻りません。このようにして、ロックを保持している最初のスレッドがそこにある間に、他のスレッドがクリティカルセクションに入ることが防止されます。

ロックの所有者が `unlock()` を呼び出すと、ロックは再び使用可能(空き)になります。他のスレッドがロックを待っていない場合(すなわち、他のスレッドが `lock()` を呼び出していない場合)、ロックの状態は単にフリーに変更されます。待機スレッド(`lock()` にスタック)がある場合、そのうちの1つは、ロック状態のこの変更を通知(または通知)し、ロックを取得し、クリティカルセクションに入ります。

ロックは、プログラマーにスケジューリングするための制御を最小限に抑えます。一般に、スレッドはプログラマによって作成されたエンティティとして認識されますが、様々な方法で OS によってスケジュールされます。ロックはその制御の一部をプログラマーに返します。コードの一部にロックをかけることで、プログラマーはそのコード内で単一のスレッドしかアクティブにならないことを保証することができます。したがつ

て、ロックは、伝統的な OS スケジューリングであるカオスをより制御されたアクティビティーに変換するのに役立ちます。

28.2 Pthread Locks

ロックのために POSIX ライブラリが使用する名前は、スレッド間の相互排他を提供するために使用される mutex です。つまり、あるスレッドがクリティカルセクションにある場合、他のスレッドはセクションを完了するまで入力を排除します。したがって、次の POSIX スレッドコードを参照すると、上記と同じことが行われていることを理解する必要があります（ロックとロック解除時にエラーをチェックするラッパーも使用します）。

```

1 pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2
3 Pthread_mutex_lock(&lock); // wrapper for pthread_mutex_lock()
4 balance = balance + 1;
5 Pthread_mutex_unlock(&lock);

```

また、異なる変数を保護するために異なるロックを使用している可能性があるため、POSIX バージョンでは変数をロックおよびロック解除することに注意してください。これにより、並行性が向上します。クリティカルセクションにアクセスするたびに使用される 1 つの大きなロック（粗いロック戦略）の代わりに、異なるデータとデータ構造を異なるロックで保護することが多くなります。したがって、一度に多くのスレッドをロックされたコードにすることができます。

28.3 Building A Lock

ここまでで、プログラマの視点から、ロックがどのように機能するかを理解しておく必要があります。しかし、どのようにロックを構築する必要がありますか？どのハードウェアサポートが必要ですか？どの OS がサポートしていますか？この章の残りの部分で取り上げるのは、この一連の質問です。

THE CRUX: HOW TO BUILD A LOCK

どのようにして効率的なロックを構築できますか？効率的なロックは低コストで相互排除を提供し、また以下で議論するいくつかの他の特性を達成するかもしれない。どのハードウェアサポートが必要ですか？どの OS がサポートしていますか？

動くロックを構築するためには、私たちの昔の友人、ハードウェア、私たちの良い仲間、OS から何らかの助けが必要です。長年にわたり、さまざまなコンピューターアーキテクチャの命令セットに、いくつかの異なるハードウェアプリミティブが追加されました。私たちはこれらの命令がどのように実装されているのかを研究するつもりはないですが（結局のところ、コンピュータ・アーキテクチャ・クラスの話題である）、それらを使用してロックのような相互排他プリミティブを構築する方法を研究します。また、OS がどのように関係してロックを完成させ、洗練されたロックライブラリを構築できるようにするかについても検討します。

28.4 Evaluating Locks

ロックを構築する前に、まず目標が何であるかを理解し、特定のロック実装の有効性を評価する方法を尋ねる必要があります。ロックが機能するかどうかを評価するには、まず基本的な基準を確立する必要があります。最初は、ロックが基本的な作業を行うかどうかです。これは相互排除を提供することです。基本的には、ロッ

クが機能し、複数のスレッドがクリティカルセクションに入るのを防ぎますか？

第二は公平です。ロックを獲得しようとしている各スレッドは、一度解放されるとすぐに獲得することができますか？ これを見るもう1つの方法は、より極端なケースを調べることです。ロックを競合するスレッドは、その間に飢えてしまい、二度と取得できないでしょうか？

最終的な基準はパフォーマンスです。具体的には、ロックを使用して追加される時間オーバーヘッドです。ここで考慮する価値のあるいくつかの異なるケースがあります。1つは競合がない場合です。単一のスレッドが実行され、ロックを取得して解放するとき、そのようなオーバーヘッドは何ですか？ もう1つは、複数のスレッドが1つのCPU上のロックに対して競合している場合です。この場合、パフォーマンスの問題はありますか？ 最後に、複数のCPUが関わっているときにロックがどのように動作し、それぞれがロックを競合しますか？ これらのさまざまなシナリオを比較することで、以下で説明するように、さまざまなロック手法を使用することによるパフォーマンスの影響をよりよく理解できます。

28.5 Controlling Interrupts

相互排除を提供するために使用された最も初期の解決策の1つは、クリティカルセクションの割り込みを無効にすることでした。この解決策は、シングルプロセッサシステム向けに開発されました。コードは次のようにになります。

```

1 void lock() {
2     DisableInterrupts();
3 }
4 void unlock() {
5     EnableInterrupts();
6 }
```

このようなシングルプロセッサシステムで実行していると仮定します。クリティカルセクションに入る前に、ある種の特別なハードウェア命令を使用して割り込みをオフにすることで、クリティカルセクション内のコードが中断されないようにし、アトミックであるかのように実行します。終了すると、(ハードウェア命令を介して) 割り込みを再度有効にして、プログラムを通常どおり実行します。

このアプローチの主な点は、単純さです。あなたは確かに、なぜこれが動作するかを理解するためにあなたの頭を悩ます必要はありません。中断することなく、スレッドは、実行するコードが実行され、他のスレッドがそのスレッドに干渉しないことを確実にすることができます。

残念ながら、欠点は多くあります。第1に、このアプローチでは、呼び出しスレッドが特権操作(割り込みのオンとオフを切り替える)を実行できるようにする必要があるため、この機能が悪用されていないことを信頼する必要があります。ご存知のように、任意のプログラムを信頼する必要がある場合は、おそらく問題に陥っている可能性があります。ここで、問題は数多くの形で現れます。欲張りなプログラムは、実行開始時にlock()を呼び出してプロセッサを独占することができます。悪質なプログラムや悪意のあるプログラムがlock()を呼び出して無限ループに陥る可能性があります。この後者の場合、OSはシステムの制御を元に戻すことはありません。解決する唯一の手段はシステムを再起動する方法です。汎用の同期ソリューションとして割り込みを無効にするには、アプリケーションをあまりにも多くの信頼を必要とします。

第2に、このアプローチはマルチプロセッサでは機能しません。複数のスレッドが異なるCPU上で実行されており、それぞれが同じクリティカルセクションを入力しようとすると、割り込みが無効かどうかは関係ありません。スレッドは他のプロセッサ上で実行できるため、クリティカルセクションに入る可能性があります。マルチプロセッサが普及している現在、我々の一般的な解決策はこれよりも優れていなければなりません。

第3に、割り込みを長時間オフにすると、割り込みが失われ、重大なシステムの問題が発生する可能性があります。たとえば、ディスクデバイスが読み取り要求を完了したという事実をCPUが逃したとします。OSは、この読み込みを待っているプロセスを復帰させる方法をどのように知っていますか？

最後に、おそらく最も重要はほんないです、このアプローチは非効率的である可能性があります。通常の命令実行と比較して、割り込みをマスクまたはアンマスクするコードは、最新のCPUによってゆっくり実行される傾向があります。これらの理由から、割り込みを無効にすることは、相互排除プリミティブとして限定されたコンテキストでのみ使用されます。たとえば、オペレーティングシステム自体が、自身のデータ構造にアクセスするとき、または少なくとも混乱した割り込み処理状況が発生しないようにするために、原子性を保証するために割り込みマスクを使用する場合があります。この使用法は理にかなっています。なぜなら、信頼問題がOS内部で消滅するためです。OSは、常に特権操作を実行することを常に信頼しています。

28.6 A Failed Attempt: Just Using Loads/Stores

割り込みベースの技術を超えて移動するためには、CPUハードウェアと適切なロックを構築するために提供される命令に依存する必要があります。最初に単一のフラグ変数を使用して単純なロックを構築しようとしたまゝ。この失敗した試みでは、ロックを構築するために必要ないくつかの基本的なアイデアが表示されます。なぜなら、単一の変数を使用し、通常のロードおよびストアを介してアクセスするだけでは不十分である理由が分かります。

この最初の試み(図28.1)では、アイデアは非常に単純です。単純な変数(フラグ)を使用して、あるスレッドがロックを所有しているかどうかを示します。クリティカルセクションに入る最初のスレッドは、フラグが1(この場合はそうではない)かどうかをテストするlock()を呼び出し、スレッドが現在ロックを保持していることを示すフラグを1に設定します。クリティカルセクションが終了すると、スレッドはunlock()を呼び出し、フラグをクリアして、ロックがもはや保持されていないことを示します。

```

1  typedef struct __lock_t { int flag; } lock_t;
2
3  void init(lock_t *mutex) {
4      // 0 -> lock is available, 1 -> held
5      mutex->flag = 0;
6  }
7
8  void lock(lock_t *mutex) {
9      while (mutex->flag == 1) // TEST the flag
10         ; // spin-wait (do nothing)
11     mutex->flag = 1;        // now SET it!
12 }
13
14 void unlock(lock_t *mutex) {
15     mutex->flag = 0;
16 }
```

Figure 28.1: First Attempt: A Simple Flag

最初のスレッドがクリティカルセクションにある間に別のスレッドがlock()を呼び出すと、そのスレッドがunlock()を呼び出してフラグをクリアするwhileループでスピン待機します。その最初のスレッドがそうすると、待機中のスレッドはwhileループから抜け出し、フラグを1に設定してクリティカルセクションに進みます。

残念ながら、コードには2つの問題があります。1つは正当性、もう1つはパフォーマンスです。正当性の

問題は、一度コンカレント・プログラミングを考えるのに慣れれば、簡単に理解できます。図 28.2 のコードインタリーブを想像してみてください。開始するには `flag = 0` とします。このインタリーブからわかるように、タイムリーな（もしくは、タイムリーではなくても）割り込みでは、両方のスレッドがフラグを 1 に設定し、両方のスレッドがクリティカルセクションに入ることができるケースを簡単に生成できます。この振る舞いは、専門家が「悪い」と呼ぶものであり、相互排除を提供するという最も基本的な要件を明らかに満たしていないません。

パフォーマンス問題は、後で詳しく説明しますが、スレッドが既に保持されているロックを取得するために待機する方法です。つまり、フラグの値を無期限にチェックするという方法です。スピン待機は、別のスレッドがロックを解放するのを待って時間を浪費します。ウェイターとして待っているスレッドは実行できません（少なくともコンテキスト切り替えが発生するまで）、ユニプロセッサー上の廃棄は非常に高いです！ したがって、より洗練されたソリューションを開発する際には、このような無駄を回避する方法も考慮する必要があります。

Thread 1	Thread 2
<code>call lock ()</code>	
<code>while (flag == 1)</code>	
<code>interrupt: switch to Thread 2</code>	
	<code>call lock ()</code>
	<code>while (flag == 1)</code>
	<code>flag = 1;</code>
	<code>interrupt: switch to Thread 1</code>
<code>flag = 1; // set flag to 1 (too!)</code>	

Figure 28.2: Trace: No Mutual Exclusion

28.7 Building Working Spin Locks with Test-And-Set

割り込みを無効にすることは複数のプロセッサでは機能しないため、ロードとストアを使用する簡単な方法（前述）が機能しないため、システム設計者はハードウェアによるロックのサポートを開始しました。1960 年代初期の Burroughs B5000 [M82] のような最も初期のマルチプロセッサシステムでは、このようなサポートがありました。今日では、単一の CPU システムであっても、すべてのシステムがこのタイプのサポートを提供しています。理解するための最も簡単なハードウェアサポートは、アトミック交換とも呼ばれるテストアンドセット命令として知られています。以下の C コードスニペットを使って、テスト・アンド・セット命令の動作を定義します。

```

1 int TestAndSet(int *old_ptr, int new) {
2     int old = *old_ptr; // fetch old value at old_ptr
3     *old_ptr = new; // store 'new' into old_ptr
4     return old; // return the old value
5 }
```

テスト・アンド・セット命令が行うことは、次のとおりです。ptr が指す古い値を返し、同時にその値を新しい値に更新します。ここで重要な点は、この一連の操作が原子的に実行されることです。“テストと設定”と呼ばれる理由は、古い値（返される値）を“テスト”しながら同時にメモリ位置を新しい値に“設定”できるよ

うにすることです。実際には、図 28.3 で検討しているように、このわずかに強力な説明は単純なスピンロックを構築するのに十分です。

ASIDE: DEKKER'S AND PETERSON'S ALGORITHMS

1960 年代、ダイクストラは彼の友人に同時性問題を提起し、そのうちの 1 人、Theodorus Jozef Dekker という数学者が解決策を思いついた [D68]。特別なハードウェア命令と OS サポートを使用することで説明するソリューションとは異なり、Dekker のアルゴリズムはロードとストアだけを使用します（ハードウェアが初期のハードウェアでは不可分であったとみなします）。デッカーのアプローチは後に Peterson [P81] によって洗練されました。再び、ロードとストアだけが使用され、2 つのスレッドがクリティカルセクションに同時に入らないようにするために考えられます。ここに Peterson のアルゴリズム（2 つのスレッド用）があります。コードを理解できるかどうかを確認してください。フラグとターン変数は何のために使われますか？

```
int flag[2];
int turn;

void init() {
    flag[0] = flag[1] = 0; // 1->thread wants to grab lock
    turn = 0; // whose turn? (thread 0 or 1?)
}

void lock() {
    flag[self] = 1; // self: thread ID of caller
    turn = 1 - self; // make it other thread's turn
    while ((flag[1-self] == 1) && (turn == 1 - self))
        ; // spin-wait
}

void unlock() {
    flag[self] = 0; // simply undo your intent
}
```

なんらかの理由で、特別なハードウェアサポートなしで動作するロックを開発することは、しばらくの間盛り上がり、理論型に多くの問題を取り組んでいます。もちろん、この作業ラインは、ハードウェアサポートを少しでも簡単に実現できると気づいたときには無用になりました（実際、サポートはマルチプロセッシングの初期から行われていました）。さらに、上記のようなアルゴリズムは、（緩和されたメモリー貫性モデルのため）現代的なハードウェアでは機能しないため、あまり有用ではありません。つまり、より多くの研究は歴史のゴミ箱に…

```

1  typedef struct __lock_t {
2      int flag;
3  } lock_t;
4
5  void init(lock_t *lock) {
6      // 0 indicates that lock is available, 1 that it is held
7      lock->flag = 0;
8  }
9
10 void lock(lock_t *lock) {
11     while (TestAndSet(&lock->flag, 1) == 1)
12         ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }

```

Figure 28.3: A Simple Spin Lock Using Test-and-set

なぜこのロックが動作するのかを理解してみましょう。最初にスレッドが `lock()` を呼び出し、現在他のスレッドがロックを保持していない場合を想像してください。したがって、フラグは 0 になります。スレッドが `TestAndSet(flag,1)` を呼び出すと、ルーチンは `flag` の古い値を返します。これは 0 です。したがって、`flag` の値をテストしている呼び出し元のスレッドは、while ループでスピンを捕らえることはなく、ロックを取得します。スレッドはアトミックに値を 1 に設定し、ロックが現在保持されていることを示します。スレッドがクリティカルセクションで終了すると、`unlock()` を呼び出してフラグをゼロに戻します。

1 つのスレッドが既にロックを保持している（すなわち、フラグが 1 である）ときに、想像できる第 2 の場合が生じます。この場合、このスレッドは `lock()` を呼び出し、`TestAndSet(flag, 1)` も呼び出します。今度は、`TestAndSet()` はフラグを 1 に戻し（ロックが保持されているため）、再び 1 に設定します。ロックが別のスレッドによって保持されている限り、`TestAndSet()` は 1 を繰り返し返します。したがって、このスレッドは、ロックが最後に解放されるまで回転して回転します。フラグが最後に他のスレッドによって 0 に設定されると、このスレッドは再び `TestAndSet()` を呼び出します。これは、値を 1 に原子的に設定している間に 0 を返し、ロックを取得しクリティカルセクションに入ります。

（以前のロック値）のテストと新しい値のセットの両方を 1 つのアトミック操作にすることで、1 つのスレッドだけがロックを取得するようにします。それで、相互排除の基本原則を構築する方法です！

このタイプのロックを通常スピンロックといいます。構築する最も単純なタイプのロックであり、ロックが利用可能になるまで CPU サイクルを使用して単純に回転します。单一のプロセッサ上で正しく動作するためには、プリエンプティブスケジューラ（すなわち、時々異なるスレッドを実行するために、タイマを介してスレッドを中断するスケジューラ）が必要である。プリエンプションなしでは、スピンロックは単一の CPU ではありません。CPU 上のスレッドが決してそれを放棄しないためです。

28.8 Evaluating Spin Locks

基本的なスピンロックが与えられているので、これまで説明した軸に沿ってどれだけ効果的かを評価することができます。ロックの最も重要な側面は正しさです。それは相互排除を提供しますか？ ここでの答えは「はい」です。スピンロックでは、一度に 1 つのスレッドだけがクリティカルセクションに入ることができます。したがって、我々は正しいロックを持っています。

TIP: THINK ABOUT CONCURRENCY AS MALICIOUS SCHEDULER

この例から、同時実行を理解するために必要なアプローチを理解することができます。何をしようとするべきなのは、あなたが悪意のある人物になってみることです。同期プリミティブをビルドする際の微妙な試行錯誤を避けるために、スレッドを最も不適切なタイミングで中断します。あなたはスケジューラなのですか？ 割り込みの正確なシーケンスは不可能かもしれません、それは可能であり、特定のアプローチが機能しないことを実証する必要があります。それは悪意を持って考えるのに役立ちます！（少なくとも、時々）

次の軸は公平です。待機スレッドへのスピンドルはどれくらい公正ですか？ 待っているスレッドがクリティカルセクションに入ることを保証できますか？ 残念ながら、ここでの答えは悪いニュースです。スピンドルは公正を保証しません。実際、糸の回転は競合の下で永遠に回転することがあります。単純なスピンドル（これまで説明したように）は公平ではなく、飢餓につながる可能性があります。

最終的な軸はパフォーマンスです。スピンドルを使用するコストはいくらですか？ これをより慎重に分析するには、いくつかの異なるケースについて考えることをお勧めします。最初は、単一のプロセッサ上でロックを競合するスレッドを想像してください。2番目の方法では、スレッドが多くCPUに分散されていることを考慮してください。

スピンドルの場合、シングルCPUの場合、パフォーマンスのオーバーヘッドは非常に苦しいことがあります。ロックを保持しているスレッドがクリティカルセクション内で先取りされている場合を想像してください。その後、スケジューラは、他のすべてのスレッド（N-1個の他のスレッドがあると想像してください）を実行し、それぞれがロックを取得しようとします。この場合、それらのスレッドはそれぞれ、CPUを諦める前にタイムスライスの期間スピンし、CPUサイクルが浪費されます。

ただし、複数のCPUでは、スピンドルは正常に機能します（スレッド数がCPU数にほぼ等しい場合）。思考は次のようになります。スレッドAをCPU1に、スレッドBをCPU2上に想像してください。どちらもロックを競合しています。スレッドA(CPU1)がロックを取得し、スレッドBがそれを試みると、Bは(CPU2上)スピンします。しかし、おそらくクリティカルセクションが短く、すぐにロックが利用可能になり、スレッドBによって取得されます。他のプロセッサで保持されているロックを待つためにスピンすることは、この場合には多くのサイクルを無駄にしないので効果的です。

28.9 Compare-And-Swap

いくつかのシステムが提供するもう1つのハードウェアプリミティブは、compare-and-swap命令（たとえばSPARC上で呼び出される命令）、またはcompare-and-exchange(x86上で呼び出される）と呼ばれます。この単一命令のC擬似コードは、図28.4にあります。基本的な考え方とは、ptrで指定されたアドレスの値が期待どおりであるかどうかをテストするための比較とスワップです。その場合は、ptrが指示するメモリ位置を新しい値で更新します。そうでない場合は、何もしないでください。どちらの場合でも、実際の値をそのメモリ位置に戻すことで、compare-and-swapを呼び出すコードが成功したかどうかを知ることができます。compare-and-swap命令を使用すると、テストセットの場合と非常によく似た方法でロックを構築できます。たとえば、上記のlock()ルーチンを次のものに置き換えることができます。

```

1 int CompareAndSwap(int *ptr, int expected, int new) {
2     int actual = *ptr;
3     if (actual == expected)
4         *ptr = new;
5     return actual;
6 }
```

Figure 28.4: Compare-and-swap

```

1 void lock(lock_t *lock) {
2     while (CompareAndSwap(&lock->flag, 0, 1) == 1)
3         ; // spin
4 }
```

コードの残りの部分は上記のテストセットの例と同じです。このコードは全く同じように動作します。フラグが 0 であるかどうかを単純にチェックし、そうであれば、アトミックに 1に入れ替えてロックを取得します。ロックが保持されている間にロックを取得しようとするスレッドは、ロックが最後に解放されるまで回転を停止します。compare-and-swap の C 呼び出し可能な x86 版を実際に作成する方法を知りたい場合は、このコードシーケンスが便利です ([S05] から)

```

1 char CompareAndSwap(int *ptr, int old, int new) {
2     unsigned char ret;
3
4     // Note that sete sets a 'byte' not the word
5     __asm__ __volatile__ (
6         " lock\n"
7         " cmpxchgl %2,%1\n"
8         " sete %0\n"
9         : "=q" (ret), "=m" (*ptr)
10        : "r" (new), "m" (*ptr), "a" (old)
11        : "memory");
12     return ret;
13 }
```

最後に、感知したように、比較とスワップはテストとセットより強力な命令です。ロックフリー同期 [H91]などのトピックを簡単に掘り下げて、今後このパワーを活用していく予定です。しかし、単純なスピノロックを作成するだけでは、その動作は上記で分析したスピノロックと同じです。

28.10 Load-Linked and Store-Conditional

プラットフォームによっては、クリティカルセクションを作成するのに役立つ一対の命令が用意されています。例えば、MIPS アーキテクチャ [H93] では、ロードリンク命令とストア条件命令は、ロックや他の並行構造を構築するために並行して使用できます。これらの命令の C 擬似コードは図 28.5 のとおりです。Alpha、PowerPC、および ARM は同様の命令 [W09] を提供しています。

```

1 int LoadLinked(int *ptr) {
2     return *ptr;
3 }
4
5 int StoreConditional(int *ptr, int value) {
6     if (no one has updated *ptr since the LoadLinked to this address) {
7         *ptr = value;
8         return 1; // success!
9     } else {
10        return 0; // failed to update
11    }
12 }

```

Figure 28.5: Load-linked And Store-conditional

ロードリンクは、一般的なロード命令のように動作し、メモリから値を取り出してレジスタに配置します。キーの違いは store-conditional にあります。store-conditional は、アドレスへの介在するストアが発生していない場合にのみ成功します（ロードされたアドレスに格納された値を更新します）成功の場合、storeconditional は 1 を返し、ptr の値を value に更新します。失敗した場合、ptr の値は更新されず、0 が返されます。あなた自身の挑戦として、ロードリンクとストア条件付きを使用してロックを構築する方法を考えてみてください。次に、終了したら、以下のコードを見て、簡単な解決策を提供してください。解決策は図 28.6 のとおりです。

```

1 void lock(lock_t *lock) {
2     while (1) {
3         while (LoadLinked(&lock->flag) == 1)
4             ; // spin until it's zero
5         if (StoreConditional(&lock->flag, 1) == 1)
6             return; // if set-it-to-1 was a success: all done
7             // otherwise: try it all over again
8     }
9 }
10
11 void unlock(lock_t *lock) {
12     lock->flag = 0;
13 }

```

Figure 28.6: Using LL/SC To Build A Lock

lock() コードは唯一興味深い部分です。最初に、スレッドは、フラグが 0 に設定されるのを待って回転します（したがって、ロックが保持されていないことを示します）。一旦そのスレッドが store-conditional を介してロックを取得しようとすると、成功した場合、スレッドはフラグの値を原子的に 1 に変更し、クリティカルセクションに進むことができます。ストア条件の失敗がどのように発生する可能性があるかに注意してください。1 つのスレッドは lock() を呼び出し、load-linked を実行し、ロックが保持されていない場合は 0 を返します。store-conditional を試みる前に、それは中断され、別のスレッドがロックコードを入力し、ロードリンクされた命令を実行し、0 も取得し続けます。この時点で、2 つのスレッドがそれぞれロードリンクされて実行され、それぞれがストア条件を試行しようとしています。これらの命令の重要な特徴は、フラグを 1 に更新してロックを獲得するのに、これらのスレッドの 1 つだけが成功することです。store-conditional を試みる第 2 のスレッドは失敗し（他のスレッドが load-linked と storeconditional の間でフラグの値を更新したため）、再度ロックを取得しようとする必要があります。

TIP: LESS CODE IS BETTER CODE (LAUER'S LAW)

プログラマーは、何かをするために書いたコードの量を自慢する傾向があります。そうすることは根本的に壊れている。自慢するべきは、むしろ、与えられた仕事を達成するために書いたコード

の量です。簡潔で簡潔なコードが常に優先されます。理解しやすくなり、バグも少なくなります。Hugh Lauer 氏は、パイロットオペレーティングシステムの構築について議論するとき、「同じ人が 2 倍の時間があれば、コードの半分でシステムを生み出すことができます」[L81] この Lauer の法則を覚えておく価値があります。次回は、割り当てを完了するために書いたコードの量がどれほどであるか自慢しています。もう一度考え直してください。できるだけ明確かつ簡潔にコードを書き直してください。

数年前の授業では、学部生の David Capel が短絡ブール条件を楽しむあなたのため、上記のより簡潔な形式を提案しました。なぜそれが同等であるかを知ることができるかどうかを見てください。それは確かに短いです！

```
1 void lock(lock_t *lock) {
2     while (LoadLinked(&lock->flag) || !StoreConditional(&lock->flag, 1))
3         ; // spin
4 }
```

28.11 Fetch-And-Add

1 つの最終的なハードウェアプリミティブはフェッチアンドアド命令です。この命令は、アトミックに値をインクリメントし、特定のアドレスに古い値を戻します。fetch-and-add 命令の C 疑似コードは次のようになります。

```
1 int FetchAndAdd(int *ptr) {
2     int old = *ptr;
3     *ptr = old + 1;
4     return old;
5 }
```

この例では、Mellor-Crummey と Scott [MS91] が紹介したように、fetch-and-add を使用してより面白いチケットロックを構築します。ロックとアンロックのコードは図 28.7 のようになります。このソリューションでは、単一の値の代わりにチケットとターン変数を組み合わせて使用してロックを構築します。基本的な操作は非常に簡単です。スレッドがロックを取得したい場合は、最初にアトミックフェッチとアペンドをチケットの値で行います。その値はこのスレッドの「ターン」(myturn) とみなされます。次に、グローバルに共有されている lock->turn を使用して、そのスレッドの順番を判断します。特定のスレッドで (myturn == turn) のときは、そのスレッドがクリティカルセクションに入るのです。アンロックは、次の待機スレッド (存在する場合) がクリティカルセクションに入ることができるように、ターンをインクリメントするだけで達成されます。

```

1  typedef struct __lock_t {
2      int ticket;
3      int turn;
4  } lock_t;
5
6  void lock_init(lock_t *lock) {
7      lock->ticket = 0;
8      lock->turn = 0;
9  }
10
11 void lock(lock_t *lock) {
12     int myturn = FetchAndAdd(&lock->ticket);
13     while (lock->turn != myturn)
14         j // spin
15     }
16
17 void unlock(lock_t *lock) {
18     lock->turn = lock->turn + 1;
19 }

```

Figure 28.7: Ticket Locks

このソリューションとの重要な違いは、これまでの試みとは異なり、すべてのスレッドの進捗状況を保証します。スレッドにチケットの値が割り当てられると、そのスレッドの前にあるスレッドがクリティカルセクションを通過してロックを解除されると、将来のある時点でスケジュールされます。以前の試みでは、そのような保証は存在しませんでした。他のスレッドがロックを取得して解放しても、テストセットで回転するスレッドは永遠に回転する可能性があります。

28.12 Too Much Spinning: What Now?

私たちのシンプルなハードウェアベースのロックはシンプルで(数行のコードしかありません)、どんなシステムやコードの2つの優れた特性でも動作します(あなたが望むなら、いくつかのコードを書いても証明できます)。しかし、場合によっては、これらのソリューションは非常に非効率的である可能性があります。1つのプロセッサで2つのスレッドを実行しているとします。今度は、1つのスレッド(スレッド0)がクリティカルセクションにあり、したがってロックが保持され、残念なことに中断されることを想像してください。2番目のスレッド(スレッド1)はロックを取得しようとしましたが、保持されています。したがって、それは回転を開始します。スピンします。その後、もう少し回転します。最後に、タイマー割込みがオフになり、スレッド0が再び実行されてロックが解除され、最後に(スレッドが次に実行されるとき)、スレッド1はそれほど回転する必要はなく、ロックを取得できるようになります。このように、スレッドがこのような状況で回転したときはいつでも、変更を行わない値をチェックするだけで時間切れ全体を無駄にします。問題は、ロックを競合するN個のスレッドで悪化します。N-1タイムスライスは同様の方法で無駄になり、単なるスレッドが回転を止めてロックを解除するのを待っています。

THE CRUX: HOW TO AVOID SPINNING

CPUの時間を無駄にしないロックを開発するにはどうしたらいいですか?

ハードウェアのサポートだけでは問題を解決することはできません。私たちも OS のサポートが必要です！それがどのように機能するかを今考えてみましょう。

28.13 A Simple Approach: Just Yield, Baby

ハードウェアのサポートは私たちをかなり遠くにしてくれました。つまり、作業ロック、ロック取得の公平性(チケットロックの場合のように)が得られるようになりました。しかし、我々はまだ問題があります。クリティカルセクションでコンテキストスイッチが発生し、スレッドが無限に回転し始め、中断された(ロックを保持している)スレッドが再び実行されるのを待っていますか？

私たちの最初の試みは、シンプルでフレンドリーなアプローチです。スピルするときに、代わりに CPU を別のスレッドであきらめてください。あるいは、アル・デイヴィスが言うように、“just yield, baby!” [D91]。図 28.8 にそのアプローチを示します。

```

1 void init() {
2     flag = 0;
3 }
4
5 void lock() {
6     while (TestAndSet(&flag, 1) == 1)
7         yield(); // give up the CPU
8 }
9
10 void unlock() {
11     flag = 0;
12 }
```

Figure 28.8: Lock With Test-and-set And Yield

このアプローチでは、CPU をあきらめて別のスレッドを実行させたいときにスレッドが呼び出すことができるオペレーティングシステムのプリミティブ `yield()` を想定しています。スレッドは 3 つの状態(実行中、準備中、ブロック中)のいずれかになります。`yield` は、呼び出し元を実行状態から準備完了状態に移動させるシステムコールであり、したがって別のスレッドを実行するように促します。したがって、諦めたプロセスは本質的にそれ自体をスケジュールから外します。

1 つの CPU 上に 2 つのスレッドを持つ例を考えてみましょう。この場合、`yield base` アプローチは非常にうまく機能します。スレッドが `lock()` を呼び出して保持されているロックを見つける場合、それは単に CPU を諦めるため、もう一方のスレッドで実行してクリティカルセクションを終了します。この単純なケースでは、利回りアプローチがうまく機能します。

ここで、ロックを繰り返し競合するスレッド(たとえば 100)が多数ある場合を考えてみましょう。この場合、一方のスレッドがロックを取得して解放する前に先取りされた場合、他方の 99 スレッドはそれぞれ `lock()` を呼び出し、保持されているロックを見つけて CPU を `y` 諦めます。いくつかの種類のラウンドロビンスケジューラーを仮定すると、ロックを保持しているスレッドが再び実行される前に 99 のそれがこの実行と `yield` をします。回転方法(99 回のスライスが回転するのを無駄にする)よりは優れていますが、この方法はまだコストがかかります。コンテキストスイッチのコストが相当に高くなる可能性があり、そのためには無駄が多くなります。

さらに悪いことに、我々は飢餓問題に全く取り組んでいません。他のスレッドが繰り返しクリティカルセクションに入り、クリティカルセクションを終了する間、スレッドは無限の `yield` ループで捕まえられることが

あります。この問題に直接対処するアプローチが必要であることは明らかです。

28.14 Using Queues: Sleeping Instead Of Spinning

私たちの以前のアプローチの本当の問題は、あまりにも多くのチャンスを残すことです。スケジューラは、次に実行するスレッドを決定します。スケジューラが悪い選択をした場合、ロックを待つ(最初のアプローチ)か、すぐにCPUを降らなければならないスレッドが実行されます(2番目のアプローチ)。いずれにしても、無駄になる可能性があり、飢餓を防ぐことはできません。

したがって、現在の保有者がロックを解放した後にスレッドが次にロックを獲得するような制御を明示的に行う必要があります。これを行うには、もう少しOSサポートと、ロックを獲得するのを待っているスレッドを追跡するキューが必要です。

わかりやすくするために、私たちは、呼び出し元のスレッドをスリープ状態にするための `park()` と、`threadID` で指定された特定のスレッドを起動させるための `unpark(threadID)` の 2つの呼び出しに関して、Solarisによって提供されるサポートを使用します。これらの 2つのルーチンは、保持されたロックを獲得しようとすると呼び出し元をスリープ状態にし、ロックが解放されたときにスリープ状態に入るロックを構築するために並行して使用できます。このようなプリミティブの 1つの可能な使用を理解するために、図 28.9 のコードを見てみましょう。

この例では、いくつか面白いことをしています。まず、以前のテストと設定のアイデアをロックを待っているスレッドの明示的なキューと組み合わせて、より効率的なロックを作成します。次に、キューを使用して、次にロックを取得するユーザーを制御し、飢餓を回避します。

```

1  typedef struct __lock_t {
2      int flag;
3      int guard;
4      queue_t *q;
5  } lock_t;
6
7  void lock_init(lock_t *m) {
8      m->flag = 0;
9      m->guard = 0;
10     queue_init(m->q);
11 }
12
13 void lock(lock_t *m) {
14     while (TestAndSet(&m->guard, 1) == 1)
15         ; //acquire guard lock by spinning
16     if (m->flag == 0) {
17         m->flag = 1; // lock is acquired
18         m->guard = 0;
19     } else {
20         queue_add(m->q, gettid());
21         m->guard = 0;
22         park();
23     }
24 }
25
26 void unlock(lock_t *m) {
27     while (TestAndSet(&m->guard, 1) == 1)
28         ; //acquire guard lock by spinning
29     if (queue_empty(m->q))
30         m->flag = 0; // let go of lock; no one wants it
31     else
32         unpark(queue_remove(m->q)); // hold lock (for next thread!)
33     m->guard = 0;
34 }

```

Figure 28.9: Lock With Queues, Test-and-set, Yield, And Wakeup

基本的には、ロックが使用しているフラグやキューの操作を中心としたスピinnロックとして、ガードがどのように使用されているかを知ることができます(図 28.9)。したがって、このアプローチはスピinn待機を完全に回避しません。ロックを取得または解放している間にスレッドが中断され、他のスレッドがこのスレッドを再び実行するために回転待ちする原因となります。しかし、スピニングに費やされる時間は非常に限られており(ユーザー定義のクリティカルセクションではなく、ロックおよびアンロックコード内のほんの数個の命令)、このアプローチは妥当である可能性があります。

第 2 に、lock() では、スレッドがロックを取得できない(既に保持されている)ときに、自分自身をキューに追加することに注意して(gettid() 関数を呼び出して現在のスレッドのスレッド ID を取得します)、ガードを 0 に設定して CPU を諦めさせます。ここで読者のための質問です。ガードロックのリリースが park() の後に来た場合、どうなりますか? ヒントは「何か悪い」です。

別のスレッドが起動したときにフラグが 0 に戻らないという興味深い事実に気付くかもしれません。どうしてこうなるのでしょうか? さて、それはエラーではなく、むしろ必然です! スレッドが起動すると、あたかも park() から戻るかのようになります。ただし、コードのその時点でガードを保持していないため、flag を 1 に設定できません。したがって、ロックをスレッドから直接渡して、次のスレッドにロックを解放します。flag は 0 の間に設定されません。

最後に、park() の呼び出しの直前に、ソリューション内で認識される競合状態に気付くかもしれません。

タイミングが間違っていると、ロックは保持されなくなるまでスリープしなければならないとして、スレッドは park() しようとしています。その時点で他のスレッド(例えば、ロックを保持しているスレッド)に切り替えると、問題が発生する可能性があります。たとえば、そのスレッドがロックを解除した場合、最初のスレッドによる次の park() は、永遠に(潜在的に)スリープ状態になり、時にはウェイクアップ/待機レースと呼ばれる問題が発生します。

ASIDE: MORE REASON TO AVOID SPINNING: PRIORITY INVERSION

スピinnロックを避ける良い理由の1つはパフォーマンスです。メインテキストで説明したように、スレッドがロックを保持している間中断された場合、スピinnロックを使用する他のスレッドは、ロックが利用可能になるのを待つだけの大量のCPU時間を費やします。しかし、いくつかのシステムでスピinnロックを避ける別の興味深い理由があることが判明しました。気をつけなければならぬ問題は、優先度の逆転として知られています。残念なことに、地球 [M15] と火星 [R97] 上で発生する銀河系の間違いです！

あるシステムに2つのスレッドがあるとします。スレッド2(T2)のスケジューリング優先度は高く、スレッド1(T1)の優先度は低くなります。この例では、実際に両方が実行可能である場合、CPUスケジューラが常にT2を実行すると仮定します。T1は、T2がそうすることができないときにのみ実行されます(例えば、T2がI/Oでブロックされるとき)

ここで問題です。何らかの理由でT2がブロックされたとします。そのため、T1が走って、スピinnロックをつかんでクリティカルセクションに入ります。T2はブロックされなくなり(おそらくI/Oが完了したため)、CPUスケジューラは直ちにスケジューリングします(T1のスケジューリング)。T2は今ではロックを取得しようとします(T1がロックを保持できないため)、回転し続けます。ロックはスピinnロックであるため、T2は永遠に回転し、システムはハングアップします。スピinnロックの使用を避けるだけで、残念ながら、反転の問題を避けることはできません。3つのスレッドT1、T2、T3を考えてみましょう。T3は最高優先度、T1は最低です。今、T1がロックを取得したと想像してください。T3が起動し、T1よりも優先順位が高いため、すぐに実行されます(T1を先取りします)。T3はT1が保持しているロックを取得しようとしていますが、T1はまだそのロックを保持しているため、スタックで待機します。T2が実行を開始すると、T1より優先度が高くなり、T2が実行されます。T2より優先度の高いT3は、T1が待機中のところにスタックされています。強烈なT3は走れず、T2がCPUをコントロールしているのは悲しいことですか？高い優先順位を持つことは、これまでのようなものではありません。

優先順位の逆転の問題には、さまざまな方法で対応できます。スピinnロックによって問題が発生する特定のケースでは、スピinnロックの使用を避けることができます(詳細は後述)。より一般的には、優先度の低いスレッドを待っている優先度の高いスレッドは、優先度の高い継承として知られている逆転を実行して克服できるように、下位スレッドの優先度を一時的に引き上げることができます。最後の解決策は最も簡単です。すべてのスレッドが同じ優先順位を持っていることを確認してください。

Solarisでは、setpark()という3番目のシステムコールを追加することでこの問題を解決しています。このルーチンを呼び出すことによって、スレッドはパークしようとしていることを示すことができます。その後、パークが実際に呼び出される前に中断され、別のスレッドがパーク解除を呼び出すと、後続のパークはスリープする代わりにすぐに戻ります。lock()内のコード変更は非常に小さいです。

```

1 queue_add(m->q, gettid());
2 setpark(); // new code
3 m->guard = 0;

```

別の解決方法では、ガードをカーネルに渡すことができます。その場合、カーネルはロックをアトミックに解放し、実行中のスレッドをデキューするための予防措置を講じることができます。

28.15 Different OS, Different Support

これまで、スレッドライブラリにより効率的なロックを構築するために、OS が提供できるサポートの 1 つを見てきました。他の OS も同様のサポートを提供しています。詳細は異なります。

たとえば、Linux では、Solaris インタフェースに似ていますが、より多くのカーネル内の機能を提供するフューテックスが提供されています。具体的には、各フューテックスには、フューテックス内カーネルキューと同様に、特定の物理メモリロケーションが関連付けられています。発信者はフューテックスコール（後述）を使用して、必要に応じて sleep と wake を行うことができます。

具体的には、2 つの呼び出しが利用可能です。futex wait(address, expected) の呼び出しは、address の値が expected と等しいと仮定して、呼び出しスレッドをスリープ状態にします。同じでない場合、呼び出しあはただちに戻ります。ルーチン futex wake(アドレス) への呼び出しは、キューで待機しているスレッドを起動します。Linux mutex でのこれらの呼び出しの使用法を図 28.10 に示します。

```

1 void mutex_lock (int *mutex) {
2     int v;
3     /* Bit 31 was clear, we got the mutex (this is the fastpath) */
4     if (atomic_bit_test_set (mutex, 31) == 0)
5         return;
6     atomic_increment (mutex);
7     while (1) {
8         if (atomic_bit_test_set (mutex, 31) == 0) {
9             atomic_decrement (mutex);
10            return;
11        }
12        /* We have to wait now. First make sure the futex value
13           we are monitoring is truly negative (i.e. locked). */
14        v = *mutex;
15        if (v >= 0)
16            continue;
17        futex_wait (mutex, v);
18    }
19 }
20
21 void mutex_unlock (int *mutex) {
22     /* Adding 0x80000000 to the counter results in 0 if and only if
23        there are not other interested threads */
24     if (atomic_add_zero (mutex, 0x80000000))
25         return;
26
27     /* There are other threads waiting for this mutex,
28        wake one of them up. */
29     futex_wake (mutex);
30 }
```

Figure 28.10: Linux-based Futex Locks

nptl ライブラリ (gnu libc ライブラリの一部)[L09] の lowlevellock.h にあるこのコードスニペットは、いくつかの理由で興味深いものです。まず、ロックが保持されているかどうか（整数の上位ビット）とロックのウェイタ数（その他のすべてのビット）を追跡するために、単一の整数を使用します。したがって、ロックが負の場合、ロックは保持されています（上位ビットが設定され、そのビットが整数の符号を決定するため）。

次に、コードスニペットは、一般的なケース、特にロックの競合がない場合に最適化する方法を示しています。ロックを取得して解放するスレッドが1つしかないと、ほとんど作業が行われません（アトミックビットのテストとロックを設定し、ロックを解除するためのアトミックな追加を行う）

この“実世界”ロックの残りの部分をパズルを当てはめて、その動作を理解できるかどうかを確認してください。それをやって、少なくとも誰かが何かこの本について聞いてきたときに答えるようなLinuxのlockマスターになります。

28.16 Two-Phase Locks

最終的な注記：Linuxのアプローチは、古いアプローチを好んで使っており、少なくとも数年にわたってオン・オフされ、1960年代初めにダム・ロックズ（Dahm Locks）[M82]までさかのぼる古いアプローチである2フェーズロックをもっています。2フェーズロックは、特にロックが解放されようとしている場合に、回転が便利であることを認識しています。したがって、最初のフェーズでは、ロックはロックを獲得できることを期待してしばらくの間、回転します。

ただし、最初のスピinnフェーズでロックが取得されない場合は、呼び出し元がスリープ状態になる2番目のフェーズに入り、後でロックが解放されたときにのみ起動します。上のLinuxロックはそのようなロックの一種ですが、一回だけ回転します。futexサポートを使用してスリープする前に、これを一般化すると固定された時間ループ内で回転する可能性があります。

2フェーズロックは、2つの良いアイデアを組み合わせることで、より良いアイデアを生むハイブリッドアプローチのもう一つの例です。もちろん、それはハードウェア環境、スレッド数、その他の仕事量の詳細など、多くのことに強く依存します。いつものように、すべての可能なユースケースに適した単一の汎用ロックを作成することは非常に困難です。

28.17 Summary

上記のアプローチは、最近のロックの構築方法を示しています。いくつかのハードウェアサポート（より強力な命令の形式）とオペレーティングシステムのサポート（例えば、Solarisのpark()やunpark()プリミティブ、futex Linuxの場合）もちろん、詳細は異なります。そのようなロックを実行する正確なコードは、通常、高度に調整されています。詳細を表示したい場合は、SolarisまたはLinuxのコードベースを調べてください。これらは魅力的な読書です[L09、S09]。現代のマルチプロセッサ[D+13]のロック戦略の比較のためのDavidらの優れた作業も参照してください。

参考文献

[D91] “Just Win, Baby: Al Davis and His Raiders”

Glenn Dickey, Harcourt 1991

There even exists a book about Al Davis and his famous “just win” quote. Or, we suppose, the book is more about Al Davis and the Raiders, and maybe not just the quote. Read the book to find out? But just to be clear: we are not recommending this book, we just needed a citation for the quote.

[D+13] “Everything You Always Wanted to Know about Synchronization but Were Afraid to Ask”

Tudor David, Rachid Guerraoui, Vasileios Trigonakis

SOSP ’13, Nemacolin Woodlands Resort, Pennsylvania, November 2013 An excellent recent paper comparing many different ways to build locks using hardware primitives. A great read to see how many ideas over the years work on modern hardware.

- [D68] “Cooperating sequential processes”
 Edsger W. Dijkstra, 1968
 Available: <http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF>
 One of the early seminal papers in the area. Discusses how Dijkstra posed the original concurrency problem, and Dekker’s solution.
- [H93] “MIPS R4000 Microprocessor User’s Manual”
 Joe Heinrich, Prentice-Hall, June 1993
 Available: http://cag.csail.mit.edu/raw/documents/R4400_Uman_book_Ed2.pdf
- [H91] “Wait-free Synchronization”
 Maurice Herlihy
 ACM Transactions on Programming Languages and Systems (TOPLAS)
 Volume 13, Issue 1, January 1991
 A landmark paper introducing a different approach to building concurrent data structures. However, because of the complexity involved, many of these ideas have been slow to gain acceptance in deployed systems.
- [L81] “Observations on the Development of an Operating System”
 Hugh Lauer
 SOSP ’81, Pacific Grove, California, December 1981
 A must-read retrospective about the development of the Pilot OS, an early PC operating system. Fun and full of insights.
- [L09] “glibc 2.9 (include Linux pthreads implementation)”
 Available: <http://ftp.gnu.org/gnu/glibc/>
 In particular, take a look at the nptl subdirectory where you will find most of the pthread support in Linux today.
- [M82] “The Architecture of the Burroughs B5000
 20 Years Later and Still Ahead of the Times?”
 Alastair J.W. Mayer, 1982
www.ajwm.net/amayer/papers/B5000.html
 From the paper: “One particularly useful instruction is the RDLK (read-lock). It is an indivisible operation which reads from and writes into a memory location.” RDLK is thus an early test-and-set primitive, if not the earliest. Some credit here goes to an engineer named Dave Dahm, who apparently invented a number of these things for the Burroughs systems, including a form of spin locks (called “Buzz Locks”) as well as a two-phase lock eponymously called “Dahm Locks.”
- [M15] “OSSpinLock Is Unsafe”
 John McCall
 Available: mjtsai.com/blog/2015/12/16/osspinlock-is-unsafe
 A short post about why calling OSSpinLock on a Mac is unsafe when using threads of different priorities – you might end up spinning forever! So be careful, Mac fanatics, even your mighty system sometimes is less than perfect... .
- [MS91] “Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors”
 John M. Mellor-Crummey and M. L. Scott
 ACM TOCS, Volume 9, Issue 1, February 1991
 An excellent and thorough survey on different locking algorithms. However, no operating systems support is used, just fancy hardware instructions.

[P81] “Myths About the Mutual Exclusion Problem”

G.L. Peterson

Information Processing Letters, 12(3), pages 115–116, 1981

Peterson’s algorithm introduced here.

[R97] “What Really Happened on Mars?”

Glenn E. Reeves

Available: research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Authoritative_Account.html

A detailed description of priority inversion on the Mars Pathfinder robot. This low-level concurrent code matters a lot, especially in space!

[S05] “Guide to porting from Solaris to Linux on x86”

Ajay Sood, April 29, 2005

Available: <http://www.ibm.com/developerworks/linux/library/l-solar/>

[S09] “OpenSolaris Thread Library”

Available: <http://src.opensolaris.org/source/xref/onnv/onnv-gate/>

`usr/src/lib/libc/port/threads/synch.c`

This is also pretty interesting to look at, though who knows what will happen to it now that Oracle owns Sun. Thanks to Mike Swift for the pointer to the code.

[W09] “Load-Link, Store-Conditional”

Wikipedia entry on said topic, as of October 22, 2009

<http://en.wikipedia.org/wiki/Load-Link/Store-Conditional>

Can you believe we referenced wikipedia? Pretty lazy, no? But, we found the information there first, and it felt wrong not to cite it. Further, they even listed the instructions for the different architectures: `ldl l/stl c` and `ldq l/stq c` (Alpha), `lwarx/stwcx` (PowerPC), `ll/sc` (MIPS), and `ldrex/strex` (ARM version 6 and above). Actually wikipedia is pretty amazing, so don’t be so harsh, OK?

[WG00] “The SPARC Architecture Manual: Version 9”

David L. Weaver and Tom Germond, September 2000

SPARC International, San Jose, California

Available: <http://www.sparc.org/standards/SPARCV9.pdf>

Also see: http://developers.sun.com/solaris/articles/atomic_sparc/ for some more details on Sparc atomic operations.

29 Lock-based Concurrent Data Structures

ロックの説明を終わる前に、いくつかの一般的なデータ構造でロックを使用する方法をまず説明します。データ構造体にロックを追加してスレッドで使用できるようにすると、構造体のスレッドが安全になります。もちろん、そのようなロックがどのように追加されるかは、データ構造の正確さとパフォーマンスの両方を決定します。

CRUX: HOW TO ADD LOCKS TO DATA STRUCTURES

特定のデータ構造が与えられたら、それを正しく動作させるために、どのようにロックを追加するべきですか？さらに、データ構造が高性能をもたらすようにロックを追加すると、多くのスレッドが同時に構造にアクセスできるようになります。

もちろん、すべてのデータ構造や並行処理を追加するためのすべてのメソッドをカバーすることは難しいでしょう。何年もの間研究されてきたトピックであり、数千もの研究論文が出版されています。したがって、私たちは、必要とされる思考のタイプを十分に紹介し、あなた自身のさらなる調査のために、いくつかの良い資料源を参照することを望みます。私たちは Moir と Shavit の調査が大きな情報源であることを発見しました [MS04]。

29.1 Concurrent Counters

最も単純なデータ構造の1つはカウンタです。これは一般的に使用され、簡単なインターフェースを持つ構造です。図 29.1 に単純な非同期カウンタを定義します。

```

1  typedef struct __counter_t {
2      int value;
3  } counter_t;
4
5  void init(counter_t *c) {
6      c->value = 0;
7  }
8
9  void increment(counter_t *c) {
10     c->value++;
11 }
12
13 void decrement(counter_t *c) {
14     c->value--;
15 }
16
17 int get(counter_t *c) {
18     return c->value;
19 }
```

Figure 29.1: A Counter Without Locks

Simple But Not Scalable

ご覧のように、非同期カウンタは簡単なデータ構造であり、実装するには少量のコードしか必要としません。私たちは次の課題を抱えています。このコードを安全にするにはどうしたらいいですか？図 29.2 に、その方法を示します。

```

1  typedef struct __counter_t {
2      int             value;
3      pthread_mutex_t lock;
4  } counter_t;
5
6  void init(counter_t *c) {
7      c->value = 0;
8      Pthread_mutex_init(&c->lock, NULL);
9  }
10
11 void increment(counter_t *c) {
12     Pthread_mutex_lock(&c->lock);
13     c->value++;
14     Pthread_mutex_unlock(&c->lock);
15 }
16
17 void decrement(counter_t *c) {
18     Pthread_mutex_lock(&c->lock);
19     c->value--;
20     Pthread_mutex_unlock(&c->lock);
21 }
22
23 int get(counter_t *c) {
24     Pthread_mutex_lock(&c->lock);
25     int rc = c->value;
26     Pthread_mutex_unlock(&c->lock);
27     return rc;
28 }
```

Figure 29.2: A Counter With Locks

この並行カウンタは単純で、正しく動作します。実際、単純で最も基本的な並行データ構造に共通する設計パターンに従います。データ構造を操作するルーチンを呼び出すときに取得され、呼び出しから戻ったときに解放される单一のロックを単純に追加します。この方法では、モニターを使用して構築されたデータ構造 [BH73] に似ています。ここでは、ロックを取得し、オブジェクトメソッドから呼び出して戻すときに自動的に解放されます。

この時点での問題はパフォーマンスです。データ構造が遅すぎる場合は、単なるロックを追加する以上が必要です。そのような最適化は、必要に応じて、この章の残りの部分のトピックです。データ構造が遅すぎない場合は、それ以上何もしないでよいことに注意してください。シンプルなものがうまくいけば何か面白くする必要はありません。

単純なアプローチのパフォーマンスコストを理解するために、各スレッドが単一の共有カウンタを一定回数更新するベンチマークを実行します。スレッドの数を変更します。図 29.3 は、1~4 つのスレッドをアクティブにしたときの合計時間を示しています。各スレッドはカウンタを 100 万回更新します。この実験は、Intel 2.7GHz i5 CPU 4 台を搭載した iMac 上で実行されました。より多くの CPU がアクティブになると、単位時

間あたりの総作業量が増えることを期待します。

図の一番上の行(正確に表示されている)から、同期カウンタのパフォーマンスが不十分であることがわかります。1回のスレッドではわずかな時間(約0.03秒)で100万回のカウンタ更新を完了できますが、2つのスレッドでそれぞれ100万回更新すると5秒以上かかることになります。スレッド数が増えれば悪化します。

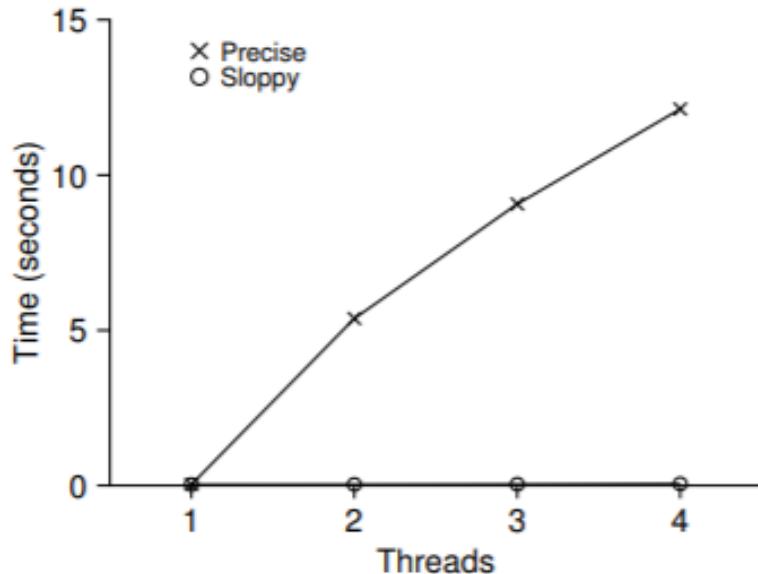


Figure 29.3: Performance of Traditional vs. Sloppy Counters

理想的には、複数のプロセッサで一つのスレッドを一つのプロセッサで完了するのが理想的です。この目的を達成することを完璧なスケーリングといいます。より多くの作業が行われても、それは並行して行われるため、タスクを完了するのにかかる時間は増加しません。

Scalable Counting

驚いたことに、研究者は、よりスケーラブルなカウンターを何年も構築する方法を研究しました[MS04]。オペレーティングシステムパフォーマンス分析の最近の作業では[B+10]が示されているため、スケーラブルカウンタが重要であるという事実はさらに驚くべきことです。スケーラブルなカウントをせずに、Linux上で実行される仕事量の中には、マルチコアマシンでのスケーラビリティの問題が深刻です。この問題を攻撃するために多くのテクニックが開発されていますが、ここでは特定のアプローチを説明します。最近の研究[B+10]で紹介されたこのアイデアは、お粗末なカウンターとして知られています。

粗雑なカウンタは、多数のローカル物理カウンタ(CPUコアごとに1つ)と単一のグローバルカウンタを一つの論理カウンタとして動作します。具体的には、4つのCPUを持つマシンでは、4つのローカルカウンタと1つのグローバルカウンタがあります。これらのカウンタに加えて、それぞれにロックがあります。(ローカルカウンタごとに1つ、グローバルカウンタに1つ)

粗末なカウントの基本的な考え方は次のとおりです。特定のコアで実行されているスレッドがカウンタをインクリメントしたい場合は、ローカルカウンタをインクリメントします。このローカルカウンタへのアクセスは、対応するローカルロックを介して同期されます。各CPUには独自のローカルカウンタがあるため、CPU間のスレッドは競合することなくローカルカウンタを更新できるため、カウンタの更新はスケーラブルです。

しかし、グローバルカウンタを最新の状態に保つために(スレッドがその値を読みたい場合)、ローカル値はグローバルロックを取得し、ローカルカウンタの値でインクリメントすることによって、グローバルカウンタ

に定期的に転送されます。ローカルカウンタはゼロにリセットされます。

どのくらいの頻度でこのローカルからグローバルへの転送が発生するかは、閾値（ここでは S と呼ぶ）によって決まります。 S が小さいほど、カウンタは上記のスケーラブルではないカウンタのように動作します。 S が大きければ大きいほど、カウンターのスケーラビリティは向上しますが、実際のカウントからグローバル値は遠く離れたものになる可能性があります。正確な値を得るためにすべてのローカル・ロックとグローバル・ロックを（デッドロックを回避するために指定された順序で）取得できますが、それはスケーラブルではありません。

これを明確にするために、例を見てみましょう（図 29.4）。この例では、しきい値 S は 5 に設定され、ローカルカウンタ $L_1 \sim L_4$ を更新する 4 つの CPU のそれぞれにスレッドがあります。トレースにはグローバルカウンタ値 (G) も表示され、時間が下がります。各時間ステップで、ローカルカウンタをインクリメントすることができます。ローカル値が閾値 S に達すると、ローカル値がグローバルカウンタに転送され、ローカルカウンタがリセットされます。

Time	L_1	L_2	L_3	L_4	G
0	0	0	0	0	0
1	0	0	1	1	0
2	1	0	2	1	0
3	2	0	3	1	0
4	3	0	3	2	0
5	4	1	3	3	0
6	$5 \rightarrow 0$	1	3	4	5 (from L_1)
7	0	2	4	$5 \rightarrow 0$	10 (from L_4)

Figure 29.4: Tracing the Sloppy Counters

図 29.3 の下段には、閾値 S が 1024 の粗末なカウンタのパフォーマンスが示されています。4 つのプロセッサで 400 万回のカウンタを更新するのにかかる時間は、1 つのプロセッサで 100 万回更新するのにかかる時間よりもほとんど変わりません。

図 29.6 に、閾値 S の重要性を示します。4 つのスレッドがそれぞれ 4 つの CPU で 100 万回カウンタをインクリメントします。 S が低い場合、パフォーマンスは低下します（ただし、グローバルカウントは常に正確です）。 S が高い場合、パフォーマンスは優れていますが、グローバルカウントが遅れます（CPU の数に S を掛けたものが最大でも）。この精度/性能のトレードオフは、厄介なカウンタが可能にするものです。

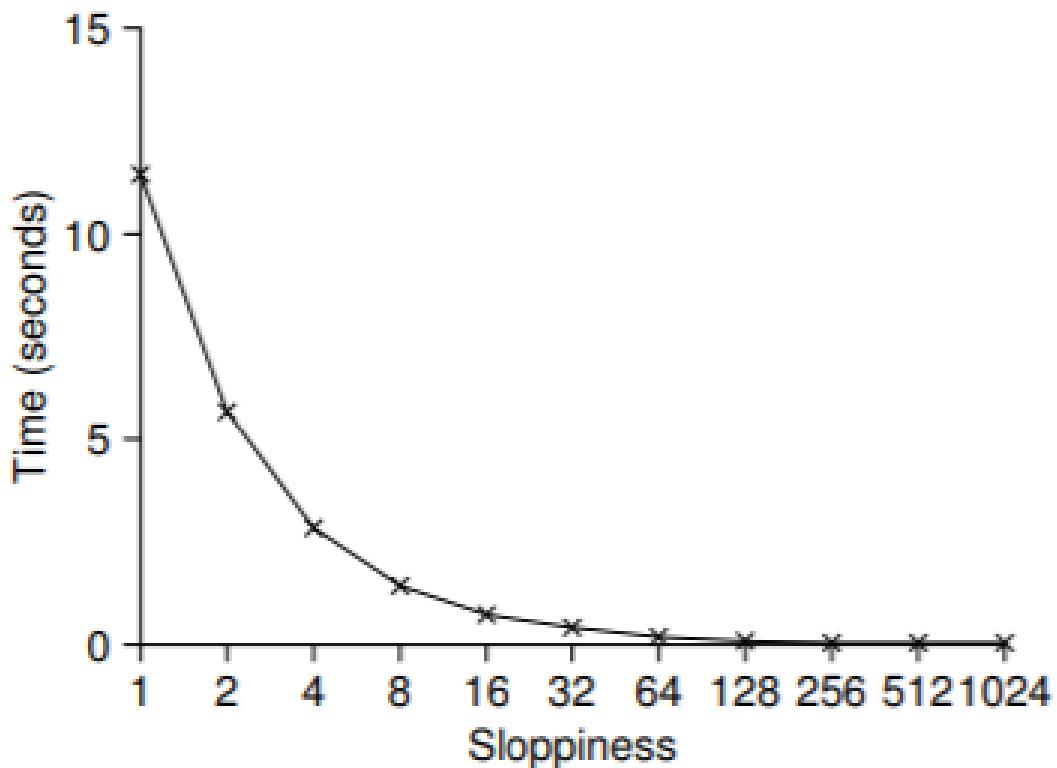


Figure 29.6: Scaling Sloppy Counters

このようなちょっとしたカウンターの大まかなバージョンが図 29.5 にあります。それを読んで、あるいはより良い方法で、それがどのように機能するかをよりよく理解するために、いくつかの実験をしてみてください。

```

1  typedef struct __counter_t {
2      int           global;          // global count
3      pthread_mutex_t glock;        // global lock
4      int           local[NUMCPUS]; // local count (per cpu)
5      pthread_mutex_t llock[NUMCPUS]; // ... and locks
6      int           threshold;     // update frequency
7 } counter_t;
8
9 // init: record threshold, init locks, init values
10 //       of all local counts and global count
11 void init(counter_t *c, int threshold) {
12     c->threshold = threshold;
13     c->global = 0;
14     pthread_mutex_init(&c->glock, NULL);
15     int i;
16     for (i = 0; i < NUMCPUS; i++) {
17         c->local[i] = 0;
18         pthread_mutex_init(&c->llock[i], NULL);
19     }
20 }
21
22 // update: usually, just grab local lock and update local amount
23 //       once local count has risen by 'threshold', grab global
24 //       lock and transfer local values to it
25 void update(counter_t *c, int threadID, int amt) {
26     int cpu = threadID % NUMCPUS;
27     pthread_mutex_lock(&c->llock[cpu]);
28     c->local[cpu] += amt;           // assumes amt > 0
29     if (c->local[cpu] >= c->threshold) { // transfer to global
30         pthread_mutex_lock(&c->glock);
31         c->global += c->local[cpu];
32         pthread_mutex_unlock(&c->glock);
33         c->local[cpu] = 0;
34     }
35     pthread_mutex_unlock(&c->llock[cpu]);
36 }
37
38 // get: just return global amount (which may not be perfect)
39 int get(counter_t *c) {
40     pthread_mutex_lock(&c->glock);
41     int val = c->global;
42     pthread_mutex_unlock(&c->glock);
43     return val; // only approximate!
44 }

```

Figure 29.5: Sloppy Counter Implementation

29.2 Concurrent Linked Lists

次に、より複雑な構造、linked list を調べます。基本的なアプローチからもう一度始めましょう。簡単にするために、このようなりストには明白なルーチンをいくつか省略し、同時に挿入することに重点を置いていきます。私たちは読者にロックアップや削除などについて考えるようになります。図 29.7 に、この初步的なデータ構造のコードを示します。

```

1 // basic node structure
2 typedef struct __node_t {
3     int                         key;
4     struct __node_t            *next;
5 } node_t;
6
7 // basic list structure (one used per list)
8 typedef struct __list_t {
9     node_t                      *head;
10    pthread_mutex_t             lock;
11 } list_t;
12
13 void List_Init(list_t *L) {
14     L->head = NULL;
15     pthread_mutex_init(&L->lock, NULL);
16 }
17
18 int List_Insert(list_t *L, int key) {
19     pthread_mutex_lock(&L->lock);
20     node_t *new = malloc(sizeof(node_t));
21     if (new == NULL) {
22         perror("malloc");
23         pthread_mutex_unlock(&L->lock);
24         return -1; // fail
25     }
26     new->key = key;
27     new->next = L->head;
28     L->head = new;
29     pthread_mutex_unlock(&L->lock);
30     return 0; // success
31 }
32
33 int List_Lookup(list_t *L, int key) {
34     pthread_mutex_lock(&L->lock);
35     node_t *curr = L->head;
36     while (curr) {
37         if (curr->key == key) {
38             pthread_mutex_unlock(&L->lock);
39             return 0; // success
40         }
41         curr = curr->next;
42     }
43     pthread_mutex_unlock(&L->lock);
44     return -1; // failure
45 }

```

Figure 29.7: Concurrent Linked List

コードでわかるように、コードは入力時に挿入ルーチン内のロックを取得し、終了時に解放します。

`malloc()` が失敗する（まれなケース）場合、小さなトリッキーな問題が発生します。この場合、コードは挿入を失敗する前にロックを解除する必要があります。

この種の例外的な制御フローは、かなりエラーを起こしやすいことが示されています。最近の Linux カーネルパッチの調査では、稀にしか使われていないコードパスでは、バグの巨大な部分（ほぼ 40 %）が検出されていることが分かりました（実際に、この観察は私たち自身の研究の一部を喚起しました。Linux ファイルシステムであり、より堅牢なシステムとなる [S+11]）

私たちは挿入とロックアップルーチンが同時挿入のもとで正しいままであるように書き直すことはできますが、失敗パスでもロックを解除する呼び出しを追加する必要がある場合は避ける必要があるのでしょうか？

この場合の答えは「はい」です。具体的には、ロックと解放が挿入コード内の実際のクリティカルセクションのみを囲み、参照コードで共通の終了パスが使用されるように、コードを少し並べ替えることができます。

前者は実際にはロックアップの一部がロックされる必要がないため動作します。`malloc()` 自体がスレッドセーフであると仮定すると、各スレッドは競合状態や他の並行性のバグを心配することなくスレッドに呼び出すことができます。共有リストを更新するときだけ、ロックを保持する必要があります。これらの変更の詳細については、図 29.8 を参照してください。

```

1 void List_Init(list_t *L) {
2     L->head = NULL;
3     pthread_mutex_init(&L->lock, NULL);
4 }
5
6 void List_Insert(list_t *L, int key) {
7     // synchronization not needed
8     node_t *new = malloc(sizeof(node_t));
9     if (new == NULL) {
10         perror("malloc");
11         return;
12     }
13     new->key = key;
14
15     // just lock critical section
16     pthread_mutex_lock(&L->lock);
17     new->next = L->head;
18     L->head = new;
19     pthread_mutex_unlock(&L->lock);
20 }
21
22 int List_Lookup(list_t *L, int key) {
23     int rv = -1;
24     pthread_mutex_lock(&L->lock);
25     node_t *curr = L->head;
26     while (curr) {
27         if (curr->key == key) {
28             rv = 0;
29             break;
30         }
31         curr = curr->next;
32     }
33     pthread_mutex_unlock(&L->lock);
34     return rv; // now both success and failure
35 }

```

Figure 29.8: Concurrent Linked List: Rewritten

ロックアップルーチンは、メインの検索ループから单一のリターンパスにジャンプする単純なコード変換です。再度コードをロックすると、コード内のロック取得/解放ポイントの数が減り、誤ってバグ(コードを返す前にロックを解除することを忘れるなど)がコードに挿入される可能性が低くなります。

Scaling Linked Lists

私たちは再び基本的な並行リンクリストを持っていますが、もう一度、それは特にうまく調整できない状況にあります。リスト内でより多くの並行処理を可能にするために研究者が検討した手法の1つは、ハンドオーバーハンドロック(a.k.a. lock coupling)[MS04]です。

アイデアはかなり簡単です。リスト全体を单一のロックにする代わりに、リストのノードごとにロックを追加します。リストをトラバースすると、コードは最初に次のノードのロックを取得し、現在のノードのロックを解除します(これが hand-over-hand の名前を意味します)

概念的には、ハンドオーバーハンドリンクリストは素晴らしいです。これはリスト操作で高度な並行性を実

現します。しかし、実際には、リストトラバーサルの各ノードのロックを取得して解放するオーバーヘッドが法外なので、単純なシングルロックアプローチよりも速くそのような構造を作るのは難しいです。

非常に大きなリストと多数のスレッドがあっても、複数の進行中のトラバーサルを許可することによって可能になる同時実行性は、単純に单一のロックを取得し、操作を実行し、解放するよりも速くなる可能性は低いです。多分、ある種のハイブリッド（あなたが非常に多くのノードごとの新しいロックを取得する場所）は調査する価値があります。

TIP: MORE CONCURRENCY ISN'T NECESSARILY FASTER

設計するスキームがオーバーヘッドを増やす場合（例えば、ロックを一度ではなく頻繁に取得して解放するなど）、それがより同時であるという事実は重要ではないかもしれません。シンプルなスキームは、特にコストのかかるルーチンをめったに使用しないとうまくいく傾向があります。より多くのロックと複雑さを追加することはあなたの没落につながります。すべてのことには、両方の選択肢（単純だが同時性は低く、複雑性は高いがそれ以上のもの）を構築し、それがどのように行われるかを実際に比べる1つの方法があります。結局のところ、あなたはパフォーマンスを欺くことはできません。あなたの考えはより速いか、そうではないかです。

TIP: BE WARY OF LOCKS AND CONTROL FLOW

並行コードおよび他の場所で有用な一般的な設計のヒントは、関数の戻り、終了、または関数の実行を停止させる他の類似のエラー条件につながる制御フローの変化に注意することです。エラーが発生したときにロックを獲得したり、メモリを割り当てたり、他の同様のステートフルな操作を行うことで多くの関数が始まるので、コードはエラーの起こりやすい状態に戻る前にすべての状態を元に戻す必要があります。したがって、このパターンを最小限にするためにコードを構造化することが最善です。

29.3 Concurrent Queues

今のところ分かっているように、同時ロック・データ構造を作る標準的な方法があります。大きなロックを追加することです。キューの場合は、それを把握できると仮定して、そのアプローチをスキップします。

代わりに、Michael と Scott [MS98] によって設計されたわずかに並行したキューを見てみましょう。このキューに使用されるデータ構造とコードは、次のページの図 29.9 にあります。

```

1  typedef struct __node_t {
2      int                  value;
3      struct __node_t     *next;
4  } node_t;
5
6  typedef struct __queue_t {
7      node_t              *head;
8      node_t              *tail;
9      pthread_mutex_t     headLock;
10     pthread_mutex_t    tailLock;
11 } queue_t;
12
13 void Queue_Init(queue_t *q) {
14     node_t *tmp = malloc(sizeof(node_t));
15     tmp->next = NULL;
16     q->head = q->tail = tmp;
17     pthread_mutex_init(&q->headLock, NULL);
18     pthread_mutex_init(&q->tailLock, NULL);
19 }
20
21 void Queue_Enqueue(queue_t *q, int value) {
22     node_t *tmp = malloc(sizeof(node_t));
23     assert(tmp != NULL);
24     tmp->value = value;
25     tmp->next = NULL;
26
27     pthread_mutex_lock(&q->tailLock);
28     q->tail->next = tmp;
29     q->tail = tmp;
30     pthread_mutex_unlock(&q->tailLock);
31 }
32
33 int Queue_Dequeue(queue_t *q, int *value) {
34     pthread_mutex_lock(&q->headLock);
35     node_t *tmp = q->head;
36     node_t *newHead = tmp->next;
37     if (newHead == NULL) {
38         pthread_mutex_unlock(&q->headLock);
39         return -1; // queue was empty
40     }
41     *value = newHead->value;
42     q->head = newHead;
43     pthread_mutex_unlock(&q->headLock);
44     free(tmp);
45     return 0;
46 }

```

Figure 29.9: Michael and Scott Concurrent Queue

このコードを注意深く調べると、キューの先頭と末尾の 2 つのロックがあることに気付くでしょう。これらの 2 つのロックの目的は、エンキュー操作とデキュー操作の同時実行を可能にすることです。一般的なケースでは、エンキュー＆チーンはテールロックにのみアクセスし、ヘッドロックのみをデキューします。

Michael と Scott が使うトリックの 1 つは、(キューの初期化コードで割り当てられた) ダミーノードを追加することです。このダミーは、頭と尾の動作の分離を可能にします。コードを研究するか、それを入力して実行し、それを測定し、それがどのように深く働くかを理解してください。

キューはマルチスレッドアプリケーションで一般的に使用されます。しかし、ここで(ロックだけで) 使用されるキューのタイプは、しばしばそのようなプログラムのニーズを完全に満たすものではありません。キューが空であるか過度に満たされている場合にスレッドが待機できるように、より完全に開発されたバウンド・キューは、条件変数に関する次の章で集中的に学んでいきます。それを見てください。

29.4 Concurrent Hash Table

シンプルで広く適用可能な並行データ構造であるハッシュテーブルを使用して、議論を終了します。サイズを変更しない単純なハッシュテーブルに焦点を当てます。私たちが読者のための練習として残すリサイズを扱うためにはもう少し作業が必要です(申し訳ありません)。

```

1  #define BUCKETS (101)
2
3  typedef struct __hash_t {
4      list_t lists[BUCKETS];
5  } hash_t;
6
7  void Hash_Init(hash_t *H) {
8      int i;
9      for (i = 0; i < BUCKETS; i++) {
10          List_Init(&H->lists[i]);
11      }
12  }
13
14  int Hash_Insert(hash_t *H, int key) {
15      int bucket = key % BUCKETS;
16      return List_Insert(&H->lists[bucket], key);
17  }
18
19  int Hash_Lookup(hash_t *H, int key) {
20      int bucket = key % BUCKETS;
21      return List_Lookup(&H->lists[bucket], key);
22  }

```

Figure 29.10: A Concurrent Hash Table

この並行ハッシュテーブルは簡単で、先に開発した並行リストを使用して作成され、非常にうまく機能します。その優れたパフォーマンスの理由は、構造全体に対して单一のロックを持つ代わりに、ハッシュ・バケットごとのロックを使用するためです(それぞれがリストで表されます)。これにより、多くの同時操作が可能になります。

図 29.11 には、同時更新中のハッシュテーブルのパフォーマンス(同一の iMac で 4 つの CPU を使用して、4 つのスレッドそれぞれから 10,000~50,000 の同時更新が行われています)を示しています。また、比較のために、linked list(单一のロックを持つ)のパフォーマンスを示します。

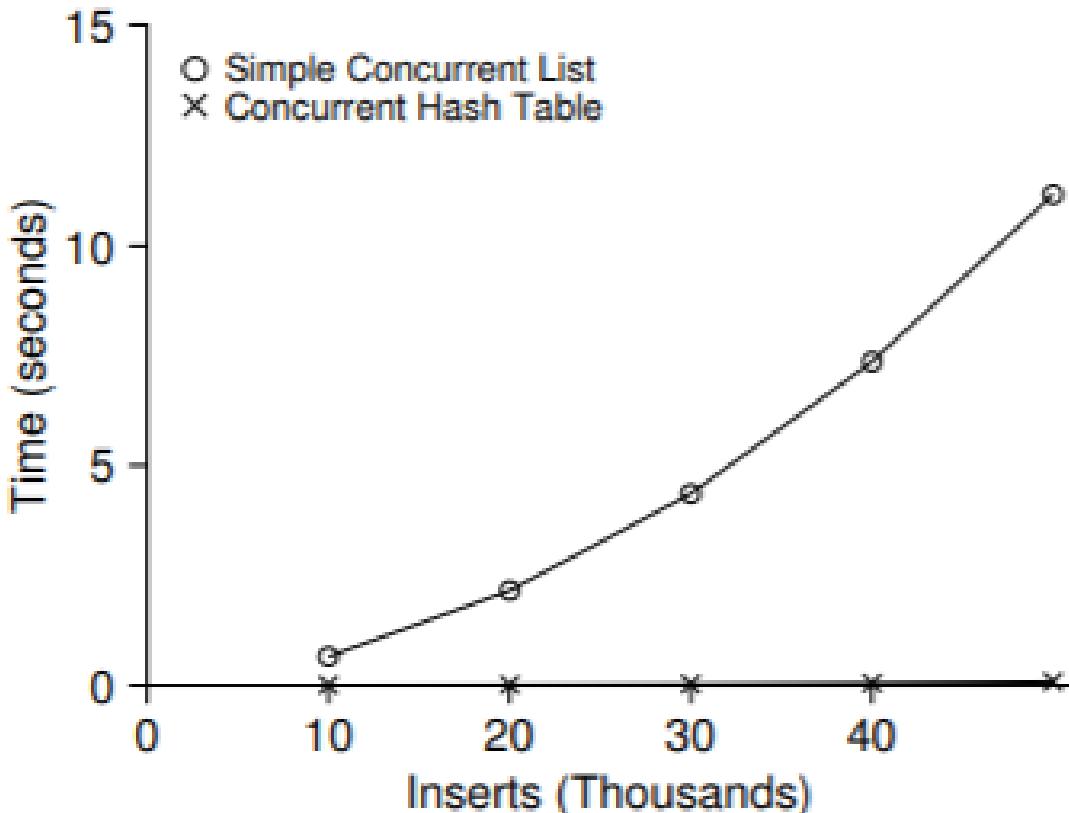


Figure 29.11: Scaling Hash Tables

グラフからわかるように、このシンプルな同時ハッシュテーブルは非常にスケーラブルです。対照的に、linked list はそうではありません。

TIP: AVOID PREMATURE OPTIMIZATION (KNUTH'S LAW)

並行データ構造を構築する場合は、最も基本的なアプローチから始めます。これは、同期アクセスを提供するために单一の大きなロックを追加することです。そうすることで、適切なロックを構築することができます。パフォーマンス上の問題を抱えていることが判明した場合は、それを改良して、必要に応じて高速化するだけです。Knuth が有名に述べたように、「時期尚早最適化はすべての悪の根源です。

多くのオペレーティングシステムでは、最初に Sun OS や Linux などのマルチプロセッサに移行するときに、单一のロックを利用していました。後者の場合、このロックには大きなカーネルロック (BKL) という名前があります。長年にわたり、この単純なアプローチは良いものでしたが、マルチ CPU システムが標準となったときに、カーネル内の单一のアクティブなスレッドを一度に許可するだけでは、パフォーマンスのボトルネックになりました。したがって、最終的には、改善された並行性の最適化をこれらのシステムに追加することになりました。Linux では、より単純なアプローチが採用されました。一つのロックを多くのものに置き換えます。

Sun の中ではより根本的な決断が下されました。Solaris と呼ばれるまったく新しいオペレーティングシステムを構築します。このオペレーティングシステムは、より基本的に同時実行性を組み込んでいます。これらの魅力的なシステムの詳細については、Linux および Solaris カーネルの本を読んでください。[BC05, MM00]

29.5 Summary

カウンターからリストとキューへの並行データ構造のサンプリングを導入し、最後にユビキタスであり、よく使用されるハッシュテーブルに導入しました。私たちは、その途中でいくつかの重要な教訓を学びました。制御フローの変更に伴うロックの獲得と解放に注意すること。並行性を高めることが必ずしもパフォーマンスを向上させることは限りません。そのパフォーマンスの問題は、それらが存在する場合にのみ修正する必要があります。この最後の点は、時期尚早な最適化を避けるパフォーマンスに配慮した開発者の中心です。最適化することでアプリケーションの全体的なパフォーマンスが改善されない場合には意味がありません。

もちろん、私たちは高性能構造の表面をみました。詳細については、Moir and Shavit の優れた調査と、他の情報源へのリンク [MS04] を参照してください。特に、他の構造 (Bツリーなど) に興味があるかもしれません。これはデータベースクラスが最善の策です。伝統的なロックをまったく使用しないテクニックに興味があるかもしれません。このようなノンブロッキングのデータ構造は、一般的な並行性のバグに関する章で味わうことができるのですが、率直に言えば、この謙虚な本よりも多くの研究を必要とする知識の全領域です。あなたが興味を持っていることは、いつものようにあなた自身でさらに調べてください。

参考文献

- [B+10] “An Analysis of Linux Scalability to Many Cores”
Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, Nickolai Zeldovich
OSDI ’10, Vancouver, Canada, October 2010
A great study of how Linux performs on multicore machines, as well as some simple solutions.
- [BH73] “Operating System Principles”
Per Brinch Hansen, Prentice-Hall, 1973
Available: <http://portal.acm.org/citation.cfm?id=540365>
One of the first books on operating systems; certainly ahead of its time. Introduced monitors as a concurrency primitive.
- [BC05] “Understanding the Linux Kernel (Third Edition)”
Daniel P. Bovet and Marco Cesati
O’Reilly Media, November 2005
The classic book on the Linux kernel. You should read it.
- [L+13] “A Study of Linux File System Evolution”
Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Shan Lu
FAST ’13, San Jose, CA, February 2013
Our paper that studies every patch to Linux file systems over nearly a decade. Lots of fun findings in there; read it to see! The work was painful to do though; the poor graduate student, Lanyue Lu, had to look through every single patch by hand in order to understand what they did.
- [MS98] “Nonblocking Algorithms and Preemption-safe Locking on Multiprogrammed Sharedmemory Multiprocessors”
M. Michael and M. Scott
Journal of Parallel and Distributed Computing, Vol. 51, No. 1, 1998
Professor Scott and his students have been at the forefront of concurrent algorithms and data structures for many years; check out his web page, numerous papers, or books to find out more.
- [MS04] “Concurrent Data Structures”

Mark Moir and Nir Shavit

In Handbook of Data Structures and Applications

(Editors D. Metha and S.Sahni)

Chapman and Hall/CRC Press, 2004

Available: www.cs.tau.ac.il/~shanir/concurrent-data-structures.pdf

A short but relatively comprehensive reference on concurrent data structures. Though it is missing some of the latest works in the area (due to its age), it remains an incredibly useful reference.

[MM00] “Solaris Internals: Core Kernel Architecture”

Jim Mauro and Richard McDougall

Prentice Hall, October 2000

The Solaris book. You should also read this, if you want to learn in great detail about something other than Linux.

[S+11] “Making the Common Case the Only Case with Anticipatory Memory Allocation”

Swaminathan Sundararaman, Yupu Zhang, Sriram Subramanian, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

FAST '11, San Jose, CA, February 2011

Our work on removing possibly-failing calls to malloc from kernel code paths. The idea is to allocate all potentially needed memory before doing any of the work, thus avoiding failure deep down in the storage stack.

