平成25年度オペレーティングシステム期末試験 解答例, 講評

2014年1月28日 (火) 実施

解答例:

所属学科	学生証番号	氏名

1

(1) O(n) (2) readシステムコールでファイルを全て読み込んでいるからね. その量に比例する時間がかかるのよ.

(3) 多くのオペレータさんがみんとちゃんのプログラムを同時に起動すると, それぞれがファイルの大きさと同じだけの物理メモリを消費して, その合計が物理メモリ量を上回ることがあるからよ. そうするとスラッシングを起こしてとてつもなく時間がかかることになるわ. 私達の今の実験では同時にいくつも起動してないから, そういうことはおこらないってわけ.

(4) mmap (5) access_record * A = mmap(NULL, sz, PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, 0);

(6) mmapを使うと, 多くのオペレータさんがみんとちゃんのプログラムを同時に起動しても, ファイルを読み込むための物理メモリは共有されるのよ. だから多数同時に起動されてもmallocを使った時みたいにスラッシングが起こることはないのよ.

(7) mmapでファイル全体をマップしても, データはその時に読み込まれるわけじゃないのよ. 実際に触ったページに対してのみページフォルトが起きて, その際に読み込まれていくのよ. 2分探索みたいにデータの一部にしか触らないアルゴリズムは, 実際に触ったデータ量, O(log n)に比例したデータしか読み込まれない. だからO(log n)になるってわけ

2

(1)
実装の概要
forkシステムコールは呼び出したプロセスのアドレス空間を複製し, プロセスidとforkの返り値が異なる以外は同一の, 2つのプロセスを作る.
その際, コピーは物理メモリを複製するのではなく, 親と子で同一の物理メモリを共有し, どちらかが書き込みを行った時点で実際に異なる物理メモリをわりあてる, copy-on-writeによる複製をおこなう.

MMUの役割
copy-on-write (書き換えがあった時のみ実際の物理メモリのコピーを作る)で複製することを可能にするために, 書き込み禁止となっているページに対するページへ書き込みが起きた際, 例外を起こして書き込みをOSに通知する役割

(2)
実装の概要
一定期間ごとに, 最近アクセスされたページを記録し, ページ置換時に長時間アクセスされていないページを, 置換の対象とする

MMUの役割
MMUにはページが読まれた, ないし書きこまれた際に reference bit, dirty bitをセットする機能がある. MMUはこれにより, 最近アクセスされたページをOSに情報提供する役割を果たす. OSはこれを一定期間ごとにこれらをクリアしつつ, 何世代かにわたりそれらの値を保存する. ページ置換時にそれらの記録を参照することで, 長期間にわたり使われていないページを特定できる. これによって, LRUの近似が実現できる.

(3)
実装の概要
共有メモリを確立したい論理アドレスの範囲に対して同じ物理メモリを割り当てる. そのためにOSはページテーブル内でそれらの論理アドレスの変換先が同じになるようにセットしておく

MMUの役割
OSが指定したページテーブルに従って論理アドレスを物理アドレスに変換する

3

(1)

(a) 生じ得る (生じ得ない)　　生じうる場合その実行例

(理由の記述は要求していないが参考までに) is_prime_x(p, n_primes) という関数自身に 擬陽性が存在しなければ, 素数でないものがprimes_ に書き込まれることが無いことは明らか.
is_prime_x(p, n_primes) の中身が示されていないため, そこを適宜補わない限り回答不能な問題. 採点方針については講評を参照.

(b) (生じ得る) 生じ得ない　　生じうる場合その実行例

スレッドA, Bがほぼ同時に素数をprimesに追加しようとすると, どちらかの追加が失われる可能性がある.
具体的には, スレッドA が n_primes から0 を読む (7行目); スレッドBもn_primes から0を読む(7行目); それぞれが primes[0] に素数を書き込む

(c) 生じ得る (生じ得ない)　　生じうる場合その実行例

(理由の記述は要求していないが参考までに) is_prime_x(p, n_primes) という関数自身が終了することが保証されており,かつ next_p を減少させるような変更を行わなければ, while (next_p < n) の条件はやがて成り立たなくなり, 各 worker が終了することは明らか.
is_prime_x(p, n_primes) の中身が示されていないため, そこを適宜補わない限り回答不能な問題. 採点方針については講評を参照.