30 Condition Variables

これまでのところ、ロックの概念を開発し、ハードウェアとOSの適切な組み合わせで適切に構築する方法を見てきました。残念ながら、ロックは、並行プログラムを構築するために必要な唯一の方法ではありません。

特に、スレッドは、実行を続行する前に条件が真であるかどうかをチェックしたい場合があります。たとえば、親スレッドは、子スレッドが完了する前に完了しているかどうかを確認したい場合があります（これはよく`join()`と呼ばれます）。どのようにそのような待って実装する必要がありますか？図30.1を見てみましょう。

```

1 void *child(void *arg) {
2     printf("child\n");
3     // XXX how to indicate we are done?
4     return NULL;
5 }
6
7 int main(int argc, char *argv[]) {
8     printf("parent: begin\n");
9     pthread_t c;
10    Pthread_create(&c, NULL, child, NULL); // create child
11    // XXX how to wait for child?
12    printf("parent: end\n");
13    return 0;
14 }
```

Figure 30.1: A Parent Waiting For Its Child

ここで見たいのは、次の出力です。

```

parent: begin
child
parent: end
```

図30.2に示すように、共有変数を使用してみることもできます。このソリューションは一般的には機能しますが、親が回転してCPU時間を無駄にするため、非常に非効率的です。ここで私たちが望むのは、私たちが待っている状態（例えば、子供が実行されている）が成立するまで、親を寝かせる何らかの方法です。

```

1  volatile int done = 0;
2
3  void *child(void *arg) {
4      printf("child\n");
5      done = 1;
6      return NULL;
7  }
8
9  int main(int argc, char *argv[]) {
10     printf("parent: begin\n");
11     pthread_t c;
12     Pthread_create(&c, NULL, child, NULL); // create child
13     while (done == 0)
14         ; // spin
15     printf("parent: end\n");
16     return 0;
17 }

```

Figure 30.2: Parent Waiting For Child: Spin-based Approach

THE CRUX: HOW TO WAIT FOR A CONDITION

マルチスレッドプログラムでは、スレッドが先に進む前に、ある条件が真となるのを待つことがしばしば役に立ちます。条件が真になるまで回転するという単純なアプローチは、非常に効率が悪く、CPU サイクルを浪費し、場合によっては正しくない可能性があります。したがって、スレッドはどのように条件を待つべきですか？

30.1 Definition and Routines

条件が真となるのを待つために、スレッドは条件変数として知られているものを利用することができます。条件変数は、ある実行状態(すなわち、ある条件)が(条件を待つことによって)望ましくないときに、スレッドが自分自身を置くことができる明示的なキューです。その状態を変更すると、他のスレッドは、それらの wait 中のスレッドのうちの 1 つ(またはそれ以上)をウェイクさせることができ、したがってそれらを継続して実行することができます(条件を通知することによって)。ダイクストラの「プライベートセマフォー」[D68]の使用に基づいています。類似のアイデアは後でホアレ(Hoare)によってモニター上の作業で「条件変数(condition variable)」と命名されました[H74]。

そのような条件変数を宣言するには、次のような文を書きます。pthread cond t c; これは条件変数として c を宣言します(注:適切な初期化も必要です) 条件変数には、wait() と signal() という 2 つの操作が関連付けられています。wait() 呼び出しはスレッドがスリープ状態にしたいときに実行されます。signal() コールは、スレッドがプログラム内で何かを変更したときに実行され、この状態で wait しているスリープ中のスレッドをスリープ解除するために必要です。具体的には、POSIX 呼び出しは次のようにになります。

```

pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);

```

わかりやすくするために、これらを wait() と signal() と呼ぶことがあります。wait() 呼び出しについて気付くかもしれないことの 1 つは、mutex もパラメータとして渡されることです。wait() が呼び出されたときにこの mutex がロックされているとみなします。wait() の役割は、ロックを解放し、呼び出しスレッドをスリープ状態(アトミック)にすることです。スレッドが起動したとき(他のスレッドがそれを通知し

た後)、呼び出し元に戻る前にロックを再取得する必要があります。この複雑さは、スレッドがスリープ状態になるときに特定の競合状態が発生しないようにするということに由来します。これをよりよく理解するために、結合問題(図30.3)の解を見てみましょう。

```

1  int done = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c = PTHREAD_COND_INITIALIZER;
4
5  void thr_exit() {
6      Pthread_mutex_lock(&m);
7      done = 1;
8      Pthread_cond_signal(&c);
9      Pthread_mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13     printf("child\n");
14     thr_exit();
15     return NULL;
16 }
17
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```

Figure 30.3: Parent Waiting For Child: Use A Condition Variable

考慮すべき2つのケースがあります。最初は、親スレッドは子スレッドを作成しますが、自分自身を実行し続けます(单一のプロセッサしか持っていないと仮定しています)。すぐに `thr_join()` を呼び出して子スレッドが完了するのを待ちます。この場合、ロックを取得し、子が完了しているかどうかをチェックし、子が完了していないかどうかを確認し、`wait()` を呼び出すことでロックを解除します。子プロセスは最終的に実行され、メッセージ“child”を出力し、`thr_exit()` を呼び出して親スレッドを起動します。このコードはロックを取得し、状態変数を設定し、親に信号を送り、それを目覚めさせるだけです。最後に、親プロセスが実行され(ロックが保持された状態で `wait()` から戻る)ロックが解除され、最後のメッセージ“parent: end”が出力されます。

2番目のケースでは、子は作成直後に実行され、1に設定されたセットはスリープしているスレッドをスリープ解除するための `signal` を呼び出します(ただし、何もないで返されます)。親が実行され、`thr_join()` が呼び出され、`done` が1であるとみなされるため、`wait`せずに戻ります。

最後の注意点: 条件を待つかどうかを決定する際には、親が `if` 文の代わりに `while` ループを使用することがわかります。これはプログラムのロジックごとに厳密には必要ではないようですが、以下に示すように、常に良いアイデアです。

`exit()` と `thr_join()` の各コードの重要性を理解するために、いくつかの代替実装を試してみましょう。まず、状態変数が必要かどうか疑問に思うかもしれません。コードが下の例のように見える場合はどうなりま

すか？ これは効果がありますか？

```

1 void thr_exit () {
2     Pthread_mutex_lock (&m);
3     Pthread_cond_signal (&c);
4     Pthread_mutex_unlock (&m);
5 }
6
7 void thr_join () {
8     Pthread_mutex_lock (&m);
9     Pthread_cond_wait (&c, &m);
10    Pthread_mutex_unlock (&m);
11 }
```

残念ながら、このアプローチではうまくいきません。子がすぐに実行され、すぐに `exit()` を呼び出した場合を想像してください。この場合、子供は `signal` をだしますが、この状態では眠っているスレッドはありません。親が実行されると、単に `wait` を呼び出してスタックします。スレッドはそれを起こすことはありません。この例から、状態変数の重要性を理解する必要があります。スレッドが知りたい値を記録します。`sleep`、`wake`、そしてすべてをロックすることは、その周りに構築されます。

ここに別の貧弱な実装があります。この例では、`signal` と `wait` のためにロックを保持する必要がないと想定しています。どのような問題がここで起こるでしょうか？ それについて考えてみましょう！

```

1 void thr_exit () {
2     done = 1;
3     Pthread_cond_signal (&c);
4 }
5
6 void thr_join () {
7     if (done == 0)
8         Pthread_cond_wait (&c);
9 }
```

ここでの問題は微妙な競争条件です。具体的には、親が `thr_join()` を呼び出して `done` の値をチェックすると、それが 0 であることがわかり、スリープ状態になります。しかし、それが寝るのを待つ直前に、親は中断され、子供は走ります。子はステート変数 `done` を 1 に変更してシグナルを送りますが、スレッドは待機していないため、スレッドは起動しません。親が再び動くとき、永遠に眠ります。

TIP: ALWAYS HOLD THE LOCK WHILE SIGNALING

すべてのケースで厳密には必要ではありませんが、条件変数を使用するときに signal を送信している間は、ロックを保持するのが最も簡単で最善の方法です。上記の例では、ロックを正しい状態に保つ必要がある場合を示しています。しかし、そうでない可能性が高い他のケースがいくつかあります、避けるべきことが多くあります。したがって、簡単にするために、信号を呼び出すときにロックを保持してください。

このヒントの逆、つまり wait を呼び出すときにロックを保持するのは、ヒントだけでなく、wait のセマンティクスによって義務づけられています。常に wait(a) はロックを保持しているとみなし、(b) 呼び出し元をスリープ状態にするときにロックを解除し、(c) 復帰する直前にロックを再獲得することを特徴とする。したがって、この tip の一般化は正しいです。signal または wait を呼び出すときにロックを保持すると、常に良い形になります。

うまくいけば、この単純な結合の例から、条件変数を適切に使用する基本的な要件の一部を見ることができます。あなたが理解していることを確かめるために、より複雑な例、つまりプロデューサ/コンシューマまたは有限バッファ問題（有限なバッファの問題）を調べます。

30.2 The Producer/Consumer (Bounded Buffer) Problem

この章で直面する次の同期の問題は、プロデューサ/コンシューマの問題、またはダイクストラ [D72] によって最初に提起された有限のあるバッファの問題（有界バッファ）として知られています。実際、ダイクストラと彼の同僚に一般化されたセマフォ（ロック変数または条件変数のいずれかとして使用することができる）を発明するのは、まさにプロデューサ/コンシューマの問題でした [D01]。セマフォについて後で詳しく説明します。

1つ以上のプロデューサスレッドと1つ以上のコンシューマスレッドを想像してください。プロデューサはデータ項目を生成し、バッファに格納します。コンシューマはバッファからアイテムを取り出し、何らかの方法でそれらを消費します。この配置は、多くの実際のシステムで発生します。例えば、マルチスレッド・ウェブ・サーバでは、プロデューサは、HTTP 要求を作業キュー（すなわち、有限バッファ）に入れます。コンシューマスレッドはこのキューから要求を取り出して処理します。

有界バッファは、あるプログラムの出力を別のプログラムにパイプするときにも使用されます（例：grep foo file.txt | wc -l）この例では、2つのプロセスを同時に実行します。grep は、file.txt の文字列を foo という文字列で標準出力とみなして書き込みます。UNIX シェルは、パイプシステムコールによって作成された UNIX パイプと呼ばれるものに出力をリダイレクトします。このパイプのもう一方の端はプロセス wc の標準入力に接続されています。これは単純に入力ストリームの行数を数え、その結果を出力します。したがって、grep プロセスはプロデューサです。wc プロセスはコンシューマです。それらの間にはカーネル内の有界バッファがあります。この例では、幸せなユーザーだけです。

有界バッファは共有リソースであるため、競合状態が発生しないように、同期アクセスを要求する必要があります。この問題をよりよく理解するには、実際のコードを調べてみましょう。最初に必要なのは、共有バッファで、プロデューサがデータを入れ、コンシューマがデータを取り込みます。単純に单一の整数を使用してみましょう（代わりに、このスロットにデータ構造体へのポインタを置くことを想像してください）そして、2つの内部ルーチンが値を共有バッファに入れ、バッファから値を取得します。詳細は図 30.4 を参照してください。

```

1 int buffer;
2 int count = 0; // initially, empty
3
4 void put(int value) {
5     assert(count == 0);
6     count = 1;
7     buffer = value;
8 }
9
10 int get() {
11     assert(count == 1);
12     count = 0;
13     return buffer;
14 }

```

Figure 30.4: The Put And Get Routines (Version 1)

かなりシンプルでしょう？ `put()` ルーチンは、バッファが空であるとみなし（アサーションでこれをチェックする）、共有バッファに値を格納し、`count` を 1 に設定してバッファをフルにします。`get()` ルーチンは反対にバッファを空にして値を返します（つまり、カウントを 0 に設定します）この共有バッファには 1 つのエントリしかないと心配しないでください。後で、複数のエントリを保持できるキューに一般化します。

バッファにアクセスしてデータを入れたり、そこからデータを取り出したりするために、いつバッファにアクセスするのが良いかを知るルーチンを書く必要があります。これは、カウントがゼロのとき（すなわち、バッファが空のとき）のみデータをバッファに入れ、カウントが 1 のとき（すなわち、バッファがいっぱいのとき）にのみデータをバッファから取得するという条件が明らかでなければならない。そうでなければ、プロデューサがデータをフルバッファに入れるか、コンシューマが空のデータからデータを取得するような同期コードを書くといった、何か間違ったことが起こります（このコードではアサーションが発生します）。

この作業は、2 種類のスレッドによって行われます。1 つはプロデューサスレッドと呼ばれ、もう 1 つはコンシューマスレッドと呼ばれます。図 30.5 は、共有バッファループ回数に整数を代入するプロデューサのコードと、共有バッファから引き出されたデータ項目を出力するたびにその共有バッファからデータを取得する（永久に）コンシューマを示しています。

```

1 void *producer(void *arg) {
2     int i;
3     int loops = (int) arg;
4     for (i = 0; i < loops; i++) {
5         put(i);
6     }
7 }
8
9 void *consumer(void *arg) {
10    int i;
11    while (1) {
12        int tmp = get();
13        printf("%d\n", tmp);
14    }
15 }

```

Figure 30.5: Producer/Consumer Threads (Version 1)

A Broken Solution

今私たちが单一のプロデューサと単一のコンシューマを持っていると想像してください。明らかに `put()` と `get()` ルーチンは、`put()` がバッファを更新し、`get()` がそこから読み込むので、それらの中にクリティカルセクションを持っています。しかし、コードの周りにロックをかけることはできません。解決策としてもつと別の仕組みが必要です。驚くことではないが、何らかの条件変数があります。この(壊れた)最初の試行(図 30.6)では、単一の条件変数 `cond` と関連するロックミューテックスがあります。

```

1  cond_t cond;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);           // p1
8          if (count == 1)                      // p2
9              Pthread_cond_wait(&cond, &mutex); // p3
10         put(i);                          // p4
11         Pthread_cond_signal(&cond);       // p5
12         Pthread_mutex_unlock(&mutex);     // p6
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);           // c1
20         if (count == 0)                      // c2
21             Pthread_cond_wait(&cond, &mutex); // c3
22         int tmp = get();                  // c4
23         Pthread_cond_signal(&cond);       // c5
24         Pthread_mutex_unlock(&mutex);     // c6
25         printf("%d\n", tmp);
26     }
27 }
```

Figure 30.6: Producer/Consumer: Single CV And If Statement

プロデューサーとコンシューマの間のシグナルロジックを調べてみましょう。プロデューサーはバッファを埋めることを望むとき、それが空であるのを待ちます(p1-p3) コンシューマはまったく同じロジックを持っていますが、満腹度(c1~c3)の異なる状態を待ちます。単一のプロデューサと単一のコンシューマだけでは、図 30.6 のコードが機能します。しかしながら、これらのスレッド(例えば、2つのコンシューマ)のうちの2つ以上を有する場合、解決策は2つの重大な問題があります。それらは一体何でしょうか？

...(考えるためにここで休憩)...

最初の問題を理解しましょう。待つ前に `if` ステートメントと関係があります。2人のコンシューマ(`Tc1`と`Tc2`)と1人のプロデューサー(`Tp`)が存在すると仮定します。まず、コンシューマ(`Tc1`)が走ります。ロック(c1)を取得し、使用準備ができるバッファがあるかどうかを確認し(c2)、存在しないことを確認し、ロックを解除する(c3)のを待ちます。

その後、プロデューサー(`Tp`)が実行されます。ロック(p1)を取得し、すべてのバッファが満杯(p2)であるかどうかを調べ、そうでないと判断した場合は、先に進みバッファ(p4)を埋めます。次に、プロデューサーは、バッファが満たされたことを通知します(p5)。クリティカルな部分は、最初のコンシューマ(`Tc1`)が条件変数をスリープ状態からレディキューに移動します。`Tc1`は今実行することができます(ただし、まだ実行さ

れていません)。プロデューサは、バッファが一杯になったことが実現するまで続きます。バッファがいっぱいになると、その時点ですリープします (p6、p1-p3)。

問題が発生した場所は次のとおりです。別のコンシューマ (T_{c2}) がバッファ内に存在する 1 つの既存の値 ($c1, c2, c4, c5, c6$ 、バッファがいっぱいなので $c3$ の待機をスキップ) 今度は T_{c1} が動作すると仮定します。待機から戻ってくる直前に、ロックを再取得してからリターンします。それから、`get()`($c4$) を呼び出しますが、消費するバッファはありません！ アクションがトリガされ、コードは必要に応じて機能しませんでした。見てわかるように、 T_{c2} が消費してきたため、 T_{c1} が消費しようとしてすることを何らかの形で妨げていたはずです。図 30.7 は、各スレッドが実行するアクションと、そのスケジューラの状態 (Ready、Running、Sleeping) を経時的に示しています。

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
$c1$	Running		Ready		Ready	0	
$c2$	Running		Ready		Ready	0	
$c3$	Sleep		Ready		Ready	0	
	Sleep		Ready	p1	Running	0	
	Sleep		Ready	p2	Running	0	
	Sleep		Ready	p4	Running	1	
	Ready		Ready	p5	Running	1	
	Ready		Ready	p6	Running	1	
	Ready		Ready	p1	Running	1	
	Ready		Ready	p2	Running	1	
	Ready		Ready	p3	Sleep	1	
	Ready	$c1$	Running		Sleep	1	T_{c2} sneaks in ...
	Ready	$c2$	Running		Sleep	1	
	Ready	$c4$	Running		Sleep	0	... and grabs data
	Ready	$c5$	Running		Ready	0	T_p awoken
	Ready	$c6$	Running		Ready	0	
$c4$	Running		Ready		Ready	0	Oh oh! No data

Figure 30.7: Thread Trace: Broken Solution (Version 1)

簡単な理由で問題が発生します。プロデューサが T_{c1} を覚ました後、 T_{c1} が実行される前に、バインドされたバッファの状態が変更されました (T_{c2} のおかげで)。スレッドへのシグナリングは、それらを目覚めさせるだけです。したがって、状態 (この場合、バッファに値が設定されている) が変更されましたが、起きたスレッドが実行されても状態は希望どおりに保たれるという保証はありません。このような方法で条件変数を構築した最初の研究の後に、信号が意味することのこの解釈は、しばしばメサの意味論と呼ばれます (LR80)。Hoare セマンティクスと呼ばれるコントラストは構築するのが難しいですが、目覚めたスレッドが目覚めた直後に実行されるという強力な保証を提供します [H74]。実際に構築されたすべてのシステムは、メサのセマンティクスを採用しています。

Better, But Still Broken: While, Not If

幸いにも、この修正は簡単です (図 30.8) : if を while に変更してください。なぜこれが働くか考えてみてください。今度はコンシューマ T_{c1} が起動し、(ロックを保持して) 共有変数 ($c2$) の状態を直ちに再チェックします。その時点でバッファが空の場合、コンシューマは単にスリープ状態に戻ります ($c3$)。当然ながら、プロデューサ ($p2$) も if から while に変更されます。

```

1  cond_t cond;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex); // p1
8          while (count == 1) // p2
9              Pthread_cond_wait(&cond, &mutex); // p3
10         put(i); // p4
11         Pthread_cond_signal(&cond); // p5
12         Pthread_mutex_unlock(&mutex); // p6
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex); // c1
20         while (count == 0) // c2
21             Pthread_cond_wait(&cond, &mutex); // c3
22         int tmp = get(); // c4
23         Pthread_cond_signal(&cond); // c5
24         Pthread_mutex_unlock(&mutex); // c6
25         printf("%d\n", tmp);
26     }
27 }

```

Figure 30.8: Producer/Consumer: Single CV And While

メサのセマンティクスのおかげで、条件変数で覚えておく簡単なルールは、常に while ループを使用することです。場合によっては、条件を再確認する必要はありませんが、いつでも行うことができます。(そうすることが確実です)しかし、このコードにはまだバグがあり、上記の 2 つの問題の 2 番目のものです。わかりますか？ 条件変数が 1 つしかないという事実と関係があります。先読みをする前に、問題が何であるかを把握してみてください。

…(もう一度、自分で考えてください。また、少し目を閉じて考えてみてください)…

あなたがそれを正しく理解したことを確認しましょう。この問題は、2 つのコンシューマが最初に起動し (Tc1 と Tc2)、両方ともスリープ状態になる (c3) 場合に発生します。次に、プロデューサが実行され、バッファに値が格納され、コンシューマの 1 つを起動させます (Tc1 など)。次に、プロデューサはループバック (途中でロックの解除と再取得) を行い、バッファにさらにデータを入れようとします。バッファがいっぱいであるため、代わりにプロデューサは条件を待機します (したがってスリープします)。これで、1 つのコンシューマは実行準備が整っており (Tc1)、2 つのスレッドが 1 つの条件 (Tc2 と Tp) でスリープしています。私たちは問題を起こそうとしています。物事はエキサイティングになっています！

次に、コンシューマ Tc1 は、`wait()(c3)` から復帰してウェイクし、条件 (c2) を再確認し、バッファが満杯であることを見つけると、値 (c4) を消費します。このコンシューマは、クリティカル条件前で待機中の唯一のスレッドを起こし、条件 (c5) にシグナルを出します。しかし、どのスレッドが目を覚ますべきですか？

コンシューマはバッファを空にしたので、明らかにプロデューサーを目覚めさせるべきです。しかし、それがコンシューマ Tc2 を目覚めさせた場合（確かに待ち行列がどのように管理されるかによっては可能ですが）、問題があります。具体的には、コンシューマ Tc2 は起こしてバッファを空にし (c2)、スリープ状態に戻ります (c3)。バッファに入れる値を持つプロデューサ Tp はスリープ状態になります。他のコンシューマスレッド Tc1 もスリープ状態に戻ります。3 つのスレッドはすべてスリープ状態になっています。この悲しいステップ

については、図 30.9 を参照してください。シグナリングは明らかに必要ですが、より直接的でなければなりません。コンシューマは、他のコンシューマやプロデューサだけを目覚めさせてはならず、またその逆もあってはいけません。

The Single Buffer Producer/Consumer Solution

システムの状態が変化したときにどのタイプのスレッドが起きるべきかを適切に伝えるために、1つではなく2つの条件変数を使用します。結果のコードを図 30.10 に示します。

```

1  cond_t empty, fill;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);
8          while (count == 1)
9              Pthread_cond_wait(&empty, &mutex);
10         put(i);
11         Pthread_cond_signal(&fill);
12         Pthread_mutex_unlock(&mutex);
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);
20         while (count == 0)
21             Pthread_cond_wait(&fill, &mutex);
22         int tmp = get();
23         Pthread_cond_signal(&empty);
24         Pthread_mutex_unlock(&mutex);
25         printf("%d\n", tmp);
26     }
27 }
```

Figure 30.10: Producer/Consumer: Two CVs And While

上のコードでは、プロデューサスレッドは空の状態で待機し、シグナルがいっぱいになります。逆に、コンシューマスレッドはいっぱいになるのを待ってシグナルを空にします。こうすることで、上記の第 2 の問題は、コンシューマが誤ってコンシューマを目覚めさせることはなく、プロデューサーは誤ってプロデューサーを目覚めさせることはありません。

The Correct Producer/Consumer Solution

私たちは現在、完全に一般的なものではありませんが、実際のプロデューサ/コンシューマのソリューションを持っています。最後に行う変更は、より並行性と効率性を実現することです。具体的には、バッファースロットを追加して、スリープする前に複数の値を生成し、同様にスリープする前に複数の値を消費することができます。単一のプロデューサとコンシューマだけでは、このアプローチはコンテクストスイッチを減らすので効率的です。複数のプロデューサまたはコンシューマ（またはその両方）を使用すると、同時の生成または消費が可能になり、並行性が向上します。幸いにも、それは現在のソリューションの小さな変更です。

この正しい解決策の最初の変更は、バッファ構造自体とそれに対応する `put()` および `get()` (図 30.11) 内に

あります。また、sleep の有無を判断するために、プロデューサとコンシューマが確認する条件を少し変更します。図 30.12 は、正しい待機およびシグナリングロジックを示しています。プロデューサは、すべてのバッファが現在いっぱいになるとスリープします (p2)。同様に、コンシューマは、すべてのバッファが現在空である場合にのみスリープする (c2)。それで、私たちはプロデューサ/コンシューマの問題を解決します。

```

1  int buffer[MAX];
2  int fill_ptr = 0;
3  int use_ptr = 0;
4  int count = 0;
5
6  void put(int value) {
7      buffer[fill_ptr] = value;
8      fill_ptr = (fill_ptr + 1) % MAX;
9      count++;
10 }
11
12 int get() {
13     int tmp = buffer[use_ptr];
14     use_ptr = (use_ptr + 1) % MAX;
15     count--;
16     return tmp;
17 }
```

Figure 30.11: The Correct Put And Get Routines

```

1 cond_t empty, fill;
2 mutex_t mutex;
3
4 void *producer(void *arg) {
5     int i;
6     for (i = 0; i < loops; i++) {
7         Pthread_mutex_lock(&mutex); // p1
8         while (count == MAX) // p2
9             Pthread_cond_wait(&empty, &mutex); // p3
10        put(i); // p4
11        Pthread_cond_signal(&fill); // p5
12        Pthread_mutex_unlock(&mutex); // p6
13    }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex); // c1
20         while (count == 0) // c2
21             Pthread_cond_wait(&fill, &mutex); // c3
22         int tmp = get(); // c4
23         Pthread_cond_signal(&empty); // c5
24         Pthread_mutex_unlock(&mutex); // c6
25         printf("%d\n", tmp);
26    }
27 }
```

Figure 30.12: The Correct Producer/Consumer Synchronization

TIP: USE WHILE (NOT IF) FOR CONDITIONS

マルチスレッドプログラムで条件をチェックするときは、while ループを使用することは常に正し

いです。if 文を使用するのは、シグナリングのセマンティクスに依存します。したがって、常に while を使用すると、コードが期待通りに動作します。

条件付きチェックの while ループを使用すると、疑似 wakeups が発生するケースも処理されます。いくつかのスレッドパッケージでは、実装の詳細により、ただ 1 つの signal が発生したにもかかわらず、2 つのスレッドが起動する可能性があります [L11]。疑似 wakeups は、スレッドが待機している状態を再確認するもう 1 つの理由です。

30.3 Covering Conditions

次に、条件変数の使用方法のもう 1 つの例を見てみましょう。このコードの研究は、先に説明した Mesa セマンティクスを最初に実装した同じグループの Pilot [LR80] の Lampson と Redell の論文 (彼らが使用した言語は Mesa なので、その名前) から引き出されています。彼らが遭遇した問題は、シンプルな例 (この場合は単純なマルチスレッドメモリ割り当てライブラリ) で最もよく分かります。図 30.13 に問題を示すコードスニペットを示します。

```

1 // how many bytes of the heap are free?
2 int bytesLeft = MAX_HEAP_SIZE;
3
4 // need lock and condition too
5 cond_t c;
6 mutex_t m;
7
8 void *
9 allocate(int size) {
10     Pthread_mutex_lock(&m);
11     while (bytesLeft < size)
12         Pthread_cond_wait(&c, &m);
13     void *ptr = ...; // get mem from heap
14     bytesLeft -= size;
15     Pthread_mutex_unlock(&m);
16     return ptr;
17 }
18
19 void free(void *ptr, int size) {
20     Pthread_mutex_lock(&m);
21     bytesLeft += size;
22     Pthread_cond_signal(&c); // whom to signal??
23     Pthread_mutex_unlock(&m);
24 }
```

Figure 30.13: Covering Conditions: An Example

コードで見てきたように、スレッドがメモリ割り当てコードを呼び出すと、より多くのメモリが解放されるのを待たなければならないかもしれません。逆に、スレッドがメモリを解放すると、より多くのメモリが解放されたことを通知します。しかし、上のコードでは問題があります。待機中のスレッド (複数のスレッドが存在する可能性があります) を起動する必要がありますか？

以下のシナリオを考えてみましょう。0 バイトの空きがあると仮定します。スレッド Ta は allocate(100) を呼び出し、スレッド Tb は allocate(10) を呼び出してメモリを少なくするよう要求します。こうして Ta と Tb の両方がこの状態を待って眠りにつきます。これらの要求のいずれかを満たすのに十分な空きバイトがありま

せん。

その時点でのスレッド Tc が free(50) を呼び出すと仮定します。残念ながら、待機中のスレッドをスリープ解除するためのシグナルを呼び出すと、解放されるのを 10 バイトだけ待っている正しい待機中のスレッド Tb をスリープ解除できないことがあります。十分な記憶がまだ空いていないので、Ta は待っておくべきである。したがって、図のコードは動作しません。他のスレッドを起動するスレッドは、起動するスレッド（またはスレッド）を認識しません。

Lampson と Redell によって提案された解決方法は簡単です。上のコードの `pthread_cond_signal()` 呼び出しを、`pthread_cond_broadcast()` の呼び出しで置き換えてください。そうすることで、目覚めすべきスレッドがあることを保証します。もちろん、（まだ）目を覚ますべきではない他の多くの待機スレッドを必要に起動させる可能性があるので、欠点はパフォーマンスに悪影響を与える可能性があります。これらのスレッドは単に起床し、条件を再確認してすぐにスリープ状態に戻ります。

Lampson と Redell は、スレッドが目を覚ます必要があるすべてのケース（控えめに）をカバーするため、このような条件をカバー条件と呼びます。これまで述べてきたように、コストはあまりにも多くのスレッドが起きる可能性があります。鋭い読者は、このアプローチを早期に使用することができたことに気づいているかもしれません（ただ一つの条件変数でプロデューサ/コンシューマの問題を参照してください）。しかし、その場合、より良い解決策が利用でき、それを使用しました。一般に、あなたのプログラムがあなたのシグナルをブロードキャストに変更したときにのみ動作することがわかった場合（ただし、必ずしもそうする必要はないです）、おそらくバグがあれば修正しましょう。しかし、上記のメモリアロケータのような場合、ブロードキャストは利用可能な最も簡単な解決策かもしれません。

30.4 Summary

ロック以外の重要な同期プリミティブ、条件変数の導入を見てきました。いくつかのプログラム状態が望ましくないときにスレッドをスリープさせることにより、CVs は、有名な（そして依然として重要な）プロデューサ/コンシューマの問題やカバー条件を含むいくつかの重要な同期問題をきれいに解決することができます。「彼はビッグブラザーを愛しました」[O49] など、より劇的な結論文がここに来るでしょう。

参考文献

- [D68] “Cooperating sequential processes”
Edsger W. Dijkstra, 1968
Available: <http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF>
Another classic from Dijkstra; reading his early works on concurrency will teach you much of what you need to know.
- [D72] “Information Streams Sharing a Finite Buffer”
E.W. Dijkstra
Information Processing Letters 1: 179180, 1972
Available: <http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD329.PDF>
The famous paper that introduced the producer/consumer problem.
- [D01] “My recollections of operating system design”
E.W. Dijkstra
April, 2001
Available: <http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1303.PDF>
A fascinating read for those of you interested in how the pioneers of our field came up with some very basic and fundamental concepts, including ideas like “interrupts” and even “a stack”!

[H74] “Monitors: An Operating System Structuring Concept”

C.A.R. Hoare

Communications of the ACM, 17:10, pages 549–557, October 1974

Hoare did a fair amount of theoretical work in concurrency. However, he is still probably most known for his work on Quicksort, the coolest sorting algorithm in the world, at least according to these authors.

[L11] “Pthread cond signal Man Page”

Available: http://linux.die.net/man/3/pthread_cond_signal

March, 2011

The Linux man page shows a nice simple example of why a thread might get a spurious wakeup, due to race conditions within the signal/wakeup code.

[LR80] “Experience with Processes and Monitors in Mesa”

B.W. Lampson, D.R. Redell

Communications of the ACM. 23:2, pages 105-117, February 1980

A terrific paper about how to actually implement signaling and condition variables in a real system, leading to the term “Mesa” semantics for what it means to be woken up; the older semantics, developed by Tony Hoare [H74], then became known as “Hoare” semantics, which is hard to say out loud in class with a straight face.

[O49] “1984”

George Orwell, 1949, Secker and Warburg

A little heavy-handed, but of course a must read. That said, we kind of gave away the ending by quoting the last sentence. Sorry! And if the government is reading this, let us just say that we think that the government is “double plus good”. Hear that, our pals at the NSA?

31 Semaphores

今のところわかっているように、関連性のある興味深い並行性の問題を解決するためには、ロックと条件変数の両方が必要です。この数年前に実現した最初の人のひとりは、Edsger Dijkstra(正確な歴史 [GR92] を知ることは難しいが) である。グラフ理論 [D59] の有名な「最短経路」アルゴリズムで知られている。「Goto Statements Considered Harmful」D68a という名前の構造化プログラミングについて説明し、ここではセマフォー [D68b, D72] と呼ばれる同期プリミティブの導入を検討します。実際、Dijkstra らは、セマフォを同期に関連するすべてのものの単一のプリミティブとして考案しました。表示されるように、セマフォはロックと条件変数の両方として使用できます。

THE CRUX: HOW TO USE SEMAPHORES

ロックと条件変数の代わりにセマフォを使用するにはどうすればよいですか？ セマフォの定義は何ですか？ バイナリセマフォとは何ですか？ ロックと条件変数からセマフォーを構築するのは簡単ですか？ セマフォからロックと条件変数を構築するにはどうすればよいでしょう？

31.1 Semaphores: A Definition

セマフォは、2つのルーチンで操作できる整数値を持つオブジェクトです。POSIX 標準では、これらのルーチンは `sem_wait()` と `sem_post()` です。セマフォの初期値は、セマフォと相互作用する他のルーチンを呼び出す前に、その動作を決定するため、図 31.1 のコードと同じように、最初にセマフォをある値に初期化する必要があります。

```

1 #include <semaphore.h>
2 sem_t s;
3 sem_init(&s, 0, 1);

```

Figure 31.1: Initializing A Semaphore

この図では、セマフォ `s` を宣言し、3番目の引数として 1 を渡して値 1 に初期化します。`sem_init()` の 2 番目の引数は、すべての例で 0 に設定されています。セマフォが同じプロセス内のスレッド間で共有されていることを示します。セマフォーの他の使用法(つまり、異なるプロセス間でアクセスを同期させるためにそれらをどのように使用することができるか)の詳細については、第 2 引数の値が異なる必要があります。

セマフォが初期化された後、`sem_wait()` または `sem_post()` の 2 つの関数のいずれかを呼び出すことができます。これらの 2 つの機能の動作を図 31.2 に示します。

```

1 int sem_wait(sem_t *s) {
2     decrement the value of semaphore s by one
3     wait if value of semaphore s is negative
4 }
5
6 int sem_post(sem_t *s) {
7     increment the value of semaphore s by one
8     if there are one or more threads waiting, wake one
9 }

```

Figure 31.2: Semaphore: Definitions Of Wait And Post

今のところ、これらのルーチンの実装には関心がありません。`sem_wait()` と `sem_post()` を呼び出す複数

のスレッドでは、これらのクリティカルセクションを管理するために必要があることは明らかです。ここでは、これらのプリミティブの使用方法に焦点を当てます。後に、それらがどのように構築されるかを議論するかもしれません。

ここではインターフェイスのいくつかの顕著な側面について議論する必要があります。`sem_wait()` は `sem_wait()` を呼び出したときにセマフォの値が 1 以上だったためすぐに返されるか、呼び出し側が後続のポストを待って実行を中断させます。もちろん、複数の呼び出しスレッドが `sem_wait()` を呼び出すことがあります。したがって、すべてが呼び出されるのを待ってキューに入れられます。

第 2 に、`sem_post()` は `sem_wait()` のように特定の条件が成立するのを待たないことがわかります。むしろ、単にセマフォの値をインクリメントし、次に目覚めようとしているスレッドがあれば、それらの 1 つを起動させます。

第 3 に、セマフォの値が負の場合、待機スレッドの数に等しい [D68b]。この値は一般的にセマフォのユーザには見られませんが、知る価値があり、セマフォがどのように機能するかを覚えておくのに役立ちます。

セマフォ一内で可能な競合状態を心配しないでください。それらのアクションがアトミックに実行されると仮定します。私たちはこれを行うためにロックと条件変数を使用します。

31.2 Binary Semaphores (Locks)

セマフォを使用する準備ができました。私たちの最初の使い方は、すでによく知られているものです。それは、セマフォをロックとして使用します。コードスニペットについては、図 31.3 を参照してください。重要なセクションを `sem_wait()`/`sem_post()` のペアで囲むだけであることがわかります。ただし、この作業を行う上で重要なのは、セマフォ `m` の初期値です(図の X に初期化されています)。X は何をすべきですか？

```

1  sem_t m;
2  sem_init(&m, 0, X); // initialize semaphore to X; what should X be?
3
4  sem_wait(&m);
5  // critical section here
6  sem_post(&m);

```

Figure 31.3: A Binary Semaphore (That Is, A Lock)

... (先に進む前にそれについて考えてみてください)...

上記の `sem_wait()` と `sem_post()` ルーチンの定義を振り返ってみると、初期値は 1 であるはずです。

これを明確にするために、2 つのスレッドを持つシナリオを想像してみましょう。最初のスレッド(スレッド 0)は `sem_wait()` を呼び出します。セマフォの値を最初にデクリメントし、0 に変更します。その後、値が 0 以上でない場合にのみ待機します。値が 0 であるため、`sem_wait()` は単に戻り値を返しますし継続します。スレッド 0 はクリティカルセクションに自由に入ります。スレッド 0 がクリティカルセクション内にある間に他のスレッドがロックを取得しようとしている場合、`sem_post()` を呼び出すと、セマフォの値を 1 に復元します(存在しないため待機スレッドを起動しません)。図 31.4 に、このシナリオのトレースを示します。

Value of Semaphore	Thread 0	Thread 1
1		
1	call sem_wait()	
0	sem_wait() returns	
0	(crit sect)	
0	call sem_post()	
1	sem_post() returns	

Figure 31.4: Thread Trace: Single Thread Using A Semaphore

スレッド 0 がロックを保持し (つまり `sem_wait()` はまだ `sem_post()` と呼ばれていない)、別のスレッド (スレッド 1)`sem_wait()` を呼び出してクリティカルセクションに入ることを試みているとき、スレッド 1 はセマフォの値を減らして -1 にし、待機します (プロセッサをスリープさせて放棄する) スレッド 0 が再び実行されると、最終的に `sem_post()` が呼び出され、セマフォの値がゼロに戻されて待機スレッド (スレッド 1) が起動され、ロックが獲得されます。スレッド 1 が終了すると、セマフォの値が再びインクリメントされ、再び 1 に戻されます。

図 31.5 に、この例のトレースを示します。スレッド動作に加えて、図は各スレッドのスケジューラ状態、すなわち実行中、実行可能 (実行可能であるが実行中ではない)、およびスリープを示しています。特に、すでに保持されているロックを取得しようとすると、スレッド 1 はスリープ状態になります。スレッド 0 が再び実行された場合にのみ、スレッド 1 を起動し、潜在的に再び実行することができます。

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call sem_wait()	Running		Ready
0	sem_wait() returns	Running		Ready
0	(crit sect: begin)	Running		Ready
0	Interrupt; Switch→T1	Ready		Running
0		Ready	call sem_wait()	Running
-1		Ready	decrement sem	Running
-1		Ready	(sem<0) → sleep	Sleeping
-1		Running	Switch→T0	Sleeping
-1	(crit sect: end)	Running		Sleeping
-1	call sem_post()	Running		Sleeping
0	increment sem	Running		Sleeping
0	wake (T1)	Running		Ready
0	sem_post() returns	Running		Ready
0	Interrupt; Switch→T1	Ready		Running
0		Ready	sem_wait() returns	Running
0		Ready	(crit sect)	Running
0		Ready	call sem_post()	Running
1		Ready	sem_post() returns	Running

Figure 31.5: Thread Trace: Two Threads Using A Semaphore

独自の例を使用して作業したい場合は、複数のスレッドがロックを待って待機するシナリオを試してみてください。このようなトレース中にセマフォの値はどのようになりますか？

このようなとき、セマフォをロックとして使用することができます。ロックは 2 つの状態 (保持され、保持されていない) しか持たないので、ロックとして使用されるセマフォをバイナリセマフォと呼ぶことがあります。注意として、このバイナリ形式でのみセマフォを使用している場合は、ここに示す汎用セマフォより簡単

な方法で実装できます。

31.3 Semaphores For Ordering

セマフォは、並行プログラムでイベントを順序付けるのにも役立ちます。例えば、スレッドは、リストが空でなくなるのを待つことを望むかもしれない、そこから要素を削除することができます。この使用パターンでは、あるスレッドが何か起こることを待っていて、別のスレッドが何か起こっていることを確認してから、それが起こったことをシグナルで待機スレッドを呼び起します。このようにして、セマフォを順序付けプリミティブとして使用しています(以前の条件変数の使用と同様)。

簡単な例は次のとおりです。スレッドが別のスレッドを作成し、実行を完了するのを待っているとします(図31.6)。このプログラムが実行されると、次の情報が表示されます。

```

1  sem_t s;
2
3  void *
4  child(void *arg) {
5      printf("child\n");
6      sem_post(&s); // signal here: child is done
7      return NULL;
8  }
9
10 int
11 main(int argc, char *argv[]) {
12     sem_init(&s, 0, X); // what should X be?
13     printf("parent: begin\n");
14     pthread_t c;
15     Pthread_create(&c, NULL, child, NULL);
16     sem_wait(&s); // wait here for child
17     printf("parent: end\n");
18     return 0;
19 }
```

Figure 31.6: A Parent Waiting For Its Child

```

parent: begin
child
parent: end
```

問題は、この効果を達成するためにセマフォを使用する方法です。それが判明したとき、答えは比較的理 解しやすいです。コードでわかるように、親は `sem_wait()` と子 `sem_post()` を呼び出して、実行を終了した子の状態が真になるのを待つだけです。しかし、これにより、このセマフォの初期値はどうなるべきかという疑問が生じます。

(やはり先読みではなく、ここで考えてみてください)

答えはもちろん、セマフォの値を 0 に設定する必要があります。考慮すべき 2 つのケースがあります。最初に、親が子を作成するが、子はまだ実行されていない(すなわち、準備完了キューに入っているが実行されていない)と仮定します。この場合(図31.7)、親は `sem_post()` を呼び出す前に `sem_wait()` を呼び出します。私たちは親が子供が走るのを待つことを望みます。これが起こる唯一の方法は、セマフォの値が 0 より大きい場合です。従って、0 が初期値になります。親が実行され、セマフォを減らして -1 にしてから、スリープします。子が最終的に実行されると、`sem_post()` を呼び出し、セマフォの値を 0 にインクリメントし、親をスリープ解除して `sem_wait()` から戻り、プログラムを終了します。

Value	Parent	State	Child	State
0	create(Child)	Running	(Child exists; is runnable)	Ready
0	call sem_wait()	Running		Ready
-1	decrement sem	Running		Ready
-1	(sem<0) → sleep	Sleeping		Ready
-1	Switch→Child	Sleeping	child runs	Running
-1		Sleeping	call sem_post()	Running
0		Sleeping	increment sem	Running
0		Ready	wake (Parent)	Running
0		Ready	sem_post() returns	Running
0		Ready	Interrupt; Switch→Parent	Ready
0	sem_wait() returns	Running		Ready

Figure 31.7: Thread Trace: Parent Waiting For Child (Case 1)

2番目のケース(図31.8、ページ364)は、親が `sem_wait()` を呼び出す前に子プロセスが完了するまで実行されたときに発生します。この場合、子プロセスは `sem_post()` を最初に呼び出し、セマフォの値を 0 から 1 にインクリメントします。親プロセスが実行されると、`sem_wait()` がコールされ、セマフォの値が 1 かどうかを検索します。そうだったとき、親はその値を(0に)デクリメントし、待たずに `sem_wait()` から戻り、望んだ効果を得ます。

Value	Parent	State	Child	State
0	create(Child)	Running	(Child exists; is runnable)	Ready
0	Interrupt; Switch→Child	Ready	child runs	Running
0		Ready	call sem_post()	Running
1		Ready	increment sem	Running
1		Ready	wake (nobody)	Running
1		Ready	sem_post() returns	Running
1	parent runs	Running	Interrupt; Switch→Parent	Ready
1	call sem_wait()	Running		Ready
0	decrement sem	Running		Ready
0	(sem≥0) → awake	Running		Ready
0	sem_wait() returns	Running		Ready

Figure 31.8: Thread Trace: Parent Waiting For Child (Case 2)

31.4 The Producer/Consumer (Bounded Buffer) Problem

この章で直面する次の問題は、プロデューサ/コンシューマの問題、時には限定されたバッファの問題[D72]として知られています。この問題は、前の章の条件変数で詳しく説明しています。詳細はこちらをご覧ください。

First Attempt

この問題を解決するための最初の試みは、空といっぱいという2つのセマフォを導入します。これらのスレッドは、バッファエントリが空になったとき、またはいっぱいになったときを示すために使用します。`put`と`get`ルーチンのコードは図31.9にあり、プロデューサとコンシューマの問題を解決するための試みは図31.10にあります。

```

1 int buffer[MAX];
2 int fill = 0;
3 int use = 0;
4
5 void put(int value) {
6     buffer[fill] = value;      // Line P1
7     fill = (fill + 1) % MAX; // Line P2
8 }
9
10 int get() {
11     int tmp = buffer[use];    // Line G1
12     use = (use + 1) % MAX;   // Line G2
13     return tmp;
14 }
```

Figure 31.9: The Put And Get Routines

```

1 sem_t empty;
2 sem_t full;
3
4 void *producer(void *arg) {
5     int i;
6     for (i = 0; i < loops; i++) {
7         sem_wait(&empty);           // Line P1
8         put(i);                  // Line P2
9         sem_post(&full);          // Line P3
10    }
11 }
12
13 void *consumer(void *arg) {
14     int i, tmp = 0;
15     while (tmp != -1) {
16         sem_wait(&full);        // Line C1
17         tmp = get();            // Line C2
18         sem_post(&empty);       // Line C3
19         printf("%d\n", tmp);
20     }
21 }
22
23 int main(int argc, char *argv[]) {
24     // ...
25     sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
26     sem_init(&full, 0, 0);    // ... and 0 are full
27     // ...
28 }
```

Figure 31.10: Adding The Full And Empty Conditions

この例では、プロデューサはまずデータを格納するためにバッファが空になるのを待ち、コンシューマはバッファを使用する前にバッファがいっぱいになるのを待ちます。最初に $MAX = 1$ (配列内に i という 1 つのバッファしかありません) と仮定し、これが機能するかどうかを見てみましょう。

再度、プロデューサとコンシューマの 2 つのスレッドがあると想像してください。単一の CPU で特定のシナリオを検討しましょう。コンシューマが最初に走ると仮定します。したがって、コンシューマは、`sem_wait(&full)` を呼び出す図 31.10 の C1 行目にヒットします。full は値 0 に初期化されているため、コールは full を減らして (-1)、コンシューマをロックし、必要に応じて別のスレッドが full で `sem_post()` を呼び出すのを待ちます。

プロデューサが実行されているとします。これは P1 行目でヒットし、`sem_wait(&empty)` ルーチンを呼

び出します。コンシューマとは異なり、empty は値 MAX(この場合は 1) に初期化されるため、プロデューサはこの行を介して継続します。したがって empty は 0 にデクリメントされ、プロデューサはデータ値をバッファの最初のエントリに入れます (P2 行目)。次に、プロデューサは P3 に進み、sem_post(& full) を呼び出し、full のセマフォの値を -1 から 0 に変更し、コンシューマを目覚めさせます (例えば、ブロックから準備完了に移動する)。

この場合、2 つのうちの 1 つが起こる可能性があります。プロデューサが走り続けると、ループして再び P1 行目にヒットします。ただし、empty のセマフォの値が 0 であるため、今回はブロックされます。プロデューサが中断される代わりに、コンシューマが実行を開始すると、sem_wait(& full)(C1 行目) が呼び出され、消費します。いずれの場合でも、望んだ挙動を達成します。

この同じ例をもっと多くのスレッドで試すことができます (複数のプロデューサや複数のコンシューマなど)。それでも動作するはずです。

ここで、MAX が 1 より大きい (たとえば MAX = 10) と想像してみましょう。この例では、複数のプロデューサと複数のコンシューマが存在すると仮定します。ここで、問題を抱えています。それは競争状態です。どこで発生するのか見えていますか? (しばらく時間をとって見てください) 見えない場合は、ヒントがあります。それは、put() と get() コード付近を詳しく見てください。

さて、問題を理解しましょう。2 人のプロデューサー (Pa と Pb) が put() をほぼ同時に呼び出すとします。プロデューサ Pa が最初に実行され、最初のバッファエントリ (F1 行目で fill = 0) を書き始めると仮定します。Pa が fill カウンタを 1 にインクリメントするチャンスを得る前に、それは中断されます。プロデューサー Pb が実行を開始し、F1 行目でバッファーの 0 番目の要素にもデータが書き込まれます。つまり、古いデータが上書きされます。これはよくありません。プロデューサからのデータが失われることは望ましくありません。

A Solution: Adding Mutual Exclusion

ご覧のとおり、ここで忘れてしまったのは相互排除です。バッファの充填とバッファへのインデックスのインクリメントはクリティカルセクションであるため、注意深く管理する必要があります。そのため、バイナリセマフォを使用していくつかのロックを追加しましょう。図 31.11 に私たちの試みを示します。

```

1  sem_t empty;
2  sem_t full;
3  sem_t mutex;
4
5  void *producer(void *arg) {
6      int i;
7      for (i = 0; i < loops; i++) {
8          sem_wait(&mutex);           // Line P0 (NEW LINE)
9          sem_wait(&empty);         // Line P1
10         put(i);                // Line P2
11         sem_post(&full);        // Line P3
12         sem_post(&mutex);       // Line P4 (NEW LINE)
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         sem_wait(&mutex);           // Line C0 (NEW LINE)
20         sem_wait(&full);          // Line C1
21         int tmp = get();          // Line C2
22         sem_post(&empty);         // Line C3
23         sem_post(&mutex);       // Line C4 (NEW LINE)
24         printf("%d\n", tmp);
25     }
26 }
27
28 int main(int argc, char *argv[]) {
29     // ...
30     sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
31     sem_init(&full, 0, 0);   // ... and 0 are full
32     sem_init(&mutex, 0, 1); // mutex=1 because it is a lock (NEW LINE)
33     // ...
34 }
```

Figure 31.11: Adding Mutual Exclusion (Incorrectly)

これで、NEW LINE コメントで示されているように、コードの `put()`/`get()` 部分全体にいくつかのロックが追加されました。それは正しいアイデアのように見えますが、実際はうまくいきません。どうしてでしょうか？ ヒントはデッドロックです。なぜデッドロックが発生するのでしょうか？ それを考えてみましょう。デッドロックが発生するケースを見つけてみましょう。プログラムがデッドロックするためにはどのような手順を実行する必要がありますか？

Avoiding Deadlock

さて、あなたは図を理解したので、ここに答えがあります。1つのプロデューサと1つのコンシューマの2つのスレッドを想像してください。まずコンシューマが最初に走ります。ミューテックス (C0 行目) を取得し、`full` セマフォ (c1 行目) で `sem_wait()` を呼び出します。まだデータがないため、この呼び出しによってコンシューマはブロックして CPU を生成します。重要なことですが、コンシューマはまだロックを保持しています。

プロデューサが実行されます。それは生産するデータを持っていて、それが動くことができれば、それはコンシューマのスレッドを起こすことができ、すべてがうまくいくでしょう。残念なことに、最初に行なうこととは、バイナリミューテックスセマフォ (P0 行目) に対して `sem_wait()` を呼び出すことです。ロックはすでに保持されています。したがって、プロデューサは現在も待っています。

ここには単純なサイクルがあります。コンシューマはミューテックスを保持し、誰かがフルシグナルするのを待っています。プロデューサはフルシグナルを送ることができます、ミューテックスを待っています。したがって、プロデューサとコンシューマはお互いに待っています。古典的なデッドロックです。

At Last, A Working Solution

この問題を解決するには、単にロックの範囲を小さくする必要があります。図 31.12 に正しい解を示します。ご覧のとおり、ミューテックスの取得と解放をクリティカルセクションのまわりで行うだけです。フルおよび空の待機およびシグナルコードは外部に残されます。その結果、マルチスレッドプログラムでよく使用されるパターンである単純で作業効率の良い有限バッファーが得られます。それを今理解してください。後でそれを使用してください。あなたは何年も私たちに感謝することでしょう。

```

1  sem_t empty;
2  sem_t full;
3  sem_t mutex;
4
5  void *producer(void *arg) {
6      int i;
7      for (i = 0; i < loops; i++) {
8          sem_wait(&empty);           // Line P1
9          sem_wait(&mutex);         // Line P1.5 (MOVED MUTEX HERE...)
10         put(i);                // Line P2
11         sem_post(&mutex);        // Line P2.5 (... AND HERE)
12         sem_post(&full);         // Line P3
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         sem_wait(&full);          // Line C1
20         sem_wait(&mutex);        // Line C1.5 (MOVED MUTEX HERE...)
21         int tmp = get();         // Line C2
22         sem_post(&mutex);        // Line C2.5 (... AND HERE)
23         sem_post(&empty);         // Line C3
24         printf("%d\n", tmp);
25     }
26 }
27
28 int main(int argc, char *argv[]) {
29     ...
30     sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
31     sem_init(&full, 0, 0);   // ... and 0 are full
32     sem_init(&mutex, 0, 1);  // mutex=1 because it is a lock
33     ...
34 }
```

Figure 31.12: Adding Mutual Exclusion (Correctly)

31.5 Reader-Writer Locks

別の古典的な問題は、異なるデータ構造アクセスが異なる種類のロックを必要とする可能性があることを認める、より柔軟なロックプリミティブに対する要望から生じます。たとえば、挿入や簡単な検索など、多数の並行リスト操作を想像してみてください。挿入がリストの状態を変える（したがって伝統的なクリティカルセクションが意味をなさない）間に、ロックアップは単にデータ構造を読み込みます。挿入が行われていないことを保証できれば、多くのロックアップを並行して進めることができます。このタイプの操作をサポートするために開発する特殊タイプのロックは、リーダライタロック [CHP71] として知られています。このようなロックのコードは、図 31.13 で使用できます。

```

1  typedef struct _rwlock_t {
2      sem_t lock;          // binary semaphore (basic lock)
3      sem_t writelock;    // used to allow ONE writer or MANY readers
4      int readers;        // count of readers reading in critical section
5  } rwlock_t;
6
7  void rwlock_init(rwlock_t *rw) {
8      rw->readers = 0;
9      sem_init(&rw->lock, 0, 1);
10     sem_init(&rw->writelock, 0, 1);
11 }
12
13 void rwlock_acquire_readlock(rwlock_t *rw) {
14     sem_wait(&rw->lock);
15     rw->readers++;
16     if (rw->readers == 1)
17         sem_wait(&rw->writelock); // first reader acquires writelock
18     sem_post(&rw->lock);
19 }
20
21 void rwlock_release_readlock(rwlock_t *rw) {
22     sem_wait(&rw->lock);
23     rw->readers--;
24     if (rw->readers == 0)
25         sem_post(&rw->writelock); // last reader releases writelock
26     sem_post(&rw->lock);
27 }
28
29 void rwlock_acquire_writelock(rwlock_t *rw) {
30     sem_wait(&rw->writelock);
31 }
32
33 void rwlock_release_writelock(rwlock_t *rw) {
34     sem_post(&rw->writelock);
35 }

```

Figure 31.13: A Simple Reader-Writer Lock

コードはかなり簡単です。問題のデータ構造を更新したいスレッドがあれば、書き込みロックを獲得するための `rwlock_acquire_writelock()` とそれを解放するための `rwlock_release_writelock()` という新しい同期操作のペアを呼び出す必要があります。内部的には、これらは単に書き込みロックセマフォを使用して、1人のライターだけがロックを解除し、問題のデータ構造を更新するクリティカルセクションに入ることができます。

さらに興味深いのは、読み取りロックを取得して解放するためのルーチンのペアです。読み取りロックを獲得するとき、読者は最初にロックを獲得し、次に読者変数をインクリメントして、現在データ構造内にある読者の数を追跡します。`rwlock_acquire_readlock()` 内で実行される重要なステップは、最初の読者がロックを取得したときに発生します。その場合、読者は、書き込みロックセマフォに対して `sem_wait()` を呼び出し、`sem_post()` を呼び出してロックを解放することによって、書き込みロックを取得します。

したがって、読者が読み取りロックを取得すると、より多くの読者が読み取りロックを取得することも許可されます。ただし、書き込みロックを取得するスレッドは、すべての読者が終了するまで待機する必要があります。クリティカルセクションを終了する最後のものは “writelock” の `sem_post()` を呼び出すので、待機中のライターはロックを獲得できます。このアプローチは(必要に応じて)機能しますが、特に公平性に関しては、いくつかのネガティブな点があります。特に、読者がライターを食えさせるのは比較的簡単です。この問題に対するより高度な解決策が存在します。おそらくあなたはより良い実装を考えることができますか? ヒント: ライターが待っていると、より多くの読者がロックに入るのを防ぐために、あなたがする必要があることを考えてください。

最後に、リーダライタのロックを注意して使用する必要があることに注意してください。それらはしばしばオーバーヘッドが発生します(特により洗練された実装では)、単純で高速なロックプリミティブ[CB08]を使用するのと比べてパフォーマンスのスピードアップにつながりません。いずれにしても、セマフォを興味深く有用な方法で使用する方法をもう一度紹介します。

TIP: SIMPLE AND DUMB CAN BE BETTER (HILL'S LAW)

シンプルで愚かなアプローチが最高のものになるという考えを過小評価するべきではありません。ロックすると、実装が簡単で高速であるため、単純なスピinnロックが最適な場合があります。リーダライタのようなものはロックされていますが、複雑で複雑なものは遅いという意味です。したがって、まず、シンプルで愚かなアプローチをまず試みてください。シンプルさに訴えるこのアイデアは、多くの場所で見られます。初期の情報源は、Mark Hill の論文[H87]であり、CPU 用のキャッシュを設計する方法を研究しました。ヒル氏は、シンプルなダイレクトマップキャッシュは、ファンシーで先進的なデザインよりも優れていることを発見しました(キャッシングでは、デザインが単純なため、検索が高速になります)。ヒルが簡潔に彼の作品を要約したように、「巨大で愚かな方が良い」と言いました。

31.6 The Dining Philosophers

ダイクストラによって提起され、解決されたもっとも有名な並行性の問題の1つは、食堂の哲学者の問題[D71]として知られています。この問題は、それが楽しく、多分知的に面白いので有名です。しかし、その実用性は低いです。しかし、その名声はここに含まれています。実際には、ある面接でそれについて尋ねられるかもしれませんし、あなたがその質問を見落として仕事を得れなかつたら、あなたは OS の教授を本当に恨むでしょう。逆に、あなたが仕事を得たら、あなたの OS 教授に素敵なメモ、またはいくつかのストックオプションを送ってください。

問題の基本的な設定はこれです(図 31.14 を参照)。テーブルの周りに5人の“哲学者”が座っているとします。哲学者の各ペアの間には1つのフォーク(したがって合計5つ)があります。哲学者はそれぞれ考えている時間があり、フォークや食べる時間は必要ありません。食べるためには、哲学者は2本のフォークを必要とします。左側のものと右側のものの両方があります。これらのフォークの競合、それに続く同期の問題は、これを同時プログラミングで研究する問題となっています。

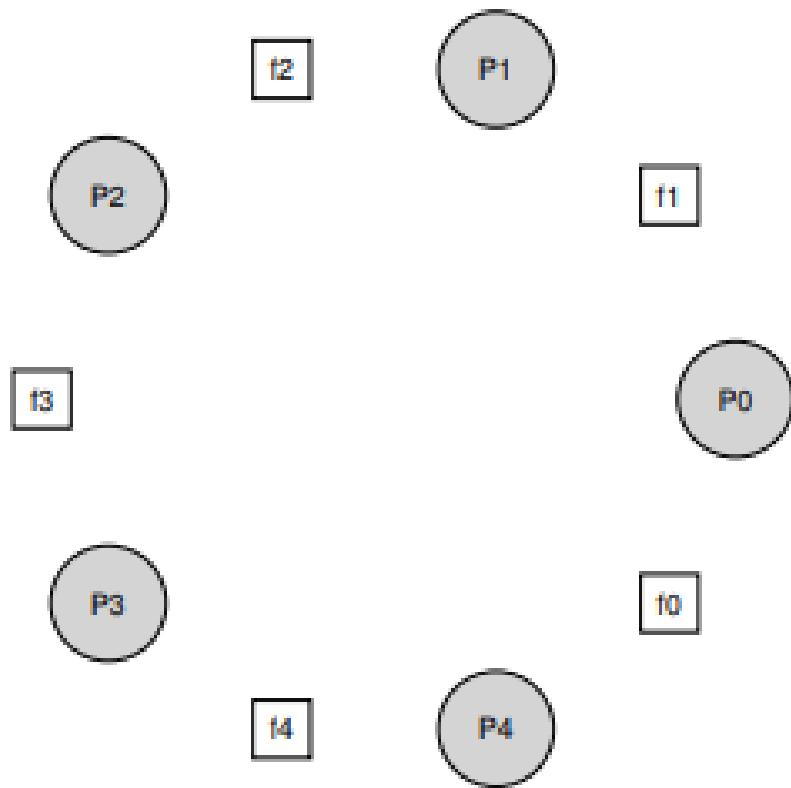


Figure 31.14: The Dining Philosophers

各哲学者の基本的なループは次のとおりです。

```
while (1) {
    think();
    getforks();
    eat();
    putforks();
}
```

キーとなる挑戦は、デッドロックがなく、哲学者が食えず、食べることもなく、並行性が高い（つまり、同時に多くの哲学者が同時に食べることができるよう）書き込みのルーチンである `getforks()` と `putforks()` をできるだけ実行できるようにすることです。

ダウニーのソリューション [D08] に続いて、いくつかのヘルパー関数を使用して、私たちを解決策に導きます。

```
int left(int p) { return p; }
int right(int p) { return (p + 1) % 5; }
```

哲学者 p が左のフォークを参照したいとき、彼らは単に `left(p)` を呼び出します。同様に、哲学者 p の右側のフォークは `right(p)` を呼び出すことによって参照されます。その中の剰余演算子は、最後の哲学者 ($p = 4$)

がフォークが 0 であるとき右手でつかもうとする処理をします。

また、この問題を解決するためにセマフォーが必要になります。それぞれのフォークに対して 1 つずつ、sem_t forks [5] が 5 つあるとします。

Broken Solution

我々はこの問題に対する最初の解決策を試みます。各セマフォー(フォーク配列内の)を 1 の値に初期化するものとします。各哲学者がそれ自身の数(p)を知っているとします。したがって、図 31.15 に示す getforks() と putforks() ルーチンを記述することができます。

```

1 void getforks() {
2     sem_wait(forks[left(p)]);
3     sem_wait(forks[right(p)]);
4 }
5
6 void putforks() {
7     sem_post(forks[left(p)]);
8     sem_post(forks[right(p)]);
9 }
```

Figure 31.15: The `getforks()` And `putforks()` Routines

この(壊れた)ソリューションの背後にある直感は次のとおりです。フォークを取得するには、まずそれぞれのロックを取得します。最初は左に、次に右にあるロックです。食べ終わると、私たちはそれを解放します。シンプルですよね？ 残念ながら、この場合、単純な手段は壊れています。発生した問題を理解できますか？ それについて考えてみましょう。

問題はデッドロックです。哲学者が自分の右にあるフォークをつかむ前に、各哲学者が左にあるフォークをつかんでしまったら、それぞれのフォークをつかんで別のものを永遠に待つことになります。具体的には、哲学者 0 はフォーク 0、哲学者 1 はフォーク 1、哲学者 2 はフォーク 2、哲学者 3 はフォーク 3 を、哲学者 4 はフォーク 4 をつかむ。すべてのフォークが獲得され、すべての哲学者が別の哲学者が所有するフォークを待っています。私たちはすぐにデッドロックを詳しく研究します。今のところ、これは実用的な解決策ではないと言っても過言ではありません。

A Solution: Breaking The Dependency

この問題を攻撃する最も簡単な方法は、哲学者の少なくとも 1 人がフォークを取得する方法を変更することです。確かに、これは Dijkstra 自身がどのように問題を解決したかです。具体的には、哲学者 4(一番高い番号の哲学者)がフォークを別の順序で取得すると仮定しましょう。これを行うコードは次のとおりです。

```

1 void getforks () {
2     if (p == 4) {
3         sem_wait (forks [right (p)] );
4         sem_wait (forks [left (p)] );
5     } else {
6         sem_wait (forks [left (p)] );
7         sem_wait (forks [right (p)] );
8     }
9 }
```

最後の哲学者は左手の前に右手をつかみようとしているので、各哲学者が1つのフォークをつかみ、別のフォークを待っている状況はありません。これで待ちのサイクルが解消されました。このソリューションの成果を考えて、それが機能することを自分自身で動かしてください。

このような他の「有名な」問題、例えばたばこ喫煙者の問題または睡眠中の理髪師の問題などがあります。それらのほとんどは、並行性について考えることの言い訳です。それらのいくつかは魅力的な名前を持っています。より多くのことを学ぶことに興味がある場合は、それらを見てください。あるいは同時に多くの練習を同時に行うことができます [D08]。

31.7 How To Implement Semaphores

最後に、低レベルの同期プリミティブ、ロック、および条件変数を使用して、…(ドラムロール) というセマフォの独自のバージョンを構築しましょう…Zemaphores。図 31.16 に示すように、この作業はかなり簡単です。

```

1  typedef struct __Zem_t {
2      int value;
3      pthread_cond_t cond;
4      pthread_mutex_t lock;
5  } Zem_t;
6
7  // only one thread can call this
8  void Zem_init(Zem_t *s, int value) {
9      s->value = value;
10     Cond_init(&s->cond);
11     Mutex_init(&s->lock);
12 }
13
14 void Zem_wait(Zem_t *s) {
15     Mutex_lock(&s->lock);
16     while (s->value <= 0)
17         Cond_wait(&s->cond, &s->lock);
18     s->value--;
19     Mutex_unlock(&s->lock);
20 }
21
22 void Zem_post(Zem_t *s) {
23     Mutex_lock(&s->lock);
24     s->value++;
25     Cond_signal(&s->cond);
26     Mutex_unlock(&s->lock);
27 }

```

Figure 31.16: Implementing Zemaphores With Locks And CVs

この図からわかるように、セマフォの値を追跡するために、ロックと条件変数を 1 つだけ使用し、状態変数を使用します。あなたが本当に理解するまで、あなた自身のためにコードを研究してください。ダイクストラによって定義された Zemaphore と純粋なセマフォの 1 つの微妙な違いは、セマフォの値が負の場合は待機スレッドの数を反映するという不变値を維持しないことです。実際には、値は決してゼロより低くなることはありません。この動作は実装が容易で、現在の Linux の実装と一致します。

TIP: BE CAREFUL WITH GENERALIZATION

したがって、一般化の抽象的な技法は、システム設計において非常に有用であり、そこでは、1 つの良いアイデアをわずかに広げてより大きなクラスの問題を解決することができます。ただし、一般化するときは注意してください。Lamson は私たちに”一般化しないでください。一般化は一般的に間違っている [L83]。

セマフォをロックと条件変数の一般化として見ることができます。しかし、そのような一般化が必要ですか？ また、セマフォの上に条件変数を実現するのが難しいことを考えると、おそらくこの一般化は一般的ではありません。

不思議なことに、セマフォから条件変数を構築することは、はるかにトリッキーな命題です。経験の豊富な並行プログラマの中には、Windows 環境でこれを実行しようとしたものがあり、さまざまなバグが発生しました [B04]。自分で試して、セマフォからビルド条件変数を表示するのがなぜ難しいかを理解できるかどうかを確認してください。

31.8 Summary

セマフォは、並行プログラムを作成するための強力かつ柔軟なプリミティブです。一部のプログラマーは、そのシンプルさと有用性に、ロックと条件変数を避け、独占的にそれらを使用しています。この章では、いくつかの古典的な問題と解決策を紹介しました。詳細を知りたい場合は、参照できる他の多くの資料があります。1つの偉大な（そして自由な参照）は、並行性とセマフォー [D08] によるプログラミングに関する Allen Downey の本です。この本は、一般的に特定の並行性と並行性の両方のセマフォの理解を向上させるために取り組むことができる多くのパズルを持っています。真の同時実行性の専門家になるには長年の努力が必要です。それが、このクラスで学んだことを超えて、マスターする鍵です。

参考文献

- [B04] “Implementing Condition Variables with Semaphores”
Andrew Birrell
December 2004
An interesting read on how difficult implementing CVs on top of semaphores really is, and the mistakes the author and co-workers made along the way. Particularly relevant because the group had done a ton of concurrent programming; Birrell, for example, is known for (among other things) writing various thread-programming guides.
- [CB08] “Real-world Concurrency”
Bryan Cantrill and Jeff Bonwick
ACM Queue. Volume 6, No. 5. September 2008
A nice article by some kernel hackers from a company formerly known as Sun on the real problems faced in concurrent code.
- [CHP71] “Concurrent Control with Readers and Writers”
P.J. Courtois, F. Heymans, D.L. Parnas
Communications of the ACM, 14:10, October 1971
The introduction of the reader-writer problem, and a simple solution. Later work introduced more complex solutions, skipped here because, well, they are pretty complex.
- [D59] “A Note on Two Problems in Connexion with Graphs”
E. W. Dijkstra
Numerische Mathematik 1, 269271, 1959
Available: <http://www-m3.ma.tum.de/twiki/pub/MN0506/WebHome/dijkstra.pdf>
Can you believe people worked on algorithms in 1959? We can't. Even before computers were any fun to use, these people had a sense that they would transform the world...
- [D68a] “Go-to Statement Considered Harmful”
E.W. Dijkstra
Communications of the ACM, volume 11(3): pages 147148, March 1968
Available: <http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF>
Sometimes thought as the beginning of the field of software engineering.
- [D68b] “The Structure of the THE Multiprogramming System”
E.W. Dijkstra
Communications of the ACM, volume 11(5), pages 341346, 1968

One of the earliest papers to point out that systems work in computer science is an engaging intellectual endeavor. Also argues strongly for modularity in the form of layered systems.

[D72] “Information Streams Sharing a Finite Buffer”

E.W. Dijkstra

Information Processing Letters 1: 179180, 1972

Available: <http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD329.PDF>

Did Dijkstra invent everything? No, but maybe close. He certainly was the first to clearly write down what the problems were in concurrent code. However, it is true that practitioners in operating system design knew of many of the problems described by Dijkstra, so perhaps giving him too much credit would be a misrepresentation of history.

[D08] “The Little Book of Semaphores”

A.B. Downey

Available: <http://greenteapress.com/sema/>

A nice (and free!) book about semaphores. Lots of fun problems to solve, if you like that sort of thing.

[D71] “Hierarchical ordering of sequential processes”

E.W. Dijkstra

Available: <http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF>

Presents numerous concurrency problems, including the Dining Philosophers. The wikipedia page about this problem is also quite informative.

[GR92] “Transaction Processing: Concepts and Techniques”

Jim Gray and Andreas Reuter

Morgan Kaufmann, September 1992

The exact quote that we find particularly humorous is found on page 485, at the top of Section 8.8: “The first multiprocessors, circa 1960, had test and set instructions . . . presumably the OS implementors worked out the appropriate algorithms, although Dijkstra is generally credited with inventing semaphores many years later.”

[H87] “Aspects of Cache Memory and Instruction Buffer Performance”

Mark D. Hill

Ph.D. Dissertation, U.C. Berkeley, 1987

Hill’s dissertation work, for those obsessed with caching in early systems. A great example of a quantitative dissertation.

[L83] “Hints for Computer Systems Design”

Butler Lampson

ACM Operating Systems Review, 15:5, October 1983

Lampson, a famous systems researcher, loved using hints in the design of computer systems. A hint is something that is often correct but can be wrong; in this use, a signal() is telling a waiting thread that it changed the condition that the waiter was waiting on, but not to trust that the condition will be in the desired state when the waiting thread wakes up. In this paper about hints for designing systems, one of Lampson’s general hints is that you should use hints. It is not as confusing as it sounds.

32 Common Concurrency Problems

研究者は長年にわたり並行性のバグを探すために多大な時間と労力を費やしてきました。初期の作業の多くは、デッドロックに焦点を絞っていました。これは過去の章で触れたことのあるトピックですが、深く [C + 71] に浸透します。最近の研究は、他のタイプの共通の並行性バグ(非デッドロックバグ)の研究に重点を置いています。この章では、実際のコードベースに見られるいくつかの並行処理の問題の例を簡単に見て、どのような問題を把握するかをより深く理解します。したがって、この章の中心的な問題は次のとおりです。

CRUX: HOW TO HANDLE COMMON CONCURRENCY BUGS

並行性のバグは、さまざまな共通パターンになる傾向があります。より堅牢で正確な並行コードを作成するための第一歩です。

32.1 What Types Of Bugs Exist?

最初の、そして最も明白な問題は、複雑な並行プログラムでどのような並行性バグが現れるかということです。この質問は一般的には答えにくいですが、幸いなことに他の人たちが私たちの仕事をしてくれました。具体的には、我々は Lu らの研究に頼っています。[L + 08] は、実用上どのような種類のバグが発生しているのかを把握するために、多数の一般的な並列アプリケーションを詳細に分析しています。

この研究では、MySQL(人気の高いデータベース管理システム)、Apache(よく知られている Web サーバー)、Mozilla(有名な Web ブラウザ)、OpenOffice(MS Office スイートの無料版で一部の人々が実際に使用しています)の 4 つの主要で重要なオープンソースアプリケーションに焦点を当てています。この調査では、開発者の作業を定量的なバグ分析に変えながら、これらのコードベースで検出され、修正された並行性のバグを調べました。これらの結果を理解することで、成熟したコードベースで実際に発生する問題の種類を理解するのに役立ちます。

図 32.1 に、Lu 氏と同僚が研究したバグの概要を示します。図から、合計 105 個のバグがあり、そのほとんどはデッドロックではなかったことが分かります(74)。残りの 31 個はデッドロックバグでした。さらに、各アプリケーションから調査されたバグの数を確認できます。OpenOffice には 8 つの同時実行性のバグしかありませんでしたが、Mozilla は約 60 個のバグがありました。

Application	What it does	Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Web Browser	41	16
OpenOffice	Office Suite	6	2
Total		74	31

Figure 32.1: Bugs In Modern Applications

これらの異なるクラスのバグ(非デッドロック、デッドロック)にもう少し深入り込むようになりました。非デッドロックバグの最初のクラスでは、この調査の例を使って議論を進めています。デッドロックバグの 2 番目のクラスについては、デッドロックの防止、回避、または処理のいずれかで行われた長い作業について説明します。

32.2 Non-Deadlock Bugs

Lu の研究によると、デッドロックのないバグが同時性バグの大部分を占めています。しかし、どのタイプのバグですか？ 彼らはどのように起きますか？ どうすれば修正できますか？ 我々は今から、Lu らによって発見された 2 種類の非デッドロックバグについて議論します。それは原子性違反バグと順序違反バグです。

Atomicity-Violation Bugs

遭遇する第 1 のタイプの問題は、原子性違反と呼ばれます。ここでは、MySQL にある簡単な例を示します。説明を読む前に、バグが何であるかを調べてみてください。

```

1  Thread 1::  

2  if (thd->proc_info) {  

3      ...  

4      fputs(thd->proc_info, ...);  

5      ...  

6  }  

7  

8  Thread 2::  

9  thd->proc_info = NULL;

```

この例では、2 つの異なるスレッドが構造体 thc の proc_info フィールドにアクセスします。最初のスレッドは、値が NULL でないかどうかをチェックし、その値を出力します。2 番目のスレッドは NULL に設定します。明らかに、最初のスレッドがチェックを実行したが fputs の呼び出しの前に中断された場合、2 番目のスレッドが中間で実行され、ポインタが NULL に設定される可能性があります。最初のスレッドが再開すると、NULL ポインタが fputs によって参照解除されるため、クラッシュします。

Lu 他によれば、「複数のメモリアクセスの間の望んだ直列化可能性が侵害されている（すなわち、コード領域はアトミックであることが意図されているが、実行中にアトミック性は強制されない）」という原則違反のより正式な定義です。上記の例では、コードは、proc_info の NULL でないことのチェックと fputs() 呼び出しでの proc_info の使用について原子性の仮定 (Lu の言葉で) を持っています。前提が正しくない場合、コードは必要に応じて機能しません。

このタイプの問題に対する修正を見つけることは、(必ずしもそうではないが) 簡単です。上記のコードを修正する方法を考えてみませんか？ この解決策では、共有変数参照をロックするだけで、いずれかのスレッドが proc_info フィールドにアクセスするときにロックが保持されるようになります (proc_info_lock)。もちろん、構造体にアクセスする他のコードでも、このロックを取得してからロックを取得する必要があります。

```

1  pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;
2
3  Thread 1::
4  pthread_mutex_lock(&proc_info_lock);
5  if (thd->proc_info) {
6      ...
7      fputs(thd->proc_info, ...);
8      ...
9  }
10 pthread_mutex_unlock(&proc_info_lock);
11
12 Thread 2::
13 pthread_mutex_lock(&proc_info_lock);
14 thd->proc_info = NULL;
15 pthread_mutex_unlock(&proc_info_lock);

```

Order-Violation Bugs

Lu らによって発見された別の一般的なタイプの非デッドロックバグは順序違反 (order violation) として知られています。ここに別の簡単な例があります。もう一度、以下のコードにバグがある理由を理解できるかどうかを確認してください。

```

1  Thread 1::
2  void init() {
3      ...
4      mThread = PR_CreateThread(mMain, ...);
5      ...
6  }
7
8  Thread 2::
9  void mMain(...) {
10     ...
11     mState = mThread->State;
12     ...
13 }

```

図からわかるとおり、スレッド 2 のコードは、変数 mThread がすでに初期化されている (NULL ではない) と仮定しているようです。ただし、スレッド 2 が一度作成されるとすぐに、スレッド 2 の mMain() 内で mThread の値がアクセスされたときに値が設定されず、NULL ポインタの逆参照でクラッシュする可能性があります。注意としては mThread の値は最初は NULL であると仮定しています。もしそうでなければ、任意のメモリ位置がスレッド 2 の逆参照によってアクセスされるので、よく分からないことが起こってしまうかもしれません。

より正式な定義の順序違反 (order violation) は、「2つの (グループの) メモリアクセスの間の望んだ順序は反転します」です。(すなわち、A は常に B の前に実行されなければならないが、実行中はその順序は強制されない)[L + 08]。

この種のバグを修正するには、一般に順序付けが必要です。前に詳細に説明したように、条件変数を使用すると、このスタイルの同期を最新のコードベースに追加するのは簡単で堅牢な方法です。上記の例では、次の

ようにコードを書き直すことができます：

```

1  pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
2  pthread_cond_t  mtCond = PTHREAD_COND_INITIALIZER;
3  int mtInit          = 0;
4
5  Thread 1::
6  void init() {
7      ...
8      mThread = PR_CreateThread(mMain, ...);
9
10     // signal that the thread has been created...
11     pthread_mutex_lock(&mtLock);
12     mtInit = 1;
13     pthread_cond_signal(&mtCond);
14     pthread_mutex_unlock(&mtLock);
15     ...
16 }
17
18 Thread 2::
19 void mMain(...) {
20     ...
21     // wait for the thread to be initialized...
22     pthread_mutex_lock(&mtLock);
23     while (mtInit == 0)
24         pthread_cond_wait(&mtCond, &mtLock);
25     pthread_mutex_unlock(&mtLock);
26
27     mState = mThread->State;
28     ...
29 }
```

この固定コードシーケンスでは、ロック (mtLock) と対応する条件変数 (mtCond) と状態変数 (mtInit) を追加しました。初期化コードが実行されると、mtInit の状態が 1 に設定され、完了したことを通知します。この時点より前にスレッド 2 が実行された場合、スレッド 2 はこの信号および対応する状態の変更を待機します。それが後で実行されると、状態をチェックし、初期化が既に行われている (すなわち、mtInit が 1 にセットされている) ことを確認し、したがって適切なまま続行します。状態変数自体として mThread を使用する可能性がありますが、ここでは簡略化のためにそうしないことに注意してください。スレッド間で問題を発注する際には、条件変数 (またはセマフォ) が助ける可能性があります。

Non-Deadlock Bugs: Summary

Lu らが研究した非デッドロックバグの大部分 (97 %) 原子性または順序違反のいずれかです。したがって、これらのタイプのバグパターンについて注意深く考えることによって、プログラマーはそれらを避けるより良い仕事をする可能性があります。さらに、より自動化されたコードチェックツールが開発されるにつれて、デプロイメントで見つかった非デッドロックバグの大部分を構成するため、これらの 2 種類のバグに重点を置くべきです。

残念ながら、すべてのバグが上記の例と同じように簡単に解決できるわけではありません。あるものは、プログラムが何をしているのか、コードやデータ構造の再編成をより多く理解する必要があります。詳細は、Lu

らの優れた（そして判読可能な）論文を読んでください。

32.3 Deadlock Bugs

上記で説明した並行性のバグ以外にも、複雑なロックプロトコルを持つ多くの同時システムで発生する古典的な問題は、デッドロックと呼ばれています。例えば、スレッド（スレッド 1）がロック（L1）を保持し、ロック（L2）を保持しているスレッドを待っています。残念ながら、ロック L2 を保持するスレッド（スレッド 2）は、L1 が解放されるのを待っています。このような潜在的なデッドロックを示すコードスニペットは次のとおりです。

```
Thread 1:           Thread 2:  
pthread_mutex_lock(L1); pthread_mutex_lock(L2);  
pthread_mutex_lock(L2); pthread_mutex_lock(L1);
```

このコードが実行されると、デッドロックが必ずしも発生しないことに注意してください。一方で起こる可能性があるとしたら、たとえば、スレッド 1 が L1 をロックしてからスレッド 2 にコンテキストスイッチが発生した場合、スレッド 2 は L2 を取得して L1 を取得しようとしています。したがって、各スレッドはもう一方を待っており、どちらも実行できないため、デッドロックが発生します。図 32.2 を参照してください。グラフ内のサイクルの存在は、デッドロックを示す。

図は問題をはっきりさせるはずです。デッドロックを何らかの方法で処理するために、プログラマーはどのようにコードを書くべきですか？

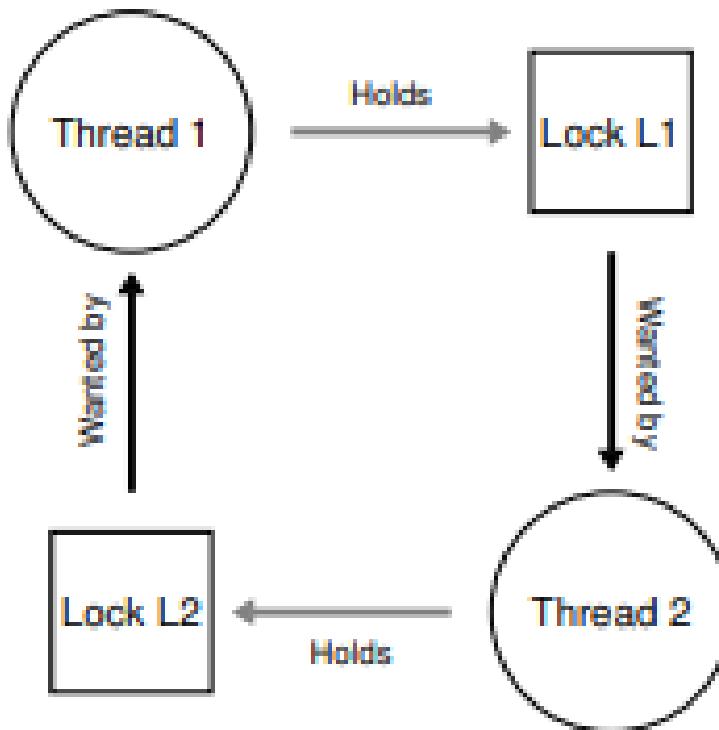


Figure 32.2: The Deadlock Dependency Graph

どのようにしてデッドロックを防止、回避、または少なくとも検出して回復するためのシステムを構築する必要がありますか？ これは今日のシステムで本当の問題ですか？

Why Do Deadlocks Occur?

あなたが思っているように、上記のような単純なデッドロックは容易に避けることができます。たとえば、スレッド 1 と 2 の両方が同じ順序でロックを取得することを確実にした場合、デッドロックは決して発生しません。なぜデッドロックが起こるのですか？

1 つの理由は、大きなコードベースでは、複雑な依存関係がコンポーネント間で発生するということです。たとえば、オペレーティングシステムを起動します。仮想メモリシステムは、ディスクからブロックをページするためにファイルシステムにアクセスする必要があります。ファイルシステムはその後、ブロックを読み込んで仮想メモリシステムに接続するためにメモリのページを要求することがあります。したがって、コード内で自然に発生する可能性のある循環依存関係の場合、デッドロックを回避するために、大規模システムでのロック戦略の設計を慎重に行う必要があります。

別の理由はカプセル化の性質によるものです。ソフトウェア開発者として、実装の詳細を隠し、ソフトウェアをモジュール化しやすくするように教えています。残念なことに、このようなモジュール化はロックとうまく合致しません。Jula らは、[J + 08] を指摘すると、いくつかの一見無害なインターフェースは、ほとんどあなたのデッドロックに招待します。たとえば、Java Vector クラスとメソッド AddAll() を使用します。このルーチンは次のように呼び出されます。

```
Vector v1, v2;
v1.AddAll(v2);
```

内部的には、メソッドがマルチスレッドセーフである必要があるため、(v1) に追加されるベクトルとパラメータ (v2) の両方のロックが取得される必要があります。ルーチンは、v2 の内容を v1 に追加するために、任意の順序 (例えば v1, v2) で前記ロックを獲得します。しかし、もし、他のスレッドが v2.AddAll(v1) をほぼ同時に呼び出すと、呼び出し元アプリケーションからは隠されたデッドロックの可能性があります。

Conditions for Deadlock

デッドロックが発生するには 4 つの条件が必要です [C + 71]。 - 相互排除：スレッドは、必要なリソースの排他制御を要求します (スレッドはロックを取得します)。

- ホールド・アンド・ウェイト：スレッドは、追加のリソース (例えば、取得したいロック) を待つ間に、スレッドに割り当てられたリソース (例えば、既に獲得したロック) を保持します。
- プリエンプションなし：リソース (ロックなど) は、リソースを保持しているスレッドから強制的に削除することはできません。
- 循環待ち：各スレッドが、チェーン内の次のスレッドによって要求されている 1 つ以上のリソース (例えば、ロック) を保持するように、スレッドの循環チェーンが存在する。

これらの 4 つの条件のいずれかが満たされない場合、デッドロックは発生しません。そこで、まず、デッドロックを防止するための手法を探る。これから説明するそれぞれの戦略は、上記の条件の 1 つが発生するのを防ぎ、デッドロック問題を処理する 1 つのアプローチです。

Prevention

Circular Wait

おそらく最も実用的な予防技術(そして確かに頻繁に採用されているもの)は、あなたが循環待ちを誘発しないようにあなたのロッキングコードを書くことでしょう。これを実行する最も簡単な方法は、ロック取得のトータルオーダーを行うことです。たとえば、システムに2つのロック(L1とL2)しかない場合、L2の前に常にL1を取得することでデッドロックを防ぐことができます。このような厳密な順序付けによって、循環待ちが発生しないことが保証されます。したがって、デッドロックは発生しません。

もちろん、より複雑なシステムでは、2つ以上のロックが存在するため、完全なロック順序を達成するのが難しい場合があります(そして、おそらく完全なロック順序付けは不要です)。したがって、部分順序付けは、デッドロックを回避するためにロック獲得を構造化するための有用な方法となり得ます。部分ロック順序の優れた実際の例は、Linux [T + 94] のメモリマッピングコードで見ることができます。ソースコードの先頭にあるコメントは、「i_mutex before i_mmap_mutex」などの単純なロック取得命令と、「i_mmap_mutex before private_lock before swap_lock before mapping->tree.lock」のような複雑な命令を含む10種類のロック取得オーダーを示しています。

あなたが想像することができるよう、全部と部分的な順序付けはロック戦略の慎重な設計を必要とし、細心の注意を払って構築しなければなりません。さらに、順序付けは単に規約に過ぎず、誤ったプログラマーは簡単にロックプロトコルを無視し、デッドロックを引き起こす可能性があります。最後に、ロックの順序付けには、コードベースの深い理解と、さまざまなルーチンの呼び出し方法が必要です。たった1つの間違いが「D」という言葉になる可能性があります。

TIP: ENFORCE LOCK ORDERING BY LOCK ADDRESS 場合によっては、関数が2つ(またはそれ以上)のロックを取得する必要があります。したがって、我々は注意または、デッドロックが発生する可能性があることを知っています。do_something(mutex t * m1, mutex t * m2)と呼ばれる関数を想像してみてください。もし、常に m2 より前に m1(または常に m1 より前に m2) を取得してしまうとデッドロックを発生する可能性があります。なぜなら、一つのスレッドが do_something(L1, L2) を呼び出すことができるため、別のスレッドが do_something(L2, L1) を呼び出すことができるからです。

この特別な問題を回避するために、巧妙なプログラマはロックの取得を順序付けする方法として各ロックのアドレスを使用することができます。high-to-low または low-to-high のどちらかのアドレス順でロックを取得することで、どの順序でも関係なく同じ順序(決まった順序)でロックを取得することが保証されます。このような例です：

```
if (m1 > m2) { // grab locks in high-to-low address order
    pthread_mutex_lock(m1);
    pthread_mutex_lock(m2);
} else {
    pthread_mutex_lock(m2);
    pthread_mutex_lock(m1);
}

// Code assumes that m1 != m2 (it is not the same lock)
```

この単純な技術を使用することにより、プログラマは、マルチロック取得のシンプルで効率的なデッドロックフリー実装を保証することができます。

Hold-and-wait

デッドロックのホールド・アンド・ウェイトの要件は、すべてのロックを一度に取得することで回避できます。実際には、これは以下のように達成します。

```

1  pthread_mutex_lock (prevention); // begin lock acquisition
2  pthread_mutex_lock (L1);
3  pthread_mutex_lock (L2);
4  ...
5  pthread_mutex_unlock (prevention); // end

```

最初にロック防止を取得することにより、このコードは、ロック取得の途中でスレッド交換が行われないこと、そのようなデッドロックを再び回避できることを保証します。もちろん、スレッドがロックを取得するたびに、グローバル防御ロックを取得する必要があります。たとえば、別のスレッドがロック L1 と L2 を別の順序で取得しようとしていた場合は、その間にロック防止を保持しているので OK です。

解決策にはいくつかの理由で問題があることに注意してください。以前のように、カプセル化は私たちに対して働いています。ルーチンを呼び出すときに、このアプローチでは、ロックを保持する必要があることを正確に把握し、事前に取得する必要があります。この技術はまた、すべてのロックを真に必要とされるのではなく、早めに(一度に)取得する必要があるため、並行性を低下させる可能性があります。

No Preemption

ロック解除が呼び出されるまで、一般にロックを保持しているとみなすので、複数のロックを取得すると、ロックを待っているときには別のロックを持っているので、問題が発生することがあります。多くのスレッドライブラリは、このような状況を回避するために、より柔軟なインターフェースを提供します。具体的には、ルーチン `pthread_mutex_trylock()` はロックを取得して(利用可能な場合)、成功を返します。ロックが保持されていることを示すエラーコードを返します。後者の場合、そのロックを取得したい場合は、後で再試行できます。このようなインターフェースは、次のように使用して、デッドロックのない、順序付けが可能なロック取得プロトコルを構築することができます。

```

1  top:
2      pthread_mutex_lock (L1);
3      if (pthread_mutex_trylock (L2) != 0) {
4          pthread_mutex_unlock (L1);
5          goto top;
6      }

```

別のスレッドは同じプロトコルに従うことができますが、他の順序(L2からL1)でロックを取得し、プログラムはデッドロックフリーであることに注意してください。しかし、新たな問題が生まれます。それは。ライブロックです。2つのスレッドがこのシーケンスを繰り返し試行し、両方のロックを繰り返し取得できない可能性があります(おそらくありません)この場合、両方のシステムはこのコードシーケンスを何度も繰り返し実行しています(したがって、デッドロックではありません)が進行中ではありません。したがって、ライブロックという名前があります。ライブロックの問題に対する解決策もあります。たとえば、ループバックする前にランダムな遅延を追加し、全体をもう一度試すことで、競合するスレッド間の干渉が繰り返される可能性を減らすことができます。

この解決策についての最後の1つのポイントは、それは `trylock` アプローチを使用するという困難な部分を

取り巻いています。再度存在する可能性のある第1の問題は、カプセル化です。これらのロックのうちの1つが呼び出されているルーチンに埋め込まれていると、先頭に戻ってジャンプするというより複雑な実装になります。コードが途中でいくつかのリソース (L1以外) を取得していた場合は、それらも注意深く解放する必要があります。たとえば、L1を取得した後でコードにメモリが割り当てられていた場合、L2を取得できなかつた場合にそのメモリを解放してから、先頭に戻ってシーケンス全体をやり直す必要があります。しかしながら、限定された状況 (例えば、前述のJavaベクトル法) では、このタイプのアプローチはうまくいく可能性があります。

Mutual Exclusion

最終的な防止手法は、相互排除の必要性を全く避けることです。一般的に、実行したいコードには本当に重要な部分があるので、これは困難です。だから私たちは何をすることができますか？ Herlihyは、ロックなしで様々なデータ構造を設計できるという考え方を持っていました [H91, H93]。これらのロックフリー（および関連する待機フリー）アプローチのアイデアは簡単です。強力なハードウェア命令を使用すると、明示的なロックを必要としない方法でデータ構造を構築することができます。

簡単な例として、compare-and-swap命令があるとしましょう。これは、ハードウェアが提供する以下のようなアトミック命令です。

```

1 int CompareAndSwap(int *address, int expected, int new) {
2     if (*address == expected) {
3         *address = new;
4         return 1; // success
5     }
6     return 0; // failure
7 }
```

ある量だけ値を原子的に増やしたかったとします。私たちは以下のようにそれを行うことができました。

```

1 void AtomicIncrement(int *value, int amount) {
2     do {
3         int old = *value;
4     } while (CompareAndSwap(value, old, old + amount) == 0);
5 }
```

ある特定の値をアトミックにインクリメントしたいとしたら、その値を新しい量に更新しようと繰り返し試み、比較とスワップを使用してアプローチを構築したとします。この方法では、ロックは取得されず、デッドロックは発生しません（ライブロックはまだ可能ですが）。もう少し複雑な例を考えてみましょう。それはリストの挿入です。リストの先頭に挿入するコードを次に示します。

```

1 void insert(int value) {
2     node_t *n = malloc(sizeof(node_t));
3     assert(n != NULL);
4     n->value = value;
5     n->next = head;
6     head = n;
7 }
```

このコードは簡単な挿入を行いますが、“同時に”複数のスレッドから呼び出された場合、競合状態になります

(理由を調べることができるかどうかを確認してください)。もちろん、このコードをロックの獲得と解放で囲むことで、これを解決できます。

```

1 void insert(int value) {
2     node_t *n = malloc(sizeof(node_t));
3     assert(n != NULL);
4     n->value = value;
5     pthread_mutex_lock(listlock); // begin critical section
6     n->next = head;
7     head = n;
8     pthread_mutex_unlock(listlock); // end critical section
9 }
```

このソリューションでは、従来の方法でロックを使用しています。代わりに、単純に compare-and-swap 命令を使用してこの挿入をロックフリーの方法で実行しようとします。可能なアプローチは次のとおりです。

```

1 void insert(int value) {
2     node_t *n = malloc(sizeof(node_t));
3     assert(n != NULL);
4     n->value = value;
5     do {
6         n->next = head;
7     } while (!CompareAndSwap(&head, n->next, n) == 0);
8 }
```

このコードは、現在のヘッドを指し示す次のポインタを更新し、新しく作成されたノードをリストの新しいヘッドとしてスワップしようとします。しかし、一方で他のスレッドが新しいヘッドでスワップを成功させ、このスレッドを新しいヘッドで再試行すると、これは失敗します。

もちろん、有用なリストを作成するには単なるリストの挿入以上のものが必要であり、意外なことに、ロックフリーの方法で挿入、削除、および検索を実行できるリストを構築することは簡単ではありません。ロックフリーと待ち時間のない同期に関する豊富な文献を読んで詳細 [H01, H91, H93] を参照してください。

Deadlock Avoidance via Scheduling

デッドロック防止の代わりに、デッドロック回避が望ましい場合もあります。回避は、様々なスレッドが実行中にどのロックを獲得するかについてのグローバルなロックに関する知識を必要とし、その後デッドロックが発生しないことを保証するように前記スレッドをスケジュールします。

たとえば、2つのプロセッサと4つのスレッドをスケジューリングする必要があるとします。さらに、スレッド1(T1)がL1とL2をロックしていることを知っているとします(T2は実行中のある時点での順序でロックします)、L1とL2も同様に把持し、T3はL2だけを捕捉し、T4はロックをまったく持ちません。これらのスレッドのロック獲得要求を表形式で示すことができます。

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

したがって、スマートスケジューラは、T1 と T2 が同時に実行されない限り、デッドロックは発生しない可能性があると計算できます。そのようなスケジュールの 1 つがあります



(T3 と T1) または (T3 と T2) が重なってもかまいません。T3 は L2 をロックしますが、1 つのロックしか持たないため、他のスレッドと並行してデッドロックを引き起こすことはありません。もう 1 つの例を見てみましょう。この場合、次の競合テーブルで示されるように、同じリソース (再び L1 および L2 のロック) に対してより多くの競合が発生します。

	T1	T2	T3	T4
L1	yes	yes	yes	no
L2	yes	yes	yes	no

特に、スレッド T1、T2、および T3 は、実行中のある時点では両方のロック L1 および L2 を取得する必要があります。デッドロックが発生しないことを保証するスケジュールがあります



ご覧のように、静的スケジューリングは、T1、T2、T3 がすべて同じプロセッサ上で実行されるという控えめなアプローチにつながります。したがって、ジョブを完了するための合計時間が大幅に長くなります。これらのタスクを同時に実行することは可能かもしれません、デッドロックの懼れが私たちを妨げ、コストとして支払うのはパフォーマンスです。

このようなアプローチの有名な例として、Dijkstra の Banker's Algorithm [D64] があり、多くの同様のアプローチが文献に記載されています。残念ながら、非常に限られた環境、例えば、実行する必要のある一連のタスクと必要なロックを完全に把握している組み込みシステムの場合にのみ役立ちます。さらに、このようなアプローチは、上記の 2 番目の例で見たように、並行性を制限する可能性があります。したがって、スケジューリングによるデッドロックの回避は、汎用的なソリューションではありません。

Detect and Recover

最終的な一般的な戦略の1つは、デッドロックが頻繁に発生するようにし、そのようなデッドロックが検出されたら何らかのアクションをとることです。たとえば、OS が1年に1回凍結した場合、OS を再起動して作業で喜んで(またはうんざりして)取得します。デッドロックがまれである場合、そのような非解決策は確かにかなり実用的です。

TIP: DON'T ALWAYS DO IT PERFECTLY (TOM WEST'S LAW)

古典的なコンピュータ業界の書籍「Soul of a New Machine [K81]」の主題で有名な Tom West は、「価値あることは何もかもうまくいくわけではない」という著名な技術的格言である。悪いことがまれにしか起こらない場合は、特に、悪いことが発生するコストが小さい場合には、そのことを防ぐために多大な努力を費やすべきではありません。一方、スペースシャトルを建設していて、間違ったことのコストがスペースシャトルが爆発している場合は、このアドバイスを無視してください。

多くのデータベースシステムは、デッドロックの検出と回復のテクニックを採用しています。デッドロック検出器は定期的に実行され、リソースグラフを作成し、サイクルごとにチェックします。サイクル(デッドロック)が発生した場合は、システムを再起動する必要があります。より複雑なデータ構造の修復が最初に必要とされる場合、プロセスを容易にするために人間が関与するかもしれません。データベースの並行性、デッドロック、および関連する問題の詳細は、他の場所で確認できます[B + 87, K87]。これらの作品を読んだり、データベースを使ってこの豊富で興味深いトピックについて学んでください。

32.4 Summary

この章では、並行プログラムで発生するバグの種類について検討しました。最初のタイプの非デッドロックバグは、驚くほど一般的ですが、そのほとんどは修正するのが簡単です。これらには、一緒に実行されるべき一連の命令ではなく、2つのスレッド間で必要な順序が強制されていない違反を命令する、原子性違反が含まれます。

デッドロックについても簡単に説明しました。デッドロックはなぜ発生するのか、それについて何ができるのですか。この問題は並行処理自体と同じくらい古いものであり、このトピックについて何百もの論文が書かれています。実際の最善の解決策は、慎重に、ロック獲得命令を開発し、最初にデッドロックが発生するのを防ぐことです。待ち受けのないデータ構造は、Linux を含む一般的なライブラリやクリティカルなシステムへの道を切り開いているため、待ち時間のないアプローチも有望です。しかし、一般性の欠如と新しい待ち時間のないデータ構造を開発する複雑さが、このアプローチの全体的な有用性を制限する可能性があります。おそらく、最良のソリューションは、新しい並行プログラミングモデルを開発することです。例えば、MapReduce(Google から)[GD02] のようなシステムでは、プログラマはロックなしで特定のタイプの並列計算を記述することができます。ロックはその性質上問題があります。おそらく、私たちが本当に必要でない限り、それらを使用しないようにするべきです。

参考文献

- [B+87] “Concurrency Control and Recovery in Database Systems”
Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman
Addison-Wesley, 1987
- The classic text on concurrency in database management systems. As you can tell, understanding concurrency, deadlock, and other topics in the world of databases is a world unto itself. Study it and find

out for yourself.

[C+71] “System Deadlocks”

E.G. Coffman, M.J. Elphick, A. Shoshani

ACM Computing Surveys, 3:2, June 1971

The classic paper outlining the conditions for deadlock and how you might go about dealing with it.

There are certainly some earlier papers on this topic; see the references within this paper for details.

[D64] “Een algorithme ter voorkoming van de dodelijke omarming”

Edsger Dijkstra

Circulated privately, around 1964

Available: <http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD108.PDF>

Indeed, not only did Dijkstra come up with a number of solutions to the deadlock problem, he was the first to note its existence, at least in written form. However, he called it the “deadly embrace”, which (thankfully) did not catch on.

[GD02] “MapReduce: Simplified Data Processing on Large Clusters”

Sanjay Ghemawhat and Jeff Dean

OSDI ’04, San Francisco, CA, October 2004

The MapReduce paper ushered in the era of large-scale data processing, and proposes a framework for performing such computations on clusters of generally unreliable machines.

[H01] “A Pragmatic Implementation of Non-blocking Linked-lists”

Tim Harris

International Conference on Distributed Computing (DISC), 2001

A relatively modern example of the difficulties of building something as simple as a concurrent linked list without locks.

[H91] “Wait-free Synchronization”

Maurice Herlihy

ACM TOPLAS, 13:1, January 1991

Herlihy’s work pioneers the ideas behind wait-free approaches to writing concurrent programs. These approaches tend to be complex and hard, often more difficult than using locks correctly, probably limiting their success in the real world.

[H93] “A Methodology for Implementing Highly Concurrent Data Objects”

Maurice Herlihy

ACM TOPLAS, 15:5, November 1993

A nice overview of lock-free and wait-free structures. Both approaches eschew locks, but wait-free approaches are harder to realize, as they try to ensure that any operation on a concurrent structure will terminate in a finite number of steps (e.g., no unbounded looping).

[J+08] “Deadlock Immunity: Enabling Systems To Defend Against Deadlocks”

Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, George Canea

OSDI ’08, San Diego, CA, December 2008

An excellent recent paper on deadlocks and how to avoid getting caught in the same ones over and over again in a particular system.

[K81] “Soul of a New Machine”

Tracy Kidder, 1980

A must-read for any systems builder or engineer, detailing the early days of how a team inside Data General (DG), led by Tom West, worked to produce a “new machine.” Kidder’s other books are also

excellent, including Mountains beyond Mountains. Or maybe you don't agree with us, comma?

[K87] "Deadlock Detection in Distributed Databases"

Edgar Knapp

ACM Computing Surveys, 19:4, December 1987

An excellent overview of deadlock detection in distributed database systems. Also points to a number of other related works, and thus is a good place to start your reading.

[L+08] "Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics"

Shan Lu, Soyeon Park, Eunsoo Seo, Yuanyuan Zhou

ASPLOS '08, March 2008, Seattle, Washington

The first in-depth study of concurrency bugs in real software, and the basis for this chapter. Look at Y.Y. Zhou's or Shan Lu's web pages for many more interesting papers on bugs.

[T+94] "Linux File Memory Map Code"

Linus Torvalds and many others

Available: <http://lxr.free-electrons.com/source/mm/filemap.c>

Thanks to Michael Waltrip (NYU) for pointing out this precious example. The real world, as you can see in this file, can be a bit more complex than the simple clarity found in textbooks...

33 Event-based Concurrency (Advanced)

これまで、並列アプリケーションを構築する唯一の方法がスレッドを使用する方法であるかのように、並行性について書いてきました。人生の多くの事のように、これは完全に真実ではありません。具体的には、GUI ベースのアプリケーション [O96] といいくつかのタイプのインターネットサーバー [PDZ99] の両方で、異なるスタイルの並行プログラミングがよく使用されます。イベントベースの並行処理と呼ばれるこのスタイルは、node.js [N13] などのサーバー側のフレームワークを含む現代的なシステムでは一般的になっていますが、その根本は以下で説明する C/UNIX システムにあります。

イベントベースの並行処理が解決する問題は 2 倍です。1 つ目は、マルチスレッドアプリケーションでの同時実行性を正しく管理することが難しいことです。私たちが議論したように、ロックの不足、デッドロック、および他の厄介な問題が発生する可能性があります。2 つ目は、マルチスレッドアプリケーションでは、開発者は特定の瞬間にスケジュールされているものをほとんど、またはまったく制御できないことです。むしろプログラマはスレッドを作成し、基盤となる OS が利用可能な CPU を介して合理的な方法でそれらをスケジュールすることを期待しています。すべての仕事量ですべてのケースでうまく動作する汎用スケジューラを構築することが難しい場合、OS は最適でない方法で作業をスケジュールすることができます。

THE CRUX: HOW TO BUILD CONCURRENT SERVERS WITHOUT THREADS

スレッドを使用せずに並行サーバーを構築することで、並行性の制御を維持するだけでなく、マルチスレッド・アプリケーションに悩まされている問題のいくつかを回避できますか？

33.1 The Basic Idea: An Event Loop

前述のように、基本的なアプローチは、イベントベースの並行処理と呼ばれます。このアプローチは非常に簡単です。何か(つまり、「イベント」)が発生するのを待つだけです。その場合、それはどのタイプのイベントであるかをチェックし、必要な少量の作業(I/O 要求の発行、または将来の処理のための他のイベントのスケジューリングなど)を行います。それで終了です！

詳細に入る前に、最初に正規のイベントベースのサーバーの外観を調べてみましょう。そのようなアプリケーションは、イベントループとして知られる簡単な構成に基づいています。イベントループの擬似コードは次のようになります。

```
while (1) {
    events = getEvents();
    for (e in events)
        processEvent(e);
}
```

それは本当に簡単です。メインループは(上記のコードで `getEvents()` を呼び出すことによって)何かを待ってから、返されたイベントごとに一度に 1 つずつ処理します。各イベントを処理するコードをイベントハンドラと呼びます。重要なことに、ハンドラがイベントを処理するとき、それはシステム内で行われる唯一のアクティビティです。したがって、次に処理するイベントを決定することは、スケジューリングと同じです。スケジューリングに対するこの明示的な制御は、イベントベースのアプローチの基本的な利点の 1 つです。

しかし、この議論から、より大きな疑問が浮かび上がります。イベントベースのサーバーは、特にネットワークやディスク I/O に関してどのようなイベントが起こっているかを正確に判断しますか？具体的には、イベント・サーバーは、メッセージが到着したかどうかをどのように伝えることができますか？

33.2 An Important API: `select()` (or `poll()`)

その基本的なイベントループを念頭に置いて、次にイベントを受け取る方法の問題に取り組まなければなりません。ほとんどのシステムでは、`select()` または `poll()` システムコールのいずれかを使用して基本 API を使用できます。これらのインターフェイスがプログラムに行うことができるることは簡単です。着信する I/O があるかどうかを確認します。たとえば、ネットワークアプリケーション (Web サーバーなど) が、サービスを提供するために、ネットワークパケットが到着したかどうかを確認したいと考えているとします。

これらのシステムコールは、あなたがそうすることを可能にします。たとえば `select()` を実行します。マニュアルページ (Mac 版) では、このように API について説明しています。

```
int select(int nfds,
fd_set *restrict readfds,
fd_set *restrict writefds,
fd_set *restrict errorfds,
struct timeval *restrict timeout);
```

マニュアルページの実際の記述：`select()` は、`readfds`、`writefds`、および `errorfds` に渡されたアドレスを持つ I/O ディスクリプタセットを調べて、ディスクリプタの一部が読み込み準備ができているか、書き込み準備ができているか、状態はそれぞれ保留中かです。最初の `nfds` ディスクリプタは、各組においてチェックされ、すなわち、ディスクリプタ集合内の 0 から `nfds`-1 までのディスクリプタが検査されます。戻り時に、`select()` は、指定されたディスクリプタセットを、要求された操作の準備ができているディスクリプタで構成されるサブセットに置き換えます。`select()` は、すべてのセットの準備完了ディスクリプタの合計数を返します。

ASIDE: BLOCKING VS. NON-BLOCKING INTERFACES

ブロッキング (または同期) インターフェイスは、呼び出し元に戻る前にすべての作業を行います。

ノンブロッキング (または非同期) インターフェイスはいくつかの作業を開始しますが、直ちに戻るため、実行する必要がある作業はすべてバックグラウンドで完了します。

コールをブロックする際の通常の原因は、ある種の I/O です。たとえば、完了するためにコールをディスクから読み取らなければならない場合、コールはブロックされ、返されるディスクに送信された I/O 要求を待機します。

ノンブロッキングインターフェースは、どのようなスタイルのプログラミング (スレッドなど) でも使用できますが、ブロックがすべての進捗を停止させるコールとして、イベントベースのアプローチでは不可欠です。

`select()` に関する 2 つの点があります。まず、ディスクリプタの読み込みと書き込みの可否をチェックできることに注意してください。前者は、新しいパケットが到着し、処理が必要であるとサーバに判断させるが、後者は、応答が OK であるとき (すなわち、送信キューが満杯でないとき) にサービスに知らせます。

次に、タイムアウト引数です。ここで的一般的な使用法の 1 つは、タイムアウトを `NULL` に設定することです。これにより、`select()` が何らかの記述子が準備できるまで無期限にブロックされます。ただし、より堅牢なサーバーは通常、何らかの種類のタイムアウトを指定します。1 つの一般的な手法は、タイムアウトをゼロに設定し、`select()` への呼び出しを使用してすぐに戻ることです。

`poll()` システムコールはかなり似ています。詳細については、マニュアルページ、または Stevens and Rago [SR05] を参照してください。いずれにしても、これらの基本的なプリミティブは、ノンブロッキングイベントループを構築する方法を提供します。単に、着信パケットをチェックし、メッセージを含むソケットか

ら読み込み、必要に応じて応答します。

33.3 Using `select()`

これをより具体的にするために、`select()` を使ってどのネットワークディスクリプタにメッセージが入ってくるかを調べる方法を調べてみましょう。図 33.1 に簡単な例を示します。

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/time.h>
4  #include <sys/types.h>
5  #include <unistd.h>
6
7  int main(void) {
8      // open and set up a bunch of sockets (not shown)
9      // main loop
10     while (1) {
11         // initialize the fd_set to all zero
12         fd_set readFDs;
13         FD_ZERO(&readFDs);
14
15         // now set the bits for the descriptors
16         // this server is interested in
17         // (for simplicity, all of them from min to max)
18         int fd;
19         for (fd = minFD; fd < maxFD; fd++)
20             FD_SET(fd, &readFDs);
21
22         // do the select
23         int rc = select(maxFD+1, &readFDs, NULL, NULL, NULL);
24
25         // check which actually have data using FD_ISSET()
26         int fd;
27         for (fd = minFD; fd < maxFD; fd++)
28             if (FD_ISSET(fd, &readFDs))
29                 processFD(fd);
30     }
31 }
```

Figure 33.1: Simple Code Using `select()`

このコードは、実際にはかなり理解しやすいです。初期化の後、サーバーは無限ループに入ります。ループ内では、まず `FD_ZERO()` マクロを使用してファイル記述子のセットをクリアした後、`FD_SET()` を使用して、`minFD` から `maxFD` までのすべてのファイル記述子をセットに含めます。この記述子の集合は、例えば、サーバが注目しているすべてのネットワークソケットを表すことができます。最後に、サーバーは `select()` を呼び出して、どの接続がデータを利用できるか調べます。その後、ループ内で `FD_ISSET()` を使用することによって、イベント・サーバーは、どのデータ記述子がデータを準備していて、入ってくるデータを処理するかを見ることができます。

もちろん、実際のサーバーはこれよりも複雑であり、メッセージの送信、ディスク I/O の発行、およびその他の多くの詳細でロジックを使用する必要があります。詳細については、API 情報については Stevens and Rago [SR05]、または Pai et. al または Welsh et al. イベントベースのサーバー [PDZ99, WCB01] の一般

的な流れの概要については、これらをご覧ください。

33.4 Why Simpler? No Locks Needed

単一の CPU とイベントベースのアプリケーションでは、並行プログラムで見つかった問題はもう存在しません。具体的には、一度に 1 つのイベントしか処理されないため、ロックを取得または解放する必要はありません。イベントベースのサーバは、明らかにシングルスレッドであるため、別のスレッドによって中断することはできません。したがって、スレッド化されたプログラムに共通する並行性のバグは、基本的なイベントベースのアプローチでは現れません。

TIP: DON'T BLOCK IN EVENT-BASED SERVERS

イベントベースのサーバーを使用すると、タスクのスケジューリングをきめ細かく制御できます。ただし、このような制御を維持するために、呼び出し元の実行をブロックする呼び出しあるなど作成できません。このデザインのヒントに従わないと、イベントベースのサーバーがロックされ、クライアントが欲求不満になり、この本を読んだかどうかについて深刻な疑問が生じます。

33.5 A Problem: Blocking System Calls

これまでのところ、イベントベースのプログラミングは素晴らしい聞こえるでしょうか？ 単純なループをプログラムし、発生したイベントを処理します。ロックについて考える必要はありません！ しかし、問題があります。イベントがブロックする可能性のあるシステムコールを発行する必要がある場合はどうなりますか？

例えば、要求がクライアントからサーバに来て、ディスクからファイルを読み込み、その内容を要求元のクライアントに返す（単純な HTTP 要求と同じように）と想像してください。そのような要求を処理するために、最終的にファイルを開くために `open()` システムコールを発行し、続いてファイルを読むために一連の `read()` を呼び出す必要があります。ファイルがメモリに読み込まれると、サーバーはおそらく結果をクライアントに送信し始めます。

`open()` と `read()` の両方の呼び出しでストレージシステムに I/O 要求が発行されます（必要なメタデータやデータがすでにメモリにない場合）。スレッドベースのサーバーでは、これは問題ではありません。I/O 要求を発行しているスレッドが中断している間（I/O が完了するのを待っている間）、他のスレッドが実行できるため、サーバーは処理を進めることができます。実際、このような I/O と他の計算の自然なオーバーラップは、スレッドベースのプログラミングを非常に自然かつ簡単にするものです。

しかし、イベントベースのアプローチでは、実行する他のスレッドはありません。メインイベントループだけです。そして、これは、イベントハンドラがブロックする呼び出しを発行した場合、サーバー全体がその処理を行うことを意味します。呼び出しが完了するまでブロックします。イベントループがブロックされると、システムはアイドル状態になり、リソースを浪費する可能性があります。したがって、イベントベースのシステムでは、ブロッキングコールは許可されていないため、ルールに従わなければなりません。

33.6 A Solution: Asynchronous I/O

この制限を克服するために、多くの最新のオペレーティングシステムでは、一般に非同期 I/O と呼ばれるディスクシステムに I/O 要求を発行する新しい方法が導入されています。これらのインターフェースを使用すると、I/O が完了する前にアプリケーションが I/O 要求を発行し、直ちに呼び出し元に制御を戻すことができます。追加のインターフェースにより、アプリケーションは様々な I/O が完了したかどうかを判断することができます。

たとえば、Mac で提供されているインターフェース（他のシステムにも同様の API があります）を調べてみ

ましょう。API は、共通の用語で基本構造、構造体 aiocb または AIO 制御ブロックを中心に展開されています。構造の簡略化されたバージョンは以下のようになります(詳細は、マニュアルページを参照してください)。

```
struct aiocb {
    int             aio_fildes;           /* File descriptor */
    off_t          aio_offset;          /* File offset */
    volatile void *aio_buf;            /* Location of buffer */
    size_t         aio_nbytes;          /* Length of transfer */
};
```

非同期読み取りをファイルに発行するには、アプリケーションは最初にこの構造体に関連する情報を書き込む必要があります。読み込むファイルのファイル記述子 (aio_fildes)、ファイル内のオフセット (aio_offset)、およびファイルの長さの要求 (aio_nbytes)、最後に、読み取り結果をコピーする対象のメモリ位置 (aio_buf) を指定します。

この構造体が埋め込まれた後、アプリケーションは非同期呼び出しを発行してファイルを読み取る必要があります。Mac では、この API は単に非同期読み取り API です。

```
int aio_read(struct aiocb *aiocbp);
```

このコールは I/O を発行しようとします。成功した場合はすぐに戻り、アプリケーション(つまり、イベントベースのサーバー)はその作業を続行できます。

しかし、解決しなければならないパズルの最後の部分が 1 つあります。I/O が完了し、バッファ (aio_buf が指す) が要求されたデータをその中に持つようになったら、どうすればわかりますか？1 つの最後の API が必要です。Mac では、aio_error() と呼ばれます(やや混乱します)。API は次のようにになります。

```
int aio_error(const struct aiocb *aiocbp);
```

このシステムコールは、aiocbp によって参照された要求が完了したかどうかをチェックします。存在する場合、ルーチンは成功を返します(ゼロで示されます)。そうでなければ、EINPROGRESS が返されます。したがって、すべての未処理の非同期 I/O に対して、アプリケーションは aio_error() の呼び出しを介してシステムを定期的にポーリングして、前記 I/O がまだ完了しているかどうかを判断できます。あなたが気付いたことの 1 つは、I/O が完了したかどうかを確認することは痛いということです。特定の時点で数十回または数百回の I/O が発行された場合、各プログラムを繰り返しチェックするか、最初に少し待つか、または...?