(2)

```
long fetch_and_incr(long * p, pthread_mutex_t * l) {
  pthread_mutex_lock(l);
  long x = *p;
  *p = x + 1;
  pthread_mutex_unlock(l);
  return x;
}
```

左のように排他制御を行った上で1をたす関数を作っておき, 右のようにnext_p, n_primes それぞれを守るロックを用意する.

```
pthread_mutex_t next_p_lock;
pthread_mutex_t n_primes_lock;
```

その上でworker内のnext_p, n_primesの更新部分を以下のように書き換える

5行目 long p = next_p++;
--> long p = fetch_and_incr(&next_p, &next_p_lock);

7行目 long idx = n_primes++;
--> long idx = fetch_and_incr(&n_primes, &n_primes_lock);

(3)

```
long fetch_and_incr(volatile long * p) {
 while (1) {
   long x = *p;
   if (cmp_and_swap(p, x, x + 1)) return x;
 }
}
```

左のように共有メモリを排他的に更新する.
それ以外は(2)の回答と同じ.

(4)

(a) 生じ得る (生じ得ない)　　生じうる場合その実行例

ある数pの素数を判定するには, primes_ 配列に, sqrt(p)以下の素数がすべて格納されていなくてはならないが, 複数のスレッドが小さい数から大きい数まで並行して判定するのでこれが成り立たない.
具体的には, n_primes = 0 の状態で, is_prime(4)が呼び出されると誤って, 4が素数であると判定される.

(b) 生じ得る (生じ得ない)　　生じうる場合その実行例

(理由の記述は要求していないが参考までに) n_primes の更新を不可分に行うよう修正したため, is_primeで素数と判定された数がprimes_ に書き込まれない心配はなくなった. 偽陰性が生ずるとしたら, is_prime自身が素数を素数でないと判定する以外ありえないが, xが素数ならば, xは1またはx以外では割り切れないのだから, (primes_に1またはxが含まれない限り)is_primeが0を返すことはない. primes_ に1が入ることが無いことは, next_p の作り方からただちにわかる. またprimes_にx自身が含まれていたら, そもそもxが偽陰性になることはない. 以上より偽陰性は生じないことがわかる.

(c) 生じ得る (生じ得ない)　　生じうる場合その実行例

(理由の記述は要求していないが参考までに) n_primesの値がプログラム全体を通じて n 以下であることは明からなので, is_primeの呼び出しは必ず終了する. next_p もスレッドが動き続ける限り増加し続けるのでいつかはnに達する. よってすべてのworkerはいずれ終了する.

(5)

いくつか考えられるが例えば, n までの素数を求めるのに, 一旦 sqrt(n)までの素数を求め, それが終わった所で昇順に整列させた上で, sqrt(n) - n までを, 並列に判定する. sqrt(n)までの素数を求める部分でも再帰的に同じ事を行う. コードの概要としては,

```
long find_primes_rec(long n) {
  if (n <= 2) return 0;
  else {
    long s = isqrt(n);
    long p;  long ns = find_primes_rec(s);
    // 以下のfor文を並列に (OpenMPやpthreadを使う; 詳細省略)
    for (p = s; p < n; p++)
      if (is_prime(p, ns)) {
        long idx = fetch_and_incr(&n_primes);
        primes[idx] = p;
      }
    qsort(&primes[ns], n_primes - ns, sizeof(long), compare_long);
    return n_primes;
  }
}
```

```
long find_primes_para(long n_, long * primes_) {
  primes = primes_;
  n_primes = 0;
  return find_primes_rec(n_);
}
```

必須ではないが, is_primeは検査すべき素数の個数を明示的に受け取るようにする

```
int is_prime(long p, long np) {
  long i;
  for (i = 0; i < np; i++) {
    if (p % primes[i] == 0) return 0;
    if (primes[i] * primes[i] > p) return 1;
  }
  return 1;
}
```

配点: 受験者 92 人. 括弧内は, 受験者 92 人中の正解者数.