この問題を解決するために、一部のシステムでは割り込みに基づく手法が提供されています。この方法では、非同期 I/O が完了したときに UNIX 信号を使用してアプリケーションに通知するので、システムに繰り返し尋ねる必要がなくなります。このポーリングと割り込みの問題は、I/O デバイスの章で見られるように(または既に見ているように) デバイスでも見られます。

ASIDE: UNIX SIGNALS

近代的な UNIX のすべての変種には、シグナルと呼ばれる巨大で魅力的なインフラが存在します。シグナルは最も単純な方法で、プロセスと通信する手段を提供します。具体的には、信号をアプリケーションに配信することができます。そうすることにより、シグナルハンドラ、すなわち、その信号を処理するためのアプリケーション内のいくつかのコードを実行するために、アプリケーションが何をしているのかを停止させます。終了すると、プロセスは以前の動作を再開します。

各信号には、HUP(ハングアップ)、INT(割り込み)、SEGV(セグメンテーション違反)などの名前があります。詳細については、マニュアルページを参照してください。興味深いことに、時にはシグナルを行うのはカーネル自体です。例えば、あなたのプログラムがセグメンテーション違反に遭

遇すると、OS はそれに SIGSEGV を送ります（シグナル名の前に SIG が付いていることが一般的です）。あなたのプログラムがそのシグナルを捕捉するように設定されていれば、この誤ったプログラムの振る舞い（デバッグに役立つ可能性がある）に応答して実際にコードを実行することができます。シグナルを処理するように構成されていないプロセスにシグナルが送信されると、いくつかのデフォルト動作が成立します。SEGV の場合、プロセスは強制終了されます。無限ループに入るシンプルなプログラムですが、最初に SIGHUP を捕捉するシグナルハンドラを設定しています：

```
#include <stdio.h>
#include <signal.h>

void handle(int arg) {
    printf("stop wakin' me up...\n");
}

int main(int argc, char *argv[]) {
    signal(SIGHUP, handle);
    while (1)
        ; // doin' nothin' except catchin' some sigs
    return 0;
}
```

kill コマンドラインツールでシグナルを送ることができます（これは奇妙で攻撃的な名前です）。そうすることで、プログラム中の main while ループが中断され、ハンドラコード handle() が実行されます。

```
prompt> ./main &
[3] 36705
prompt> kill -HUP 36705
stop wakin' me up...
prompt> kill -HUP 36705
stop wakin' me up...
prompt> kill -HUP 36705
stop wakin' me up...
```

シグナルについて学ぶにはさらに多くのことがありますので、単一のページではなく、1つの章で十分ではありません。いつものように、Stevens と Rago という素晴らしい出典があります [SR05]。興味があればもっと読んでください。

非同期 I/O のないシステムでは、純粋なイベントベースのアプローチは実装できません。しかし、賢明な研究者は、その場所でかなりうまく機能する方法を導いてきました。例えば、Pai et al. [PDZ99] は、ネットワークパケットを処理するためにイベントが使用され、未処理の I/O を管理するためにスレッドプールが使用されるハイブリッドアプローチを説明しています。詳細は論文を参照してください。

33.7 Another Problem: State Management

イベントベースのアプローチのもう 1 つの問題は、そのようなコードは、一般に従来のスレッドベースのコードよりも書くのがより複雑であるということです。理由は次のとおりです。イベントハンドラが非同期 I/O を発行すると、I/O が最後に完了したときに使用する次のイベントハンドラのプログラム状態をパッケージ化する必要があります。スレッドベースの場合は、プログラムが必要とする状態がスレッドのスタック上にあるので、この追加の作業は必要ありません。Adya et al. これは、この作業マニュアルのスタック管理と呼ばれ、イベントベースのプログラミング [A+02] の基本です。

この点をより具体的にするために、スレッドベースのサーバがファイルディスクリプタ (fd) から読み込み、完了したらファイルから読み込んだデータをネットワークソケットディスクリプタ (sd) に書き込むという単純な例を見てみましょう。エラーチェックを無視したコードは次のようになります。

```
int rc = read(fd, buffer, size);
rc = write(sd, buffer, size);
```

ご覧のように、マルチスレッドプログラムでは、この種類の作業を行うのは簡単です。最終的に `read()` が返ってくると、コードはその情報がスレッドのスタック (変数 `sd` 内) にあるため、書き込むソケットをすぐに知ることができます。

イベントベースのシステムでは、それほど簡単ではありません。同じタスクを実行するには、先に説明した AIO 呼び出しを使用して、まず非同期に読み取りを発行します。次に、`aio_error()` 呼び出しを使用して読み取りの完了を定期的に確認します。その呼び出しによって読み取りが完了したことが通知されると、イベントベースのサーバーはどのように処理すべきかを知っていますか？

この解決策は、Adya et al に詳細が書かれています。[A + 02] は、継続 [FHK84] と呼ばれる古いプログラミング言語の構造を使用することです。複雑に思えますが、アイデアは単純です。基本的に、このイベントの処理を終了するために必要な情報をいくつかのデータ構造に記録します。イベントが発生したとき (すなわち、ディスク I/O が完了したとき) に、必要な情報を調べてイベントを処理します。この特定の場合、解決策は、ソケットディスクリプタ (sd) を、ファイルディスクリプタ (fd) によってインデックス付けされた何らかの種類のデータ構造 (例えば、ハッシュテーブル) に記録することです。ディスク I/O が完了すると、イベントハンドラはファイルディスクリプタを使用して継続を検索し、ソケットディスクリプタの値を呼び出し側に返します。この時点で (最後に)、サーバーはデータをソケットに書き込むために最後の作業を行えます。

33.8 What Is Still Difficult With Events

私たちが言及すべきイベントベースのアプローチには他にもいくつかの困難があります。たとえば、システムが単一の CPU から複数の CPU に移行したとき、イベントベースのアプローチの単純さのいくつかは消えました。具体的には、複数の CPU を使用するには、イベント・サーバーが複数のイベント・ハンドラーを並行して実行する必要があります。そうするときには、通常の同期問題 (例えば、クリティカルセクション) が発生し、通常の解決法 (例えば、ロック) が採用されなければならない。したがって、現代のマルチコアシステムでは、ロックなしの簡単なイベント処理はもはや不可能です。

イベントベースのアプローチのもう 1 つの問題は、ページングなどの特定の種類のシステムアクティビティとうまく統合できないことです。たとえば、イベント・ハンドラ・ページに障害が発生すると、そのイベント・ハンドラ・ページはブロックされ、ページ・フォルトが完了するまでサーバーは処理を進めません。サーバーが明示的なブロッキングを回避するように構成されていても、ページフォルトによるこの種類の暗黙のブロッキングは回避しにくいため、一般的な場合に大きなパフォーマンス上の問題が発生する可能性があります。

第3の問題は、さまざまなルーチンの正確なセマンティクスが変更されるため、イベントベースのコードを時間外管理するのが難しいことです[A + 02]。たとえば、ルーチンが非ブロッキングからブロッキングに変更された場合、そのルーチンを呼び出すイベントハンドラも、2つの部分に分けることによって、その新しい性質に適応するように変更する必要があります。ブロッキングはイベントベースのサーバーにとって非常に悲惨であるため、プログラマーは、各イベントが使用するAPIのセマンティクスにおけるそのような変更を常に把握している必要があります。

最後に、ほとんどのプラットフォームで非同期ディスクI/Oが可能になりましたが、そこに到達するまでには長い時間がかかりましたが、シンプルで均一な方法で非同期ネットワークI/Oと決して統合することは決してありません。たとえば、`select()`インターフェースを使用してすべての未処理I/Oを管理するのは簡単ですが、通常はネットワーキング用の`select()`とディスクI/OのAIO呼び出しの組み合わせが必要です。

33.9 Summary

私たちは、イベントに基づいて異なるスタイルの同時実行性について骨組みを紹介しました。イベントベースのサーバは、アプリケーション自体へのスケジューリングを制御するが、現代のシステム（例えば、ページング）の他の側面との統合の複雑さおよび困難性をいくらかのコストを払うことで実行しています。これらの課題のために、単一のアプローチが最良のものとして浮上したわけではありません。したがって、スレッドとイベントの両方は、今後数年間、同じ同時実行性問題に対する2つの異なるアプローチとして存続する可能性があります。いくつかの研究論文（例：[A + 02, PDZ99, vB + 03, WCB01]）を読んだり、イベントベースのコードを書いて詳細を調べることができます。

参考文献

- [A+02] “Cooperative Task Management Without Manual Stack Management”
Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, John R. Douceur
USENIX ATC '02, Monterey, CA, June 2002
This gem of a paper is the first to clearly articulate some of the difficulties of event-based concurrency, and suggests some simple solutions, as well explores the even crazier idea of combining the two types of concurrency management into a single application!
- [FHK84] “Programming With Continuations”
Daniel P. Friedman, Christopher T. Haynes, Eugene E. Kohlbecker
In Program Transformation and Programming Environments, Springer Verlag, 1984
The classic reference to this old idea from the world of programming languages. Now increasingly popular in some modern languages.
- [N13] “Node.js Documentation”
By the folks who build node.js
Available: <http://nodejs.org/api/>
One of the many cool new frameworks that help you readily build web services and applications. Every modern systems hacker should be proficient in frameworks such as this one (and likely, more than one). Spend the time and do some development in one of these worlds and become an expert.

- [O96] “Why Threads Are A Bad Idea (for most purposes)”
John Ousterhout
Invited Talk at USENIX '96, San Diego, CA, January 1996
A great talk about how threads aren't a great match for GUI-based applications (but the ideas are more general). Ousterhout formed many of these opinions while he was developing Tcl/Tk, a cool scripting

language and toolkit that made it 100x easier to develop GUI-based applications than the state of the art at the time. While the Tk GUI toolkit lives on (in Python for example), Tcl seems to be slowly dying (unfortunately).

[PDZ99] “Flash: An Efficient and Portable Web Server”

Vivek S. Pai, Peter Druschel, Willy Zwaenepoel

USENIX ’99, Monterey, CA, June 1999

A pioneering paper on how to structure web servers in the then-burgeoning Internet era. Read it to understand the basics as well as to see the authors’ ideas on how to build hybrids when support for asynchronous I/O is lacking.

[SR05] “Advanced Programming in the UNIX Environment”

W. Richard Stevens and Stephen A. Rago

Addison-Wesley, 2005

Once again, we refer to the classic must-have-on-your-bookshelf book of UNIX systems programming. If there is some detail you need to know, it is in here.

[vB+03] “Capriccio: Scalable Threads for Internet Services”

Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, Eric Brewer

SOSP ’03, Lake George, New York, October 2003

A paper about how to make threads work at extreme scale; a counter to all the event-based work ongoing at the time.

[WCB01] “SEDA: An Architecture for Well-Conditioned, Scalable Internet Services”

Matt Welsh, David Culler, and Eric Brewer

SOSP ’01, Banff, Canada, October 2001

A nice twist on event-based serving that combines threads, queues, and event-based hanlding into one streamlined whole. Some of these ideas have found their way into the infrastructures of companies such as Google, Amazon, and elsewhere.

第 III 部

Persistence

36 I/O Devices

本書のこの部分(永続性)の主な内容を掘り下げる前に、入出力(I/O)デバイスの概念を紹介し、オペレーティングシステムがそのようなエンティティとどのように対話するかを示します。もちろん、I/Oはコンピュータシステムにとって非常に重要です。入力なしのプログラムを想像してください(毎回同じ結果を出します)。今は出力のないプログラムを想像してください(何が目的ですか?)。明らかに、コンピュータシステムが面白いためには、入力と出力の両方が必要です。

CRUX: HOW TO INTEGRATE I/O INTO SYSTEMS

I/Oをシステムにどのように組み込むべきですか？一般的な仕組みは何ですか？どのように効率的にすることができますか？

36.1 System Architecture

議論を始めるには、典型的なシステムの構造を見てみましょう(図36.1)。画像は、ある種のメモリバスまたは相互接続を介してシステムのメインメモリに接続された単一のCPUを示しています。デバイスの中には、一般的なI/Oバスを介してシステムに接続されているものがあります。多くの現代システムでは、PCI(またはその多くの派生品)の1つです。グラフィックスやその他の高性能I/Oデバイスがここにあります。最後に、さらに低いものは、SCSI、SATA、またはUSBなどの周辺バスと呼ばれるものの1つ以上です。これらは、ディスク、マウス、およびその他の同様のコンポーネントを含む、最も遅いデバイスをシステムに接続します。

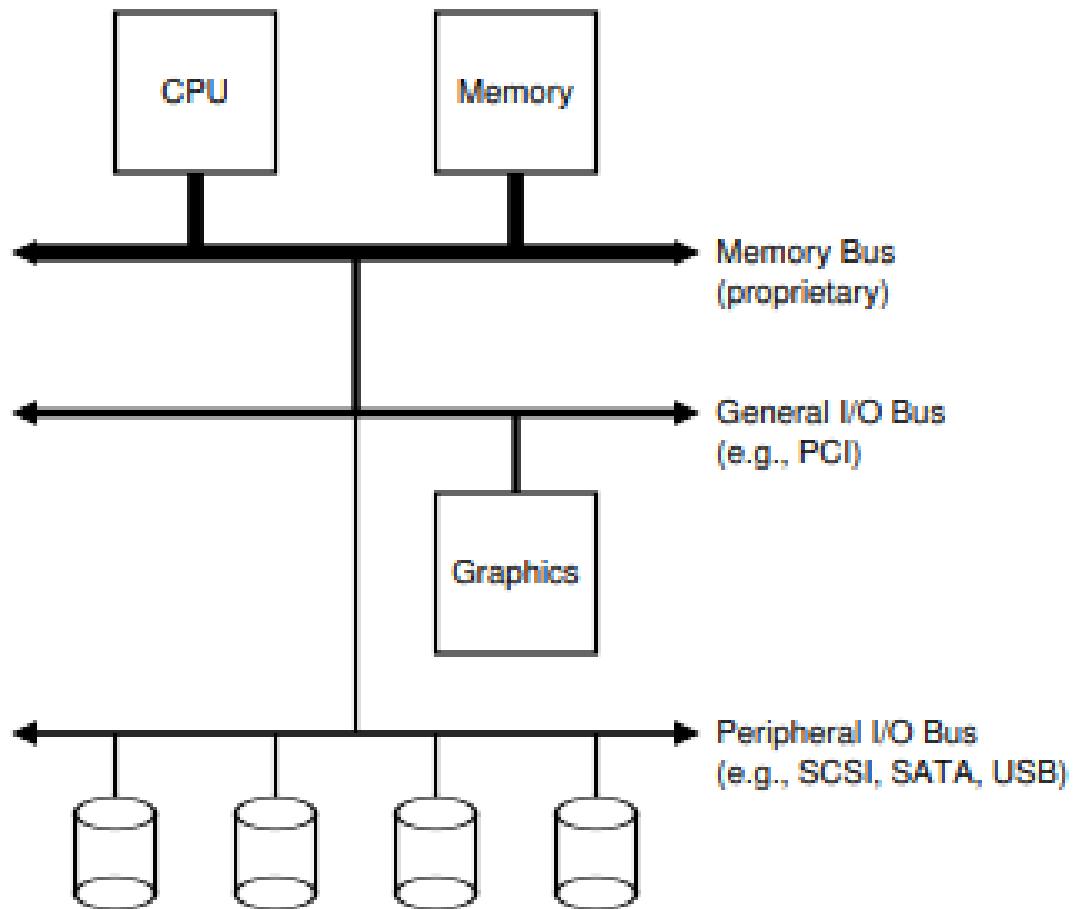


Figure 36.1: Prototypical System Architecture

あなたが求めるかもしれない 1 つの質問は、なぜこのような階層構造が必要なのかということです。単純に言えば、物理学とコストです。バスが速ければ速いほど短くなければなりません。したがって、高性能メモリバスは、デバイスなどをプラグインする余地があまりありません。さらに、高性能のためにバスを設計することはかなりコストがかかります。したがって、システム設計者は、グラフィックカードなどの高性能を要求するコンポーネントが CPU に近いほど、この階層的アプローチを採用しています。性能の低いコンポーネントはさらに遠くにあります。周辺バスにディスクやその他の低速デバイスを配置する利点は多岐にわたっています。特に、多数のデバイスを配置することができます。

36.2 A Canonical Device

標準のデバイス（実際のものではない）を見て、このデバイスを使用してデバイスの相互作用を効率的にするために必要な機械のいくつかの理解を促進しましょう。図 36.2 から、デバイスには 2 つの重要なコンポーネントがあることがわかります。1 つ目は、システムの残りの部分に提示するハードウェアインターフェイスです。ソフトウェアのように、ハードウェアは、システムソフトウェアがその動作を制御することを可能にする何らかの種類のインターフェースも提示しなければいけません。したがって、すべてのデバイスは、典型的な相互作用のための特定のインターフェースおよびプロトコルがあります。

任意のデバイスの第 2 の部分は、その内部構造です。デバイスのこの部分は実装固有のものであり、デバイスがシステムに提示する抽象化を実装する責任があります。非常にシンプルなデバイスは、その機能を実装するために 1 つまたはいくつかのハードウェアチップを持ちます。より複雑なデバイスには、シンプルな CPU、

汎用メモリ、および他のデバイス固有のチップが含まれ、仕事を得て、おわらせます。例えば、最新の RAID コントローラは、その機能を実現するために、数十万行のファームウェア(すなわち、ハードウェアデバイス内のソフトウェア)から構成されています。

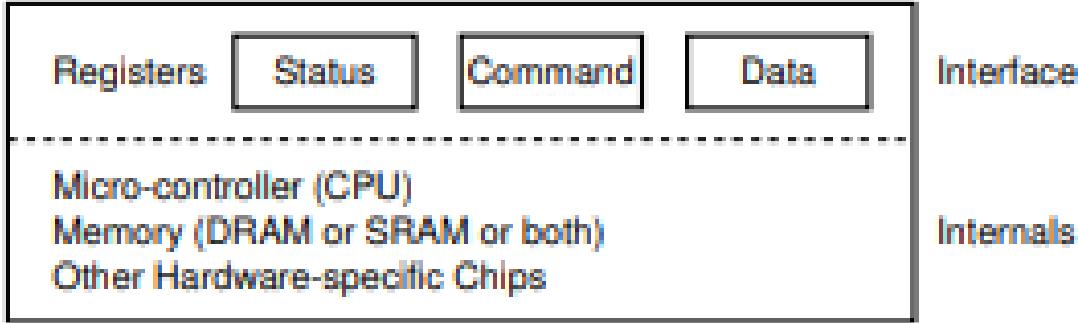


Figure 36.2: A Canonical Device

36.3 The Canonical Protocol

上記の図では、(簡略化された) デバイスインターフェースは 3 つのレジスタで構成されています。ステータスレジスタは、デバイスの現在の状態を見るために読み出すことができます。コマンドレジスタは特定のタスクを実行するようにデバイスに指示します。データレジスタはデバイスにデータを渡すか、またはデバイスからデータを取得します。これらのレジスタを読み書きすることにより、オペレーティングシステムはデバイスの動作を制御できます。ここでは、デバイスがそのために何かを行うために、OS とデバイスが持つ可能性がある典型的な相互作用について説明します。プロトコルは次のとおりです。

```

While (STATUS == BUSY)
    ; // wait until device is not busy
Write data to DATA register
Write command to COMMAND register
    (Doing so starts the device and executes the command)
While (STATUS == BUSY)
    ; // wait until device is done with your request

```

プロトコルには 4 つのステップがあります。最初に、OS は、デバイスがステータスレジスタを繰り返し読み出すことによってコマンドを受信する準備ができるまで待機します。このポーリングをデバイスと呼びます(基本的に、何が起こっているのかを尋ねるだけです)。第 2 に、OS はいくつかのデータをデータレジスタに送る。これがディスクの場合、たとえばディスクブロック(たとえば 4KB)をデバイスに転送するために複数の書き込みが行われる必要があると想像することができます。メイン CPU が(このプロトコル例のように)データ移動に関与する場合、これをプログラム I/O(PIO)と呼びます。第 3 に、OS はコマンドレジスタにコマンドを書き込む。そうすることで、暗黙的に、データが存在し、コマンドで作業を開始する必要があることをデバイスに知らせることができます。最後に、OS はデバイスが終了するのを待って、ループでポーリングして終了するかどうかを確認します(成功または失敗を示すエラーコードが表示されることがあります)。

この基本的なプロトコルは、単純で働きやすいという肯定的な側面を持っています。しかし、いくつかの非効率性と不便さがあります。プロトコルで最初に気つく問題は、ポーリングが非効率的であるように見えることです。具体的には、デバイスがそのアクティビティを完了するのを待つだけで CPU 時間を大量に無駄する代わりに、別の準備完了プロセスに切り替え(潜在的に遅い)、CPU をより有効に利用しています。

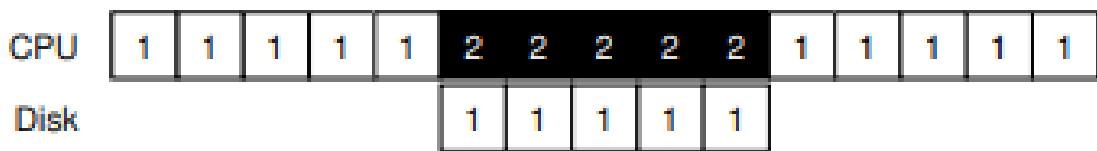
36.4 Lowering CPU Overhead With Interrupts

この相互作用を改善するために何年も前に多くのエンジニアが出てきた発明は、すでに見えてきたことです。デバイスを繰り返しポーリングする代わりに、OS は要求を発行し、呼び出しプロセスをスリープ状態にし、コンテキストを別のタスクに切り替えることができます。デバイスが最終的に動作を終了すると、ハードウェア割り込みが発生し、CPU が所定の割り込みサービスルーチン (ISR) またはより単純に割り込みハンドラで OS にジャンプします。ハンドラは、要求を終了する（たとえば、データおよびおそらくはデバイスからのエラーコードを読み取る）オペレーティングシステムコードの一部に過ぎず、I/O を待っているプロセスを復帰させ、必要に応じて処理をすすめることができます。

したがって、割り込みは、計算と I/O の重複を可能にします。これは、使用率を向上させるための鍵です。このタイムラインは問題を示しています



この図では、プロセス 1 は CPU で一定時間実行され (CPU 行で 1 が繰り返されます)、ディスクに I/O 要求を発行してデータを読み取ります。割り込みがなければ、システムは単に回転し、I/O が完了するまで (p で示される)、デバイスの状態を繰り返しポーリングします。ディスクは要求を処理し、最後にプロセス 1 を再度実行できます。代わりに、割り込みを利用して重複を許可すると、OS はディスクを待っている間に何か他のことをすることができます。



この例では、ディスクサービスプロセス 1 の要求中に、OS は CPU 上でプロセス 2 を実行します。ディスク要求が終了すると、割り込みが発生し、OS はプロセス 1 を起動して再度実行します。したがって、CPU とディスクの両方は、中程度の時間に適切に利用されます。

割り込みを使用することは常に最適な解決方法ではないことに注意してください。たとえば、タスクを非常に迅速に実行するデバイスを想像してください。最初のポーリングでは通常、タスクを実行するデバイスが見つけられます。この場合に割り込みを使用すると、実際にはシステムの速度が遅くなります。別のプロセスに切り替え、割り込みを処理し、発行プロセスに戻すのはコストがかかります。したがって、デバイスが高速であれば、ポーリングするのが最善の方法です。遅い場合、オーバーラップを許す割り込みが最適です。デバイスの速度がわかっていないか、時には高速で時には遅い場合は、しばらくポーリングするハイブリッドを使用し、デバイスがまだ終了していない場合は割り込みを使用するのが最善の方法です。この 2 段階アプローチは、両方の世界のベストを達成するかもしれません。

TIP: INTERRUPTS NOT ALWAYS BETTER THAN PIO

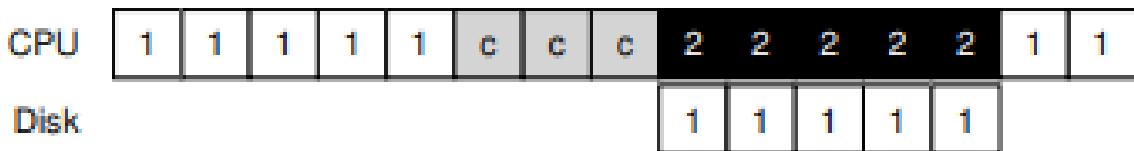
割り込みは計算と I/O のオーバーラップを可能にしますが、遅いデバイスには本当に意味があります。それ以外の場合は、割込み処理とコンテキスト切り替えのコストが割込みの利点を上回る可能性があります。また、割り込み flood がシステムに過負荷をかけ、それをライブロックに導く場合もあります [MR96]。そのような場合、ポーリングは OS のスケジューリングにおいてより多くの制御を提供し、したがって再び有用です。

割り込みを使用しない別の理由は、ネットワーク [MR96] で発生します。膨大なストリームの着信パケットがそれぞれ割り込みを生成すると、OS はライブロックすることができます。つまり、割り込み処理のみを実行し、ユーザーレベルのプロセスを実行して実際に要求を処理することはできません。たとえば、スラッシュドット効果のために突然負荷が高くなる Web サーバーを想像してください。この場合、ポーリングを使用して、システムで発生していることをよりうまく制御し、Web サーバーがいくつかの要求にサービスを提供してから、デバイスに戻ってより多くのパケット到着を確認することをお勧めします。

別の割り込みベースの最適化が統合されています。このような設定では、割り込みを発生させる必要のあるデバイスは、最初に CPU に割り込みを送信する前に少し待っています。待機している間に、他の要求が間もなく完了し、複数の割り込みを 1 つの割り込みデリバリにまとめて割り込み処理のオーバーヘッドを減らすことができます。もちろん、待ち時間が長すぎると、システムの一般的なトレードオフであるリクエストの待ち時間が長くなります。Ahmad et al. [A + 11] は素晴らしい要約です。

36.5 More Efficient Data Movement With DMA

残念ながら、私たちの注意を必要とする標準プロトコルのもう一つの側面があります。特に、プログラムされた I/O(PIO) を使用して大量のデータをデバイスに転送する場合、CPU はもう少し過酷な作業で再び過負荷になり、実行に多くの時間と労力を費やし、他のプロセスの実行にもっと費やすことができる努力を無駄にします。このタイムラインは問題を示しています



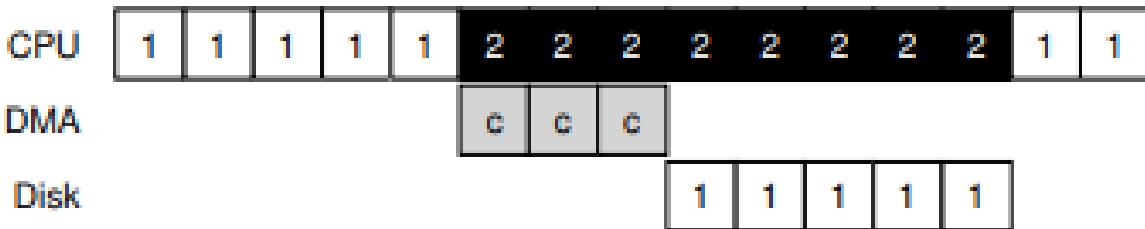
タイムラインでは、プロセス 1 が実行されていて、ディスクにデータを書きたいと考えています。その後、I/O を開始します。この I/O は、一度に 1 ワードずつメモリからデバイスに明示的にデータをコピーする必要があります (図の c)。コピーが完了すると、I/O はディスク上で開始され、CPU は最終的に他の用途に使用されます。

THE CRUX: HOW TO LOWER PIO OVERHEADS

PIO を使用すると、CPU は手作業でデバイス間でデータを移動するのに時間がかかりすぎます。どのようにしてこの作業をオフロードし、CPU をより効率的に利用できるようにするか？

この問題に対する解決策は、ダイレクトメモリアクセス (DMA) と呼ばれるものです。DMA エンジンは基本的にシステム内の非常に特殊なデバイスであり、多くの CPU の介入なしにデバイスとメインメモリ間の転送を調整することができます。

DMA は次のように動作します。たとえば、デバイスにデータを転送するには、OS はメモリにデータがどこにあるのか、どのくらいのデータをコピーするのか、どのデバイスにデータを送るのかを伝えることで DMA エンジンをプログラムします。その時点で、OS は転送を終え、他の作業を進めることができます。DMA が完了すると、DMA コントローラは割り込みを発生させ、したがって OS は転送が完了したことを知ります。改訂されたタイムラインでは以下のようになります。



タイムラインから、データのコピーが DMA コントローラによって処理されることがわかります。その間に CPU はフリーであるため、OS はプロセス 2 を実行することを選択して別の処理を行うことができます。プロセス 1 はプロセス 1 が再び実行される前に、より多くの CPU を使用します。

36.6 Methods Of Device Interaction

I/O を実行する際の効率の問題があるので、デバイスを最新のシステムに組み込むために処理する必要があるいくつかの問題があります。これまでに気がついた問題の 1 つは、OS が実際にデバイスとどのように通信しているかについては何も言いませんでした。

THE CRUX: HOW TO COMMUNICATE WITH DEVICES

ハードウェアとデバイスとの通信方法明示的な指示があるべきか？ それともそれ以外の方法がありますか？

時間の経過と共に、デバイス通信の 2 つの主要な方法が開発されています。最初の、(IBM のメインフレームで長年使用されている) 最も古い方法は、明示的な I/O 命令を持つことです。これらの命令は、OS が特定のデバイスレジスタにデータを送信する方法を指定し、したがって上述のプロトコルの構築を可能にします。

たとえば、x86 では、in および out 命令を使用してデバイスと通信することができます。たとえば、デバイスにデータを送信するには、呼び出し元はデータが入っているレジスタとデバイスに名前を付ける特定のポートを指定します。命令を実行すると、目的の動作になります。

そのような指示は通常特権が与えられます。OS はデバイスを制御するため、OSだけは直接通信できるエンティティとして許可されています。プログラムがディスクを読み書きできるかどうかを想像してみましょう。例えば、すべてのカオスはいつものように、どんなユーザプログラムもそのような抜け穴を使ってマシンを完全に制御することができます。

デバイスと対話する第 2 の方法は、メモリマップ I/O と呼ばれます。このアプローチでは、ハードウェアはデバイスのレジスタをあたかもメモリの場所のように使用可能にします。特定のレジスタにアクセスするために、OS はアドレスをロード (読み込み) または格納 (書き込み) します。ハードウェアはメインメモリではなくデバイスにロード/ストアをルーティングします。あるアプローチや他のアプローチに大きな利点はありません。メモリマップされたアプローチは、それをサポートするための新しい命令は必要ないという点で素晴らしいですが、どちらのアプローチも今日でもまだ使用されています。

36.7 Fitting Into The OS: The Device Driver

最終的に私たちが議論する問題の 1 つは、それぞれ固有のインターフェースを持つデバイスを OS に組み込む方法です。できるだけ一般的なものにしたいと考えています。たとえば、ファイルシステムを考えてみましょう。私たちは、SCSI ディスク、IDE ディスク、USB キーチェーンドライブなどの上で動作するファイルシステムを構築したいと考えています。また、ファイルシステムがこれらの差分があるタイプのドライブに対してどのように読み書き要求を発行するかの詳細すべてを比較的知らないことを望んでいます。

THE CRUX: HOW TO BUILD A DEVICE-NEUTRAL OS

どのようにして OS のデバイスの中立性を最大限に保ち、主要な OS サブシステムからのデバイスの相互作用の詳細を隠すことができますか？

この問題は、古くからの抽象化技術によって解決されています。最低レベルでは、OS の一部のソフトウェアは、デバイスがどのように動作するかを詳細に把握している必要があります。このソフトウェアはデバイスドライバと呼ばれ、デバイスの相互作用の詳細はすべてカプセル化されています。

この抽象化が、Linux ファイルシステムソフトウェアスタックを調べることによって、OS の設計と実装にどのように役立つかを見てみましょう。図 36.3 は、Linux ソフトウェアの組織の概略図です。ダイアグラムから分かるように、ファイルシステム（もちろん、上記のアプリケーション）は、使用しているディスククラスの詳細を完全に無視しています。ロック読み出しおよび書き込み要求を汎用ロックレイヤーに発行するだけで、適切なデバイスドライバにルーティングされ、特定の要求を発行する詳細が処理されます。簡略化されていますが、この図は、そのような詳細がほとんどの OS からどのように隠れるかを示しています。

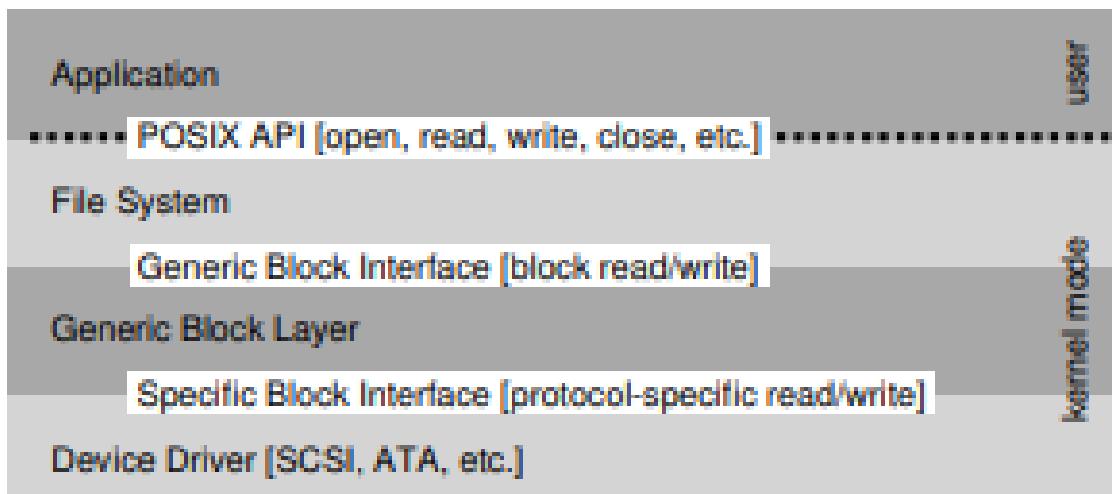


Figure 36.3: The File System Stack

このようなカプセル化は、同様にその欠点を有する可能性があることを覚えておいてください。例えば、多くの特殊な機能を備えているが、残りのカーネルに汎用インターフェースを提示しなければならないデバイスがある場合、それらの特殊機能は使用されなくなります。このような状況は、例えば、SCSI デバイスを搭載した Linux では非常に多くのエラー報告があります。他のロックデバイス（例えば、ATA/IDE）はずっと簡単なエラー処理をもっているので、これまでに受け取ったより高いレベルのソフトウェアの全ては一般的な EIO(一般的な IO エラー) エラーコードです。したがって、SCSI が提供している可能性のあるエラーの詳細は、ファイルシステム（高いレベルのソフトウェア）[G08] では失われます。

興味深いことに、あなたのシステムに接続する可能性のあるデバイスにはデバイスドライバが必要であるため、時間の経過とともにカーネルコードの膨大な割合を占めています。Linux カーネルの研究では、OS コードの 70 %以上がデバイスドライバ [C01] に存在することが明らかになりました。Windows ベースのシステムでも、かなり高い可能性で同じくらいあります。したがって、OS に何百万行ものコードが含まれていると言うと、実際には OS には何百万行ものデバイスドライバコードが含まれています。当然のことながら、任意のインストールに対して、そのコードの大部分はアクティブではないです（すなわち、一度にいくつかのデバイスのみがシステムに接続される）。ドライバーは（フルタイムのカーネル開発者の代わりに）「アマチュア」によって書かれていることが多いため、より多くのバグを持つ傾向があり、カーネルクラッシュの主な原因となります [S03]。

36.8 Case Study: A Simple IDE Disk Driver

ここでもう少し詳しく調べるために、実際のデバイスを簡単に見てみましょう。IDE ディスクドライブ [L94] です。この参考文献 [W10] に記載されているプロトコルを要約します。実際の IDE ドライバ [CK + 08] の簡単な例については xv6 ソースコードを見ていきます。

IDE ディスクは、コントロール、コマンドブロック、ステータス、およびエラーの 4 種類のレジスタで構成されるシンプルなインターフェイスをシステムに提供します。これらのレジスタは、I/O 命令の入出力 (x86 では) を使用して、特定の “I/O アドレス”(以下の 0x3F6 など) を読み書きすることで利用できます。

```

Control Register:
  Address 0x3F6 = 0x08 (0000 1RE0) : R=reset, E=0 means "enable interrupt"

Command Block Registers:
  Address 0x1F0 = Data Port
  Address 0x1F1 = Error
  Address 0x1F2 = Sector Count
  Address 0x1F3 = LBA low byte
  Address 0x1F4 = LBA mid byte
  Address 0x1F5 = LBA hi byte
  Address 0x1F6 = 1B1D TOP4LBA: B=LBA, D=drive
  Address 0x1F7 = Command/status

Status Register (Address 0x1F7):
    7      6      5      4      3      2      1      0
    BUSY   READY  FAULT SEEK  DRQ   CORR IDDEX ERROR

Error Register (Address 0x1F1): (check when Status ERROR==1)
    7      6      5      4      3      2      1      0
    BBK    UNC    MC     IDNF   MCR   ABRT  TONF AMNF

BBK = Bad Block
UNC = Uncorrectable data error
MC = Media Changed
IDNF = ID mark Not Found
MCR = Media Change Requested
ABRT = Command aborted
TONF = Track 0 Not Found
AMNF = Address Mark Not Found

```

Figure 36.4: The IDE Interface

デバイスと相互作用するための基本的なプロトコルは、既に初期化されているものとします。

- ドライブの準備が整うのを待ちます。ドライブが READY で BUSY になるまでステータスレジスタ (0x1F7) を読み込みます。
- コマンドレジスタにパラメータを書き込む。アクセスするセクタのセクタ数、論理ブロックアドレス (LBA)、およびコマンド番号 (0x1F2-0x1F6) にドライブ番号 (IDE が 2 つのドライブのみを許可するため、マスター = 0x00 またはスレーブ = 0x10) を書き込みます。
- I/O を開始します。コマンドレジスタへの読出し/書込みを発行することによって、コマンドレジスタ (0x1F7) に READ-WRITE コマンドを書き込みます。
- データ転送 (書き込み用)：ドライブステータスが READY および DRQ(データのドライブ要求) にな

るまで待ちます。データポートにデータを書き込みます。

- 割り込みを処理します。最も単純なケースでは、転送されるセクタごとに割り込みを処理します。より複雑なアプローチはバッチ処理と転送全体が完了したときに最終的な割り込みを可能にします。
- エラー処理。各操作の後、ステータスレジスタを読み出します。ERROR ビットがオンの場合は、エラー・レジスタを参照してください。

このプロトコルの大部分は、xv6 IDE ドライバ (図 36.5) にあります。初期化後、4 つの主な機能が動作します。最初のものは `ide_rw()` で、要求がキューに入れられているかどうかを確認したり、`ide_start_request()` を使ってディスクに直接発行したりします。どちらの場合も、ルーチンは要求が完了するのを待つため、呼び出しプロセスがスリープ状態になります。2 番目は `ide_start_request()` で、要求 (ディスクの場合はデータ) をディスクに送信するために使用されます。インおよびアウト x86 命令は、それぞれデバイスレジスタを読み書きするために呼び出されます。開始要求ルーチンは、3 番目の関数 `ide_wait_ready()` を使用して、要求を発行する前にドライブが準備状態になっていることを確認します。最後に、割り込みが発生すると `ide_intr()` が呼び出されます。(要求が書き込みではなく読み込みである場合) デバイスからデータを読み込み、I/O が完了するのを待つプロセスを起動し、(I/O キューにさらに要求がある場合)、`ide_start_request()` を介して次の I/O に移ります。

```

static int ide_wait_ready() {
    while (((int r = inb(0x1f7)) & IDE_BSY) || !(r & IDE_DRDY))
        ; // loop until drive isn't busy
}

static void ide_start_request(struct buf *b) {
    ide_wait_ready();
    outb(0x3f6, 0); // generate interrupt
    outb(0x1f2, 1); // how many sectors?
    outb(0x1f3, b->sector & 0xff); // LBA goes here ...
    outb(0x1f4, (b->sector >> 8) & 0xff); // ... and here
    outb(0x1f5, (b->sector >> 16) & 0xff); // ... and here!
    outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
    if(b->flags & B_DIRTY) {
        outb(0x1f7, IDE_CMD_WRITE); // this is a WRITE
        outsl(0x1f0, b->data, 512/4); // transfer data too!
    } else {
        outb(0x1f7, IDE_CMD_READ); // this is a READ (no data)
    }
}

void ide_rw(struct buf *b) {
    acquire(&ide_lock);
    for (struct buf **pp = &ide_queue; *pp; pp=&(*pp)->qnext)
        ; // walk queue
    *pp = b; // add request to end
    if (ide_queue == b) // if q is empty
        ide_start_request(b); // send req to disk
    while ((b->flags & (B_VALID|B_DIRTY)) != B_VALID)
        sleep(b, &ide_lock); // wait for completion
    release(&ide_lock);
}

void ide_intr() {
    struct buf *b;
    acquire(&ide_lock);
    if (!(b->flags & B_DIRTY) && ide_wait_ready() >= 0)
        insl(0x1f0, b->data, 512/4); // if READ: get data
    b->flags |= B_VALID;
    b->flags &= ~B_DIRTY;
    wakeup(b); // wake waiting process
    if ((ide_queue = b->qnext) != 0) // start next request
        ide_start_request(ide_queue); // (if one exists)
    release(&ide_lock);
}

```

Figure 36.5: The xv6 IDE Disk Driver (Simplified)

36.9 Historical Notes

終了する前に、これらの基本的なアイデアの起源に関する簡単な歴史的な注記を掲載します。あなたがもつと学ぶことに興味があるなら、Smotherman の優れた要約 [S08] を読んでください。

割り込みは古代の考えであり、最も初期のマシンに存在します。例えば、1950 年代初期の UNIVAC は、こ

の機能が利用可能であったのかどうかは明確ではありませんが、何らかの形の割り込みベクタリングを持っていました [S08]。残念なことに、搖籃期（物事が張ってする初期の段階）であっても、私たちはコンピューティングの歴史の始まりを失い始めています。

DMA の考え方をどのマシンが最初に導入したのかについていくつかの議論があります。例えば、Knuth らは、DYSEAC(その時点でトレーラーで運ばれることができる「モバイル」マシン)を指していますが、他の人は IBM SAGE が最初の [S08] であったかもしれませんと考えています。どちらの方法でも、50 年代半ばまでに、メモリと直接通信し、終了時に CPU を中断する I/O デバイスを持つシステムが存在しました。

この発明の歴史は現実、または時にはあいまいに機械と結びついているので、ここまで歴史は辿るのが難しいです。たとえば、リンカーン・ラボの TX-2 マシンが最初にベクトル化された割込みであると考える人もいますが (S08)、これはほとんどわかりません。

アイデアは比較的分かりやすいので、遅い I/O を待っている間に CPU に何か他のことをさせるというアイデアは自然に生まれたものでしょう。おそらく、“誰が先に？”というのは関係ありません。確かに明らかなのは、人々がこれらの初期のマシンを構築すると、I/O サポートが必要だったということです。割り込み、DMA、および関連するアイデアは、すべて高速の CPU と低速デバイスの性質の自然な結果です。あなたがその時にそこにいたなら、あなたは似た考えを持っていたかもしれません。

36.10 Summary

これで、OS がデバイスとどのようにやりとりするかについて、基本的に理解しているはずです。デバイスの効率を助けるために、割り込みと DMA の 2 つの手法が導入されており、デバイス・レジスタ、明示的な I/O 命令、およびメモリ・マップされた I/O にアクセスする 2 つの方法の詳細が説明されています。最後に、デバイスドライバの概念が提示され、OS 自体がどのように低レベルの詳細をカプセル化して、残りの OS をデバイス中立的な方法で簡単に構築できるかを示しています。

参考文献

- [A+11] “vIC: Interrupt Coalescing for Virtual Machine Storage Device IO”
Irfan Ahmad, Ajay Gulati, Ali Mashtizadeh
USENIX ’11
A terrific survey of interrupt coalescing in traditional and virtualized environments.
- [C01] “An Empirical Study of Operating System Errors”
Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, Dawson Engler
SOSP ’01
One of the first papers to systematically explore how many bugs are in modern operating systems. Among other neat findings, the authors show that device drivers have something like seven times more bugs than mainline kernel code.
- [CK+08] “The xv6 Operating System”
Russ Cox, Frans Kaashoek, Robert Morris, Nickolai Zeldovich
From: <http://pdos.csail.mit.edu/6.828/2008/index.html>
See ide.c for the IDE device driver, with a few more details therein.
- [D07] “What Every Programmer Should Know About Memory”
Ulrich Drepper
November, 2007
Available: <http://www.akkadia.org/drepper/cpumemory.pdf>
A fantastic read about modern memory systems, starting at DRAM and going all the way up to virtu-

alization and cache-optimized algorithms.

[G08] “EIO: Error-handling is Occasionally Correct”

Haryadi Gunawi, Cindy Rubio-Gonzalez, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Ben Liblit
FAST '08, San Jose, CA, February 2008

Our own work on building a tool to find code in Linux file systems that does not handle error return properly. We found hundreds and hundreds of bugs, many of which have now been fixed.

[L94] “AT Attachment Interface for Disk Drives”

Lawrence J. Lamers, X3T10 Technical Editor

Available: <ftp://ftp.t10.org/t13/project/d0791r4c-ATA-1.pdf>

Reference number: ANSI X3.221 - 1994 A rather dry document about device interfaces. Read it at your own peril.

[MR96] “Eliminating Receive Livelock in an Interrupt-driven Kernel”

Jeffrey Mogul and K. K. Ramakrishnan

USENIX '96, San Diego, CA, January 1996

Mogul and colleagues did a great deal of pioneering work on web server network performance. This paper is but one example.

[S08] “Interrupts”

Mark Smotherman, as of July '08

Available: <http://people.cs.clemson.edu/~mark/interrupts.html>

A treasure trove of information on the history of interrupts, DMA, and related early ideas in computing

[S03] “Improving the Reliability of Commodity Operating Systems”

Michael M. Swift, Brian N. Bershad, and Henry M. Levy

SOSP '03

Swift's work revived interest in a more microkernel-like approach to operating systems; minimally, it finally gave some good reasons why address-space based protection could be useful in a modern OS.

[W10] “Hard Disk Driver”

Washington State Course Homepage

Available: <http://eecs.wsu.edu/~cs460/cs560/HDdriver.html>

A nice summary of a simple IDE disk drive's interface and how to build a device driver for it.

37 Hard Disk Drives

最後の章では、I/O デバイスの一般的な概念を紹介し、OS がそのような歴史とどのように相互作用するかを示しました。この章では、特に 1 つのデバイス、ハードディスクドライブについて詳しく説明します。これらのドライブは、何十年もの間、コンピュータシステムにおける永続的なデータストレージの主要な形態であり、ファイルシステム技術の開発の多くは、その動作を前提としています。したがって、それを管理するファイルシステムソフトウェアを構築する前に、ディスク操作の詳細を理解することは価値があります。これらの詳細の多くは、Ruemmler と Wilkes [RW92] と Anderson、Dykes、Riedel [ADR03] の優れた論文で利用できます。

CRUX: HOW TO STORE AND ACCESS DATA ON DISK

最新のハードディスクドライブはどのようにデータを保存しますか？ インターフェイスとは何ですか？ データは実際にどのように割り当てられ、アクセスされますか？ ディスクスケジューリングはどのようにパフォーマンスを改善しますか？

37.1 The Interface

現代のディスクドライブとのインターフェースを理解することから始めましょう。現代のすべてのドライブの基本的なインターフェースは簡単です。ドライブは、多数のセクタ (512 バイトごとのブロック) で構成され、それぞれのセクタは読み書き可能です。セクタは、n セクタのディスク上で 0 から n-1 まで番号が付けられます。したがって、ディスクをセクタの配列として見ることができます。0～n-1 はドライブのアドレス空間です。

マルチセクタ操作が可能です。確かに、多くのファイルシステムは一度に 4KB を読み書きします。しかし、ディスクを更新するとき、ドライブ製造業者が行う唯一の保証は、単一の 512 バイト書き込みがアトミックである (すなわち、その全体が完了するか、または完了しない) ことである。したがって、不意の電力損失が発生した場合、より大きな書き込みの一部のみが完了することがある (時には、torn write(破れた書き込み) と呼ばれる)。

ディスクドライブのほとんどのクライアントはいくつかの前提がありますが、それはインターフェイスに直接指定されていません。Schlosser と Ganger はこれをディスクドライブの「未書き込み契約」と呼んでいます [SG04]。具体的には、通常、ドライブのアドレス空間内で互いに隣接する 2 つのブロックにアクセスすることは、離れている 2 つのブロックにアクセスするよりも高速であると想定することができます。連続したチャンク (すなわち、順序読み込みまたは書き込み) 内のブロックにアクセスすることが最速のアクセスモードであり、通常はそれ以上のランダムアクセスパターンよりもはるかに高速であると通常仮定してもよいです。

37.2 Basic Geometry

現代のディスクのいくつかのコンポーネントを理解してみましょう。まずは、磁気的な変化を誘発することによってデータが永続的に保存される円形の硬い表面であるプラッターから始めます。ディスクには 1 つまたは複数のプラッタがあります。各プラッターは 2 面を持ち、それぞれの面はサーフェスと呼ばれます。これらのプラッタは、通常、アルミニウムなどの硬い材料で作られ、薄い磁性層でコーティングされているため、ドライブの電源を切ってもドライブがビットを永続的に保存できます。

プラッタはスピンドルの周りに一緒に拘束されています。スピンドルは、一定の速度で (ドライブの電源がオンの状態で) プラッタを回転させるモータに接続されています。回転速度は RPM(回転数/分) で測定される

ことが多く、典型的な現代的な値は 7,200 RPM～15,000 RPM の範囲です。1 回転の時間に興味があることが多いことに注意してください。たとえば、10,000 RPM で回転するドライブは、1 回転に約 6 ミリ秒 (6 ms) かかるることを意味します。

データはセクタの同心円内の各サーフェスでエンコードされます。そのような同心円を 1 つのトラックと呼びます。単一のサーフェスには、数千、数千のトラックが含まれています。これらのトラックはしっかりと詰め込まれており、人間の髪の毛の幅に収まる数百のトラックがあります。

表面から読み書きするためには、ディスク上の磁気パターンを感知 (すなわち、読み取る) するか、またはそれらに変化を誘発 (すなわち書き込む) することができる機構が必要です。この読み書きのプロセスは、ディスクヘッドによって解決されます。ドライブの表面に 1 つのそのようなヘッドがあります。ディスクヘッドは単一のディスクアームに取り付けられ、ディスクアームは表面を横切って移動し、ヘッドを望んだトラック上に位置決めします。

37.3 A Simple Disk Drive

一度に 1 つのモデルを 1 つのトラックにまとめることで、ディスクの仕組みを理解してみましょう。1 つのトラックを持つシンプルなディスクがあるとします (図 37.1)。このトラックには 12 のセクタがあり、それぞれのセクタのサイズは 512 バイト (通常のセクタサイズ、リコール) であり、したがって 0～11 の数字で表されます。ここにある単一のプラッタは、モータが取り付けられたスピンドルを中心に回転します。もちろん、トラック自体は興味深いものではありません。これらのセクタを読み書きできるようにしたいので、今見ているようにディスクアームにディスクヘッドが必要です (図 37.2)。図では、アームの端部に取り付けられたディスクヘッドがセクター 6 の上に配置され、表面が反時計回りに回転しています。

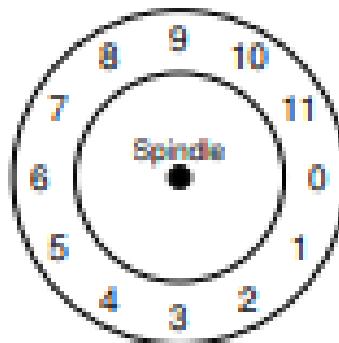


Figure 37.1: A Disk With Just A Single Track

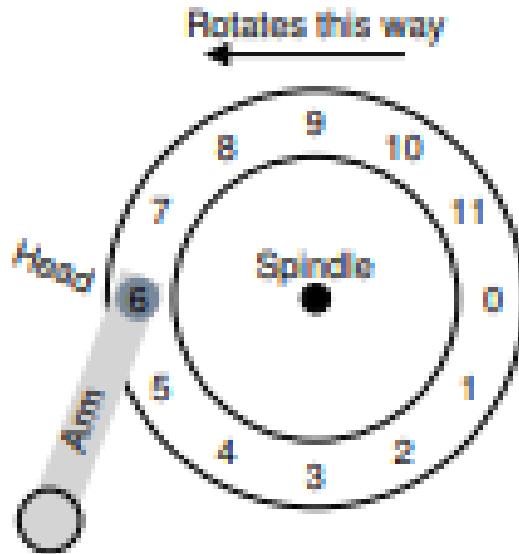


Figure 37.2: A Single Track Plus A Head

Single-track Latency: The Rotational Delay

単純な onetrack ディスク上でリクエストがどのように処理されるかを理解するために、今ブロック 0 の読み込み要求を受け取ったとします。ディスクサービスはどのようにこの要求を処理する必要がありますか？

私たちのシンプルなディスクでは、ディスクはそれほど多くする必要はありません。特に、ディスクヘッドの下で望んだセクタが回転するのを待たなければならない。この待ち時間は、現代のドライブではよく起りますが、I/O サービス時間の重要な要素です。rotational delay(回転遅延)(ときどき回転遅延、それは奇妙に聞こえるかもしれません) という特殊な名前があります。この例では、完全回転遅延が R である場合、ディスクは、読み書きヘッドがセクタ 0 で待っていた場合(もしセクタ 6 で開始すると)、平均で $R/2$ の回転遅延を生じます。この単一のトラックに対する最悪の場合の要求は、セクタ 5 へのものであり、そのような要求に対応するためにほぼ完全回転遅延を引き起こします。

Multiple Tracks: Seek Time

これまでのところ、私たちのディスクには単一のトラックしかありません。これはあまり現実的ではありません。現代のディスクには数百万ものものがあります。このようにして、少しだけ現実的なディスク面を見てみましょう。このディスク面は 3 つのトラックで構成されています(図 37.3)

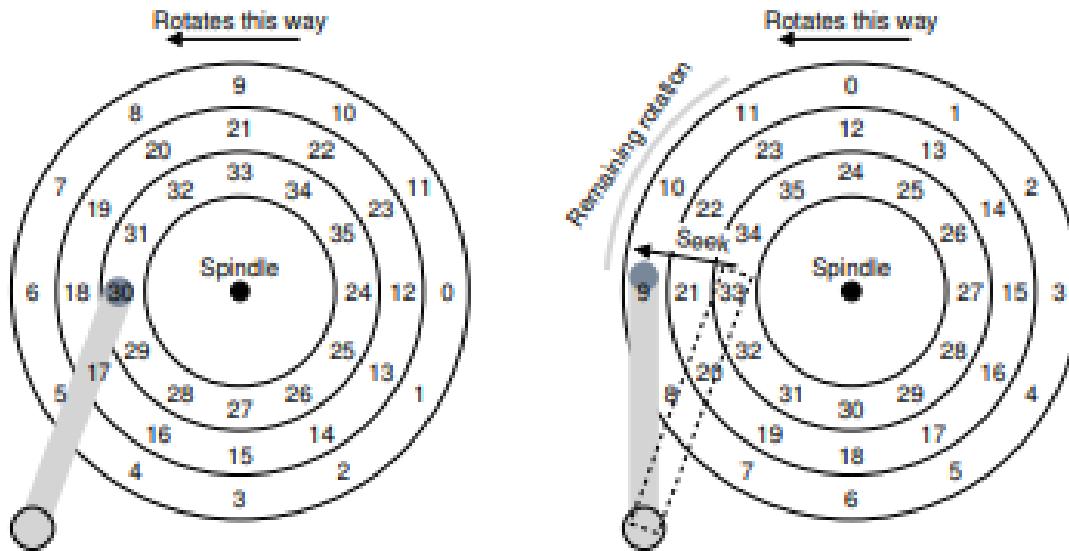


Figure 37.3: Three Tracks Plus A Head (Right: With Seek)

図では、ヘッドは現在、最も内側のトラック（セクタ 24~35 を含む）上に配置されている。次のトラックは次のセクタセット（12~23）を含み、最外側トラックは第 1 のセクタ（0~11）を含みます。

ドライブがどのセクタにアクセスする可能性があるかを理解するために、セクタ 11 への読み込みなど、離れたセクタへの要求で何が起きるかを追跡します。この読み込みを処理するには、まずドライブをディスクアームを正しいトラック（この場合、最外側のトラック）に移動させるプロセスを行います。このプロセスシーク（seek）と呼びます。シークは回転と一緒に、最もコストのかかるディスク操作の 1 つです。

シークは注目すべきですが、多くのフェーズがあります。最初にディスクアームが動くように加速フェーズを行います。腕が最高速度で動いているときに惰性走行し、次に腕が減速すると減速します。最終的にはヘッドは正しいトラック上に慎重に配置されると settling(沈む) します。ドライブが正しいトラックを見つけることが確実でなければならないので、settling time(沈む時間) は非常に重要であり、例えば、0.5~2ms となることが多いです。

シークの後、ディスクアームはヘッドを正しいトラックの上に配置します。シークの描写は図 37.3 にあります。

見てわかるように、シーク中にアームが所望のトラックに移動し、プラッタが回転しています。この場合は約 3 セクタである。したがって、セクタ 9 はディスクヘッドの下をちょうど通過しようとしており、転送を完了するために短い回転遅延に耐える必要があります。

セクタ 11 がディスクヘッドの下を通過すると、転送と呼ばれる I/O の最終フェーズが実行され、データの読み書きが行われます。したがって、I/O 時間の完全な図が得られます。最初にシークし、次に回転遅延を待ってから最後に転送します。

Some Other Details

あまり時間を費やすことはありませんが、ハードドライブの動作に関する他の面白い詳細があります。多くのドライブでは、トラックの境界を越えても順次読み込みが適切に処理できるように、何らかのトラックスキューを使用しています。私たちの単純な例のディスクでは、これは図 37.4 のように見えるかもしれません。

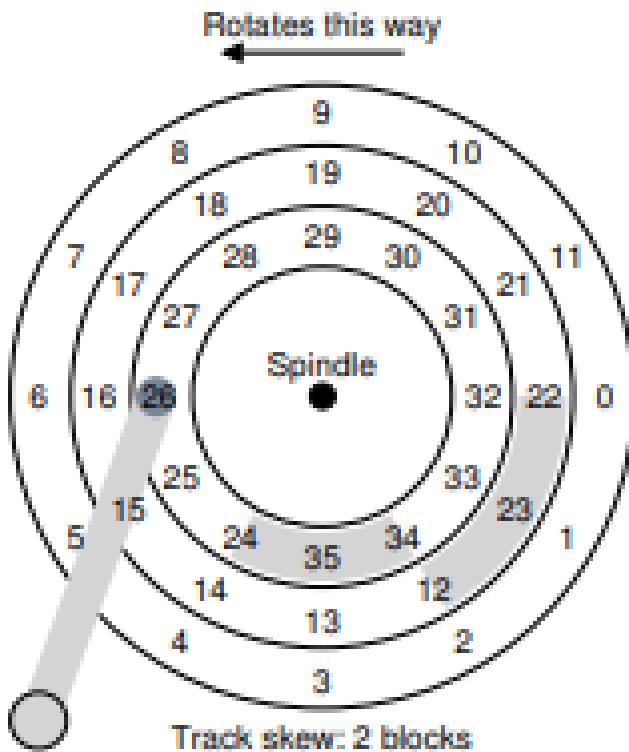


Figure 37.4: Three Tracks: Track Skew Of 2

あるトラックから別のトラックに切り替えると、ディスクはヘッドを再配置する時間が必要となるため（隣接するトラックまで）、セクタはこのように歪んでいることがよくあります。このようなスキー（歪み）がなければ、ヘッドは次のトラックに移動されるが、望んだセクタの次のブロックはヘッドの下で既に回転しているため、ドライブは次のブロックにアクセスするためにほぼ全回転遅延を待たなければいけません。

もう一つの現実は、アウタートラックは、ジオメトリの結果であるインナートラックより多くのセクタを持つ傾向があります。そこには単に余裕があります。これらのトラックは、ディスクが複数のゾーンに編成され、ゾーンがサーフェス上のトラックの連続したセットであるマルチゾーンのディスクドライブとしてよく使用されます。各ゾーンは 1 トラックあたりのセクタ数が同じであり、外側ゾーンは内側ゾーンよりも多くのセクタを持ちます。

最後に、最近のディスクドライブの重要な部分は、歴史的な理由からトラックバッファと呼ばれるキャッシュです。このキャッシュは、ドライブがディスクから読み書きするデータを保持するために使用できるわずかな量のメモリ（通常は約 8 または 16 MB）です。たとえば、ディスクからセクタを読み取る場合、ドライブはそのトラックのすべてのセクタを読み込み、そのセクタをそのメモリにキャッシュすることになります。これにより、ドライブは、同じトラックへの後続の要求にすばやく応答することができます。

書き込みの場合、ドライブには選択肢があります。データがメモリに格納されたとき、または書き込みが実際にディスクに書き込まれた後に、書き込みが完了したことを確認する必要がありますか？ 前者はライトバックキャッシング（または時には即時報告）と呼ばれ、後者はライトスルーと呼ばれています。ライトバックキャッシングを行うと、ドライブが「高速」に見えることがあります（危険です）。ファイルシステムまたはアプリケーションで、正しい順序でデータをディスクに書き込む必要がある場合は、ライトバックキャッシングで問題が発生する可能性があります（詳細は、ファイルシステムのジャーナリングに関する章を参照してください）。

ASIDE: DIMENSIONAL ANALYSIS

化学の授業では、単位を単純に設定し打ち消しあい、結果として何か答えをだしました。どのようにしたかを覚えていますか？その化学的な魔法は、高次元ファクター解析の名で知られており、それはコンピュータシステム分析にも有用であることが判明しています。

次元分析がどのように機能するのか、なぜそれが有用なのかを見てみましょう。この場合、ディスクの1回転にかかる時間(ミリ秒)を把握しておく必要があると仮定します。残念ながら、ディスクのRPM、または1分あたりの回転のみが与えられます。10K RPMディスク(つまり、1分間に10,000回回転するディスク)を想定しているとします。1回転あたりの時間をミリ秒単位で取得できるようにしますか？

これを行うには、まず、目的のユニットを左に置きます。この場合、1回転あたりの時間(ミリ秒)を取得したいので、正確には $(\text{Time(ms)}) / (1 \text{ Rotation})$ を書きます。私たちが知っているすべてを書き、できるだけユニットを取り消してください。最初に、 $(1 \text{ 分}) / (10,000 \text{ 回転})$ (回転を下にし、それが左にあるように)、分を $(60 \text{ 秒} / 1 \text{ 分})$ 秒に変換し、最後に $(1000 \text{ ms}) / (1 \text{ 秒})$ でミリ秒単位の秒数に変換します。最終的な結果は以下の通りです(ユニットは素早くキャンセルされています)。

$$\frac{\text{Time (ms)}}{1 \text{ Rot.}} = \frac{1 \text{ minute}}{10,000 \text{ Rot.}} \cdot \frac{60 \text{ seconds}}{1 \text{ minute}} \cdot \frac{1000 \text{ ms}}{1 \text{ second}} = \frac{60,000 \text{ ms}}{10,000 \text{ Rot.}} = \frac{6 \text{ ms}}{\text{Rotation}}$$

この例からわかるように、次元分析は、直感的に見えるものを単純で反復可能なプロセスにします。上記のRPM計算以外にも、定期的にI/O分析を行うことができます。たとえば、ディスクの転送速度(100 MB/秒など)が与えられることがあります。次に、512 KBブロック(ミリ秒単位)の転送にはどのくらいの時間がかかりますか？次元分析を使用すると簡単です。

$$\frac{\text{Time (ms)}}{1 \text{ Request}} = \frac{512 \text{ KB}}{1 \text{ Request}} \cdot \frac{1 \text{ MB}}{1024 \text{ KB}} \cdot \frac{1 \text{ second}}{100 \text{ MB}} \cdot \frac{1000 \text{ ms}}{1 \text{ second}} = \frac{5 \text{ ms}}{\text{Request}}$$

37.4 I/O Time: Doing The Math

ディスクの抽象モデルが完成したので、ディスクのパフォーマンスをよりよく理解するために、少し分析することができます。特に、I/O時間を3つの主要コンポーネントの合計として表すことができるようになりました。

$$T_{I/O} = T_{seek} + T_{rotation} + T_{transfer} \quad (37.1)$$

ドライブ間の比較のためにしばしばより容易に使用されるI/O(R.I/O)の割合(以下で説明する)は、時間から容易に計算されることを覚えておいてください。単純に転送のサイズをそれにかかる時間で割ってください：

	Cheetah 15K.5	Barracuda
Capacity	300 GB	1 TB
RPM	15,000	7,200
Average Seek	4 ms	9 ms
Max Transfer	125 MB/s	105 MB/s
Platters	4	4
Cache	16 MB	16/32 MB
Connects via	SCSI	SATA

Figure 37.5: Disk Drive Specs: SCSI Versus SATA

I/O 時間をより良く感じるには、次の計算を実行してみましょう。私たちが興味を持っている 2 つの仕事量があると仮定します。最初のものは、ランダム仕事量として知られ、ディスク上のランダムな場所に小さな(たとえば 4KB) 読み取りを発行します。ランダム仕事量は、データベース管理システムを含む多くの重要なアプリケーションで一般的です。順次仕事量と呼ばれる第 2 のものは、ジャンプすることなく、ディスクから連続して多数のセクタを読み取るだけです。順次アクセスパターンは非常に一般的であり、したがって同様に重要です。

ランダム仕事量と順次仕事量のパフォーマンスの違いを理解するには、まずディスクドライブについていくつかの仮定を立てる必要があります。Seagate の最新のディスクをいくつか見てみましょう。Cheetah 15K.5 [S09b] と呼ばれる最初のものは、高性能 SCSI ドライブです。2 番目のバラクーダ [S09a] は、容量のために作られたドライブです。両方の詳細は図 37.5 にあります。

$$R_{I/O} = \frac{Size_{Transfer}}{T_{I/O}} \quad (37.2)$$

ご覧のように、ドライブは全く異なる特性を持ち、多くの点でディスクドライブ市場の 2 つの重要なコンポーネントを素早く要約しています。1 つは、ドライブができるだけ速く回転し、シーク時間が短く、データをすばやく転送できるように設計された「高性能」ドライブ市場です。2 番目は「容量」市場で、1 バイトあたりのコストが最も重要な側面です。したがって、ドライブはより低速ですが、使用可能なスペースにできるだけ多くのビットをパックします。

これらの数字から、上に概説した 2 つの仕事量でドライブがどれくらいかを計算することができます。ランダムな仕事量を見てみましょう。ディスク上のランダムな位置で各 4 KB の読み取りが発生すると仮定して、このような読み取りがどれぐらいかかるかを計算することができます。

$$T_{seek} = 4 \text{ ms}, T_{rotation} = 2 \text{ ms}, T_{transfer} = 30 \text{ microsecs} \quad (37.3)$$

TIP: USE DISKS SEQUENTIALLY

可能な限り、シーケンシャルな方法でディスクとの間でデータを転送します。シーケンシャルが不可能な場合は、少なくとも大量のチャンクでデータを転送することを検討してください。I/O がランダムではない場合、I/O パフォーマンスは劇的に低下します。また、ユーザーは苦します。また、不注意なランダム I/O で何が苦しんでいるのかを知ることで、苦します。

平均シーク時間(4ミリ秒)は、製造元によって報告された平均時間とみなされます。表面の一方の端から他方への完全なシークは2~3倍長くかかる可能性があることに注意してください。平均回転遅延はRPMから直接計算されます。15000 RPMは250 RPS(1秒あたりの回転数)に等しい。従って、各回転は4msを要する。平均して、ディスクは半回転するため、平均時間は2msです。最後に、転送時間は、ピーク転送レートでの転送サイズにすぎません。ここではそれほど小さくはありません(30マイクロ秒です、1ミリ秒は1000マイクロ秒です)。

したがって、上記の方程式から、チーターのT.I/Oはおよそ6ミリ秒に等しい。I/Oの速度を計算するには、転送のサイズを平均時間で除算するだけで、約0.66 MB/sのランダムな仕事量でチーターのR.I/Oに到着します。バラクーダと同じ計算では、T.I/Oは約13.2ミリ秒、2倍以上の遅さ、したがって約0.31 MB/sの速度をもたらします。

次に、順次仕事量を見てみましょう。ここでは、非常に長い転送の前に単一のシークとローテーションがあると仮定できます。簡単にするために、転送のサイズが100 MBであると仮定します。したがって、BarracudaとCheetahのT.I/Oはそれぞれ約800 msと950 msです。従って、I/Oのレートは、それぞれ125 MB/s及び105 MB/sのピーク転送速度に非常に近い。図37.6にこれらの数値をまとめました。

	Cheetah	Barracuda
R.I/O Random	0.66 MB/s	0.31 MB/s
R.I/O Sequential	125 MB/s	105 MB/s

Figure 37.6: Disk Drive Performance: SCSI Versus SATA

図は私たちに多くの重要なことを示しています。まず第一に、最も重要なのは、ランダムとシーケンシャルの仕事量の間には、チーターの場合はほぼ200倍、バラクーダの場合は300倍以上の差があることです。そして、我々はコンピューティングの歴史の中で最も明白なデザインのヒントに到達します。

もう1つ、より微妙な点であるハイエンドの「パフォーマンス」ドライブとローエンドの「容量」ドライブのパフォーマンスには大きな違いがあります。この理由(およびその他)のために、人々は後者をできるだけ安く手に入れようとしながら、よく前者のようなパフォーマンスを望んでいます。

ASIDE: COMPUTING THE “AVERAGE” SEEK

多くの書籍や論文では、平均シーク時間の約3分の1のディスクシーク時間が表示されます。これはどこから来たのですか?

それは、時間ではなく平均シーク距離に基づく単純な計算から生じることが分かります。ディスクを0からNまでのトラックのセットとして想像してください。したがって、任意の2つのトラックxとyの間のシーク距離は、|x - y|の差の絶対値として計算されます。

平均シーク距離を計算するには、まずシーク距離をすべて加算するだけです。

$$\sum_{x=0}^N \sum_{y=0}^N |x - y|. \quad (37.4)$$

次に、これを可能なシークの数N^2で割ります。合計を計算するには、整数型を使用します

$$\int_{x=0}^N \int_{y=0}^N |x - y| dy dx. \quad (37.5)$$

内部積分を計算するには、絶対値を分解しましょう

$$\int_{y=0}^x (x - y) dy + \int_{y=x}^N (y - x) dy. \quad (37.6)$$

$$(xy - \frac{1}{2}y^2) \Big|_0^x + (\frac{1}{2}y^2 - xy) \Big|_x^N$$

これを解決すると、

$$(x^2 - Nx + \frac{1}{2}N^2)$$

に簡略化でき、

が生成されます。今度は、外積分を計算しなければなりません

$$\int_{x=0}^N (x^2 - Nx + \frac{1}{2}N^2) dx, \quad (37.7)$$

which results in:

$$(\frac{1}{3}x^3 - \frac{N}{2}x^2 + \frac{1}{2}N^2 x) \Big|_0^N = \frac{N^3}{3}. \quad (37.8)$$

平均シーク距離を計算するには、シークの総数 (N^2) で除算する必要があります：

$$(\frac{N^3}{3}) / (N^2) = \frac{1}{3}N.$$

したがって、ディスク上の平均シーク距離は、すべての可能なシークにわたって、全距離の 3 分の 1 です。そして今、平均的なシークが完全なシークの 3 分の 1 であると聞くと、それはどこから来たのかを知るでしょう。

37.5 Disk Scheduling

I / O のコストが高いため、OS は歴史的にディスクに発行された I/O の順序を決定する役割を果たしました。より具体的には、ディスクスケジューラは、一連の I/O 要求があれば、その要求を検査し、次にスケジューリングするものを決定します [SCO90、JW91]。

ジョブスケジューリングとは異なり、各ジョブの長さは通常不明ですが、ディスクスケジューリングでは、「ジョブ」(ディスク要求) の所要時間を推測することができます。要求のシークと可能な回転遅延を見積ることにより、ディスクスケジューラは、各要求がどれくらいの時間かかるかを知ることができ、最初にサービスする時間が最も短いものを選びます(貪欲に)。したがって、ディスクスケジューラは、その操作において SJF(shortest job first) の原則に従おうとします。

SSTF: Shortest Seek Time First

早期ディスクスケジューリングアプローチの 1 つは、最短シークタイムファースト (SSTF)(最短シークファーストまたは SSF とも呼ばれる) として知られている。SSTF は、I/O 要求のキューをトラックごとに順序付けし、最寄りのトラックで要求をピックして最初に完了します。例えば、ヘッドの現在位置が内側トラック上にあり、セクタ 21(ミドルトラック) と 2(外側トラック) に対する要求があると仮定すると、最初に 21 へリクエストを発行し、その完了を待ってから 2 を要求するように発行します(図 37.7)。

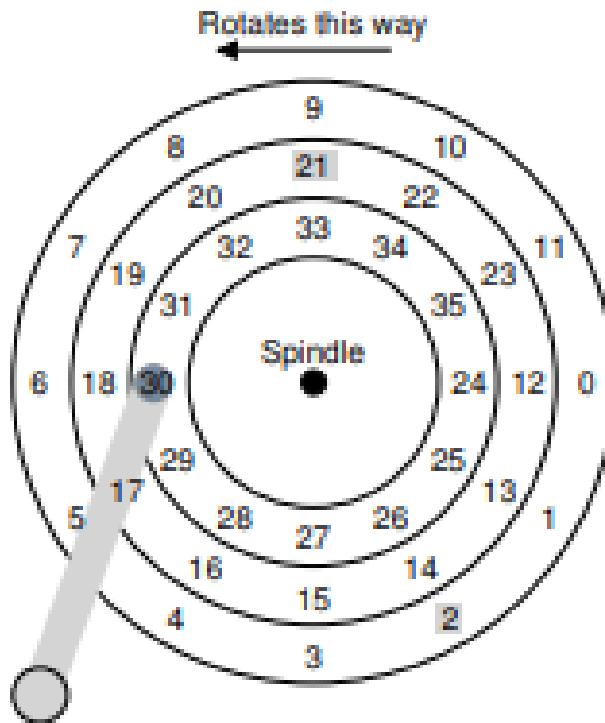


Figure 37.7: SSTF: Scheduling Requests 21 And 2

SSTF はこの例ではうまく機能し、最初はミドルトラック、次にアウタートラックを探します。しかし、SSTF は万能ではないです。まず、ドライブのジオメトリはホスト OS で使用できません。むしろ、それはブロックの配列を見ます。幸いにも、この問題はかなり簡単に修正されています。OS は、SSTF の代わりに、次の最も近いブロックアドレスを持つ要求をスケジューリングする nearest-block-first(NBF) を実装するだけです。

済みます。

第2の問題はより根本的です。上記の例では、ヘッドが現在位置している内側のトラックへの要求が安定しているとします。他のトラックへのリクエストは、純粋な SSTF アプローチによって完全に無視されます。したがって、問題の要点は次のとおりです。

CRUX: HOW TO HANDLE DISK STARVATION

どのようにして SSTF のような、しかし、飢餓を避けるようなスケジューリングを実装できますか？

Elevator (a.k.a. SCAN or C-SCAN)

このクエリーへの答えは、しばらく前に開発されました (たとえば [CKR72] を参照)。これは比較的簡単です。もともと SCAN と呼ばれていたこのアルゴリズムは、トラックを横切って順番に要求を処理するディスクを横切って前後に移動するだけです。ディスクを横切る単一のパス (外側から内側トラック、または内側から外側) をスイープと呼ぶことにしましょう。したがって、ディスクのこのスイープですでに処理されているトラックのブロックに対する要求が発生すると、すぐには処理されず、次のスイープまで (他の方向の) キューに入れられます。

SCAN には多数の変種があり、そのすべてがほぼ同じことをしています。例えば、Coffman et al. スイープを実行しているときに処理されるキューをフリーズする F-SCAN が導入されました [CKR72]。このアクションによって、スイープ中にに入った要求がキューに入れられ、後で処理されます。そうすることで、遅く到着する (しかしより近い) 要求のサービスを遅らせることによって、遠く離れた要求の枯渇を回避することができます。

C-SCAN は Circular SCAN の略語でもあります。ディスク上の両方向にスイープするのではなく、アルゴリズムは外側から内側へのみスイープし、外側のトラックでリセットして再び開始します。これは、純粋なバック・アンド・フォワード・スキヤンは中間トラックに有利です、すなわち外側トラックを処理した後、SCAN が再び外側トラックに戻る前に 2 回真ん中を通過するので、内側トラックおよび外側トラックに対して少し公平である。

SCAN アルゴリズム (およびその従兄弟) は、エレベータアルゴリズムと呼ばれることがあります。エレベータは、エレベータのように動作します。エレベータは、上または下に移動し、どちらのフロアが近いかというベースでは動きません。ここで、フロアのどちらが近いかというアルゴリズムで動いていると想像してみてください。あなたがもし 10 階からエレベーターに載って 1 階へ行っているとき、途中で誰かが 3 階から乗つて 4 階へ行くためにボタンを押したとき、エレベーターは現時点で 1 階より 4 階が近いので、エレベーターが 4 階にあがります。どれほど迷惑なことでしょう。ご覧のように、エレベーターのアルゴリズムは実際に使用すると、さきほどのような迷惑な行為を防ぎます。ディスクでは、単に飢餓を防ぎます。

残念なことに、SCAN とその従兄弟は最良のスケジューリング技術ではありません。特に、SCAN(または SSTF even) は、実際には SJF の原理に厳密に準拠していません。特に、それらは回転を無視します。

CRUX: HOW TO ACCOUNT FOR DISK ROTATION COSTS

シークと回転の両方を考慮に入れて、SJF にさらに近似したアルゴリズムを実装するにはどうすればよいですか？

SPTF: Shortest Positioning Time First

shortest positioning time first(最短の位置時間)、問題の解決策である SPTF スケジューリング (時には shortest access time first(最短アクセス時間) とも呼ばれる) を検討する前に、問題をより詳細に理解してください。図 37.8 に例を示します。

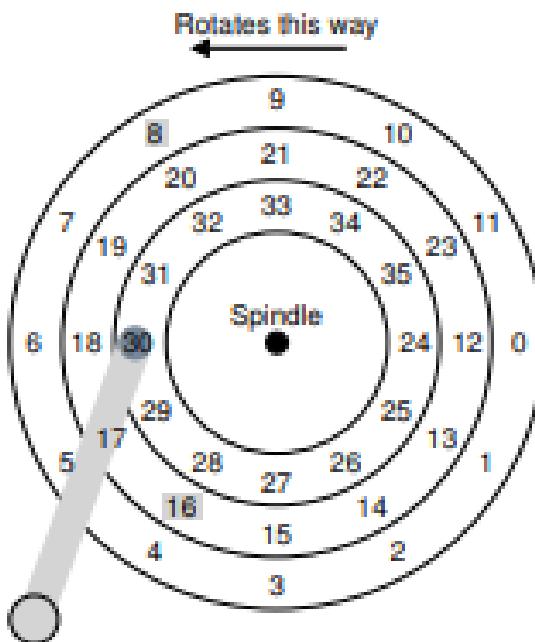


Figure 37.8: SSTF: Sometimes Not Good Enough

この例では、ヘッドは、現在、内側トラック上のセクタ 30 上に配置されています。したがって、スケジューラは、次の要求に対してセクタ 16(中間トラック上) またはセクタ 8(外側トラック上) をスケジューリングすべきかどうかを決定しなければいけません。そのため、どのサービスを次にすればよいでしょうか？

答えは、もちろん、「それは依存している」です。エンジニアリングでは、「それは依存している」ことがほとんど常に答えです。トレードオフはエンジニアの生活の一部です。そのような格言は、ピンチでも良いです。たとえば、あなたの上司の質問に対する答えがわからないときは、この宝石を試してみてください。しかし、なぜそれが依存するのかを知ることは、ほとんどいつもより良いことです。それはここで議論するものです。

ここに依存するのは、回転と比較したシークの相対的な時間です。この例では、シーク時間が回転遅延よりもはるかに高い場合、SSTF(およびバリエント) は正常です。しかし、シークがローテーションよりも速いと想像してください。次に、この例では、外側のトラックでサービス要求 8 を求めるよりも、サービス 16 の中間トラックをシークするのがより短くなります。これは、ディスクヘッドの下を通過する前に全周にわたって回転しなければいけないからです。

現代のドライブでは、上記のように、シークとローテーションの両方がほぼ同等です(もちろん、正確な要求にもよるが)。したがって、SPTF は有用であり、パフォーマンスを向上させます。しかし、OS で実装することはさらに難しく、ディスク境界がどこにあるか、またはディスクヘッドが現在(回転の意味で) どこにあるかはよく分かっていません。したがって、SPTF は通常、以下で説明するドライブ内で実行されます。

TIP: IT ALWAYS DEPENDS (LIVNY'S LAW)

私たちの同僚である Miron Livny はいつも言っているように、ほとんどの質問は「それは依存している」と答えることができます。しかし、あまりにも多くの質問にこのように答えると、人々はあなたに質問をするのをやめます。たとえば、誰かが「昼食に行きたいですか？」と尋ねると、あなたは「それは、あなたがくるのかによる」というでしょう。

Other Scheduling Issues

この基本的なディスク操作、スケジューリング、および関連するトピックのこの簡単な説明では、他にも多くの問題がありますが、議論しませんでした。そのような問題の1つは、最新のシステムでディスクスケジューリングを実行する場所はどこですか？ということです。古いシステムでは、オペレーティングシステムはすべてスケジューリングを行いました。保留中の要求のセットを調べた後、OSは最良のものを選択し、ディスクに発行します。その要求が完了すると、次の要求が選択されます。ディスクは単純でした。

現代のシステムでは、ディスクは複数の未解決の要求に対応でき、高度な内部スケジューラ自体を持っています (SPTF を正確に実装できる、ディスクコントローラ内では正確なヘッド位置を含む関連するすべての詳細が利用可能です)。したがって、OSスケジューラは、通常、いくつかの要求が最良であると思うもの(たとえば 16)を選択し、すべてをディスクに発行します。ディスクは、詳細な内部情報のヘッド位置および詳細なトラックレイアウト情報を使用して、可能な限り最善の順序 (SPTF) で前記要求を処理します。

ディスクスケジューラによって実行されるもう1つの重要な関連タスクは、I/O マージです。たとえば、図 37.8 のように、ブロック 33、次に 8、そして 34 を読み込む一連の要求を想像してみてください。この場合、スケジューラは、ブロック 33 および 34 の要求を单一の 2 ブロック要求にマージする必要があります。スケジューラが行うリオーダリングは、マージされた要求に対して実行されます。マージは、ディスクに送信される要求の数を減らし、オーバーヘッドを減らすため、OS レベルで特に重要です。

現代のスケジューラが直面する最後の問題の1つは、システムがディスクへの I/O を発行するまでにどれくらいの時間待つべきかということです。ディスクが一度 I/O があっても、すぐにドライブに要求を発行する必要があると考えるかもしれません。このアプローチは、work-conserving(作業を保存する)と呼ばれます。これは、処理する必要がある場合、ディスクは決してアイドル状態にならないためです。しかし、先行ディスクスケジューリングに関する研究 [ID01] では、非作業保存アプローチと呼ばれるもので、待つためのビットがある方が良いことが示されています。

待つことによって、新しい“より良い”要求がディスクに到着する可能性があるため、全体的な効率が向上します。もちろん、いつ待つか、どのくらい長くするかを決めるのは難しいことです。詳細については研究論文を参照するか、Linux カーネルの実装をチェックして、そのようなアイディアがどのように実践されているかを見てみましょう(あなたが野心的な場合)

37.6 Summary

ディスクの仕組みの概要を示しました。要約は実際には詳細な機能モデルです。それは実際のドライブ設計に入る驚くべき物理学、エレクトロニクス、物質科学を記述するものではありません。しかし、その性質の詳細についてもっと興味を持っている人には、別のメジャー(またはマイナーな)な分野を提案します。これらの別のメジャーの分野のモデルに興味を持っているのであればそれは良かったです。このモデルを使用して、これらの素晴らしいデバイスの上にさらに興味深いシステムを構築することができます。

参考文献

- [ADR03] “More Than an Interface: SCSI vs. ATA”
 - Dave Anderson, Jim Dykes, Erik Riedel
 - FAST '03, 2003
 - One of the best recent-ish references on how modern disk drives really work; a must read for anyone interested in knowing more.
- [CKR72] “Analysis of Scanning Policies for Reducing Disk Seek Times”
 - E.G. Coffman, L.A. Klimko, B. Ryan

SIAM Journal of Computing, September 1972, Vol 1. No 3.