1

(1) 4 点 (72)

(2) 6 点 (54)

(3) 6 点 (50)

(4) 6 点 (81)

(5) 6 点 (52)

(6) 6 点 (50)

(7) 6 点 (51)

2

(1) 概要 5 点 (70), MMU の役割 5 点 (34)

(2) 概要 5 点 (58), MMU の役割 5 点 (13)

(3) 概要 5 点 (31), MMU の役割 5 点 (11)

3

(1) (a) 起き得ない 1 点 (63), (b) 起きうる 1 点 (87), (b) 実行例 6 点 (38), (c) 起き得ない 1 点 (79)

(2) 5 点 (29)

(3) 5 点 (7)

(4) (a) 起き得る 1 点 (50), 実行例 6 点 (34), (b) 起き得ない 1 点 (65), (c) 起き得ない 1 点 (74),

(5) 4 点 (3)

100 点満点. 最高点 98 点. 最低点 0 点.

講評:

1

全ての問題で正答率は半分以上. 92 人の受験者のうち, 授業にいた人はどう見ても半分以下だから, さすがに授業にいた人でこの問題ができないという人はほとんどいなかったものと期待したい.

(3) ポイントとなるのは次の 2 点.

- read を用いると, ファイルのサイズ × プロセスの物理メモリが要求される
- そのサイズが, マシンの搭載物理メモリ量を超えると, スラッシング (大量のページフォルト) が発生する

採点はそれぞれについて書かれているかを判定している. 片方だけ書かれている場合, 点数は半分.

(5) 採点の際, mmap の引数の細かい点は見ていない. 以下の 2 点を見ている.

- まずファイルを open して, その返り値を mmap へ渡していること
- mmap の返り値がアドレスであり, それが配列 A になっていること

それぞれについて半分ずつの点を与えている.

配列 A を別途 malloc で割り当ててそれを mmap へ渡しているような答案があったが, それは mmap がメモリ割り当ての役割も果たしているという点を理解していないと思われる.

(6) ポイントは以下の 2 点

- mmap で同じ領域をマップして得られた領域は (書き込みが起きない限り) 物理メモリを共有している
- したがって mmap を使った場合, 多数のプロセスを同時に立ち上げても, 要求物理メモリ量が, 搭載物理メモリ量を上回ることがない

それぞれの点について書けていれば半分ずつの得点. なお, 2 点目については, (3) の 2 点目が書かれていれば, わかっているものとして, ここで明示的に書かれていなくても不問にしている.

なお, この問題の答えとして,

- mmap では実際にアクセスした所しか物理メモリを消費しない
- したがってスラッシングが生じない

という答えを書いた人もいた. こちらも正解としている.

2

この問題は, OS がある機能をどう実現しているかということと, その中で, MMU というハードウェアがどういう役割を果たしているか, を区別して訪ねているのだが, 後者についてまともな解答は少なかった.

(1) fork はアドレス空間を複製するシステムコールである. OS はそれを copy-on-write という仕組みで高速化している. copy-on-write とは, アドレス空間を構成するページの複製を, 物理ページの複製を書き込みが起きるまで遅延させる方法である. OS は, (論理的に) 複製されたページに対して「書き込み不可」の設定をして, 書き込みがあったらその際にあがる例外を補足して, その時点でページをコピーする.

MMU はその中で, 「書き込みを検出する (書きこみ発生時に例外をあげる)」という役割を果たしている.

まるで, MMU 自身がページをコピーするかのような誤解が多かった. 一般的なセンスとして, ハードウェアが直接提供する機能が, そこまで複雑なことは稀である, というセンスがあって良い.

また, 採点では大目に見ているが, 気になった勘違いとして, 書き込み不可に設定されるのは子プロセスの側 (のみ) である, という書き方が多かった. copy-on-write を実現するには親と子の, 両者からの書き込みを検出して, **どちらか一方でも**書きこんだら物理メモリを複製する必要がある. これは目的を考えれば当然のことなのだが, なぜか, 子プロセスによる書き込みだけを検出すれば良いと勘違いしている書き方が目立った.

(2) 実現例として以下がある. OS は, 一定時間 (T とする) ごとに各ページへ最近 T 時間内のアクセスがあったかどうかを, reference bit, dirty bit によって調べる. その記録を何世代かにわたって保存し, 各ページが最後にアクセスされた時刻の, (T 程度の誤差を含む) 近似値を把握する.

MMU は, ページへの読み出し時に reference bit, 書き込み時に dirty bit をセットし, 最近アクセスがあったかどうか, OS に情報提供する役割を果たす.