Some of the early work in the field of disk scheduling.

[ID01] "Anticipatory Scheduling: A Disk-scheduling Framework To Overcome Deceptive Idleness In Synchronous I/O"

Sitaram Iyer, Peter Druschel

SOSP '01, October 2001

A cool paper showing how waiting can improve disk scheduling: better requests may be on their way!

[JW91] "Disk Scheduling Algorithms Based On Rotational Position"

D. Jacobson, J. Wilkes

Technical Report HPL-CSP-91-7rev1, Hewlett-Packard (February 1991)

A more modern take on disk scheduling. It remains a technical report (and not a published paper) because the authors were scooped by Seltzer et al. [SCO90].

[RW92] "An Introduction to Disk Drive Modeling"

C. Ruemmler, J. Wilkes

IEEE Computer, 27:3, pp. 17-28, March 1994

A terrific introduction to the basics of disk operation. Some pieces are out of date, but most of the basics remain.

[SCO90] "Disk Scheduling Revisited"

Margo Seltzer, Peter Chen, John Ousterhout

USENIX 1990

A paper that talks about how rotation matters too in the world of disk scheduling.

[SG04] "MEMS-based storage devices and standard disk interfaces:A square peg in a round hole?"

Steven W. Schlosser, Gregory R. Ganger

FAST '04, pp. 87-100, 2004

While the MEMS aspect of this paper hasn't yet made an impact, the discussion of the contract between file systems and disks is wonderful and a lasting contribution.

[S09a] "Barracuda ES.2 data sheet"

http://www.seagate.com/docs/pdf/datasheet/disc/ds_barracuda_es.pdf

A data sheet; read at your own risk. Risk of what? Boredom.

[S09b] "Cheetah 15K.5"

<http://www.seagate.com/docs/pdf/datasheet/disc/ds-cheetah-15k-5-us.pdf>

See above commentary on data sheets.

38 Redundant Arrays of Inexpensive Disks (RAIDs)

ディスクを使用する場合、ディスクを高速化したい場合があります。I/O 操作が遅いため、システム全体のボトルネックになる可能性があります。ディスクを使用する場合、ディスクを大きくしたい場合があります。多くのデータをオンラインに置かなければ、ローカルディスクがデータでいっぱいになってしまいます。ディスクを使用する場合、ディスクをより信頼できるものにすることを望みます。ディスクに障害が発生した場合、データがバックアップされていなければ、その重要なデータはすべて失われます。

CRUX: HOW TO MAKE A LARGE, FAST, RELIABLE DISK

大規模で高速で信頼性の高いストレージシステムを構築するにはどうすればよいですか？ 重要な技術は何ですか？ 異なるアプローチ間のトレードオフは何ですか？

この章では、RAID [P + 88] というよりよく知られている安価なディスクの冗長配列を紹介します。これは、複数のディスクを一度に使用して、より速く、より大きく、より信頼性の高いディスクシステムを構築するテクニックです。この用語は、1980 年代後半に米国バークレーの研究者グループによって導入されました。(David Patterson と Randy Katz 教授、そして Garth Gibson 教授が率いる) 多くの異なる研究者が複数のディスクを使用してより優れたストレージシステムを構築するという基本的な考え方同時に到達したのはこの頃でした [BG88、K86、K88、PB86、SG86]。

外部的には、RAID はディスクのように見えます。つまり、読み書きできるブロックのグループです。内部的には、RAID は複数のディスク、メモリ (揮発性と非揮発性の両方)、システムを管理する 1 つ以上のプロセッサで構成される複雑な獣です。ハードウェア RAID は、ディスクグループを管理するタスクに特化したコンピュータシステムに非常によく似ています。

RAID は、1 つのディスクに比べて多くの利点を提供します。1 つの利点はパフォーマンスです。並列に複数のディスクを使用すると、I/O 時間が大幅に短縮されます。もう 1 つの利点は容量です。大きなデータセットには大きなディスクが必要です。最後に、RAID は信頼性向上させることができます。複数のディスクにデータを分散すると (RAID 技術なし)、データは单一ディスクが壊れると消失するという脆弱に繋がります。何らかの形の冗長性では、RAID はディスクの損失を許容し、何も問題がないかのように動作し続けることができます。

TIP: TRANSPARENCY ENABLES DEPLOYMENT

システムに新しい機能を追加する方法を検討するときは、システムの残りの部分に変更を加える必要がないように、そのような機能を透過的に追加できるかどうかを常に検討する必要があります。既存のソフトウェア (または過激なハードウェア変更) の完全な書き換えを要求することは、アイデアのすばらしい可能性がなくなってしまいます。RAID は完璧な例であり、確かにその透過性は成功に貢献しました。管理者は SCSI ディスクの代わりに SCSI ベースの RAID ストレージ配列をインストールでき、残りのシステム (ホストコンピュータ、OS など) は 1 ビットも変更する必要がありませんでした。この展開の問題を解決することにより、RAID は最初からすばらしい成功を収めました。

驚いたことに、RAID は、これらの利点を使用するシステムにこれらの利点を透過的に提供します。つまり、RAID はホストシステムにとって大きなディスクのように見えます。透明性の美しさはもちろん、ディスクを RAID に置き換えるだけで、ソフトウェアの 1 行を変更する必要はありません。オペレーティング・システムおよびクライアント・アプリケーションは変更せずに動作し続けます。このように、透過性は RAID の展開性を大幅に向上させ、ユーザと管理者がソフトウェア互換性の心配なしに RAID を使用できるようにします。

ここで、RAID の重要な側面のいくつかについて説明します。インターフェイス、障害モデルから始めて、

容量、信頼性、パフォーマンスの3つの重要な軸に沿って RAID 設計を評価する方法について説明します。次に、RAID の設計と実装にとって重要なその他の多くの問題について説明します。

38.1 Interface And RAID Internals

上のファイルシステムでは、RAID は大きくて、うまくいけば高速で、(うまくいけば) 信頼できるディスクのように見えます。単一のディスクの場合と同様に、ファイルシステム (または他のクライアント) によって読み書きできるブロックの線形配列として表示されます。

ファイルシステムが RAID に論理 I/O 要求を発行すると、RAID は内部要求を完了するためにアクセスしたディスク (またはディスク) を計算しなければいけません、そしてそうするために、1つまたは複数の物理 I/O を発行します。これらの物理 I/O の正確な性質は、RAID レベルによって異なります (詳細は後述)。しかし、単純な例として、各ブロック (それぞれが別個のディスクにある) の2つのコピーを保持する RAID を考えてみましょう。このようなミラー化された RAID システムに書き込む場合、RAID は発行される1つの論理 I/O ごとに2つの物理 I/O を実行する必要があります。

RAID システムは、よくホストへの標準的な接続 (例えば、SCSI または SATA) を有する別個のハードウェアボックスとして構築されます。しかし、内部的に RAID はかなり複雑で、RAID の動作を指示するファームウェアを実行するマイクロコントローラ、DRAM のような揮発性メモリが読み書きされたデータブロックをバッファリングし、場合によっては不揮発性メモリがバッファに安全な書き込みや特殊なロジックにも、パーティ計算を実行するかもしれません (いくつかの RAID レベルで有効です、以下も参照してください)。高レベルでは、RAID は特殊なコンピュータシステムです。プロセッサ、メモリ、およびディスクを備えています。ただし、アプリケーションを実行する代わりに、RAID を動作させるように設計された専用ソフトウェアを実行します。

38.2 Fault Model

RAID を理解し、さまざまなアプローチを比較するには、障害モデルを考慮する必要があります。RAID は、特定の種類のディスク障害を検出してリカバリするように設計されています。したがって、どのような不具合が予期されるかを正確に知ることは、実際の設計に着手する上で重要です。

我々が想定する最初の故障モデルは非常に単純であり、fail stop fault model (フェイルストップフォールトモデル) と呼ばれています [S84]。このモデルでは、ディスクは正確に2つの状態のうちの1つになる可能性があります。二つの状態というのは作業中と故障です。作業ディスクを使用すると、すべてのブロックを読み書きすることができます。対照的に、ディスクに障害が発生した場合、ディスクは永久に失われたとみなされます。

フェイルストップモデルの重要な側面の1つは、障害検出について想定していることです。具体的には、ディスクが故障した場合、これは容易に検出されると想定します。たとえば、RAID 配列では、RAID コントローラのハードウェア (またはソフトウェア) が、ディスクに障害が発生したときに直ちに監視することができます。

したがって、今のところ、ディスクの破損などのより複雑な「サイレント」エラーについては心配する必要はありません。また、別の方法で動作しているディスク (潜在的なセクタエラーと呼ばれることがあります) では、単一のブロックがアクセス不能になることを心配する必要はありません。私たちは後に、より複雑な (そして残念なことにより現実的な) ディスク障害を考えていきます。

38.3 How To Evaluate A RAID

すぐにわかるように、RAID を構築するにはいくつかの異なるアプローチがあります。これらのアプローチのそれぞれは、その強みと弱みを理解するために、評価する価値のあるさまざまな特性を持っています。

具体的には、各 RAID 設計を 3 つの軸に沿って評価します。最初の軸は容量です。それぞれ B ブロックを持つ N 個のディスクのセットが与えられた場合、RAID のクライアントはどれくらいの有用な容量を利用できますか？冗長性がなければ、答えは $N \cdot B$ です。対照的に、各ブロックの 2 つのコピーを保持するシステム（ミラーリングと呼ばれる）があれば、 $(N \cdot B)/2$ の容量が得られます。また、異なるスキーム（例えば、パリティベースのもの）がその間にいることがあります。

評価の第 2 の軸は信頼性です。特定の設計で許容されるディスク障害の数はいくつですか？私たちのフォルト・モデルと一致して、ディスク全体が故障すると仮定します。後の章（データの完全性）では、より複雑な障害モードの処理方法について考えていきます。最後に、第 3 の軸はパフォーマンスです。パフォーマンスは、ディスク配列に提示される負荷に大きく依存するため、評価するのはやや難しいです。

したがって、パフォーマンスを評価する前に、最初に検討すべき典型的な仕事量のセットを提示します。RAID レベル 0（ストライピング）、RAID レベル 1（ミラーリング）、RAID レベル 4/5（パリティベースの冗長性）の 3 つの重要な RAID 設計を検討します。これらのデザインの「レベル」という命名は、Berkeley [P + 88] の Patterson、Gibson、および Katz の先駆的な仕事に由来します。

38.4 RAID Level 0: Striping

最初の RAID レベルは、冗長性がないという点で、実際には RAID レベルではありません。ただし、RAID レベル 0、つまりストライピングはパフォーマンスと容量の優れた上限になり、理解を深めることができます。

ストライピングの最も単純な形式は、次のようにシステムのディスク全体にブロックをストライプします（ここでは 4 ディスク配列を仮定します）。

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 38.1: RAID-0: Simple Striping

図 38.1 から、ブロックの配列をラウンドロビン方式でディスクに分散させるという基本的な考え方を得られます。このアプローチは、連続したチャンクの配列に対して要求が行われたとき（たとえば、大規模なシーケンシャル読み取りの場合など）、配列から最も多くの並列性を抽出するように設計されています。同じ行のブロックをストライプと呼びます。したがって、ブロック 0,1,2 および 3 は、上の同じストライプにあります。

この例では、各ディスクに次のブロックに移動する前に、1 つのブロック（それぞれサイズ 4KB）が配置されていることを単純化しています。しかしながら、この配置は、必ずしもそうである必要はありません。たとえば、ブロックを図 38.2 のように配置することができます。

Disk 0	Disk 1	Disk 2	Disk 3	
0	2	4	6	chunk size:
1	3	5	7	2 blocks
8	10	12	14	
9	11	13	15	

Figure 38.2: Striping with a Bigger Chunk Size

この例では、各ディスクに 2 つの 4KB ブロックを配置してから、次のディスクに移動します。したがって、この RAID アレイのチャンクサイズは 8KB であり、したがってストライプは 4 チャンクまたは 32KB のデータで構成されます。

ASIDE: THE RAID MAPPING PROBLEM

RAID の容量、信頼性、および性能の特性を調べる前に、まずマッピング問題と呼ばれるものを脇に示します。この問題はすべての RAID 配列で発生します。論理ブロックを読み書きすると、RAID はアクセスする物理ディスクとオフセットを正確にどのように認識していますか？ これらのシンプルな RAID レベルでは、論理ブロックを物理的な場所に正しくマップするために高度な知識は必要ありません。上記の最初のストライピングの例を考えてください（チャンクサイズ = 1 ブロック = 4KB）。この場合、論理ブロックアドレス A が与えられれば、RAID は簡単に 2 つの簡単な式で所望のディスクとオフセットを計算できます。

```
Disk = A % number_of_disks
Offset = A / number_of_disks
```

これらはすべて整数演算であることに注意してください（たとえば、 $4/3 = 1$ ではなく 1.33333 ...）。これらの方程式が簡単な例のためにどのように機能するかを見てみましょう。上記の最初の RAID で、ブロック 14 の要求が到着したとします。4 つのディスクがあると、ブロック 14 が ($14 \% 4 = 2$) ディスク 2 であることを意味します。したがって、ブロック 14 は、3 番目のディスク (0 から始まるので) の 4 番目のブロック (0 から始まるので) に存在しなければいけません。さまざまなチャンクサイズをサポートするためにこれらの方程式がどのように変更されるかについて考えることができます。それを試してみてください！ それほど難しいことではありません。

Chunk Sizes

チャンクサイズは主に配列のパフォーマンスに影響します。たとえば、チャンクサイズが小さいと、多くのファイルが多くのディスクにストライプ化され、1 つのファイルに対する読み書きの並列性が向上します。ただし、要求全体の位置決め時間は、すべてのドライブの要求の位置決め時間の最大値によって決まるため、複数のディスクにまたがるブロックにアクセスするための位置決め時間が長くなります。

一方、大きなチャンクサイズは、このようなイントラファイルの並列性を低減し、従って、高いスループットを達成するために複数の並行要求（並行処理）に依存します。しかし、チャンクサイズが大きいと、位置決め時間が短縮されます。例えば、单一のファイルがチャンク内に収まり、单一のディスク上に置かれた場合、それにアクセスする際に発生する位置決め時間は、单一のディスクの位置決め時間と同じです。

したがって、「最良の」チャンク・サイズを決定することは、ディスク・システムに提示される仕事量に関する

る多くの知識を必要とするため、実行するのが難しいです。[CL95]。ここでは、1 ブロック (4KB) のチャンクサイズを使用すると仮定します。ほとんどの配列はより大きいチャンクサイズ (たとえば 64 KB) を使用しますが、以下で説明する問題については正確なチャンクサイズは関係ありません。したがって、わかりやすくするために単一のブロックを使用しています。

Back To RAID-0 Analysis

ストライピングの容量、信頼性、パフォーマンスを評価しましょう。容量の観点からは、それは完璧です。サイズ B ブロックの N 個のディスクがあれば、ストライピングは $N \cdot B$ ブロックの有用な容量を提供します。信頼性の観点からは、ストライピングも完璧ですが、悪いことにディスク障害が発生するとデータが失われます。最後に、パフォーマンスは優れています。ユーザーの I/O 要求を処理するために、すべてのディスクが頻繁に並行して使用されます。

Evaluating RAID Performance

RAID パフォーマンスの分析では、2 つの異なるパフォーマンスマトリックを考慮することができます。1 つは single request latency(单一の要求の待ち時間) です。RAID に対する単一の I/O 要求の待ち時間を理解することは、単一の論理 I/O 操作中にどれだけの並列性が存在する可能性があるかを明らかにするのに役立ちます。第 2 は、steady state throughput of the RAID(RAID の定常状態スループット)、すなわち多くの同時要求の総帯域幅です。RAID は高性能環境で頻繁に使用されるため、定常状態の帯域幅は非常に重要なので、分析の主な焦点になります。

スループットをより詳細に理解するには、いくつかの重要な仕事量 (仕事量) を提示する必要があります。ここでは、sequential(順次) と random(ランダム) の 2 種類の仕事量があると仮定します。シーケンシャル仕事量では、配列へのリクエストは大きな連続したチャンクになると想定します。例えば、ブロック x で始まりブロック $(x + 1 \text{ MB})$ で終わる 1 MB のデータにアクセスする要求 (または一連の要求) は、連続しているとみなされます。シーケンシャル仕事量は多くの環境で共通しています (キーワードのために大きなファイルを検索することを考えると)、重要と考えられます。

ランダム仕事量の場合、各要求はかなり小さく、各要求はディスク上の別のランダムな場所にあると想定します。たとえば、要求のランダムなストリームは、最初に論理アドレス 10 で 4KB、次に論理アドレス 550,000、次に 20,100 などにアクセスすることができます。データベース管理システム (DBMS) 上のなどの transactional workloads の重要な仕事量は、このタイプのアクセス・パターンを示します。したがって、重要な仕事量とみなされます。

もちろん、実際の仕事量はそれほど単純ではなく、しばしば、シーケンシャルとランダムに見えるコンポーネントの組み合わせと、その 2 つの間の動作を備えています。簡単にするために、これらの 2 つの可能性を考えます。

わかるように、シーケンシャルおよびランダムな仕事量は、ディスクとは大きく異なるパフォーマンス特性をもたらします。シーケンシャルアクセスでは、ディスクは最も効率的なモードで動作し、回転で待機している時間はほとんどなく、ほとんどの時間はデータを転送します。ランダムアクセスの場合、正反対です。ほとんどの時間は回転で待機しており、データの転送にはほとんど時間を費やされません。分析のこの違いを把握するために、ディスクはシーケンシャル仕事量で $S \text{ MB /秒}$ 、ランダム仕事量で $R \text{ MB /秒}$ でデータを転送できると仮定します。一般に、 S は R よりもはるかに大きい (すなわち、 $S >> R$)。

この違いを理解するために、簡単な運動をしましょう。具体的には、次のディスク特性を考慮して S と R を計算しましょう。平均 10MB のサイズの転送と平均 10KB のランダム転送を仮定します。また、次のディスク特性を仮定します。

Average seek time	7 ms
Average rotational delay	3 ms
Transfer rate of disk	50 MB/s

Sを計算するには、まず標準的な10 MB転送で時間が費やされるかどうかを調べる必要があります。まず、7ミリ秒のシークと3ミリ秒の回転を行います。最後に転送が開始されます。10 MB @ 50 MB/sの場合、転送に費やされる時間は1/5秒、つまり200ミリ秒になります。したがって、10 MB要求ごとに、要求を完了するのに210ミリ秒を費やします。Sを計算するには、次のように分割する必要があります。

$$S = \frac{\text{Amount of Data}}{\text{Time to access}} = \frac{10 \text{ MB}}{210 \text{ ms}} = 47.62 \text{ MB/s}$$

わかるように、データ転送に時間がかかるため、Sはディスクのピーク帯域幅に非常に近くなります(シークおよびローテーションのコストは償却されます)。Rも同様に計算できます。シークと回転は同じです。転送に費やされた時間を計算します。転送時間は10KB @ 50MB / s、つまり0.195msです。

$$R = \frac{\text{Amount of Data}}{\text{Time to access}} = \frac{10 \text{ KB}}{0.195 \text{ ms}} = 0.981 \text{ MB/s}$$

わかるように、Rは1 MB/sより小さく、S/Rはほぼ50です。

Back To RAID-0 Analysis, Again

ストライピングのパフォーマンスを評価しましょう。上記のように、一般的には良いです。たとえば、レイテンシの観点からは、シングルブロック要求のレイテンシは、単一のディスクのレイテンシとほぼ同じでなければなりません。結局のところ、RAID-0は単にその要求をそのディスクの1つにリダイレクトします。

定常状態のスループットの観点からは、システムの全帯域幅を得ることが期待されます。したがって、スループットはN(ディスクの数)にS(単一ディスクの連続帯域幅)を掛けたものに等しくなります。多数のランダムI/Oに対して、すべてのディスクを再び使用することができ、N・R MB/sを得ることができます。以下に示すように、これらの値は計算するのが最も簡単で、他のRAIDレベルと比較して上限として機能します。

38.5 RAID Level 1: Mirroring

ストライピングを超える最初のRAIDレベルは、RAIDレベル1またはミラーリングと呼ばれます。ミラーリングされたシステムでは、システム内の各ブロックのコピーを複数作成するだけです。もちろん、それぞれのコピーは別々のディスクに置かなければなりません。これにより、ディスク障害を許容することができます。

典型的なミラーリングされたシステムでは、各論理ブロックについて、RAIDは2つの物理コピーを保持していると仮定します。次に例を示します。

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

Figure 38.3: Simple RAID-1: Mirroring

この例では、ディスク 0 とディスク 1 の内容は同じで、ディスク 2 とディスク 3 も同じです。これらのミラーペアにデータがストライピングされます。実際、ディスクにブロックコピーを置く方法はいくつかあることに気が付いているかもしれません。上記の構成は一般的なもので、ミラーペア (RAID-1)、ストライプ (RAID-0) を使用するため、RAID-10 または (RAID 1 + 0) と呼ばれることもあります。もう 1 つの一般的な構成は、RAID-01(または RAID 0 + 1) です。この配列には、2 つの大きなストライピング (RAID-0) 配列と、その上にミラー (RAID-1) が含まれています。ここでは、上記のレイアウトを想定してミラーリングについて説明します。

ミラーリングされたアレイからブロックを読み取る場合、RAID には選択肢があります。どちらのコピーも読み取ることができます。たとえば、論理ブロック 5 への読み取りが RAID に対して発行された場合、ディスク 2 またはディスク 3 のいずれかから自由に読み取ることができます。ただし、ブロックを書き込む場合、そのような選択肢はありません。信頼性を維持するために両方のコピーのデータを更新します。ただし、これらの書き込みは並行して行うことができます。例えば、論理ブロック 5 への書き込みは、同時にディスク 2 および 3 に書き込むことができます。

RAID-1 Analysis

RAID-1 を評価してみましょう。容量の観点から、RAID-1 は高価です。ミラーリングレベル = 2 の場合、ピークの有効容量の半分しか得られません。B ブロックの N 個のディスクの場合、RAID-1 の有効容量は $(N \cdot B) / 2$ です。

信頼性の観点から、RAID-1 はうまくいきます。いずれかのディスクの障害を許容することができます。また、RAID-1 がこれよりも実際にはうまくいくかもしれないことに気がつくかもしれません。上の図では、ディスク 0 とディスク 2 の両方が故障したとします。このような状況では、データが失われることはあります！ より一般的には、ミラーリングされたシステム (ミラーリングレベル 2) は、特定のディスク障害を 1 回、失敗したディスクに応じて最大 $N / 2$ 回の障害を許容することができます。実際には、私たちは一般的にこのようなことを偶然に残すことは良くないと思っています。したがって、ほとんどの人は、障害を処理するためにミラーリングが良いと考えています。

最後に、パフォーマンスを分析します。単一の読み取り要求の待ち時間の観点から、単一のディスク上の待ち時間と同じであることがわかります。RAID-1 はすべてそのコピーをそのコピーの 1 つに転送します。書き込みは少し異なります。完了する前に 2 回の物理書き込みを完了する必要があります。これらの 2 つの書き込みは並行して行われるため、1 回の書き込み時間とほぼ同じになります。ただし、論理書き込みは両方の物理書き込みが完了するまで待機する必要があるため、2 つの要求の最悪の場合のシークと回転遅延が発生するため、平均して 1 つのディスクへの書き込みよりもわずかに高くなります。

RAID-1 を分析する前に、一貫した更新の問題 [DAA05] と呼ばれるマルチディスク RAID システムで発生する問題についてまず説明します。この問題は、単一の論理操作中に複数のディスクを更新する必要がある RAID への書き込みで発生します。この場合、ミラーリングされたディスク配列を検討しているものとします。

書き込みが RAID に発行されたとすると、RAID はディスク 0 とディスク 1 の 2 つのディスクに書き込む必要があると判断します。次に、RAID はディスク 0 に書き込みを発行しますが、RAID がディスクに要求を発行する直前図 1 に示すように、電力損失(またはシステムクラッシュ)が発生したとします。このような不幸なケースでは、ディスク 0 への要求が完了したと仮定します(ただし、ディスク 1 へ明確に要求はされていないので、ディスク 1 への要求は発行されません)。

不十分な電力損失の結果、ブロックの 2 つのコピーが矛盾していることがあります。ディスク 0 のコピーは新しいバージョンで、ディスク 1 のコピーは古いバージョンです。両方のディスクの状態が原子的に変化すること、すなわち、両方とも新しいバージョンとして終了するか、どちらも終了しないことである。

この問題を解決する一般的な方法は、何らかの種類の先読みログを使用して、その前に RAID が何をするかを最初に記録する(すなわち、2 つのディスクを特定のデータで更新する)ことです。このアプローチをとることで、クラッシュが発生した場合に適切なことが起こることを保証することができます。保留中のすべてのトランザクションを RAID にリカバリする recovery procedure(リカバリ手順)を実行することで、2 つのミラーリングされたコピー(RAID-1 の場合)が同期していないことを確認できます。

最後の注意: 書き込みごとにディスクへのロギングが非常に高価であるため、ほとんどの RAID ハードウェアには、このタイプのロギングを実行する少量の不揮発性 RAM(バッテリバックアップなど)が含まれています。したがって、ディスクへの高コストのロギングを行うことなく、一貫性のあるアップデートが提供されます。

定常状態のスループットを分析するには、順次仕事量から始めましょう。シーケンシャルにディスクに書き込む場合、各論理書き込みでは 2 回の物理書き込みが必要です。たとえば、論理ブロック 0(上の図)を書き込むと、RAID はそれを内部的にディスク 0 とディスク 1 の両方に書き込みます。したがって、ミラーリングされたアレイへのシーケンシャル書き込み中に取得される最大帯域幅は $(N / 2 \cdot S)$ 、またはピーク帯域幅の半分である。

残念ながら、私たちはシーケンシャルリードの間に全く同じ性能を得ています。シーケンシャルな読み込みは、データの 1 つのコピーだけを読み込む必要があり、両方を読み込む必要がないため、よりうまくいくと考えるかもしれません。しかし、これがなぜあまり役に立たないのかを例を挙げて説明しましょう。ブロック 0,1,2,3,4,5,6,7 を読む必要があるとしましょう。ディスク 0 に 0 の読み取り、ディスク 2 に 1 の読み取り、ディスク 1 に 2 の読み取り、ディスク 3 に 3 の読み取りを発行します。ディスク 0,2,1,3 にそれぞれ 4,5,6,7 への読み取りを発行し続けます。すべてのディスクを利用しているため、アレイの全帯域幅を達成していると思うかもしれません。

ただし、必ずしもそうではないことを確認するには、1 つのディスクが受け取る要求(たとえばディスク 0)を考えてみてください。まず、ブロック 0 の要求を取得します。次に、ブロック 4(ブロック 2 をスキップ)に対する要求を取得します。実際、各ディスクは 1 ブロックおきに要求を受け取ります。スキップされたブロック上で回転している間、クライアントに有用な帯域幅を提供していません。したがって、各ディスクはピーク帯域幅の半分しか配信しません。従って、シーケンシャルリードは、 $(N / 2 \cdot S)MB / s$ の帯域幅しか得られません。

ミラーリングされた RAID の場合、ランダム読み取りが最適です。この場合、すべてのディスクに読み取りを配布して、可能な限りの帯域幅を確保できます。したがって、ランダム読み出しの場合、RAID-1 は $N \cdot R$

MB/s で読み取ります。

最後に、期待どおりにランダム書き込みが実行されます。 $N / 2 \cdot R$ MB/s です。各論理書き込みは 2 つの物理書き込みに変わる必要があるため、すべてのディスクが使用されている間というのは、クライアントは利用可能な帯域幅の半分と認識します。論理ブロック x への書き込みが 2 つの異なる物理ディスクへの 2 つの並列書き込みに変わったとしても、多くの小さな要求の帯域幅は、ストライピングで見たものの半分にしか達しません。すぐに利用可能な帯域幅の半分を得ることは、実際にはかなり良いことです。

38.6 RAID Level 4: Saving Space With Parity

ここでは、パリティと呼ばれるディスクアレイに冗長性を追加する別の方法を示します。パリティベースのアプローチでは、容量を少なくして、ミラーリングされたシステムが支払う膨大なスペースペナルティを克服しようとします。コストはかかりません。しかし、パフォーマンスがかかります。

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

Figure 38.4: RAID-4 with Parity

次に、5 ディスク RAID-4 システムの例を示します (図 38.4)。データの各ストライプについて、そのストライプブロックの冗長情報を格納する単一のパリティブロックを追加しました。例えば、パリティブロック P1 は、ブロック 4,5,6、および 7 から計算した冗長な情報を持っています。

パリティを計算するには、ストライプからのブロックのいずれかの損失に耐えることができる数学関数を使用する必要があります。単純な関数 XOR がそのトリックをきちんとやります。与えられたビットの組について、それらのビットのすべての XOR は、ビットに 1 の偶数がある場合は 0 を返し、1 の奇数がある場合は 1 を返します。例えば：

C0	C1	C2	C3	P
0	0	1	1	XOR(0,0,1,1) = 0
0	1	0	0	XOR(0,1,0,0) = 1

第 1 行 (0,0,1,1) には 2 つの 1(C2、C3) があり、したがってこれらの値の XOR はすべて 0(P) になります。同様に、第 2 の行には 1 つの C1(C1) しかないので、XOR は 1(P) でなければいけません。簡単な方法でこれを覚えておくことができます。つまり、任意の行の 1 の数が偶数 (奇数でない) でなければなりません。パリティが正しいように RAID が維持しなければならない不变量です。

上記の例から、パリティ情報を使用して障害から回復する方法を推測することができます。C2 というラベルの付いた列がなくなつたとします。どの値が列内になければならなかつたかを理解するためには、その他のすべての値 (XOR されたパリティビットを含む) を読み込み、正しい答えを再構成するだけです。具体的

には、列 C2 の最初の行の値が失われたとします (1)。その行の他の値 (C0 から 0、C1 から 0、C3 から 1、パリティ列 P から 0) を読み取ることによって、0、0、1、0 の値を取得します。各行に 1 の偶数がある場合、欠落しているデータが何であるかを知っています。それは 1 です。これは、再構築が XOR ベースのパリティ方式でどのように機能するかです。再構成された値の計算方法にも注意してください。最初にパリティを計算したのと同じ方法で、データビットとパリティビットと一緒に XOR します。

今、あなたは疑問に思っているかもしれません。これらのビットすべてを排他的論理和 (XOR) といいますが、RAID によって各ディスク上に 4KB(またはそれ以上) のブロックが配置されています。パリティを計算するために XOR を複数のブロックに適用するにはどうすればよいですか？ これは容易であることが判明しました。データブロックの各ビットにビット単位の XOR を実行するだけです。各ビット単位の XOR の結果をパリティブロックの対応するビットスロットに入れます。たとえば、サイズが 4 ビットのブロックがある場合 (はい、これはまだ 4KB ブロックよりもかなり小さいですが、画像を取得する場合)、次のようにになります。

Block0	Block1	Block2	Block3	Parity
00	10	11	10	11
10	01	00	01	10

図からわかるように、パリティは各ブロックの各ビットごとに計算され、結果はパリティブロックに配置されます。

RAID-4 Analysis

RAID-4 を分析してみましょう。容量の観点から、RAID-4 は保護しているすべてのディスクグループのパリティ情報として 1 つのディスクを使用します。したがって、RAID グループの私たちの有用な容量は $(N - 1) \cdot B$ です。

信頼性も非常に理解しやすいです：RAID-4 は 1 つのディスク障害を許容します。複数のディスクが失われた場合、失われたデータを再構築する方法はありません。

最後に、パフォーマンスがあります。今度は、定常状態のスループットを分析してみましょう。順次読み出しのパフォーマンスは、パリティ・ディスクを除くすべてのディスクを利用できるため、 $(N-1) \cdot S$ MB/s のピーク実効帯域幅を提供します (簡単なケース)。

順次書き込みのパフォーマンスを理解するには、まずそれらがどのように行われているかを理解する必要があります。大量のデータをディスクに書き込む場合、RAID-4 はフルストライプ書き込みと呼ばれる簡単な最適化を実行できます。たとえば、書き込み要求の一部としてブロック 0,1,2,3 が RAID に送信されたとします (図 38.5)。

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

Figure 38.5: Full-stripe Writes In RAID-4

この場合、RAID は P0 の新しい値を単純に計算し (ブロック 0,1,2,3 の XOR を実行して)、上の 5 つのディスクにすべてのブロック (パリティブロックを含む) を並行して書き込みます (図の灰色で強調表示)。したがって、フルストライプ書き込みは、RAID-4 がディスクに書き込むための最も効率的な方法です。

フルストライプ書き込みを理解したら、RAID-4 でのシーケンシャル書き込みのパフォーマンスを計算するのは簡単です。実効帯域幅も $(N - 1) \cdot S \text{ MB/s}$ です。パリティディスクは操作中に常に使用されていますが、クライアントはパフォーマンス上の利点を得られません。

さて、ランダムな読み込みのパフォーマンスを分析しましょう。上記の図からもわかるように、1 ブロックのランダムな読み込みセットは、システムのデータディスク全体に分散されますが、パリティディスクには分散されません。従って、有効な性能は、 $(N - 1) \cdot R \text{ MB/s}$ です。

私たちが最後に保存したランダム書き込みは、RAID-4 にとって最も興味深いケースです。上記の例でブロック 1 を上書きするとします。パリティブロック P0 はストライプの正しいパリティ値を正確に反映しなくなります。この例では、P0 も更新する必要があります。正しく更新するにはどうすればよいですか？

2 つの方法があります。最初のものは、additive parity(加法パリティ)として知られています。新しいパリティブロックの値を計算するには、ストライプ内の他のすべてのデータブロックを並列に読み込み (例ではブロック 0,2,3)、それらを新しいブロック (1) と XOR します。結果が新しいパリティブロックになります。書き込みを完了するために、新しいデータと新しいパリティをそれぞれのディスクに同時に書き込むことができます。

この手法の問題点は、ディスクの数に応じて拡張するため、大規模な RAID ではパリティを計算するために多くの読み込みが必要になります。したがって、subtractive parity method(減法パリティ法)です。たとえば、このビット列 (4 データビット、1 パリティ) を想像してみてください。

C0	C1	C2	C3	P
0	0	1	1	XOR(0,0,1,1) = 0

ビット C2 を C2_new と呼ばれる新しい値で上書きしたいとしましょう。subtractive parity method は 3 つのステップで動作します。まず、C2(C2.old = 1) と古いパリティ (P.old = 0) の古いデータを読み込みます。

次に、古いデータと新しいデータを比較します。それらが同じ場合 (例えば、C2.new = C2.old)、パリティビットも同じままである (すなわち、P.new = P.old) ことがわかる。しかし、それらが異なる場合は、古いパリティビットを現在の状態の反対に、つまり ($P.old == 1$ なら)、P.new を 0 に設定する必要があります。もし ($P.old == 0$) なら、P.new は 1 にセットされます。私たちはこの全体を XOR できれいに表現することができます (\oplus は XOR 演算子です) :

$$P_{new} = (C_{old} \oplus C_{new}) \oplus P_{old} \quad (38.1)$$

ビットではなくブロックを扱うので、ブロック内のすべてのビット (たとえば、各ブロックの 4096 バイトに 1 バイトあたり 8 ビットを掛けたもの) にわたってこの計算を実行します。したがって、ほとんどの場合、新しいブロックは古いブロックとは異なるため、新しいパリティブロックも同様になります。

これで、加算パリティ方式をいつ使うのか、減算方式を使うのかを理解できるはずです。加算方式では減算方式よりも少ない I/O を実行できるように、システムに必要なディスクの数を考える必要があります。クロスオーバーポイントは何ですか？

この性能分析のために、減算方式を使用していると仮定します。したがって、書き込みごとに、RAID は 4 つの物理 I/O(2 回の読み出しと 2 回の書き込み) を実行する必要があります。RAID に提出された多くの書き

込みがあると想像してください。RAID-4は何台並行して実行できますか？理解を深めるために、RAID-4 レイアウトをもう一度見てみましょう（図 38.6）。

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
*4	5	6	7	+P1
8	9	10	11	P2
12	*13	14	15	+P3

Figure 38.6: Example: Writes To 4, 13, And Respective Parity Blocks

今度は、ブロック 4 とブロック 13(図中の*でマークされている)とほぼ同時に、RAID-4 に 2 回の小さな書き込みが行われたとします。

これらのディスクのデータはディスク 0 と 1 にあり、データの読み書きは並行して行われる可能性があります。発生する問題は、パリティディスクにあります。両方の要求は、4 と 13 のパリティブロック 1 と 3(+でマークされている)の関連パリティブロックを読み取らなければなりません。うまくいけば、この問題は明らかです。パリティディスクはこのタイプの仕事量の下でボトルネックです。私たちはこのように時々これをパリティベースの RAID の小さな書き込み問題と呼んでいます。したがって、たとえデータディスクに並列にアクセスすることができたとしても、パリティディスクは並列化が実現するのを防ぎます。パリティディスクのため、システムへのすべての書き込みがシリアル化されます。

パリティディスクは論理 I/O ごとに 2 つの I/O(1 つの読み取り、1 つの書き込み)を実行する必要があるためです。これらの 2 つの I/O でパリティディスクのパフォーマンスを計算することで、RAID-4 での小さなランダム書き込みのパフォーマンスを計算することができ、(R/2)MB/s を達成します。ランダムな小さな書き込みの下で RAID-4 のスループットはひどいです。システムにディスクを追加しても改善されません。

私たちは、RAID-4 の I/O レイテンシを分析して結論づけます。あなたが今知っているように、単一の読み取り(障害がないと仮定)はただ 1 つのディスクにマップされているため、その待ち時間は 1 回のディスク要求の待ち時間と同等です。1 回の書き込みのレイテンシには 2 回の読み取りと 2 回の書き込みが必要です。書き込みはできるだけ並行して行われるため、合計レイテンシは 1 つのディスクの約 2 倍です(いくつかの違いがありますが、両方の読み取りが完了するまで待ち、最悪の場合の位置決め時間を取得する必要があるためです。しかし、そのとき更新は探索コストを必要とせず、したがって、より優れた測位コストとなる可能性があります)。

38.7 RAID Level 5: Rotating Parity

小さな書き込み問題(少なくとも部分的に)に対処するために、Patterson、Gibson、および Katz は RAID-5 を導入しました。RAID-5 は、ドライブ間でパリティブロックを回転させることを除いて、RAID-4 とほぼ同じように機能します(図 38.7)。

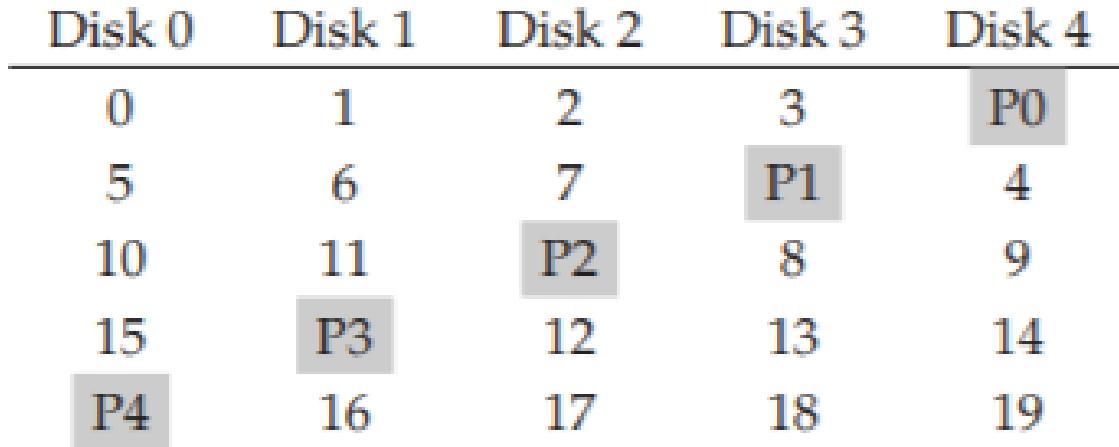


Figure 38.7: RAID-5 With Rotated Parity

ご覧のとおり、RAID-4 のパリティディスクのボトルネックを解消するために、各ストライプのパリティブロックがディスク間で回転しています。

RAID-5 Analysis

RAID-5 の分析の多くは、RAID-4 と同じです。たとえば、2 つのレベルの実効容量と耐障害性は同じです。シーケンシャルな読み書き性能も同様です。単一の要求 (読み込みか書き込みか) のレイテンシも RAID-4 と同じです。

私たちはすべてのディスクを利用するようになったので、ランダムな読み込みパフォーマンスは少し良くなりました。最後に、ランダム書き込みパフォーマンスが RAID-4 よりも大幅に向上し、要求間で並列処理が可能です。ブロック 1 への書き込みとブロック 10 への書き込みを想像してください。これは、ディスク 1 とディスク 4(ブロック 1 とそのパリティ用) とディスク 0 とディスク 2(ブロック 10 とそのパリティ用) に対する要求になります。したがって、彼らは並行して進めることができます。実際、多数のランダムな要求があれば、すべてのディスクを均等に使用することができると一般的に想定できます。そうであれば、小規模の書き込みの総帯域幅は $N/4 \cdot R$ MB/s になります。4 つの損失の要因は、各 RAID-5 書き込みが 4 つの合計 I/O 操作を生成するという事実になります。これは単純にパリティベースの RAID を使用するコストです。

	RAID-0	RAID-1	RAID-4	RAID-5
Capacity	$N \cdot B$	$(N \cdot B)/2$	$(N - 1) \cdot B$	$(N - 1) \cdot B$
Reliability	0	1 (for sure) $\frac{N}{2}$ (if lucky)	1	1
<hr/>				
Throughput				
Sequential Read	$N \cdot S$	$(N/2) \cdot S$	$(N - 1) \cdot S$	$(N - 1) \cdot S$
Sequential Write	$N \cdot S$	$(N/2) \cdot S$	$(N - 1) \cdot S$	$(N - 1) \cdot S$
Random Read	$N \cdot R$	$N \cdot R$	$(N - 1) \cdot R$	$N \cdot R$
Random Write	$N \cdot R$	$(N/2) \cdot R$	$\frac{1}{2} \cdot R$	$\frac{N}{4} R$
Latency				
Read	T	T	T	T
Write	T	T	$2T$	$2T$

Figure 38.8: RAID Capacity, Reliability, and Performance

RAID-5 は基本的に RAID-4 と同じですが、ほんのわずかな場合を除いて、市場では RAID-4 をほぼ完全に

置き換えていきます。唯一の場所は、大きな書き込み以外のことを決して実行しないことを知っているシステムにあるため、小さな書き込みの問題を完全に回避することです [HLM94]。そのような場合は、RAID-4 が使用されることがあります。これは、構築がやや簡単であるためです。

38.8 RAID Comparison: A Summary

ここで、図 38.8 の単純化された RAID レベルの比較をまとめます。分析を簡略化するために、いくつかの詳細を省略したことに注意してください。たとえば、ミラーリングされたシステムでは、シーク時間が 2 シーク (各ディスクに 1 つ) の最大値であるため、单一のディスクに書き込む場合よりも平均シーク時間が少し長くなります。したがって、2 つのディスクへのランダム書き込み性能は、一般に、1 つのディスクのランダム書き込み性能よりも少し低くなります。また、RAID-4/5 のパリティディスクを更新する場合、古いパリティの最初の読み取りは完全なシークと回転を引き起こす可能性がありますが、パリティの 2 回目の書き込みは回転だけになります。

ただし、図 38.8 の比較では本質的な違いを把握でき、RAID レベルのトレードオフを理解するのに役立ちます。レイテンシ解析では、T を使用して、单一のディスクへの要求にかかる時間を表します。

結論として、パフォーマンスを厳密に求め、信頼性に気にしない場合は、明らかにストライピングが最適です。ただし、ランダム I/O パフォーマンスと信頼性が必要な場合は、ミラーリングが最適です。あなたが支払うコストは容量です。容量と信頼性が主な目標である場合は、RAID5 が最適です。あなたが支払うコストは小さな書き込みパフォーマンスです。最後に、シーケンシャル I/O を常に行い、容量を最大化したい場合は、RAID-5 が最適です。

38.9 Other Interesting RAID Issues

RAIDについて考えるときに話し合うことができる(そしておそらくそうすべき)多くの他の興味深い考えがあります。私たちが最終的に書くかもしれないものは次のとおりです。

たとえば、元のタクソノミーのレベル 2 と 3、複数のディスク障害に耐えるレベル 6 など、他の多くの RAID 設計があります [C + 04]。ディスクに障害が発生したときに RAID が行うこともあります。場合によっては、故障したディスクをカバーするための hot spare(ホットスペア) があります。また、障害発生時のパフォーマンス、障害が発生したディスクの再構築中のパフォーマンスはどうなりますか？さらに、潜在的なセクタのエラーやブロックの破損 [B + 08] を考慮する、より現実的なフォルトモデルや、そのようなフォルトを処理する多くのテクニックがあります(詳細については、データの完全性の章を参照してください)。最後に、RAID をソフトウェアレイヤーとして構築することもできます。ソフトウェア RAID システムは安価ですが、一貫性のあるアップデート問題 [DAA05] を含む他の問題があります。

38.10 Summary

我々は RAID について議論しました。RAID は、多数の独立したディスクを、大規模で大容量で信頼性の高い単一のエンティティに変換します。重要なのは、それは透過的に行われるため、上記のハードウェアとソフトウェアは、その変換を忘れないことです。

選択可能な RAID レベルは数多くあり、使用する RAID レベルはエンドユーザーにとって重要なものに大きく依存します。たとえば、ミラーリングされた RAID はシンプルで信頼性が高く、一般的にパフォーマンスは高くなりますが、容量にコストがかかります。対照的に、RAID-5 は信頼性が高く、容量面では優れていますが、仕事量に書き込みが少ない場合はパフォーマンスが非常に悪くなります。RAID を選択し、特定の仕事量に対してそのパラメータ(チャンクサイズ、ディスク数など)を適切に設定することは困難であり、科学というよりか、もはや芸術です。

参考文献

- [B+08] “An Analysis of Data Corruption in the Storage Stack”
 Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
 FAST ’08, San Jose, CA, February 2008
 Our own work analyzing how often disks actually corrupt your data. Not often, but sometimes! And thus something a reliable storage system must consider.
- [BJ88] “Disk Shadowing”
 D. Bitton and J. Gray
 VLDB 1988
 One of the first papers to discuss mirroring, herein called “shadowing”.
- [CL95] “Striping in a RAID level 5 disk array”
 Peter M. Chen, Edward K. Lee
 SIGMETRICS 1995
 A nice analysis of some of the important parameters in a RAID-5 disk array.
- [C+04] “Row-Diagonal Parity for Double Disk Failure Correction”
 P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, S. Sankar
 FAST ’04, February 2004
 Though not the first paper on a RAID system with two disks for parity, it is a recent and highlyunderstandable version of said idea. Read it to learn more.
- [DAA05] “Journal-guided Resynchronization for Software RAID”
 Timothy E. Denehy, A. Arpaci-Dusseau, R. Arpaci-Dusseau
 FAST 2005
 Our own work on the consistent-update problem. Here we solve it for Software RAID by integrating the journaling machinery of the file system above with the software RAID beneath it.
- [HLM94] “File System Design for an NFS File Server Appliance”
 Dave Hitz, James Lau, Michael Malcolm
 USENIX Winter 1994, San Francisco, California, 1994
 The sparse paper introducing a landmark product in storage, the write-anywhere file layout or WAFL file system that underlies the NetApp file server.
- [K86] “Synchronized Disk Interleaving”
 M.Y. Kim.
 IEEE Transactions on Computers, Volume C-35: 11, November 1986
 Some of the earliest work on RAID is found here.
- [K88] “Small Disk Arrays - The Emerging Approach to High Performance”
 F. Kurzweil.
 Presentation at Spring COMPCON ’88, March 1, 1988, San Francisco, California Another early RAID reference.
- [P+88] “Redundant Arrays of Inexpensive Disks”
 D. Patterson, G. Gibson, R. Katz.
 SIGMOD 1988
 This is considered the RAID paper, written by famous authors Patterson, Gibson, and Katz. The paper

has since won many test-of-time awards and ushered in the RAID era, including the name RAID itself!

39 Interlude: Files and Directories

ここまででは、CPU の仮想化であるプロセスと、メモリの仮想化であるアドレス空間の 2 つの主要なオペレーティングシステムの抽象化の発展を見てきました。この 2 つの抽象化によって、プログラムは、あたかも自分の世界であるかのように動くことができます。あたかもそれ自身のプロセッサ（または複数のプロセッサ）、自らのメモリを持っているかのように動くことができます。このような錯覚は、システムのプログラミングをはるかに容易にし、今日ではデスクトップやサーバだけでなく、携帯電話などのプログラマブルなプラットフォーム上でますます普及しています。

このセクションでは、仮想化パズルに重要な要素である永続ストレージを追加します。古典的なハードディスクドライブまたはより現代的なソリッドステートストレージデバイスなどの永続ストレージデバイスは、情報を永続的に（または少なくとも長期間）保存します。停電が発生したときに内容が失われるメモリとは異なり、永続記憶装置はそのようなデータをそのまま維持します。したがって、OS はそのようなデバイスに特別な注意を払わなければなりません。これは、ユーザーが実際に気にかけているデータを保存する場所です。

CRUX: HOW TO MANAGE A PERSISTENT DEVICE

OS は永続的なデバイスをどのように管理すべきですか？ API とは何ですか？ 実装の重要な側面は何ですか？

したがって、今後のいくつかの章では、パフォーマンスと信頼性を向上させる方法に焦点を当て、永続データを管理するための重要なテクニックについて検討します。ただし、API の概要、つまり UNIX ファイルシステムと対話するときに表示されると予想されるインターフェイスについて説明します。

39.1 Files and Directories

ストレージの仮想化では、2 つの主要な抽象概念が時間とともに発展しています。最初はファイルです。ファイルは単なる線形バイト配列です。各バイトは読み書きできます。各ファイルには、ある種の低レベルの名前が付いています。通常、いくつかの種類があります。ユーザーはこの名前を認識していないことがあります（次のとおりです）。歴史的な理由から、ファイルの低レベルの名前はよくその inode 番号と呼ばれます。私たちは将来の章で inode についてもっと学びます。今のところ、各ファイルに関連付けられている inode 番号があると仮定します。

ほとんどのシステムでは、OS はファイルの構造（例えば、画像であるか、テキストファイルであるか、C ソースコードであるか）をほとんど知りません。むしろ、ファイルシステムの責任は、そのようなデータをディスクに永続的に保存し、データを再度要求するときに、最初にそこに置いたものを確実に取得することです。そうすることはそれほど単純ではありません！

2 番目の抽象化はディレクトリの抽象化です。ファイルのようなディレクトリも低レベルの名前（つまり、inode 番号）を持っていますが、その内容は非常に特殊です（ユーザが読める名前、低レベルの名前）のリストを含んでいます。たとえば、低レベルの名前「10」を持つファイルがあり、ユーザが判読可能な「foo」という名前で参照されているとします。「foo」が存在するディレクトリは、ユーザが読める名前を低レベルの名前にマッピングするエントリ（“foo”、“10”）を持ちます。ディレクトリ内の各エントリは、ファイルまたは他のディレクトリを参照します。ディレクトリを他のディレクトリに配置することにより、ユーザは任意のディレクトリツリー（またはディレクトリ階層）を構築することができ、その下にすべてのファイルおよびディレクトリが格納されます。

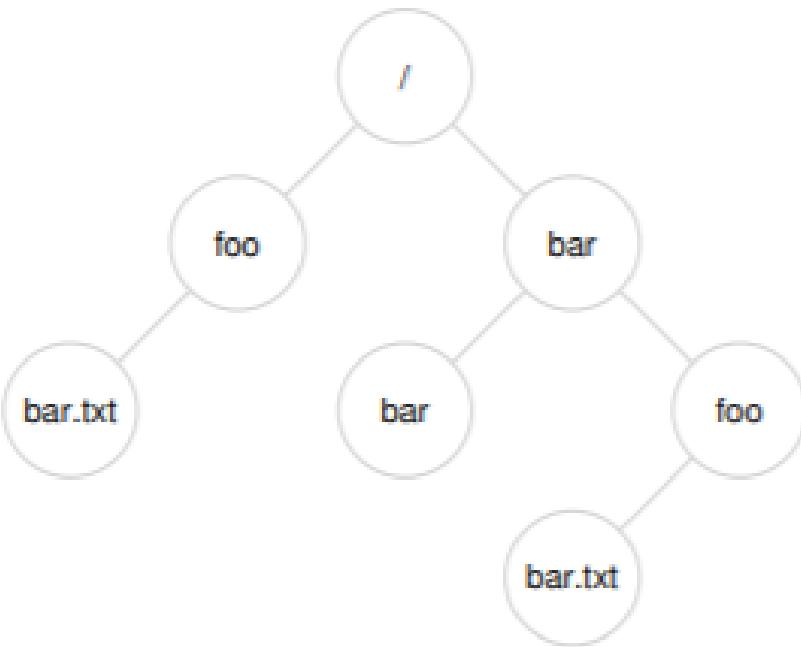


Figure 39.1: An Example Directory Tree

ディレクトリ階層はルートディレクトリ (UNIX ベースのシステムではルートディレクトリは単に / と呼ばれます) から始まり、目的のファイルまたはディレクトリが指定されるまで、ある種のセパレータを使用して後続のサブディレクトリの名前を付けます。たとえば、ユーザーがルートディレクトリ / にディレクトリ foo を作成し、ディレクトリ foo に bar.txt というファイルを作成した場合、絶対パス名でファイルを参照できます。この場合は /foo/bar.txt です。より複雑なディレクトリツリーについては、図 39.1 を参照してください。この例の有効なディレクトリは /、/foo、/bar、/bar/bar、/bar/foo です、有効なファイルは /foo/bar.txt および /bar/foo/bar.txt です。ディレクトリとファイルは、ファイルシステムツリー内の異なる場所にある限り、同じ名前を持つことができます (たとえば、図の bar.txt という 2 つのファイル /foo/bar.txt と /bar/foo/bar.txt があります)

TIP: THINK CAREFULLY ABOUT NAMING

命名は、コンピュータシステムの重要な側面である [SK09]。UNIX システムでは、あなたが考えることができる事実上すべてがファイルシステムによって命名されます。ファイル、デバイス、パイプ、さらにはプロセス [K84] だけでなく、普通の古いファイルシステムのように見えます。名前の統一性により、システムの概念モデルが簡単になり、シンプルでモジュラーなシステムになります。したがって、システムまたはインターフェースを作成するときは、使用している名前について注意深く考えてください。

また、この例のファイル名には、bar と txt の 2 つの部分があり、ピリオドで区切られています。最初の部分は任意の名前ですが、ファイル名の 2 番目の部分は通常、C コード (例 : .c) であるか、画像 (例 : .jpg) であるか、音楽ファイル (例 : .mp3) であるなど、ファイルの種類を示すために使用されます。しかし、通常これは規約に過ぎません。通常、main.c という名前のファイルに含まれているデータは確かに C のソースコードであるという強制はありません。

したがって、ファイルシステムが提供するすばらしいことの 1 つがわかります。関心のあるすべてのファイルの名前を付ける便利な方法です。名前は、すべてのリソースにアクセスするための最初のステップで名前を

付けることができるため、システムでは重要です。UNIX システムでは、このように、ファイルシステムは、ディスク、USB スティック、CD-ROM、その他多くのデバイス上のファイルにアクセスする統一された方法を提供します。事実、多くのその他のものは、単一のディレクトリツリーの配下にあります。

39.2 The File System Interface

次に、ファイルシステムのインターフェースについて詳しく説明します。まず、ファイルの作成、アクセス、削除の基本について説明します。これは簡単だと思うかもしれません、途中で `unlink()` と呼ばれるファイルを削除するために使用される不思議な呼び出しを発見します。うまくいけば、この章の最後では、この謎はあなたにとって、神秘的なものではなくなるでしょう！

39.3 Creating Files

まず、ファイルの作成という最も基本的な操作から始めます。これは、`open` システムコールで実現できます。`open()` を呼び出して `O_CREAT` フラグを渡すことにより、プログラムは新しいファイルを作成することができます。現在の作業ディレクトリに “foo” というファイルを作成するコードの例をいくつか示します。

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);
```

ルーチン `open()` はいくつかの異なるフラグをとります。この例では、2 番目のパラメータは存在しない場合はファイル (`O_CREAT`) を作成し、ファイルは書き込むことのみ (`O_WRONLY`) ができるようになります。ファイルがすでに存在する場合は、0 バイトのサイズに切り捨てて、既存のコンテンツ (`O_TRUNC`) を削除します。3 番目のパラメータは、パーミッションを指定します。この場合、所有者がファイルを読み書き可能になります。

ASIDE: THE `CREAT()` SYSTEM CALL ファイルを作成する古い方法は、次のように `creat()` を呼び出すことです。

```
int fd = creat("foo"); // option: パーミッションを設定するための第 2 フラグを追加する
```

`creat()` は `open()` として次のフラグで考えることができます：`O_CREAT` | `O_WRONLY` | `O_TRUNC` です。`open()` はファイルを作成することができるので、`creat()` の使用法は少し賛成できません（実際には `open()` へのライブラリ呼び出しとして実装することができます）。しかし、それは UNIX の知識の特別な場所を保持しています。具体的には、ケン・トンプソンが UNIX を再設計していた場合、彼が何をどうやってやるのかと質問されたとき、彼は「私は e をスペルとして作るだろう」と答えました。

`open()` の重要な点の 1 つは、ファイルディスクリプタです。ファイルディスクリプタは、単なる整数で、プロセスごとのプライベートなもので、ファイルにアクセスするために UNIX システムで使用されます。したがって、ファイルが開かれると、そのファイル記述子を使用して、そのファイルを読み書きする権限を持っているとみなします。

このようにして、ファイルディスクリプタは機能 [L84]、つまり、特定の操作を実行する権限を与える不透明なハンドルです。ファイルディスクリプタを考えるもう一つの方法は、ファイル型オブジェクトへのポインタです。そのようなオブジェクトがあれば、`read()` や `write()` のようにファイルにアクセスするための他の“メソッド”を呼び出すことができます。以下に、ファイル記述子がどのように使用されるかを見ていきます。

39.4 Reading and Writing Files

一度私たちがいくつかのファイルを持っていれば、私たちはそれらを読み書きしたいかもしれません。既存のファイルを読むことから始めましょう。コマンドラインで入力していた場合は、プログラム cat を使用して、ファイルの内容を画面にダンプすることができます。

```
prompt> echo hello > foo
prompt> cat foo
hello
prompt>
```

このコードスニペットでは、プログラム echo の出力をファイル foo にリダイレクトします。ファイル foo には単語 “hello” が含まれています。次に、cat を使用してファイルの内容を表示します。しかし、cat プログラムはどのようにしてファイル foo にアクセスしますか？

これを見つけるために、私たちは非常に便利なツールを使用して、プログラムによるシステムコールをトレースします。Linux では、このツールは strace と呼ばれます。他のシステムにも同様のツールがあります (Mac の場合は dtruss、古い UNIX の場合は truss を参照)。strace は、プログラムの実行中にシステムコールが発生したときにそれをトレースし、画面にトレースをダンプして表示します。

TIP: USE STRACE (AND SIMILAR TOOLS)

strace ツールは、どんなプログラムであるかを確認する素晴らしい方法を提供します。これを実行することで、プログラムが作るシステムコールをトレースしたり、引数や戻りコードを調べたり、一般的に何が起きているかを知ることができます。

また、このツールにはかなり役に立ついくつかの議論があります。例えば、-f はフォークされた子もトレースします。-t は各呼び出し時の時刻を報告します。-e は trace = open、close、read、write のシステムコールの呼び出しをトレースし、他のすべてを無視します。その他にもより多くの強力なフラグがあります。マニュアルページを読んで、このすばらしいツールを活用する方法を見つけてください。

strace を使用して cat が何をしているかを把握する例を示します (読みやすくするためにいくつかの呼び出しを削除しました)。

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE)      = 3
read(3, "hello\n", 4096)               = 6
write(1, "hello\n", 6)                  = 6
hello
read(3, "", 4096)                     = 0
close(3)                               = 0
...
prompt>
```

cat が最初に行なうことは、読み込み用のファイルを開くことです。私たちはこれについて注意すべき事柄をいくつか挙げておきます。最初に、ファイルは O_RDONLY フラグで示されているように、読み込み専用 (書き

込みではない)に開かれています。第2に、64ビットオフセットを使用します(O_LARGEFILE)。open()の呼び出しが成功し、値3を持つファイルディスクリプタを返します。

最初にopen()を呼び出すと、期待どおり0またはおそらく1ではなく3が返されます。各実行中のプロセスには、すでに3つのファイルが開いています。標準入力(プロセスが入力を受け取るために読み取ることができます)、標準出力(プロセスを情報を画面にダンプするために書き込むことができます)、標準エラー(そのプロセスはエラーメッセージを書き込むことができます)があります。これらは、それぞれファイル記述子0,1,2で表されます。したがって、最初に別のファイルを開くと(上記のcatのように)、ファイル記述子3になります。

openが成功すると、catはread()システムコールを使用してファイルからいくつかのバイトを繰り返し読み込みます。read()の最初の引数はファイルディスクリプタで、ファイルシステムにどのファイルを読み込ませるかを指示します。プロセスはもちろん、一度に複数のファイルを開くことができ、したがって、ディスクリプタは、オペレーティングシステムが特定の読み取りがどのファイルを参照するかを知ることを可能にします。2番目の引数は、read()の結果が格納されるバッファを指します。上記のシステムコールトレースでは、straceはこの箇所の読み込み結果を表示します("hello")。3番目の引数はバッファのサイズで、この場合は4KBです。read()の呼び出しも正常に戻ります。ここでは、読み込んだバイト数(今回の場合は6バイト、単語"hello"の文字は5、行末マーカーは1で合わせて6)を返します。

この時点でのstraceのもう1つの興味深い結果が得られます。これは、write()システムコールをファイルディスクリプタ1で1回呼び出すことです。前述のように、この記述子は標準出力として知られています。プログラムのcatが意味するように画面に"hello"という単語が表示されます。しかし、それは直接write()を呼び出すでしょうか?おそらくはそうでしょう(高度に最適化されている場合)しかし、もしそうでなければ、どのようにcatが出力するのかというと、ライブラリルーチンprintf()を呼び出します。printf()は、渡されたすべての書式設定の詳細を調べ、最終的に標準出力で書き込みを呼び出して結果を画面に出力します。

次に、catプログラムはファイルからさらに多くを読み込もうとしますが、ファイルに残っているバイトがないので、read()は0を返し、プログラムはファイル全体を読み取ったことを知ります。したがって、プログラムはclose()を呼び出して、ファイル"foo"が完了したことを示し、対応するファイルディスクリプタを渡します。ファイルはこのようにして閉じられ、読み込みが完了します。ファイルの書き込みは、同様の手順で行います。まず、書き込みのためにファイルが開かれた後、大きなファイルに対してwrite()システムコールが呼び出され、その後close()が呼び出されます。straceを使用して、あなた自身が書いたプログラムのようなファイルへの書き込みをトレースするか、ddユーティリティでトレースします(例: dd if=foo of=bar)。

39.5 Reading And Writing, But Not Sequentially

ここまででは、ファイルの読み書き方法について説明しましたが、アクセスはすべて順番に行われています。つまり、最初から最後までファイルを読み込んだり、最初から最後までファイルを書き出したりしています。

ただし、ファイル内の特定のオフセットを読み書きできることが便利な場合もあります。たとえば、テキスト文書上に索引を作成し、それを使用して特定の単語を検索すると、文書内のランダムな一部のオフセットから読み込まれることがあります。これを行うには、lseek()システムコールを使用します。ここに関数プロトタイプがあります:

```
off_t lseek(int fildes, off_t offset, int whence);
```

最初の引数は使い慣れたものです(ファイルディスクリプタ)。2番目の引数はオフセットで、ファイルオフセットをファイル内の特定の場所に配置します。歴史的な理由からwhenceと呼ばれる第3引数は、シークの実行方法を正確に決定します。マニュアルページから:

```
If whence is SEEK_SET, the offset is set to offset bytes.
If whence is SEEK_CUR, the offset is set to its current
location plus offset bytes.
If whence is SEEK_END, the offset is set to the size of
the file plus offset bytes.
```

この説明からわかるように、プロセスが開く各ファイルについて、OSは「現在の」オフセットを追跡します。このオフセットは、次の読み取りまたは書き込みがファイル内の読み取りまたは書き込みの開始位置を決定します。したがって、オープンファイルの抽象化の一部は、現在のオフセットを持つことであり、現在のオフセットは2つの方法のいずれかで更新されます。第1は、Nバイトの読み出しありまたは書き込みが行われるときであり、Nが現在のオフセットに加算されます。したがって、各読み取りまたは書き込みは、暗黙的にオフセットを更新します。2番目はlseekで明示的に指定されており、上で指定したオフセットを変更します。

ASIDE: CALLING LSEEK() DOES NOT PERFORM A DISK SEEK

名前のわからないシステムコールlseek()は、多くの学生がディスクを理解しようとしているのを混乱させ、その上にあるファイルシステムの仕組みを混乱させます。2つを混同しないでください！lseek()コールは、OSメモリ内の変数を変更するだけで、特定のプロセスについて、次の読み込みまたは書き込みが開始されるオフセットを追跡します。ディスクシークは、ディスクに発行された読み取りまたは書き込みが最後の読み取りまたは書き込みと同じトラックにないときに発生し、したがってヘッドの移動が必要になります。これをさらに混乱させることは、lseek()を呼び出してファイルのランダムな部分を読み書きすること、そしてそれらのランダムな部分を読み書きすることが実際にはより多くのディスクシークをもたらすことです。したがって、lseek()を呼び出すと、今後の読み取りまたは書き込みで確実にシーク(オフセットの変更)はしますが、どんなディスクI/Oもlseek()で発生することはありません。

この呼び出し lseek() は、ディスクアームを動かすディスクのシーク操作とは何の関係もないことに注意してください。lseek() の呼び出しは、単にカーネル内の変数の値を変更します。I/O が実行されるとき、ディスクヘッドがどこにあるかに応じて、ディスクは要求を実行するための実際のシークを実行してもよいし、しなくともよいです。

39.6 Writing Immediately with fsync()

ほとんどの場合、プログラムがwrite()を呼び出すと、それはファイルシステムに伝えているだけです。このデータを将来のある時点で永続ストレージに書き込んでください。パフォーマンス上の理由から、ファイルシステムはこのような書き込みをしばらく(例えば5秒または30秒)メモリにバッファします。後の時点で、書き込みは実際に記憶装置に発行されます。呼び出し元のアプリケーションの観点からは、書き込みはすぐ完了するよう(たとえば、write()呼び出し後でディスクへの書き込みの前にマシンがクラッシュするなど)、データが失われることがあります。

しかし、一部のアプリケーションでは、この最終的な保証以上のものが必要です。たとえば、データベース管理システム(DBMS)では、正しい回復プロトコルを開発するには、時々ディスクへの書き込みを強制する能力が必要です。

これらのタイプのアプリケーションをサポートするために、ほとんどのファイルシステムはいくつかの追加の制御APIを提供します。UNIXの世界では、アプリケーションに提供されるインタフェースはfsync(int fd)として知られています。特定のファイルディスククリプタのプロセスがfsync()を呼び出すと、ファイルシステムは、指定されたファイルディスククリプタによって参照されるファイルに対して、ディスクにすべてのデータ(つまりまだ書き込まれていない)データを強制します。これらの書き込みがすべて完了すると、

`fsync()` ルーチンが戻ります。

`fsync()` の使い方の簡単な例を次に示します。コードはファイル `foo` を開き、そのファイルに単一のチャンクを書き込んだ後、`fsync()` を呼び出して書き込みがディスクに強制的に強制されるようにします。`fsync()` が返されると、アプリケーションはデータが永続化されていることを知り（もし、`fsync()` が正しく実装されていれば）、安全に動かすことができます。

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);
assert(fd > -1);
int rc = write(fd, buffer, size);
assert(rc == size);
rc = fsync(fd);
assert(rc == 0);
```

興味深いことに、このシーケンスでは、あなたが期待できるすべてを保証するものではありません。場合によっては、ファイル `foo` を含むディレクトリを `fsync()` する必要があります。この手順を追加すると、ファイル自体がディスク上にあるだけでなく、新しく作成された場合でも、そのファイルが永続的にディレクトリの一部になります。驚くことではないが、この種の詳細は見落とされがちで、多くのアプリケーションレベルのバグが発生します [P + 13, P + 14]。

39.7 Renaming Files

いったんファイルがあると、ファイルに別の名前を付けることができると便利なことがあります。コマンドラインで入力するときは、これは `mv` コマンドで行います。この例では、ファイル `foo` の名前が `bar` に変更されています。

```
prompt> mv foo bar
```

`strace` を使用すると、`mv` はシステムコール `rename(char * old, char * new)` を使用します。これは、ファイルの元の名前 (`old`) と新しい名前 (`new`) の 2 つの引数を正確にとります。`rename()` 呼び出しが提供する興味深い保証の 1 つは、システムクラッシュに関してアトミックな呼び出しとして（通常）実装されていることです。名前の変更中にシステムがクラッシュした場合、ファイルは古い名前または新しい名前のいずれかに名前が付けられるといった、奇妙な中間状態は発生しません。

したがって、`rename()` は、ファイル状態へのアトミックな更新を必要とする特定の種類のアプリケーションをサポートするために重要です。ここで少し具体的に話します。ファイルエディタ（`emacs` など）を使用していて、ファイルの途中に行を挿入したとします。この例のファイル名は `foo.txt` です。新しいファイルが元の内容と行が挿入されていることを保証するために、エディタがファイルを更新する方法は次のとおりです（単純化のためにエラーチェックを無視します）。

```
int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC,
              S_IRUSR|S_IWUSR);
write(fd, buffer, size); // write out new version of file
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

この例では、エディタは単純です。新しい名前のファイルを一時的な名前 (`foo.txt.tmp`) で書き出し、`fsync()` でディスクに強制的に書き出し、アプリケーションが新しいファイルを特定したら、メタデータと内容がディ

スク上にある場合は、一時ファイルの名前を元のファイルの名前に変更します。この最後のステップでは、ファイルの古いバージョンを同時に削除しながら、新しいファイルをアトミックにスワップし、アトミックなファイルの更新を実現します。

39.8 Getting Information About Files

ファイルアクセス以外にも、ファイルシステムは、格納している各ファイルについてかなりの量の情報を保持することが期待されます。一般に、ファイルメタデータに関するこのようなデータを呼び出します。特定のファイルのメタデータを確認するには、`stat()` または `fstat()` システムコールを使用します。これらの呼び出しは、ファイルにパス名(またはファイルディスクリプタ)をとり、ここに示すような `stat` 構造体を埋めます：

```
struct stat {
    dev_t      st_dev;      /* ID of device containing file */
    ino_t      st_ino;      /* inode number */
    mode_t     st_mode;     /* protection */
    nlink_t    st_nlink;    /* number of hard links */
    uid_t      st_uid;      /* user ID of owner */
    gid_t      st_gid;      /* group ID of owner */
    dev_t      st_rdev;     /* device ID (if special file) */
    off_t      st_size;     /* total size, in bytes */
    blksize_t  st_blksize;  /* blocksize for filesystem I/O */
    blkcnt_t   st_blocks;   /* number of blocks allocated */
    time_t     st_atime;    /* time of last access */
    time_t     st_mtime;    /* time of last modification */
    time_t     st_ctime;    /* time of last status change */
};
```

ファイルのサイズ(バイト数)、低レベルの名前(つまり、inode番号)、所有権情報、ファイルへのアクセスまたは変更に関する情報など、各ファイルについて多くの情報が保持されていることがわかります。この情報を表示するには、コマンドラインツール `stat` を使用します。

```
prompt> echo hello > file
prompt> stat file
  File: 'file'
  Size: 6 Blocks: 8          IO Block: 4096   regular file
Device: 811h/2065d Inode: 67158084      Links: 1
Access: (0640/-rw-r-----)  Uid: (30686/  remzi) Gid: (30686/  remzi)
Access: 2011-05-03 15:50:20.157594748 -0500
Modify: 2011-05-03 15:50:20.157594748 -0500
Change: 2011-05-03 15:50:20.157594748 -0500
```

実際には、各ファイルシステムは通常、この種の情報を `inode` という構造体に保持しています。私たちがファイルシステムの実装について話すとき、`inode`についてもっと学びます。今のところ、`inode` はファイルシステムによって保持されている永続的なデータ構造であり、その中に上記のような情報があると考えてください。

39.9 Removing Files

この時点では、ファイルを作成、アクセスして、それらを順次アクセスするかどうかを知っています。しかし、どのようにファイルを削除しますか？ UNIX を使ったことがあるなら、あなたはおそらく知っていると思うでしょう：プログラム `rm` を実行するだけです。しかし、`rm` がファイルを削除するために使用するシステムコールは何ですか？ `strace` でもう一度調べてみましょう。ここでは、厄介なファイル “foo” を削除します。

```

prompt> strace rm foo
...
unlink ("foo") = 0
...

```

トレースされた出力から無関係なクラフトを削除し、不思議な名前のシステムコール `unlink()` を1回だけ呼び出します。ご覧のように、`unlink()` は削除されるファイルの名前だけを受け取り、成功するとゼロを返します。しかし、これは大きなパズルにつながります。なぜこのシステムコールが「リンク解除」と呼ばれていますか？単に「削除する」または「削除する」だけではありません。このパズルへの答えを理解するためには、単にファイルだけでなくディレクトリも理解する必要があります。

39.10 Making Directories

ファイル以外のディレクトリ関連のシステムコールを使用すると、ディレクトリの作成、読み取り、および削除を行うことができます。ディレクトリに直接書き込むことはできません。ディレクトリの形式はファイルシステムのメタデータと見なされるため、たとえばファイル、ディレクトリ、またはその他のオブジェクト型を作成するなどして間接的にディレクトリを更新することはできます。このようにして、ファイルシステムはディレクトリの内容が常に期待どおりであることを確認します。

ディレクトリを作成するには、単一のシステムコール `mkdir()` を使用できます。mkdir プログラムを実行して `foo` という単純なディレクトリを作成するとどうなるか見てみましょう。

```

prompt> strace mkdir foo
...
mkdir ("foo", 0777) = 0
...
prompt>

```

TIP: BE WARY OF POWERFUL COMMANDS

プログラム `rm` は、強力なコマンドの素晴らしい例を私たちに提供し、時にはあまりにも多くのパワーが悪いことになることがあります。たとえば、たくさんのファイルを一度に削除するには、次のように入力します。

```
prompt> rm *
```

ここで、`*`は現在のディレクトリ内のすべてのファイルと一致します。しかし、時にはディレクトリも、実際にはすべての内容を削除したいこともあります。これを行うには、`rm` に再帰的に各ディレクトリへの降下を指示し、その内容も削除します。

```
prompt> rm -rf *
```

この小さな文字列で問題が発生するのは、ファイルシステムのルートディレクトリから偶発的にコマンドを発行し、そこからすべてのファイルとディレクトリを削除する場合です。したがって、強力なコマンドの両刃の剣を覚えておいてください。少数のキーストロークで多くの作業を行うことができますが、すばやく簡単に大きな被害を受けることができます。

このようなディレクトリが作成されると、最低限の内容しか持たないものの、“空”とみなされます。具体的

には、空のディレクトリには、自身を参照するエントリとその親を参照するエントリの2つのエントリがあります。前者は「.」(ドット)ディレクトリと呼ばれ、後者は「..」(ドットドット)と呼ばれます。これらのディレクトリは、プログラムlsにフラグ(-a)を渡すことで確認できます。

```
prompt> ls -a
./ ..
prompt> ls -al
total 8
drwxr-x--- 2 remzi remzi 6 Apr 30 16:17 .
drwxr-x--- 26 remzi remzi 4096 Apr 30 16:17 ..
```

39.11 Reading Directories

ディレクトリを作成したので、それを読むこともできます。確かに、それはまさにlsプログラムのことです。lsのような独自の小さなツールを書いて、それがどのように行われるか見てみましょう。

あたかもファイルであるかのようにディレクトリを開くのではなく、代わりに新しい呼び出しを使用します。以下は、ディレクトリの内容を出力するプログラム例です。このプログラムは、opendir()、readdir()、およびclosedir()の3つの呼び出しを使用してジョブを完了させます。どのように簡単にインターフェースを作っているのかというと、単純なループを使用して一度に1つのディレクトリエンタリを読み込み、ディレクトリ内の各ファイルの名前とiノード番号を出力します。

```
int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL) {
        printf("%lu %s\n", (unsigned long) d->d_ino, d->d_name);
    }
    closedir(dp);
    return 0;
}
```

以下の宣言は、struct direntデータ構造内の各ディレクトリエンタリ内で利用可能な情報を示しています。

```
struct dirent {
    char          d_name[256]; /* filename */
    ino_t         d_ino;       /* inode number */
    off_t         d_off;       /* offset to the next dirent */
    unsigned short d_reclen;  /* length of this record */
    unsigned char d_type;    /* type of file */
};
```

ディレクトリは情報が軽いので(基本的に、名前をinode番号にマッピングするだけで、他の詳細もいくつかあります)、プログラムは各ファイルのstat()を呼び出して、それぞれの長さやその他の詳細情報を取得したいでしょう。確かに、これはあなたが-lフラグを渡したときのこととまったく同じです。そのフラグの有無にかかわらず、自分でstraceを試してみてください。

39.12 Deleting Directories

最後に、`rmdir()`(同じ名前のプログラム `rmdir` が使用する) を呼び出して、ディレクトリを削除することができます。ただし、ファイルの削除とは異なり、ディレクトリを削除する方が危険です。単一のコマンドで大量のデータを削除する可能性があるためです。したがって、`rmdir()` は、削除される前にディレクトリが空である(すなわち、“.”と“..”エントリのみを有する)という要件を持っています。空でないディレクトリを削除しようとすると、`rmdir()` の呼び出しは失敗します。

39.13 Hard Links

ここでは、ファイルシステムのツリーにエントリを作る新しい方法を理解することによって、`link()` と呼ばれるシステムコールを通して、ファイルを削除することがなぜ `unlink()` によって行われるのかという謎に戻ってきました。`link()` システムコールは古いパス名と新しいパス名の 2 つの引数をとります。新しいファイル名を古いファイル名に「リンク」すると、同じファイルを参照する別の方法が基本的に作成されます。この例のように、コマンドラインプログラム `ln` を使用してこれを行います。

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2
prompt> cat file2
hello
```

ここでは、“hello”という単語を含むファイルを作成し、そのファイル (`file`) を呼び出しました。その後、`ln` プログラムを使用してそのファイルへのハードリンクを作成します。その後、`file`を開くか、`file2`を開くかのどちらかでファイルを調べることができます。リンクが動作する方法は、リンクを作成するディレクトリに別の名前を作成し、元のファイルの同じ i ノード番号(つまり、低レベルの名前)を参照するだけです。ファイルはどのようにもコピーされません。むしろ、同じファイルを参照する 2 つの人の名前 (`file` と `file2`) だけを持つようになりました。ディレクトリ自体にも、各ファイルの inode 番号を表示することでこれを見ることができます：

```
prompt> ls -i file file2
67158084 file
67158084 file2
prompt>
```

`ls` に-i フラグを渡すことで、各ファイルの i ノード番号(ファイル名と同様に)を出力します。そして、あなたはリンクが実際に何をしているかを知ることができます：同じ正確な inode 番号(この例では 67158084)への新しい参照を作成するだけです。

これで、`unlink()` が `unlink()` と呼ばれている理由が分かり始めました。ファイルを作成すると、実際には 2 つのことが行われます。最初に、サイズ、ブロックがディスク上にある場所など、ファイルに関するすべての関連情報を実質的に追跡する構造体(i ノード)を作成しています。次に、人が読める名前をそのファイルにリンクし、そのリンクをディレクトリに入れます。

ファイルへのハードリンクを作成した後、元のファイル名(ファイル)と新しく作成されたファイル名(ファイル 2)に違いはありません。実際には、これらのファイルは、ファイルに関する基となるメタデータへのリンクにすぎず、inode 番号 67158084 にあります。

したがって、ファイルシステムからファイルを削除するには、`unlink()` を呼び出します。上記の例では、`file` という名前のファイルを削除しても問題なくファイルにアクセスできます。

```
prompt> rm file
removed 'file'
prompt> cat file2
hello
```

これは、ファイルシステムがファイルのリンクを解除すると、iノード番号内の参照カウントをチェックするためです。この参照カウント（リンクカウントとも呼ばれる）によって、ファイルシステムは、この特定の i ノードにリンクされているファイル名の数を追跡できます。`unlink()` が呼び出されると、人間が読める名前（削除されるファイル）と指定された inode 番号との間の“リンク”が削除され、参照カウントが減少します。参照カウントが 0 になった場合にのみ、ファイルシステムは i ノードおよび関連するデータブロックも解放し、ファイルを本当に「削除」します。

もちろん `stat()` を使ってファイルの参照カウントを見ることができます。ファイルへのハードリンクを作成して削除するときの状態を見てみましょう。この例では、同じファイルへのリンクを 3 つ作成してから削除します。リンク数を見てください！

```
prompt> echo hello > file
prompt> stat file
... Inode: 67158084      Links: 1 ...
prompt> ln file file2
prompt> stat file
... Inode: 67158084      Links: 2 ...
prompt> stat file2
... Inode: 67158084      Links: 2 ...
prompt> ln file2 file3
prompt> stat file
... Inode: 67158084      Links: 3 ...
prompt> rm file
prompt> stat file2
... Inode: 67158084      Links: 2 ...
prompt> rm file2
prompt> stat file3
... Inode: 67158084      Links: 1 ...
prompt> rm file3
```

39.14 Symbolic Links

本当に便利なもう1つのタイプのリンクがあります。これは、シンボリックリンクまたは時にはソフトリンクと呼ばれています。ハードリンクはいくらか限定されています。つまり、ディレクトリツリーにサイクルを作成する恐れがあるため、ディレクトリには作成できません。inode番号はファイルシステム全体ではなく、特定のファイルシステム内で一意であるため、他のディスクパーティション内のファイルにハードリンクすることはできません。したがって、シンボリックリンクと呼ばれる新しいタイプのリンクが作成されました。

このようなリンクを作成するには、同じプログラム `ln` を使用できますが、`-s` フラグを使用します。次に例を示します。

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
```

ご覧のように、ソフトリンクの作成はほとんど同じように見え、元のファイルはファイル名ファイルとシンボリックリンク名 `file2` でアクセスできるようになりました。

しかし、この表面の類似性を超えて、シンボリックリンクは実際にはハードリンクとはかなり異なっています。最初の違いは、シンボリックリンクは実際は別の種類のファイルそのものです。私たちはすでに正規のファイルとディレクトリについて話しました。シンボリックリンクは、ファイルシステムが知っている第3のタイプです。シンボリックリンクの統計はすべてを示します：

```
prompt> stat file
... regular file ...
prompt> stat file2
... symbolic link ...
```

`ls` を実行すると、この事実も明らかになります。`ls` の出力の長い形式の最初の文字を注意深く見ると、一番左の列の最初の文字は- 通常のファイル、ディレクトリの場合はd、ソフトリンクの場合はlです。シンボリックリンクのサイズ(この場合は4バイト)と、リンクが指しているもの(`file`という名前のファイル)を見ることもできます。

```
prompt> ls -al
drwxr-x--- 2 remzi remzi 29 May 3 19:10 .
drwxr-x--- 27 remzi remzi 4096 May 3 15:14 ..
-rw-r----- 1 remzi remzi 6 May 3 19:10 file
lrwxrwxrwx 1 remzi remzi 4 May 3 19:10 file2 -> file
```

`file2` が4バイトである理由は、シンボリックリンクが形成される方法がリンク先ファイルのパス名をリンクファイルのデータとして保持するためです。`file`というファイルにリンクしているので、リンクファイル `file2` は小さい(4バイト)です。より長いパス名にリンクすると、リンクファイルが大きくなります：

```
prompt> echo hello > longerfilename
prompt> ln -s longerfilename file3
prompt> ls -al longerfilename file3
-rw-r----- 1 remzi remzi 6 May 3 19:17 longerfilename
lrwxrwxrwx 1 remzi remzi 15 May 3 19:17 file3 -> longerfilename
```

最後に、シンボリックリンクが作成される方法のために、彼らはぶら下がった参照 (dangling reference) として知られているものの可能性を残します：

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
prompt> rm file
prompt> cat file2
cat: file2: No such file or directory
```

この例のように、ハードリンクとはまったく違うので、元のファイル file を削除するとリンクはもはや存在しないパス名を指し示します。

39.15 Making and Mounting a File System

これで、ファイル、ディレクトリ、特定の種類の特殊な種類のリンクにアクセスするための基本的なインターフェイスを見てきました。しかし、多くの基となるファイルシステムから完全なディレクトリツリーを組み立てる方法について、議論すべきトピックがもう 1 つあります。この作業は、最初にファイルシステムを作成し、マウントして内容にアクセスできるようにすることで実現します。

ファイルシステムを作るために、ほとんどのファイルシステムは、通常このタスクを実行する mkfs(make fs と発音) と呼ばれるツールを提供します。アイデアは以下の通りです：ツールに入力としてデバイス (例えば、/dev/sda1 などのディスクパーティション) とファイルシステムタイプ (ext3 など) を与え、空のファイルシステムを書き始めるだけです。ルートディレクトリとして、そのディスクパーティションにコピーします。mkfs によると、ファイルシステムがあります！

しかし、いったんこのようなファイルシステムが作成されると、それは一様なファイルシステムツリー内でアクセス可能にする必要があります。この作業は、マウントプログラム (実際の作業を行うための、基となるシステムコール mount() を呼び出す) によって実現されます。マウントとは、既存のディレクトリをターゲットマウントポイントとして使用し、その時点で新しいファイルシステムをディレクトリツリーに貼り付けることです。

こここの例は役に立つかかもしれません。デバイスパーティション/dev/sda1 に格納されている ext3 ファイルシステムがアンマウントされているとします。ルートディレクトリには a と b という 2 つのサブディレクトリがあり、それぞれに foo という単一のファイルが格納されています。このファイルシステムをマウントポイント/home/users にマウントしたいとしましょう。次のように入力します。

```
prompt> mount -t ext3 /dev/sda1 /home/users
```

成功した場合、マウントによってこの新しいファイルシステムが利用可能になります。ただし、新しいファイルシステムへのアクセス方法に注意してください。ルートディレクトリの内容を見るには、ls を次のように使用します。

```
prompt> ls /home/users/
a b
```

ご覧のとおり、パス名/home/users/は、新しくマウントされたディレクトリのルートを参照します。同様に、パス名が/home/users/a と /home/users/b のディレクトリ a と b にアクセスできます。最後に、foo という名前のファイルに/home/users/a/foo と /home/users/b/foo からアクセスできます。そして、マウントの美しさとして、複数のファイルシステムを持つ代わりに、マウントはすべてのファイルシステムを 1 つのツリーに統合し、命名を統一して便利にします。

あなたのシステムに何がマウントされているのか、どのポイントにマウントされているのかを確認するには、mount プログラムを実行するだけです。次のようなものが表示されます：

```
/dev/sda1 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
/dev/sda5 on /tmp type ext3 (rw)
/dev/sda7 on /var/vice/cache type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
AFS on /afs type afs (rw)
```

この狂った組み合わせは、ext3(標準のディスクベースのファイルシステム)、proc ファイルシステム(現在のプロセスに関する情報をアクセスするためのファイルシステム)、tmpfs(一時ファイル用のファイルシステム)、AFS(分散ファイルシステム)はすべて、この 1 つのマシンのファイルシステムツリーにまとめられています。

39.16 Summary

UNIX システム(実際にはどのシステムでも)のファイルシステムインターフェースは、一見して非常に初步的ですが、それを習得したいかどうかを理解することはたくさんあります。もちろん、単純にそれを使用するよりも、何も良いことはありません(やることはたくさんあります)。だからしてください！もちろん、もっと読んでください。いつものように、スティーブンス [SR05] が始まる場所です。

私たちは基本的なインターフェースを見学して、うまくいかに動作するかを少しは理解していました。さらに興味深いのは、API のニーズを満たすファイルシステムを実装する方法です。これについては、次に詳しく解説します。

参考文献

- [K84] “Processes as Files”
 - Tom J. Killian
 - USENIX, June 1984
 - The paper that introduced the /proc file system, where each process can be treated as a file within a pseudo file system. A clever idea that you can still see in modern UNIX systems.
- [L84] “Capability-Based Computer Systems”
 - Henry M. Levy
 - Digital Press, 1984
 - Available: <http://homes.cs.washington.edu/~levy/capabook>
 - An excellent overview of early capability-based systems.
- [P+13] “Towards Efficient, Portable Application-Level Consistency”

Thanumalayan S. Pillai, Vijay Chidambaram, Joo-Young Hwang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau HotDep '13, November 2013

Our own work that shows how readily applications can make mistakes in committing data to disk; in particular, assumptions about the file system creep into applications and thus make the applications work correctly only if they are running on a specific file system.

[P+14] “All File Systems Are Not Created Equal:

On the Complexity of Crafting Crash-Consistent Applications” Thanumalayan S. Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

OSDI '14, Broomfield, Colorado

The full conference paper on this topic – with many more details and interesting tidbits than the first workshop paper above.

[SK09] “Principles of Computer System Design”

Jerome H. Saltzer and M. Frans Kaashoek

Morgan-Kaufmann, 2009

This tour de force of systems is a must-read for anybody interested in the field. It’s how they teach systems at MIT. Read it once, and then read it a few more times to let it all soak in.

[SR05] “Advanced Programming in the UNIX Environment”

W. Richard Stevens and Stephen A. Rago

Addison-Wesley, 2005

We have probably referenced this book a few hundred thousand times. It is that useful to you, if you care to become an awesome systems programmer.

40 File System Implementation

この章では、vsfs(Very Simple File System)と呼ばれる単純なファイルシステムの実装を紹介します。このファイルシステムは、一般的なUNIXファイルシステムの簡略化されたバージョンであるため、今日の多くのファイルシステムで見られる基本的なオンディスク構造、アクセス方法、およびさまざまなポリシーを紹介しています。

ファイルシステムは純粋なソフトウェアです。CPUやメモリの仮想化の開発とは異なり、ファイルシステムの一部の機能を向上させるハードウェア機能を追加することはできません(ただし、ファイルシステムがうまく機能するようにデバイスの特性に注意する必要があります)。ファイルシステムの構築には大きな柔軟性があるため、AFS(Andrew File System)[H + 88]からZFS(SunのZettabyte File System)[B07]に至るまで、さまざまなファイルシステムが構築されています。これらのファイルシステムはすべて、さまざまなデータ構造を持ち、同輩よりも良くて悪いものもあります。したがって、ファイルシステムについて学ぶ方法は、事例研究です。まず、この章では、簡単なファイルシステム(vsfs)と、実際のファイルシステムについての一連の調査を行い、どう違うのかを演習のなかで理解していきましょう。

THE CRUX: HOW TO IMPLEMENT A SIMPLE FILE SYSTEM

単純なファイルシステムを構築するにはどうすればよいですか？ディスクにはどのような構造が必要ですか？彼らは何を追跡する必要がありますか？彼らはどのようにアクセスされますか？

40.1 The Way To Think

ファイルシステムについて考えるには、通常、2つの異なる側面について考えることをお勧めします。これらの側面の両方を理解すれば、ファイルシステムが基本的にどのように動作するかを理解していると思います。

1つめはファイルシステムのデータ構造です。言い換えれば、ファイルシステムがデータとメタデータを整理するためにどのような種類のオンディスク構造が利用されていますか？SGIのXFSのようなより洗練されたファイルシステムは、より複雑なツリーベースの構造(S + 96)を使用するのに対し、最初のファイルシステム(以下のvsfsを含む)はブロック配列やその他のオブジェクトのような単純な構造を採用しています。

ASIDE: MENTAL MODELS OF FILE SYSTEMS

前に説明したように、メンタルモデルは、システムについて学ぶときに実際に開発しようとしているものです。ファイルシステムでは、質問の中に答えがあることがあります。例えば、ファイルシステムのデータとメタデータを格納するディスク上の構造はどうなりますか？プロセスがファイルを開くとどうなりますか？どのオンディスク構造が読み書き中にアクセスされるのですか？などです。メンタルモデルの作成と改善によって、ファイルシステムのソースコードの詳細を理解しようとする代わりに、何が起きているのかを抽象的に理解することができます。(もちろん、ソースコードを理解することも大事です)

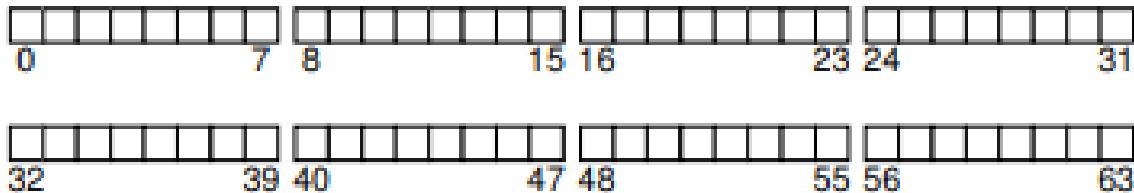
ファイルシステムの第2の側面は、そのアクセス方法です。`open()`、`read()`、`write()`などのプロセスによって呼び出された呼び出しをその構造にどのようにマップしますか？特定のシステムコールの実行中に読み取られる構造はどれですか？どれに書きこまれますか？これらのステップはどれくらい効率的に実行されますか？

ファイルシステムのデータ構造とアクセス方法を理解している場合、システムの考え方の重要な部分である、それが本当にどのように機能するかについての良いメンタルモデルを開発しました。私たちの最初の実装を掘り下げながら、メンタルモデルの開発に取り掛かりましょう。

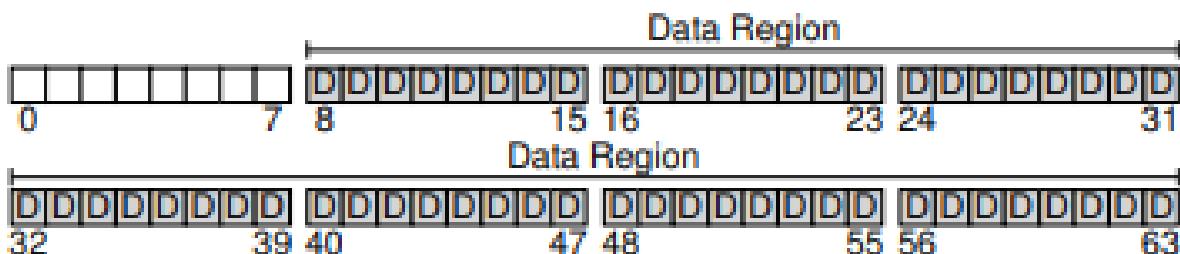
40.2 Overall Organization

現在、ディスク上の vsfs ファイルシステムのデータ構造の全体的な構成を開発しています。まず、ディスクをブロックに分割する必要があります。単純なファイルシステムはブロックサイズを 1つしか使用しません、そして今からそれを行います。一般的に使用される 4 KB のサイズを選択しましょう。

そのため、ファイルシステムを構築するディスクパーティションについては、サイズが 4KB の一連のブロックが簡単です。ブロックは、サイズ N の 4KB ブロックのパーティション内で、0 から N-1 までアドレス指定されます。実際には 64 ブロックの非常に小さなディスクがあるとします。

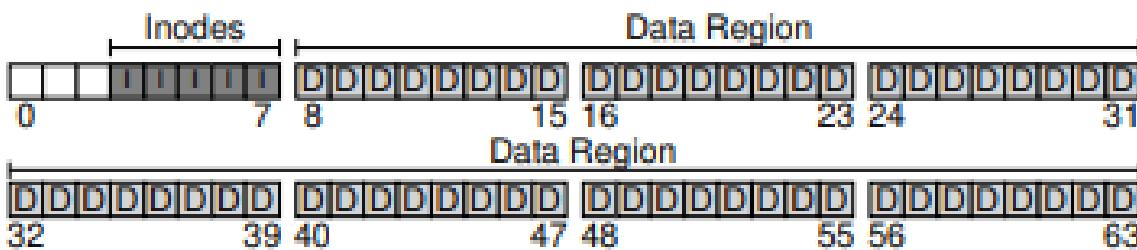


ファイルシステムを構築するためにこれらのブロックに格納する必要があるものについて考えてみましょう。もちろん、最初に気になるのはユーザーデータです。実際には、どのファイルシステムのほとんどの領域もユーザーデータです（そして、そうでなければなりません）。ユーザーデータ用に使用するディスクの領域をデータ領域と呼びましょう。また、簡単にするために、ディスク上の固定された部分、たとえばディスク上の 64 個のブロックのうちの後ろの 56 個を予約します。



最後の章で学んだように、ファイルシステムは各ファイルに関する情報を追跡する必要があります。この情報は、メタデータの重要な部分であり、ファイル、ファイルのサイズ、所有者とアクセス権、アクセスと変更時刻、およびその他の同様の種類の情報を含むデータブロック（データ領域内）のようなものを追跡します。この情報を格納するために、ファイルシステムは通常、inode と呼ばれる構造を持っています（以下の inode についてさらに詳しく説明します）。

i ノードに対応するためには、ディスク上にもスペースを確保する必要があります。ディスクのこの部分を i ノード表と呼びましょう。これは単にディスク上の i ノードの配列を保持しています。このように、ディスク上のイメージは、図のようになります。ここでは、私たちの 64 個のブロックのうちの 5 個を i ノードに使用していると仮定しています（I で示されています）。

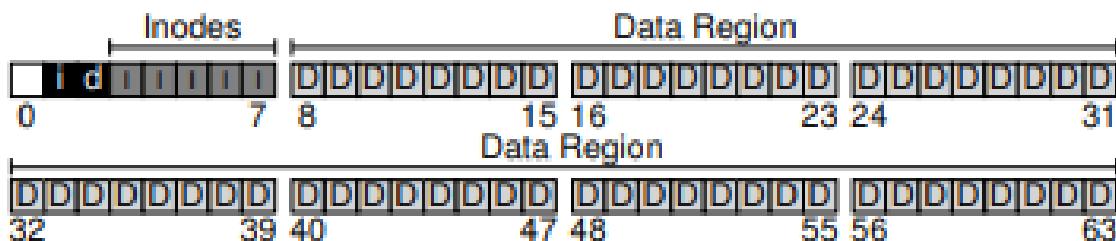


inode は通常 128 または 256 バイトのように大きくないことに注意してください。inode 当たり 256 バイトと

仮定すると、4KB のブロックは 16 個の inode を保持でき、上記のファイルシステムは 80 個の合計 inode を含んでいます。この私たちの小さな 64 ブロックのパーティションに構築されたシンプルなファイルシステムでは、この数字はファイルシステムに保存できるファイルの最大数を表します。ただし、より大きなディスク上に構築された同じファイルシステムでは、より大きな inode テーブルを割り当てるだけで、より多くのファイルに対応できるということに注意してください。

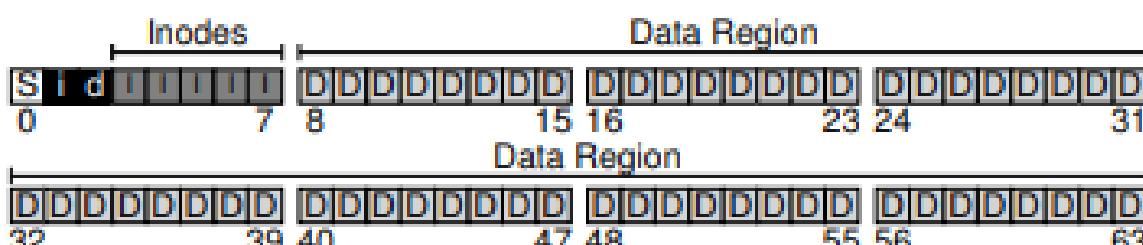
これまでの私たちのファイルシステムは、データブロック (D) と inode(I) を持っていますが、まだいくつか欠けています。あなたが推測したように、依然として必要な 1 つの主要なコンポーネントは、inode やデータブロックが空いているか割り当てられているかを追跡する方法です。したがって、このような割り当て構造は、どのファイルシステムにおいても必須の要素です。

もちろん、多くの割り当て追跡方法が可能です。たとえば、最初の空きブロックを指す空きリストを使用して、次の空きブロックを指示することができます。代わりに、データ領域 (データビットマップ) と i ノードテーブル (i ノードビットマップ) 用のビットマップという単純で一般的な構造を選択します。ビットマップは、単純な構造です。各ビットは、対応するオブジェクト/ブロックが空き (0) か使用中 (1) かを示すために使用されます。そして、私たちの新しいディスク上のレイアウトには、inode ビットマップ (i) とデータビットマップ (d) があります。



これらのビットマップには 4 KB のブロック全体を使用するのはちょっと残念ですが、そのようなビットマップは 32K オブジェクトが割り当てられているかどうかを追跡できますが、80 個の inode と 56 個のデータブロックしか持っていないません。しかし、単純化のためにこれらのビットマップのそれぞれに 4 KB のブロック全体を使用しています。

慎重な読者（まだ起きている読者）は、非常に単純なファイルシステムのオンディスク構造の設計に 1 ブロック残っていることに気づいているかもしれません。私たちはスーパーブロックのためにこれを予約しています。下図の S で示されています。スーパーブロックには、この特定のファイルシステムに関する情報が含まれます。たとえば、ファイルシステム内にいくつの i ノードとデータブロックがあるか（この場合は 80 と 56）、i ノードテーブルが開始する場所（ブロック 3）などがあります。おそらく、ファイルシステムの種類（この場合は vsfs）を特定するための何らかのマジックナンバーも含まれています。



したがって、ファイルシステムをマウントするとき、オペレーティングシステムはまずスーパーブロックを読み取り、さまざまなパラメータを初期化し、ボリュームをファイルシステムツリーにアタッチします。ボリューム内のファイルにアクセスすると、システムは必要なオンディスク構造を探す場所を正確に認識します。

40.3 File Organization: The Inode

ファイルシステムの最も重要なオンディスク構造の1つは、iノードです。事実上すべてのファイルシステムは、これと同様の構造を持っています。名前 inode は、索引ノードの略語であり、UNIX[RT74] やこれまでのシステムで使用されていた歴史的な名前で、簡単なシステムかもしれません。これらのノードはもともと配列に配列されていたために使用され、配列は特定の inode にアクセスする際に索引付けされます。

ASIDE: DATA STRUCTURE — THE INODE

inode は、長さ、パーミッション、および構成ブロックの位置など、特定のファイルのメタデータを保持する構造を記述するために、多くのファイルシステムで使用される汎用名です。その名前は、少なくとも UNIX まで(そしておそらく、以前のシステムではないにしても Multics に戻ります)戻ります。inode 番号がディスク上の inode の配列にインデックスを付けてその番号の i ノードを見つけるために使用されるので、インデックスノードの略です。ここからわかるように、inode の設計はファイルシステム設計の重要な部分の1つです。現代のシステムのほとんどは、追跡しているすべてのファイルに対してこのような構造を持っていますが、呼び方は違います(dnode、fnode などの別のものを呼び出すことがあります。)

各 i ノードは暗黙のうちにファイルの低レベルの名前と呼ばれる番号 (inumber) で暗黙的に参照されます。vsfs(およびその他の単純なファイルシステム)では、inumber(i 番号) が与えられているので、対応する inode がディスク上のどこにあるかを直接計算することができます。例えば、上記のような vsfs の inode テーブルを取ると、20 KB のサイズ(5つの4 KB ブロック)が得られ、80 個の i ノードから構成されます(各 i ノードが 256 バイトであると仮定します)。さらに inode 領域が 12KB から始まると仮定する(すなわち、スーパー ブロックが 0KB から始まり、inode ビットマップがアドレス 4KB にあり、データビットマップが 8KB であり、したがって inode テーブルが直後に来る)。したがって、vsfs では、ファイルシステムパーティションの先頭に、次のようなレイアウトがあります。

The Inode Table (Closeup)									
			iblock 0	iblock 1	iblock 2	iblock 3	iblock 4		
Super	i-bmap	d-bmap	0 1 2 3 18 17 18 19 32 33 34 35 48 49 50 51 64 65 66 67	4 5 6 7 20 21 22 23 36 37 38 39 52 53 54 55 68 69 70 71	8 9 10 11 24 25 26 27 40 41 42 43 58 57 58 59 72 73 74 75	12 13 14 15 28 29 30 31 44 45 46 47 60 61 62 63 76 77 78 79			
0KB	4KB	8KB	12KB	16KB	20KB	24KB	28KB	32KB	

inode 番号 32 を読み込むには、ファイルシステムは最初に inode 領域($32 \cdot \text{sizeof(inode)}$ または 8192)へのオフセットを計算し、それをディスク上の inode テーブルの開始アドレス(inodeStartAddr = 12KB)に追加し、したがって、望んだ inode ブロックの正しいバイトアドレス 20KB に達します。ディスクは byte addressable(バイト・アドレス可能)ではなく、多数の addressable(アドレス可能)なセクタ、通常は 512 バイトで構成されていることを思い出してください。したがって、inode32 を含む inode のブロックをフェッチするために、ファイルシステムは、望んだ inode ブロックをフェッチするために、セクタ($20 \times 1024 / 512$ すなわち 40)への読み出しを発行します。より一般的には、i ノードブロックのセクタアドレス iaddr は、以下のように計算することができます。

```
blk = (inumber * sizeof(inode_t)) / blockSize;
sector = ((blk * blockSize) + inodeStartAddr) / sectorSize;
```

それぞれの inode の中には、ファイルの種類(通常ファイル、ディレクトリなど)、ファイルのサイズ、割り当てられたブロック数、保護情報(ファイルの所有者、アクセス権)、ファイルが作成、変更、または最後にアクセスされた時を含むいくつかの時間情報、およびそのデータブロックがディスク上のどこに存在するかに関する情報(例えば、何らかのポインタ)といったファイルに関するすべての必要な情報を含んでいます。このようなファイルに関するすべての情報をメタデータといいます。実際、純粋なユーザーデータではないファイルシステム内の情報は、よくそのように呼ばれます。ext2 [P09] の i ノードの例を図 40.1 に示します。

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

Figure 40.1: Simplified Ext2 Inode

inode の設計における最も重要な決定の 1 つは、データブロックがどこにあるかをどのように参照するかです。1 つの簡単なアプローチは、i ノード内に 1 つ以上の直接ポインタ(ディスクアドレス)を持つことです。各ポインタは、ファイルに属する 1 つのディスクブロックを参照します。そのようなアプローチは限定されています。たとえば、実際に大きなファイル(ブロックのサイズに直接ポインタの数を掛けたものよりも大きい)を作成する場合は、不運です。

The Multi-Level Index

より大きいファイルをサポートするために、ファイルシステム設計者は inode 内に異なる構造を導入しなければなりませんでした。1 つの一般的な考え方とは、間接ポインタと呼ばれる特別なポインタを持つことです。ユーザーデータを含むブロックを指すのではなく、より多くのポインターを含むブロックを指し、各ポインターはユーザーデータを指します。したがって、i ノードは、いくつかの固定数の直接ポインタ(例えば、12 個)を修正して、および单一の間接ポインタを持つようにします。ファイルが十分に大きくなると、(ディスクのデータブロック領域から)間接ブロックが割り当てられ、間接ポインタのための inode のスロットは、それを指すように設定されます。4KB のブロックと 4 バイトのディスクアドレスを想定すると、さらに 1024 個のポインタが追加されます。ファイルは $(12 + 1024) \cdot 4K$ または 4144KB になります。

TIP: CONSIDER EXTENT-BASED APPROACHES

別のアプローチは、ポインタの代わりにエクステントを使うことです。エクステントは単にディスクポインタと長さ(ブロック単位)です。したがって、ファイルの各ブロックにポインタを必要と

する代わりに、ファイルのディスク上の場所を指定するポインタと長さが必要です。ファイルを割り当てるときにディスク上の空き領域が連続しているのを見つけていく場合があるため、一つ(ブロック単位)のエクステントが制限されています。したがって、エクステントベースのファイルシステムでは、二つ以上(ブロック単位)のエクステントが許可されることが多く、ファイルの割り当て中にファイルシステムに自由度が増します。

2つのアプローチを比較すると、ポインタベースのアプローチが最も柔軟ですが、ファイルごとに大量のメタデータを使用します(特に大きなファイルの場合)。エクステントベースのアプローチは柔軟性は低くなりますが、コンパクトです。特に、ディスク上に十分な空き領域があり、ファイルを連続して配置できる(実際にはどのようなファイル割り当て方針の目標でもあります)場合にうまく機能します。

驚くべきことではありませんが、このようなアプローチでは、さらに大きなファイルをサポートしたいことがあります。これを行うには、別のポインタ inode を追加してください: 二重間接ポインタです。このポインタは、間接ブロックへのポインタを含むブロックを指し、各ブロックはデータブロックへのポインタを含みます。したがって、間接的な二重ブロックは、 1024×1024 または 100 万個の 4KB ブロックを追加してファイルを拡張する可能性、つまり 4GB を超えるファイルの追加をサポートすることができるでしょう。あなたはさらに大きいファイルを追加することを望むかもしれません。そして、これがどこに向かうのかを知っています: トリプル間接ポインタです。

全体として、この imbalanced tree(不均衡なツリー) は、ファイルブロックを指すためのマルチレベルインデックスアプローチと呼ばれています。例えば、12 個の直接ポインタ、1 個の間接ブロック、2 個の間接ブロックの三つが合わさった例を調べてみましょう。4KB のブロックサイズと 4 バイトのポインタを仮定すると、この構造は 4GB をちょうど上回るファイル(すなわち、 $(12 + 1024 + 1024^2) \times 4KB$) を収容することができます。トリプル間接ブロックを追加することで、どれだけのファイルを扱うことができるのか分かりますか? (ヒント: かなり大きい)

多くのファイルシステムでは、Linux ext2 [P09] や ext3、NetApp の WAFL などの一般的に使用されているファイルシステムや元の UNIX ファイルシステムなど、マルチレベルのインデックスを使用しています。SGI XFS や Linux ext4 などの他のファイルシステムでは、単純なポインタの代わりにエクステントを使用します。エクステントベースのスキームがどのように機能するかの詳細については、前のセクションを参照してください(仮想メモリの議論のセグメントに似ています)。

あなたは疑問に思うかもしれません:なぜこのような imbalanced tree(不均衡なツリー) を使用しますか? どうして別のアプローチはありませんか? さて、多くの研究者が、ファイルシステムとその使用方法、そして何十年にもわたって一定の「真実」を見つける研究してきました。そのような発見の 1 つは、ほとんどのファイルが小さいことです。この不均衡なデザインはそのような現実を反映しています。ほとんどのファイルが実際に小さい場合、この場合に最適化することは理にかなっています。したがって、少数の直接ポインタ(12 が典型的な数) では、inode は直接的に 48KB のデータを指し示すことができ、大きなファイルに対しては 1 つ(またはそれ以上) の間接ブロックが必要です。最近の研究 Agrawal et.al [A + 07] については、図 40.2 にその結果をまとめています。

ASIDE: LINKED-BASED APPROACHES

inode を設計するもう一つの簡単な方法は、linked list を使うことです。したがって、inode 内では、複数のポインタを持つ代わりに、ファイルの最初のブロックを指すポインタが必要です。大きなファイルを処理するには、そのデータブロックの末尾に別のポインタを追加するなどして、大きなファイルをサポートすることができます。

あなたが推測したように、リンクされたファイルの割り当ては、一部の仕事量ではうまく機能しません。たとえば、ファイルの最後のブロックを読み込むことや、ランダムアクセスを行うことなど

について考えてみてください。したがって、リンクされた割り当てをより良くするために、一部のシステムでは、データブロック自体に次のポインタを格納するのではなく、リンク情報のメモリ内テーブルを保持します。このテーブルは、データブロック D のアドレスによって索引付けされます。エントリの内容は、単に D の次のポインタ、すなわち D に続くファイル内の次のブロックのアドレスです。NULL もそこに入ります。この NULL は、ファイルの終わり、または特定のブロックがフリーであることを示します。そのような次のポインタのテーブルを持つことにより、リンクされた割り当てスキームは、最初に (メモリ内の) テーブルをスキャンして目的のブロックを見つけて、それを直接 (ディスク上に) アクセスすることによって、ランダムなファイルアクセスを効果的に行うことができます。

そのようなテーブルはよい方法に思えるでしょうか？我々が説明したことは、ファイルアロケーションテーブル、すなわち FAT ファイルシステムと呼ばれるものの基本的な構造です。これは、NTFS [C94] より前のこの古典的な古い Windows ファイルシステムは、単純なリンクベースの割り当て方式に基づいています。標準的な UNIX ファイルシステムとの相違点もあります。例えば、i ノード自体ではなく、ファイルに関するメタデータを格納し、前記ファイルの最初のブロックを直接参照するディレクトリエントリであり、ハードリンクの作成を不可能にします。これらの詳細については、Brouwer [B02] を参照してください。

もちろん、inode 設計の空間では、他にも多くの可能性があります。結局のところ、inode は単なるデータ構造であり、関連する情報を格納し、それを効果的に参照できるデータ構造であれば十分です。ファイルシステムソフトウェアは容易く変更されるので、仕事量やテクノロジが変化した場合、さまざまな設計を検討してください。

Most files are small

Average file size is growing

Most bytes are stored in large files

File systems contains lots of files

File systems are roughly half full

Directories are typically small

Roughly 2K is the most common size

Almost 200K is the average

A few big files use most of the space

Almost 100K on average

Even as disks grow, file systems remain ~50% full

Many have few entries; most have 20 or fewer

Figure 40.2: File System Measurement Summary

40.4 Directory Organization

vsfs(多くのファイルシステムのように) では、ディレクトリは単純な構成です。ディレクトリは基本的に(エントリ名、i ノード番号) のペアのリストを含んでいます。特定のディレクトリ内のファイルまたはディレクトリごとに、そのディレクトリのデータブロックに文字列と数字があります。各文字列には、長さもあります(可変サイズの名前を仮定)。

たとえば、ディレクトリ dir(i ノード番号 5) に 3 つのファイル (foo、bar、および foobar_is_a.pretty_longname) があり、それぞれ inode 番号 12,13,24 であるとします。dir のディスク上のデータは次のようにになります。

inum	recflen	strlen	name
5	12	2	.
2	12	3	..
12	12	4	foo
13	12	4	bar
24	36	28	foobar_is_a.pretty_longname

この例では、各エントリには、inode 番号、レコード長（名前の合計バイト数、残り余白の合計バイト数）、文字列の長さ（実際の名前の長さ）、および最後にエントリの名前があります。各ディレクトリに 2 つの余分なエントリがあることに注意してください。“.”（ドット）と“..”（ドットドット）です。ドットディレクトリは現在のディレクトリ（この例では dir）ですが、ドットドットは親ディレクトリ（この場合はルート）です。

ファイルを削除すると（例えば `unlink()` を呼び出す）、ディレクトリの途中に空きスペースが残る可能性があります。そのため、（ゼロなどの予約済みの i ノード番号などで）同様にマークする方法が必要です。このような削除は、レコードの長さが使用される理由の 1 つです。より大きなエントリや余分なスペースがあった場合、新しいエントリは古いエントリを再利用し、使用する可能性があります。

正確にディレクトリがどこに格納されているのか疑問に思うかもしれません。よくファイルシステムはディレクトリを特別なタイプのファイルとして扱います。したがって、ディレクトリには inode テーブルのどこかに inode があります（inode の type フィールドは “regular file” ではなく “directory” とマークされています）。ディレクトリには、inode が指すデータブロック（およびおそらく間接ブロック）があります。これらのデータブロックは、シンプルなファイルシステムのデータブロック領域に存在します。したがって、ディスク上の構造は変わりません。

また、ディレクトリエントリのこの単純な線形リストは、そのような情報を格納する唯一の方法ではないことに再度注意する必要があります。前述のように、任意のデータ構造が可能です。たとえば、XFS [S + 96] はディレクトリを B ツリー形式で保存し、全体をスキャンする必要がある単純なリストを持つシステムよりも高速にファイルを作成する操作します（ただし、ファイル名を作成する前にファイル名が使用されていないことを保証する必要があります）。

40.5 Free Space Management

ファイルシステムは、inode とデータブロックが空いているかどうかを追跡する必要があります。もし見つからなかったとき、新しいファイルやディレクトリに割り当てるような領域を見つける必要があります。したがって、空き領域の管理はすべてのファイルシステムにとって重要です。vsfs には、このタスクのための 2 つの単純なビットマップがあります。

ASIDE: FREE SPACE MANAGEMENT

空き領域を管理する多くの方法があります。ビットマップは単なる 1 つの方法です。一部の初期のファイルシステムでは、スーパーブロック内の单一のポインタが最初の空きブロックを保持していました。その空きブロックの中で次の空きブロックのポインタを持ち、システムの空きブロックを介してリストが形成されました。ブロックが必要になったときに、ヘッドブロックが使用され、それに応じてリストが更新されました。

最新のファイルシステムは、より洗練されたデータ構造を使用します。たとえば、SGI の XFS [S + 96] は、ディスクのどのチャンクが空いているかをコンパクトに表現するために、何らかの形の B ツリーを使用します。どのようなデータ構造でも、異なるタイムスペースのトレードオフが可能です。

たとえば、ファイルを作成する場合、そのファイルに i ノードを割り当てる必要があります。したがって、ファイルシステムは、空いている inode のビットマップを検索し、それをファイルに割り当てます。ファイルシステムは inode を使用中で（1 で）マークし、最終的にディスク上のビットマップを正しい情報で更新する必要があります。同様の一連のアクティビティが、データブロックが割り当てられるときに行われます。

新しいファイルにデータブロックを割り当てるときには、他のいくつかの考慮事項が有効になります。たとえば、ext2 や ext3 などの Linux ファイルシステムの中には、新しいファイルが作成され、データブロックが必要なときに解放される一連のブロック（たとえば 8）があります。そのような一連の空きブロックを見つけて

新しく作成したファイルに割り当てることで、ファイルシステムはファイルの一部がディスク上で連続していることを保証し、パフォーマンスを向上させます。したがって、このような事前割り当てポリシーは、データブロックのためのスペースを割り当てる際に一般的に使用されるヒューリスティックです。

40.6 Access Paths: Reading and Writing

ファイルとディレクトリがディスクにどのように格納されているかを知ったので、ファイルの読み書きの動作中に操作の流れに沿うことができます。したがって、このアクセスパスで何が起こるかを理解することは、ファイルシステムの仕組みを理解する上での第 2 の鍵です。注意を払いましょう！

以下の例では、ファイルシステムがマウントされており、スーパー ブロックがすでにメモリに入っていると仮定します。他のすべてのもの(つまり、inode、ディレクトリ)はまだディスク上にあります。

Reading A File From Disk

この簡単な例では、まずファイル(例: /foo/bar)を開いて読み込み、それを閉じたいとします。この単純な例では、ファイルのサイズがちょうど 4KB(つまり 1 ブロック)であると仮定します。

`open("/foo/bar", O_RDONLY)` 呼び出しを発行するとき、ファイルシステムは最初に bar ファイルの inode を見つけ、ファイルに関する基本情報(アクセス許可情報、ファイルサイズなど)を取得する必要があります。そのためには、ファイルシステムは inode を見つけることができなければなりませんが、今のところ完全なパス名しか存在しません。ファイルシステムはパス名を探し(トランザクション)、目的の i ノードを特定する必要があります。

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
<code>open(bar)</code>				<code>read</code>				<code>read</code>		
					<code>read</code>			<code>read</code>		
					<code>read</code>				<code>read</code>	
<code>read()</code>					<code>write</code>				<code>read</code>	
					<code>read</code>				<code>read</code>	
<code>read()</code>					<code>write</code>				<code>read</code>	
					<code>read</code>				<code>read</code>	
<code>read()</code>					<code>write</code>					<code>read</code>

Figure 40.3: File Read Timeline (Time Increasing Downward)

すべてのトランザクションは、/と呼ばれるルートディレクトリのファイルシステムのルートから始まります。したがって、FS がディスクから読み込む最初のことは、ルートディレクトリの i ノードです。しかし、この inode はどこですか？ i ノードを見つけるには、その i number(i 番号)を知っていなければなりません。通常、ファイルまたはディレクトリの i number(i 番号)は親ディレクトリにあります。ルートには親がありません(定義によって)。したがって、ルートの inode 番号は「よく知られている」必要があります。FS は、ファイルシステムがマウントされているときに、その内容を知る必要があります。ほとんどの UNIX ファイルシステムでは、ルートの i ノード番号は 2 です。したがって、プロセスを開始するために、FS は i ノード番号 2(最初の i ノードブロック)を含むブロックを読み取ります。

inode が読み込まれると、FS は内部を調べて、ルートディレクトリの内容を含むデータブロックへのポイ

ンタを見つけることができます。したがって、FS はこれらのディスク上のポインタを使用してディレクトリを読み込みます。この場合、foo のエントリを探します。1つまたは複数のディレクトリデータブロックを読み込むことによって、foo のエントリが見つかります。一度検出されると、FS は、次に必要となる foo の i ノード番号 (44 と言う) も検出します。

次のステップは、目的の inode が見つかるまでパス名を再帰的に走査することです。この例では、FS は foo の i ノードを含むブロックとそのディレクトリデータを読み取り、最終的に bar の i ノード番号を検出します。`open()` の最後のステップは、bar の inode をメモリに読み込むことです。FS は最終的なパーミッションチェックを行い、このプロセスのファイルディスククリプタをプロセス単位のオープンファイルテーブルに割り当て、それをユーザに返します。

オープンすると、プログラムは `read()` システムコールを発行してファイルから読み込むことができます。最初の読み込み (`lseek()` が呼び出されていなければオフセット 0) は、ファイルの最初のブロックを読み込み、inode を参照してそのようなブロックの位置を探します。新しい最後のアクセス時間で i ノードを更新することもできます。この読み込みは、このファイル記述子のメモリ内オープンファイルテーブルをさらに更新し、次の読み込みが第 2 のファイルブロックなどを読み込むようにファイルオフセットを更新します。

ASIDE: READS DON'T ACCESS ALLOCATION STRUCTURES

私たちは、多くの学生がビットマップなどの割り当て構造によって混乱するのを見てきました。特に、ファイルを読み込んで、新しいブロックを割り当てないとき、ビットマップに引き続き参照されていると多くの人は考えていました。しかし、それは本当ではありません。ビットマップなどの割り当て構造は、割り当てが必要な場合にのみアクセスされます。inode、ディレクトリ、および間接ブロックには、読み取り要求を完了するために必要なすべての情報が含まれています。つまり、inode が既にそれ (割り当てられているブロック) を指しているときに、ブロックが割り当てられていることを確認する必要はありません。

ある時点でファイルが閉じられます。ここでやるべき仕事ははるかに少ない。明らかに、ファイルディスククリプタの割り当てを解除する必要がありますが、今のところそれは FS が本当に必要とするものです。ディスク I/O は行われません。

このプロセス全体を図 40.3 に示します (時間が長くなる)。この図では、`open` によってファイルの inode を最終的に見つけるために多数の読み込みが行われます。その後、各ブロックを読み込むには、まずファイルシステムが inode を参照してからブロックを読み込み、次に inode の最終アクセス時のフィールドを書き込みで更新する必要があります。時間をかけて、何が起こっているのか理解しましょう。

また、オープンによって生成される I/O の量は、パス名の長さに比例することにも注意してください。パス内の追加ディレクトリごとに、その inode とそのデータを読み込む必要があります。これを悪化させると大きなディレクトリが存在することになります。ここでは、ディレクトリの内容を取得するために 1 つのブロックだけを読み取る必要がありますが、大きなディレクトリでは、目的のエントリを見つけるために多くのデータブロックを読み取る必要があります。つまり、ファイルを読むときにかなり悪くなることがあります。あなたが見つけたいと思うように、ファイルを書き出す (そして特に新しいものを作る) ことはさらに悪いことです。

Writing to Disk

ファイルへの書き込みも同様のプロセスです。まず、ファイルを開く必要があります (上記のように)。次に、アプリケーションは新しいコンテンツでファイルを更新するために `write()` 呼び出しを発行できます。最後に、ファイルが閉じられます。

読み込みとは異なり、ファイルへの書き込みはブロックを割り当てるこもできます (たとえば、ブロックが上書きされている場合を除きます)。新しいファイルを書き出すとき、各書き込みはディスクにデータを書き込むだけでなく、どのブロックをファイルに割り当てるかを最初に決定し、それに応じてディスクの他の構造

(例えば、データビットマップおよび i ノード) を更新する必要がある。したがって、ファイルへの各書き込みは、論理的に 5 つの I/O を生成します。1 つはデータビットマップを読み込み(新しく割り当てられたブロックを使用するように更新する)、1 つはビットマップを書き込み(新しい状態をディスクに反映させる)、2 つ以上のブロック読み込んだ後に inode(新しいブロックの場所で更新されます) に書き込み、最後に実際のブロック自体を書き込む 1 つのブロックです。

書き込みトラフィックの量は、ファイル作成などの単純で一般的な操作を考慮するとさらに悪化します。ファイルを作成するには、ファイルシステムは i ノードを割り当てるだけでなく、新しいファイルを含むディレクトリ内に領域を割り当てる必要があります。inode ビットマップへの読み込み(空き inode を見つける)、inode ビットマップへの書き込み(割り当て済みとしてマークする)、新しい inode 自体への書き込み(その初期化)、ディレクトリのデータに 1 つ(ファイルの高いレベルの名前をその i ノード番号にリンクするため)、ディレクトリの inode を読み書きして更新をします。新しいエントリを収容するためにディレクトリを拡張する必要がある場合、追加の I/O(すなわち、データビットマップおよび新しいディレクトリブロック)も必要になります。これら全てはただファイルを作成するだけのためです。

ファイル /foo/bar が作成され、3 つのブロックが書き込まれる特定の例を見てみましょう。図 40.4 は、open()(ファイルを作成する) と 3 回の 4KB 書き込みがそれぞれの間に何が起こるかを示しています。

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
			read				read			
create (/foo/bar)		read write			read write			read		
					read write				write	
						read				
write()		read write						write		
					write				write	
write()		read write			read					write
						write				
write()		read write							write	
					write read					
						write				

Figure 40.4: File Creation Timeline (Time Increasing Downward)

この図では、ディスクへの読み込みと書き込みがグループ化され、その中でシステムコールが発生したときに発生します。発生する可能性のある大雑把な順序は、図の上から下に向かっています。ファイルを作成するにはどれくらいの作業が必要ですか？この場合は 10 I/O です、パス名を探して最終的にファイルを作成します。また、各割り当て書き込みには 5 つの I/O が必要であることがわかります。つまり、i ノードを読み込んで更新するペアと、データビットマップを読み込んで更新するペアと、最後にデータ自体の書き込みです。ど

のようにしてファイルシステムはこれを妥当な効率で達成できますか？

THE CRUX: HOW TO REDUCE FILE SYSTEM I/O COSTS

ファイルのオープン、読み取り、書き込みなどの操作が最も単純な場合でも、ディスク上に散在する膨大な I/O 操作が発生します。非常に多くの I/O を実行するための高いコストを削減するために、ファイルシステムは何をすることができますか？

40.7 Caching and Buffering

上記の例に示すように、ファイルの読み取りと書き込みは高価になり、(遅い) ディスクに対して多くの I/O が発生します。明らかに大きなパフォーマンス上の問題を改善するために、ほとんどのファイルシステムは、システムメモリ (DRAM) を積極的に使用して重要なブロックをキャッシュします。

キャッシュを使用しない上記の例を想像してみましょう。キャッシュを使用しないと、すべてのファイルを開くには、ディレクトリ階層内の各レベル (少なくとも 1 つはそのディレクトリの inode を読み込み、データを読み込むには少なくとも 2 回) が必要です。長いパス名 (たとえば、`1/2/3/.../100/file.txt`) では、ファイルシステムは文字通りファイルを開くために何百もの読み込みを実行します。

初期のファイルシステムでは、一般的なブロックを保持するために固定サイズのキャッシュが導入されました。仮想メモリの議論のように、LRU やさまざまなバリエントなどの戦略によって、どのブロックをキャッシュに保持するかが決定されます。この固定サイズのキャッシュは通常、起動時にメモリの約 10 % になるように割り当てられます。

しかし、このメモリの静的な partitioning は無駄です。ファイルシステムが特定の時点で 10 % のメモリを必要としない場合はどうなりますか？ 上述した固定サイズのアプローチでは、ファイルキャッシュ内の未使用ページを他の用途のために再利用することができず、したがって無駄になる。これとは対照的に、現代のシステムでは、動的 partitioning 手法が採用されています。具体的には、最新のオペレーティングシステムの多くは、仮想メモリページとファイルシステムページを統合ページキャッシュ [S00] に統合しています。このようにして、与えられた時間に多くのメモリを必要とするものに応じて、仮想メモリとファイルシステムの間でより柔軟にメモリを割り当てることができます。

今度は、キャッシュを使ったファイルオープンの例を想像してみましょう。最初のオープンでは、ディレクトリの inode とデータを読み込むために大量の I/O トラフィックが生成されることがあります、同じファイル (または同じディレクトリ内のファイル) の後続のファイルが大部分キャッシュに書き込まれるため、I/O は必要ありません。

キャッシュへの書き込みに対する影響についても考慮してみましょう。読み取り I/O は十分に大きなキャッシュで回避することができますが、書き込みトラフィックは永続的になるためにディスクに移動する必要があります。したがって、キャッシュは、読み取りのために行うフィルタでは、書き込みトラフィックのフィルタとして機能しません。つまり、別のものが必要です。それは書き込みバッファリング (ときどき呼び出されるように) です。これには、多くのパフォーマンス上の利点があります。まず、書き込みを遅らせることによって、ファイルシステムはいくつかの更新のより小さな I/O セットをバッチすることができます。たとえば、もし、1 つのファイルが作成され、別のファイルが作成されるとすぐに更新されたときに、i ノードのビットマップが更新された場合、ファイルシステムは最初の更新後に書き込みを遅らせることによって I/O を保存します。第 2 に、多数の書き込みをメモリにバッファリングすることによって、システムは後続の I/O をスキューリングし、パフォーマンスを向上させることができます。

最後に、いくつかの書き込みはそれらを遅らせることで完全に避けられます。たとえば、アプリケーションがファイルを作成してから削除すると、ファイルの作成をディスクに反映させるために書き込みを遅せると、完全にそれら (書き込み) を回避します。この場合、怠惰 (ディスクへの書き込みブロックすること) は徳です。

TIP: UNDERSTAND STATIC VS. DYNAMIC PARTITIONING

異なるクライアント/ユーザー間でリソースを分割する場合は、静的パーティショニングまたは動的パーティショニングを使用できます。静的アプローチでは、単にリソースを固定された割合に1回だけ分割します。たとえば、メモリの利用者が2人いる場合、あるユーザーに固定量のメモリを割り当て、もう一人のユーザーには残りを割り当てます。ダイナミックなアプローチは柔軟性があり、時間の経過とともにリソースの量が異なる。たとえば、1人のユーザーが一定期間ディスク帯域幅のパーセンテージを高くすることができますが、その後、システムが切り替わり、別のユーザーに使用可能なディスク帯域幅のより大きな部分を与えることを決定するかもしれません。

それぞれのアプローチには利点があります。静的パーティショニングにより、各ユーザーはリソースの一部を受け取ることができ、通常は予測可能なパフォーマンスが向上し、実装がより簡単になります。動的パーティショニングは、(リソースを必要とするユーザーにアイドル状態のリソースを消費させることで)より優れた使用率を達成できますが、実装がより複雑になる可能性があり、アイドル状態のリソースが他のユーザーによって消費された後、必要な時に再び使うのに時間がかかります。よくあることですが、最良の方法はありません。むしろ、当面の問題について考える必要があります、どちらのアプローチが最も適切かを判断する必要があります。

上記の理由から、現代のファイルシステムの大部分は、5秒から30秒の間のどこかでメモリに書き込みをバッファリングします。これは、もう一つのトレードオフを表します。アップデートがディスクに伝播される前にシステムがクラッシュした場合、そのアップデートは失われます。ただし、書き込みを長時間メモリに保存することで、バッチ処理、スケジューリング、さらには書き込みの回避によってもパフォーマンスを向上させることができます。

一部のアプリケーション(データベースなど)では、このトレードオフが楽しめません。したがって、書き込みバッファリングによる予期せぬデータ損失を避けるため、`fsync()`を呼び出すか、キャッシングを回避するダイレクトI/Oインターフェースを使用するか、またはrawディスクインターフェースを使用してファイルシステム全体を回避します。ほとんどのアプリケーションはファイルシステムのトレードオフがありますが、デフォルトで満足させることができない場合は、システムが望むように動作するのに十分なコントロールがあります。

TIP: UNDERSTAND THE DURABILITY/PERFORMANCE TRADE-OFF

ストレージシステムは、多くの場合、ユーザーに耐久性とパフォーマンスのトレードオフをもたらします。ユーザが直ちに耐久性があるように書かれたデータを望む場合、システムは新しく書き込まれたデータをディスクにコミットする必要があり、したがって書き込みは遅いです(しかし安全である)。しかし、ユーザが小さなデータの損失を許容することができれば、システムはしばらくの間、書き込みをメモリにバッファリングし、後でディスクに書き込むことができます(バックグラウンドで)。そうすることで、書き込みが素早く完了したように見えるため、パフォーマンスが向上します。ただし、クラッシュが発生した場合、まだディスクにコミットされていない書き込みは失われ、その結果、トレードオフが発生します。このトレードオフを適切に行う方法を理解するには、ストレージシステムを使用するアプリケーションが必要とするものを理解することが最善です。たとえば、Webブラウザでダウンロードした最後の数枚のイメージを失うことは許容されるかもしれません、銀行口座に資金を追加しているデータベーストランザクションの一部が失うことは許容できないかもしれません。あなたが金持ちでない限りそうでしょう。

40.8 Summary

私たちはファイルシステムを構築するのに必要な基本的な機械を見てきました。各ファイル(メタデータ)についての情報が必要です。通常、inodeという構造体に格納されます。ディレクトリは、名前→inode番号の

マッピングを格納する特定の種類のファイルです。そして、他の構造も必要です。たとえば、ファイルシステムでは、ビットマップなどの構造を使用して、どの inode またはデータブロックが空いているか、割り当てられているかを追跡することがよくあります。

ファイルシステム設計の素晴らしい点は、その自由性です。次の章で調べるファイルシステムは、この自由性を利用してファイルシステムの一部の側面を最適化します。私たちが未知のまま残した多くのポリシーの決定も明らかにあります。たとえば、新しいファイルが作成されると、そのファイルをディスク上のどこに配置する必要がありますか？この方針と今後の方針については、今後の章の対象となります。

参考文献

- [A+07] Nitin Agrawal, William J. Bolosky, John R. Douceur, Jacob R. Lorch
A Five-Year Study of File-System Metadata
FAST '07, pages 31–45, February 2007, San Jose, CA
An excellent recent analysis of how file systems are actually used. Use the bibliography within to follow the trail of file-system analysis papers back to the early 1980s.
- [B07] “ZFS: The Last Word in File Systems”
Jeff Bonwick and Bill Moore
Available: http://www.ostep.org/Citations/zfs_last.pdf
One of the most recent important file systems, full of features and awesomeness. We should have a chapter on it, and perhaps soon will.
- [B02] “The FAT File System”
Andries Brouwer
September, 2002
Available: <http://www.win.tue.nl/~aeb/linux/fs/fat/fat.html>
A nice clean description of FAT. The file system kind, not the bacon kind. Though you have to admit, bacon fat probably tastes better.
- [C94] “Inside the Windows NT File System”
Helen Custer
Microsoft Press, 1994
A short book about NTFS; there are probably ones with more technical details elsewhere.
- [H+88] “Scale and Performance in a Distributed File System”
John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, Michael J. West.
ACM Transactions on Computing Systems (ACM TOCS), page 51-81, Volume 6, Number 1, February 1988 A classic distributed file system; we'll be learning more about it later, don't worry.
- [P09] “The Second Extended File System: Internal Layout”
Dave Poirier, 2009
Available: <http://www.nongnu.org/ext2-doc/ext2.html>
Some details on ext2, a very simple Linux file system based on FFS, the Berkeley Fast File System. We'll be reading about it in the next chapter.
- [RT74] “The UNIX Time-Sharing System”
M. Ritchie and K. Thompson
CACM, Volume 17:7, pages 365-375, 1974
The original paper about UNIX. Read it to see the underpinnings of much of modern operating systems.

[S00] “UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD”

Chuck Silvers

FREENIX, 2000

A nice paper about NetBSD’s integration of file-system buffer caching and the virtual-memory page cache. Many other systems do the same type of thing.

[S+96] “Scalability in the XFS File System”

Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson,

Mike Nishimoto, Geoff Peck

USENIX ’96, January 1996, San Diego, CA

The first attempt to make scalability of operations, including things like having millions of files in a directory, a central focus. A great example of pushing an idea to the extreme. The key idea behind this file system: everything is a tree. We should have a chapter on this file system too.

41 Locality and The Fast File System

UNIX オペレーティングシステムが初めて導入されたとき、UNIX ウィザードの Ken Thompson 氏が最初のファイルシステムを作成しました。“古い UNIX ファイルシステム”と呼ぶことにしましょう。それは本当に簡単でした。基本的に、そのデータ構造はディスク上で次のようになりました。



スーパーブロック (S) には、ボリュームの大きさ、inode の数、ブロックの空きリストの先頭へのポインタなど、ファイルシステム全体に関する情報が含まれていました。ディスクの i ノード領域には、ファイルシステムのすべての i ノードが含まれていました。最後に、ディスクの大部分がデータブロックに取り込まれました。

古いファイルシステムについての良いことは、単純であり、ファイルシステムが提供しようとしていた基本的な抽象化、すなわちファイルとディレクトリ階層をサポートしていることでした。この使いやすいシステムは、過去の不器用なレコードベースのストレージシステムからの真の前進でした。ディレクトリ階層は、以前のシステムで提供されていたよりシンプルな 1 レベル階層以上の真の進歩でした。

41.1 The Problem: Poor Performance

問題：パフォーマンスはひどいものです。Berkeley [MJLF84] の Kirk McKusick とその同僚たちが測定したように、パフォーマンスは悪くなり、ファイルシステムが全体のディスク帯域幅のわずか 2 % というところまで悪化しました。

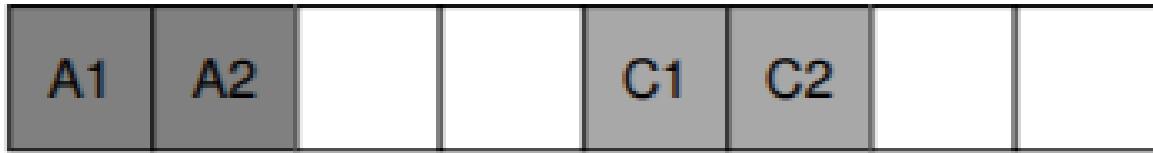
主な問題は、古い UNIX ファイルシステムがランダムアクセスメモリのようにディスクを処理していたことです。データを保持する媒体がディスクであることに関係なく、データは場所全体に広がっていたため、現実的で高価な位置決めコストが発生していました。例えば、あるファイルのデータブロックは、しばしば inode から非常に遠く離れているため、最初に inode を読み込み、次にファイルのデータブロックを読み込むときには高価なシークを引き起こします（かなり一般的な操作です）。

さらに悪いことに、空き領域が慎重に管理されていないため、ファイルシステムはかなり断片化してしまいます。フリーリストは、ディスク全体に散らばっているブロックを指し、ファイルが割り当てられると、次の空きブロックを取るだけです。その結果、論理的に連続したファイルには、ディスク全体を行き来してアクセスすることができ、パフォーマンスが大幅に低下しました。

たとえば、次のデータブロック領域を想定します。この領域には、サイズが 2 ブロックのそれぞれに 4 つのファイル (A、B、C、D) が含まれています。



B と D が削除された場合、結果のレイアウトは次のようにになります。



見て分かるように、空き領域は 4 つの連続チャンクではなく、1 つのブロックの 2 つのチャンクごとにそれぞれ分割されています。たとえば、サイズが 4 ブロックのファイル E を割り当てたいとします。



何が起こるかを見ることができます。E はディスク全体に広がってしまい、その結果、E にアクセスすると、ディスクからピーク（シーケンシャル）なパフォーマンスが得られません。むしろ、まず E1 と E2 を読んでから、シークしてから E3 と E4 を読み込みます。この断片化の問題は、古い UNIX ファイルシステムでは常に発生し、パフォーマンスが低下します。副題：この問題は、ディスクデフラグツールが役立つものです。オンラインディスクデータを再編成してファイルを連続して配置し、連続した 1 つまたは複数の領域に空き領域を作ったり、データを移動したり、i ノードなどを書き換えたりして変更を反映させます。

もう 1 つの問題：元のブロックサイズが小さすぎます (512 バイト)。従って、ディスクからデータを転送することは、本質的に非効率的でした。小さなブロックは内部断片化（ブロック内の無駄）を最小限に抑えたため良好でしたが、各ブロックに到達するためにはオーバーヘッドが必要な場合があったため、転送には問題がありました。したがって、問題：

THE CRUX: HOW TO ORGANIZE ON-DISK DATA TO IMPROVE PERFORMANCE
パフォーマンスを向上させるためにファイルシステムのデータ構造をどのように整理できますか？
これらのデータ構造に加えて、どのようなタイプの割り当てポリシーが必要ですか？ ファイルシステムを「ディスク対応」にするにはどうすればよいですか？

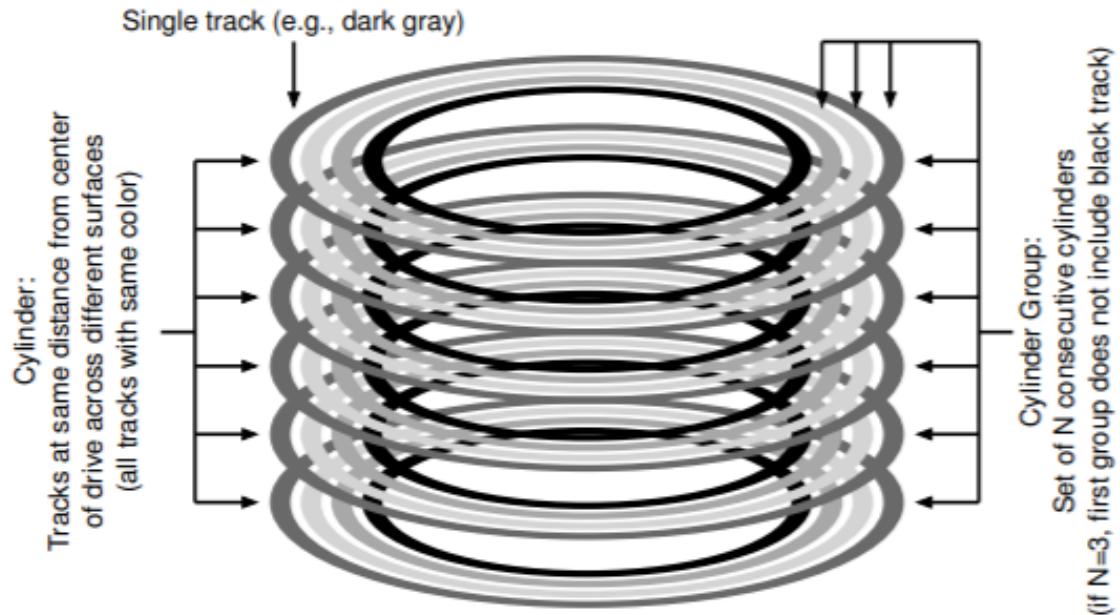
41.2 FFS: Disk Awareness Is The Solution

Berkeley のあるグループは、より高速でより高速なファイルシステムを構築することを決めました。そのファイルシステムは、巧みに Fast File System(FFS) と呼ばれていました。この考え方は、ファイルシステムの構造と割り当てポリシーを「ディスクを意識した」ものにすることでパフォーマンスを向上させることでした。FFS はファイルシステム研究の新しい時代を迎えるました。（`open()`、`read()`、`write()`、`close()`、および、その他の同じ API であるファイルシステムコールを含む）同じファイルシステムに維持しながら、内部実装を変更することによって、新しいファイルシステムの構築を今日まで続けています。事実上、最新のファイルシステムはすべて、パフォーマンス、信頼性、その他の理由から内部を変更しながら、既存のインターフェイスに準拠している（したがって、アプリケーションとの互換性を維持します）。

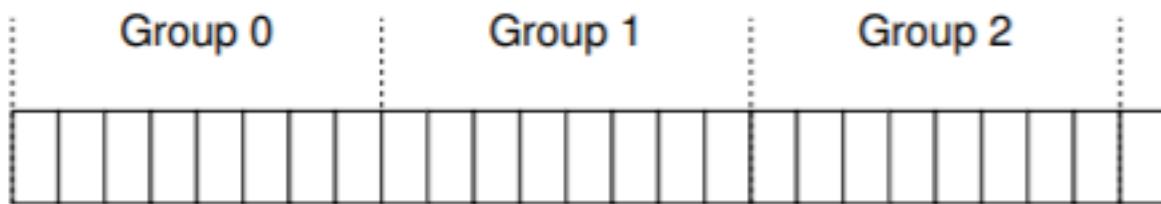
41.3 Organizing Structure: The Cylinder Group

最初のステップは、ディスク上の構造を変更することでした。FFS は、ディスクを複数のシリンドルグループに分割します。1 つのシリンドルは、ドライブの中央から同じ距離にあるハードドライブの異なる面にある一連のトラックです。いわゆる幾何学的形状とは明らかに類似しているため、円筒と呼ばれています。FFS は、N

個の連続したシリンダをグループに集約するので、ディスク全体はシリンダグループの集合として見ることができます。ここには、6つのプラッターを持つドライブの4つあるうちの最も外側のトラックから3つのシリンダーで構成されるシリンダー・グループを示す簡単な例があります。(今回の場合 $N=3$ であるため、黒色のトラックはシリンダーに含まれません)



最新のドライブは、特定のシリンダが使用されているかどうかを本当に理解するためにファイルシステムに十分な情報をエクスポートしないことに注意してください。前述の [AD14a] のように、ディスクはブロックの論理アドレス空間をエクスポートし、ジオメトリの詳細をクライアントから隠します。したがって、現代のファイルシステム (Linux ext2、ext3、ext4など) ではなく、ドライブをブロックグループに編成します。各ブロックグループは、ディスクのアドレススペースのちょうど連続した部分です。下の図は、8つのブロックがすべて異なるブロックグループに編成されている例を示しています (実際のグループはもっと多くのブロックで構成されています)。



シリンダーグループまたはブロックグループと呼ばれても、FFS がパフォーマンスを向上させるために使用する中心的なメカニズムです。重大なことに、同じグループ内に2つのファイルを配置することで、FFS は、次々にアクセスすることによって、ディスク全体で長くシークを発生させることはありません。

これらのグループを使用してファイルとディレクトリを格納するには、ファイルとディレクトリをグループに配置し、そこに必要なすべての情報を追跡する必要があります。そうするために、FFS には、inode、データブロック、およびそれらのそれぞれが割り当てられているか解放されているかを追跡するいくつかの構造体のスペースなど、ファイルシステムが各グループ内に持つと考えられるすべての構造が含まれています。ここでは、FFS が単一のシリンダグループ内に保持するものを示しています。



ここで、この単一のシリンドグループのコンポーネントを詳細に調べてみましょう。FFS は、信頼性の理由からスーパーブロック (S) のコピーを各グループに保持します。スーパーブロックは、ファイルシステムをマウントするために必要です。複数のコピーを保持することで、1 つのコピーが破損した場合でも、動作中のレプリカを使用してファイルシステムをマウントしてアクセスできます。

各グループ内で、FFS はグループの inode とデータブロックが割り当てられているかどうかを追跡する必要があります。グループごとの inode ビットマップ (ib) とデータビットマップ (db) は、各グループの inode とデータブロックに対してこの役割を果たします。ビットマップは、ファイルシステムの空き領域を管理する優れた方法です。古いファイルシステムの空きリストの断片化の問題を避けるために、大きな空き領域を見つけてファイルに割り当てるのが簡単だからです。

最後に、i ノードとデータブロック領域は、以前の非常に単純なファイルシステム (VSFS) の領域とまったく同じです。通常、各シリンドグループのほとんどはデータブロックで構成されています。

ASIDE: FFS FILE CREATION たとえば、ファイルの作成時に更新するデータ構造を考えてみましょう。この例では、ユーザーが新しいファイル /foo/bar.txt を作成し、ファイルが 1 ブロック (4KB) であると仮定します。このファイルは新しいもので、新しい inode が必要です。したがって、inode ビットマップと新しく割り当てられた i ノードの両方がディスクに書き込まれます。ファイルにはデータも含まれているため、割り当ても必要です。データビットマップとデータブロックはこうして (最終的に) ディスクに書き込まれます。したがって、現在のシリンドグループへの少なくとも 4 回の書き込みが行われます (これらの書き込みは、発生前に一時的にメモリにバッファリングされる可能性があります)。ただこれが全てではありません！特に、新しいファイルを作成するときには、そのファイルをファイルシステム階層に配置する必要があります。つまり、ディレクトリを更新する必要があります。具体的には、親ディレクトリ foo を更新して bar.txt のエントリを追加する必要があります。この更新は、foo の既存のデータブロックに収まるか、または関連するデータビットマップとともに新しいブロックを割り当てる必要があります。ディレクトリの新しい長さを反映し、時間フィールドを更新するために (last modified time などの) foo の inode も更新する必要があります。これらが、新しいファイルを作成するだけの全作業です！

41.4 Policies: How To Allocate Files and Directories

このグループ構造を導入した FFS は、パフォーマンスを向上させるために、ファイルとディレクトリおよび関連メタデータをディスクに配置する方法を決定する必要があります。基本的なマントラ (真言) はシンプルです。関連するものを一緒にしておくことです (そしてその結果として、関連性のないものを遠くに保ちます)。

したがって、マントラに従うためには、FFS は何が「関連する」ものであるかを決定し、それを同じブロックグループ内に置かなければいけません。反対に、関係のないアイテムは異なるブロックグループに配置する必要があります。この目的を達成するために、FFS はいくつかの簡単な配置ヒューリスティックを利用します。

最初はディレクトリの配置です。FFS は単純なアプローチを採用しています。割り当てられたディレクトリの数が少ない (グループ間でディレクトリのバランスを調整する) シリンダ数と、空き inode 数が多いシリンドグループ (その後にたくさんのファイルを割り当てることができます) を見つけて、ディレクトリデータと i ノードをグループに置きます。もちろん、他のヒューリスティックをここで使用することができます (例えば、

空きデータブロックの数を考慮に入れて)。ファイルの場合、FFS は 2 つのことを行います。まず(一般的なケースでは)、ファイルのデータブロックを inode と同じグループに割り当てて、(古いファイルシステムのように)inode とデータの間の長いシークを防ぎます。

次に、同じディレクトリーにあるすべてのファイルを、それらが入っているディレクトリーのシリンドー・グループに入れます。つまり、ユーザーが、/a/b、/a/c、/a/d、/b/f の 4 つのファイルを作成すると、FFS は、最初の 3 つを互いに近く(同じグループ)に配置し、4 つ目を遠くに(他のグループに)配置しようとします。このような割り当ての例を見てみましょう。この例では、各グループに 10 個の inode と 10 個のデータブロック(非現実的に小さな番号)があり、3 つのディレクトリ(ルートディレクトリ /、/a、/b) と 4 つのファイル(/a/c、/a/d、/a/e、/b/f) は、FFS ポリシーごとに配置されます。通常のファイルはそれぞれ 2 ブロックのサイズであり、ディレクトリにはただ 1 ブロックのデータしかないものとします。この図では、各ファイルまたはディレクトリ(/ はルートディレクトリ、/a は a、/b/f は f など)に明瞭な記号を使用しています。

group	inodes	data
0	/-----	/-----
1	acde-----	accddee---
2	bf-----	bff-----
3	-----	-----
4	-----	-----
5	-----	-----
6	-----	-----
7	-----	-----
■ ■ ■		

FFS ポリシーは 2 つのポジティブなことを行います。FFS ポリシーは、各ファイルのデータブロックが各ファイルの i ノードの近くにあり、同じディレクトリ内のファイルが互いに近くにあることです。(すなわち、/a/c、/a/d、/a/e はグループ 1 にすべてあり、ディレクトリ/b とそのファイル/b/f はグループ 2 で互いに近くにあります) これとは対照的に、グループ全体で i ノードを広げ、グループの i ノードテーブルがすぐにいっぱいにならないようにしようとする i ノード割り当てポリシーを見てみましょう。したがって、最終的な割り当ては次のようになります。

group	inodes	data
0	/-----	/-----
1	a-----	a-----
2	b-----	b-----
3	c-----	cc-----
4	d-----	dd-----
5	e-----	ee-----
6	f-----	ff-----
7	-----	-----
■ ■ ■		

図からわかるように、このポリシーは実際にファイル(およびディレクトリ)データをそれぞれのiノードの近くに保ちますが、ディレクトリ内のファイルは任意にディスクの周りに広がっているため、名前ベースの場所は保持されません。ファイル/a/c, /a/d, /a/eへのアクセスは、FFSアプローチのように1つではなく3つのグループに分かれています。

FFSポリシーヒューリスティックは、ファイルシステムのトラフィックや、特に微妙なものについての広範な調査に基づいているわけではありません。むしろ、彼らは昔ながらの良い常識に基づいています(CSは結局何を意味するのでしょうか?)。ディレクトリ内のファイルは、多くの場合、一緒にアクセスされます。ファイルの束をコンパイルし、それらを单一の実行可能ファイルにリンクすることを想像してください。このような名前空間ベースのローカリティが存在するため、FFSはパフォーマンスを向上させ、関連ファイル間のシークが素敵で短いのです。

41.5 Measuring File Locality

これらのヒューリスティックが意味を成すかどうかをよりよく理解するために、ファイルシステムのアクセスの痕跡を分析し、実際に名前空間の局所性があるかどうかを見てみましょう。なんらかの理由で、このトピックについての良い研究は文献にないようです。

具体的には、SEERトレース[K94]を使用して、ディレクトリツリー内の「遠く離れた」ファイルアクセスが互いにどのように関連していたかを分析します。たとえば、ファイルfを開いた後、トレース内の次のファイルを開くと、ディレクトリツリー内のこれら2つのファイル間の距離はゼロになります(同じファイルであるため)。ディレクトリdir(すなわちdir/f)内のファイルfが開かれ、続いて同じディレクトリ(すなわちdir/g)内のファイルgが開いている場合、2つのファイルアクセス間の距離は1です。同じディレクトリですが、同じファイルではありません。私たちの距離メトリックは、言い換えれば、2つのファイルの共通の祖先を見つけるために移動するディレクトリツリーの距離を測定します。それらがツリーに近づくほど、メトリックは低くなります。

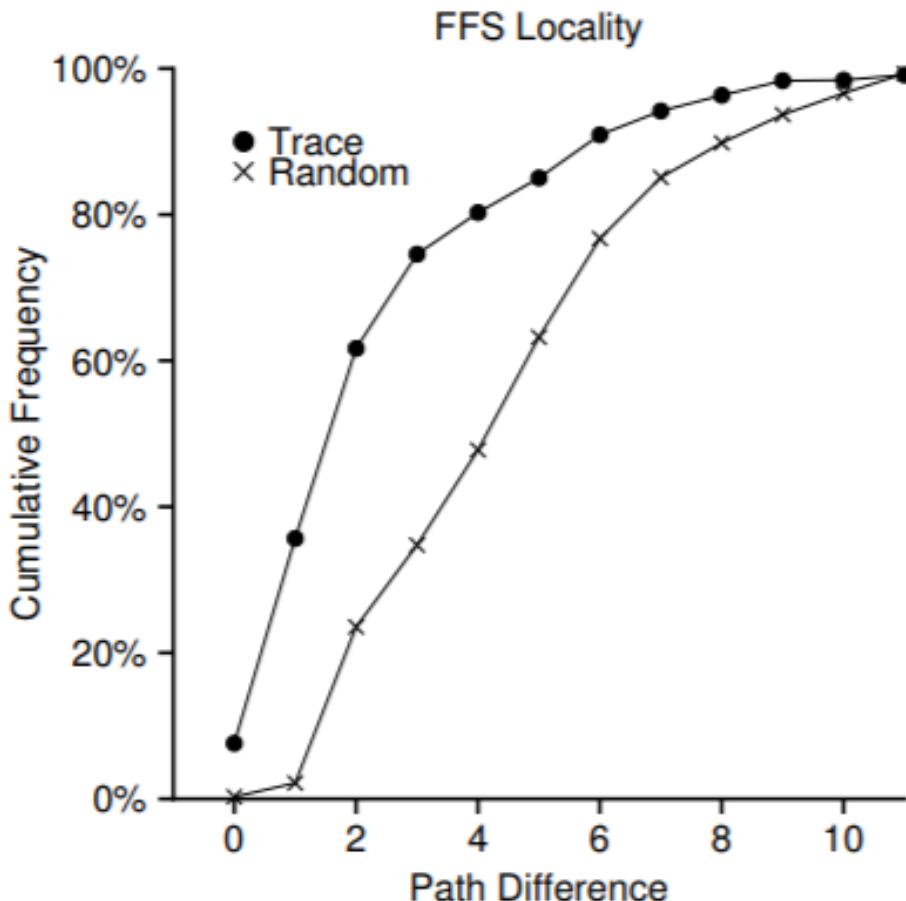


Figure 41.1: FFS Locality For SEER Traces

図 41.1 は、すべてのトレースの全体にわたって SEER クラスタ内のすべてのワークステーションで SEER トレースで観測された地域を示しています。このグラフは、x 軸は差メトリックをプロットし、y 軸はその差異のファイルオープン率の累積パーセントを示しています。具体的には、SEER トレース（グラフの「Trace」と表示）では、以前に開かれたファイルに対するファイルアクセスが約 7 % であり、ファイルアクセスの 40 % 近くが同じファイルまたは同じディレクトリ内の 1 つに（つまり、0 または 1 の差）を返します。したがって、FFS の局所性の仮定は（少なくともこれらの痕跡に対して）意味があります。

興味深いことに、25 % 程度のファイルへのアクセスは、2 の距離を持つファイルに対するものでした。このタイプのローカリティは、ユーザーが複数のレベルの方法で関連ディレクトリのセットを構成し、それらの間で一貫してジャンプするときに発生します。たとえば、ユーザーが src ディレクトリを持ち、オブジェクトファイル (.o ファイル) を obj ディレクトリに構築し、これらのディレクトリが両方ともメインの proj ディレクトリのサブディレクトリである場合、共通のアクセスパターンは proj/src/foo になります.c に続いて proj/obj/foo.o が続きます。proj は共通の祖先であるため、これら 2 つのアクセス間の距離は 2 です。FFS は、そのポリシー内でこのタイプのローカリティを取得しないため、そのようなアクセスの間にさらに悪影響が生じる可能性があります。

比較のため、グラフには「ランダム」トレースのローカリティも表示されます。ランダムトレースは、既存の SEER トレース内からランダムな順序でファイルを選択し、これらのランダムに順序付けられたアクセス間の距離メトリックを計算することによって生成されました。ご覧のように、予想どおり、ランダムトレースには名前空間の局所性が少なくなります。しかし、最終的にはすべてのファイルが共通の祖先（例えば root）を共有し、一部局所性があるため、ランダムは（あくまで）比較する対象としては有用である。

41.6 The Large-File Exception

FFS では、ファイルの配置に関する一般的なポリシーの 1 つの重要な例外があり、大きなファイルの場合に発生します。別の規則がなければ、大きなファイルは最初に配置されたブロックグループ（もしくは別の場所）を完全に埋めるでしょう。このようにブロックグループを充填することは、後続の「関連する」ファイルが、このブロックグループ内に配置されることを防ぐため、ファイルアクセスの局所性を損なう可能性があり、望ましくないです。

したがって、大きなファイルの場合、FFS は次の処理を行います。FFS は、いくつかのブロックが第 1 のブロックグループに割り当てられた後（例えば、12 ブロックまたは i ノード内で利用可能な直接ポインタの数）、ファイルの次の「大きな」チャンク（例えば、第 1 のブロック間接ブロック）を別のブロックグループ（利用率が低いために選択されている可能性があります）に配置します。そのとき、ファイルの次のチャンクは、さらに別のブロックグループに配置されます。

この方針をよりよく理解するためにいくつかの図を見てみましょう。大きなファイルの例外がなければ、1 つの大きなファイルはすべてのブロックをディスクの 1 つの部分に配置します。10 個の inode と 1 グループあたり 40 個のデータブロックで構成された FFS 内の 30 個のブロックを含むファイル（/a）の小さな例を調べます。ここでは、大きなファイルの例外を除いた FFS を示します。

```
group inodes      data
  0 /a----- /aaaaaaaaaa aaaaaaaaaa aaaaaaaaaa a-----
  1 ----- -----
  2 ----- -----
 ...

```

この図でわかるように、/a はグループ 0 のデータブロックのほとんどを占めていますが、他のグループは空のままです。現在ルートディレクトリ（/）に他のファイルが作成されている場合、そのグループのデータのための余裕はありません。

大容量ファイルの例外（ここでは各チャンクに 5 つのブロックが設定されています）では、FFS はファイルをグループ全体に広げ、その結果 1 つのグループ内の利用率が高すぎません。

```
group inodes      data
  0 /a----- /aaaaaa-----
  1 ----- aaaaa-----
  2 ----- aaaaa-----
  3 ----- aaaaa-----
  4 ----- aaaaa-----
  5 ----- aaaaa-----
  6 ----- -----
 ...

```

巧みな読者（それはあなたです）は、気づいているかもしれません。シーケンシャルなファイルアクセスの比較的一般的なケース（ユーザー や アプリケーション が 順番に 0~29 の チャンク を 読み込んだ 場合など）では、ディスク全体にファイルのブロックを広げるとパフォーマンスが低下することに気付くでしょう。しかし、チャンクサイズを慎重に選択することで、この問題に対処できます。

具体的には、チャンクサイズが十分に大きい場合、ファイルシステムはディスクからデータを転送する時間のほとんどを費やし、ブロックのチャンク間を探すシークで（比較的）少し時間を費やします。支払ったオーバーヘッドごとにさらに多くの作業を行うことによってオーバーヘッドを削減するこのプロセスは、amortization（償却）と呼ばれ、コンピュータシステムでは一般的な手法です。

例を挙げておきましょう。ディスクの平均位置決め時間（すなわち、シークおよび回転）が 10 ミリ秒である

と仮定します。更に、ディスクが 40MB/s でデータを転送すると仮定します。半分のチャンクのシーク時間で半分のデータの転送時間を費やす（ピークディスク性能の 50 % を達成する）ことが目標だった場合は、10ms の位置決めごとに 10ms のデータ転送に費やす必要があります。ですから、問題は次のようにになります。転送で 10 ミリ秒を費やすにはチャンクがどれくらい大きくなればなりませんか？ それは簡単です。ちょうど我々の古い友人、数学、特にディスクの章で言及された寸法分析を使用してください [AD14a]：

$$\frac{40 \text{ MB}}{\text{sec}} \cdot \frac{1024 \text{ KB}}{1 \text{ MB}} \cdot \frac{1 \text{ sec}}{1000 \text{ ms}} \cdot 10 \text{ ms} = 409.6 \text{ KB} \quad (41.1)$$

基本的には、この方程式の意味は次のとおりです。データを 40 MB/秒で転送する場合は、シークするたびに 409.6KB だけ転送する必要があります。このとき、半分のシーク時間で半分のデータ転送を達成する必要があります。同様に、ピーク帯域幅の 90 % (約 3.69MB)、またはピーク帯域幅 (40.6MB！) の 99 % を達成するために必要なチャンクのサイズを計算できます。ご覧のように、ピークに近づきたいほど、これらのチャンクは大きくなります（これらの値のプロットについては、図 41.2 を参照してください）。

しかし、大規模なファイルをグループ全体に広げるために、FFS はこのタイプの計算を使用しませんでした。代わりに、i ノード自体の構造に基づいて簡単なアプローチをとっていました。最初の 12 個の直接ブロックは、inode と同じグループに配置されました。後続の各間接ブロックと、それが指し示すすべてのブロックが異なるグループに配置されました。ブロックサイズが 4KB で、ディスクのアドレスが 32 ビットの場合、この方法では、ファイル (4MB) の 1024 ブロックが別々のグループに配置され、直接ポインタによって指し示されるファイルの最初の 48KB が例外となります。

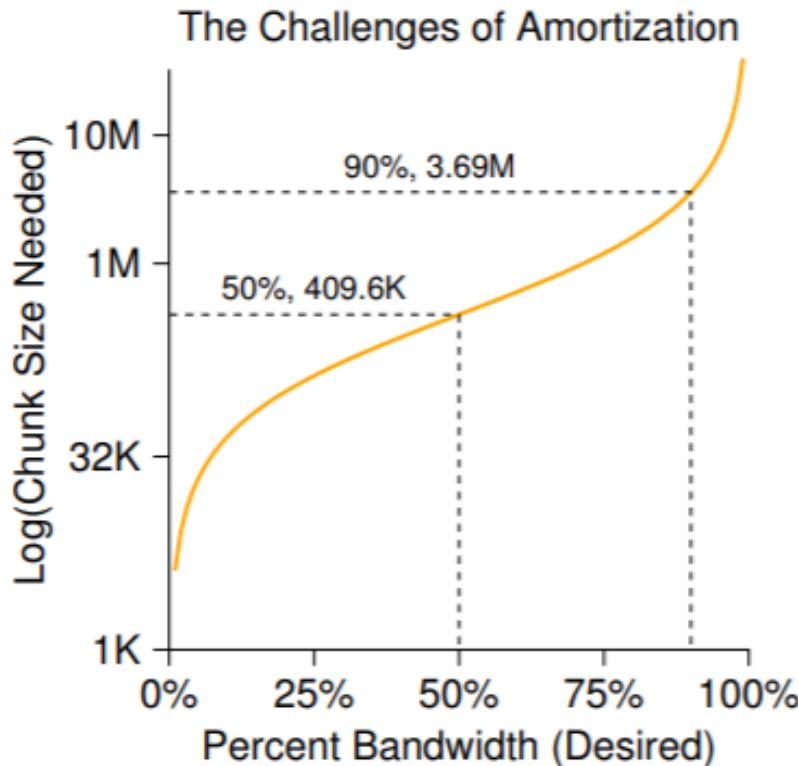


Figure 41.2: Amortization: How Big Do Chunks Have To Be?

ディスクメーカーの傾向としては、ディスクメーカーが表面にビット数を増やしたら、転送速度がかなり速くなりますが、シークに関連するドライブの機械的な側面（ディスクアームの速度と回転速度）はゆっくりと向

上します [P98]。時間が経つにつれて、機械的なコストは比較的高価になり、そのコストを償却するためには、シークの間にもっと多くのデータを転送する必要があります。

41.7 A Few Other Things About FFS

FFS はいくつかの革新も導入しました。特に、デザイナーは小さなファイルを扱うことを非常に心配していました。それが判明したので、多くのファイルは 2KB 程度の大きさであったが、4KB のブロックを使用することでデータを転送するのは効率は良かったのですが、ディスクスペースの効率は悪くなってしまいました。したがって、この内部的な断片化は、典型的なファイルシステムではディスクのおよそ半分が無駄になる可能性があります。

FFS の設計者がヒットした解決策は簡単で、問題を解決しました。彼らは、ファイルシステムがファイルに割り当てることができる 512 バイトの小さなブロックであるサブブロックを導入することに決めました。したがって、小さなファイル (サイズが 1KB) を作成した場合、2 つのサブブロックを占有し、4KB ブロック全体を無駄にすることはありません。ファイルが大きくなり、4KB のデータを取得するまで、ファイルシステムは、512 バイトのブロックを割り当て続けます。4KB のデータを取得した場合、FFS は 4KB ブロックを見つけて、そのサブブロックを 4KB ブロックにコピーし、将来の使用のためにサブブロックを解放します。

このプロセスは非効率的であり、ファイルシステムに多くの余分な作業 (特に、コピーを実行するための余分な I/O) を必要とすることがあります。そして、あなたはもう一度やって来るだろう！ したがって、FFS は、一般的に、libc ライブラリを変更することでこの最悪な動作を回避しました。ライブラリは書き込みをバッファリングしてからファイルシステムの 4KB のチャンクに発行するので、ほとんどの場合、サブブロックの特殊化は完全に回避されます。

FFS が導入したもう一つのすばらしい点は、パフォーマンスのために最適化されたディスクレイアウトでした。その時代 (SCSI や他の最新のデバイスインターフェースの前の)、ディスクは全く洗練されておらず、ホスト CPU は操作をより実践的に制御する必要がありました。図 41.3 の左側のように、ファイルがディスクの連続するセクタに置かれたときに FFS に問題が発生しました。

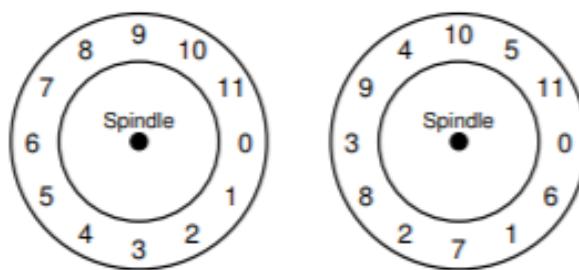


Figure 41.3: FFS: Standard Versus Parameterized Placement

特に、この問題は順序読み出し時に発生しました。FFS は最初にブロック 0 への読み出しを発行します。読み出しが完了し、FFS がブロック 1 への読み出しを発行した時点では、それは遅すぎました。すなわち、ブロック 1 の読み出しの前にヘッドを通り過ぎるため、一度回転を待たなければいけません。

図 41.3 の右側に示すように、FFS はこの問題を別のレイアウトで解決しました。他のすべてのブロック (この例では) をスキップすることによって、FFS はディスクヘッドを通過する前に次のブロックを要求するのに十分な時間を確保します。実際、FFS は余分な回転を避けるためにレイアウト (設計) を行い、特定のディスクに対して、スキップする必要があるブロックの数を十分に把握するほどスマートでした。このテクニックは、FFS がディスクの特定のパフォーマンスパラメータを把握し、それらを使って厳密なズラしたレイアウトスキーム (設計計画) を決定するので、parameterization(パラメータ化) と呼ばれていました。

あなたは考えているかもしれません。この計画は結局大したことではありません。実際には、このタイプのレイアウトでピーク帯域幅の 50 % しか得られません。なぜなら、各ブロックを一度読み取るためには、各トラックを 2 回周回しなければならないからです。幸運なことに、現代のディスクはずっとスマートです。内部的にトラック全体を読み込んで内部のディスクキャッシュ（これはまさにトラックバッファと呼ばれます）にバッファリングします。その後、トラックへの次の読み込みで、ディスクはキャッシュから目的のデータを返します。したがって、ファイルシステムはこれらの信じられないほど低いレベルの詳細を心配する必要はありません。抽象化と高水準のインターフェースは、適切に設計されたときに良いことになります。

他のいくつかのユーザビリティ改善も加えられました。FFS は、長いファイル名を可能にする最初のファイルシステムの 1 つであったため、従来の固定サイズのアプローチ（例えば、8 文字）ではなくファイルシステムでより表現力のある名前が可能になりました。さらに、シンボリックリンクと呼ばれる新しい概念が導入されました。前の章 [AD14b] で説明したように、ハードリンクは、（ファイルシステム階層にループを導入する恐れがあるために）ディレクトリを指すことができず、同じボリューム内のファイルのみを指示することができます（inode 番号はまだ意味があるはずです）。シンボリックリンクを使用すると、システム上の他のファイルやディレクトリの「エイリアス」を作成することができ、柔軟性が向上します。FFS は、ファイルの名前を変更するための原子的な `rename()` 操作も導入しました。基本技術を超えた使いやすさの向上により、FFS はより強力なユーザーベースになる可能性が高くなりました。

TIP: MAKE THE SYSTEM USABLE

おそらく、FFS の最も基本的な教訓は、ディスク対応レイアウトの概念的な概念を導入しただけでなく、単にシステムをより使いやすくした多くの機能を追加したということです。長いファイル名、シンボリックリンク、およびリネーム操作はすべてアトミックに機能し、システムのユーザビリティを改善しました。この研究論文について書くのは難しい（「The Symbolic Link: Hard Link's Long Lost Cousin」についての 14 ページを読んでみよう）が、そのような小さな機能は FFS をより有用にし、採用の機会を増やす可能性が高いです。システムを使いやすくすることは、深い技術革新よりも重要であることが多いです。

41.8 Summary

ファイル管理の問題は、オペレーティングシステム内で最も興味深い問題の 1 つであることが明らかになり、最も重要なハードディスクというデバイスに対処する方法が示されていたことから、FFS の導入はファイルシステムの歴史における大きな潮流でした。それ以来、何百もの新しいファイルシステムが開発されてきましたが、今日でも多くのファイルシステムが FFS からの手がかりを取ります（例えば、Linux ext2 と ext3 は明らかに FFS の子孫です）。明らかなのはすべての現代のシステムは FFS の主な教訓から成り立っていることです。それは、「ディスクをディスクのように扱う」ということです。

参考文献

[AD14a] “Operating Systems: Three Easy Pieces”

Chapter: Hard Disk Drives

Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau

There is no way you should be reading about FFS without having first understood hard drives in some detail. If you try to do so, please instead go directly to jail; do not pass go, and, critically, do not collect 200 much-needed simoleons.

[AD14b] “Operating Systems: Three Easy Pieces”

Chapter: File System Implementation

Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau

As above, it makes little sense to read this chapter unless you have read (and understood) the chapter on file system implementation. Otherwise, we'll be throwing around terms like "inode" and "indirect block" and you'll be like "huh?" and that is no fun for either of us.

[K94] "The Design of the SEER Predictive Caching System"

G. H. Kuenning

MOBICOMM '94, Santa Cruz, California, December 1994

According to Kuenning, this is the best overview of the SEER project, which led to (among other things) the collection of these traces.

[MJLF84] "A Fast File System for UNIX"

Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry

ACM Transactions on Computing Systems, 2:3, pages 181-197.

August, 1984. McKusick was recently honored with the IEEE Reynold B. Johnson award for his contributions to file systems, much of which was based on his work building FFS. In his acceptance speech, he discussed the original FFS software: only 1200 lines of code! Modern versions are a little more complex, e.g., the BSD FFS descendant now is in the 50-thousand lines-of-code range.

[P98] "Hardware Technology Trends and Database Opportunities"

David A. Patterson

Keynote Lecture at the ACM SIGMOD Conference (SIGMOD '98)

June, 1998

A great and simple overview of disk technology trends and how they change over time.

42 Crash Consistency: FSCK and Journaling

これまで見てきたように、ファイルシステムは、ファイルシステムから期待される基本的な抽象化をサポートするために必要なファイル、ディレクトリ、その他すべてのメタデータなど、一連のデータ構造を管理します。ほとんどのデータ構造(例えば、実行中のプログラムのメモリ内にあるもの)とは異なり、ファイルシステムのデータ構造は永続的でなければならず、長時間に渡って生き残り、電力損失があってもデータを保持しなければいけません(ハードディスクやデバイスフラッシュベースのSSDのように)

ファイルシステムが直面する大きな課題の1つは、停電やシステムクラッシュがあっても永続的なデータ構造を更新する方法です。具体的には、ディスク上の構造を更新する途中で、誰かが電源コード上を移動してマシンの電源が失われたらどうなりますか? または、オペレーティングシステムにバグが発生し、クラッシュしたらどうなりますか? 停電やクラッシュのため、永続的なデータ構造を更新するのは非常に難しく、クラッシュ貫性の問題として知られているファイルシステムの実装において新しい興味深い問題につながります。

この問題は非常に理解しやすいです。特定の操作を完了するために、2つのオンディスク構造AとBを更新する必要があるとします。ディスクは一度に1つの要求のみを処理するため、これらの要求の1つが最初にディスクに到達します(AまたはB)。1回の書き込みが完了してからシステムがクラッシュしたり電源が切れた場合、ディスク上の構造は矛盾した状態になります。したがって、すべてのファイルシステムを解決する必要があるという問題があります。

THE CRUX: HOW TO UPDATE THE DISK DESPITE CRASHES

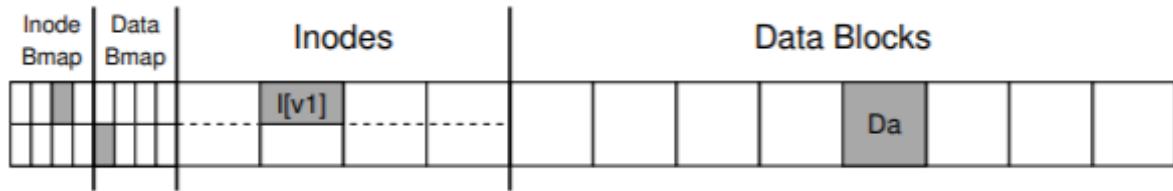
システムは、2つの書き込みの間でクラッシュまたは電力を失う可能性があり、したがって、ディスク上の状態が部分的にしか更新されない可能性があります。クラッシュ後、システムは起動し、(ファイルにアクセスするために)ファイルシステムを再度マウントします。任意の時点でのクラッシュが発生する可能性があることを考えれば、ファイルシステムがディスク上のイメージを合理的な状態に保つにはどうすればよいでしょうか?

この章では、この問題をより詳しく説明し、ファイルシステムがそれを克服するために使用したいつかの方法を見ていきます。fsckやファイルシステムチェックと呼ばれる古いファイルシステムのアプローチを調べてみましょう。次に、書き込みごとに若干のオーバーヘッドを追加するが、クラッシュや電力損失からより迅速に回復するテクニックである、ジャーナリング(write ahead loggingとも呼ばれる)という別のアプローチに注目します。私たちは、Linux ext3 T98、PAA05が実装しているいくつかの異なるジャーナリングの仕組みを含めて、ジャーナリングの基本的な機構について議論する予定です。

42.1 A Detailed Example

ジャーナリングの調査を開始するために、例を見てみましょう。ディスク上の構造を何らかの方法で更新する仕事量を使用する必要があります。ここでは、仕事量が単純であると仮定します。単一のデータブロックを既存のファイルに追加することです。追加は、ファイルを開き、ファイルのオフセットをファイルの最後に移動するために`lseek()`を呼び出し、ファイルを閉じる前にファイルに4KBの単一の書き込みを発行することによって行われます。

これまでに見たようなファイルシステムと同様に、ディスク上に標準的な単純なファイルシステム構造を使用しているとします。この小さな例には、inodeビットマップ(inodeあたり8ビット、inodeあたり一つ)データビットマップ(8ビット、データブロックあたり1つ)、iノード(合計8つ、0から7までの番号、4つのブロックにまたがる)、データブロック(合計8つ、0から7までの番号付け)を含んでいます。このファイルシステムの図は次のとおりです。



図に示されている構造を見ると、inode ビットマップにマークされ、单一に割り当てられた 1 つの inode(inode 番号 2) とデータビットマップにマークされ、单一に割り当てられたデータブロック (データブロック 4) があります。inode はこの i ノードの最初のバージョンであるため、I[v1] と表示されます。それはすぐに更新されます (上記の仕事量のため)。この単純化された inode の内部を見てみましょう。さっそく I[v1] の内部を見ていきます：

<code>owner</code>	<code>:</code>	<code>remzi</code>
<code>permissions</code>	<code>:</code>	<code>read-write</code>
<code>size</code>	<code>:</code>	<code>1</code>
<code>pointer</code>	<code>:</code>	<code>4</code>
<code>pointer</code>	<code>:</code>	<code>null</code>
<code>pointer</code>	<code>:</code>	<code>null</code>
<code>pointer</code>	<code>:</code>	<code>null</code>

この単純化された i ノードでは、ファイルのサイズは 1(1 つのブロックが割り当てられています)、最初のダイレクトポインタがブロック 4(ファイルの最初のデータブロックである Da) を指し、3 つのダイレクトポインターはすべて null に設定されます (使用されていないことを示す)。もちろん、実際の inode にはさらに多くのフィールドがあります。詳細については、前の章を参照してください。

ファイルに追加するときに新しいデータブロックを追加するので、ディスク上の 3 つの構造を更新する必要があります：inode(新しいブロックを指し示す必要があり、追加のためにサイズが大きくなる) 新しいデータブロックである Db、および新しいデータブロックが割り当てられたことを示すためにデータビットマップの新しいバージョン (それを B[v2] と呼ぶ) を含みます。

したがって、システムのメモリには、ディスクに書き込む必要がある 3 つのブロックがあります。更新された i ノード (i ノードバージョン 2、または I[v2] の短縮形) は、次のようにになります。