ここでも MMU 自身がページの入れ替えを行うかのような解答が多かったが, そこまで複雑な (それ自身が I/O を伴うような) ことは, ソフトにやらせるのが普通である.

(3) OS は, 共有を指定された論理アドレスに対応する物理アドレスを, プロセス間で同じものに設定する. それによってプロセス間のメモリの共有が実現される.

MMU は, OS がページテーブルにセットしたアドレス変換情報に従って論理アドレスを物理アドレスに変換する役割を果たす.

3

(1) まず, `is_prime_x` というタイポについてお詫びします. 何か特別な処置が必要かを答案を見ながら考えましたが, 結局以下の通りにしています. まず, 「起き得る・起き得ない」という 2 択の問題は配点が 1 点としました. これは特別な処置と言うよりも, 単なる 2 択であるという点, および「起き得ない」と答えたらそれ以上何も答えることが無いので, わからなければこちらを選択する人がどうせ多いだろうということで, ここに大きな点を配分する意味がない, という常識的な判断です. 結局, 「起きうる」場合の理由を正しく答えられるかどうかが実質的な問題となりますが, これについては, is_prime_x が正しくても生じうる間違いを考えてもらうよう, 期待するのはひどいことではないでしょうということ, そして, 実際そのように考えている人が数多くいたと言うことで, その前提で採点をしています.

なお, 試験直後にホームページで, (a) の間違い (擬陽性: 素数でない数を誤って素数と判定すること) が生ずると書いてしまいましたが, 誤りです. 解答にあるとおり, このプログラムではそれは生じません.

多かった勘違い: 以下の

```
long p = next_p++;
```

という文を 2 つのスレッドがきわどいタイミングで実行すると, ある数が「飛ばされる」事がありうるという勘違いが非常に多かった. 「飛ばされる」という意味は, どのスレッドも, ある値—例えば 5—を next_p から読み出すことがない, したがって 5 が素数であるにも関わらず判定に漏れてしまう, ということである.

このプログラムは間違いですが, そのようなことは生じません. 実際個々のスレッドは, next_p から読みだした数に 1 を足した数を書いているわけですから, 直感的に考えても, next_p の数が重複したり, 戻ったりすることはあっても, ある数が一度も読まれずに「飛ばされる」ことはありません. もう少し厳密に議論すれば以下のようになります. 以下が補題として成り立ちます.

補題 1 ある時点であるスレッドが*,* next_p から最後に読みだした値が x であるとき*,* x **以下の値は全て***,* 最低一度は next_p から読み出されている*.*

帰納法で証明する.

- $x = 2$ の時に成り立つことはあきらか.
- $x \leq k$ で成り立つとすると, $x = k+1$ でも両者が成り立つことを示そう. あるスレッドが next_p から最後に読みだした値が, $(k+1)$ だっとする. すると, どれかのスレッドが next_p に $(k+1)$ を書き込んだことになる. プログラムより, $(k+1)$ を書き込んだスレッドは, next_p から k を読みだしたはずである. 従って, 帰納法の仮定より, k 以下の値は全て一度は読み出されている. 結局 $(k+1)$ 以下の値が全て読まれていることになる.

直感的には当たり前すぎて, 証明しろと言われると却って困ってしまうような命題だが, 証明するならばこういう事になる.

(2) 正解は, next_p++, n_primes++ それぞれが排他的に行われるようにするというもの.

実を言うとこの問題の正解としては, next_p++の方は排他的にしなくても構わない. next_p++を排他的に行わないと, 同じ素数が 2 度入ることは生ずるかも知れない. 常識的に考えればこれも避けるべき事態だが, (1) で述べた問題が生じないという意味では問題はない. したがってここでは書いても書かなくても良いとしている.

なお, 「同じ素数が 2 度入る」事の問題点として, 実践的には, primes 配列の大きさがどれだけ必要かがわかりにくくなるという問題点がある. 最悪のケースを考えると, スレッド数倍だけ要素数が必要になる.

多かった勘違い: 勘違いと言い切ることはできないかもしれないが, 排他制御を while 文全体に施す (つまり, while 文の前に lock(), while 文を抜けた後に unlock()) とか, while 文の各 iteration 全体に施すなどの答案が多かった.