```

owner          : remzi
permissions   : read-write
size          : 2
pointer       : 4
pointer       : 5
pointer       : null
pointer       : null

```

更新されたデータビットマップ (B[v2]) は 00001100 のようになります。最後に、データブロック (Db) があります。それはユーザーがファイルに入れたものです。おそらく音楽を盗んだのでしょうか？ 私たちが望むのは、ファイルシステムの最終的なオンディスクイメージが次のことになります。



この移行を達成するためには、ファイルシステムは、inode(I[v2])、ビットマップ (B[v2])、およびデータブロック (Db) の 3 つのディスクへの個別書き込みを実行する必要があります。これらの書き込みは通常、ユーザーが `write()` システムコールを発行したときにすぐには起こらないことに注意してください。むしろ、dirty inode、ビットマップ、および新しいデータは、メインメモリ (ページキャッシュまたはバッファキャッシュ内) に最初に少しの時間でおかれます。ファイルシステムが最終的にそれらをディスクに書き込むことを決定すると (例えば 5 秒または 30 秒後に)、ファイルシステムは必要な書き込み要求をディスクに発行します。残念ながら、クラッシュが発生すると、ディスクへのこれらの更新が妨げられます。特に、これらの書き込みのうちの 1 つまたは 2 つが実行された後にクラッシュが発生し、3 つすべてが完了していない場合、ファイルシステムは面白い状態になる可能性があります。### Crash Scenarios 問題をよりよく理解するために、いくつかのクラッシュシナリオの例を見てみましょう。1 回の書き込みだけが成功したとします。このように 3 つの可能な結果があります。

- データブロック (Db) だけがディスクに書き込まれます。

この場合、データはディスク上にありますが、それを指す inode ではなく、ブロックが割り当てられていることを示すビットマップもありません。したがって、書き込みが起こらなかったかのようになります。このケースは、ファイルシステムの一貫性の観点から、まったく問題ではありません。

- 更新された inode(I[v2]) だけがディスクに書き込まれます。

この場合、inode は Db が書き込まれる直前のディスクアドレス (5) を指していますが、Db は書き込まれていません。したがって、そのポインタを信頼すれば、ディスクからガベージデータ (ディスクアド

レス 5 の古い内容) を読み込みます。さらに、ファイルシステムの矛盾と呼ばれる新しい問題があります。ディスク上のビットマップは、データブロック 5 が割り当てられていないが、inode はそれが持っていると言っています。ビットマップと i ノードとの間の不一致は、ファイルシステムのデータ構造の不一致です。ファイルシステムを使用するには、何らかの形でこの問題を解決する必要があります(詳細は後述)。

- 更新されたビットマップ (B[v2]) のみがディスクに書き込まれます。

この場合、ビットマップはブロック 5 が割り当てられていることを示しますが、それを指す i ノードはありません。したがって、ファイルシステムには一貫性がありません。ブロック 5 がファイルシステムによって決して使用されないので、この書き込みは未解決のまま残された場合、space leak(スペースリーク) が生じます。

3 つのブロックをディスクに書き込む試みには、さらに 3 つのクラッシュシナリオがあります。これらの場合、2 回の書き込みは成功し、最後の書き込みは失敗します。

- inode(I[v2]) とビットマップ (B[v2]) はディスクに書き込まれますが、データ (Db) は書き込まれません。

この場合、ファイルシステムのメタデータは完全に一貫しています。つまり、inode にはブロック 5 へのポインタがあり、ビットマップには 5 が使用されていることが示されているため、ファイルシステムのメタデータの観点から見ても問題ありません。しかし 1 つの問題があります: 5 は再びそれにゴミ(古いデータ)を入れています。

- inode(I[v2]) とデータブロック (Db) は書き込まれますが、ビットマップ (B[v2]) は書き込まれません。

この場合、inode はディスク上の正しいデータを指していますが、inode と古いバージョンのビットマップ (B1) の間に矛盾があります。したがって、ファイルシステムを使用する前に問題を解決する必要があります。

- ビットマップ (B[v2]) とデータブロック (Db) は書き込まれますが、i ノード (I[v2]) は書き込まれません。

この場合、i ノードとデータビットマップの間に再び矛盾があります。しかし、ブロックが書き込まれ、ビットマップがその使用法を示していても、inode がファイルを指していないので、どのファイルに属しているかわかりません。

The Crash Consistency Problem

うまくいけば、これらのクラッシュシナリオから、クラッシュのためにディスク上のファイルシステムイメージに発生する可能性がある多くの問題を見ること、ファイルシステムのデータ構造に不整合、スペースリークが発生する可能性、ガベージデータをユーザーに返すこと等々あります。私たちが理想的にやってみたいのは、ファイルシステムをある一貫性のある状態(例えば、ファイルが追加される前)から別のものに(例えば、inode、ビットマップ、新しいデータブロックがディスクに書き込まれた後)に移動することです。残念ながら、ディスクは一度に 1 つの書き込みしかコミットしないため、これらの更新の間にクラッシュまたは電力損失が発生する可能性があるため、簡単には実行できません。この一般的な問題をクラッシュ一貫性の問題と呼びます(一貫性のある更新の問題とも呼ぶことができます)。

42.2 Solution #1: The File System Checker

初期のファイルシステムは、簡単なアプローチをとっていました。基本的に、彼らは不一致を起こさせ、後で(リブート時に)修正することに決めました。このアプローチの古典的な例は、これを行うツール fsck があります。fsck は、そのような矛盾を見つけて修復するための UNIX ツールです。ディスクパーティションを

チェックして修復するための同様のツールが、異なるシステムに存在します。このような方法ではすべての問題を解決できないことに注意してください。例えば、inode がガベージデータを指しているファイルシステムが一貫しているように見える上のケースがあります。唯一の実際の目標は、ファイルシステムのメタデータが内部的に一貫していることを確認することです。

ツール fsck は、McKusick と Kowalski の論文 [MK96] に要約されているように、いくつかの段階で動作します。これは、ファイルシステムがマウントされ、使用可能になる前に実行されます (fsck は、実行中に他のファイルシステムアクティビティが実行されていないことを前提としています)。一度終了すると、オンディスクファイルシステムは一貫していて、ユーザーがアクセスできるようにする必要があります。ここでは、fsck の動作の基本的な概要を示します。

- Superblock(スーパーブロック)

fsck は、スーパーブロックが妥当であるかどうか最初にチェックします。ほとんどの場合、ファイルシステムのサイズが割り当てられたブロック数よりも大きいかどうかを確認するなどの健全性チェックが行われます。通常、これらの sanity(健全性) チェックの目的は、疑わしい(破損した) スーパーブロックを見つけることです。この場合、システム(または管理者)は、スーパーブロックの代替コピーを使用することを決定することができます。

- Free blocks(フリーブロック)

次に、fsck は inode、間接ブロック、二重間接ブロックなどをスキャンして、ファイルシステム内でどのブロックが現在割り当てられているかを理解します。この知識を使用して、割り当てビットマップの正しいバージョンを生成します。したがって、ビットマップと i ノードの間に矛盾がある場合、i ノード内の情報を信頼することによって解決されます。同じタイプのチェックがすべての inode に対して実行され、使用中のように見えるすべての inode が inode ビットマップでそのようにマークされることを確認します。

- Inode state(ノードの状態)

各 i ノードは、破損またはその他の問題がないかどうかチェックされます。たとえば、fsck は、割り当てられた各 i ノードが有効なタイプのフィールド(通常のファイル、ディレクトリ、シンボリックリンクなど)を持っているかどうかを確認します。簡単に修正できない inode フィールドに問題がある場合、inode は疑わしいとみなされ、fsck によってクリアされます。このとき、inode ビットマップが対応して更新されます。

- Inode links (i ノードリンク)

また、fsck は各割り当てられた i ノードのリンク数も確認します。ご存知のように、リンク数は、この特定のファイルへの参照(つまり、リンク)を含む異なるディレクトリの数を示します。リンクカウントを確認するために、fsck はルートディレクトリから開始してディレクトリツリー全体をスキャンし、ファイルシステム内のすべてのファイルとディレクトリの独自のリンク数を作成します。新しく計算されたカウントと i ノード内で見つかったカウントとの間に不一致がある場合、通常は inode 内のカウントを固定することによって是正措置を講じる必要があります。割り当てられた inode が検出されたがディレクトリを参照していない場合、lost + found ディレクトリに移動されます。

- Duplicates(重複)

また、fsck は、重複ポインタ、すなわち、2 つの異なる i ノードが同じブロックを参照する場合もチェックします。1 つの i ノードが明らかに悪い場合は、消去される可能性があります。あるいは、指示しブロックをコピーして、各 i ノードに必要に応じて独自のコピーを与えることもできます。

- Bad blocks (不良ブロック)

すべてのポインタのリストをスキャンしながら、不良ブロックポインタのチェックも実行されます。ポインタは、有効な範囲外のものを明示的に指す場合(例えば、パーティションサイズより大きなブロッ

クを参照するアドレスを有する場合など)、「不良」とみなされます。この場合、fsck はインテリジェントな処理を行うことはできません。i ノードまたは間接ブロックからポインタを削除(クリア)するだけです。

- Directory checks (ディレクトリチェック)

fsck はユーザーファイルの内容を理解しません。ただし、ディレクトリには、ファイルシステム自体によって作成された特別な形式の情報が格納されています。したがって、fsck は、各ディレクトリの内容について追加の整合性チェックを実行し、“.” と “..” が最初のエントリであり、ディレクトリエントリで参照されている各 i ノードが割り当てられていることを確認し、ディレクトリが階層全体で複数回リンクされないようにします。

ご覧のように、動作中の fsck を構築するには、ファイルシステムの複雑な知識が必要です。そのようなコードがすべての場合に正しく動作することを確認することは困難です [G + 08]。しかし、fsck(と同様のアプローチ)は、より大きな、おそらくより根本的な問題を抱えています。それは、非常に遅いということです。ディスクボリュームが非常に大きい場合は、ディスク全体をスキャンして割り当てられたすべてのブロックを検索し、ディレクトリツリー全体を読み取るには数分または数時間かかることがあります。ディスクが大容量になり、RAID が普及したときの fsck のパフォーマンスは、(最近の進歩 [M + 13] にもかかわらず) 法外になりました。

より高いレベルでは、fsck の基本的な前提はちょっと不合理なようです。ディスクに 3 つのブロックしか書き込まれていない上記の例を考えてみましょう。わずか 3 ブロックの更新中に発生した問題を解決するためにディスク全体をスキャンするのは非常に高価です。このような状況は、寝室の床に鍵を落とした後、地下室から出発してすべての部屋を通って鍵検索アルゴリズムを検索することに似ています。それは確実に動作しますが、全てを探索することは無駄です。このように、ディスク(および RAID)が増えるにつれて、研究者や実務者は他のソリューションを探すようになりました。

42.3 Solution #2: Journaling (or Write-Ahead Logging)

おそらく一貫した更新の問題に対する最も一般的な解決策は、データベース管理システムの世界からアイデアを盗むことです。このアイデアは、write ahead logging(先行書き込みログ)と呼ばれ、この種の問題を正確に解決するために考案されたものです。ファイルシステムでは、歴史的な理由から、write ahead logging journaling(先行書き込みログ記録ジャーナリング)を呼び出します。これを行うための最初のファイルシステムは Cedar [H87] でしたが、Linux ext3 と ext4、reiserfs、IBM の JFS、SGI の XFS、Windows NTFS など、現代の多くのファイルシステムがこの考え方を使用しています。

基本的な考え方は次のとおりです。ディスクを更新するときは、構造を上書きする前に、まずあなたがしようとしていることを説明する小さなメモ(ディスク上の他の場所、よく知られている場所)を書き留めます。つまり、このメモを書くことは“先行書き込み”的部分であり、“ログ”として整理する構造である write ahead logging に書きます。

メモをディスクに書き込むことで、構造の更新(上書き)中にクラッシュが発生した場合、元に戻って作成したメモを見て再試行することができます。したがって、ディスク全体をスキャンするのではなく、クラッシュ後に修正する対象(および修正方法)を正確に知ることができます。設計上、ジャーナリングは更新中に少しの作業を追加することで、リカバリ時に必要な作業量を大幅に削減します。

ここでは、普及しているジャーナリングファイルシステムである Linux ext3 がジャーナリングをファイルシステムに組み込む方法について説明します。ディスク上の構造のほとんどは Linux ext2 と同じです。例えば、ディスクはブロックグループに分割され、各ブロックグループには inode とデータビットマップ、inode とデータブロックがあります。新しい鍵構造はジャーナル自体であり、パーティション内または他のデバイス内のスペースをいくらか占有します。したがって、ext2 ファイルシステム(ジャーナルなし)は次のようにな

ります。



ジャーナルが同じファイルシステムイメージ内に置かれているとします(ただし、別のデバイスやファイルシステム内のファイルに配置されることもあります)。ジャーナルを持つ ext3 ファイルシステムは次のようになります。

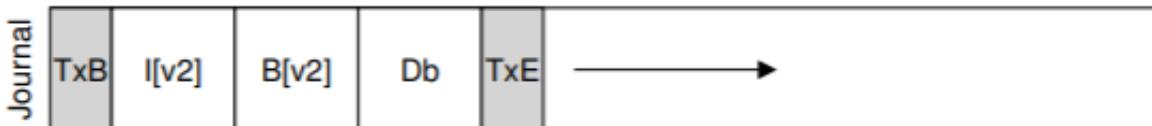


実際の違いはジャーナルの存在だけであり、もちろん使い方もです。

Data Journaling

データジャーナリングの仕組みを理解するための簡単な例を見てみましょう。データジャーナリングは、Linux ext3 ファイルシステムのモードとして利用できます。ここから、この議論の多くがベースになっています。

inode(I[v2])、ビットマップ(B[v2])、およびデータブロック(Db)をディスクに書き戻したい場合には、もう一度標準アップデートを行ってください。最終的なディスクの場所に書き込む前に、最初にそれらをログ(a.k.a. journal)に書き込む予定です。このログのようになります：



ここに5つのブロックを書きました。トランザクション開始(TxB)は、ファイルシステムに対する保留中の更新に関する情報(ブロック I[v2]、B[v2]、および Db の最終アドレスなど)を含むこの更新について教えてくれます、またその中に何らかの種類トランザクション識別子(TID)があります。中央の3つのブロックには、ブロック自体の正確な内容が含まれています。これは physical logging(物理ロギング)と呼ばれ、更新の正確な物理的な内容をジャーナルに入れています(logical logging(論理ロギング)はジャーナルにアップデートのよりコンパクトな論理表現を置きます。例えば、“このアップデートはデータブロック Db にファイル X を追加したい”です。これはもう少し複雑ですが、ログの領域を節約し、パフォーマンスを向上させることができます)。最終ブロック(TxE)は、このトランザクションの最後のマーカーであり、TID も含みます。

このトランザクションが安全にディスクに保存されると、ファイルシステム内の古い構造を上書きする準備が整います。このプロセスをチェックポイントと呼びます。したがって、ファイルシステムをチェックポイントする(すなわち、ジャーナル内の保留中の更新を最新のものにする)ために、I[v2]、B[v2] および Db の書き込みを上記のディスク位置に発行します。これらの書き込みが正常に完了すると、ファイルシステムのチェックポイントが成功し、基本的に完了しています。したがって、私たちの最初の一連の操作は以下になります：

1. Journal write(ジャーナルライト)

トランザクション開始ブロック、保留中のすべてのデータおよびメタデータ更新、およびトランザクション終了ブロックを含むトランザクションをログに書き込みます。これらの書き込みが完了するのを待ちます。

2. Checkpoint(チェックポイント)

保留中のメタデータとデータの更新をファイルシステムの最終的な場所に書き込みます。

この例では、最初に TxB、I[v2]、B[v2]、Db、TxE をジャーナルに書きます。これらの書き込みが完了すると、I[v2]、B[v2]、Db をディスク上の最終位置にチェックポイントすることで更新を完了します。

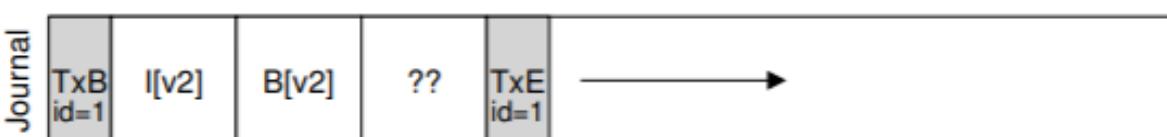
ジャーナルへの書き込み中にクラッシュが発生した場合、状況は少し難解になります。ここでは、トランザクションのブロックセット（たとえば、TxB、I[v2]、B[v2]、Db、TxE）をディスクに書き込もうとしています。これを行う簡単な方法の 1 つは、それぞれに一度一回ずつ発行し、それが完了するのを待ってから、次を発行することです。しかし、これは遅いです。理想的には、5 回の書き込みをすべて一度に発行したいと思います。5 回の書き込みを 1 回の書き込みに変えて高速化することができます。しかし、以下の理由から、これは安全ではありません：このような大きな書き込みがある場合、ディスクは内部的にスケジューリングを実行し、大きな書き込みの小さな部分を任意の順序で完了することができます。したがって、ディスクは、(1)TxB、I[v2]、B[v2]、TxE を書き込み、後で (2)Db だけ内部的に書き込むかもしれません。残念ながら、ディスクが (1) と (2) の間で電源を失った場合、これがディスク上で終了します。

ASIDE: FORCING WRITES TO DISK

2 つのディスク書き込みの間で順序付けを行うためには、現代のファイルシステムにはいくつかの注意が必要です。昔は、A と B の 2 つの書き込み間の順序を強制するのは簡単でした。ディスクに A の書き込みを発行し、書き込みが完了したらディスクが OS に割り込むのを待ってから、B の書き込みを発行します。

ディスク内のライトキャッシュの使用が増えたため、状況はやや複雑になりました。書き込みバッファリングを有効にすると（即時報告とも呼ばれることもあります）、書き込みはディスクのメモリキャッシュに置かれ、まだディスクに到達していないときに書き込みを完了したことをディスクは OS に通知します。その後、OS が次の書き込みを発行する場合、以前の書き込み後にディスクに到達することを保証しません。したがって、書き込み間の順序付けは保持されません。1 つの解決策は、書き込みバッファリングを無効にすることです。しかし、より現代的なシステムは、特別な注意を払い、明示的な書き込みバリアを発行します。バリアが完了すると、バリアの後に発行された書き込みの前である、バリアがディスクに到着する前に発行されたすべての書き込みが保証されます。

このすべての機械は、ディスクの正しい動作に大きな信頼を必要とします。残念なことに、最近の調査では、「高性能」ディスクを提供しようとしているディスクメーカーの中には、書き込みバリア要求を明示的に無視しているため、ディスクが一見高速に動作するものの、誤動作の危険があります [C + 13, R + 11]。Kahan は、「速いことは間違っていたとしても、遅い時間にほとんどいつも勝つ。」と言いました。

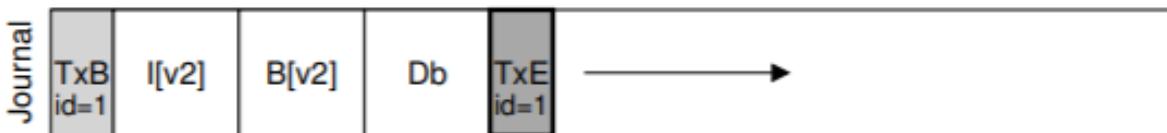


なぜこれが問題なのですか？ さて、トランザクションは有効なトランザクションのように見えます（シーケンス番号が一致する開始点と終了点があります）。さらに、ファイルシステムはその第 4 のブロックを見ることができず、それが間違っていることを知ることができません。結局のところ、それは任意のユーザーデータです。したがって、システムがリブートしてリカバリを実行すると、このトランザクションが再生され、ゴミ・ブロックである'???' の内容が Db がある場所に無意識にコピーされます。これは、ファイル内の任意のユーザーデータにとって悪いことです。スーパー・ブロックのようなファイルシステムの重要な部分にファイルシステムがマウントできなくなる可能性がある場合は、それはずっと悪いことです。

この問題を回避するために、ファイルシステムは 2 つのステップでトランザクション書き込みを発行します。まず、TxE ブロックを除くすべてのブロックをジャーナルに書き込み、一度にこれらの書き込みを発行します。これらの書き込みが完了すると、ジャーナルは次のようにになります(私たちの追加作業を再度仮定します)。



これらの書き込みが完了すると、ファイルシステムは TxE ブロックの書き込みを発行し、ジャーナルをこの safe state(安全状態) という最終状態にします。



このプロセスの重要な側面は、ディスクによって提供される原子性の保証です。ディスクでは、512 バイトの書き込みが発生するか、それとも発生しないのかが保証されています(半分書きされることはありません)。したがって、Tx E の書き込みがアトミックであることを確認するには、それを単一の 512 バイトブロックにする必要があります。したがって、ファイルシステムを更新する現在のプロトコルは、3 つのフェーズのそれぞれが次のようにラベル付けされています。

1. Journal write(ジャーナルライト)

トランザクションの内容(TxB、メタデータ、データを含む)をログに書き込みます。これらの書き込みが完了するのを待ちます。

2. Journal commit(ジャーナルコミット)

トランザクションコミットブロック(TxE を含む)をログに書き込みます。書き込みが完了するまで待ちます。トランザクションはコミットされていると言います。

3. Checkpoint(チェックポイント)

アップデートの内容(メタデータとデータ)を最終的なディスク上の場所に書き込みます。

Recovery

ファイルシステムがクラッシュから回復するためにジャーナルの内容をどのように使用できるかを理解しましょう。この一連の更新中にいつでもクラッシュが発生する可能性があります。トランザクションがログに安全に書き込まれる前にクラッシュが発生した場合(つまり、上記の手順 2 が完了する前)、私たちの仕事は簡単です。保留中の更新は単にスキップされます。トランザクションがログにコミットした後で、チェックポイントが完了する前にクラッシュが発生した場合、ファイルシステムは次のように更新を回復できます。システムが起動すると、ファイルシステムのリカバリプロセスはログをスキャンし、ディスクにコミットしたトランザクションを探します。これらのトランザクションはこのように(順番に)再生され、ファイルシステムはトランザクション内のブロックを最終的なオンディスク位置に書き戻そうと再度試みる。この形式のロギングは、最も簡単な形式の 1 つで、redo logging と呼ばれます。

ジャーナル内のコミットされたトランザクションをリカバリすることにより、ファイルシステムはディスク上の構造が一貫していることを保証し、ファイルシステムをマウントして新しい要求の準備を進めることができます。

チェックポイント処理中の任意の時点でのブロックの最終位置の更新が完了した後でも、クラッシュが発生することは問題ありません。最悪の場合、これらの更新のうちのいくつかは、回復中に再び単に実行されます。

リカバリはまれな操作である（予期しないシステムクラッシュの後でのみ行われる）ため、いくつかの重複した書き込みは気にしません。

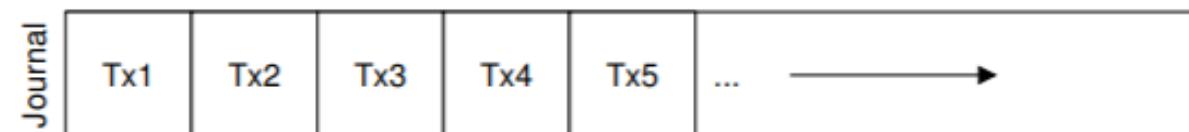
Batching Log Updates

基本的なプロトコルが余計なディスクトラフィックを増やす可能性があることに気づいたかもしれません。たとえば、file1 と file2 という 2 つのファイルを同じディレクトリに作成するとします。1 つのファイルを作成するには、inode のビットマップ（新しい i ノードを割り当てるため）、新しく作成したファイルの i ノード、新しいディレクトリを含む親ディレクトリのデータブロックディレクトリエンティ、および親ディレクトリの i ノード（現在は新しい変更時刻があります）が表示されます。ジャーナリングでは、この 2 つのファイル作成のそれぞれについて、この情報を論理的にジャーナルにコミットします。ファイルが同じディレクトリにあり、同じ i ノードブロック内の inode を持っていたとしても、これは慎重でなければ、これらの同じブロックを何度も何度も書くことになります。

この問題を解決するために、ファイルシステムの中には、一度に 1 つずつディスクを更新するものはありません（Linux の ext3 など）。むしろ、すべての更新をグローバルトランザクションにバッファリングすることができます。上記の例では、2 つのファイルが作成されると、ファイルシステムは、メモリ内の inode ビットマップ、ファイルの inode、ディレクトリデータ、ディレクトリの inode を dirty とマークし、現在のトランザクションを構成するブロックのリストにそれらを追加します。最後に、これらのブロックをディスクに書き込む（例えば、5 秒のタイムアウトの後）と、この単一のグローバルトランザクションは、上記のすべての更新の詳細を含むコミットがされます。したがって、更新をバッファリングすることによって、ファイルシステムは多くの場合ディスクへの過剰な書き込みトラフィックを回避することができます。

Making The Log Finite

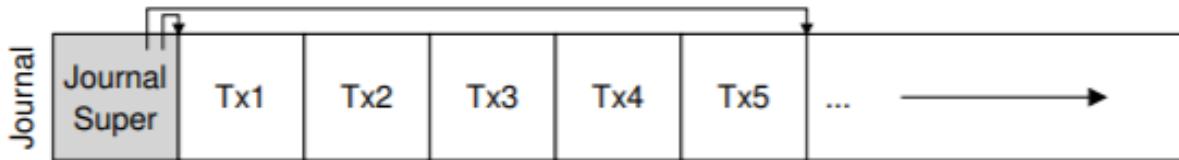
したがって、我々は、ファイルシステム上のディスク構造を更新するための基本的なプロトコルに到達しました。ファイルシステムのバッファは、しばらくの間メモリ内の更新します。最終的にディスクに書き込むとき、ファイルシステムは最初にトランザクションの詳細をジャーナル（write ahead log）に慎重に書き出します。トランザクションが完了すると、ファイルシステムはそれらのブロックをディスク上の最終的な場所にチェックポイントします。ただし、ログは有限のサイズです。この図に示すようにトランザクションを追加し続けると、すぐに処理が完了します。何が起こったと思いますか？



ログがいっぱいになると、2 つの問題が発生します。最初の方が簡単ですが、それほど重要ではありません。ログが大きければ大きいほど、回復プロセスは回復するためにログ内のすべてのトランザクションを順番に再生する必要があります。第 2 の問題の方が問題です。ログがいっぱい（またはほぼ満杯）になるとディスクにコミットすることができなくなり、ファイルシステムが「役に立たない」状態になります。

これらの問題に対処するために、ジャーナリングファイルシステムでは、ログを循環データ構造として扱い、何度も繰り返し使用します。このため、ジャーナルは circular log（循環ログ）と呼ばれることがあります。これを行うには、ファイルシステムがチェックポイントの後で何らかのアクションを実行する必要があります。具体的には、トランザクションがチェックポイントされると、ファイルシステムはジャーナル内で占有していたスペースを解放し、ログスペースを再利用できるようにする必要があります。この目的を達成する方法はたくさんあります。たとえば、ジャーナルスーパー ブロック内のログ内で最も古いもの、チェックポイントのな

い最新のトランザクションを単にマークすることができます。他のすべてのスペースはフリーです。ここに図解があります：



journal superblock(ジャーナルスーパー ブロック)(メインファイルシステムのスーパー ブロックと混同しないでください)では、ジャーナリングシステムは、チェックポイントされていないトランザクションを知るための十分な情報を記録します。したがって、循環モデルのログを再利用することでリカバリ時間を短縮することを可能にします。そして、私たちは基本的なプロトコルにもう一つのステップを追加します：

1. Journal write(ジャーナルライト)

トランザクションの内容(TxBと更新内容を含む)をログに書き込みます。これらの書き込みが完了するのを待ちます。

2. Journal commit(ジャーナルコミット)

トランザクションコミットブロック(TxEを含む)をログに書き込みます。書き込みが完了するまで待ちます。

トランザクションは現在コミットされています。

3. checkpoint

アップデートの内容をファイルシステム内の最終的な場所に書き込みます。

4. free

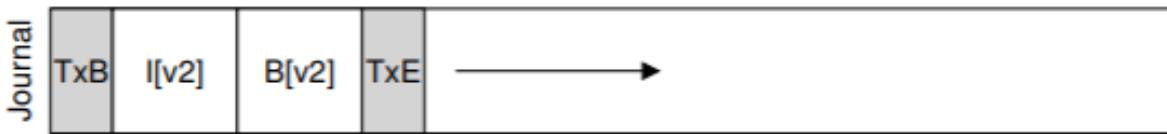
その後、ジャーナルのスーパー ブロックを更新して、ジャーナル内のトランザクションをフリーでマークしてください。

したがって、私たちは最終的な data journaling protocol(データジャーナリングプロトコル)を持っています。しかし、まだ問題が残っています。各データブロックをディスクに2回書き込んでいます。これは、システムクラッシュのような稀な問題に対して支払うにはコストがかかり過ぎています。データを2回書き込まずに一貫性を保つ方法を見つけられるでしょうか？

Metadata Journaling

リカバリは速くなりました(ジャーナルをスキャンし、ディスク全体をスキャンするのではなく、いくつかのトランザクションを再生する)が、ファイルシステムの通常の動作はわれわれが望むよりも遅くなります。特に、ディスクへの書き込みごとに、最初にジャーナルに書き込むので、書き込みトラフィックが倍増します。この倍増は、シーケンシャル・ライト・仕事量中に特に苦労し、ドライブのピーク書き込み帯域幅の半分で処理を進めます。さらに、ジャーナルへの書き込みとメインファイルシステムへの書き込みとの間には、コストがかかるシークがあり、いくつかの仕事量に顕著なオーバーヘッドが加わります。

すべてのデータブロックをディスクに2回書き込むコストが高いため、パフォーマンスを向上させるためにいくつかの方法を試しました。例えば、上で説明したジャーナリングモードは、すべてのユーザーデータ(ファイルシステムのメタデータを追加)をジャーナリングするため、(Linux ext3のように)データジャーナリングと呼ばれることがあります。ジャーナリングのより単純な(そしてより一般的な)形式は、ordered journaling(順序付けされたジャーナリング)(または単にメタデータジャーナリング)と呼ばれることもありますが、ユーザーデータがジャーナルに書き込まれない点を除いてほぼ同じです。したがって、上記と同じ更新を実行すると、次の情報がジャーナルに書き込まれます。



以前にログに書き込まれたデータブロック Db は、余分な書き込みを避けてファイルシステムの適切な場所に書きされます。ディスクへの I/O トラフィックのほとんどがデータであるため、データを 2 回書き込まないことは、ジャーナリングの I/O 負荷を大幅に削減します。しかし、変更は面白い質問を提起します：いつデータブロックをディスクに書き込むべきですか？

問題をよりよく理解するためにファイルの例を追加してみましょう。更新は I[v2]、B[v2]、Db の 3 つのブロックで構成されます。最初の 2 つはメタデータであり、ログに記録され、チェックポイントされます。後者はファイルシステムに一度だけ書きされます。Db をいつディスクに書き込む必要がありますか？それは重要ですか？

判明したように、データ書き込みの順序付けは、メタデータだけのジャーナリングに関して重要です。たとえば、トランザクション (I[v2] と B[v2] を含む) が完了した後に Db をディスクに書き込むとどうなりますか？残念ながら、この方法には問題があります。ファイルシステムは一貫していますが、私は I[v2] がガーベージデータを指してしまうことになります。具体的には、I[v2] と B[v2] が書かれていますが、Db がそれをディスクに作られていない場合を考えてみてください。ファイルシステムは次にリカバリを試みます。なぜなら、Db がログに記録されていないからです。そのため、ファイルシステムは I[v2] と B[v2] への書き込みを再生し、(ファイルシステムメタデータの観点から) 一貫したファイルシステムを生成します。しかし、I[v2] は、ガーベージデータを指しています。すなわち、Db の先頭にあったスロットにあったものが何であれです。

このような状況が起こらないようにするために、いくつかのファイルシステム (Linux ext3 など) は、関連するメタデータがディスクに書き込まれる前に、(通常のファイルの) データブロックをディスクに書き込みます。具体的には、プロトコルは次のとおりです。

1. Data write(データ書き込み)

データを最終的な場所に書き込む。完了するのを待ちます (待機はオプションです；詳細は下記参照)。

2. Journal metadata write(ジャーナルメタデータの書き込み)

書き込み開始ブロックとメタデータをログに書き込みます。書き込みが完了するのを待ちます。

3. Journal commit(ジャーナルコミット)

トランザクションコミットブロック (TxE を含む) をログに書き込みます。書き込みが完了するまで待ちます。

トランザクション (データを含む) は現在コミットされています。

4. Checkpoint metadata(チェックポイントのメタデータ)

メタデータ更新の内容をファイルシステム内の最終的な場所に書き込みます。

5. Free

その後、ジャーナルスーパー ブロックで取引をフリーでマークします。

データの書き込みを最初に強制することで、ファイルシステムはポインタがゴミを指し示すことがないことを保証することができます。実際には、「オブジェクトが参照する前に参照されるオブジェクトを書く」というこのルールは、クラッシュ一貫性のコアであり、他のクラッシュ一貫性スキーム GP94 によってさらに悪用されます。

ほとんどのシステムでは、完全なデータジャーナリングよりもメタデータジャーナリング (ext3 の注文ジャーナルに似ています) が一般的です。たとえば、Windows NTFS と SGI の XFS は両方とも、ある形式のメタデータジャーナリングを使用します。Linux ext3 では、data モード、ordered モード、unordered モードのいずれかを選択できます (unordered モードでは、いつでもデータを書き込むことができます)。これらのモードはすべてメタデータの一貫性を保ちます。これらはデータのセマンティクスが異なります。

最後に、上記のプロトコルに示されているように、ジャーナルへの書き込みを発行する前に (ステップ 2)、

データ書き込みを完了させる(ステップ1)ことは正確である必要はないことに注意してください。具体的には、トランザクション開始ブロックとメタデータをジャーナルに書き込むことを発行することは問題ありません。唯一の実際の要件は、ジャーナルコミットブロックの発行前にステップ1と2が完了していることです(ステップ3)。

Tricky Case: Block Reuse

興味深いコーナーケースがいくつかあり、ジャーナリングをもっと難しくし、議論する価値があります。それらの多くはブロックの再利用を中心に展開されています。ext3の主力の1人であるStephen Tweedie氏は次のように述べています。「システム全体のなんとなく悲惨な部分は何ですか?... ファイルの削除です。削除に関係することはすべて業が深いです。削除とは何か... ブロックが削除されてから再割り当てされると、何が起こるかという悪夢があります。」[T00]

Tweedieの具体的な例は次のとおりです。何らかの形式のメタデータジャーナリングを使用しているとします(ファイルのデータブロックはジャーナリングされません)。fooというディレクトリがあるとしましょう。ユーザーはfooにエントリを追加し(ファイルを作成するなど)、fooの内容(ディレクトリはメタデータとみなされるため)がログに書き込まれます。fooディレクトリのデータの場所がブロック1000であると仮定します。したがって、ログには次のような内容が含まれます。

Journal	TxB id=1	I[foo] ptr:1000	D[foo] [final addr:1000]	TxE id=1	→
---------	-------------	--------------------	-----------------------------	-------------	---

この時点で、ユーザーはディレクトリ内だけでなくディレクトリ自体もすべて削除し、ブロック1000を解放して再利用します。最後に、ユーザーは新しいファイル(foobarなど)を作成し、fooに属していた同じブロック(1000)を再利用します。foobarのiノードは、データと同様にディスクにコミットされます。ただし、メタデータジャーナリングが使用されているため、foobarのiノードだけがジャーナルにコミットされます。ファイルfoobar内のブロック1000の新しく書き込まれたデータはジャーナリングされません。

Journal	TxB id=1	I[foo] ptr:1000	D[foo] [final addr:1000]	TxE id=1	TxB id=2	I[foobar] ptr:1000	TxE id=2	→
---------	-------------	--------------------	-----------------------------	-------------	-------------	-----------------------	-------------	---

今、クラッシュが発生し、これらの情報はすべてログに残っていると仮定します。再生中、回復プロセスは、ブロック1000におけるディレクトリデータの書き込みを含むログ内のすべてを単に再生します。この再生は古いディレクトリの内容で現在のファイルfoobarのユーザデータを上書きします!明らかに、これは正しいリカバリアクションではありません。そして、確実にfoobarファイルを読むときにユーザは驚いてしまいます。

この問題にはいくつかの解決策があります。たとえば、前記ブロックの削除がジャーナルからチェックポイントが外に出るまで、ブロックを再使用することはできないようにすればよいです。Linux ext3が代わりに使っているのは、新しいタイプのレコードをジャーナルに追加することです。revoke record(取り消しレコード)と呼ばれます。上記の場合、ディレクトリを削除すると、取り消しレコードがジャーナルに書き込まれます。ジャーナルを再生するとき、システムは最初にそのような取り消しレコードをスキャンします。そのような取り消されたデータは決して再生されないので、上記の問題は回避されます。

Wrapping Up Journaling: A Timeline

ジャーナリングの議論を終了する前に、議論したプロトコルをそれぞれのタイムラインで要約します。図42.1にメタデータだけでなくデータをジャーナリングする場合のプロトコルを示します。図42.2に、メタ

データのみをジャーナリングする場合のプロトコルを示します。

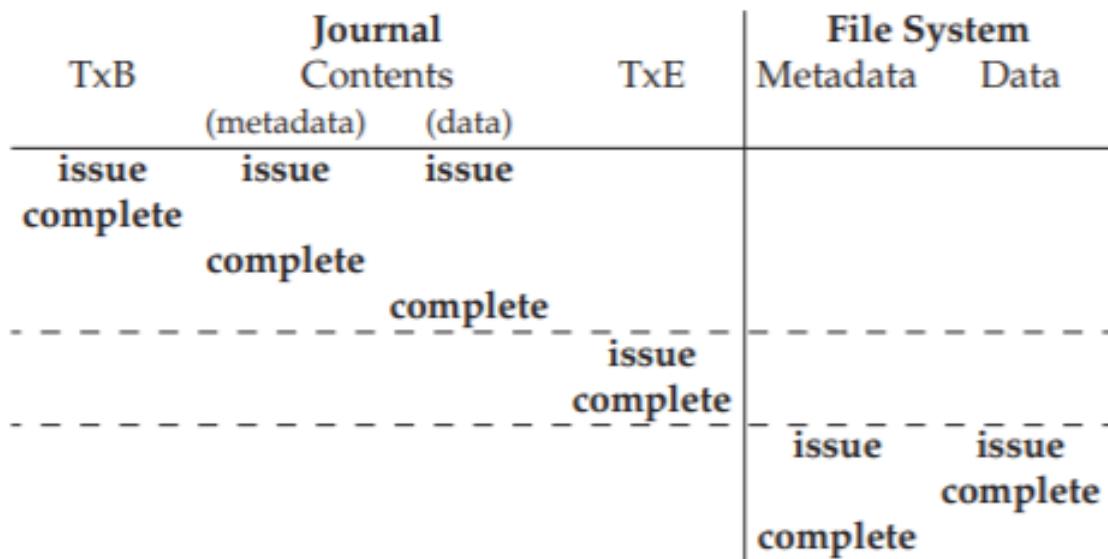


Figure 42.1: Data Journaling Timeline

各図では、時間が下方向に増加し、図の各行は書き込みが発行されるか完了するかの論理時間を示しています。たとえば、データジャーナリングプロトコル(図42.1)では、トランザクション開始ブロック(TxB)の書き込みとトランザクションの内容を論理的に同時に発行することができ、したがって任意の順序で完了することができます。しかし、トランザクション終了ブロック(TxE)への書き込みは、前の書き込みが完了するまで発行してはいけません。同様に、データブロックとメタデータブロックへのチェックポイント書き込みは、トランザクション終了ブロックがコミットされるまで開始できません。水平方向の破線は、書き込み注文要件を遵守しなければならない場所を示しています。

メタデータジャーナリングプロトコルについても同様のタイムラインが表示されます。データ書き込みは、トランザクションへの書き込みが開始され、ジャーナルの内容と同時に論理的に発行されることに注意してください。ただし、トランザクション終了が発行される前に、それが発行され完了する必要があります。

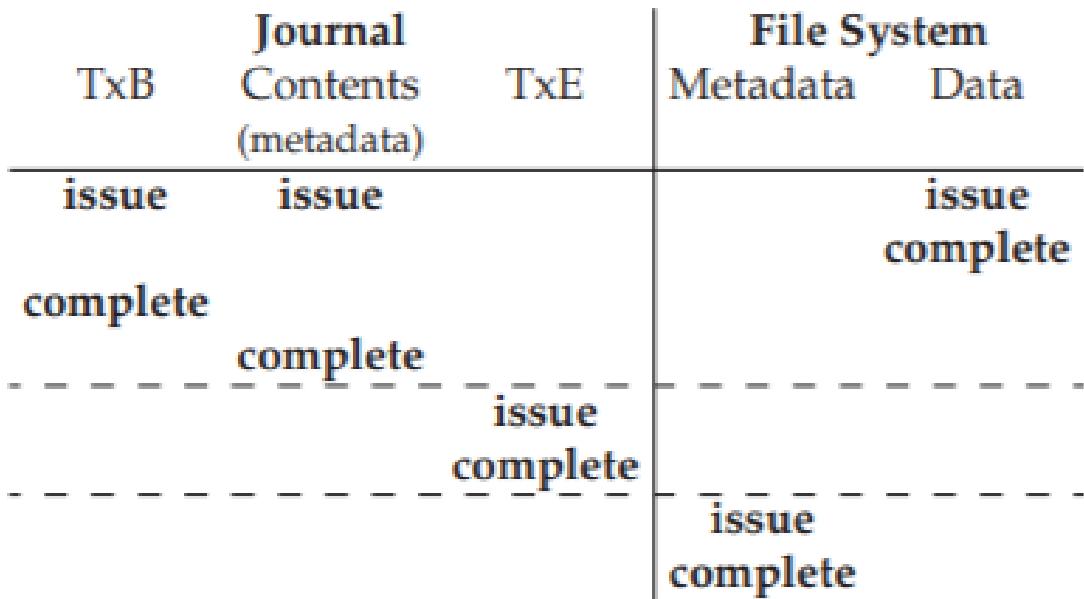


Figure 42.2: Metadata Journaling Timeline

最後に、タイムラインの各書き込みについてマークされた完了時間は任意であることに注意してください。実際のシステムでは、I/O サブシステムによって完了時刻が決定され、書き込みを並べ替えてパフォーマンスを向上させることができます。命令の唯一の保証は、プロトコルの正確さのために強制されなければならないものです(図の水平の破線で示されています)。

42.4 Solution #3: Other Approaches

これまで、ファイルシステムのメタデータを一貫して保つための 2 つのオプションである `fsck` に基づく遅延アプローチ、ジャーナリングと呼ばれるより積極的なアプローチについて説明しました。しかしながら、これらは唯一の 2 つのアプローチではありません。ソフトアップデート [GP94] として知られているこのようなアプローチの 1 つが、Ganger と Patt によって導入されました。

このアプローチでは、ファイルシステムへのすべての書き込みを慎重に指示して、ディスク上の構造が一貫性のない状態にならないようにします。たとえば、参照されているデータブロックを、それを参照している `inode` より前にディスクに書き込むことで、`inode` がゴミを指していないことを保証できます。同様なルールでファイルシステムのすべての構造を得ることができます。

しかし、ソフトアップデートを実装することは難しいことです。上記のジャーナリング層は、正確なファイルシステム構造の知識をほとんど持たずに実装することができますが、ソフトアップデートは各ファイルシステムデータ構造の複雑な知識を必要とし、システムにかなりの複雑さを加えます。

別のアプローチは、コピーオンライト (yes, COW) と呼ばれ、Sun の ZFS [B07] を含む多くの一般的なファイルシステムで使用されています。この手法は、ファイルやディレクトリを上書きすることはありません。むしろ、ディスク上の未使用の場所に新しい更新を置きます。多数の更新が完了した後、COW ファイルシステムは、ファイルシステムのルート構造を反転して、新しく更新された構造へのポインタを含みます。そうすることで、ファイルシステムの整合性が保たれます。このテクニックについては、今後の章で log structured file system(LFS) について議論するときに詳しく学習します。LFS は COW の初期の例です。

もう一つのアプローチはウィスコンシンで開発したアプローチです。backpointer based consistency(または BBC) と題されたこの技法では、書き込みの間に順序付けは行われません。整合性を達成するために、シス

ム内のすべてのブロックに追加のバックポインタが追加されます。たとえば、各データブロックには、それが属する i ノードへの参照があります。ファイルにアクセスするとき、ファイルシステムは、フォワードポインタ (例えば、inode または直接ブロック内のアドレス) がそれを参照するブロックを指し示しているかどうかをチェックすることによって、ファイルが一貫しているかどうかを判定することができます。その場合、すべてが安全にディスクに到達していなければならず、したがってファイルは一貫しています。そうでなければ、ファイルは不整合であり、エラーが返されます。ファイルシステムにバックポインタを追加することで、新しい形式の遅延クラッシュ整合性を達成することができます [C + 12]。

最後に、ジャーナルプロトコルがディスク書き込みが完了するのを待たなければならない回数を減らす手法も検討しました。楽観的な衝突の一貫性 [C + 13] が付与されたこの新しい手法では、可能な限り多くのディスクへの書き込みが行われ、トランザクションチェックサム [P + 05] の一般化された形式や他のいくつかの手法を使用して、発生した不整合性を検出します。一部の仕事量では、これらの楽観的な手法により、パフォーマンスが向上します。しかし、本当にうまく機能するには、わずかに異なるディスクインターフェースが必要です [C + 13]。

42.5 Summary

クラッシュの一貫性の問題を導入し、この問題を攻撃するさまざまなアプローチについて説明しました。ファイルシステムチェッカーを構築する古いアプローチは機能しますが、現代のシステムでは回復が遅すぎる可能性があります。したがって、多くのファイルシステムでジャーナリングが使用されています。ジャーナリングは、O(ディスク容量の大きさ) から O(サイズのログ) までのリカバリ時間を短縮し、クラッシュや再起動後のリカバリを大幅に高速化します。このため、現代の多くのファイルシステムではジャーナリングが使用されています。ジャーナリングはさまざまな形で提供されることがあります。最も一般的に使用されるのは ordered metadata journaling で、ジャーナルへのトラフィック量を削減しながら、ファイルシステムのメタデータとユーザーデータの両方に対して合理的な一貫性の保証を維持します。

参考文献

[B07] “ZFS: The Last Word in File Systems”

Jeff Bonwick and Bill Moore

Available: http://www.ostep.org/Citations/zfs_last.pdf

ZFS uses copy-on-write and journaling, actually, as in some cases, logging writes to disk will perform better.

[C+12] “Consistency Without Ordering”

Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

FAST '12, San Jose, California

A recent paper of ours about a new form of crash consistency based on back pointers. Read it for the exciting details!

[C+13] “Optimistic Crash Consistency”

Vijay Chidambaram, Thanu S. Pillai, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

SOSP '13, Nemacolin Woodlands Resort, PA, November 2013

Our work on a more optimistic and higher performance journaling protocol. For workloads that call fsync() a lot, performance can be greatly improved.

[GP94] “Metadata Update Performance in File Systems”

Gregory R. Ganger and Yale N. Patt

OSDI '94

A clever paper about using careful ordering of writes as the main way to achieve consistency. Implemented later in BSD-based systems.

[G+08] “SQCK: A Declarative File System Checker”

Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
OSDI ’08, San Diego, California

Our own paper on a new and better way to build a file system checker using SQL queries. We also show some problems with the existing checker, finding numerous bugs and odd behaviors, a direct result of the complexity of fsck.

[H87] “Reimplementing the Cedar File System Using Logging and Group Commit”

Robert Hagmann

SOSP ’87, Austin, Texas, November 1987

The first work (that we know of) that applied write-ahead logging (a.k.a. journaling) to a file system.

[M+13] “ffsck: The Fast File System Checker”

Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

FAST ’13, San Jose, California, February 2013

A recent paper of ours detailing how to make fsck an order of magnitude faster. Some of the ideas have already been incorporated into the BSD file system checker [MK96] and are deployed today.

[MK96] “Fsck - The UNIX File System Check Program”

Marshall Kirk McKusick and T. J. Kowalski

Revised in 1996

Describes the first comprehensive file-system checking tool, the eponymous fsck. Written by some of the same people who brought you FFS.

[MJLF84] “A Fast File System for UNIX”

Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry

ACM Transactions on Computing Systems.

August 1984, Volume 2:3

You already know enough about FFS, right? But yeah, it is OK to reference papers like this more than once in a book.

[P+05] “IRON File Systems”

Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

SOSP ’05, Brighton, England, October 2005

A paper mostly focused on studying how file systems react to disk failures. Towards the end, we introduce a transaction checksum to speed up logging, which was eventually adopted into Linux ext4.

[PAA05] “Analysis and Evolution of Journaling File Systems”

Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

USENIX ’05, Anaheim, California, April 2005

An early paper we wrote analyzing how journaling file systems work.

[R+11] “Coerced Cache Eviction and Discreet-Mode Journaling”

Abhishek Rajimwale, Vijay Chidambaram, Deepak Ramamurthi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

DSN ’11, Hong Kong, China, June 2011

Our own paper on the problem of disks that buffer writes in a memory cache instead of forcing them to disk, even when explicitly told not to do that! Our solution to overcome this problem: if you want A to

be written to disk before B, first write A, then send a lot of “dummy” writes to disk, hopefully causing A to be forced to disk to make room for them in the cache. A neat if impractical solution.

[T98] “Journaling the Linux ext2fs File System”

Stephen C. Tweedie

The Fourth Annual Linux Expo, May 1998

Tweedie did much of the heavy lifting in adding journaling to the Linux ext2 file system; the result, not surprisingly, is called ext3. Some nice design decisions include the strong focus on backwards compatibility, e.g., you can just add a journaling file to an existing ext2 file system and then mount it as an ext3 file system.

[T00] “EXT3, Journaling Filesystem”

Stephen Tweedie

Talk at the Ottawa Linux Symposium, July 2000

olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html

A transcript of a talk given by Tweedie on ext3.

[T01] “The Linux ext2 File System”

Theodore Ts’o, June, 2001.

Available: <http://e2fsprogs.sourceforge.net/ext2.html>

A simple Linux file system based on the ideas found in FFS. For a while it was quite heavily used; now it is really just in the kernel as an example of a simple file system.

43 Log-structured File Systems

90年代初頭、John Ousterhout教授と大学院生 Mendel Rosenblum が率いるバークレー校のグループが、ログ構造ファイルシステム [RO91] と呼ばれる新しいファイルシステムを開発しました。そうする動機は、以下の観察に基づいていました。

- システムメモリが増えているメモリが大きくなるにつれ、より多くのデータをメモリにキャッシュすることができます。より多くのデータがキャッシュされるにつれて、ディスクトラフィックはますますキャッシュからの読み取りが処理されるため、書き込みから構成されます。したがって、ファイルシステムのパフォーマンスは主にその書き込みパフォーマンスによって決まります。
- ランダム I/O パフォーマンスとシーケンシャル(順次)I/O パフォーマンスの間に大きなギャップがありますハードドライブの転送帯域幅は、長年にわたって大きく増加しています [P98]。より多くのビットがドライブの表面にパックされると、アクセスするときの帯域幅が増加します。しかし、シークおよび回転遅延のコストは緩やかに低下しています。安価で小型のモーターを高速で回転させたり、ディスクアームをより迅速に動かすことは困難です。したがって、シーケンシャルにディスクを使用できる場合は、シークとローテーションを引き起こすアプローチよりも大きなパフォーマンス上の利点があります。
- 既存のファイルシステムは、多くの一般的な仕事量でパフォーマンスが低下します。たとえば、FFS [MJLF84] は、FFS [MJLF84] は、一つのサイズブロックの新しいファイルを作成するために多数の書き込みを実行します。新しい inode に 1 つ、inode のビットマップの更新のために 1 つ、ファイルが存在するディレクトリデータブロックのために 1 つ、ディレクトリ i ノードを更新するために 1 つ、一部の新しいファイルの新しいデータブロックのために 1 つ、割り当てられたデータブロックをマークするためのデータビットマップに 1 つがあります。したがって、FFS はこれらすべてのブロックを同じブロックグループ内に配置しますが、FFS は多くの短いシークとそれに続く回転遅延を招きます。そのため、パフォーマンスはピークシーケンシャル帯域幅をはるかに下回ります。
- ファイルシステムは RAID 対応ではありませんたとえば、RAID-4 と RAID-5 の両方で、1 つのブロックへの論理書き込みによって 4 つの物理 I/O が発生する小さな書き込み問題が発生します。既存のファイルシステムは、この最悪の場合の RAID 書き込み動作を回避しようとしません。

TIP: DETAILS MATTER

すべての興味深いシステムは、いくつかの一般的なアイデアと多くの詳細から構成されています。時には、あなたがこれらのシステムについて学んでいる時、あなたは自分自身に”おっ！、良いアイデア発見！ 後は詳細考えるだけだ”と思うでしょう。また、これを使って、物事の実際の仕組みを半分だけ学べばいいと思うでしょう。しかし、このようなことをしないでください！ 多くの場合、詳細は重要です。LFS で見るよう、一般的な考え方を理解しやすいですが、実際には動作するシステムを構築するためには、すべての難しいケースを考える必要があります。

理想的なファイルシステムは、書き込みパフォーマンスに重点を置き、ディスクの帯域幅を使用します。また、データを書き出すだけでなく、ディスク上のメタデータ構造を頻繁に更新する一般的な仕事量でも、パフォーマンスが向上します。最後に、RAID とシングルディスクでうまくいくものです。

導入された Rosenblum と Ousterhout の新しいタイプのファイルシステムは、Log structured File System の略で LFS と呼ばれていました。ディスクに書き込むとき、LFS は最初にすべての更新(メタデータを含む)をメモリセグメントにバッファリングします。セグメントがいっぱいになると、ディスクの未使用部分に 1 回の長いシーケンシャル転送でディスクに書き込まれます。LFS は既存のデータを上書きすることはありま

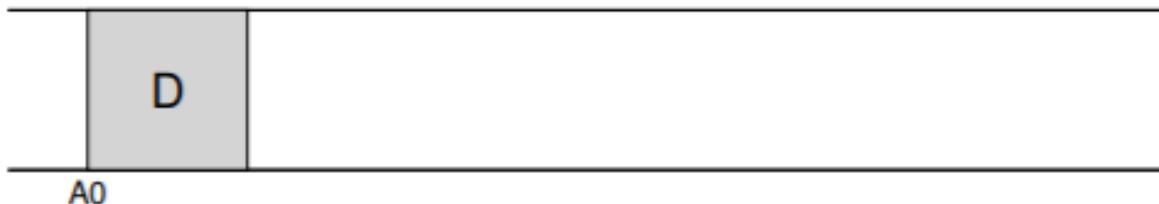
せんが、むしろ常にフリーな場所にセグメントを書き込みます。セグメントが大きいため、ディスク（またはRAID）が効率的に使用され、ファイルシステムのパフォーマンスがその限界に近づきます。

THE CRUX: HOW TO MAKE ALL WRITES SEQUENTIAL WRITES?

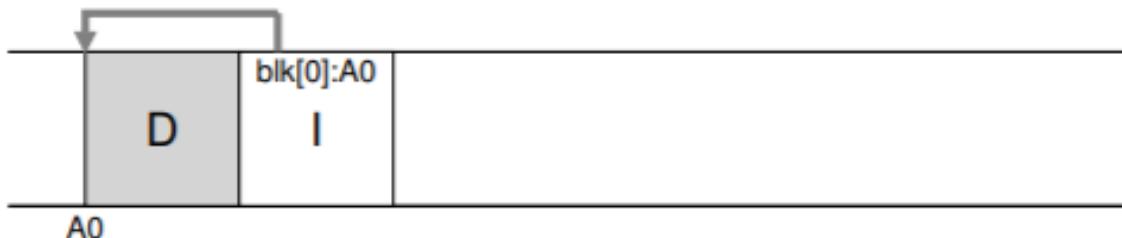
ファイルシステムはどのようにしてすべての書き込みを順次書き込みに変換できますか？読み取りの場合、このタスクは不可能です。読み取るブロックはディスク上のどこにあってもかまいません。しかし、書き込みの場合、ファイルシステムには常に選択肢があります。これはまさに私たちが開発しようとしている選択肢です。

43.1 Writing To Disk Sequentially

したがって、私たちは最初の課題を抱えています。ファイルシステム状態へのすべての更新をディスクへの一連の順次書き込みに変換するにはどうすればよいですか？これをよりよく理解するために、簡単な例を使用してみましょう。データブロック D をファイルに書き込んでいるとします。データブロックをディスクに書き込むと、次のオンディスクレイアウトが発生し、D はディスクアドレス A0 に書き込まれます。



ただし、ユーザーがデータブロックを書き込むときは、ディスクに書き込まれるデータだけではありません。更新が必要な他のメタデータもあります。この場合、ファイルの i ノード (I) をディスクに書き込んで、データブロック D を指します。ディスクに書き込むと、データブロックと i ノードは次のようにになります (注意してほしいのは、i ノードはデータブロックと同じくらいに大きく見えます。これ一般的ではないケースです。大部分のシステムでは、データブロックは 4 KB のサイズですが、inode はだいたい 128 バイトくらいで小さいです)。



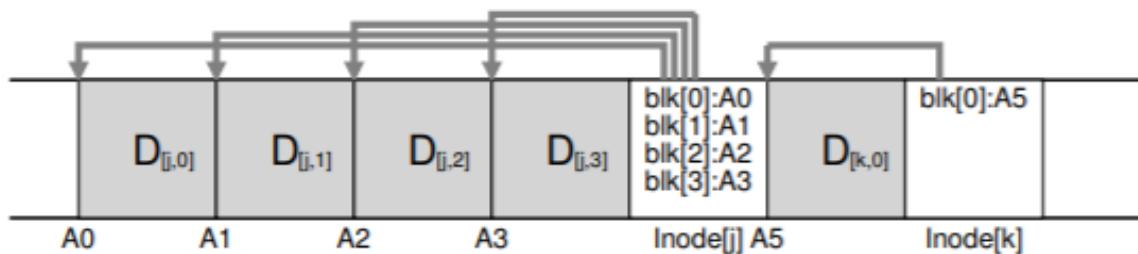
43.2 Writing Sequentially And Effectively

残念ながら、ディスクへの順次書き込みは、効率的な書き込みを保証するのに十分ではありません。たとえば、時間 T に A をアドレス指定する单一のブロックを書き込無ことを考えてみてください。そのとき、ちょっと待ってからアドレス A + 1(次のブロックアドレスが順番に) に書き込まれますが、その時刻は T + δ です。最初の書き込みと 2 番目の書き込みの間に、残念ながら、ディスクが回転しています。2 回目の書き込みを発行すると、コミットされる前に回転のためにほとんどの時間待機します (具体的には、回転に時間 T_rotation がかかる場合、ディスクは T_rotation - δ を待ってからディスク表面に 2 回目の書き込みをコミットできます)。したがって、順番にディスクに書き込むだけでは、ピークパフォーマンスを達成するには不十分であることがうかがえます。むしろ、良好な書き込みパフォーマンスを達成するためには、ドライブに多数の連續書き

込み（または1回の大きな書き込み）を発行する必要があります。

この目的を達成するために、LFSは書き込みバッファリングと呼ばれる古くからの技術を使用しています（実際、この考え方はコンピューティングの歴史の中でも早い時期に発明された可能性が高いです。その利点については Solworth と Orji [SO90] を参照してください。その潜在的な危険については、Mogul [M94] を参照してください。）ディスクに書き込む前に、LFSはメモリ内の更新を追跡します。十分な数の更新を受信すると、ディスクに一度に書き込むので、ディスクの効率的な使用が保証されます。

LFSが一度に書き込む更新の大部分は、セグメントの名前で参照されます。この用語はコンピュータシステムでは過度に使用されていますが、ここでは LFS が書き込みをグループ化するために使用する大規模なチャンクを意味します。したがって、ディスクに書き込むとき、LFS はメモリ内セグメントの更新をバッファし、そのセグメントを一度にディスクに書き込みます。セグメントが十分大きければ、これらの書き込みは効率的になります。ここでは、LFS が 2 セットの更新を小さなセグメントにバッファリングする例を示します。実際のセグメントは大きくなります（数 MB）。最初の更新は、ファイル j への 4 回のブロック書き込みです。2 番目のブロックはファイル k に追加される 1 つのブロックです。LFS は、7 ブロックのセグメント全体を一度にディスクにコミットします。これらのブロックの結果として得られるディスク上のレイアウトは次のとおりです。



43.3 How Much To Buffer?

ディスクに書き込む前に LFS が何回更新すべきでしょうかという質問を提起します。答えはもちろん、ディスクそのものです。具体的には転送速度と比較して位置決めのオーバーヘッドがどれほど高いかによって異なります。同様の分析については、FFS の章を参照してください。例えば、各書き込みの前に位置決め（すなわち、回転およびシークオーバーヘッド）がおよそ $T_{position}$ 秒かかると仮定します。さらに、ディスク転送速度は R_{peak} MB/s であると仮定します。このようなディスクで実行するときに LFS が書き込む前にどれくらいバッファするべきですか？

これについて考える方法は、書き込むたびにオーバーヘッドである位置決めコストの固定費を支払うことです。したがって、その費用を償却するためには、どれくらい書く必要がありますか？書き込みが多いほど、（明らかに）良くなり、ピーク帯域幅を近づけることができます。具体的な答えを得るには、 D MB を書き出すと仮定しましょう。このデータのチャンク (T_{write}) を書き出す時間は、位置決め時間 $T_{position}$ プラス転送時間 $D(D / R_{peak})$ 、または

$$T_{write} = T_{position} + \frac{D}{R_{peak}} \quad (43.1)$$

したがって、書き込まれたデータの量を書き込む合計時間で割った有効な書き込み速度 ($R_{effective}$) は次のようにになります。

$$R_{\text{effective}} = \frac{D}{T_{\text{write}}} = \frac{D}{T_{\text{position}} + \frac{D}{R_{\text{peak}}}}. \quad (43.2)$$

私たちが興味を持っているのは、効率レート ($R_{\text{effective}}$) をピークレートに近づけることです。具体的には、効率レートをピークレートの何分の F にする必要があります ($0 < F < 1$ 、典型的な F は 0.9、ピークレートの 90 %) 数学的形式では、これは、 $R_{\text{effective}} = F \times R_{\text{peak}}$ を望むことを意味します。この時点での D について解くことができます

$$R_{\text{effective}} = \frac{D}{T_{\text{position}} + \frac{D}{R_{\text{peak}}}} = F \times R_{\text{peak}} \quad (43.3)$$

$$D = F \times R_{\text{peak}} \times \left(T_{\text{position}} + \frac{D}{R_{\text{peak}}} \right) \quad (43.4)$$

$$D = (F \times R_{\text{peak}} \times T_{\text{position}}) + (F \times R_{\text{peak}} \times \frac{D}{R_{\text{peak}}}) \quad (43.5)$$

$$D = \frac{F}{1 - F} \times R_{\text{peak}} \times T_{\text{position}} \quad (43.6)$$

例として、位置決め時間が 10 ミリ秒、ピーク転送速度が 100 MB/s のディスクを例に挙げてみましょう。ピークの 90 % ($F = 0.9$) の実効帯域幅が必要であると仮定します。この場合、 $D = 0.9 / 0.1 \times 100 \text{MB/s} \times 0.01 \text{秒} = 9 \text{MB}$ となります。いくつかの異なる値を試して、ピーク帯域幅に近づくためにバッファリングする必要があるかどうかを確認してください。ピークの 95 % に達するにはどれだけの量が必要ですか？ 99 % ではどうでしょうか？

43.4 Problem: Finding Inodes

LFS での inode の発見方法を理解するには、典型的な UNIX ファイルシステムで inode を見つける方法を簡単に見てみましょう。FFS や古い UNIX ファイルシステムのような典型的なファイルシステムでは、inodeを見つけるのは簡単です。なぜなら、それらは配列で編成され、固定された場所にディスク上に配置されるからです。たとえば、古い UNIX ファイルシステムでは、すべての i ノードがディスクの固定部分に保持されます。したがって、特定の inode を見つけるために、inode 番号と開始アドレスを指定すると、その inode 番号に inode のサイズを掛け、その値をディスク上の配列の開始アドレスに加えるだけで、正確なディスクアドレスを計算することができます。inode 番号を指定すると、配列ベースの索引付けは迅速かつ簡単です。

FFS は inode テーブルをチャンクに分割し、各シリンドグループ内に i ノードのグループを配置するため、FFS で i ノード番号が指定された i ノードを見つけることはわずかに複雑です。したがって、inode の各チャンクの大きさとそれぞれの開始アドレスを知る必要があります。その後、計算は同様で簡単です。

LFS では、人生はより困難です。どうしてでしょうか？ 私たちはすべてのディスクに i ノードを散らしてしまいました！ さらに悪いことに、上書きすることはありません。したがって、最新バージョンの inode(私たちが望むもの) は動いています。

43.5 Solution Through Indirection: The Inode Map

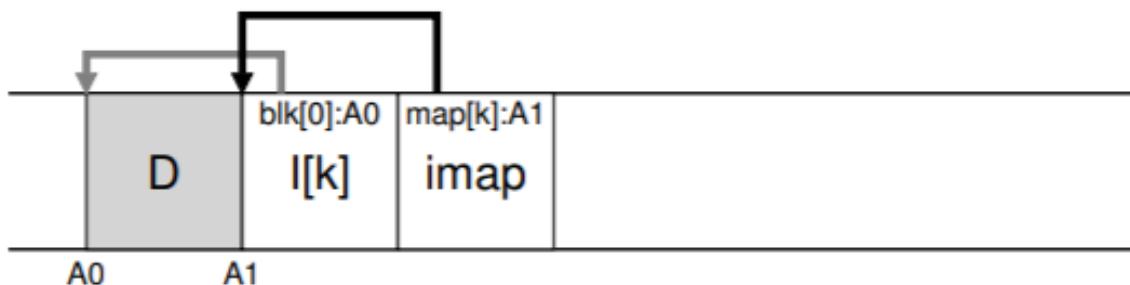
この問題を解決するために、LFS の設計者は、i ノードマップ (imap) と呼ばれるデータ構造を通じて、i ノード番号と i ノード間の間接レベルを導入しました。imap は inode 番号を入力として受け取り、最新のバ

ジョンの inode のディスクアドレスを生成する構造体です。したがって、エントリごとに 4 バイト (ディスクポインタ) の単純な配列として実装されることが多いと想像できます。inode がディスクに書き込まれるたびに、imap は新しい場所で更新されます。

残念ながら、imap は永続的 (すなわち、ディスクに書き込まれる) に保たれる必要があります。そのようにすることで、LFS はクラッシュ中の i ノードの位置を追跡することができ、必要に応じて動作します。したがって質問があります、ディスク上のどこに imap を置くべきですか？

それはもちろん、ディスクの固定された部分に生きることができます。残念なことに、頻繁に更新されると、ファイル構造の更新に続いて imap への書き込みが必要になるため、パフォーマンスが低下します (つまり、それぞれの更新と imap の固定場所とのディスクシークが多くなります)。

代わりに、LFS は inode マップのチャンクを、他のすべての新しい情報を書いている場所のすぐ隣に置きます。したがって、以下のように、データブロックをファイル k に追加するとき、LFS は実際には次のように、新しいデータブロック、その i ノード、i ノードマップの一部をすべて一緒にディスクに書き込みます。



この図では、imap と書かれているブロックに格納されている imap 配列の部分は、inode k がディスクアドレス A1 にあることを LFS に伝え、格納します。この i ノードは、そのデータブロック D がアドレス A0 にあることを LFS に指示します。

43.6 Completing The Solution: The Checkpoint Region

するどい読者であれば、ここで問題に気づいたかもしれません。どのようにして inode マップを見つけることができますか？ それらはディスク全体に広がっていますか？ 最終的には魔法はありません。ファイルルックアップを開始するには、ファイルシステムにディスク上の固定された既知の場所が必要です。

LFS はこれのために checkpoint region(CR) と呼ばれる固定された場所をディスク上に持っています。checkpoint region は、最新の inode マップのポインタ (すなわち、そのアドレス) を含み、したがって、最初に CR を読み取ることによって inode マップ部分を見つけることができます。checkpoint region は定期的に更新されるだけなので (30 秒ごとなど)、パフォーマンスに悪影響はありません。したがって、ディスク上のレイアウトの全体的な構造には checkpoint region(最新の inode マップを示す) が含まれています。i ノードマップ部分にはそれぞれ i ノードのアドレスが含まれています。inode は典型的な UNIX ファイルシステムと同じようにファイル (およびディレクトリ) を指しています。

次に、checkpoint region の例を示します (ディスクの先頭、アドレス 0 にあります)。また、单一の imap チャンク、inode、データブロックがあります。実際のファイルシステムはもちろん大きな CR(実際には 2 つあります。後で学んでいきます)、多くの imap チャンク、さらにはもっと多くの inode、データブロックなどがあります。



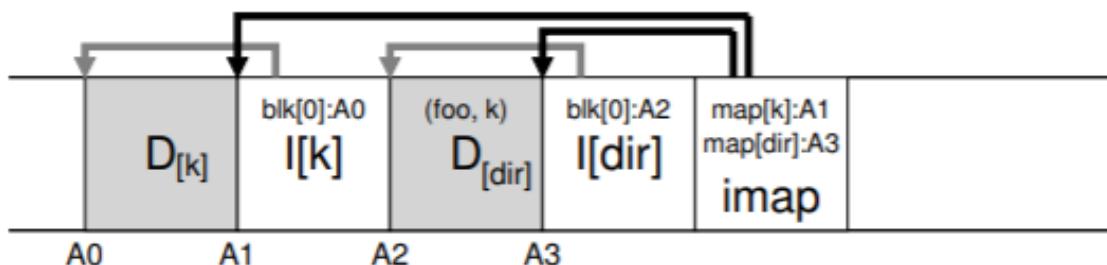
43.7 Reading A File From Disk: A Recap

LFS がどのように動作するのかを理解するために、ディスクからファイルを読み込むために必要なことを見てみましょう。私達が始めるものをメモリに何も持っていないと仮定してください。最初に読み込まなければならぬディスク上のデータ構造は、checkpoint region です。checkpoint region は、inode マップ全体に対するポインタ（すなわち、ディスクアドレス）を含み、したがって LFS は inode マップ全体を読み込み、メモリ内にキャッシュします。この時点以降、ファイルの i ノード番号が与えられると、LFS は単に imap の inode number から inode disk address へのマッピングを検索し、最新の inode のバージョンを読み込みます。この時点でファイルからブロックを読み込むには、直接ポインタ、間接ポインタ、二重間接ポインタを必要に応じてを使用して、LFS は典型的な UNIX ファイルシステムとまったく同じように処理します。一般的なケースでは、LFS はディスクからファイルを読み込む際に、通常のファイルシステムと同じ数の I/O を実行する必要があります。imap 全体がキャッシュされるので、読み込み中に LFS が行う余計な作業は、imap 内の inode のアドレスを検索することです。

43.8 What About Directories?

ここまででは、inode とデータブロックだけを考慮して、少し議論を単純化しました。しかし、ファイルシステム内のファイル（私たちの好きな偽のファイル名の 1 つである /home/remzi/foo など）にアクセスするには、いくつかのディレクトリにもアクセスする必要があります。では、LFS はディレクトリデータをどのように格納していますか？

幸いにも、ディレクトリ構造は基本的に古典的な UNIX ファイルシステムと同じです。ディレクトリは（名前、i ノード番号）マッピングの集まりにすぎません。たとえば、ディスク上にファイルを作成する場合、LFS は新しい i ノード、いくつかのデータ、ディレクトリデータ、このファイルを参照する i ノードを書き込む必要があります。LFS はディスク上で順番に実行することを覚えておいてください（しばらくの間、更新をバックファーリングした後）。したがって、ディレクトリにファイル foo を作成すると、ディスク上に次の新しい構造が生成されます。



i ノードマップの部分には、ディレクトリファイル dir と新しく作成されたファイル f の両方の場所に関する情報が含まれています。したがって、ファイル foo(i ノード番号 k) にアクセスする場合は、最初に inode マップ（通常はメモリにキャッシュされている）を調べて、ディレクトリ dir(A3) の i ノードの場所を探します。次に

ディレクトリの inode を読んで、ディレクトリのデータの場所 (A2) を取得します。このデータブロックを読み取ると、(foo, k) の名前から i ノード番号へのマッピングが得られます。その後、inode 番号 k(A1) の位置を見つけるために inode マップを再度参照し、最後にアドレス A0 で目的のデータブロックを読み込みます。

再帰的更新問題 [Z + 12] と呼ばれる、inode マップが解決するもう 1 つの重大な問題が LFS にあります。この問題は、(LFS のような) 決して更新されず、ディスク上の新しい場所に更新を移動するようなファイルシステムで発生します。

具体的には、i ノードが更新されるたびに、ディスク上のその位置が変わります。私たちが注意していなかつた場合、これはこのファイルを指すディレクトリへの更新を伴い、そのディレクトリの親ディレクトリの変更を強制していたでしょう。そしてそうなっていくと、ずっとファイルシステムツリーの上になるでしょう。

LFS は、この問題を賢明に inode マップで回避します。inode の場所が変更されても、変更はディレクトリ自体に反映されません。むしろ、imap 構造体は更新されますが、ディレクトリは同じ名前から番号へのマッピングを保持します。したがって、間接参照によって、LFS は再帰的な更新の問題を回避します。

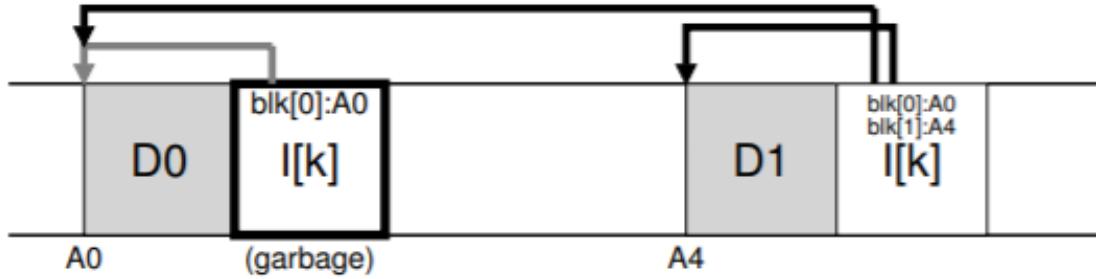
43.9 A New Problem: Garbage Collection

あなたは LFS に関する別の問題に気づいたかもしれません。ファイルの最新バージョン (その inode とデータを含む) をディスク上の新しい場所に繰り返し書き込みます。このプロセスは、効率的な書き込みを維持しながら、LFS が古いバージョンのファイル構造をディスク全体に分散させることを意味します。私たちは (むしろ無意識のうちに) これらの古いバージョンをゴミと呼んでいます。

たとえば、i ノード番号 k で参照される既存のファイルがある場合を考えてみましょう。このファイルは単一のデータブロック D0 を指しています。このブロックを更新し、新しい inode と新しいデータブロックの両方を生成します。その結果、LFS のディスク上のレイアウトは次のようにになります (簡単にするために imap やその他の構造体は省略しています。imap の新しいチャンクは新しい i ノードを指すようにディスクに書き込む必要があります)。



この図では、i ノードとデータブロックの両方に古いバージョン (左側のバージョン) と現在のバージョン (右側のバージョン) の 2 つのバージョンがあることがわかります。データブロックを (論理的に) 更新する単純な行為によって、いくつかの新しい構造が LFS によって永続化されなければならず、したがって、前記ブロックの古いバージョンはディスク上に残されなければならない。もう 1 つの例として、元のファイル k にブロックを追加するとします。この場合、新しいバージョンの inode が生成されますが、古いデータブロックはまだ inode によって指示されます。したがって、それはまだ生きており、現在のファイルシステムの大部分です。



では、古いバージョンの inode やデータブロックなどはどうしたらいいですか？ 古いバージョンのファイルを保持して、古いファイルのバージョンを復元できるようにすることもできます（たとえば、誤ってファイルを上書きまたは削除した場合など）。そのようなファイルシステムは、ファイルの異なるバージョンを追跡するため、バージョン管理ファイルシステムとして知られています。

しかし、代わりに LFS は最新のライブバージョンのみを保持します。したがって（バックグラウンドで）、LFS は定期的にファイルデータ、inode、およびその他の構造の古いバージョンを見つけて、それらをクリーニングする必要があります。このようにして、次の書き込みに使用するために、ディスク上のブロックを再びフリーにする必要があります。クリーニングのプロセスはガベージコレクションの一種であり、プログラムのために未使用のメモリを自動的に解放するプログラミング言語で発生するテクニックです。

以前は、LFS のディスクへの大きな書き込みを可能にするメカニズムと同じくらい重要なセグメントについても説明しました。それが判明したとき、彼らはまた、効果的なクリーンに不可欠です。LFS クリーナーが、クリーニング中に単一のデータブロック、i ノードなどを単に通過して解放した場合に起こることを想像してください。その結果、ディスク上の割り当てられたスペースの間にいくつかのフリーの穴が混在したファイルシステムが作成されました。LFS は連続して高性能なディスクに書き込むための大きな連続領域を見つけることができないため、書き込みパフォーマンスはかなり低下します。

その代わりに、LFS クリーナーはセグメント単位で動作します。したがって、後続の書き込みのために大きなスペースのチャンクをクリアします。基本的なクリーニングプロセスは次のように動作します。定期的に、LFS クリーナーは古い（部分的に使用されている）セグメントを読み込み、これらのセグメント内にどのブロックが存在するかを判断し、その中にライブブロックだけを含む新しいセグメントセットを書き出し、古いものを解放し、書き込みます。具体的には、クリーナーは既存の M 個のセグメントを読み込み、その内容を N 個の新しいセグメント ($N < M$) に圧縮し、N 個のセグメントを新しい場所にディスクに書き込みます。その後、古い M 個のセグメントは解放され、後続の書き込みのためにファイルシステムによって使用することができます。

しかし、今私たちは 2 つの問題を残しています。1 つはメカニズムです：LFS はどのセグメント内のどのブロックが生きているのか、どのブロックが死んでいるのかを教えてくれますか？ 2 番目はポリシーです：どのくらいの頻度でクリーナーを実行する必要がありますか、どのセグメントをクリーニングする必要がありますか？

43.10 Determining Block Liveness

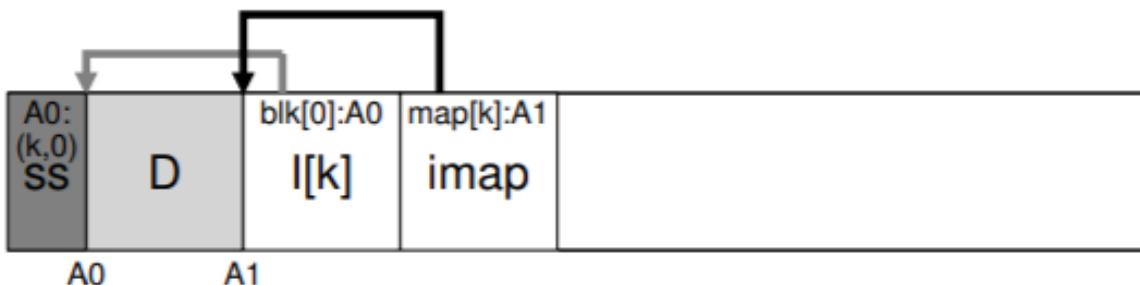
最初にそのメカニズムに取り組んでいます。オンディスクセグメント S 内のデータブロック D が与えられると、LFS は D が生存しているかどうかを判定できなければなりません。そうするために、LFS は、各ブロックを記述する各セグメントに少し余分な情報を追加します。具体的には、LFS は、各データブロック D について、その i ノード番号（それが所属するファイル）およびそのオフセット（ファイルのどのブロックであるか）を含みます。この情報は、segment summary block と呼ばれるセグメントの先頭の構造体に記録されます。

この情報があれば、ブロックが生存しているか死んでいるかを判断するのは簡単です。ディスク上のアドレ

ス A にあるブロック D については、segment summary block を調べ、その inode 番号 N とオフセット T を見つけます。次に、imap を調べて、N がどこにあるかを調べ、ディスクから N を読み込みます(おそらく、優れていればメモリ上に存在しています)。最後に、オフセット T を使用して、i ノード(または間接ブロック)を調べ、i ノードがこのファイルの T 番目のブロックがディスク上にあると考える場所を確認します。それがディスクアドレス A を正確に指す場合、LFS はブロック D がライブであると結論付けることができます。他の場所を指す場合、LFS は D が使用されていない(すなわち、死んでいる)と判断し、このバージョンがもはや必要ではないことを知ることができます。このプロセスの擬似コードの要約を以下に示します。

```
(N, T) = SegmentSummary[A];
inode = Read(imap[N]);
if (inode[T] == A)
    // block D is alive
else
    // block D is garbage
```

次に、セグメントサマリブロック(SS とマークされている)が、アドレス A0 のデータブロックが実際にオフセット 0 のファイル k の一部であることを記録するメカニズムを示す図です。k の imap をチェックすることによって、それが実際にその場所を指していることを確認してください。



LFS が生存率を決定するプロセスをより効率的にするために必要な、いくつかのショートカットがあります。たとえば、ファイルが切り捨てられたり削除されたりすると、LFS はそのバージョン番号を増やし、新しいバージョン番号を imap に記録します。また、オンディスクセグメントにバージョン番号を記録することで、LFS は、ディスク上のバージョン番号と imap 内のバージョン番号を単純に比較することにより、上記のより長いチェックを短絡することができます。

43.11 A Policy Question: Which Blocks To Clean, And When?

上記のメカニズムに加えて、LFS には、クリーニングの時期とクリーニングの対象となるブロックの両方を決定する一連のポリシーが含まれている必要があります。クリーニングする時期を決めるのは簡単です。定期的、アイドル時間中、ディスクがいっぱいであるときです。

どのブロックを清掃すべきかを決定することはより困難であり、多くの研究論文の主題となっています。元の LFS の論文 [RO91] では、著者はホットセグメントとコールドセグメントに分離しようとするアプローチを述べています。ホットセグメントは、内容が頻繁に上書きされるセグメントです。したがって、そのようなセグメントでは、より多くのブロックが(新しいセグメントで)上書きされ、使用のために解放されるように、そ

れをクリーニングする前に長い時間待つことが最善の方針です。対照的に、コールドセグメントは、デッドロックをいくつか持つことがあります、残りの内容は比較的安定しています。したがって、著者らは、コールドセグメントをより早くクリーニングし、ホットセグメントを後でクリーニングし、正確にそれを行うものを開発すべきであると結論づけています。しかし、ほとんどのポリシーと同様に、このポリシーは完全ではありません。後のアプローチでは、よりうまくいく方法が示されています [MR + 97]。

43.12 Crash Recovery And The Log

1つの最終的な問題：LFS がディスクに書き込んでいるときにシステムがクラッシュするとどうなりますか？以前の章でジャーナリングについて思い出してきたように、更新中のクラッシュはファイルシステムにとっては厄介なものであり、LFS も考慮する必要があります。

通常の操作では、LFS はセグメントに書き込みを行い（バッファリング）、その後（セグメントがいっぱいになるか、またはある程度の時間が経過したときに）セグメントをディスクに書き込みます。LFS はこれらの書き込みのログを作成します。すなわち、checkpoint region はヘッドおよびテールセグメントを指し、各セグメントは書き込まれる次のセグメントを指します。LFS は checkpoint region も定期的に更新します。これらの操作（セグメントへの書き込み、CR への書き込み）のいずれかでクラッシュが発生する可能性があります。LFS はこれらの構造への書き込み中のクラッシュをどのように処理しますか？

最初に 2 番目のケースを対処してみましょう。CR 更新が原子的に確実に行われるよう、LFS は実際にはディスクの両端に 1 つずつ 2 つの CR を保持し、交互に書き込みます。また、LFS は、i ノードマップやその他の情報への最新のポイントで CR を更新するときに注意するプロトコルを実装します。具体的には、最初にヘッダー（タイムスタンプ付き）、次に CR の本文、最後に最後のブロック（タイムスタンプ付き）を書き出します。CR の更新中にシステムがクラッシュした場合、LFS は一貫性のないタイムスタンプのペアを見ることでこれを検出できます。LFS は常に一貫性のあるタイムスタンプを持つ最新の CR を使用することを選択し、CR の一貫した更新が達成されます。

最初のケースに対処しましょう。LFS は 30 秒ごとに CR を書き込むので、ファイルシステムの最後の整合性のあるスナップショットはかなり古いかかもしれません。したがって、再起動時に LFS は checkpoint region、それが指す imap 部分、および後続のファイルとディレクトリを読み込むだけで簡単に回復できます。しかし、更新の最後の数秒は失われます。これを改善するために、LFS は、データベースコミュニティでロールフォワードと呼ばれる手法を使用して、これらのセグメントの多くを再構築しようとします。

基本的な考え方は、最後の checkpoint region から始まり、ログの終わりを見つけて（CR に含まれています）、それを使って次のセグメントを読み込み、そこに有効な更新があるかどうかを確認します。存在する場合、LFS はファイルシステムをそれに応じて更新し、最後のチェックポイント以降に書き込まれたデータとメタデータの多くのを回復します。詳細については、Rosenblum 賞を受賞した論文 [R92] を参照してください。

43.13 Summary

LFS は、ディスクを更新する新しい方法を導入しています。LFS は、場所のファイルを上書きするのではなく、常にディスクの未使用部分に書き込み、後で古い空間をクリーニングによって再利用します。このアプローチは、データベースシステムではシャドウページング [L77] と呼ばれ、file system speak ではコピーオンラインとも呼ばれ、それらを高効率で書き出すことが可能で、LFS はインメモリセグメントにすべての更新を集めて書き込むことができ、順次に一緒に出します。

TIP: TURN FLAWS INTO VIRTUES

システムに根本的な欠陥がある場合はいつでも、それらの機能を調べ有用なものに変えることができるかどうかを確認してください。NetApp の WAFL は古いファイルの内容でこれを行います。

古いバージョンを利用できるようにすることで、WAFL はかなり頻繁にするクリーンを心配する必要はなくなりました(最終的にはバックグラウンドで古いバージョンは削除されます)ので、クールな機能を提供し、すべて1つの素晴らしい考え方でLFSクリーニングの問題の多くを対処します。システムにこれの他の例がありますか？間違いなく、あなたは自分で考える必要があります。なぜなら、この章は頭文字“O”で終わっているからです。...OVER

LFSが生成する一般的な大きな書き込みは、さまざまなデバイスでのパフォーマンスに優れています。ハードドライブでは、書き込みが大きくなるため、位置決め時間が最小限に抑えられます。RAID-4やRAID-5などのパリティベースのRAIDでは、書き込みの問題が完全に回避されます。最近の研究では、フラッシュベースのSSD[H+17]の高性能化には大きなI/Oが必要であることが示されています。したがって、おそらく驚くべきことに、LFS形式のファイルシステムは、これらの新しい媒体であっても優れた選択肢となる可能性があります。

このアプローチの欠点は、ゴミを生成することです。データの古いコピーがディスク全体に散在し、後で使用するためにそのようなスペースを再利用したい場合は、古いセグメントを定期的にクリーニングする必要があります。クリーニングはLFSの多くの論争の焦点となり、クリーニングコスト(SS+95)に対する懸念はおそらくLFSの当初のフィールドへの影響を制限していました。しかし、NetAppのWAFL[HLM94]、SunのZFS[B07]、Linuxのbtrfs[R+13]、さらに現代のフラッシュベースのSSD[AD14]を含む最新の商用ファイルシステムでは、同様のコピー・オンラインアプローチでディスクに書き込んでいます。LFSの知的遺産は、これらの現代的なファイルシステム上で生き続けています。特に、WAFLはそれらを機能に変えることでクリーニング問題を回避しました。スナップショットを使用して古いバージョンのファイルシステムを提供することにより、ユーザーは誤って現在のファイルを削除するたびに古いファイルにアクセスすることができます。

参考文献

[AD14] “Operating Systems: Three Easy Pieces”

Chapter: Flash-based Solid State Drives

Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau

A bit gauche to refer you to another chapter in this very book, but who are we to judge?

[B07] “ZFS: The Last Word in File Systems”

Jeff Bonwick and Bill Moore

Copy Available: <http://www.ostep.org/Citations/zfs last.pdf>

Slides on ZFS; unfortunately, there is no great ZFS paper (yet). Maybe you will write one, so we can cite it here?

[H+17] “The Unwritten Contract of Solid State Drives”

Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

EuroSys '17, April 2017

In this paper, we study which unwritten rules one must follow in order to extract high performance from an SSD, both in the short term and over the long haul. Interestingly, both request scale (having large requests) and request locality still matter, even on solid-state storage. The more things change ...

[HLM94] “File System Design for an NFS File Server Appliance”

Dave Hitz, James Lau, Michael Malcolm

USENIX Spring '94

WAFL takes many ideas from LFS and RAID and puts it into a high-speed NFS appliance for the multi-billion dollar storage company NetApp.

[L77] “Physical Integrity in a Large Segmented Database”

R. Lorie

ACM Transactions on Databases, 1977, Volume 2:1, pages 91-104 The original idea of shadow paging is presented here.

[MJLF84] “A Fast File System for UNIX”

Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry

ACM TOCS, August, 1984, Volume 2, Number 3

The original FFS paper; see the chapter on FFS for more details.

[MR+97] “Improving the Performance of Log-structured File Systems with Adaptive Methods”

Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, Thomas E. Anderson
SOSP 1997, pages 238-251, October, Saint Malo, France

A more recent paper detailing better policies for cleaning in LFS.

[M94] “A Better Update Policy”

Jeffrey C. Mogul

USENIX ATC '94, June 1994

In this paper, Mogul finds that read workloads can be harmed by buffering writes for too long and then sending them to the disk in a big burst. Thus, he recommends sending writes more frequently and in smaller batches.

[P98] “Hardware Technology Trends and Database Opportunities”

David A. Patterson

ACM SIGMOD '98 Keynote Address, Presented June 3, 1998, Seattle, Washington

Available: <http://www.cs.berkeley.edu/~pattrsn/talks/keynote.html>

A great set of slides on technology trends in computer systems. Hopefully, Patterson will create another of these sometime soon.

[R+13] “BTRFS: The Linux B-Tree Filesystem”

Ohad Rodeh, Josef Bacik, Chris Mason

ACM Transactions on Storage, Volume 9 Issue 3, August 2013

Finally, a good paper on BTRFS, a modern take on copy-on-write file systems.

[RO91] “Design and Implementation of the Log-structured File System”

Mendel Rosenblum and John Ousterhout

SOSP '91, Pacific Grove, CA, October 1991

The original SOSP paper about LFS, which has been cited by hundreds of other papers and inspired many real systems.

[R92] “Design and Implementation of the Log-structured File System”

Mendel Rosenblum

<http://www.eecs.berkeley.edu/Pubs/TechRpts/1992/CSD-92-696.pdf>

The award-winning dissertation about LFS, with many of the details missing from the paper.

[SS+95] “File system logging versus clustering: a performance comparison”

Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, Venkata Padmanabhan

USENIX 1995 Technical Conference, New Orleans, Louisiana, 1995

A paper that showed the LFS performance sometimes has problems, particularly for workloads with many calls to `fsync()` (such as database workloads). The paper was controversial at the time.

[SO90] “Write-Only Disk Caches”

Jon A. Solworth, Cyril U. Orji

SIGMOD '90, Atlantic City, New Jersey, May 1990

An early study of write buffering and its benefits. However, buffering for too long can be harmful: see Mogul [M94] for details.

[Z+12] “De-indirection for Flash-based SSDs with Nameless Writes”

Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

FAST '13, San Jose, California, February 2013

Our paper on a new way to build flash-based storage devices. Because FTLs (flash-translation layers) are usually built in a log-structured style, some of the same issues arise in flash-based devices that do in LFS. In this case, it is the recursive update problem, which LFS solves neatly with an imap. A similar structure exists in most SSDs.

44 Flash-based SSDs

何十年ものハードディスクドライブの支配から、永続的なストレージデバイスの新しい形態が最近、世界で重要な役割を果たしました。一般にソリッドステートストレージと呼ばれるこのようなデバイスには、ハードドライブのような機械的または可動的な部品はありません。むしろ、メモリとプロセッサによく似たトランジスタから簡単に構築されています。しかし、典型的なランダムアクセスメモリ(例えば、DRAM)とは異なり、このようなソリッドステートストレージデバイス(SSD)は、電力損失にもかかわらず情報を保持するので、データの永続的記憶に使用するための理想的な候補です。

私たちが注目する技術は、1980年代に舛岡富士雄さんが作ったフラッシュ(より具体的には、NANDベースのフラッシュ)[M + 14]と呼ばれています。Flashには、いくつかのユニークな特性があります。例えば、与えられているチャンク(フラッシュページ)に書き込むためには、より大きなチャンク(すなわち、フラッシュブロック)を消去しなければならず、これはかなり高価になる可能性があります。さらに、あまりにも頻繁にページに書き込むと、ページが消耗します。これらの2つの特性は、フラッシュベースのSSDの構築を興味深い課題になっています。

CRUX: HOW TO BUILD A FLASH-BASED SSD

どのようにフラッシュベースのSSDを構築できますか？どのようにして高価な消去の性質を処理できますか？繰り返し上書きするとデバイスが消耗してしまうので、どのようにして長時間続くデバイスを構築できますか？技術進歩は止まるだろうか？または驚くことがなくなるでしょうか？

44.1 Storing a Single Bit

フラッシュ・チップは、1つのトランジスタに1つ以上のビットを格納するように設計されています。トランジスタ内にトラップされた電荷のレベルは2進値にマッピングされます。Single Level Cell(SLC)フラッシュでは、トランジスタ内に1ビットのみが記憶される(すなわち、1または0)。Multi Level Cell(MLC)フラッシュでは、2ビットが異なるレベルの電荷に符号化され、例えば、00,01,10,11は、低、若干低、若干高、高レベルで表される。セルあたり3ビットをエンコードするTriple Level Cell(TLC)フラッシュもあります。全体として、SLCチップはより高い性能を達成し、より高価です。

TIP: BE CAREFUL WITH TERMINOLOGY

フラッシュが文脈の中で私たちが何度も(ブロック、ページ)何度も使用してきた言葉ですが、以前とは若干異なる方法で使用されることにお気づきかもしれません。新しい用語は、あなたの人生をより困難にするために作成されたものではありませんが(しかし、そうしたことがあるかもしれません)、用語の決定が行われる中心的な権限がないために生じます。文脈に応じて、他の人によってはページというものがブロックであったり、その逆もあります。あなたの仕事はシンプルです。各文章内の適切な用語を知り、規律に精通した人々があなたが話していることを理解できるようにそれらを使用します。唯一の解決策はシンプルですが、時には痛みを伴うことです。あなたの記憶を使用してください。

もちろん、どのようなビットレベルのストレージがデバイス物理のレベルでどのように動作するかについては、多くの詳細があります。この本の範囲を超えて、あなた自身の[J10]でそれについてもっと読むことができます。

44.2 From Bits to Banks/Planes

彼らが古代ギリシャで言うように、単一のビット（またはいくつか）を格納することはストレージシステムを作成することではありません。したがって、フラッシュチップは、多数のセルからなる banks(バンク達) から作られます。または多数のセルからである planes(プレーン達) から構成されます。

バンクは 2 つの異なるサイズの単位、すなわち、一般に 128KB または 256KB のサイズのブロック（消去ブロックとも呼ばれる）、サイズが数 KB（例えば 4KB）のページごとにアクセスされます。各バンク内には多数のブロックがあります。各ブロック内に多数のページがあります。フラッシュを考えるときは、この新しい用語を覚えておく必要があります。これは、ディスクや RAID で参照するブロック、および仮想メモリで参照するページとは異なります。

図 44.1 に、ブロックとページを持つフラッシュ・プレーンの例を示します。この単純な例では、それぞれが 4 ページを含む 3 つのブロックがあります。ブロックとページを区別する理由は次のとおりです。この区別は、読み取りや書き込みなどのフラッシュ操作で重要なものです。さらにデバイスの全体的なパフォーマンスにとっても重要です。あなたが学ぶ最も重要な（そして奇妙な）ことは、ブロック内のページに書き込むためには、最初にブロック全体を消去しなければならないということです。このトリッキーな詳細は、フラッシュベースの SSD を面白くて価値のある課題にすることと、この章の後半の主題になります。

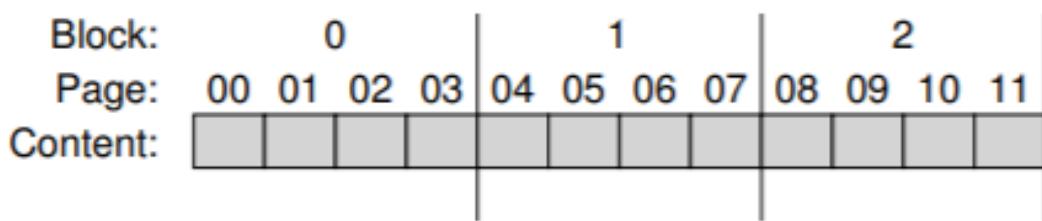


Figure 44.1: A Simple Flash Chip: Pages Within Blocks

44.3 Basic Flash Operations

このフラッシュ構成を考えると、フラッシュチップがサポートする 3 つの低レベル動作があります。読み取りコマンドは、フラッシュからページを読み取るために使用されます。消去とプログラムは並行して書き込みに使用されます。

- Read (1 ページ) :

フラッシュチップのクライアントは、読み出しコマンドと適切なページ番号をデバイスに指定するだけで、任意のページ（例えば、2KB または 4KB）を読み取ることができます。この操作は通常、デバイス上の場所に関係なく、以前の要求の場所にかかわらず（ディスクとはまったく異なります）、10 マイクロ秒程度以上と非常に高速です。どの場所にも一様に迅速にアクセスできることは、そのデバイスがランダムアクセスデバイスであることを意味します。

- Erase(ブロック) :

フラッシュ内のページに書き込む前に、デバイスの性質上、ページ内にあるブロック全体を最初に消去する必要があります。重要なことに、ブロックの内容を消去します（各ビットを値 1 に設定します）。そのため、ブロック内の必要なデータが、消去を実行する前に他の場所（メモリまたは別のフラッシュブロック）にコピーされていることを確認する必要があります。消去コマンドは非常に高価で、数ミリ秒かかります。終了すると、ブロック全体がリセットされ、各ページがプログラム可能な状態になります。

- Program(1 ページ) :

一度ブロックが消去されると、プログラムコマンドを使用して、ページ内の 1 を 0 に変更するなど、望んだ内容をフラッシュに書き込むことができます。ページのプログラミングは、ブロックを消去するよりも安価ですが、ページを読むよりもコストがかかります。通常、最新のフラッシュチップでは約 100 マイクロ秒です。

フラッシュチップについて考える一つの方法は、各ページに関連する状態があることです。ページは INVALID 状態で開始します。ページが存在するブロックを消去することによって、ページの状態 (およびそのブロック内のすべてのページ) を ERASED に設定します。これはブロック内の各ページの内容をリセットしますが、それらを (重要な) プログラマブルにします。ページをプログラミングすると、その状態が VALID に変わり、その内容が設定され、読み込めるなどを意味します。読み込みはこれらの状態に影響しません (ただし、プログラムされたページからのみ読み込めます)。ページがプログラムされると、その内容を変更する唯一の方法は、ページが存在するブロック全体を消去することです。次に、4 ページブロック内のさまざまな消去およびプログラム操作の後の状態遷移の例を示します。

	<i>ffff</i>	<i>Initial: pages in block are invalid (i)</i>
Erase()	→ EEEE	<i>State of pages in block set to erased (E)</i>
Program(0)	→ VEEE	<i>Program page 0; state set to valid (V)</i>
Program(0)	→ error	<i>Cannot re-program page after programming</i>
Program(1)	→ VVEE	<i>Program page 1</i>
Erase()	→ EEEE	<i>Contents erased; all pages programmable</i>

A Detailed Example

書き込みプロセス (すなわち、消去およびプログラミング) は非常に珍しいので、それが意味を成すことを確認するための詳細な例を見てみましょう。この例では、4 ページのブロック内に次の 4 つの 8 ビットページがあるとします (非現実的に小さいサイズですが、この例では便利です)。各ページは以前にプログラムされているので VALID です。

Page 0	Page 1	Page 2	Page 3
00011000	11001110	00000001	00111111
VALID	VALID	VALID	VALID

今度は、新しい内容をページ 0 に書きたいとします。ページを書き込むには、最初にブロック全体を消去する必要があります。ブロックをこの状態にしておくと仮定しましょう

Page 0	Page 1	Page 2	Page 3
11111111	11111111	11111111	11111111
ERASED	ERASED	ERASED	ERASED

例えば、コンテンツ 00000011 でページ 0 をプログラムし、古いページ 0(内容 00011000) を上書きします。そうした後、ブロックは次のようにになります。

Page 0	Page 1	Page 2	Page 3
00000011	11111111	11111111	11111111
VALID	ERASED	ERASED	ERASED

ページ 1,2,3 の以前の内容はすべて消えてしまいました！ したがって、ブロック内のページを上書きする前に、まず必要なデータを別の場所（メモリ、フラッシュ上のどこかなど）に移動する必要があります。消去の性質は、私たちがすぐに学ぶように、フラッシュベースの SSD をどのように設計するかで強く影響を与えます。

Summary

要約すると、ページを読むのは簡単です。単にページを読むだけです。フラッシュチップはこれを非常にうまくやっています。パフォーマンス面では、機械的なシークおよびローテーションコストのために遅い現在のディスクドライブのランダムな読み取りパフォーマンスを大幅に上回る可能性があります。

ページを書くのは手間がかかります。ブロック全体が最初に消去されなければなりません（最初に必要なデータを別の場所に移動してください）そして望んだページをプログラムします。これは高価なだけでなく、このプログラム/消去サイクルを頻繁に繰り返すと、フラッシュチップには大きな信頼性の問題が生じる可能性があります。フラッシュを使用してストレージシステムを設計する場合、書き込みのパフォーマンスと信頼性が中心的な焦点です。近代の SSD がこれらの問題をどのように攻撃し、これらの制限にもかかわらず優れたパフォーマンスと信頼性を提供する方法について、すぐにわかります。

44.4 Flash Performance And Reliability

私たちは、生のフラッシュチップからストレージデバイスを構築することに興味があるので、基本的な性能特性を理解することは価値があります。図 44.2 は、一般的なプレス [V12] に見られるいくつかの数字の概略を示しています。ここでは、セルごとに 1,2,3 ビットの情報を格納する SLC、MLC、TLC フラッシュ全体での読み取り、プログラム、消去の基本動作レイテンシを示します。

Device	Read (μs)	Program (μs)	Erase (μs)
SLC	25	200-300	1500-2000
MLC	50	600-900	~3000
TLC	~75	~900-1350	~4500

Figure 44.2: Raw Flash Performance Characteristics

表からわかるように、読み取りのレイテンシは非常に良いので、完了するのはわずか 10 マイクロ秒です。プログラムのレイテンシは、SLC の場合は 200 マイクロ秒と低く、より多くのビットを各セルにパックするほど高くなります。優れた書き込み性能を得るために複数のフラッシュ・チップを並行して使用する必要があります。最後に、消去はかなり高価で、典型的には数ミリ秒かかる。このコストを扱うことは、現代のフラッシュ記憶装置設計の中心です。

フラッシュチップの信頼性を考えてみましょう。（ドライブヘッドが記録面と実際に接触するような、厄介でかなり物理的なヘッドクラッシュを含む）様々な理由で故障する可能性がある機械的ディスクとは異なり、フラッシュチップは純粋なシリコンであり、その点で、少し信頼性の問題は心配です。主な関心事は摩耗です。フラッシュブロックが消去されてプログラムされると、少し余分な電荷がゆっくりと発生します。時間が経つにつれて、その余分な電荷が蓄積するにつれて、0 と 1 とを区別するのがますます困難になります。不可能になる時点では、ブロックは使用不能になります。

ブロックの典型的な寿命は、現在よく知られていません。製造元は、MLC ベースのブロックを 10,000 P/E（プログラム/消去）サイクルの寿命として評価します。つまり、各ブロックを消去してから 10,000 回プロ

グラムすると壊れます。SLC ベースのチップは、1 つのトランジスタにつき 1 ビットしか記憶しないため、より長い寿命、通常 100,000 P/E サイクルで定格されます。しかし、最近の研究では、生存期間が予想以上に長いことが示されています [BD10]。

フラッシュチップ内の 1 つの他の信頼性問題は、disturbance(妨害)として知られています。フラッシュ内の特定のページにアクセスすると、隣接するページのビットが反転する可能性があります。このようなビット反転は、ページがそれぞれ読み出されているかプログラマされているかに応じて、read disturbs または program disturbs として知られています。

TIP: THE IMPORTANCE OF BACKWARDS COMPATIBILITY

下位互換性は、階層化されたシステムで常に懸念されます。2 つのシステム間で安定したインターフェースを定義することにより、相互運用性を確保しながらインターフェースの両側で革新を実現できます。このアプローチは多くの領域で素晴らしい成功を収めてきました。オペレーティングシステムはアプリケーション用に比較的安定した API を持ち、ディスクはファイルシステムと同じブロックベースのインターフェースを提供し、IP ネットワーキングスタックの各レイヤーは上記のレイヤーに固定された変更のないインターフェースを提供します。驚くことではないが、ある世代で定義されたインターフェースが次の世代では適切でない可能性があるため、そのような剛性には欠点があります。場合によっては、システム全体を完全に再設計することについて考えるのが有益な場合もあります。優れた例は、Sun ZFS ファイルシステム [B07] にあります。ZFS の作成者は、ファイルシステムと RAID の相互作用を再考することで、より効果的な統合全体を構想しました。

44.5 From Raw Flash to Flash-Based SSDs

フラッシュ・チップの基本的な理解を踏まえて、次のタスク、すなわち、フラッシュ・チップの基本セットを典型的なストレージ・デバイスのようなものに変える方法があります。標準記憶インターフェースは、単純なブロックベースのものであり、ブロックアドレスが与えられた場合、512 バイト (またはそれ以上) のブロック (セクタ) が読み書き可能である。フラッシュベースの SSD のタスクは、標準のブロックインターフェースを内部の生のフラッシュチップが提供することです。内部的には、SSD はいくつかのフラッシュチップ (永続ストレージ用) で構成されています。SSD はある量の揮発性 (すなわち、非永続的) メモリ (例えば、SRAM) も含みます。このようなメモリは、データのキャッシング、バッファリングのためだけでなく、マッピングテーブルも役立ちます。最後に、SSD にはデバイスの操作を調整するための制御ロジックが含まれています。詳細は [A + 08] を参照してください。図 44.3 に簡略化したブロック図を示します。

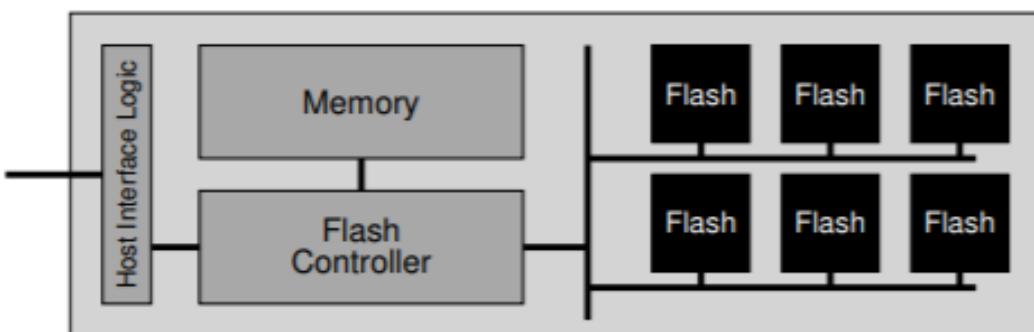


Figure 44.3: A Flash-based SSD: Logical Diagram

この制御ロジックの本質的な機能の 1 つは、クライアントの読み取りと書き込みを満足させ、必要に応じて内部のフラッシュ操作に変換することです。Flash Translation Layer(FTL) は、この機能を正確に提供しま

す。FTL は、論理ブロック (デバイス・インターフェースを構成する) 上で読み取りと書き込み要求を取り出し、物理ブロックおよび物理ページ (実際のフラッシュ・デバイスを構成する) 上の低レベル読み取り、消去、プログラム・コマンドに変換します。FTL は、優れた性能と高い信頼性を提供するという目的でこの作業を達成する必要があります。

優れたパフォーマンスは、技術の組み合わせによって実現できます。1つの鍵は、複数のフラッシュ・チップを並列に使用することです。この技術についてはこれ以上検討するつもりはありませんが、現代の SSD はすべて、内部で複数のチップを使用してより高い性能を得ています。もう1つのパフォーマンス目標は、書き込みの増幅を減らすことです。これは、FTL でフラッシュ・チップに発行された合計書き込みトラフィック (バイト単位) を合計書き込みトラフィック (バイト単位) で割ったものが、クライアントによって SSD に送信されます。以下で説明するように、FTL 構築に対する単純なアプローチは、高い書き込み増幅率と低い性能につながります。

いくつかの異なるアプローチを組み合わせることで高い信頼性が達成されます。上記で議論したように、1つの主な問題は摩耗です。1つのブロックがあまりにも頻繁に消去され、プログラムされると、使用できなくなります。結果として、FTL は、フラッシュのブロック間にできるだけ均等に書き込みを広げて、デバイスのすべてのブロックがほぼ同時に消耗するようにする必要があります。そうすることは wear leveling と呼ばれ、現代の FTL に不可欠な部分です。

もう一つの信頼性の懸念は、プログラム妨害である。このような妨害を最小限に抑えるために、FTL は、通常、消去されたブロック内のページを、低いページから高いページの順にプログラムします。この sequential programming(順序プログラミング) 手法は妨害を最小限に抑え、広く利用されています。

44.6 FTL Organization: A Bad Approach

FTL の最も単純な構成は、direct mapped(ダイレクトマップ) と呼ばれるものです。このアプローチでは、論理ページ N への読み出しありは、物理ページ N の読み出しだけでマッピングされます。論理ページ N への書き込みは、より複雑です。FTL は、最初に、ページ N が内包されているブロック全体を読み取らなければいけません。そのとき、そのブロックを消去する必要があります。最後に、FTL は古いページと新しいページをプログラムします。

おそらく推測できるように、ダイレクトマップ FTL には、パフォーマンスと信頼性の両方の点で多くの問題があります。各書き込みではパフォーマンス上の問題が発生します。デバイスはブロック全体の読み取り (コスト高)、消去 (非常に高価)、プログラム (コスト高) の順に処理する必要があります。最終的な結果は、(ブロック内のページ数に比例する) 厳しいライト増幅と、その結果として、機械的なシークと回転遅延を伴う典型的なハードドライブよりも遅い、ひどい書き込み性能です。

さらに悪いのは、このアプローチの信頼性です。ファイルシステムのメタデータまたはユーザーファイルのデータが繰り返し上書きされると、同じブロックが消去、プログラムされ、何度も繰り返し書き換えられ、すぐに消耗してデータが失われる可能性があります。ダイレクト・マップ・アプローチでは、クライアント・仕事量の消耗を制御することができません。仕事量が論理ブロック全体に均等に書き込み負荷を分散しない場合、一般的なデータを含む物理ブロックがすぐに消耗します。信頼性とパフォーマンスの両方の理由から、ダイレクトマップ FTL は悪い考えです。

44.7 A Log-Structured FTL

このような理由から、現在の FTL は、ログ構造であり、ストレージデバイス (上に示すように) とその上のファイルシステム (log structured file systems の章を参照) の両方で有用なアイデアです。論理ブロック N への書き込みの際に、デバイスは、書き込まれているブロック内の次の空き領域に書き込みを追加します。私たちはこれを書き込みロギングスタイル呼んでいます。ブロック N の後続の読み取りを可能にするために、デバ

イスはマッピングテーブルを保持しています(メモリ上に、そしてデバイス上の何らかの形で永続的に)。このテーブルには、システム内の各論理ブロックの物理アドレスが格納されます。

基本的なログベースのアプローチがどのように機能するかを理解するための例を見てみましょう。クライアントには、デバイスは一般的なディスクのように見え、512 バイトのセクタ（またはセクタのグループ）に読み書きできます。簡単にするために、クライアントが 4KB のチャンクを読み書きしているとします。SSD には、4 KB の 4 つのページにそれぞれ分割された 16 KB サイズのブロックがいくつか含まれているとします。これらのパラメータは非現実的です（フラッシュブロックは通常より多くのページで構成されます）が、私たちの教訓の目的を十分に果たします。クライアントが次の一連の操作を発行するとします。

- Write(100) with contents a1
 - Write(101) with contents a2
 - Write(2000) with contents b1
 - Write(2001) with contents b2

これらの論理ブロックアドレス(例えば100)は、SSDのクライアント(例えば、ファイルシステム)が情報がどこに位置しているかを記憶するために使用されます。内部的には、デバイスは、これらのブロック書き込みを、生のハードウェアによってサポートされている消去およびプログラム動作に変換し、各論理ブロックアドレスに対して、SSDのどの物理ページがそのデータを記憶するかを何らかの形で記録しなければなりません。SSDのすべてのブロックが現在有効ではないと仮定し、ページをプログラムする前に消去する必要があります。ここでは、すべてのページがINVALID(i)とマークされたSSDの初期状態を示します。

第1の書き込みが(論理ブロック100への)SSDによって受信されると、FTLは、それを4つの物理ページ: 0,1,2,3を含む物理ブロック0に書き込むことを決定します。ブロックは消去されていないので、まだそれに書き込むことはできません。デバイスは最初に消去コマンドをブロック0に発行する必要があります。削除すると、次の状態になります。

これでブロック 0 はプログラム可能な状態になります。ほとんどの SSD は、順番にページに書き込みます（すなわち、低から高に）、プログラム妨害に関連する信頼性の問題を低減します。そのとき SSD は、論理ブロック 100 の書き込みを物理ページ 0 に指示する。

しかし、クライアントが論理ブロック 100 を読み込みたい場合はどうなりますか？どのようにそれがどこにあるのか見つけることができますか？SSD は、論理ブロック 100 に発行された読み取りを物理ページ 0 の読み取りに変換しなければなりません。このような機能に対応するために、FTL は論理ブロック 100 を物理ページ 0 に書込むとき、この事実をメモリ内マッピングテーブルに記録します。このマッピングテーブルの状態を図でも追跡します

Table: 100 → 0 Memory

Block:	0	1	2	Flash Chip
Page:	00 01 02 03	04 05 06 07	08 09 10 11	
Content:	a1			
State:	V E E E	i i i i	i i i i	

これで、クライアントが SSD に書き込むときに何が起こるかを確認できます。SSD は書き込みの場所を見つけます。通常、次の空きページを選択するだけです。そのときブロックの中身であるそのページをプログラムし、マッピングテーブルに論理から物理へのマッピングを記録します。後続の読み取りは、テーブルを使用して、クライアントが提示した論理ブロックアドレスを、データを読み取るために必要な物理ページ番号に変換するだけです。例題である書き込みストリームの残りの書き込み(101、2000、2001)を調べてみましょう。これらのブロックを書き込んだ後、デバイスの状態は次のようにになります。

Table: 100 → 0 101 → 1 2000 → 2 2001 → 3 Memory

Block:	0	1	2	Flash Chip
Page:	00 01 02 03	04 05 06 07	08 09 10 11	
Content:	a1 a2 b1 b2			
State:	V V V V	i i i i	i i i i	

ログベースのアプローチでは、パフォーマンスが向上し(しばらくの間一回しか必要な消去が行われず、ダイレクト・マップ・アプローチによる高価な read modify write が回避されます)、信頼性が大幅に向上します。FTL は、すべてのページにわたって書き込みを分散し、wear leveling と呼ばれる処理を実行し、デバイスの寿命を延ばすことができます。さらに下の wear leveling について説明します。

ASIDE: FTL MAPPING INFORMATION PERSISTENCE

あなたは疑問に思うかもしれません。デバイスが電力を失う場合はどうなりますか？メモリ内のマッピングテーブルは消えますか？明らかに、このような情報は本当に失われてはいけません。なぜなら、デバイスは永続的なストレージデバイスとして機能しないためです。SSD には、マッピング情報を回復する手段が必要です。

最も簡単なことは、Out Of Band(OOB) 領域と呼ばれる、各ページでマッピング情報を記録することです。デバイスの電源が切れて再起動されると、デバイスは OOB 領域をスキャンしてメモリ内のマッピングテーブルを再構築します。この基本的なアプローチには問題があります。必要なすべてのマッピング情報を見つけるために大きな SSD をスキャンするのは遅いです。この制限を克服するために、一部のハイエンド端末では、より複雑なロギングとチェックポイント手法を使用して復旧を高速化しています。ロギングについては後でファイルシステムで詳しく説明します。

残念なことに、このログ構造化の基本的なアプローチにはいくつかの欠点があります。1つ目は、論理ブロックの上書きが、ガベージと呼ばれるもの、つまりドライブの周りの古いバージョンのデータとスペースを

占有することにつながるということです。デバイスは、前記ブロックおよび将来の書き込みのための空き領域を見つけるためにガーベッジコレクション (GC) を定期的に実行しなければいけません。しかし、過度のガーベッジコレクションは書き込み増幅を引き上げ、パフォーマンスを低下させます。2つめは、メモリ内マッピングテーブルのコストが高いことです。デバイスが大きくなればなるほど、そのようなテーブルに必要なメモリが増えます。今度は順番にこれらについて話し合っていきます。

44.8 Garbage Collection

このようなログ構造アプローチの第 1 のコストは、ゴミが作成されることであり、ガーベッジコレクション(すなわち、dead-block reclamation)が行われなければいけません。私たちの継続的な例を使ってこれを理解しましょう。論理ブロック 100,101,2000,2001 がデバイスに書き込まれたことを思い出してください。さて、ブロック 100 とブロック 101 が内容 c1 と c2 で再び書き込まれたとしましょう。書き込みは次の空きページ(この場合は物理ページ 4 と 5) に書き込まれ、それに応じてマッピングテーブルが更新されます。このようなプログラミングを可能にするには、最初にブロック 1 を消去する必要があります。

Table:	100 → 4	101 → 5	2000 → 2	2001 → 3	Memory
Block:	0	1	2		
Page:	00 01 02 03	04 05 06 07	08 09 10 11		Flash Chip
Content:	a1 a2 b1 b2	c1 c2			
State:	V V V V	V V E E	i i i i		

物理ページ 0 と 1 は VALID とマークされていますが、ブロック 100 と 101 の古いバージョンのようなゴミがあります。デバイスのログ構造上の性質のため、上書きするとガーベッジブロックが作成されます。そのため、新しい書き込みを実行するための空き領域を確保するためにデバイスを再利用する必要があります。

ガーベッジ・ブロック (デッド・ブロックとも呼ばれます) を見つけ出し、将来使用するためにそれらを再利用するプロセスは、ガーベッジ・コレクションと呼ばれ、現代の SSD の重要なコンポーネントです。基本的なプロセスは、1つまたは複数のゴミ・ページを含むブロックを見つけ、そのブロックからライブ(非ゴミ)ページを読み込み、それらのライブ・ページをログに書き出し、そして(最後に)書き込みに使用するためにブロック全体を再利用します。

例をあげて説明しましょう。デバイスは、上記のブロック 0 内の死んだページを再利用したいと決定します。ブロック 0 には、2つのデッドブロック (ページ 0 と 1) と 2つのライフブロック (ページ 2 と 3、それぞれブロック 2000 と 2001 が含まれています) があります。そうするために、デバイスは以下を行います。

- ブロック 0 からライブデータ (2 および 3 ページ) を読み込む
- ライブデータをログの最後に書き込む
- ブロック 0 を消去する (後で使用するために解放する)

ガーベッジコレクタを機能させるには、SSD が各ページがライブかデッドかを判断できるように、各ブロック内に十分な情報がなければなりません。この目的を達成する 1 つの自然な方法は、各ブロック内のある場所に、各ページ内にどの論理ブロックが格納されているかに関する情報を格納することです。デバイスは、マッピングテーブルを使用して、ブロック内の各ページがライブデータを保持するかどうかを判断できます。

(ガーベッジコレクションが行われる前の) この例から、ブロック 0 は論理ブロック 100,101,2000,2001 を保持していました。マッピングテーブルをチェックして(ガーベッジコレクション前のマッピングテーブル 100 -> 4,101 -> 5, 2000 -> 2, 2001 -> 3 を含んでいる)、デバイスは、SSD ブロック内の各ページがライブ情報を保持しているかどうかを容易に判定することができます。例えば、2000 と 2001 はまだマップに参照されています。100 と 101 はそうではないため、ガーベッジコレクションの候補となります。このガーベッジコレクション

プロセスがこの例で完了すると、デバイスの状態は次のようにになります。

Table: 100 → 4 101 → 5 2000 → 6 2001 → 7 Memory

Block:	0	1	2	Flash Chip
Page:	00 01 02 03	04 05 06 07	08 09 10 11	
Content:	c1 c2 b1 b2			
State:	E E E E	V V V V	i i i i	

ご覧のように、ガベージコレクションは高価な場合があり、ライブデータの読み込みと書き換えが必要です。再利用の理想的な候補は、死んだページだけで構成されるブロックです。この場合、高価なデータ移行なしでブロックを即座に消去して新しいデータに使用することができます。GCのコストを削減するために、一部のSSDはデバイスを overprovisionをします[A + 08]。余分なフラッシュ容量を追加することで、クリーニングを遅らせてバックグラウンドにプッシュすることができ、デバイスが忙しくないときに実行されます。容量を増やすと、内部帯域幅も増加します。これは、クリーニングに使用できるため、クライアントに認識される帯域幅に害を与えません。現代の多くのドライブは、このように過度に overprovisionし、優れた全体的なパフォーマンスを達成するための1つの鍵です。

44.9 Mapping Table Size

ログ構造化の第2のコストは、非常に大きなマッピングテーブルの可能性であり、デバイスの4 KBページごとに1つのエントリがあります。たとえば、1 TBの大きなSSDを使用すると、4 KBのページごとに1つの4バイトエントリが発生するため、これらのマッピングのためだけに1 GBのメモリが必要になります。従って、このページレベルFTL方式は実用的ではありません。

Block-Based Mapping

マッピングのコストを削減する1つのアプローチは、ページごとではなく、デバイスのブロックごとにポインタを保持することだけであり、マッピング情報の量を Sizeblock/Sizepageの係数だけ減少させます。このブロック・レベルのFTLは、仮想メモリー・システムでより大きなページ・サイズを持つことに似ています。その場合は、VPNのビット数を少し使って、仮想アドレスごとのオフセットを大きくします。

残念ながら、ログベースのFTL内でブロックベースのマッピングを使用することは、パフォーマンスの理由からうまく機能しません。最大の問題は、「小さな書き込み」で発生します(すなわち、物理ブロックのサイズより小さいもの)。この場合、FTLは古いブロックから大量のライブデータを読み込み、それを新しい書き込みにコピーする必要があります(小さな書き込みのデータと一緒に)。このデータコピーは、書き込み増幅を大幅に増加させ、したがって性能を低下させます。

この問題をより明確にするために、例を見てみましょう。クライアントが以前に論理ブロック2000,2001,2002,2003(内容、a, b, c, d)を書き出し、それらが物理ページ4,5,6,7の物理ブロック1内にあると仮定します。ページごとのマッピングでは、変換テーブルは、2000 → 4, 2001 → 5, 2002 → 6, 2003 → 7の4つの論理ブロックのマッピングを記録する必要があります。

代わりに、ブロックレベルのマッピングを使用する場合、FTLはすべてのデータに対して单一のアドレス変換を記録するだけで済みます。ただし、アドレスマッピングは前の例と少し異なります。具体的には、フラッシュ内の物理ブロックのサイズであるチャンクに細断されるデバイスの論理アドレス空間を考えます。したがって、論理ブロックアドレスは、チャンク番号とオフセットの2つの部分からなります。4つの論理ブロックが各物理ブロック内に収まると仮定しているので、論理アドレスのオフセット部分は2ビット必要になります。残りの(最上位)ビットがチャンク番号を形成します。

Table: 500 → 4

Block:	0				1				2				Flash Chip
Page:	00	01	02	03	04	05	06	07	08	09	10	11	
Content:					a	b	c	d					
State:	i	i	i	i	V	V	V	V	i	i	i	i	

ブロックベースの FTL では、読み込みが容易です。最初に、FTL は、アドレスから最上位のビットを取り出すことによって、クライアントによって提示された論理ブロックアドレスからチャンク番号を抽出する。そのとき、FTL はテーブル内のチャンク番号から物理ページへのマッピングを検索します。最後に、FTL は、論理アドレスからのオフセットをブロックの物理アドレスに追加することによって、望んだフラッシュページのアドレスを計算します。

例えば、クライアントが論理アドレス 2002 への読み出しを発行する場合、デバイスは論理チャンク番号 (500) を抽出し、マッピングテーブル内の変換を検索し (4 を見つける)、論理アドレス (2) からのオフセットを加算して変換します (4)。結果の物理ページアドレス (6) は、データが配置されている場所です。FTL はその物理アドレスに読み出しを発行し、望んだデータ (c) を得ることができます。

しかし、クライアントが論理ブロック 2002(内容 c') に書き込むとどうなるでしょうか？この場合、FTL は 2000,2001,2003 を読み込み、4 つの論理ブロックすべてを新しい場所に書き出し、それに応じてマッピング・テーブルを更新する必要があります。ここに示すように、ブロック 1(常駐していたデータ) は消去して再利用できます。

Table: 500 → 8

Block:	0				1				2				
Page:	00	01	02	03	04	05	06	07	08	09	10	11	
Content:									a	b	c'	d	Flash Chip
State:	i	i	i	i	E	E	E	E	V	V	V	V	

この例からわかるように、ブロックレベルのマッピングは変換に必要なメモリ量を大幅に削減しますが、書き込みがデバイスの物理ブロックサイズよりも小さくなるとパフォーマンスに大きな問題が発生します。実際の物理ブロックは 256KB 以上になる可能性があるため、このような書き込みは非常に頻繁に発生する可能性があります。従って、よりよい解決策が必要です。その解決策が何であるかを私たちが教えてきた中で思いつくことはできますか？ あなたはこの後を読む前に、あなた自身でそれを理解できますか？

Hybrid Mapping

柔軟な書き込みを可能にするとともにマッピングコストを削減するために、最新の FTL の多くはハイブリッドマッピング技術を採用しています。このアプローチでは、FTL はいくつかのブロックを消去したままにして、すべての書き込みをそれらに指示します。これらはログブロックと呼ばれます。FTL は、純粋なブロック・ベースのマッピングで必要とされるすべてのコピーを行わずに、ログ・ブロック内の任意の場所の任意のページに書き込めるように、これらのログ・ブロックのページ単位のマッピングを保持します。

したがって、FTLには論理的に、メモリに2種類のマッピングテーブルがあります。すなわち、ログテーブルと呼ばれるページ単位のマッピングの小さなセットと、データテーブルのブロック単位のマッピングの大きなセットです。特定の論理ブロックを探すとき、FTLはまずログテーブルを調べます。論理ブロックの位置がそこに見つからない場合、FTLはデータテーブルを参照してその位置を見つけ、要求されたデータにアクセスします。

します。

ハイブリッドマッピング戦略の鍵は、ログブロックの数を小さく保つことです。ログブロックの数を少なく保つために、FTL はログブロック（ページごとのポインタを持つ）を定期的に調べて、单一のブロックポインタだけが指示することができるブロックに切り替える必要があります。このスイッチは、ブロック [KK + 02] の内容に基づいて 3 つの主な技術のうちの 1 つによって実行されます。

たとえば、FTL が以前に論理ページ 1000,1001,1002,1003 を書き出し、物理ブロック 2(物理ページ 8,9,10,11) に配置したとします。1000,1001,1002,1003 への書き込みの内容をそれぞれ a, b, c, d とします。

Log Table:

Data Table: 250 → 8

Memory

Block:	0				1				2				
Page:	00	01	02	03	04	05	06	07	08	09	10	11	Flash Chip
Content:	a	b	c	d					a	b	c	d	Flash Chip
State:	i	i	i	i	i	i	i	i	v	v	v	v	

ここで、クライアントが、現在利用可能なログブロックの 1 つ、例えば物理ブロック 0(物理ページ 0,1,2,3) に、これらのページ（データ a', b', c', d'）を全く同じ順序で上書きすると仮定します。この場合、FTL の状態は次のようにになります。

Log Table: 1000 → 0 1001 → 1 1002 → 2 1003 → 3

Data Table: 250 → 8

Memory

Block:	0				1				2				
Page:	00	01	02	03	04	05	06	07	08	09	10	11	Flash Chip
Content:	a'	b'	c'	d'					a	b	c	d	Flash Chip
State:	v	v	v	v	i	i	i	i	v	v	v	v	

これらのブロックは以前と全く同じ方法で記述されているため、FTL は switch merge と呼ばれる処理を実行できます。この場合、ログブロック (0) はページ 0,1,2,3 の格納場所になり、1 つのブロックポインタで参照されます。古いブロック (2) は消去され、ログブロックとして使用されます。この最良の場合、ページ単位のポインタはすべて单一のブロックポインタに置き換えられます。

Log Table:

Data Table: 250 → 0

Memory

Block:	0				1				2				
Page:	00	01	02	03	04	05	06	07	08	09	10	11	Flash Chip
Content:	a'	b'	c'	d'									Flash Chip
State:	v	v	v	v	i	i	i	i	i	i	i	i	

この switch merge は、ハイブリッド FTL の最良のケースです。残念ながら、FTL はあまり運がない場合もあります。同じ初期条件（物理ブロック 2 に格納されている論理ブロック 0,1,2,4）があり、クライアントが論理ブロック 0 と 1 だけを上書きする場合を考えてみましょう。

Log Table:	$1000 \rightarrow 0$	$1001 \rightarrow 1$		
Data Table:	$250 \rightarrow 8$			Memory
Block:	0	1	2	
Page:	00 01 02 03	04 05 06 07	08 09 10 11	Flash Chip
Content:	a' b'		a b c d	
State:	V V i i	i i i i	V V V V	

この物理ブロックの他のページを再統一するために、単一のブロックポインタだけでそれらを参照できるようにするために、FTL は partial merge(部分マージ) と呼ばれる処理を実行します。この操作では、2 と 3 がブロック 4 から読み込まれ、ログに追加されます。結果として得られる SSD の状態は、上記の switch merge と同じです。ただし、この場合、FTL はその目的を達成するために余分な I/O を実行する必要がありました(物理ページ 18 と 19 から論理ブロック 2 と 3 を読み取り、物理ページ 22 と 23 に書き出す)。したがって、書き込み増幅が増加します。

フル・マージ (full merge) と呼ばれる FTL が直面する最後のケースであり、さらに多くの作業が必要です。この場合、FTL は、クリーニングを実行するために他の多くのブロックからページを集める必要があります。たとえば、ページ 0, 4, 8, 12 がログブロック A に書き込まれたとします。このログブロックをブロックマップページに切り替えるには、まず FTL が論理ブロック 0,1,2,3 を含むデータブロックを作成する必要があります。したがって、FTL は 1,2,3 を別の場所から読み取り、0,1,2,3 を一緒に書き出す必要があります。次に、マージは、論理ブロック 4 について同じことをして、5,6,7 を見つけてそれを单一のデータブロックに調整する必要があります。論理ブロック 8 および 12 に対して同じことが行われなければならず、次に(最終的に)、ログブロック A は解放されます。驚くことではないが、頻繁な完全なマージはパフォーマンスに重大な損害を与える可能性があります [GY + 09]。

44.10 Wear Leveling

最後に、現代の FTL が実装しなければならない関連するバックグラウンド活動は、上記のようにウェアレベリングです。基本的な考え方は簡単です。複数の消去/プログラム・サイクルがフラッシュ・ブロックを消耗するため、FTL はデバイスのすべてのブロックにその作業を均等に広げるために最善を尽くすべきです。このようにして、すべてのブロックは、「人気の」ブロックがすぐに使用できなくなる代わりに、ほぼ同じ時間に消耗します。

基本的なログ構造化アプローチは、書き込み負荷を分散させるための最初の良い仕事を行い、ガベージコレクションも役立ちます。ただし、ブロックが上書きされない長寿命のデータでいっぱいになることがあります。この場合、ガベージコレクションはブロックを再利用することはないため、書き込み負荷の公平性を受け取ることはありません。

この問題を解決するには、FTL は定期的にそのブロックからライブデータをすべて読み込み、別の場所に書き直す必要があります。したがって、ブロックを再度書き込み可能にする必要があります。このウェアレベリングのプロセスは、SSD の書き込み増幅を増加させ、したがって、すべてのブロックがほぼ同じレートで確実に摩耗するように、余分な I/O が必要になるため、パフォーマンスが低下します。多くの異なるアルゴリズムが文献 [A + 08, M + 14] が存在します。これらに興味を持ったのであれば読んでみてください。

44.11 SSD Performance And Cost

クロージングする前に、最新の SSD のパフォーマンスとコストを調べて、永続的なストレージシステムでどのように使用されるのかを理解してみましょう。どちらの場合も、従来のハードディスクドライブ (HDD) と比較し、両者の最大の違いを強調します。

Performance

ハード・ディスク・ドライブとは異なり、フラッシュ・ベースの SSD は機械的なコンポーネントを備えておらず、実際には「ランダム・アクセス」デバイスであるという点で DRAM に多くの点で似ています。ディスクドライブと比較してパフォーマンスの最大の違いは、ランダムな読み書きを実行するときに実現されます。一般的なディスクドライブでは 1 秒間に数百回のランダム I/O しか実行できませんが、SSD の方がはるかに優れています。ここでは、近代的な SSD のデータを使用して、SSD の性能がどれだけ優れているかを確認します。我々は、FTL が生のチップの性能問題をどれくらいうまく隠すかに特に関心があります。

表 44.4 に、3 つの異なる SSD と最先端のハードドライブのパフォーマンスデータを示します。データはいくつかの異なるオンラインソースから取得しました [S13, T15]。左の 2 つの列はランダムな I/O パフォーマンスを示し、右の 2 つの列は順次な I/O です。最初の 3 行は Samsung、Seagate、Intel の 3 種類の SSD のデータを表示し、最後の行はハードディスクドライブ (または HDD) のパフォーマンスを示します。この場合は Seagate のハイエンドドライブです。

Device	Random		Sequential	
	Reads (MB/s)	Writes (MB/s)	Reads (MB/s)	Writes (MB/s)
Samsung 840 Pro SSD	103	287	421	384
Seagate 600 SSD	84	252	424	374
Intel SSD 335 SSD	39	222	344	354
Seagate Savvio 15K.3 HDD	2	2	223	223

Figure 44.4: SSDs And Hard Drives: Performance Comparison

テーブルから興味深い事実をいくつか学ぶことができます。まず、最も劇的なのは、SSD と唯一のハードドライブ間のランダム I/O パフォーマンスの違いです。SSD はランダム I/O で数十 MB/s から数百 MB/s を達成していますが、この「高性能」ハードドライブでは数 MB/s のピークがあります (実際には 2 MB/s になるように丸めました)。第 2 に、順次 I/O パフォーマンスの点で、違いがはるかに少ないことがわかります。順次 I/O パフォーマンスが必要なだけであれば、SSD のパフォーマンスでも向上しますが、ハードドライブはまだまだ良い選択です。第 3 に、SSD のランダム読み取りパフォーマンスが SSD ランダム書き込みパフォーマンスほど良くないことがわかります。

このような予期しなかった良好なランダム書き込みパフォーマンスの理由は、ランダム書き込みを順次書き込みに変換してパフォーマンスを向上させる、多くの SSD のログ構造設計によるものです。最後に、SSD は順次 I/O とランダム I/O のパフォーマンスに差があるため、ハードドライブのファイルシステムを構築する方法については、後続の章で学習する多くのテクニックが SSD に適用されます。順次 I/O とランダム I/O の差の大きさは小さくなりますが、ランダム I/O を減らすためにファイルシステムを設計する方法を慎重に考慮するには十分なギャップがあります。

Cost

上で見たように、SSD のパフォーマンスは、順次 I/O を実行している場合でも、現代のハードドライブを大幅に上回ります。それでは、なぜ SSD は記憶媒体としてハードドライブを完全に置き換わっていないのですか？ その答えは簡単です。コスト、具体的には、容量単位あたりのコストです。現在、[A15] の SSD は、250GB のドライブでは 150 ドルの費用がかかります。そのような SSD のコストは 60 セント/GB です。一般的なハードドライブは 1TB のストレージで約 50 ドルかかります。つまり、1GB あたり 5 セントです。これら 2 つの記憶媒体の間にはまだ 10 倍以上のコスト差があります。

これらのパフォーマンスとコストの違いにより、大規模なストレージシステムの構築方法が決まります。パフォーマンスが主な関心事である場合、特にランダムな読み取りパフォーマンスが重要な場合は、SSD を使用するのが最適です。一方、大規模なデータセンターを組み立てて膨大な量の情報を保管したい場合、大きなコスト差がハードドライブに向かうでしょう。もちろん、ハイブリッドアプローチである、SSD とハードドライブの両方で組み立てられているストレージシステムがあります。より使用頻度が高い「ホット」データ用の少数の SSD で高性能を提供しつつ、コストを節約するためにハードドライブ上の(使用頻度の低い)データを保管するハイブリッドアプローチがあります。価格差が存在する限り、ハードドライブは使われ続けます。

44.12 Summary

フラッシュベースの SSD は、世界の経済を動かすデータセンター内のラップトップ、デスクトップ、およびサーバーで共通の存在になっています。したがって、あなたはおそらくそれらについて何かを知るべきでしょうか？

ここで悪いニュースがあります。この章(この本の多くのもののように)は、最新の状態を理解するための最初のステップにすぎません。実際のデバイス性能に関する研究(Chen et al らの [CK + 09] や Grupp et al らの [GC + 09] など)、FTL デザインの問題(作品を含む Gupta らの [GY + 09]、Huang らの [H + 14]、Kim らの [KK + 02]、Lee らの [L + 07]、Agrawal らの [A + 08]、Zhang らの [Z + 12])、フラッシュで構成される分散システム(Gordon らの [CG + 09] および CORFU らの [B + 12] を含む)があります。

学術論文を読むだけではありません。一般的なプレス(例えば、[V12])の最近の進歩についても読無必要があります。そこでは、実用的な(まだ、便利ではない)情報を学ぶことができます。サムスンが同じ SSD 内で TLC と SLC セルの両方を使用してパフォーマンスを最大化する(SLC が書き込みをすればやくバッファに入れることができる)だけでなく、最大容量(TLC はセルあたりより多くのビットを格納できる)を行っています。そして、彼らが言うように、これは氷山の先端です。おそらく Ma らの優秀な(そして最近の)調査[M + 14]から始まって、あなた自身がこの研究の“氷山”についてもっと詳しく知ってください。しかし、注意してください。氷山は船の中でも最も強力なものを沈めることができます[W15]。

参考文献

- [A+08] “Design Tradeoffs for SSD Performance”
 - N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy
 - USENIX '08, San Diego California, June 2008
 - An excellent overview of what goes into SSD design.
- [A15] “Amazon Pricing Study”
 - Remzi Arpaci-Dusseau
 - February, 2015
 - This is not an actual paper, but rather one of the authors going to Amazon and looking at current prices of hard drives and SSDs. You too can repeat this study, and see what the costs are today. Do it!

[B+12] “CORFU: A Shared Log Design for Flash Clusters”

M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, J. D. Davis

NSDI ’12, San Jose, California, April 2012

A new way to think about designing a high-performance replicated log for clusters using Flash.

[BD10] “Write Endurance in Flash Drives: Measurements and Analysis”

Simona Boboila, Peter Desnoyers

FAST ’10, San Jose, California, February 2010

A cool paper that reverse engineers flash-device lifetimes. Endurance sometimes far exceeds manufacturer predictions, by up to 100 ×.

[B07] “ZFS: The Last Word in File Systems”

Jeff Bonwick and Bill Moore

Available: http://www.ostep.org/Citations/zfs_last.pdf

Was this the last word in file systems? No, but maybe it’s close.

[CG+09] “Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications”

Adrian M. Caulfield, Laura M. Grupp, Steven Swanson

ASPLOS ’09, Washington, D.C., March 2009

Early research on assembling flash into larger-scale clusters; definitely worth a read.

[CK+09] “Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives”

Feng Chen, David A. Koufaty, and Xiaodong Zhang

SIGMETRICS/Performance ’09, Seattle, Washington, June 2009

An excellent overview of SSD performance problems circa 2009 (though now a little dated).

[G14] “The SSD Endurance Experiment”

Geoff Gasior

The Tech Report, September 19, 2014

Available: <http://techreport.com/review/27062>

A nice set of simple experiments measuring performance of SSDs over time. There are many other similar studies; use google to find more.

[GC+09] “Characterizing Flash Memory: Anomalies, Observations, and Applications”

L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, J. K. Wolf

IEEE MICRO ’09, New York, New York, December 2009

Another excellent characterization of flash performance.

[GY+09] “DFTL: a Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings”

Aayush Gupta, Youngjae Kim, Bhuvan Urgaonkar

ASPLOS ’09, Washington, D.C., March 2009

This paper gives an excellent overview of different strategies for cleaning within hybrid SSDs as well as a new scheme which saves mapping table space and improves performance under many workloads.

[H+14] “An Aggressive Worn-out Flash Block Management Scheme

To Alleviate SSD Performance Degradation”

Ping Huang, Guanying Wu, Xubin He, Weijun Xiao

EuroSys ’14, 2014

Recent work showing how to really get the most out of worn-out flash blocks; neat!

- [J10] “Failure Mechanisms and Models for Semiconductor Devices”
 Report JEP122F, November 2010
 Available: <http://www.jedec.org/sites/default/files/docs/JEP122F.pdf>
 A highly detailed discussion of what is going on at the device level and how such devices fail. Only for those not faint of heart. Or physicists. Or both.
- [KK+02] “A Space-Efficient Flash Translation Layer For Compact Flash Systems”
 Jesung Kim, Jong Min Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho
 IEEE Transactions on Consumer Electronics, Volume 48, Number 2, May 2002
 One of the earliest proposals to suggest hybrid mappings.
- [L+07] “A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation”
 Sang-won Lee, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, Ha-Joo Song
 ACM Transactions on Embedded Computing Systems, Volume 6, Number 3, July 2007
 A terrific paper about how to build hybrid log/block mappings.
- [M+14] “A Survey of Address Translation Technologies for Flash Memories”
 Dongzhe Ma, Jianhua Feng, Guoliang Li
 ACM Computing Surveys, Volume 46, Number 3, January 2014
 Probably the best recent survey of flash and related technologies.
- [S13] “The Seagate 600 and 600 Pro SSD Review”
 Anand Lal Shimpi
 AnandTech, May 7, 2013
 Available: <http://www.anandtech.com/show/6935/seagate-600-ssd-review>
 One of many SSD performance measurements available on the internet. Haven’t heard of the internet? No problem. Just go to your web browser and type “internet” into the search tool. You’ll be amazed at what you can learn.
- [T15] “Performance Charts Hard Drives”
 Tom’s Hardware, January 2015
 Available: <http://www.tomshardware.com/charts/enterprise-hdd-charts/>
 Yet another site with performance data, this time focusing on hard drives.
- [V12] “Understanding TLC Flash”
 Kristian Vatto
 AnandTech, September, 2012
 Available: <http://www.anandtech.com/show/5067/understanding-tlc-nand>
 A short description about TLC flash and its characteristics.
- [W15] “List of Ships Sunk by Icebergs”
 Available: http://en.wikipedia.org/wiki/List_of_ships_sunk_by_icebergs
 Yes, there is a wikipedia page about ships sunk by icebergs. It is a really boring page and basically everyone knows the only ship the iceberg-sinking-mafia cares about is the Titanic.
- [Z+12] “De-indirection for Flash-based SSDs with Nameless Writes”
 Yiyi Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
 FAST ’13, San Jose, California, February 2013
 Our research on a new idea to reduce mapping table space; the key is to re-use the pointers in the file system above to store locations of blocks, instead of adding another level of indirection.

45 Data Integrity and Protection

これまでに研究してきたファイルシステムの基本的な進歩を超えて、多くの機能を検討する価値があります。この章では、信頼性にまた重点を置いて説明します (RAID の章で以前にストレージシステムの信頼性を検討したことがあります)。具体的には、ファイルシステムやストレージシステムは、最新のストレージデバイスの信頼性の低い性質を考慮して、データが安全であることをどのように保証する必要がありますか？この一般的な領域は、data integrity(データ完全性) または data protection(データ保護) と呼ばれます。そこで、ストレージシステムがデータをあなたに返すとき、ストレージシステムに入力したデータと同じであることを保証するために使用される手法を検討します。

CRUX: HOW TO ENSURE DATA INTEGRITY

ストレージに書き込まれたデータがシステムによってどのように確実に保護されるべきですか？どんな技術が必要ですか？そのようなテクニックを効率的にするには、省スペースと時間の両方のオーバーヘッドはどうですか？

45.1 Disk Failure Modes

RAIDについての章で学んだように、ディスクは完璧ではなく、失敗することもあります。初期の RAID システムでは、障害のモデルは非常に単純でした。ディスク全体が動作しているかまたは完全に故障しているか、これであれば、そのような障害の検出は簡単です。このようなディスク障害のフェールストップモデルは、RAID の構築を比較的簡単にします [S90]。

あなたが学んでいないのは、現代のディスクが示す他のタイプの故障モードのすべてです。具体的には、Bairavasundaram et al です。現代のディスクは時には大部分は動作しているように見えますが、1つまたは複数のブロックに正常にアクセスするのに問題があります [B + 07, B + 08]。具体的には、Latent Sector Errors(LSEs) と block corruption の2つのタイプのシングルブロック障害が共通して考慮する価値があります。ここで、それぞれをより詳しく説明します。

LSEs は、ディスクセクタ（またはセクタグループ）が何らかの形で損傷を受けたときに発生します。たとえば、何らかの理由でディスクヘッドが表面に接触した場合（ヘッドクラッシュ、通常の操作では起こらないもの）、表面が損傷してビットが判読不能になる可能性があります。宇宙線もビットを反転させることができ、不正確な内容につながります。幸いにも、ディスク内の Error Correcting Codes(ECC) は、ブロック内のディスク上のビットが正常であるかどうかを判断し、場合によっては、ブロックを修正するためにドライブで使用されます。ドライブがエラーを解決するのに十分な情報を持っていない場合、ディスクは、要求を発行されたときにエラーを返して読み取ります。

また、ディスク自体が検出できないような、ディスクブロックが破損することもあります。たとえば、バグのあるディスクファームウェアは、間違った場所にブロックを書き込む可能性があります。そのような場合、ディスク ECC はブロックの内容が正常であることを示しますが、クライアントの観点からは、後でアクセスしたときに間違ったブロックが返されます。同様に、ブロックが障害のあるバスを経由してホストからディスクに転送されると、クライアントが望んでいない破損したデータがディスクに格納されます。これらのタイプの障害は、サイレントフォールトであるため特に危険です。故障したデータを戻すときに問題の兆候は見られません。

Prabhakaran et al は、ディスク障害のこのより現代的な見解を、部分ディスク故障モデル [P + 05] として説明しています。このビューでは、従来のフェイルストップモデルの場合のように、ディスク全体が引き続き故障する可能性があります。つまり、ディスクは見かけ上動作していますが、1つまたは複数のブロックがア

クセス不能(すなわち LSE)になつたり、間違った内容(すなわち、破損)を保持する可能性があります。したがつて、一見して動作するディスクにアクセスすると、与えられたブロックを読み書きしようとするとき(サイレントではない部分的なフォールト)、エラーを返したり、間違ったデータを返すことがあります(部分的なサイレントフォールト)。これらの両方のタイプの障害は、まれですが、どれだけ稀でしょうか? 図 45.1 は、2つのバイラバンダラム研究 [B + 07, B + 08] の結果の一部をまとめたものです。

	Cheap	Costly
LSEs	9.40%	1.40%
Corruption	0.50%	0.05%

Figure 45.1: Frequency Of LSEs And Block Corruption

図は、調査の過程で少なくとも 1 つの LSE またはブロック破損を示したドライブの割合を示しています(約 3 年間、150 万台以上のディスクドライブ)。この図はさらに、結果を「安価な」ドライブ(通常は SATA ドライブ)と「高価な」ドライブ(通常は SCSI または FibreChannel)に細分しています。お分かりのように、より良いドライブを購入することで、両方のタイプの問題の頻度が(桁違いに)低下しますが、ストレージシステムでの処理方法を慎重に考えなければならないほど頻繁に発生します。

LSEs に関する追加的な調査結果は次のとおりです。

- 複数の LSEs を使用する高価なドライブは、安価なドライブと同じくらい追加のエラーを発生させる可能性が高い - ほとんどのドライブでは、年間エラー率は 2 年目に増加します - ディスクサイズに合わせて LSEs の数が増えます - LSEs を持つほとんどのディスクは 50 未満です - LSEs を持つディスクは、追加の LSEs を開発する可能性が高い - かなりの量の空間的および時間的局所性が存在する - ディスクスクラブが便利です(ほとんどの LSEs はこのように見つかっています)

破損に関するいくつかの発見は次の通りです。

- 破損の可能性は、同じドライブクラス内のさまざまなドライブモデルによって大きく異なります - 年季による影響はモデルによって異なります - 仕事量とディスクサイズが破損にほとんど影響しない - 破損しているほとんどのディスクにはいくつかの破損しかありません - 破損は、ディスク内または RAID 内のディスク間で独立していない - 空間的局所性と時間的局所性が存在する - LSEs との相関は低い

これらの失敗の詳細については、元の論文 [B + 07, B + 08] を読むべきでしょう。しかし、うまくいけば、要点は明らかです。信頼できるストレージシステムを本当に構築したいのであれば、LSEs と破損のブロックのそれぞれの検出と復旧のための機械を組み込む必要があります。

45.2 Handling Latent Sector Errors

部分的なディスク障害のこれらの 2 つの新しいモードを考えると、我々は今、それらについて何ができるのかを見極めるべきです。最初に 2 つのうちの latent sector errors と名前がついている簡単な方に取り組んでみましょう。

CRUX: HOW TO HANDLE LATENT SECTOR ERRORS

ストレージシステムは latent sector errors をどのように処理すべきですか? このような部分的な失敗を処理するには、どれくらい余分な機械が必要ですか?

結果として、latent sector errors は(定義によって)容易に検出されるので、扱うのが簡単です。ストレージシステムがブロックにアクセスしようとしたときに、ディスクがエラーを返した場合、ストレージシステムは正しいデータを返すために必要な冗長メカニズムを使用するだけです。たとえば、ミラーリングされた RAID では、システムは代替コピーにアクセスする必要があります。パリティベースの RAID-4 または RAID-5 シ

システムでは、パリティグループの他のブロックからブロックを再構築する必要があります。したがって、LSEsなどの容易に検出される問題は、標準的な冗長性メカニズムを通じて容易に回復されます。

LSEs の普及率は、長年にわたり RAID 設計に影響を与えてきました。フルディスク障害と LSEs の両方が同時に発生すると、特に興味深い問題が RAID-4/5 システムで発生します。具体的には、ディスク全体が故障すると、RAID はパリティグループ内の他のすべてのディスクを読み取り、欠損値を再計算することによってディスクを(たとえば、ホットスペアに)再構築しようと試みます。再構築中に他のディスクのいずれかで LSE が発生した場合、問題が発生します。再構成が正常に完了できません。

この問題を解決するために、いくつかのシステムでは余分な冗長性が追加されています。たとえば、NetApp の RAID-DP は、一つの [C + 04] ではなく 2 つのパリティディスクに相当します。再構成中に LSE が検出されると、余分なパリティが欠損ブロックを再構築するのに役立ちます。いつものように、各ストライプの 2 つのパリティブロックを維持する方がコストがかかります。しかし、NetApp WAFL ファイルシステムのログ構造は、多くの場合、コストを軽減します [HLM94]。残りのコストは、第 2 のパリティブロック用の余分なディスクのスペースです。

45.3 Detecting Corruption: The Checksum

では、より挑戦的な問題であるデータ破損による silent failures の問題に取り組んでみましょう。破損が発生した場合にユーザーが悪意のあるデータを取得するのを防ぐことができ、悪意あるデータを返すディスクに導けるでしょうか？

CRUX: HOW TO PRESERVE DATA INTEGRITY DESPITE CORRUPTION

このようなサイレントな障害の性質を考えると、ストレージシステムは破損が発生したときに何を検出することができますか？どんな技術が必要ですか？どのように効率的に実装できますか？

latent sector errors とは異なり、破損の検出は重要な問題です。ブロックが不良であるとクライアントがどのように伝えることができますか？特定のブロックが不良であることが分かったら、復旧は前と同じです。ブロックのコピーをいくつか用意する必要があります(うまくいけば、壊れていないこともあります)。したがって、我々はここで検出技術に焦点を当てます。

最新のストレージシステムでデータの整合性を保持するために使用される主なメカニズムはチェックサムと呼ばれます。チェックサムは、単にデータの塊(例えば 4KB のブロック)を入力とし、そのデータ上の関数を計算してデータの内容(例えば 4 または 8 バイト)の小さな要約を生成する関数の結果です。この概要はチェックサムと呼ばれます。このような計算の目的は、データにチェックサムを格納し、その後にデータの現在のチェックサムが元のストレージ値と一致することによって、データが何らかの形で壊れているか、変更されているかどうかを検出できるようにすることです。

TIP: THERE'S NO FREE LUNCH

There's No Such Thing As A Free Lunch、つまり TNSTAAFL は古いアメリカのイディオムであり、あなたが何かを無料で手に入れているときには、実際には多少の費用がかかります。ダイナーが顧客に無料のランチを宣伝し、飲み物を引き出そうと考えていた昔からのものです。あなたが入ったときにだけ、あなたは“無料”ランチを取得するために、あなたは 1 つ以上のアルコール飲料を購入しなければならないことを認識しましたか？もちろん、特にあなたがアルコール依存症(または典型的な学部生)である場合、これは実際問題ではないかもしれません。

Common Checksum Functions

多数の異なる関数がチェックサムを計算するために使用され、強さ（すなわち、それらがデータ保全性をいかに良好に保っているか）および速度（すなわち、それらをいかに迅速に計算することができるか）で変化します。ここではシステムに共通するトレードオフが発生します。通常、より多くの保護を受けるほど、コストが高くなります。free lunchなどはありません。

あるものは排他的論理和 (XOR) に基づく単純なチェックサム関数です。XOR ベースのチェックサムの場合、チェックサムは、チェックサムされるデータブロックの各チャンクを排他的論理和 (XOR) することによって計算され、ブロック全体の XOR を表す单一の値を生成します。

これをより具体的にするために、16 バイトのブロックに対して 4 バイトのチェックサムを計算していると想像してください（このブロックは実際にディスクセクタまたはブロックになるには小さすぎますが、例のために役立ちます）。16 データバイトの 16 進数で、次のようにになります。

365e c4cd ba14 8a92 ecef 2c3a 40be f666

バイナリ形式で表示すると、次のようにになります。

0011 0110 0101 1110	1100 0100 1100 1101
1011 1010 0001 0100	1000 1010 1001 0010
1110 1100 1110 1111	0010 1100 0011 1010
0100 0000 1011 1110	1111 0110 0110 0110

行ごとに 4 バイトのグループにデータを並べたので、結果のチェックサムがどのようになるかを簡単に確認できます。各列で XOR を実行して最終的なチェックサム値を取得します。

0010 0000 0001 1011 1001 0100 0000 0011

結果は 16 進数で 0x201b9403 です。XOR は合理的なチェックサムですが、限界があります。たとえば、チェックサム単位内の同じ位置の 2 ビットが変化した場合、チェックサムは破損を検出しません。このため、人々は他のチェックサム機能を調査しました。

もう 1 つの基本的なチェックサム機能は追加です。このアプローチには、高速であるという利点があります。オーバーフローを無視して、データの各チャンクに対して 2 の補数加算を実行するだけです。それはデータの多くの変化を検出することができるが、例えばデータがシフトされた場合には良くありません。

ちょっと複雑なアルゴリズムが Fletcher チェックサムとして知られていますが、これは発明者 John F. Fletcher 氏 [F82] の名前です。計算が非常に簡単で、2 つのチェックバイト s_1 と s_2 の計算が必要です。具体的には、ブロック D がバイト $d_1 \dots d_n$ で構成され、 s_1 は以下のように定義されます： $s_1 = (s_1 + d_i) \text{mod} 255$ (全ての d_i に対して計算される)。 s_2 は、 $s_2 = (s_2 + s_1) \text{mod} 255$ (全ての d_i に対して計算される) [F04] です。Fletcher のチェックサムは、すべてのシングルビット、ダブルビットエラー、および多くのバーストエラー [F04] を検出して、CRC とほぼ同じくらい強いです（下記参照）。

1 つの最終的に使用されるチェックサムは、Cyclic Redundancy Check(CRC) として知られています。データブロック D を介してチェックサムを計算したいと仮定します。 D を大規模な 2 進数であるかのように扱います（結局のところビット列です）、合意された値 (k) で割ります。この除算の残りは CRC の値です。明らかになつたように、このバイナリモジュロ演算をむしろ効率的に実装することができ、したがってネットワーキ

ングにおける CRC の人気は高いです。詳細は [M13] を参照してください。

どのような方法を使用しても、完全なチェックサムがないことは明らかです。内容が同一ではない 2 つのデータブロックに同じチェックサムがある可能性があります。結局のところ、チェックサムを計算することは、何かを大きく (例えば 4KB) 取って、はるかに小さい (例えば 4 または 8 バイト) 要約を生成することです。つまり、直感的でなければいけません。良いチェックサム関数を選ぶ際に、我々は衝突の可能性を最小限に抑えながら計算しやすいものを見つけることを試みています。

Checksum Layout

チェックサムを計算する方法について少し理解したので、次にストレージシステムでチェックサムを使用する方法を分析しましょう。最初に問題となるのは、チェックサムのレイアウト、つまりどのようにチェックサムをディスクに保存すればよいでしょうか？

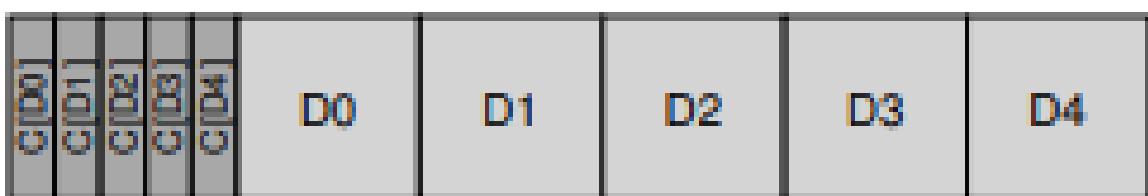
最も基本的なアプローチでは、各ディスクセクタ (またはブロック) にチェックサムを格納するだけです。データブロック D が与えられたら、そのデータ C(D) でチェックサムを呼び出します。したがって、チェックサムがないと、ディスクレイアウトは次のようにになります。



チェックサムを使用すると、レイアウトはすべてのブロックに対して 1 つのチェックサムを追加します。



チェックサムは通常小さい (例えば 8 バイト) ディスクであり、ディスクはセクタサイズのチャンク (512 バイト) またはその倍数でしか書き込めないので、上記のレイアウトをどのように達成するかという問題があります。ドライブメーカーが採用している解決策の 1 つは、ドライブを 520 バイトのセクタでフォーマットすることです。1 セクタあたり余分な 8 バイトを使用してチェックサムを格納することができます。このような機能を持たないディスクでは、ファイルシステムは 512 バイトのブロックにパックされたチェックサムを格納する方法を理解しなければなりません。そのような可能性の 1 つは次のとおりです。



このスキームでは、n 個のチェックサムがセクタ内に一緒に格納され、その後に n 個のデータブロックが続き、次の n 個のブロック用のチェックサムセクタが... と繰り返します。このスキームは、すべてのディスクで作業する利点がありますが、効率が低下する可能性があります。たとえば、ファイルシステムがブロック D1 を上書きしたい場合、C(D1) を含むチェックサムセクタを読み込み、その中の C(D1) を更新してから、チェックサムセクタおよび新しいデータブロックを書き出す必要があります D1(したがって、1 回の読み出しと 2 回の

書き込み)。以前のアプローチ(1セクタあたり1つのチェックサム)は単なる書き込みを実行するだけです。

45.4 Using Checksums

チェックサムレイアウトが決定したら、チェックサムの使用方法を実際に理解することができます。ブロック D を読むとき、クライアント(すなわち、ファイルシステムまたはストレージコントローラ)は、ディスク C_s(D) からそのチェックサムを読み取ります。これを stored checksum(したがって添え字 C_s) と呼びます。次に、クライアントは、検索されたブロック D に対するチェックサムを計算します。これは、computed checksum(C_c(D)) と呼ばれます。この時点で、クライアントは stored checksum と computed checksum を比較します。もしそれらが等しければ(すなわち、 $C_s(D) == C_c(D)$)、データは破損していない可能性があり、したがって安全にユーザに戻すことができます。もしそれらが等しくないとき(すなわち、 $C_s(D) != C_c(D)$)、これは、格納されたチェックサムがその時点でのデータの値を反映しているため、データが格納されてから変更されたことを意味します。この場合、私たちはチェックサムが検出するのに役立つ破損があります。

破損を考えると、自然な疑問は私たちがそれについて何をすべきかということです。ストレージシステムに冗長コピーがある場合、その答えは簡単です。代わりに使用してみてください。ストレージシステムにそのようなコピーがない場合は、エラーを返す可能性があります。どちらの場合でも、破損の検出は魔法の弾丸ではないことを認識してください。他に破損していないデータを取得する方法がない場合、あなたは単に運がないだけです。

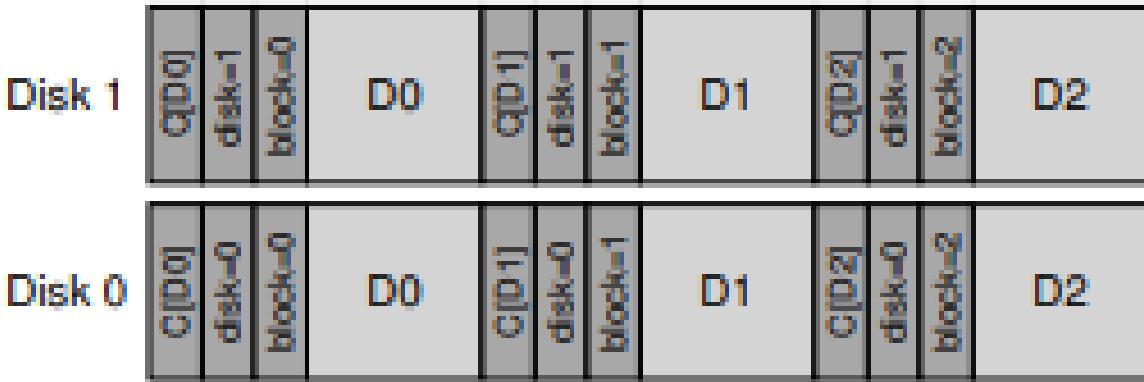
45.5 A New Problem: Misdirected Writes

上記の基本的なスキームは、破損したブロックの一般的なケースでうまくいきます。しかし、現代のディスクには、さまざまなソリューションを必要とする異常ないいくつかの failure モードがあります。関心のある第1の failure モードは、misdirected write(誤った方向の書き込み)と呼ばれます。これは、間違った場所以外のデータをディスクに正しく書き込むディスクコントローラと RAID コントローラで発生します。单一ディスクシステムでは、これは、ディスクがブロック D_x を(おそらくは)xに対応するのではなく、y をアドレス指定する(したがって、“破損する”D_y)ように書いたことを意味します。さらに、マルチ・ディスク・システムでは、コントローラは、ディスク i の x にアドレスするのではなく、むしろ他のディスク j に書き込むかもしれません。

CRUX: HOW TO HANDLE MISDIRECTED WRITES

ストレージシステムまたはディスクコントローラは、misdirected writes をどのように検出すべきですか？ チェックサムにはどのような追加機能が必要ですか？

驚くことではないですが、答えは簡単です。各チェックサムに少しだけ情報を追加してください。この場合、物理識別子(物理 ID)を追加することは非常に役に立ちます。例えば、格納された情報がブロックのディスク及びセクタ番号と共にチェックサム C(D) を含むならば、クライアントが正しい情報がブロック内に存在するかどうかを容易に判断することができます。具体的には、クライアントがディスク 10 のブロック 4(D_10,4) を読み取っている場合、格納された情報には、そのディスク番号とセクタオフセットが含まれている必要があります(下図参照)。情報が一致しない場合、間違った書き込みが行われ、破損が検出されます。2ディスクシステムでこのような追加情報がどのように表示されるかの例を次に示します。この図では、チェックサムは通常は小さい(例えば 8 バイト)一方、ブロックははるかに大きい(例えば、4KB 以上)ので、前の他のものと同様に、この数字は縮尺通りではないことに注意してください。



ディスク上には冗長性がかなりあることがディスク上のフォーマットからわかります。各ブロックごとにディスク番号が各ブロック内で繰り返され、ブロックのオフセットもブロック自体の横に保持されます。冗長な情報の存在は驚くべきことではありません。冗長性は、エラー検出(この場合)および回復(他の場合)の鍵です。わずかな余分な情報は、完璧なディスクで厳密には必要とされませんが、問題が発生した場合に問題のある状況を検出するのに役立ちます。

45.6 One Last Problem: Lost Writes

残念ながら、間違った書き込みは私たちが直面する最後の問題ではありません。具体的には、現代の記憶装置の中には、書き込みが完了したことを装置が上位層に通知したにもかかわらず、失われた書き込みとして知られる問題もあります。したがって、残っているのは、更新された新しいコンテンツではなく、ブロックの古いコンテンツです。

明らかな問題は、上記のチェックサム戦略(基本的なチェックサムや物理的な識別情報など)が失われた書き込みを検出するのに役立つかどうかです。残念なことに、答えはノーです。古いブロックは一致するチェックサムを持つ可能性が高く、上記の物理 ID(ディスク番号とブロックオフセット)も正しいでしょう。

CRUX: HOW TO HANDLE LOST WRITES

ストレージシステムまたはディスクコントローラは、失われた書き込みをどのように検出する必要がありますか？ チェックサムにはどのような追加機能が必要ですか？

[K + 08] に役立つさまざまな解決策があります。1つの古典的なアプローチ [BS04] は、書き込み検証または read after write を実行することです。書き込み後にデータを直ちに読み戻すことによって、システムはデータが実際にディスク表面に到達することを保証することができます。しかし、このアプローチは非常に遅く、書き込みを完了するために必要な I/O の数が倍になります。

一部のシステムでは、システム内の他の場所でチェックサムを追加して、失われた書き込みを検出します。たとえば、Sun の ZFS(Zettabyte File System)には、各ファイルシステムの i ノードとファイルに含まれるすべてのブロックの間接ブロックにチェックサムが含まれています。したがって、データブロック自体への書き込みが失われても、inode 内のチェックサムは古いデータと一致しません。inode とデータの両方への書き込みが同時に失われた場合にのみ、そのようなスキームは失敗しますが、(残念ながら可能です！) そのような状況はありません。

45.7 Scrubbing

この議論のすべてを考えると、あなたは疑問に思うかもしれません。いつこれらのチェックサムが実際にチェックされますか？ もちろん、データがアプリケーションによってアクセスされるときにいくらかのチェック

クが行われますが、ほとんどのデータはアクセスすることはほとんどないため、チェックされません。チェックされていないデータは、信頼性の高いストレージシステムでは問題になります。ビット腐敗が特定のデータのすべてのコピーに最終的に影響する可能性があるからです。

この問題を解決するために、多くのシステムではさまざまな形式のディスクスクラビングが使用されています [K + 08]。システムのすべてのブロックを定期的に読み取り、チェックサムが有効かどうかをチェックすることで、ディスクシステムは、特定のデータ項目のすべてのコピーが破損する可能性を減らすことができます。典型的なシステムは、夜間または週単位でスキャンをスケジュールします。

45.8 Overheads Of Checksumming

終了する前に、データ保護のためにチェックサムを使用するオーバーヘッドのいくつかについて説明します。コンピュータシステムで一般的であるように、スペースと時間という 2 つの異なる種類のオーバーヘッドがあります。

スペースのオーバーヘッドには 2 つの形式があります。最初はディスク (または他の記憶媒体) 自体にあります。格納された各チェックサムはディスク上の空き領域を占有します。これはもはやユーザーデータに使用できません。典型的な比率は、ディスク上のスペースのオーバーヘッドが 0.19 % であるため、4 KB のデータブロックにつき 8 バイトのチェックサムになります。

第 2 の種類のスペースオーバーヘッドは、システムのメモリ内にある。データにアクセスするときに、チェックサムとデータそのものためにメモリに余裕がなければなりません。しかし、システムが単にチェックサムをチェックし、それが一旦終了するこのオーバーヘッドは短命であり、懸念するものではありません。チェックサムがメモリに保持されている場合 (メモリ破損 [Z + 13] に対する保護レベルが追加されている場合) に限り、この小さなオーバーヘッドは観測可能です。

スペースのオーバーヘッドは小さいものの、チェックサムによる時間オーバーヘッドはかなり目立つことがあります。最低限、CPU は、データが格納されているとき (格納されているチェックサムの値を決定するとき) とアクセスされたとき (チェックサムを再度計算してそれを格納されたチェックサムと比較するとき) の各ブロックのチェックサムを計算する必要があります。チェックサム (ネットワークスタックを含む) を使用する多くのシステムで採用されている CPU オーバーヘッドを削減する方法の 1 つは、データコピーとチェックサムを 1 つの効率的なアクティビティに組み合わせることです。何らかの形で (例えば、カーネルページキャッシュからユーザバッファにデータをコピーするために) コピーが必要とされるので、コピー/チェックサムを組み合わせることは非常に効果的です。

CPU オーバーヘッド以外にも、いくつかのチェックサム方式により、余分な I/O オーバーヘッドが発生する可能性があります。具体的には、チェックサムがデータとは区別されて格納されている (特にアクセスするための余計な I/O が必要な場合)、background scrubbing に必要な余分な I/O です。前者は設計によって減らすことができます。おそらくこのような scrubbing 活動がいつ行われるかを制御することによって、後者は調整され、その影響は制限されます。真夏の夜中に、生産労働者のほとんどが就寝してしまったので、このような scrubbing 活動を実行してストレージシステムの堅牢性を高める良い機会になるかもしれません。

45.9 Summary

現代のストレージシステムでは、チェックサムの実装と使用に焦点を当てて、データ保護について説明しました。異なるチェックサムが異なるタイプのフォルトに対して保護します。記憶装置が進化するにつれて、新しい failure modes が間違いなく発生するでしょう。おそらくこのような変化は、研究界や産業界にこれらの基本的アプローチのいくつかを再び使ったり、まったく新しいアプローチを発明したりすることになるでしょう。それは時間が教えてくれるかもしれませんし、そうではないかもしれません。そういう意味で時間というのは面白いです。

参考文献

[B+07] “An Analysis of Latent Sector Errors in Disk Drives”

Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, Jiri Schindler

SIGMETRICS '07, San Diego, California, June 2007

The first paper to study latent sector errors in detail. As described in the next citation [B+08], a collaboration between Wisconsin and NetApp. The paper also won the Kenneth C. Sevcik Outstanding Student Paper award; Sevcik was a terrific researcher and wonderful guy who passed away too soon. To show the authors it was possible to move from the U.S. to Canada and love it, he once sang us the Canadian national anthem, standing up in the middle of a restaurant to do so.

[B+08] “An Analysis of Data Corruption in the Storage Stack”

Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder,

Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

FAST '08, San Jose, CA, February 2008

The first paper to truly study disk corruption in great detail, focusing on how often such corruption occurs over three years for over 1.5 million drives. Lakshmi did this work while a graduate student at Wisconsin under our supervision, but also in collaboration with his colleagues at NetApp where he was an intern for multiple summers. A great example of how working with industry can make for much more interesting and relevant research.

[BS04] “Commercial Fault Tolerance: A Tale of Two Systems”

Wendy Bartlett, Lisa Spainhower

IEEE Transactions on Dependable and Secure Computing, Vol. 1, No. 1, January 2004

This classic in building fault tolerant systems is an excellent overview of the state of the art from both IBM and Tandem. Another must read for those interested in the area.

[C+04] “Row-Diagonal Parity for Double Disk Failure Correction”

P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, S. Sankar

FAST '04, San Jose, CA, February 2004

An early paper on how extra redundancy helps to solve the combined full-disk-failure/partial-disk-failure problem. Also a nice example of how to mix more theoretical work with practical.

[F04] “Checksums and Error Control”

Peter M. Fenwick

Copy Available: <http://www.ostep.org/Citations/checksums-03.pdf>

A great simple tutorial on checksums, available to you for the amazing cost of free.

[F82] “An Arithmetic Checksum for Serial Transmissions”

John G. Fletcher

IEEE Transactions on Communication, Vol. 30, No. 1, January 1982

Fletcher's original work on his eponymous checksum. Of course, he didn't call it the Fletcher checksum, rather he just didn't call it anything, and thus it became natural to name it after the inventor. So don't blame old Fletch for this seeming act of braggadocio. This anecdote might remind you of Rubik and his cube; Rubik never called it “Rubik's cube”; rather, he just called it “my cube.”

[HLM94] “File System Design for an NFS File Server Appliance”

Dave Hitz, James Lau, Michael Malcolm

USENIX Spring '94

The pioneering paper that describes the ideas and product at the heart of NetApp’s core. Based on this system, NetApp has grown into a multi-billion dollar storage company. If you’re interested in learning more about its founding, read Hitz’s autobiography “How to Castrate a Bull: Unexpected Lessons on Risk, Growth, and Success in Business” (which is the actual title, no joking). And you thought you could avoid bull castration by going into Computer Science.

[K+08] “Parity Lost and Parity Regained”

Andrew Krioukov, Lakshmi N. Bairavasundaram, Garth R. Goodson, Kiran Srinivasan, Randy Thelen, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

FAST ’08, San Jose, CA, February 2008

This work of ours, joint with colleagues at NetApp, explores how different checksum schemes work (or don’t work) in protecting data. We reveal a number of interesting flaws in current protection strategies, some of which have led to fixes in commercial products.

[M13] “Cyclic Redundancy Checks”

Author Unknown

Available: <http://www.mathpages.com/home/kmath458.htm>

Not sure who wrote this, but a super clear and concise description of CRCs is available here. The internet is full of information, as it turns out.

[P+05] “IRON File Systems”

Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

SOSP ’05, Brighton, England, October 2005

Our paper on how disks have partial failure modes, which includes a detailed study of how file systems such as Linux ext3 and Windows NTFS react to such failures. As it turns out, rather poorly! We found numerous bugs, design flaws, and other oddities in this work. Some of this has fed back into the Linux community, thus helping to yield a new more robust group of file systems to store your data.

[RO91] “Design and Implementation of the Log-structured File System”

Mendel Rosenblum and John Ousterhout

SOSP ’91, Pacific Grove, CA, October 1991

Another reference to this ground-breaking paper on how to improve write performance in file systems.

[S90] “Implementing Fault-Tolerant Services Using The State Machine Approach: A Tutorial”

Fred B. Schneider

ACM Surveys, Vol. 22, No. 4, December 1990

This classic paper talks generally about how to build fault tolerant services, and includes many basic definitions of terms. A must read for those building distributed systems.

[Z+13] “Zettabyte Reliability with Flexible End-to-end Data Integrity”

Yupu Zhang, Daniel S. Myers, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

MSST ’13, Long Beach, California, May 2013

Our own work on adding data protection to the page cache of a system, which protects against memory corruption as well as on-disk corruption.

48 Distributed Systems

分散システムは世界の顔を変えました。あなたの Web ブラウザが地球上の他のどこかの Web サーバに接続するとき、それは単純な形式のクライアント/サーバ分散システムのように見えます。しかし、Google や Facebook などの最新の Web サービスにコンタクトすると、ただ 1 台のマシンと対話するだけではありません。これらの複雑なサービスの舞台裏では、大規模な収集（すなわち、数千件）のマシンから構築され、それが協力してサイトの特定のサービスを提供します。したがって、分散システムを研究することが興味深いのは明らかです。実際、それはクラス全体にふさわしいものです。ここでは、主要なトピックのいくつかを紹介します。

分散システムを構築する際には、数多くの新たな課題が生じます。我々が重点的に取り組むのは失敗です。「完璧な」コンポーネントやシステムを構築する方法を知らない（そして、決してそうなることはありません）ので、機械、ディスク、ネットワーク、およびソフトウェアはすべて時々失敗します。しかし、現代の Web サービスを構築するときには、失敗しないかのようにクライアントに表示したいと考えています。どうすればこの作業を達成できますか？

THE CRUX: HOW TO BUILD SYSTEMS THAT WORK WHEN COMPONENTS FAIL

常に正しく動作しない部品から作業システムを構築するにはどうすればよいですか？ 基本的な質問は、RAID ストレージアレイで議論したトピックのいくつかを思い出させるはずです。しかし、ここでの問題は、ソリューションと同様に、より複雑になる傾向があります。

面白いことに、障害は分散システムを構築する上での中心的課題ですが、チャンスもあります。マシンは失敗します。マシンが故障したという事実だけでは、システム全体が故障しなければならないということを意味するものではありません。一連のマシンをまとめて、コンポーネントが定期的に故障しているにもかかわらず、ほとんど失敗しないシステムを構築することができます。この現実は、分散システムの中心的な美しさと価値です。なぜなら、それらが Google や Facebook など、あなたが使っているすべての最新の Web サービスの下にある理由です。

TIP: COMMUNICATION IS INHERENTLY UNRELIABLE

事実上すべての状況において、通信を根本的に信頼できない活動と見なすことは良いことです。ビットの破損、ダウンや動いていないリンクやマシン、着信パケットのバッファースペースの不足は、すべて同じ結果につながります。パケットが宛先に到達しないことがあります。そのような信頼できないネットワークの上に信頼できるサービスを構築するためには、パケット損失に対処できる技術を検討する必要があります。

その他の重要な問題も存在します。システムのパフォーマンスはよく重要です。私たちの分散システムと一緒に接続するネットワークで、システム設計者は、与えられたタスクをどのように達成するか、送信されるメッセージの数を減らし、通信を効率的（低い待ち時間、高帯域幅）にできる限り慎重に考えなければならないことがあります。

最後に、セキュリティも必要な考慮事項です。遠隔地に接続するとき、遠隔地の当事者が誰であるかを保証することが中心的な問題になります。さらに、第三者が 2 つの間の進行中の通信を監視または変更できないようにすることもまた課題です。

ここでは、分散システムで最も新しい通信の基本的な側面について説明します。すなわち分散システム内のマシンは、どのように相互に通信する必要がありますか？ 利用可能な最も基本的なプリミティブ、メッセージから始めそれらの上にいくつかのより高いレベルのプリミティブを構築します。上記したように、障害は中心的な焦点になります。通信レイヤが障害をどのように処理すべきですか？

48.1 Communication Basics

現代のネットワークの中心的な教えは、通信は根本的に信頼性がないということです。広域インターネットでも、Infinibandなどのローカルエリア高速ネットワークでも、パケットは定期的に失われたり、破損したり、宛先に到達しない場合があります。

パケットの損失や破損の原因は多数あります。時には伝送中に、いくつかのビットが電気的または他の同様の問題のために反転されます。場合によっては、ネットワーククリンクやパケットルーター、リモートホストなど、システム内の要素が何らかの形で損傷を受けたり正しく動作しないことがあります。ネットワークケーブルが誤って切断されることがあります。

しかし、より根本的なのは、ネットワークスイッチ、ルータ、エンドポイントでのバッファリングの不足によるパケット損失です。具体的には、すべてのリンクが正しく機能し、システム内のすべてのコンポーネント（スイッチ、ルータ、エンドホスト）が期待どおりに稼動していることを保証できる場合でも、次の理由により、まだ失われる可能性があります。パケットがルータに到着したとします。処理されるパケットについては、ルータ内のどこかのメモリに配置する必要があります。このようなパケットが多数到着すると、ルータ内のメモリがすべてのパケットに対応できない可能性があります。その時点でルータが持つ唯一の選択肢は、1つまたは複数のパケットをドロップすることです。この同様の現象はエンドホストでも発生します。単一のマシンに大量のメッセージを送信すると、マシンのリソースが容易に圧倒され、パケット損失が再び発生します。

従って、パケット損失はネットワーキングにおいて基本的なものです。したがって、問題は次のようになります。これをどう処理しますか？

```
// client code
int main(int argc, char *argv[]) {
    int sd = UDP_Open(20000);
    struct sockaddr_in addrSnd, addrRcv;
    int rc = UDP_FillSockAddr(&addrSnd, "machine.cs.wisc.edu", 10000);
    char message[BUFFER_SIZE];
    sprintf(message, "hello world");
    rc = UDP_Write(sd, &addrSnd, message, BUFFER_SIZE);
    if (rc > 0) {
        int rc = UDP_Read(sd, &addrRcv, message, BUFFER_SIZE);
    }
    return 0;
}

// server code
int main(int argc, char *argv[]) {
    int sd = UDP_Open(10000);
    assert(sd > -1);
    while (1) {
        struct sockaddr_in addr;
        char message[BUFFER_SIZE];
        int rc = UDP_Read(sd, &addr, message, BUFFER_SIZE);
        if (rc > 0) {
            char reply[BUFFER_SIZE];
            sprintf(reply, "goodbye world");
            rc = UDP_Write(sd, &addr, reply, BUFFER_SIZE);
        }
    }
    return 0;
}
```

Figure 48.1: Example UDP/IP Client/Server Code

48.2 Unreliable Communication Layers

1つの簡単な方法はこれです。それに対処しないことです。一部のアプリケーションでは、パケットロスを処理する方法が分かっているため、よく聞かれるエンドツーエンド(章の最後を見てください)の議論の一例である、基本的な信頼性の低いメッセージレイヤーと通信できるようにするのが便利な場合があります。このような信頼性の低いレイヤの優れた例は、今日のほぼすべての現代システムで利用可能な UDP/IP ネットワーキングスタックにあります。UDP を使用するには、通信エンドポイントを作成するためにソケット API を使用します。他のマシン上(または同じマシン上)のプロセスは元のプロセスに UDP データグラムを送ります(データグラムは最大サイズまでの固定サイズのメッセージです)。

```

int UDP_Open(int port) {
    int sd;
    if ((sd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) { return -1; }
    struct sockaddr_in myaddr;
    bzero(&myaddr, sizeof(myaddr));
    myaddr.sin_family      = AF_INET;
    myaddr.sin_port        = htons(port);
    myaddr.sin_addr.s_addr = INADDR_ANY;
    if (bind(sd, (struct sockaddr *) &myaddr, sizeof(myaddr)) == -1) {
        close(sd);
        return -1;
    }
    return sd;
}

int UDP_FillSockAddr(struct sockaddr_in *addr, char *hostName, int port) {
    bzero(addr, sizeof(struct sockaddr_in));
    addr->sin_family = AF_INET;           // host byte order
    addr->sin_port   = htons(port);       // short, network byte order
    struct in_addr *inAddr;
    struct hostent *hostEntry;
    if ((hostEntry = gethostbyname(hostName)) == NULL) { return -1; }
    inAddr = (struct in_addr *) hostEntry->h_addr;
    addr->sin_addr = *inAddr;
    return 0;
}

int UDP_Write(int sd, struct sockaddr_in *addr, char *buffer, int n) {
    int addrLen = sizeof(struct sockaddr_in);
    return sendto(sd, buffer, n, 0, (struct sockaddr *) addr, addrLen);
}

int UDP_Read(int sd, struct sockaddr_in *addr, char *buffer, int n) {
    int len = sizeof(struct sockaddr_in);
    return recvfrom(sd, buffer, n, 0, (struct sockaddr *) addr,
                   (socklen_t *) &len);
}

```

Figure 48.2: A Simple UDP Library

図 48.1 と図 48.2 に、UDP/IP の上に構築された単純なクライアントとサーバーを示します。クライアントは、サーバーにメッセージを送信し応答を返します。この少量のコードで分散システムの構築を始めてみてください。

UDP は、信頼性の低い通信レイヤの大きな例です。それを使用すると、パケットが紛失(ドロップ)し、宛先に到達しないという状況に遭遇します。送信者は決して損失の通知されません。しかし、それは UDP が何の失敗に対しても守らないということを意味するものではありません。たとえば、UDP にはいくつかの形式

のパケット破損を検出するためのチェックサムが含まれています。

しかし、多くのアプリケーションは単に宛先にデータを送信し、パケットの消失を心配する必要がないため、もっと多くの仕組みが必要です。具体的には、信頼できないネットワーク上に信頼できる通信が必要です。

TIP: USE CHECKSUMS FOR INTEGRITY

チェックサムは、最新のシステムで迅速かつ効果的に破損を検出するためによく使用される方法です。単純なチェックサムは加算です。データのまとまりのバイトを合計します。もちろん基本的な巡回冗長コード (CRC)、フレッチャーチェックサム、その他多数の他の多くの洗練されたチェックサムが作成されています [MK09]。ネットワークでは、チェックサムは次のように使用されます。あるマシンから別のマシンにメッセージを送信する前に、そのメッセージのバイトのチェックサムを計算します。次に、メッセージとチェックサムの両方を宛先に送信します。宛先では、受信側は受信メッセージ上のチェックサムを計算します。この計算されたチェックサムが送信されたチェックサムと一致する場合、受信者は、データが送信中に破損しない可能性があるという確信を感じることができます。チェックサムは、いくつかの異なる軸に沿って評価することができます。効率性の主な考慮事項の1つは、データの変更がチェックサムの変更につながるかどうかです。チェックサムが強ければ強いほど、データの変化が気付かれなくなります。パフォーマンスはもう一つの重要な基準です。チェックサムの計算にはどのくらいのコストがかかりますか？ 残念なことに、有効性とパフォーマンスはよく不安定です。つまり、高品質のチェックサムは計算コストがかかることがあります。人生はまた完璧ではありません。

48.3 Reliable Communication Layers

信頼性の高い通信レイヤを構築するには、パケットロスを処理するための新しいメカニズムと技術が必要です。クライアントが信頼性の低い接続を介してサーバーにメッセージを送信する簡単な例を考えてみましょう。最初に質問しなければならないのは、送信者は受信者が実際にメッセージを受信したことをどのように知っていますか？ ということです。私たちが使用するテクニックは、肯定応答または ack と呼ばれています。アイデアは単純です。送信者は受信者にメッセージを送信します。受信者はその受付の肯定応答のために短いメッセージを送り返します。図 48.3 にプロセスを示します。

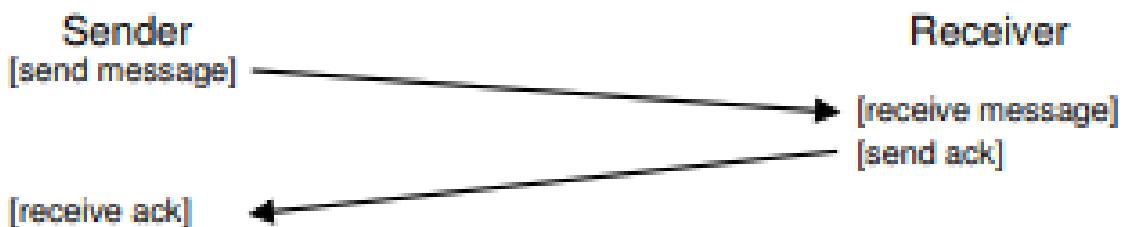


Figure 48.3: Message Plus Acknowledgment

送信者がメッセージの肯定応答を受信すると、受信者が実際に元のメッセージを受信したことを安心することができます。しかし、肯定応答を受信しなかった場合、送信者は何をすべきですか？

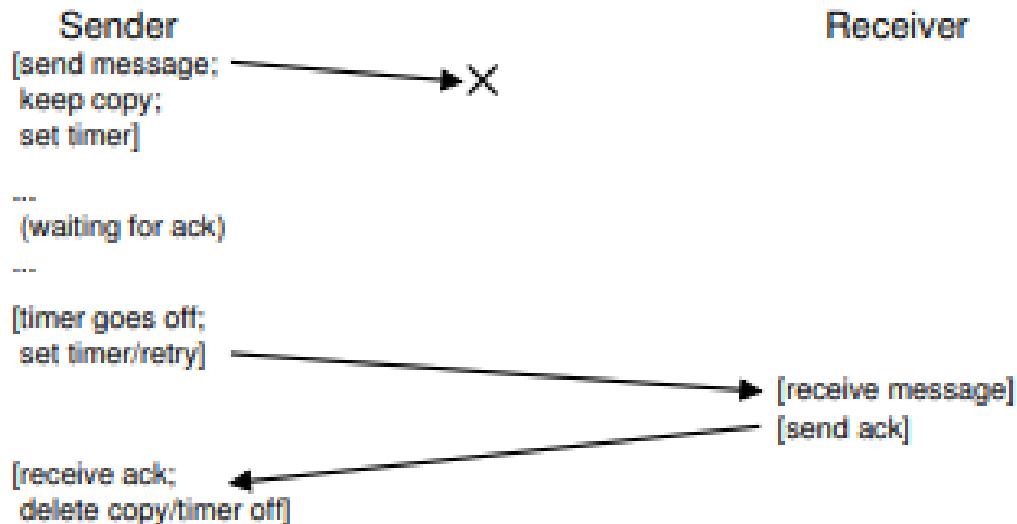


Figure 48.4: Message Plus Acknowledgment: Dropped Request

このケースを処理するには、タイムアウトと呼ばれる追加のメカニズムが必要です。送信者がメッセージを送信すると、送信者は一定時間後にタイマーをオフにするように設定するようになります。その時点で確認応答が受信されなかった場合、送信者はメッセージが失われたと判断します。次に、送信者は単に送信の再試行を行い今度は同じメッセージを再度送信し、この時間が経過することを期待します。このアプローチが機能するためには、送信者はメッセージをもう一度送信する必要がある場合に備えて、メッセージのコピーを保持する必要があります。タイムアウトと再試行の組み合わせにより、アプローチのタイムアウト/再試行と呼ぶ人もいました。かなり賢いネットワーキングタイプでしょう？違いますか？図 48.4 に例を示します。

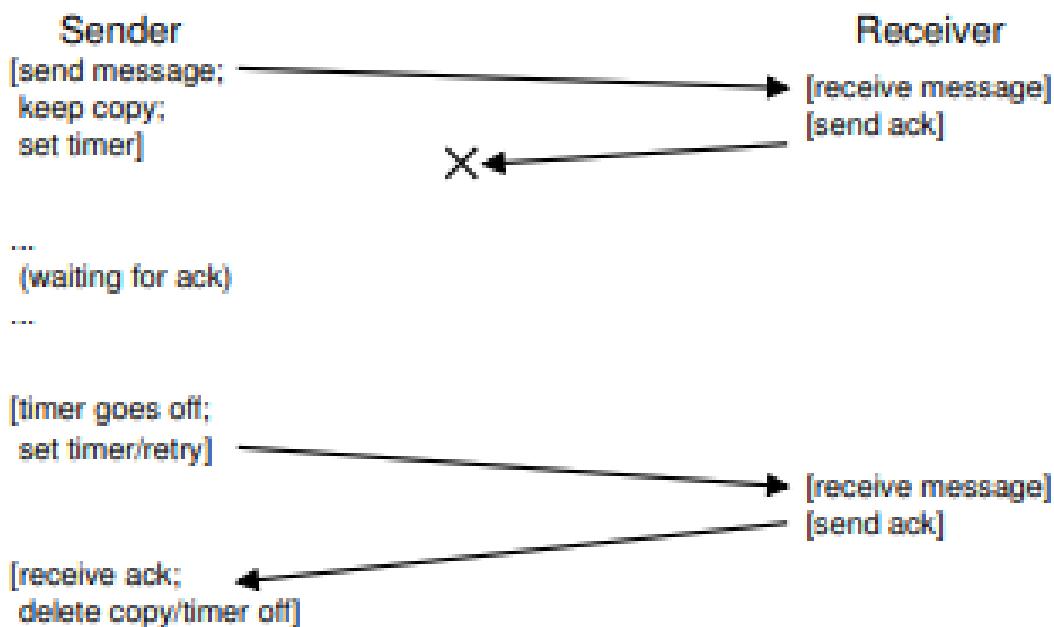


Figure 48.5: Message Plus Acknowledgment: Dropped Reply

残念ながら、この形式のタイムアウト/再試行では十分ではありません。図 48.5 にトラブルの原因となるパケットロスの例を示します。この例では、失われた元のメッセージではなく確認応答です。送信者の観点からは状況は同じように見えます。応答が受信されず、タイムアウトと再試行が順番に行われます。しかし受信機の観点からは、それは全く異なっています。今では同じメッセージが 2 回受信されました！これが OK ですか？

る場合もありますが、一般的にはそうではありません。ファイルをダウンロードしているときに何が起こり、余分なパケットがダウンロード内で繰り返されるか想像してください。したがって、信頼できるメッセージ層を目指しているときには、通常、各メッセージが受信者によって1回だけ受信されることを保証したいと考えています。

受信者が重複メッセージの送信を検出できるようにするには、送信者は各メッセージを独自の方法で識別しなければならず、受信者は以前に各メッセージを見たかどうかを追跡する必要があります。受信者が重複した送信を見た場合、単にメッセージを確認するだけですが、(重要なことに)データを受信するアプリケーションにメッセージを渡しません。したがって、送信者は ack を受信しますが、メッセージは2回受信されず、上記の1回のセマンティクスは維持されます。

重複したメッセージを検出する方法はたくさんあります。たとえば、送信者はメッセージごとに一意の ID を生成できます。受信者はそれまでに見たすべての ID を追跡することができます。このアプローチはうまくいくかもしれません、非常にコストがかかり、無制限のメモリですべての ID を追跡する必要があります。

メモリをほとんど必要としない簡単なアプローチではこの問題が解決され、メカニズムはシーケンスカウンタとして知られています。シーケンスカウンタでは、送信側と受信側は、各側が維持するカウンタの開始値(例えば、1)で合意します。メッセージが送信されるたびに、カウンタの現在の値がメッセージとともに送信されます。このカウンタ値(N)はメッセージの ID になります。メッセージが送信された後、送信側は値を(N + 1) インクリメントします。

受信者は、その送信者からの着信メッセージの ID の期待値としてそのカウンタ値を使用します。受信したメッセージ(N)の ID が受信者のカウンタ(N)と一致する場合、メッセージを確認してアプリケーションに渡します。この場合、受信者はこのメッセージが最初に受信されたと判断します。次に受信者はそのカウンタを(N + 1) インクリメントし、次のメッセージを待ちます。

ack が失われた場合、送信者はメッセージ N をタイムアウトして再送します。この時点では、受信者のカウンタはより高い(N + 1)ため、受信者は既にこのメッセージを受信したことを知ります。したがって、メッセージを確認しますが、アプリケーションに渡すことはありません。この簡単な方法で、シーケンスカウンタを使用して重複を避けることができます。

最も一般的に使用される信頼性の高い通信レイヤーは、TCP/IP と呼ばれます。TCP は、ネットワーク[VJ88] の輻輳を処理するための機械、複数の未解決の要求、その他数百の小さな微調整と最適化を含めて、上で説明したよりもはるかに洗練されています。あなたが好奇心が強いならそれについてぜひ読んでください。より良い方法として、ネットワーキングコースを受講することで、その教材をうまく学ぶことができます。

48.4 Communication Abstractions

基本的なメッセージング層が与えられたら、この章の次の質問でアプローチします。分散システムを構築するときに、通信の概念を使用する必要がありますか？

TIP: BE CAREFUL SETTING THE TIMEOUT VALUE

議論から推測できるように、タイムアウト値を正しく設定することは、タイムアウトを使用してメッセージ送信を再試行する重要な侧面です。タイムアウトが小さすぎると、送信者は不必要にメッセージを再送信し、送信者とネットワークのリソースに CPU 時間を浪費します。タイムアウトが大きすぎる場合、送信者は再送信に時間がかかり過ぎるため、送信者のパフォーマンスが低下します。したがって、单一のクライアントとサーバーの観点から見た「正しい」値は、パケットの損失を検出するのに十分な時間だけ待ちますが、それ以上は待機しません。しかし、分散システムには单一のクライアントとサーバー以上のものがあります。これについては後の章で説明します。多くのクライアントが单一のサーバーに送信するシナリオでは、サーバーでのパケット損失が、サーバーが過負荷になっていることを示す指標になる場合があります。もしそうであった場合、クライ

アントは異なる適応方法で再試行することができます。たとえば、最初のタイムアウトの後、クライアントはタイムアウト値をより高い量、おそらく元の値の2倍に増やす可能性があります。初期のアロハネットワークの先駆けで、初期イーサネットで採用されている[A70]、このような指数関数的なバックオフスキームは、過剰な再送信によってリソースが過負荷になる状況を回避します。ロバストシステムは、この性質の過負荷を避けるために努力しています。

システムコミュニティは、長年にわたり多くのアプローチを開発してきました。1つの作業でOSの抽象化が行われ、分散環境で動作するように拡張されました。例えば、Distributed Shared Memory(DSM)システムは、異なるマシン上のプロセスが大きな仮想アドレス空間[LH89]を共有することを可能にします。この抽象化によって、分散計算はマルチスレッドアプリケーションのようなものに変わります。唯一の違いは、これらのスレッドが同じマシン内の異なるプロセッサではなく、異なるマシン上で実行されることです。

ほとんどのDSMシステムの動作する方法は、OSの仮想メモリシステムを使用する方法です。あるマシンでページにアクセスすると、2つのことが起こります。最初の(最良の)ケースでは、ページは既にマシン上にローカルに存在しているため、データが迅速にフェッチされます。2番目のケースでは、ページは現在他のマシンにあります。ページフォルトが発生し、ページフォルトハンドラが他のマシンにメッセージを送信してページをフェッチし、要求プロセスのページテーブルにインストールして実行を続行します。

このアプローチは今日多くの理由で広く使用されていません。DSMの最大の問題は、障害の処理方法です。たとえば、マシンが故障した場合などを想像してみてください。そのマシンのページはどうなりますか？分散計算のデータ構造がアドレス空間全体に広がっている場合はどうでしょうか？この場合、これらのデータ構造の一部は突然使用できなくなります。あなたのアドレス空間の一部が失われたときの失敗を扱うことは困難です。「次の」ポインタがそのアドレス空間の一部分を指し示すlinked listを想像してください。とんでもない！

さらなる問題はパフォーマンスです。通常、コードを書くとき、メモリへのアクセスは安いと仮定します。DSMシステムでは、一部のアクセスは安価ですが、他のものはページフォルトやリモートマシンからの高価なフェッチを引き起こします。したがって、このようなDSMシステムのプログラマは、ほとんど通信が全く起こらず、そのようなアプローチのポイントの多くを解決するような計算を構成することにかなり注意する必要がありました。この分野では多くの研究が行われましたが、実用的な影響はほとんどありませんでした。今日では、誰もDSMを使用して信頼性の高い分散システムを構築していません。

48.5 Remote Procedure Call (RPC)

OSの抽象化は分散システムを構築するうえで貧弱な選択であることが判明しましたが、プログラミング言語(PL)抽象化ははるかに理にかなっています。最も支配的な抽象化は、Remote Procedure Callまたは略してRPCのアイデア[BN84]に基づいています。

リモートプロシージャコールパッケージはすべて、単純な目的を持っています。つまり、リモートマシン上でコードを実行するプロセスをローカル関数を呼び出すのと同じくらい簡単にすることです。したがって、クライアントに対してプロシージャ呼び出しが行われ、しばらくしてから結果が戻されます。サーバーは、エクスポートしたいルーチンを単に定義します。残りの魔法はRPCシステムによって処理され、一般的に二つあります。1つ目はRPCシステムには一般にスタブジェネレータ(プロトコルコンパイラーと呼ばれることもあります)、二つ目は、実行時ライブラリがあります。ここでこれらの各部分をより詳しく見ていきます。

Stub Generator

スタブジェネレータの仕事は簡単です。関数の引数と結果をメッセージの中に入れ、パッキングする際の苦痛の一部を自動化して取り除くことです。数多くの利点が生まれます。設計上、手作業でそのようなコードを書く際に起こる単純な間違いを避けることができます。さらに、スタブコンパイラーは、おそらくそのような

コードを最適化して、パフォーマンスを向上させることができます。このようなコンパイラへの入力は、単にサーバーがクライアントにエクスポートする呼び出しのセットです。概念的には、これは次のような単純なものです。