これは, たしかに問題は解決しているが, 「元も子もない」解決方法である. この並列プログラムは, 素数の判定 (is_prime) を並列に行なって初めて意味のあるもので, いくら排他的にやれば良いと行っても, こんなことをしてしまったらそれは最初から逐次プログラムでよいということになる.

いくらなんでもこれは正解とはできない.

(3) あまりにも正解率が低かった. compare-and-swap の説明を, 授業のスライドなどで見返しておいて欲しい.

(4) (a) に関して, `primes` 配列に, 素数が「足りない」状態で, 判定が行われてしまうのが問題, というのは, 「起き得る」と答えた人の多くが正解できていた.

(5) 時間も足りないかも知れないが, この問題にきちんと答えられた人はいなかった. 要は, 「$\sqrt{p}$ までの素数が primes 配列に格納されてから」p の判定を行う, ということである. 単純には, まずある数 a までの素数を (並列に) 判定し, それが全て終わったら a^2 までの素数を (並列に) 判定し, それが全て終わったら a^4 までの素数を (並列に) 判定し, ... と言う具合にやることである. 解答にあるのはそれを再帰的に書いているだけのことである (N までの判定をするのに, まず $\sqrt{N}$ までの判定をしてから, 残りを並列に判定する). ついでに, 各段階で primes 配列を昇順に並べ変えてしまえば, (4) の `is_prime` がやっている,

```
1  if (primes[i] * primes[i] > p) return 1;
```

という判定終了方法も問題ではなくなる.

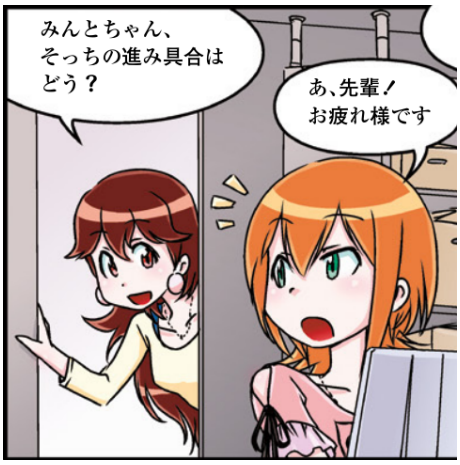
それが以外のやり方も考えられるとは思うが, primes 配列には, 素数でないと判定された数は入らないので, ある数以下の判定が (素数であったにせよなかったにせよ) 終わっていることを primes 配列だけから知るのは困難である.

例えば, A という変数を用意して, 「A 以下の数は既に判定された」という意味をもたせるとしよう. こうすると, p を判定しているスレッドは, $A * A > p$ を確認すればよいことになる. しかし, A の更新をどう行うかが問題となる. たとえば, $p = 101$ の判定を終えたスレッドが A を 101 にして良いのかというとそうではない. まだ 100 は判定中かも知れないからである. 一方でこのスレッドが, $A = 100$ になるまで待ってから, $A = 101$ としたのでは, 並列実行の効果はほとんどなくなってしまう.

1

次の会話を読んで後の, (1) から (7) の問いに答えよ (イラスト: Piro. シス管系女子より. 問題の内容とは関係ありません).

みんとちゃん (以下 M): う〜ん, おかしいなぁ.



大野桜子先輩 (以下 O): みんとちゃん, どうしたの?

M: あ, 大野先輩! お疲れ様です. これ, 私が書いた, ウェブサーバへのアクセスのログの解析をするプログラムなんですけど, これを使って現場で解析をしているオペレータさんたちから「遅すぎる!」って苦情が出て, へこんでるんです〜.

O: どれどれ, 見せてみて.

M: はい, 日付と時刻を 2 つ与えて, その間にあったアクセスの件数を返すんです.

O: てことはどれどれ, 例えば, 去年の紅白の時間帯だったら (カタカタ...),

```
$ count_accesses 2013-12-31-18-00-00 2013-12-31-23-45-00
385
```

385 件ってなるわけね.

M: はい. で, 今みたいに私のマシンでこのコマンドを実行すると, 大概の場合, 1 秒もせずに答えが帰ってくるんです. でも, オペレータたちが言うには, 20 秒以上かかることもざらだって.

O: ふーん, データはどこにあって, どのくらいの大きさで, どんな形式で格納されてるの?

M: はい, まず私の開発用のマシンと, オペレータたちが使うマシンとがあって, 両方に同じものが置かれています. マシンの種類は同じです. メモリはどちらも 2GB ほど搭載しています.

O: データの形式は?

M: 一個のアクセスにつき, アクセスの時刻を表す `seconds_since_epoch` っていうフィールドと, その他の情報が格納されたレコードがあって, 一個のレコードはぴったり 256 バイトになっています. ウェブサーバの生のログをこの形式に変換するプログラムを別に作って, それを使ってこの形式に変換しています. `seconds_since_epoch` は, ある基準の時点からの秒数です.

```
typedef struct {
  long seconds_since_epoch;
    ... ;                    // 以降，詳細省略
} access_record;
```

O: ファイルにはこの形式のレコードが並んでいるわけね. レコードの数はどのくらい?

M: 200 万くらいです. で, レコードは**時刻の昇順**にならんでいます. なので, ファイルを配列に読み込んだら, 「2 分探索」で $O(\log n)$ で検索しています. あ, n はレコードの数です. こんな感じです.

```
int main(int argc, char ** argv) {
  long t0 = convert_to_abstime(argv[1]);
  long t1 = convert_to_abstime(argv[2]);
  long n = n_records_in_file("records");
  int fd = open("records", O_RDONLY);
  long sz = sizeof(access_record) * n;
  access_record * A = (access_record *)malloc(sz);
  long r = read(fd, A, sz);
  long n_founds = binary_search(A, n, t0, t1);
  printf("%ld\n", n_founds);
  return 0;
}
```

注: `convert_to_abstime` は, 2013-12-31-23-45-00 のような形式の時刻を, 基準点からの秒数に変換する関数, `n_records_in_file` は, ファイルのサイズを調べ, ファイル中レコードの数を求める関数である. ともにそれらにかかる時間はほとんど無視できる.

O: じゃ, まずはレコードの数を色々変えて測ってみましょ.

M: ... $O(\log n)$ のアルゴリズムなんだから, 爆速のはずです...

みんとは, 横軸に検索対象のレコード数 (上記プログラムの `n`), 縦軸にプログラムが起動してから終了するまでの時間をとって, グラフを書いた. `n` は最大で, 200 万くらいまで増やした. すると, ...

M: じぇじぇじぇ, ぜんぜん $O(\log n)$っぽくない!

O: ね, まずはちゃんと測定してみないとね. これはどちらかというと $O(\log n)$ じゃなくて, [(1)] のグラフだね. で, そうなっている理由は, [(2)]

M: 言われてみれば当たり前ですね... で, でも, n が 200 万でも, 実際にかかっている時間は, 250 ミリ秒くらいです. オペレータさんたちが言うような, 20 秒とか, 全然そんなんじゃありません. やっぱりこれって, 他に原因があると思うんですけど...

O: そうね. オペレータさんたちは, どんなふうにこのプログラムを利用しているの?

M: はい, 10 人くらいのオペレータさんが使っています. データが置かれているマシンにウェブサーバをたてて, ウェブページ経由で, 日付と時刻を入力します. そうするとそれを受け取ったウェブサーバが私のプログラムを起動するという仕組みです.

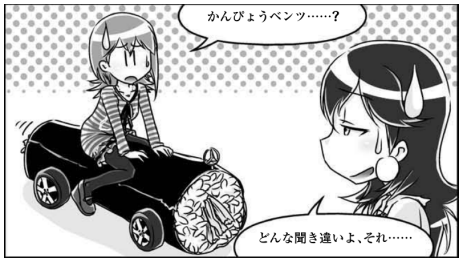
O: へぇ, ってことは, どういうタイミングでいくつ, みんとちゃんの書いたプログラムが起動されるかわからないってことね. だったら, 「20 秒かかることもざら」っていうのも, わかる気がするわ.

M: え, どうしてですか?

O: (3)

M: そ, そうなんですか. 先輩, 私はどうしたらいいんでしょう?

O: 今回みたいなケースに, 実に簡単な解決方法があるんだな. (4) を使えばいいよ.