```
interface {
    int func1(int arg1);
    int func2(int arg1, int arg2);
};
```

スタブジェネレータはこのようなインターフェースをとり、いくつかの異なるコードを生成します。クライアントの場合、クライアントスタブが生成されます。クライアントスタブは、インターフェースで指定された各関数を含みます。このRPCサービスを使用したいクライアントプログラムはこのクライアントスタブにリンクし、RPCを作成するためにそのクライアントスタブを呼び出します。

内部的には、クライアントスタブのこれらの各機能は、リモートプロシージャコールを実行するために必要なすべての作業を行います。クライアントに対して、コードは単に関数呼び出しとして現れます(例えば、クライアントは `func1(x)` を呼び出す)。内部的には、`func1()` のクライアントスタブ内のコードはこれを行います:

- メッセージバッファを作成します。メッセージバッファは通常、あるサイズのバイトの連続した配列です。

- 必要な情報をメッセージバッファにパックする。この情報には、関数が必要とするすべての引数(例えば、上の例では `func1` の整数)など、呼び出される関数の識別子が含まれています。この情報をすべて1つの連続したバッファに入れるプロセスは、the marshaling of arguments または the serialization of the message と呼ばれることがあります。
- 宛先 RPC サーバーにメッセージを送信します。RPC サーバーとの通信、および RPC サーバーが正しく動作するために必要なすべての詳細は、RPC ランタイムライブラリによって処理されます(後述)。
- 返事を待ちます。関数呼び出しが通常同期的であるため、呼び出しがその完了を待ちます。
- リターンコードと他の引数をアンパックします。関数が单一の戻りコードを返すだけの場合、このプロセスは簡単です。しかし、より複雑な関数はより複雑な結果(例えばリスト)を返す可能性があり、したがってスタブはそれらをアンパックする必要があります。この手順は、unmarshaling または deserialization とも呼ばれます。
- 発信者に戻ります。最後に、クライアント・スタブからクライアント・コードに戻ります。

サーバーの場合は、コードも生成されます。サーバーで実行される手順は次のとおりです。- メッセージを解凍します。unmarshaling または deserialization と呼ばれるこの手順は、着信メッセージから情報を取り出します。関数の識別子と引数が抽出されます。

- 実際の関数を呼び出します。最後に! リモート関数が実際に実行されるところに達しました。RPC ランタイムは ID で指定された関数を呼び出し、目的の引数を渡します。
- 結果をパッケージ化する。戻り引数は、单一の応答バッファにマーシャリングされます。
- 収信を送信します。最後に返信が発信者に送信されます。

スタブコンパイラで考慮すべき他の重要な問題がいくつかあります。最初のものは複雑な引数です。つまり、複雑なデータ構造をどのようにパッケージ化して送信しますか? たとえば、`write()` システムコールを呼び出すときには、整数ファイル記ディスクリプタ、バッファへのポインタ、書き込まれるバイト数(ポインターで始まる)を示すサイズの3つの引数が渡されます。RPCパッケージにポインタが渡された場合、そのポインターをどのように解釈するのかを把握し、正しい動作を実行できる必要があります。通常これはよく知られていない

るタイプ (RPC コンパイラが理解できるサイズのデータをチャunk を渡すために使用されるバッファ t など) や、データ構造に詳細情報を付けることによって、コンパイラがどのバイトをシリアル化する必要があるか知ることが可能です。

別の重要な問題は、並行性に関するサーバーの構成です。シンプルなサーバーは単純なループで要求を待機し、各要求を 1 つずつ処理します。しかし、あなたが推測したように、これは非常に非効率的である可能性があります。1 つの RPC コールを (例えば、I/O 上で) ブロックすると、サーバリソースが浪費されます。したがって、ほとんどのサーバーは、何らかの並行方式で構築されます。一般的な構成はスレッドプールです。この構成では、サーバーの起動時に有限のスレッドセットが作成されます。メッセージが到着すると、これらのワーカースレッドの 1 つにディスパッチされ、RPC 呼び出しの処理が行われ、最終的に応答します。この間、メインスレッドは他の要求を受信し続け、おそらくそれらを他のワーカーにディスパッチします。このような構成により、サーバー内での同時実行が可能になり、その使用率が向上します。RPC 呼び出しが正しい動作を保証するためにロックおよび他の同期プリミティブを使用する必要があるため、プログラミングがかなり複雑であり、標準コストも同様に発生します。

Run-Time Library

ランタイムライブラリは、RPC システムでの重い作業の多くを処理します。ほとんどのパフォーマンスと信頼性の問題がここで処理されます。このようなランタイムレイヤを構築する際の主要な課題のいくつかについて説明します。

我々が克服しなければならない最初の課題の 1 つは、リモートサービスを見つける方法です。命名のこの問題は分散システムにおける共通の問題であり、ある意味では現在の議論の範囲を超えていません。最も簡単なアプローチは、既存のネーミングシステム (例えば、現在のインターネットプロトコルによって提供されるホスト名およびポート番号) 上に構築されます。このようなシステムでは、クライアントは、使用しているポート番号だけでなく、目的の RPC サービスを実行しているマシンのホスト名または IP アドレスを知っていなければなりません (ポート番号は、複数の通信チャネルを一度に許可する)。プロトコルスイートは、システム内の他のマシンからパケットを特定のアドレスにルーティングするメカニズムを提供する必要があります。命名についての良い議論のためには、インターネット上の DNS と名前解決について読むか、Saltzer と Kaashoek の本 [SK09] の優秀な章を読んだほうが良いでしょう。

クライアントが特定のリモートサービスのためにどのサーバーと通信するべきかを知ったら、次の質問はどのトранスポートレベルのプロトコルが RPC を構築すべきかということです。具体的には、RPC システムが TCP/IP などの信頼性の高いプロトコルを使用するか、UDP/IP などの信頼性の低い通信レイヤー上に構築する必要がありますか？

純粋にその選択は容易です。明らかに、リモートサーバーに確実にリクエストを送付したいと願っており、確実に回答を受け取ることを希望しています。したがって、TCP などの信頼性の高いトランスポートプロトコルを選択する必要があります。

残念ながら、信頼できる通信層の上に RPC を構築すると、パフォーマンスが大幅に低下する可能性があります。上記の議論から、信頼性の高い通信レイヤーがどのように機能するかを思い出してください。肯定応答とタイムアウト/リトライです。したがって、クライアントが RPC 要求をサーバーに送信すると、サーバーは応答を返して、呼び出し元が要求を受信したことを認識します。同様に、サーバーがクライアントに応答を送信すると、クライアントは受信したことをサーバーが認識するように応答します。信頼できる通信レイヤの上に要求/応答プロトコル (RPC など) を構築することによって、2 つの「余分な」メッセージが送信されます。

このため、多くの RPC パッケージは、UDP などの信頼性の低い通信レイヤーの上に構築されています。これにより、より効率的な RPC レイヤーが可能になりますが、RPC システムに信頼性を提供する責任が追加されます。RPC 層は、前述のように、タイムアウト/リトライと確認応答を使用して、望ましいレベルの責任を達成します。何らかの形式のシーケンス番号付けを使用することによって、通信層は、各 RPC が正確に 1 回

(障害がない場合)、または最大で1回(障害が発生した場合)発生することを保証できます。

Other Issues

RPC ランタイムにも同様に処理する必要のある問題がいくつかあります。たとえば、リモートコールの完了に時間がかかる場合はどうなりますか？私たちのタイムアウト機構が与えられると、長時間実行される遠隔呼び出しは、クライアントに失敗として現れる可能性があり、したがって再試行を引き起こし、したがってここでは注意が必要です。1つの解決方法は、応答がすぐに生成されない場合に明示的な確認応答(受信者から送信者へ)を使用することです。これにより、クライアントは要求を受信したサーバーを知ることができます。その後、しばらくしてから、クライアントは定期的にサーバーがリクエストの作業しているかどうかを尋ねることができます。サーバが“yes”と言っている場合、クライアントは待っていなければなりません(結局のところ、手続き呼び出しが実行を完了するのに時間がかかることがあります)。

ランタイムには、単一のパケットに収まるものよりも大きな引数を持つプロシージャ・コールも処理する必要があります。いくつかの下位レベルのネットワークプロトコルは、送信側の断片化(より大きなパケットをより小さいパケットのセットにする)と受信側の再アセンブリ(小さな部分をより大きな論理全体に分解する)を提供します。そうでなければ、RPC ランタイムはそのような機能自体を実装しなければならないかもしれません。詳細は、Birrell と Nelson の優れた RPC ペーパーを参照してください [BN84]。

多くのシステムが扱う1つの問題は、バイトオーダーの問題です。ご存じのように、値を格納するのにビッグエンディアンのオーダー、またはリトルエンディアンのオーダーを使用されるマシンもあります。ビッグエンディアンは、アラビア数字のように、最上位ビットから最下位ビットまでのバイト(整数)を格納します。リトルエンディアンはその逆です。どちらも数値情報を保存するのにも同様に有効です。ここで問題は、異なるエンディアンのマシン間でのやりとり方法です。

Aside: The End-to-End Argument

エンド・ツー・エンドの引数は、システム内の最高レベル、すなわち通常「終わり」のアプリケーションが最終的に、特定の機能を真に実装できる階層化されたシステム内の唯一の場所である場合をもたらします。彼らの画期的な論文 [SRC84] では、優れた例として、2つのマシン間での信頼性の高いファイル転送が挙げられます。もし、マシン A からマシン B にファイルを転送したいとき、B 上で終了するバイトが A で開始したバイトとまったく同じであることを確認するには、この「エンドツーエンド」チェックが必要です。ネットワークまたはディスクなどの低レベルの信頼できる機械は、そのような保証を提供しません。

コントラストは、信頼性の高いファイル転送の問題を、システムの下位レイヤに信頼性を追加することによって解決しようとするアプローチです。たとえば、信頼性の高い通信プロトコルを構築し、それを使用して信頼性の高いファイル転送を構築するとします。通信プロトコルは、タイムアウト/リトライ、肯定応答、およびシーケンス番号を使用して、送信者によって送信された各バイトが受信者によって順番に受信されることを保証します。残念なことに、このようなプロトコルを使用しても信頼性の高いファイル転送はできません。通信が行われる前に送信者のメモリでバイトが壊れているとか、受信者がデータをディスクに書き込んだときに何か悪いことが起きたとします。そのような場合、たとえバイトがネットワーク上で確実に配信されたとしても、私たちのファイル転送は最終的には信頼できません。信頼性の高いファイル転送を構築するには、エンドツーエンドチェックの信頼性が必要です。具体的には、転送の完了後、受信者ディスクのファイルを読み戻し、チェックサムを計算し、そのチェックサムと送信者のファイルと比べる必要があります。

この必然的なことは、下位層が余分な機能を提供することによって、実際にシステムの性能を向上させることができ、あるいはシステムを最適化できることである。したがって、システム内の低レベルでこのような機械を使用することを排除すべきではありません。むしろ、全体的なシステムや

アプリケーションで最終的に使用される場合、そのような機械の有用性を慎重に検討する必要があります。

RPC パッケージは、メッセージフォーマット内で明確なエンディアンを提供することによって、これを処理します。Sun の RPC パッケージでは、XDR(eXternal Data Representation) レイヤーがこの機能を提供します。メッセージを送信または受信するマシンが XDR のエンディアンと一致する場合、メッセージは単に期待どおりに送受信されます。しかし、マシンの通信がエンディアンが異なる場合、メッセージ内の各情報を変換する必要があります。したがって、エンディアンの差はパフォーマンスコストを小さくすることができます。

最後の問題は、通信の非同期性をクライアントに公開するかどうかであるため、パフォーマンスの最適化が可能になります。具体的には、典型的な RPC は同期的に行われます。すなわち、クライアントがプロシージャコールを発行するとき、プロシージャコールが戻るのを待ってから続行する必要があります。この待ち時間は長くなる可能性があり、クライアントが他の作業を行う可能性があるため、RPC パッケージによっては RPC を非同期に呼び出すことができます。非同期 RPC が発行されると、RPC パッケージは要求を送信してすぐに戻ります。クライアントは他の RPC やその他の有用な計算を呼び出すなど、自由に他の作業を行うことができます。クライアントはある時点での非同期 RPC の結果を見たいと思うでしょう。RPC 層にコードバックして、未処理の RPC が完了するのを待つように指示します。この時点で戻り引数にアクセスできます。

48.6 Summary

私たちは、新しいトピック、分散システム、そしてその主要な問題の導入を見てきました。これは、現在は一般的なイベントである障害をどのように処理するかです。彼らが Google の中で言うように、デスクトップマシンだけでは障害はまれです。何千ものマシンを持つデータセンターにいると、常に障害が発生しています。どの分散システムの鍵も、その失敗をどのように処理するかです。

通信は、分散システムの核心を形成することもわかりました。その通信の一般的な抽象化は、クライアントがサーバー上でリモート呼び出しを行うことを可能にするリモートプロシージャコール (RPC) があります。RPC パッケージは、ローカルプロシージャコールを厳密に反映したサービスを提供するために、タイムアウト/リトライと確認応答を含む詳細情報のすべてを処理します。

RPC パッケージを実際に理解する最良の方法は、もちろん自分で使用することです。Sun の RPC システムは、スタブコンパイラ rpcgen を使用して古いものです。Google の gRPC と Apache Thrift は、現代的なものと同じです。1 つを試し、すべての気に病むことが何であるかを見てください。

参考文献

- [A70] “The ALOHA System — Another Alternative for Computer Communications”
Norman Abramson
The 1970 Fall Joint Computer Conference
The ALOHA network pioneered some basic concepts in networking, including exponential back-off and retransmit, which formed the basis for communication in shared-bus Ethernet networks for years.
- [BN84] “Implementing Remote Procedure Calls”
Andrew D. Birrell, Bruce Jay Nelson
ACM TOCS, Volume 2:1, February 1984
The foundational RPC system upon which all others build. Yes, another pioneering effort from our friends at Xerox PARC.
- [MK09] “The Effectiveness of Checksums for Embedded Control Networks”
Theresa C. Maxino and Philip J. Koopman

IEEE Transactions on Dependable and Secure Computing, 6:1, January '09

A nice overview of basic checksum machinery and some performance and robustness comparisons between them.

[LH89] “Memory Coherence in Shared Virtual Memory Systems”

Kai Li and Paul Hudak

ACM TOCS, 7:4, November 1989

The introduction of software-based shared memory via virtual memory. An intriguing idea for sure, but not a lasting or good one in the end.

[SK09] “Principles of Computer System Design”

Jerome H. Saltzer and M. Frans Kaashoek

Morgan-Kaufmann, 2009

An excellent book on systems, and a must for every bookshelf. One of the few terrific discussions on naming we've seen.

[SRC84] “End-To-End Arguments in System Design”

Jerome H. Saltzer, David P. Reed, David D. Clark

ACM TOCS, 2:4, November 1984

A beautiful discussion of layering, abstraction, and where functionality must ultimately reside in computer systems.

[VJ88] “Congestion Avoidance and Control”

Van Jacobson

SIGCOMM '88

A pioneering paper on how clients should adjust to perceived network congestion; definitely one of the key pieces of technology underlying the Internet, and a must read for anyone serious about systems, and for Van Jacobson's relatives because well relatives should read all of your papers.

49 Sun's Network File System (NFS)

分散クライアント/サーバーコンピューティングの最初の用途の1つは、分散ファイルシステムの分野でした。このような環境では、多数のクライアントマシンと1つのサーバー（またはいくつか）が存在します。サーバはそのデータをディスクに保存し、クライアントは整形式プロトコルメッセージを通じてデータを要求します。図49.1に基本設定を示します。

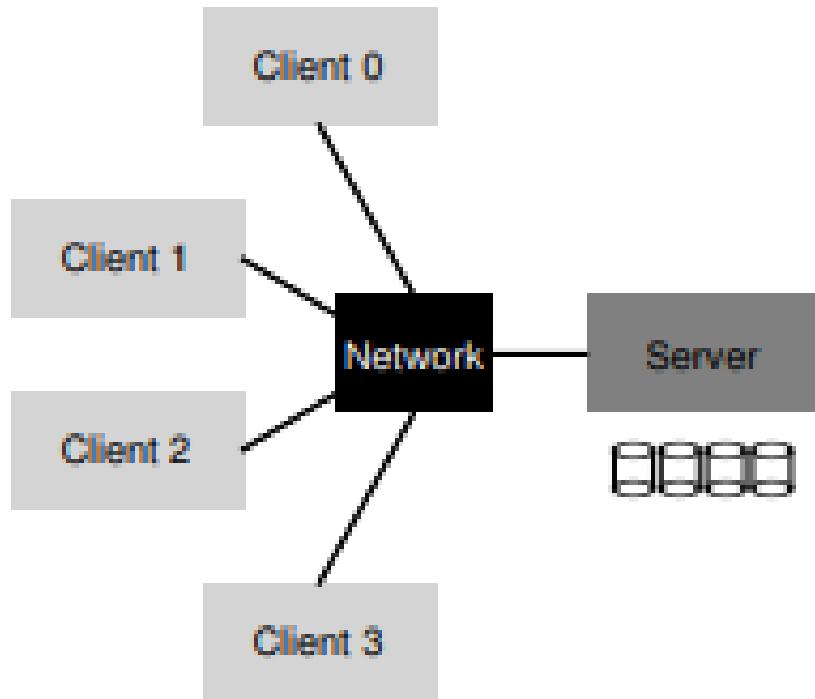


Figure 49.1: A Generic Client/Server System

画像からわかるように、サーバーにはディスクがあり、クライアントはネットワーク上のメッセージを送信して、それらのディスク上のディレクトリとファイルにアクセスします。なぜこの取り決めに迷惑をかけるのですか？（つまり、クライアントにローカルディスクを使用させるのはどうですか？）主に、この設定により、クライアント間でデータを簡単に共有できます。したがって、あるマシン（クライアント0）のファイルにアクセスしてから別のマシン（クライアント2）を使用すると、ファイルシステムのビューは同じになります。あなたのデータは、これらの異なるマシン間で自然に共有されます。二次的な利点は、集中管理です。たとえば、多数のクライアントからではなく、少数のサーバーマシンからファイルをバックアップすることができます。もう1つの利点はセキュリティです。ロックされたマシンルーム内のすべてのサーバーを使用すると、特定の種類の問題が発生するのを防ぐことができます。

CRUX: HOW TO BUILD A DISTRIBUTED FILE SYSTEM

どのように分散ファイルシステムを構築しますか？ 考えるべき重要な側面は何ですか？ 何が悪くなるのは簡単ですか？ 既存のシステムから何を学ぶことができますか？

49.1 A Basic Distributed File System

ここでは、単純化された分散ファイルシステムのアーキテクチャについて検討します。シンプルなクライアント/サーバ分散ファイルシステムには、これまで検討してきたファイルシステムよりも多くのコンポーネントがあります。クライアント側には、クライアント側のファイルシステムを通じてファイルとディレクトリにアクセスするクライアントアプリケーションがあります。クライアントアプリケーションは、サーバに格納されているファイルにアクセスするために、クライアント側ファイルシステムにたいしてシステムコール(`open()`、`read()`、`write()`、`close()`、`mkdir()`など)を発行します。したがって、クライアントアプリケーションにとって、ファイルシステムは、おそらくパフォーマンスを除いて、ローカル(ディスクベース)ファイルシステムと異なるものではないようです。このように、分散ファイルシステムはファイルへの透過的なアクセスを提供します。結局のところ、誰が異なる API セットを必要とするファイルシステムを使用したいのか、また、そうでなければ使用する苦痛があつたのでしょうか？

クライアントサイドファイルシステムの役割は、これらのシステムコールを処理するために必要なアクションを実行することです。たとえば、クライアントが `read()` 要求を発行した場合、クライアント側のファイルシステムは、特定のブロックを読み取るために、サーバー側のファイルシステム(またはファイルサーバーと呼ばれる)にメッセージを送信することができます。ファイルサーバはディスク(またはそれ自身のインメモリキャッシュ)からブロックを読み込み、要求されたデータのメッセージをクライアントに返します。クライアント側のファイルシステムは、`read()` システムコールに提供されたユーザバッファにデータをコピーします。これで要求は完了します。クライアント上の同じブロックの後続の `read()` は、クライアントメモリまたはクライアントのディスクにもキャッシュされることに注意してください。そのような場合には、ネットワークトラフィックを生成する必要はありません。

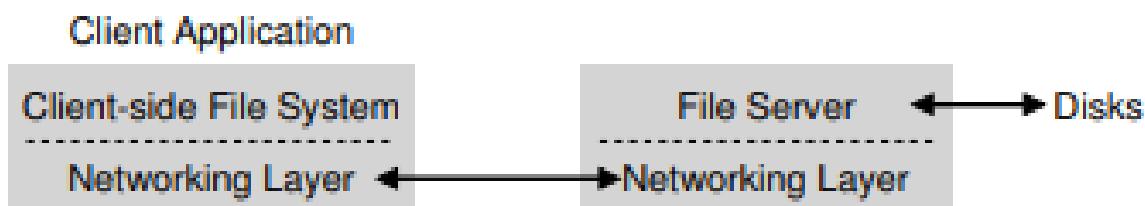


Figure 49.2: Distributed File System Architecture

この簡単な概要から、クライアント/サーバ分散ファイルシステムには、クライアント側のファイルシステムとファイルサーバーという 2 つの重要なソフトウェアが存在することがわかります。一緒に動作することによって、分散ファイルシステムの動作が決まります。今では、特定のシステムの Sun の Network File System(NFS) を検討する必要があります。

ASIDE: WHY SERVERS CRASH

NFSv2 プロトコルの詳細に入る前に、なぜサーバがクラッシュするのでしょうか？ まあ、あなたが推測するかもしれないが、多くの理由があります。サーバは単に停電に悩まされることがあります(一時的に)。電源が復旧した場合にのみ、マシンを再起動することができます。サーバーは、よく数十万行または数百万行のコードで構成されています。したがって、彼らにはバグがあります(良いソフトウェアでも 100~1000 行のコードあたりいくつかのバグがあります)ので、最終的にバグが発生し、クラッシュする可能性があります。また、メモリリークもあります。メモリリークがわずかであっても、システムのメモリ不足やクラッシュが発生します。最後に、分散システムでは、クライアントとサーバーの間にネットワークがあります。ネットワークが変な動作している場

合(たとえば、パーティション化され、クライアントとサーバーは動作しているが通信できない場合など)、リモートマシンがクラッシュしたように見えるかもしれません、実際にはネットワーク経由で到達できないのです。

49.2 On To NFS

最も初期で成功した分散システムの1つがSun Microsystemsによって開発され、Sun Network File System(またはNFS)として知られています[S86]。Sunは独自のクローズドシステムを構築する代わりに、クライアントとサーバーが通信するために使用する正確なメッセージフォーマットを指定するオープンプロトコルを開発しました。異なるグループが独自のNFSサーバーを開発し、相互運用性を維持しながらNFS市場で競合する可能性があります。現在、NFSサーバー(Oracle/Sun、NetApp[HLM94]、EMC、IBMなど)を販売する企業が数多くあり、NFSの普及はこの「公開市場」アプローチに起因する可能性が高いです。

49.3 Focus: Simple and Fast Server Crash Recovery

この章では、古典的なNFSプロトコル(バージョン2、a.k.a. NFSv2)について説明します。これは長年の標準です。NFSv3への移行では小さな変更が加えられ、NFSv4への移行ではより大きなプロトコル変更が行われました。しかし、NFSv2はすばらしくかつ不満足である、この両方の焦点は役立ちます。

NFSv2では、プロトコルの設計における主な目標は、シンプルで高速なサーバークラッシュリカバリでした。マルチクライアント、単一サーバー環境ではこの目標は大きな意味を持ちます。サーバーがダウンしている(または利用できない)場合、すべてのクライアントマシン(およびそのユーザー)は不幸で非生産的になります。したがって、サーバーが進むにつれて、システム全体も同様になります。

49.4 Key To Fast Crash Recovery: Statelessness

この単純な目標は、NFSv2ではstateless protocolと呼ばれるものを設計することによって実現されています。サーバーは、設計上、各クライアントで何が起こっているかについては把握していません。たとえば、どのクライアントがどのブロックをキャッシュしているか、または各クライアントで現在開いているファイル、ファイルの現在のファイルポインタの位置などは、サーバーが認識しません。単純に言えば、サーバーはクライアントが何をやっているかを知らないでよいのです。むしろ、プロトコルは、要求を完了するために必要なすべての情報を各プロトコル要求で配信するように設計されています。それが今ではない場合、このステートレスなアプローチは、我々がプロトコルを以下でより詳細に議論するにつれてより意味を持っていくでしょう。

ステートフル(ステートレスではない)プロトコルの例については、`open()`システムコールを考慮してください。パス名を指定すると、`open()`はファイルディスクリプタ(整数)を返します。このディスクリプタは、このアプリケーションコードのように、後続の`read()`または`write()`要求でさまざまなファイルブロックにアクセスするために使用されます(システムコールの適切なエラーチェックはスペース上の理由から省略されています)。

```

char buffer[MAX];
int fd = open("foo", O_RDONLY); // get descriptor "fd"
read(fd, buffer, MAX);        // read MAX bytes from foo (via fd)
read(fd, buffer, MAX);        // read MAX bytes from foo
...
read(fd, buffer, MAX);        // read MAX bytes from foo
close(fd);                   // close file

```

Figure 49.3: Client Code: Reading From A File

クライアント側のファイルシステムが、「ファイル'foo'を開いてディスクリプタを返す」というプロトコルメッセージをサーバーに送信することによってファイルを開くとします。ファイルサーバーはファイルをローカルに開き、ディスクリプタをクライアントに返します。その後の読み込みでは、クライアントアプリケーションはそのディスクリプタを使用して `read()` システムコールを呼び出します。クライアント側のファイルシステムは、ファイル内のディスクリプタをファイルサーバーに渡し、「ディスクリプタによって参照されるファイルからいくつかのバイトを読み取って、ここに渡します」と言っています。

この例では、ファイルディスクリプタはクライアントとサーバーの間の共有状態の一部です (Ousterhout は分散状態 [O91] と呼んでいます)。私たちが上記のように共有状態はクラッシュリカバリを複雑にします。最初の読み取りが完了した後で、クライアントが 2 番目の読み取りを発行する前に、サーバーがクラッシュしたとします。サーバーが起動して再び実行されると、クライアントは次に 2 回目の読み取りを発行します。残念ながら、サーバは `fd` がどのファイルを参照しているのか全く分かりません。その情報は一時的 (すなわち、メモリ内) であり、したがってサーバがクラッシュしたときに失われました。この状況に対処するために、クライアントとサーバーは何らかのリカバリープロトコルに従わなければなりません。つまり、クライアントは、サーバーが知る必要のある情報をサーバーに伝えるために十分な情報をメモリに保持しないといけません (この場合そのファイルディスクリプタ `fd` はファイル `foo` を参照します)。

ステートフルなサーバーがクライアントのクラッシュに対処しなければならないという事実を考えると、さらに悪化します。たとえば、ファイルを開いてクラッシュするクライアントを想像してみてください。`open()` はサーバー上のファイルディスクリプタを使います。サーバーはどのようにしてファイルを閉じることができますか？通常の操作では、クライアントは最終的に `close()` を呼び出して、ファイルを閉じるべきであることをサーバーに通知します。しかし、クライアントがクラッシュすると、サーバは `close()` を受け取ることはできません。そのため、ファイルを閉じるためにクライアントがクラッシュしたこと気に気づく必要があります。

これらの理由から、NFS の設計者はステートレスなアプローチを追求することに決めました。各クライアント操作には、要求を完了するために必要なすべての情報が含まれています。派手なクラッシュリカバリは不要です。サーバーはただちに再起動し、クライアントは最悪の場合、要求を再試行する必要があります。

49.5 The NFSv2 Protocol

したがって、NFSv2 プロトコル定義に到達します。私たちの問題は簡単です。

THE CRUX: HOW TO DEFINE A STATELESS FILE PROTOCOL

ステートレスな操作を可能にするためにネットワークプロトコルをどのように定義できますか？明らかに、`open()` のようなステートフルな呼び出しは（サーバーが開いているファイルを追跡するために）議論の一部になることはできません。ただし、クライアントアプリケーションは、`open()`、`read()`、`write()`、`close()` などの標準 API call を呼び出して、ファイルやディレクトリにアクセスする必要があります。したがって、洗練された質問として、どのようにプロトコルが POSIX ファイルシステム API をステートレスにサポートするかを定義することができますか？

NFS プロトコルの設計を理解するための鍵は、ファイルハンドルを理解することです。ファイルハンドルは、特定の操作で操作されるファイルまたはディレクトリを一意に記述するために使用されます。したがって、多くのプロトコル要求にはファイルハンドルが含まれています。

ファイルハンドルには、ボリューム識別子、i ノード番号、世代番号という 3 つの重要な要素があると考えることができます。これらの 3 つの項目は、クライアントがアクセスしたいファイルまたはディレクトリを一意の識別子で構成します。ボリューム識別子は、要求が参照するファイルシステムをサーバーに通知します (NFS サーバーは複数のファイルシステムをエクスポートできます)。inode 番号は、要求がそのパーティション内のどのファイルにアクセスしているかをサーバーに伝えます。最後に、i ノード番号を再利用する場合は世代番号が必要です。inode 番号が再利用されるたびにインクリメントすることで、古いファイルハンドルを持つクライアントが、新しく割り当てられたファイルに誤ってアクセスすることがないようにします。

ここでは、プロトコルの重要な部分の概要を示します。完全なプロトコルは他の場所でも利用可能です (NFS [C00] の詳細な概要については、Callaghan の本を参照してください)。

```

NFSPROC_GETATTR
    expects: file handle
    returns: attributes
NFSPROC_SETATTR
    expects: file handle, attributes
    returns: nothing
NFSPROC_LOOKUP
    expects: directory file handle, name of file/directory to look up
    returns: file handle
NFSPROC_READ
    expects: file handle, offset, count
    returns: data, attributes
NFSPROC_WRITE
    expects: file handle, offset, count, data
    returns: attributes
NFSPROC_CREATE
    expects: directory file handle, name of file, attributes
    returns: nothing
NFSPROC_REMOVE
    expects: directory file handle, name of file to be removed
    returns: nothing
NFSPROC_MKDIR
    expects: directory file handle, name of directory, attributes
    returns: file handle
NFSPROC_RMDIR
    expects: directory file handle, name of directory to be removed
    returns: nothing
NFSPROC_READDIR
    expects: directory handle, count of bytes to read, cookie
    returns: directory entries, cookie (to get more entries)

```

Figure 49.4: The NFS Protocol: Examples

ここでは、プロトコルの重要な要素について簡単に説明します。最初に、LOOKUP プロトコルメッセージを使用してファイルハンドルを取得し、その後ファイルデータにアクセスするために使用します。クライアントは、ディレクトリファイルのハンドルと検索するファイルの名前を渡し、そのファイル (またはディレクトリ) のハンドルに加えて、その属性もサーバーからクライアントに返されます。

たとえば、クライアントがファイルシステム (/) のルートディレクトリのディレクトリファイルハンドルをすでに持っているとします (実際には、NFS マウントプロトコルで取得されますが、これはクライアント

とサーバーが最初に接続された方法です。簡潔さのためにマウントプロトコルをここで議論しません)。クライアント側で実行中のアプリケーションが /foo.txt ファイルを開くと、クライアント側のファイルシステムはルックアップ要求をサーバーに送信し、ルートファイルハンドルと名前 foo.txt を渡します。成功すると、foo.txt のファイルハンドル (および属性) が返されます。

不思議に思われる場合は、属性は、ファイル作成時間、最終変更時刻、サイズ、所有権とアクセス許可情報などのフィールドといったファイルシステムが各ファイルについて追跡するメタデータだけです。つまり同じタイプの情報を含む、ファイルに対して `stat()` を呼び出すと元に戻ります。

ファイルハンドルが利用可能になると、クライアントはファイルに対して READ および WRITE プロトコルメッセージを発行してファイルを読み書きすることができます。READ プロトコルメッセージは、プロトコルが、ファイル内のオフセットと、読み込むバイト数をファイルのファイルハンドルに沿って渡すことを要求します。次に、サーバーは読み取りを発行することができます (結局のところ、ハンドルはサーバーにどのボリュームとどの i ノードから読み取るべきかを通知し、オフセットとカウントは読み取るファイルのバイトを指示します)。クライアントにデータを返します (または障害が発生した場合はエラー)。WRITE は、データがクライアントからサーバーに渡され、成功コードのみが返される点を除いて、同様に処理されます。

最後の 1 つの興味深いプロトコルメッセージは、GETATTR 要求です。ファイルハンドルがあれば、ファイルの最後の変更時刻を含めて、そのファイルの属性を取り出せます。私たちがキャッシュを議論するときに、なぜこのプロトコル要求が NFSv2 で重要であるのかを見ていきます (なぜそうなっているか推測できますか？)。

49.6 From Protocol to Distributed File System

うまくいけば、このプロトコルがクライアント側のファイルシステムとファイルサーバを介してファイルシステムにどのように変換されているのかが分かるはずです。クライアント側のファイルシステムは開いているファイルを追跡し、一般にアプリケーション要求を関連する一連のプロトコルメッセージに変換します。サーバーは単にプロトコルメッセージに応答します。プロトコルメッセージには、要求を完了するために必要なすべての情報が含まれています。

たとえば、ファイルを読み取る単純なアプリケーションを考えてみましょう。図 (図 49.5) では、アプリケーションがどのようなシステムコールを行うか、そしてクライアント側のファイルシステムとファイルサーバがそのような呼び出しに応答する際の動作を示します。

図においてのいくつかのコメントがあります。まず、NFS ファイルハンドルへの整数ファイルディスクリプタである、現在のファイルポインタのマッピングなど、クライアントがファイルアクセスのすべての関連状態をどのように追跡しているかに注目してください。これにより、クライアントは、各読み取り要求 (もしかしたら、あなたは明示的に読み取るオフセットを指定していないことに気づいたかもしれません) を、正しくフォーマットされた読み取りプロトコルメッセージに変換して、ファイルから読み取るバイトを正確にサーバーに通知します。読み込みが成功すると、クライアントは現在のファイルの位置を更新します。後続の読み取りは同じファイルハンドルで発行されますが、異なるオフセットが発行されます。

次に、サーバーとのやりとりがどこに発生しているかがわかります。最初にファイルを開くと、クライアント側のファイルシステムは LOOKUP 要求メッセージを送信します。実際には長いパス名を渡す必要がある場合 (例: /home/remzi/foo.txt)、クライアントは 3 つのルックアップを送信します。1 つはディレクトリ / のホームをルックアップし、もう 1 つは remzi のホームをルックアップし、最後に remzi の foo.txt をルックアップします。

第 3 に、各サーバー要求が、要求全体を完了するために必要なすべての情報をどのように持っているかに気付くことがあります。この設計ポイントは、サーバーの障害から正常に回復できるようにするために重要です。ここで詳しく議論していきます。サーバーは要求に応答できる状態を必要としないことを保証します。

Client	Server
<pre>fd = open("/foo", ...); Send LOOKUP (rootdir FH, "foo")</pre>	<p>Receive LOOKUP request look for "foo" in root dir return foo's FH + attributes</p>
<p>Receive LOOKUP reply allocate file desc in open file table store foo's FH in table store current file position (0) return file descriptor to application</p>	
<hr/> <pre>read(fd, buffer, MAX); Index into open file table with fd get NFS file handle (FH) use current file position as offset Send READ (FH, offset=0, count=MAX)</pre>	<p>Receive READ request use FH to get volume/inode num read inode from disk (or cache) compute block location (using offset) read data from disk (or cache) return data to client</p>
<p>Receive READ reply update file position (+bytes read) set current file position = MAX return data/error code to app</p>	
<hr/> <pre>read(fd, buffer, MAX); Same except offset=MAX and set current file position = 2*MAX</pre>	
<hr/> <pre>read(fd, buffer, MAX); Same except offset=2*MAX and set current file position = 3*MAX</pre>	
<hr/> <pre>close(fd); Just need to clean up local structures Free descriptor "fd" in open file table (No need to talk to server)</pre>	

Figure 49.5: Reading A File: Client-side And File Server Actions

TIP: IDEMPOTENCY IS POWERFUL

Idempotency(幂等:ある操作を 1 回行っても複数回行っても結果が同じであること) は、信頼性の高いシステムを構築する際に役立ちます。操作を複数回発行できる場合、操作の失敗を処理する方がはるかに簡単です。あなたはそれを再試行することができます。操作が幂等でない場合、人生は

より困難になります。

49.7 Handling Server Failure with Idempotent Operations

クライアントがサーバーにメッセージを送信すると、応答を受信しないことがあります。この失敗に対するさまざまな理由が考えられます。場合によっては、ネットワークによってメッセージが破棄されることがあります。ネットワークはメッセージを失うので、要求または応答のいずれかが失われる可能性があり、したがってクライアントは決して応答を受信しません。

また、サーバーがクラッシュしたため、現在メッセージに応答していない可能性もあります。少し後に、サーバーは再起動され、再度実行を開始しますが、その間にすべての要求が失われます。これらのすべてのケースでは、クライアントに質問が残されます。サーバーがタイムリーに返信しないときにはどうすればよいでしょうか？

NFSv2 では、クライアントはこれらのすべての障害を、一様かつ均一な方法で処理します。単純に要求を再試行します。具体的には、要求を送信した後、クライアントは、指定された時間が経過した後にタイマーをオフにするように設定します。タイマーがオフになる前に応答が受信されると、タイマーはキャンセルされ、すべて正常となります。ただし、応答が受信される前にタイマーが切れた場合、クライアントは要求が処理されていないとみなして再送信します。サーバーが応答すると、すべて正常であり、クライアントは問題をきちんと処理しています。

クライアントの要求を単純に再試行する能力（障害の原因にかかわらず）は、ほとんどの NFS 要求の重要な特性によるものです。それらは幂等です。操作を複数回実行した結果が操作を 1 回実行した結果と同等である場合、操作は幂等と呼ばれます。たとえば、値をメモリ位置に 3 回格納する場合は、1 回と同じです。したがって、「値をメモリに格納する」は、等価な演算です。ただし、カウンターを 3 回インクリメントした場合は、カウンターを 1 回だけ増やすのとは異なる量になります。従って、「インクリメントカウンタ」は幂等ではありません。より一般的には、データを読み取るだけの操作は明らかに幂等です。データを更新する操作は、このプロパティがあるかどうかを判断するために、より慎重に検討する必要があります。

NFS におけるクラッシュリカバリの設計の中心は、最も一般的な操作の idempotency です。LOOKUP と READ の要求は、ファイルサーバからの情報の読み取りのみを行い、更新はしないため、幂等があります。さらに興味深いことに、WRITE リクエストも幂等でもあります。たとえば、WRITE が失敗した場合、クライアントは単純に WRITE を再試行できます。WRITE メッセージには、データ、カウント、およびデータを書き込む正確なオフセットが含まれています。したがって、複数の書き込みの結果が单一の結果の結果と同じであるという知識をもって繰り返すことができます。

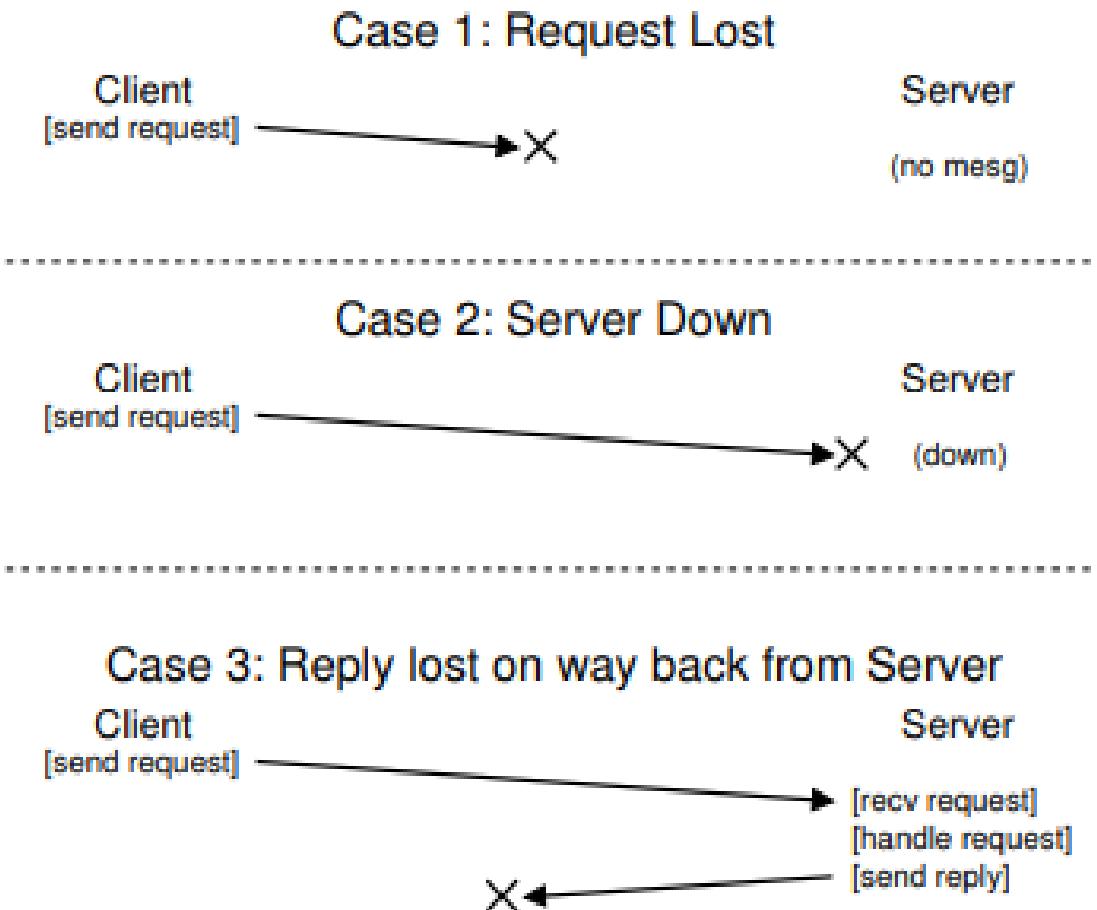


Figure 49.6: The Three Types of Loss

このようにして、クライアントは統一された方法ですべてのタイムアウトを処理できます。WRITE 要求が単純に失われた場合（上記のケース 1）、クライアントは再試行し、サーバは書き込みを実行し、すべて正常です。リクエストが送信されている間にサーバがダウンした場合でも、2 番目のリクエストが送信されたときにバックアップと実行が行われ、すべてが正常に動作します（ケース 2）。最後に、サーバは実際に WRITE 要求を受信し、そのディスクへの書き込みを発行し、応答を送信します。この返信が失われる可能性があり（ケース 3）、再びクライアントがリクエストを再送信します。サーバが要求を再度受け取ると、まったく同じことを単に実行します。データをディスクに書き込んで、それが完了したことを応答します。今度はクライアントが応答を受け取ると、すべて正常にクライアントはメッセージ損失とサーバー障害の両方を一様に処理します。

いくつかの操作は幂等にするのが難しいです。たとえば、すでに存在するディレクトリを作成しようとすると、mkdir 要求が失敗したことが通知されます。したがって、NFS では、ファイルサーバが MKDIR プロトコルメッセージを受信してそれを正常に実行しますが、応答が失われた場合、クライアントはそれを繰り返して、実際に操作が最初に成功した後に再試行で失敗したときに、その障害に遭遇します。したがって、人生は完璧ではありません。

TIP: PERFECT IS THE ENEMY OF THE GOOD (VOLTAIRE'S LAW)

あなたが美しいシステムを設計するときできえ、すべてのコーナーケースがあなたが思うように正確に動作しないことがあります。上記の mkdir の例を参照してください。異なるセマンティクスを持つように mkdir を再設計することができます。それによって、それは幂等になります（あなた

がそうする方法を考えてください)。しかし、なぜ煩わしいのでしょうか? NFS 設計の哲学は重要なケースの大部分をカバーしており、全体的にシステム設計は障害に関してきれいで簡単です。言い換えると、システムは完璧ではなく、まだシステムを構築していくことは良いエンジニアリングです。明らかに、この知恵はヴォルテールによって言及されています。「賢明なイタリア人は、最高は良の敵であると言っている」そのため、これをヴォルテールの法則と呼んでいます。

49.8 Improving Performance: Client-side Caching

分散ファイルシステムはさまざまな理由から有効ですが、ネットワーク全体ですべての読み取りおよび書き込み要求を送信すると、パフォーマンスに大きな問題が発生する可能性があります。ネットワークは通常、ローカルメモリやディスクと比べて高速ではありません。したがって、別の問題があります。分散ファイルシステムのパフォーマンスをどのように改善できますか?

上のサブタイトルの大膽な言葉を読んで、あなたが推測するかもしれない答えは、クライアント側のキャッシュです。NFS クライアント側のファイルシステムは、サーバーから読み取ったファイルデータ(およびメタデータ)をクライアントメモリにキャッシュします。したがって、第 1 のアクセスが高価である(すなわち、ネットワーク通信を必要とする)間に、その後のアクセスはクライアントメモリから非常に迅速にサービスされます。

キャッシュは、書き込み用の一時バッファとしても機能します。クライアントアプリケーションが最初にファイルに書き込むとき、クライアントはデータをサーバーに書き出す前に、データをクライアントメモリ(ファイルサーバーから読み取ったデータと同じキャッシュ内)にバッファリングします。このような書き込みバッファリングは、実際の書き込みパフォーマンスとアプリケーション `write()` の待ち時間を切り離すために便利です。つまり、`write()` へのアプリケーションの呼び出しがただちに成功します(クライアント側ファイルシステムのキャッシュにデータを入れます)。あとでデータはファイルサーバーに書き出されます。

したがって、NFS クライアントはデータをキャッシュし、パフォーマンスは通常優れており、完了しました。本当にそうでしょうか? 残念ながら、それほどではありません。複数のクライアントキャッシュを持つあらゆる種類のシステムにキャッシュを追加することは、キャッシュの一貫性の問題と呼ばれる大きく興味深い課題を引き起こします。

49.9 The Cache Consistency Problem

キャッシュ一貫性の問題は、2つのクライアントと1つのサーバーで最もよく説明されています。クライアント C1 がファイル F を読み取り、そのファイルのコピーをローカルキャッシュに保持すると仮定します。次に、別のクライアント C2 がファイル F を上書きし、その内容を変更するとします。新しいバージョンのファイル F(バージョン 2)、または F[v2] と古いバージョン F[v1] を呼び出すことで、2つの異なるファイル名を保持することができます。(ただし、ファイル名は同じですが、内容が異なるだけです) 最後に、ファイル F にまだアクセスしていない第 3 のクライアント C3 が存在します。

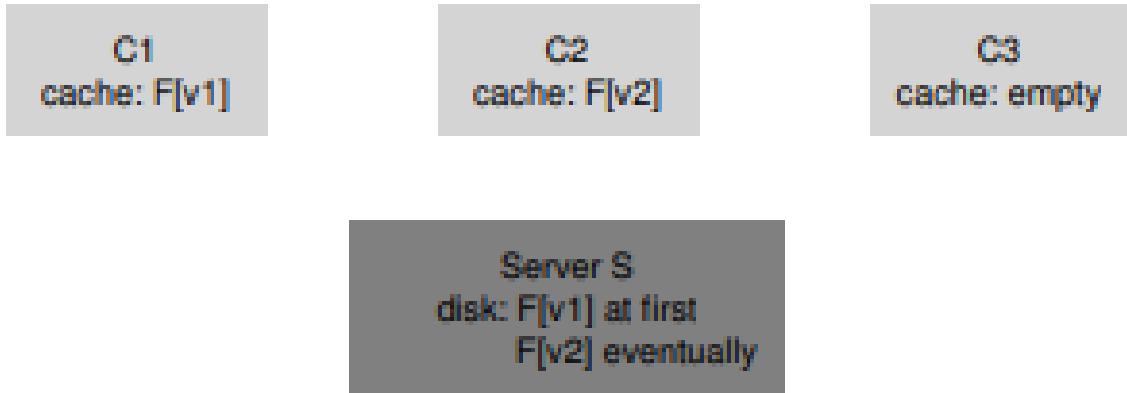


Figure 49.7: The Cache Consistency Problem

あなたはおそらく、今後の問題を見ることがあります(図 49.7)。実際、2つの副問題があります。第1の副問題は、クライアント C2 がそれらの書き込みをそのキャッシュに一時的にバッファリングしてから、それをサーバに送ることができます。この場合、F[v2] は C2 のメモリに格納されていますが、別のクライアント(たとえば C3)から F にアクセスすると、古いバージョンのファイル(F[v1])が取り出されます。したがって、クライアントでの書き込みをバッファリングすることによって、他のクライアントがファイルの古いバージョンを取得する可能性があります。マシン C2 にログインし、F を更新してから C3 にログインしてファイルを読み込もうとすると、古いコピーを取得する場合を想像してください。確かに、これはイライラする可能性があります。したがって、キャッシュ一貫性問題の update visibility(更新の可視性)と呼ばれている側面を見てみましょう。あるクライアントからの更新が他のクライアントでいつ見えるようになるのですか？

キャッシュ一貫性の第2の副問題は失効したキャッシュです。この場合、C2 は最終的にファイルサーバーへの書き込みをフラッシュし、したがってサーバーは最新バージョン(F[v2])を持ちます。しかし、C1 はキャッシュ内に F[v1] を保持しています。C1 上で実行されているプログラムがファイル F を読み込んだ場合、それは古いバージョン(F[v1])であり、最新のコピー(F[v2])ではなく(よく)望ましくありません。

NFSv2 実装は、これらのキャッシュ一貫性の問題を2つの方法で解決します。第1に、更新の可視性に対処するために、クライアントは flush on close(close to open)consistency semantics(整合性セマンティクス)と呼ばれることを実装します。具体的には、ファイルがクライアントアプリケーションに書き込まれ、続いてクライアントアプリケーションによって閉じられると、クライアントはすべての更新(つまり、キャッシュ内のダーティページ)をサーバーにフラッシュします。flush on close の一貫性により、NFS は、別のノードからの次のオープンに最新のファイル・バージョンが確実に表示されるようにします。

次に、古いキャッシュの問題に対処するために、NFSv2 クライアントはまず、キャッシュされたコンテンツを使用する前にファイルが変更されているかどうかを確認します。具体的には、ファイルを開くときに、クライアント側のファイルシステムは、ファイルの属性を取得するために GETATTR 要求をサーバーに発行します。重要なことに、属性には、ファイルがサーバー上で最後に変更された時期に関する情報が含まれています。変更時刻がクライアントキャッシュにフェッチされた時刻よりも新しい場合、クライアントはファイルを無効にしてクライアントキャッシュから削除し、その後の読み込みがサーバーに送られ、最新バージョンのファイルを取得するようにします。一方、クライアントがファイルの最新バージョンを持っていると判断すると、キャッシュされたコンテンツが使用され、パフォーマンスが向上します。

Sun のオリジナルチームがこの解決策を古いキャッシュの問題に実装したとき、彼らは新しい問題を認識しました。突然、NFS サーバーに GETATTR 要求があふれていました。従うべき良いエンジニアリングの原則は、一般的なケースを設計し、それをうまく機能させることです。ここでは、ファイルが单一のクライアント(おそらくは繰り返し)からのみアクセスされたのが一般的でしたが、クライアントは常に他の誰もファイルを

変更していないことを確認するために GETATTR 要求をサーバーに送信しなければなりませんでした。このように、ほとんどの場合、誰もが「誰かこのファイルを変更した?」と尋ね、當時確認し、クライアントはサーバを攻撃してしまいます。

このような状況(多少)を改善するために、属性キャッシュが各クライアントに追加されました。クライアントはそれにアクセスする前にファイルを検証しますが、たいていの場合、属性キャッシュを調べて属性を取得するだけです。特定のファイルの属性は、ファイルが最初にアクセスされたときにキャッシュに格納され、一定の時間(たとえば3秒)後にタイムアウトになります。したがって、これらの3秒間に、すべてのファイルアクセスは、キャッシュされたファイルを使用することがOKであると判断し、サーバーとのネットワーク通信を行わずに実行します。

49.10 Assessing NFS Cache Consistency

NFS キャッシュの一貫性についての最後の言葉。flush on close の振る舞いが「意味をなす」ために追加されましたが、特定のパフォーマンス問題が発生しました。具体的には、一時ファイルまたは短命ファイルがクライアントで作成され、すぐに削除された場合でも、サーバーに強制されます。より理想的な実装では、このような短命ファイルは削除されるまでメモリ内に保持されるため、サーバーのやりとりが完全になくなり、パフォーマンスが向上する可能性があります。

さらに重要なのは、NFS に属性キャッシュを追加すると、ファイルのどのバージョンが取得されているかを正確に理解することが難しくなりました。時々あなたは最新バージョンを手に入れます。属性キャッシュがまだタイムアウトしていない場合、クライアントがクライアントメモリにあったものをあなたに提供するため、古いバージョンを取得することができます。ほとんどの場合、これは大丈夫でしたが、時には変な行動につながることもあります。

そこで、NFS クライアントのキャッシュというおかしな点について説明しました。これは、実装の詳細が、ユーザの観測可能なセマンティクスを定義する役目をする興味深い例です。

49.11 Implications on Server-Side Write Buffering

これまでのところ、クライアントのキャッシュに焦点が当てられていました。それは興味深い問題のほとんどが発生する場所です。しかし、NFS サーバーはメモリが大量に消費されるマシンであることが多いため、キャッシュに関する懸念もあります。データ(およびメタデータ)がディスクから読み込まれると、NFS サーバーはそのデータをメモリに保持し、その後のデータ(およびメタデータ)の読み取りはディスクには行きません。これにパフォーマンスが(少し)向上します。

より興味深いのは、書き込みバッファリングの場です。NFS サーバは、書き込みが安定した記憶装置(例えば、ディスクまたは他の永続的デバイス)に強制されるまで、WRITE プロトコル要求で成功を返すことは絶対にありません。サーバーのメモリにデータのコピーを置くことができますが、WRITE プロトコル要求でクライアントに成功を返すと、正しく動作しなくなる可能性があります。なぜかあなたは理解できますか?

答えは、クライアントがサーバーの障害をどのように処理するかについての前提にあります。クライアントが発行した次の書き込み順序を想像してみてください。

```
write(fd, a_buffer, size); // fill first block with a's
write(fd, b_buffer, size); // fill second block with b's
write(fd, c_buffer, size); // fill third block with c's
```

これらの書き込みは、ファイルの3つのブロックを a、b、c のブロックで上書きします。したがって、ファイルは最初に次のように見えます。

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
```

これらの書き込み後の最終的な結果は、x、y、z がそれぞれ a、b、c で上書きされることを期待するかもしれません。

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
cccccccccccccccccccccccccccccccccccccccccccccc
```

ここでは、この 3 つのクライアントの書き込みが、3 つの異なる WRITE プロトコル・メッセージとしてサーバーに発行されたと想定します。第 1 の WRITE メッセージがサーバによって受信され、ディスクに発行され、クライアントがその成功を通知したと仮定します。ここで、2 番目の書き込みがメモリにバッファリングされていると仮定し、サーバーはクライアントに成功を報告してから、強制的にディスクに書き込みます。残念ながら、サーバーはディスクに書き込む前にクラッシュします。サーバーはすぐに再起動し、3 回目の書き込み要求を受信します。これも成功します。したがって、クライアントに対してはすべての要求は成功しましたが、ファイルの内容が次のようにになっていることに驚いています。

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy <--- oops
cccccccccccccccccccccccccccccccccccccccccccccccc
```

Yikes ! サーバーは、2 番目の書き込みがディスクにコミットする前に成功したことをクライアントに通知したため、古いチャンクがファイルに残されます。これは、アプリケーションによっては致命的なものです。この問題を回避するには、クライアントに成功を通知する前に、NFS サーバーが各書き込みを安定した(永続的な)ストレージにコミットする必要があります。これにより、クライアントは書き込み中にサーバー障害を検出し、最終的に成功するまで再試行できます。そうすることで、上の例のようにファイルの内容が混ざってしまうことはありません。

この要件が NFS サーバーの実装で発生する問題は、書き込みパフォーマンスに対して大きな注意を払っていないため、パフォーマンスの大きなボトルネックになる可能性があることです。事実、書き込みを迅速に実行できる NFS サーバを構築するという単純な目的で、一部の企業 (Network Appliance など) が生まれました。彼らが使用するトリックの 1 つは、最初に battery backed memory に書き込みを入れることで、データを失うことがなく、すぐにディスクに書き込む必要もなく、書き込み要求にすばやく応答することができます。2 番目のトリックは、最終的にそうする必要があるときにディスクにすばやく書き込むように設計されたファイルシステム設計を使用することです [HLM94, RO91]。

49.12 Summary

私たちは、NFS 分散ファイルシステムの導入を見てきました。NFS は、サーバの障害に直面している単純かつ高速なりカバリのアイデアを中心にしており、慎重なプロトコル設計によってこの目的を達成しています。操作の強制力は不可欠です。クライアントが失敗した操作を安全に再生できるので、サーバーが要求を実行したかどうかにかかわらず、そうすることは大丈夫です。

また、マルチクライアント、シングルサーバシステムへのキャッシングの導入が複雑になる可能性もあります。特に、システムは、合理的に動作するためにキャッシング一貫性の問題を解決する必要があります。しかし、NFS はやや特殊なやり方でこれを実行しますが、これは時折、観察可能な変な動作を引き起こすことがあります。最後に、サーバーのキャッシングをどのように扱うのが難しいのかを見てみましょう。成功を返す前に、

サーバーへの書き込みを安定したストレージに強制する必要があります（それ以外の場合はデータが失われる可能性があります）。

私たちは確かに関連性の高い他の問題、特にセキュリティについては話していません。初期の NFS 実装におけるセキュリティは非常に緩慢でした。クライアント上の任意のユーザーが他のユーザーとして偽装して、実質的にすべてのファイルにアクセスすることはかなり容易でした。より重大な認証サービス（例えば、Kerberos [NT94]）とのその後の統合は、これらの明白な欠点に対処してきました。

参考文献

[C00] “NFS Illustrated”

Brent Callaghan

Addison-Wesley Professional Computing Series, 2000

A great NFS reference; incredibly thorough and detailed per the protocol itself.

[HLM94] “File System Design for an NFS File Server Appliance”

Dave Hitz, James Lau, Michael Malcolm

USENIX Winter 1994. San Francisco, California, 1994

Hitz et al. were greatly influenced by previous work on log-structured file systems.

[NT94] “Kerberos: An Authentication Service for Computer Networks”

B. Clifford Neuman, Theodore Ts'o

IEEE Communications, 32(9):33-38, September 1994

Kerberos is an early and hugely influential authentication service. We probably should write a book chapter about it sometime...

[O91] “The Role of Distributed State”

John K. Ousterhout

Available: <ftp://ftp.cs.berkeley.edu/ucb/sprite/papers/state.ps>

A rarely referenced discussion of distributed state; a broader perspective on the problems and challenges.

[P+94] “NFS Version 3: Design and Implementation”

Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, Dave Hitz

USENIX Summer 1994, pages 137-152

The small modifications that underlie NFS version 3.

[P+00] “The NFS version 4 protocol”

Brian Pawlowski, David Noveck, David Robinson, Robert Thurlow

2nd International System Administration and Networking Conference (SANE 2000)

Undoubtedly the most literary paper on NFS ever written.

[RO91] “The Design and Implementation of the Log-structured File System”

Mendel Rosenblum, John Ousterhout

Symposium on Operating Systems Principles (SOSP), 1991

LFS again. No, you can never get enough LFS.

[S86] “The Sun Network File System: Design, Implementation and Experience”

Russel Sandberg

USENIX Summer 1986

The original NFS paper; though a bit of a challenging read, it is worthwhile to see the source of these wonderful ideas.

[Sun89] “NFS: Network File System Protocol Specification”

Sun Microsystems, Inc. Request for Comments: 1094, March 1989

Available: <http://www.ietf.org/rfc/rfc1094.txt>

The dreaded specification; read it if you must, i.e., you are getting paid to read it. Hopefully, paid a lot.

Cash money!

[V72] “La Begueule”

Francois-Marie Arouet a.k.a. Voltaire

Published in 1772

Voltaire said a number of clever things, this being but one example. For example, Voltaire also said “If you have two religions in your land, the two will cut each others throats; but if you have thirty religions, they will dwell in peace.” What do you say to that, Democrats and Republicans?

50 The Andrew File System (AFS)

Andrew File System は、カーネギーメロン大学 (CMU) で 1980 年代に導入されました [H + 88]。カーネギーメロン大学の Satyanarayanan 教授 (略して「Satya」) が率いるこのプロジェクトの主な目標は簡単でした。それはスケールです。具体的には、サーバーができるだけ多くのクライアントをサポートできるように、分散ファイルシステムをどのように設計できますか？

興味深いことに、スケーラビリティに影響を与える設計と実装には、さまざまな側面があります。最も重要なのは、クライアントとサーバーの間のプロトコルの設計です。たとえば、NFS では、プロトコルによって、クライアントは定期的にサーバーをチェックして、キャッシュされた内容が変更されたかどうかを判断します。各チェックではサーバーリソース (CPU およびネットワーク帯域幅を含む) が使用されるため、このようなチェックを頻繁に行うと、サーバーが応答できるクライアントの数が制限され、スケーラビリティが制限されます。

AFS はまた、当初から合理的な目に見える動作がファーストクラスの関心事であった点で NFS とは異なります。NFS では、クライアント側のキャッシュタイムアウト間隔を含む低レベルの実装の詳細に直接依存するため、キャッシュの一貫性は記述しにくいです。AFS では、キャッシュの一貫性は簡単で分かりやすく、ファイルがオープンされると、クライアントは通常、サーバーから最新の一貫性のあるコピーを受け取ります。

50.1 AFS Version 1

AFS [H + 88, S + 85] の 2 つのバージョンについて説明します。最初のバージョン (AFSv1 と呼ばれます) が、実際は元のシステムは ITC 分散ファイルシステム [S + 85] と呼ばれていました) では、基本的な設計の一部が行われていましたが、スケールの設計はされていません。それを含めて再設計された最終的なプロトコルがあります (AFSv2、または単に AFS と呼ぶ)[H + 88]。ここで最初のバージョンについて説明します。

<code>TestAuth</code>	<code>Test whether a file has changed (used to validate cached entries)</code>
<code>GetFileStat</code>	<code>Get the stat info for a file</code>
<code>Fetch</code>	<code>Fetch the contents of file</code>
<code>Store</code>	<code>Store this file on the server</code>
<code>SetFileStat</code>	<code>Set the stat info for a file</code>
<code>ListDir</code>	<code>List the contents of a directory</code>

Figure 50.1: AFSv1 Protocol Highlights

AFS のすべてのバージョンの基本的な考え方の 1 つは、ファイルにアクセスしているクライアント・マシンのローカル・ディスク上のファイル全体のキャッシュです。ファイルを `open()` するとき、ファイル全体 (存在する場合) がサーバーからフェッチされ、ローカルディスク上のファイルに保存されます。後続のアプリケーションの `read()` および `write()` 操作は、ファイルが格納されているローカルファイルシステムにリダイレクトされます。したがって、これらの操作はネットワーク通信を必要とせず、高速です。最後に、`close()` のときに、ファイル (変更されている場合) がサーバーにフラッシュされます。NFS はブロックをキャッシュします (ファイル全体ではなく、NFS はファイル全体のすべてのブロックをキャッシュすることはもちろんですが) が、クライアントメモリ (ローカルディスクではありません) にキャッシュするキャッシュとは明らかに対照的です。

もう少し詳しく説明しましょう。クライアント・アプリケーションが最初に `open()` を呼び出すと、(AFS

設計者が Venus と呼ぶ) AFS クライアント側コードは、フェッチプロトコル・メッセージをサーバーに送信します。フェッチプロトコルメッセージは、ファイルサーバ (Vice と呼ばれるグループ) に目的のファイル (たとえば、/home/remzi/notes.txt) のパス名全体を渡します。その後、パス名をたどって目的のファイルを探し、ファイル全体をクライアントに戻します。クライアントサイドのコードは、クライアントのローカルディスクにファイルを (ローカルディスクに書き込むことによって) キャッシュします。上記のように、後続の `read()` および `write()` システム・コールは AFS では厳密にローカルです (サーバとの通信は発生しません)。それらは単にファイルのローカルコピーにリダイレクトされます。`read()` と `write()` の呼び出しはローカルファイルシステムへの呼び出しと同じように動作するため、ブロックにアクセスするとクライアントのメモリにもキャッシュされます。したがって、AFS はクライアント・メモリーも使用して、ローカル・ディスクにあるブロックのコピーをキャッシュします。最後に、終了すると、AFS クライアントは、ファイルが変更された (すなわち、書き込みのために開かれた) か否かをチェックします。もしそうであったら、新しいバージョンを Store プロトコルメッセージでサーバーにフラッシュし、ファイルとパス名の全体を永続的な保存のためにサーバーに送信します。

次回にファイルにアクセスするとき、AFSv1 はずっと効率的です。具体的には、クライアント側コードは、ファイルが変更されたかどうかを判断するために、最初に (TestAuth プロトコルメッセージを使用して) サーバに接続します。そうでない場合、クライアントはローカルにキャッシュされたコピーを使用し、ネットワーク転送を回避してパフォーマンスを向上させます。上記の図は、AFSv1 のプロトコル・メッセージの一部を示しています。この初期のバージョンのプロトコルでは、ファイルの内容のみがキャッシュされていました。たとえば、ディレクトリはサーバーに保存されていました。

TIP: MEASURE THEN BUILD (PATTERSON'S LAW)

私たちのアドバイザーである David Patterson (RISC と RAID の名声) は、この問題を解決する新しいシステムを構築する前に、常にシステムの測定と問題の実証を促していました。本能ではなく実験的証拠を用いることで、システム構築のプロセスをより科学的な取り組みに変えることができます。そうすることで、改善されたバージョンが開発される前に、システムをどの程度正確に測定するかを考えさせるという利点があります。最終的に新しいシステムを構築するときには、結果的に 2 つの点が優れています。まず、実際の問題を解決しているという証拠があります。次に、新しいシステムを適切な場所で測定し、実際に最新の状態を改善する方法を示しました。それで、私たちはこれをパターソンの法則と呼んでいます。

50.2 Problems with Version 1

この最初のバージョンの AFS のいくつかの重要な問題は、設計者がファイルシステムを再考する動機となりました。問題を詳細に調べるために、AFS の設計者は、何が間違っていたのかを見つけるために、既存のプロトタイプを測定するのに多大な時間を費やしました。このような実験は良いことです。なぜなら、測定はシステムの仕組みとその改善方法を理解する鍵です。具体的なデータを得ることは、システム構築の必要な部分です。彼らの研究では、AFSv1 の主な 2 つの問題点を発見しました。

- パストラバーサルコストが高すぎます：

フェッチまたはストアプロトコル要求を実行するとき、クライアントはパス名全体 (たとえば、/home/remzi/notes.txt) をサーバーに渡します。サーバは、ファイルにアクセスするためには、完全なパス名トラバーサルを実行しなければなりません。最初にルートディレクトリで home を探し、次に home で remzi を探します。そして最終的に目的のファイルが位置しているところまで下がっていきます。一度に多くのクライアントがサーバーにアクセスすると、AFS の設計者は、サーバーがディレクトリー・パスを辿るだけの CPU 時間の大部分を費やしていることに気付きました。

- クライアントが多すぎる TestAuth プロトコルメッセージを発行します。

NFS や GETATTR プロトコル・メッセージの過多のように、AFSv1 はローカル・ファイル (またはその統計情報) が TestAuth プロトコル・メッセージで有効かどうかを確認するために大量のトランザクションを生成しました。したがって、サーバーは、キャッシュされたファイルのコピーを使用することが OK であったかどうかをクライアントに伝えるのに多くの時間を費やしました。ほとんどの場合、答えはファイルが変更されていないということでした。

実際には、AFSv1 には 2 つの問題がありました。サーバー間で負荷が分散されていないため、サーバーはクライアントごとに異なるプロセスを使用し、コンテキスト切り替えやその他のオーバーヘッドを引き起こしました。負荷の不均衡の問題は、負荷を分散するために管理者がサーバー間を移動できるボリュームを導入することで解決されました。プロセスの代わりにスレッドを使用してサーバーを構築することにより、コンテキスト・スイッチの問題が AFSv2 で解決されました。しかし、文書のスペースのために、ここでは、システムの規模を制限する上記の 2 つの主要なプロトコル問題に焦点を当てています。

50.3 Improving the Protocol

上記の 2 つの問題は、AFS のスケーラビリティを制限しました。サーバ CPU がシステムのボトルネックになり、各サーバは過負荷にならないために 20 クライアントにしかサービスできませんでした。サーバーが TestAuth メッセージを受信しすぎていて、フェッチまたはストアメッセージを受信したときに、ディレクトリ階層を走査するのに時間がかかり過ぎていました。したがって、AFS 設計者は問題に直面していました。

THE CRUX: HOW TO DESIGN A SCALABLE FILE PROTOCOL

どのようにしてサーバ相互作用の数を最小限にするためにプロトコルを再設計するべきですか？つまり、TestAuth メッセージの数をどのように減らすことができますか？さらに、これらのサーバーのやりとりを効率的にするためにプロトコルをどのように設計することができますか？これらの問題の両方を攻撃することにより、新しいプロトコルは AFS というよりスケーラブルなバージョンになります。

50.4 AFS Version 2

AFSv2 では、クライアント/サーバーの対話数を減らすためのコールバックという概念が導入されました。コールバックとは、サーバーからクライアントへの約束であり、クライアントがキャッシュしているファイルが変更されたときにサーバーがクライアントに通知することを意味します。この状態をシステムに追加することで、クライアントはサーバーに接続しなくとも、キャッシュされたファイルがまだ有効かどうかを調べる必要がなくなります。むしろ、サーバーはそれがそうでないと指示するまでファイルが有効であるとみなします。ポーリングと割り込みの類推に注意してください。

AFSv2 では、クライアントが関心を持っているファイルを指定するために、パス名ではなく file identifier(FID)(NFS ファイルハンドルに似ています) という概念も導入されました。AFS の FID は、ボリューム識別子、ファイル識別子、uniquifier(ファイルが削除されたときにボリュームとファイル ID の再利用を可能にするため) があります。したがって、サーバーにパス名全体を送信し、サーバーがパス名を調べて目的のファイルを見つけるのではなく、クライアントはパス名を一度に 1 つずつ歩き、結果をキャッシングしてサーバーの負荷を軽減します。

たとえば、クライアントが /home/remzi/notes.txt ファイルにアクセスし、home が / にマウントされた AFS ディレクトリー (つまり、/ がローカル・ルート・ディレクトリーであったが、ホームとその子が AFS にある) であった場合、クライアントは最初に home のディレクトリ内容を取得し、それらをローカルディスクに

キャッシュし、ホームにコールバックを設定します。次に、クライアントはディレクトリ remzi をフェッチし、ローカルディスクにキャッシュし、remzi でコールバックを設定します。最後に、クライアントは notes.txt を取得し、この通常のファイルをローカルディスクにキャッシュし、コールバックを設定し、最後ファイルディスクリプタを呼び出しアプリケーションに返します。要約については、図 50.2 を参照してください。

Client (C ₁)	Server
<pre>fd = open("/home/remzi/notes.txt", ...); Send Fetch (home FID, "remzi")</pre>	<pre>Receive Fetch request look for remzi in home dir establish callback(C₁) on remzi return remzi's content and FID</pre>
<pre>Receive Fetch reply write remzi to local disk cache record callback status of remzi Send Fetch (remzi FID, "notes.txt")</pre>	<pre>Receive Fetch request look for notes.txt in remzi dir establish callback(C₁) on notes.txt return notes.txt's content and FID</pre>
<pre>Receive Fetch reply write notes.txt to local disk cache record callback status of notes.txt local open () of cached notes.txt return file descriptor to application</pre>	<pre>Read(fd, buffer, MAX); perform local read () on cached copy</pre>
<pre>close(fd); do local close () on cached copy if file has changed, flush to server</pre>	<pre>fd = open("/home/remzi/notes.txt", ...); Foreach dir (home, remzi) if (callback(dir) == VALID) use local copy for lookup(dir) else Fetch (as above) if (callback(notes.txt) == VALID) open local cached copy return file descriptor to it else Fetch (as above) then open and return fd</pre>

Figure 50.2: Reading A File: Client-side And File Server Actions

しかし、NFS との主な違いは、ディレクトリまたはファイルの各フェッチで AFS クライアントがサーバーとコールバックを確立し、サーバーがキャッシュ状態の変更をクライアントに通知することを保証することです。利点は明白です。/home/remzi/notes.txt への最初のアクセスは（上記のように）多数のクライアント/サーバ・メッセージを生成しますが、ファイル notes.txt と同様にすべてのディレクトリのコールバックを確立します。アクセスは全てローカルであり、サーバーとのやり取りはまったく必要ありません。したがって、クライアントでファイルがキャッシュされる一般的なケースでは、AFS はローカルディスクベースのファイル

システムとほぼ同じように動作します。1つのファイルに複数回アクセスすると、2回目のアクセスはファイルにローカルでアクセスするのと同じくらい速くなければなりません。

ASIDE: CACHE CONSISTENCY IS NOT A PANACEA

分散ファイルシステムについて論じるときには、ファイルシステムが提供するキャッシュの一貫性が非常に重要です。しかし、このベースラインの一貫性は、複数のクライアントからのファイルアクセスに関するすべての問題を解決するものではありません。たとえば、複数のクライアントがチェックインとコードのチェックアウトを実行するコードリポジトリを構築している場合、基礎となるファイルシステムに依存してすべての作業を行うことはできません。そのような同時アクセスが発生した場合に「正しい」ことが起こることを確実にするために、明示的なファイルレベルのロックを使用する必要があります。実際、同時更新を本当に気にするアプリケーションは、競合を処理するための余分な機構を追加します。この章で説明したベースラインの一貫性と以前のものは、主にカジュアルな使用に役立ちます。つまり、ユーザーが別のクライアントにログインすると、合理的なバージョンのファイルが表示されます。これらのプロトコルからもっと多くのことを期待することは、失敗、失望、そして涙が詰まった不満のために自分自身で設定することです。

50.5 Cache Consistency

私たちが NFS について議論したとき、私たちが検討したキャッシュ一貫性の 2つの側面、update visibility(更新の可視性) と cache staleness(キャッシュが古くなる) がありました。更新の可視性では、問題は次のような場合です。サーバーはいつ新しいバージョンのファイルで更新されますか？ cache staleness の問題は次のとおりです。サーバーに新しいバージョンがインストールされると、古いバージョンのキャッシュされたコピーではなく新しいバージョンがクライアントに表示されるまでの時間はどのくらいですか？

コールバックとファイル全体のキャッシュのために、AFS によって提供されるキャッシュ一貫性は、記述し理解しやすいです。考慮すべき重要な 2つのケースがあります。異なるマシン上のプロセス間の一貫性と、同じマシン上のプロセス間の一貫性です。

異なるマシン間では、AFS により、サーバーで更新が表示され、更新されたファイルがクローズされたときとまったく同じ時刻にキャッシュ・コピーが無効になります。クライアントはファイルを開き、それに（おそらくは繰り返して）書き込みを行います。最終的に閉じられると、新しいファイルはサーバーにフラッシュされます（したがって可視化されます）。この時点で、サーバーはキャッシュされたコピーを持つクライアントのコールバックを「break(中断)」します。ブレークは、各クライアントに連絡して、ファイル上のコールバックがもはや有効ではないことを通知することで実現されます。この手順では、クライアントがファイルの失効したコピーを読み取らないようにします。これらのクライアントで後で開くと、サーバーから新しいバージョンのファイルを再度取得する必要があります（また、新しいバージョンのファイルでコールバックを再確立する役割も果たします）。

AFS は、同じマシン上のプロセス間でこの単純なモデルを例外にします。この場合、ファイルへの書き込みは、他のローカルプロセスに即座に見えます（すなわち、最新の更新を見るためにファイルが閉じられるまで待つ必要はない）。この動作は、一般的な UNIX のセマンティクスに基づいているため、单一のマシンを使用すると、期待どおりに動作します。別のマシンに切り替える場合にのみ、より一般的な AFS 整合性メカニズムを検出できます。

さらなる議論に値する興味深いものがクロスマシンの場合に 1 つあります。特に、異なるマシン上のプロセスが同時にファイルを変更している場合、AFS は当然のことながら最後の書き込んだもの（おそらく last closer wins と呼ばれる）を採用しています。具体的には、`close()` を最後に呼び出したクライアントは、最後にサーバー上のファイル全体を更新し、したがって「勝った」ファイル、つまり他の人が見ることができる

ようにサーバー上に残っているファイルになります。結果は、1つのクライアントまたは別のクライアントによって全体的に生成されたファイルです。NFS のようなブロックベースのプロトコルとの違いに注意してください。NFS では、個々のブロックの書き込みが、各クライアントがファイルを更新するときにサーバにフラッシュされる可能性があり、最終的にサーバ上のファイルが両方のクライアントからの複数の更新が混ざった状態になります。多くの場合、そのような混合ファイル出力はありません。すなわち、2つのクライアントによって JPEG 画像が改変されると想像してみてください。結果として生じる書き込みの混合は、有効な JPEG を構成しない可能性があります。

これらの異なるシナリオのいくつかを示すタイムラインを図 50.3 に示します。列は、Client1 とそのキャッシュ状態の 2 つのプロセス (P1 と P2)、Client2 の 1 つのプロセス (P3) とそのキャッシュ状態、およびサーバー (Server) の動作を示しています。全てが想像上、F と呼ばれる单一のファイル上で動作します。サーバーの場合、図は左の操作が完了した後のファイルの内容を単純に示しています。それを読んで、なぜそれぞれの読み取りが結果を返すのか理解できるかどうかを確認してください。あなたが迷った場合、右のコメント欄が役立ちます。

	Client ₁			Client ₂		Server	Comments
	P ₁	P ₂	Cache	P ₃	Cache	Disk	
open(F)	-			-		-	File created
write(A)	A			-		-	
close()	A			-		A	
	open(F)	A		-		A	
	read() → A	A		-		A	
	close()	A		-		A	
open(F)	A			-		A	
write(B)	B			-		A	
	open(F)	B		-		A	Local processes
	read() → B	B		-		A	see writes immediately
	close()	B		-		A	
		B	open(F)	A		A	Remote processes
		B	read() → A	A		A	do not see writes...
		B	close()	A		A	
close()	B			A		B	... until close() has taken place
	B		open(F)	B		B	
	B		read() → B	B		B	
	B		close()	B		B	
	B		open(F)	B		B	
open(F)	B			B		B	
write(D)	D			B		B	
	D		write(C)	C		B	
	D		close()	C		C	
close()	D			C		D	
	D		open(F)	D		D	Unfortunately for P ₃
	D		read() → D	D		D	the last writer wins
	D		close()	D		D	

Figure 50.3: Cache Consistency Timeline

50.6 Crash Recovery

上記の説明から、クラッシュリカバリが NFS よりも関与していることがわかります。あなたは正しいでしょう。例えば、クライアント C1 が再起動しているときなど、サーバ (S) がクライアント (C1) に接続できない短

い時間があるとします。C1 は利用できませんが、S は 1 つ以上のコールバックリコールメッセージを送信しようとしている可能性があります。例えば、C1 がそのローカルディスク上にファイル F をキャッシュし、次に C2(別のクライアント) が F を更新したとすると、S はそのファイルをキャッシュしてローカルキャッシュから削除するメッセージをすべてのクライアントに送信します。C1 は再起動時にクリティカルなメッセージを見逃す可能性があるため、システムに再参加する際に C1 はすべてのキャッシュ内容を疑わしいものとして扱う必要があります。したがって、ファイル F への次のアクセス時に、C1 は、ファイル F のキャッシュされたコピーがまだ有効であるかどうかをサーバに (TestAuth プロトコルメッセージを用いて) 尋ねるべきです。もしそうなら、C1 はそれを使うことができます。もしそうでなければ、C1 はサーバから新しいバージョンを取り出すべきです。

クラッシュ後のサーバ回復もより複雑です。発生する問題は、コールバックがメモリ内に保持されていることです。したがって、サーバが再起動すると、どのクライアントマシンがどのファイルを持っているかは分かりません。したがって、サーバーを再起動すると、サーバーの各クライアントは、サーバーがクラッシュして、すべてのキャッシュの内容を疑わしいものとして扱い、それを使用する前にファイルの有効性を再確立する必要があります。したがって、サーバクラッシュは、各クライアントがタイムリーにクラッシュを認識していることを保証しなければならない、またはクライアントが古いファイルにアクセスする危険性がある大きなイベントです。このような回復を実装するには多くの方法があります。たとえば、サーバーを起動して再度実行するときに、それぞれのクライアントに対して (「キャッシュ内容を信頼しないで」) とメッセージを送ります。またはクライアントが定期的にサーバーが稼働していることを確認します (heartbeat メッセージと呼ばれています)。ご覧のように、よりスケーラブルで合理的なキャッシングモデルを構築するコストがあります。NFS では、クライアントはほとんどサーバクラッシュに気付けませんでした。

50.7 Scale And Performance Of AFSv2

新しいプロトコルを導入したことでの AFSv2 が測定され、元のバージョンよりもはるかにスケーラブルであることがわかりました。実際、各サーバーは約 20 のクライアントではなく約 50 のクライアントをサポートできます。さらに利点は、クライアント側のパフォーマンスがローカルのパフォーマンスに非常に近くなることでした。一般的なケースでは、すべてのファイルアクセスがローカルであったためです。ファイルの読み込みは通常、ローカルのディスクキャッシング (および場合によってはローカルメモリ) に送られます。クライアントが新しいファイルを作成したり、既存のファイルに書き込んだ場合にのみ、Store メッセージをサーバーに送信し、新しい内容でファイルを更新する必要がありました。

一般的なファイル・システム・アクセス・シナリオを NFS と比較することにより、AFS パフォーマンスに関するいくつかの見通しを得ることもできます。図 50.4 は、我々の定性的比較の結果を示しています。

Workload	NFS	AFS	AFS/NFS
1. Small file, sequential read	$N_s \cdot L_{net}$	$N_s \cdot L_{net}$	1
2. Small file, sequential re-read	$N_s \cdot L_{mem}$	$N_s \cdot L_{mem}$	1
3. Medium file, sequential read	$N_m \cdot L_{net}$	$N_m \cdot L_{net}$	1
4. Medium file, sequential re-read	$N_m \cdot L_{mem}$	$N_m \cdot L_{mem}$	1
5. Large file, sequential read	$N_L \cdot L_{net}$	$N_L \cdot L_{net}$	1
6. Large file, sequential re-read	$N_L \cdot L_{net}$	$N_L \cdot L_{disk}$	$\frac{L_{disk}}{L_{net}}$
7. Large file, single read	L_{net}	$N_L \cdot L_{net}$	N_L
8. Small file, sequential write	$N_s \cdot L_{net}$	$N_s \cdot L_{net}$	1
9. Large file, sequential write	$N_L \cdot L_{net}$	$N_L \cdot L_{net}$	1
10. Large file, sequential overwrite	$N_L \cdot L_{net}$	$2 \cdot N_L \cdot L_{net}$	2
11. Large file, single write	L_{net}	$2 \cdot N_L \cdot L_{net}$	$2 \cdot N_L$

Figure 50.4: Comparison: AFS vs. NFS

図では、さまざまなサイズのファイルに対して、一般的な読み書きパターンを分析的に調べます。小さなファイルには N_s 個のブロックがあります。中規模のファイルには N_m 個のブロックがあります。大きなファイルには N_L ブロックがあります。中小のファイルはクライアントのメモリに収まると仮定します。大容量のファイルはローカルディスクに収められますが、クライアントメモリには収まりません。

また、分析のために、ファイルブロックのリモートサーバーへのネットワーク経由のアクセスには、 L_{net} の時間単位がかかります。ローカルメモリへのアクセスには L_{mem} が必要で、ローカルディスクへのアクセスには L_{disk} が必要です。一般的な前提は、 $L_{net} > L_{disk} > L_{mem}$ です。最後に、ファイルへの最初のアクセスがどのキャッシュでもヒットしないと仮定します。関連するキャッシュがファイルを保持するのに十分な容量を有する場合、引き続くファイルアクセス(すなわち、「再読み込み」)はキャッシュ内でヒットします。

図の列は、特定の操作(例えば、小さなファイル順次読み取り)が NFS または AFS のいずれかにおおよそかかる時間を示しています。右端の列には、AFS と NFS の比率が表示されています。

我々は以下の観察を行います。まず、多くの場合、各システムのパフォーマンスはほぼ同等です。例えば、最初にファイル(例えば、仕事量 1, 3, 5)を読み取るとき、リモートサーバからファイルをフェッチする時間がほとんどであり、両方のシステムで同じです。この場合、ファイルをローカル・ディスクに書き込む必要があるため、AFS が遅いと考えるかもしれません。ただし、これらの書き込みはローカル(クライアント側)のファイルシステムキャッシュによってバッファされるため、コストは隠されている(上手く消している)可能性があります。同様に、AFS がキャッシュされたコピーをディスクに保管するため、ローカル・キャッシュ・コピーからの AFS 読み取りが遅くなると考えるかもしれません。ただし、AFS はここでもローカル・ファイル・システムのキャッシングから利益を得ています。AFS 上の読み取りはクライアント側のメモリー・キャッシュでヒットし、パフォーマンスは NFS と似ています。

第 2 に、大きなファイルシーケンシャル再読み込み(Workload 6)の間に興味深い違いが生じます。AFS は大きなローカル・ディスク・キャッシュを持っているため、ファイルに再度アクセスするとそこからファイルにアクセスします。対照的に、NFS はクライアントメモリ内のブロックのみをキャッシュすることができます。その結果、大きなファイル(すなわち、ローカルメモリよりも大きいファイル)が再読された場合、NFS クライアントはリモートサーバからファイル全体を再フェッチする必要があります。したがって、AFS は、リモートアクセスが実際にローカルディスクよりも遅いと仮定すると、この場合は L_{net} / L_{disk} の要因によって NFS より速くなります。この場合の NFS はサーバーの負荷を増加させ、規模にも影響します。

第 3 に、(新しいファイルの)順次書き込みは、両方のシステム(Workloads 8,9)で同様に実行する必要があることに注意してください。この場合、AFS はファイルをローカル・キャッシュ・コピーに書き込みます。ファイルがクローズされると、AFS クライアントはプロトコルに従ってサーバーへの書き込みを強制します。

NFS はクライアント側のメモリ圧迫のために、おそらくいくつかのブロックをサーバーに強制しますが、ファイルが閉じられたときにサーバーに書き込むことで、NFS フ flush on close の一貫性を維持します。すべてのデータをローカル・ディスクに書き込むため、AFS の方が遅いと考えるかもしれません。しかし、それがローカルのファイルシステムに書き込んでいることに気づいてください。これらの書き込みは最初にページ・キャッシュにコミットされ、後で(バックグラウンドで)ディスクにのみ行われるため、AFS はクライアント側のメモリキャッシング基盤のパフォーマンスが向上するため、メリットを受けます。

第 4 に、順次ファイルの上書き (Workload 10) で AFS が悪化することに注意してください。ここまででは、書き込んだ仕事量も新しいファイルを作成していると想定していました。今回の場合、ファイルが存在し上書きされます。上書きは AFS にとって特に悪いケースです。これは、クライアントが古いファイルを最初にフェッチし、後でそのファイルを上書きするためです。対照的に、NFS は単にブロックを上書きして、最初の(役に立たない)読み込みを回避します。

最後に、大容量ファイル内の小さなデータ・サブセットにアクセスする仕事量は、AFS(Workloads 7,11)よりも NFS が優れたパフォーマンスを発揮します。このような場合、AFS プロトコルは、ファイルがオープンされたときにファイル全体をフェッチします。残念ながら、わずかな読み取りまたは書き込みしか実行されません。さらに悪いことに、ファイルが変更されると、ファイル全体がサーバーに書き戻され、パフォーマンスの影響が倍増します。NFS は、ブロックベースのプロトコルとして、読み取りまたは書き込みのサイズに比例する入出力を実行します。

全体的に見て、NFS と AFS は異なる仮定をしており、結果としてパフォーマンスの結果が異なることは驚くことではありません。これらの違いが重要かどうかは、いつものように仕事量の問題です。

ASIDE: THE IMPORTANCE OF WORKLOAD

どのシステムを評価するかの課題は、仕事量の選択です。コンピュータシステムは非常に多くの異なる方法で使用されるため、選択する仕事量は多種多様です。適切な設計上の決定を下すために、ストレージシステムの設計者はどの仕事量を重要なものにするべきですか？

AFS の設計者は、ファイルシステムの使用状況を測定した経験から、特定の仕事量の前提を設定しました。特に、ほとんどのファイルは頻繁に共有されず、すべてが順番にアクセスされると想定していました。これらの仮定を前提とすると、AFS 設計は完璧な意味を持ちます。

しかし、これらの仮定は必ずしも正しいとは限りません。たとえば、情報を定期的にログに追加するアプリケーションを考えてみましょう。既存の大きなファイルに少量のデータを追加するこれらの小さなログ書き込みは、AFS にとってかなり問題です。他の多くの困難な仕事量、例えばトランザクションデータベースにおけるランダムな更新も同様に存在します。

どのタイプの仕事量が共通しているかについての情報を得るための 1 つの場所は、実行されたさまざまな調査研究によるものです。AFS の遡及 [H + 88] を含む仕事量分析 [B + 91, H + 11, R + 00, V99] の良い例については、これらの調査のいずれかを参照してください。

50.8 AFS: Other Improvements

バークレーの FFS(シンボリックリンクやその他の機能を追加した) の紹介で見たように、AFS の設計者は、システムを構築してシステムを使いやすく管理するための多くの機能を追加する機会を得ました。たとえば、AFS は真のグローバル名前空間をクライアントに提供し、すべてのファイルの名前がすべてのクライアント・マシンで同じになります。対照的に NFS では、各クライアントが NFS サーバーを好みの方法でマウントすることができます。したがって、慣例(および管理上の努力)によってのみ、クライアント間で同様のファイル名が付けられます。

AFS はまた、セキュリティを真剣に受け入れ、ユーザーを認証するメカニズムを組み込み、ユーザーが望む

場合に一連のファイルを秘密に保つことができるようになりました。NFS はこれとは対照的に、長年にわたりセキュリティはきわめて原始的なサポートを持っていました。

AFS には、柔軟なユーザー管理アクセス制御のための機能も含まれています。したがって、AFS を使用する場合、ユーザーは誰がどのファイルに正確にアクセスできるかを非常に制御できます。NFS は、ほとんどの UNIX ファイルシステムと同様に、この種の共有のサポートをまったくしません。

最後に、前述したように AFS はツールを追加して、システム管理者のサーバー管理を簡素化します。システム管理について考えてみると、AFS は遙か先をいっていました。

50.9 Summary

AFS は、NFS で見たものとはまったく異なる分散ファイルシステムの構築方法を示しています。AFS のプロトコル設計は特に重要です。(ファイル全体のキャッシュとコールバックを通じて) サーバーのやりとりを最小限に抑えることで、各サーバーは多くのクライアントをサポートし、特定のサイトを管理するために必要なサーバーの数を減らすことができます。単一の名前空間、セキュリティ、アクセス制御リストを含む他の多くの機能は、AFS を使用するうえできわめてうれしいものです。AFS によって提供される一貫性モデルは、理解しやすく、理由を説明するのが簡単であり、時々 NFS で観測されるような時折変わった動作につながることはありません。

おそらく残念ながら、AFS は減少傾向にあるでしょう。NFS はオープンスタンダードとなつたため、多くの異なるベンダーがそれをサポートし、CIFS(Windows ベースの分散ファイルシステムプロトコル)とともに NFS が市場を支配していました。ウィスコンシンを含む様々な教育機関のように、AFS のインストールは隨時見られますが、唯一の影響は実際のシステムそのものではなく、AFS のアイデアによるものです。実際、NFSv4 は、サーバ状態(例えば、「オープン」プロトコルメッセージ)を追加するので、基本的な AFS プロトコルと似ているようなものが増えてきています。

参考文献

[B+91] “Measurements of a Distributed File System”

Mary Baker, John Hartman, Martin Kupfer, Ken Shirriff, John Ousterhout

SOSP '91, Pacific Grove, California, October 1991

An early paper measuring how people use distributed file systems. Matches much of the intuition found in AFS.

[H+11] “A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications”

Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

SOSP '11, New York, New York, October 2011

Our own paper studying the behavior of Apple Desktop workloads; turns out they are a bit different than many of the server-based workloads the systems research community usually focuses upon. Also a good recent reference which points to a lot of related work.

[H+88] “Scale and Performance in a Distributed File System”

John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, Michael J. West

ACM Transactions on Computing Systems (ACM TOCS), page 51-81, Volume 6, Number 1, February 1988

The long journal version of the famous AFS system, still in use in a number of places throughout the world, and also probably the earliest clear thinking on how to build distributed file systems. A wonderful combination of the science of measurement and principled engineering.

[R+00] “A Comparison of File System Workloads”

Drew Roselli, Jacob R. Lorch, Thomas E. Anderson

USENIX ’00, San Diego, California, June 2000

A more recent set of traces as compared to the Baker paper [B+91], with some interesting twists.

[S+85] “The ITC Distributed File System: Principles and Design”

M. Satyanarayanan, J.H. Howard, D.A. Nichols, R.N. Sidebotham, A. Spector, M.J. West

SOSP ’85, Orcas Island, Washington, December 1985

The older paper about a distributed file system. Much of the basic design of AFS is in place in this older system, but not the improvements for scale. The name change to “Andrew” is an homage to two people both named Andrew, Andrew Carnegie and Andrew Mellon. These two rich dudes started the Carnegie Institute of Technology and the Mellon Institute of Industrial Research, respectively, which eventually merged to become what is now known as Carnegie Mellon University.

[V99] “File system usage in Windows NT 4.0”

Werner Vogels

SOSP ’99, Kiawah Island Resort, South Carolina, December 1999

A cool study of Windows workloads, which are inherently different than many of the UNIX-based studies that had previously been done.