M: エ, エンマ君? ··· 

O: どんな聞き間違いよ, それ··· みんとちゃんのプログラムをこんなふうに書き換えればいいのよ.

```
1  int main(int argc, char ** argv) {
2    long t0 = convert_to_abstime(argv[1]);
3    long t1 = convert_to_abstime(argv[2]);
4    long n = n_records_in_file("records");
5    (5)
6    long n_founds = binary_search(A, n, t0, t1);
7    printf("%ld\n", n_founds);
8    return 0;
9  }
```

M: で, でもどうしてこうすると, オペレータさんの環境で遅い問題が解決されるのでしょうか?

O: それは, (6).

M: ふーん, なんかよくわかりませんが, とりあえず現場に送ってみます.

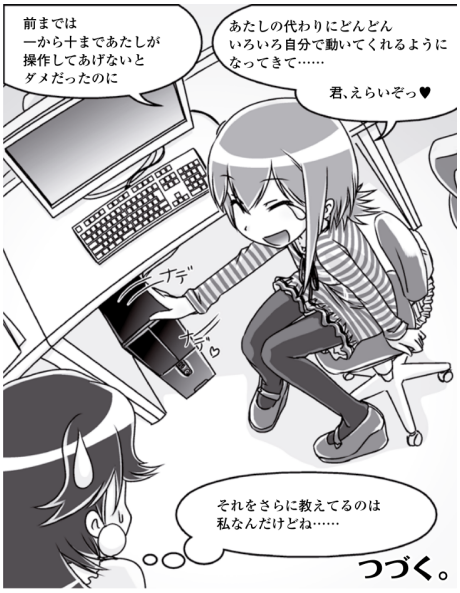
(しばらくして)

M: あ, オペレータさんからメールで反応がかえって来ました. (メールを読みながら) え, すごい, n が 200 万でも, 40 ミリ秒くらいですって! 遅くならないどころか, さっきまでの私のマシンでの結果よりも断然速い!

O: そうそう, (4) を使って書きなおしたやつは, さっきまでの (1) の性能ではなく, 本当にほぼ $O(\log n)$ と言って良い性能になるのよ. それも (4) がファイルを読む仕組みと関係あるのよ. (7).

M: ふーんなるほどね. うまくできてますね. [(4)], えらいぞっ! あ, そうだ, こないだ書いたウェブサーバの生のログを変換するプログラムも, [(4)]を使って書き換えちゃおーっと

O: あ, いや, いつでも使えばいいってもんじゃ無いんだけど... (でも説明してると長くなるから今日はまあいっか ^^;)



(1) [(1)]に当てはまる計算量を答えよ.

(2) [(2)]には, 前問の答えがそのようになる理由が入る. 適切な文章を答えよ.

(3) [(3)]には, みんとちゃんのプログラムが, 「20 秒かかることもざら」になる理由が入る. 適切な文章を答えよ. その際, みんとちゃんと大野先輩の実験ではせいぜい 250 ミリ秒だったのに, オペレータさんたちの環境では 20 秒かかることもざらだった理由がわかるように説明した文章を入れよ.

(4) [(4)]に当てはまる単語を答えよ.

(5) [(5)]に当てはまる適切なプログラムの断片を答えよ. 1 行とは限らない. 呼び出す API の詳細 (引数の順番など) が不安な場合は適宜コメントで補え.

(6) [(6)]には, [(4)]を使うと, 「20 秒かかることもざら」だった問題が解消された理由が入る. 適切な文章を答えよ.

(7) [(7)]には, 大野先輩のプログラムでは, 本当にほぼ $O(\log n)$ の計算量が達成される理由が入る. 適切な文章を答えよ.

2

今日のオペレーティングシステムを構成するにあたって, CPU に備わる MMU は必須の機能である. MMU を用いて実現されている以下の機能について, その実現方法の概要, MMU がそこでどのような役割を果たしているかについて述べよ.

(1) fork() システムコール (プロセス生成) の高速化

(2) LRU を近似したページ置換アルゴリズムの実現

(3) プロセス間共有メモリ

3

以下の find_primes_serial(n, $primes$) は, n 未満の素数を全て求め, 配列 $primes$ に格納し, その個数を返すプログラムである. $primes$ は, n 未満の素数を格納するだけの十分な領域を持っているものとする (配列の添字あふれは気にしなくて良い).

```
1  long find_primes_serial(long n, long * primes) {
2    long n_primes = 0;
3    long next_p = 2;
4    while (next_p < n) {
5      long p = next_p++;
6      if (is_prime(p)) {
7        long idx = n_primes++;
8        primes[idx] = p;
9      }
10   }
11   return n_primes;
12 }
```

ここで 6 行目の is_prime(p) は, $\sqrt{p}$ 以下の整数全てで p を試し割り算して, 素数かどうかを判定する関数で, 例えば以下である.

```
1  int is_prime(long p) {
2    long x;
3    for (x = 2; x * x <= p; x++) {
4      if (p % x == 0) return 0;
5    }
6    return 1;
7  }
```

find_primes_serial を並列化することを考える. 具体的には, 複数のスレッドが並行して, 4 行目から始まる while 文を実行する. スレッド間で仕事や結果を共有するために, n, primes, n_primes, next_p は大域変数にする. つまり, 各スレッドが実行する以下の関数 (worker) を作り,

```
1  long n, n_primes, next_p;
2  long * primes;
3  void * worker(void * _) {
4    while (next_p < n) {
5      long p = next_p++;
6      if (is_prime_x(p, n_primes)) {
7        long idx = n_primes++;
8        primes[idx] = p;
9      }
10   }
11   return 0;
12 }
```

本体は以下のようにする (N_THREADS はスレッドの個数).

```
long find_primes_para(long n_, long * primes_) {
  /* 大域変数でデータを共有 */
  n = n_;
  primes = primes_;
  n_primes = 0;
  next_p = 2;
  /* N_THREADS 個, スレッドを起動 */
  int i;
  pthread_t tid[N_THREADS];
  for (i = 0; i < N_THREADS; i++)
    pthread_create(&tid[i], NULL, worker, 0);
  /* スレッドの終了待ち */
  for (i = 0; i < N_THREADS; i++)
    pthread_join(tid[i], NULL);
  return n_primes;
}
```

以下の問いに答えよ.

(1) この, find_primes_para を実行した結果, 以下のような間違いが生じ得るか否かを答えよ (解答欄の生じ得る, 生じ得ないのいずれかに○をつけよ). 生じ得ると答えた場合, それが生ずる実行例を具体的に示せ.

(a) 偽陽性. つまり, 素数でないものが配列 primes_ に格納された状態で終了する.

(b) 偽陰性. n 未満の素数が, 配列 primes_ に格納されていない状態で終了する.

(c) 終了しない.

(2) (1) で述べた問題点を解消するため, 排他制御を用いた解決方法を示せ. プログラムのどこをどのように直したら良いかはっきりと示せ. 必要ならば適宜変数や関数を追加して良い. ただし, 渡された配列 primes_ に, どのような順番で素数が格納されるかは問わない.

(3) (1) で述べた問題点を解消するため, **排他制御を用いず,** compare&swap 命令を用いた解決方法を示せ. compare&swap 命令が以下の関数を呼び出すことで使用可能であるとし, 必要ならば適宜変数や関数を追加して良い. 渡された配列 primes_ に, どのような順番で素数が格納されるかは問わない.

bool cmp_and_swap(T * p, T a, T b);

ここで, T は, int または long とする. この関数は, 以下に相当する動作を不可分に行う

```
bool cmp_and_swap(T * p, T a, T b) {
  if (*p == a) { *p = b; return 1; }
  else         {         return 0; }
}
```

(4) 上記の is_prime(p) 関数は, 2 以上 $\sqrt{p}$ 以下の全ての数で試し割り算をしているが, 実際には 2 以上 $\sqrt{p}$ 以下の全ての**素数**で試し割り算をすれば十分である. これに注目して, 配列 primes にすでに格納されている素数を使って試し割り算をするよう, 以下のように is_prime を書き換えたとする.

```
1 int is_prime(long p) {
2   long i;
3   for (i = 0; i < n_primes; i++) {
4     if (p % primes[i] == 0) return 0;
5     if (primes[i] * primes[i] > p) return 1;
6   }
7   return 1;
8 }
```

このように変更したプログラムで生じ得る間違いについて, (1) と同様に答えよ. ただし, (2) または (3) の修正はすでに施されているものとする. ここでも, 渡された配列 primes_に, どのような順番で素数が格納されるかは問わない.

(5) find_primes_para を正しくするためにさらに必要な修正について, 概要を述べよ. 渡された配列 primes_に, どのような順番で素数が格納されるかは問わない.

問題は以上である