2019年度オペレーティングシステム期末試験

2020年1月21日 (火)

- **問題は 3 問**
- **この冊子は，表紙 1 ページ (このページ), 問題 2-7 ページからなる**
- 各問題の解答は所定の解答欄に書くこと.
- 昨問には細心の注意をしているが, それでもタイポなどがあるかもしれない. そのようなの疑いを見つけたらおおらかな気持ちで,「もしかしてこの (3) は (2) のことではないのか, (1)〜(3) は (a)〜(c) の間違いなのではないか」などと想像・注釈しつつ答える気持ちが大切である

1

以下の文書を読みその下の問に答えよ.

CPU のデータ読み出し (ロード) 命令を考える. まずロード命令でどのデータを読み出すべきかを指定するのに, プログラムは (a) を用いる. CPU は (a) を (b) に変換してからメモリをアクセスする. このために CPU は, メモリ上におかれた (c) を参照する. この変換を高速化するために CPU 内には (d) があり, (c) 中の情報の一部を複製 (キャッシュ) している.

さて CPU は (c) を参照して, (a) に対する (b) が存在していた場合は, その (b) に対するアクセスを行うが, 存在していない場合は, (e) という例外を発生させる.

このような一連の処理を行う, CPU 内のハードウェアを (f) と呼ぶ.

(e) は OS によって処理される. まずアクセスされた (a) が, そのプロセスがアクセスして良い場所か (プログラムに割当済みの領域であって, 読み出し許可がある場所であるか) どうかを判定 (*) する.

(*) で, アクセスして良い場所ではなかった場合, (g) という (h) を, プロセス (正確にはそこをアクセスしたスレッド) に配達する. 通常それを受け取ったプロセスは終了するが, それ対する (i) を設定した場合はそれが呼び出され, 終了後に処理が継続する.

(*) で, アクセスして良い場所であった場合, 二つの場合が考えられる. ひとつは, 今回のアクセスが, その場所が割り当てられてから初めてのアクセスであった場合で, この場合 OS は (j) を割り当て, その中身を (k) て (e) の処理を終了する (プロセスに処理を戻す). そうでない場合は, (j) を割り当て, その中身を (l) て (e) の処理を終了する (プロセスに処理を戻す). 同じ (e) でも前者と後者では処理時間に大きな違いがあるため, 名前でも区別されている. Unix では前者を, (m), 後者を (n) と呼ぶ. プログラムがよく参照している領域が物理メモリに収まらないと, (n) が頻発する. このような状況を (o) と呼ぶ.

OS はプロセスからメモリ割り当てを要求された時, 通常その時点では (j) を割り当てず, 上記のような仕組みで, 初めてアクセスが起きたときに割り当てる. このような OS のメモリ管理の方式を (p) と呼ぶ.

(1) (a)〜(p) の空欄に当てはまる語句や言葉を書け.

(2) OS は, 上記で述べた仕組みをどのように利用して, コンピュータをより安全にしているか? 簡潔に説明せよ.

(3) OS が同じ仕組みを利用して, コンピュータをより高機能に (この仕組みなしでは実現が困難な機能を実現) したり, または効率的にして (この仕組みなしでは処理が遅くなったり大量にメモリを消費したりする事態を防いで) いる例をひとつ示せ.

2

N 個のスレッドが実行しているとする. それらのスレッド間の「バリア同期」とは, 「それらのスレッドすべてがある地点に到達するまで待つ」というタイプの同期である. ここではそれがどのように実現できるかを検討する.

具体的には以下の API を持つ.

```
typedef struct { ... } barrier_t;
void barrier_init(barrier_t * b, long N);
void barrier(barrier_t * b);
```

- barrier_t はバリア同期のための構造体
- barrier_init(b, N) は b を初期化する
- その後, N 個のスレッドが barrier(b) が呼び出す. 全部 (N 個) のスレッドがそれを呼び出すまで, 全スレッドの barrier(b) がリターンせずに待つ.

以下の問いに答えよ. なお, (1), (2) までは, 各スレッドは barrier(b) を一度しか呼ばない (ひとつの b で行うバリア同期は一回だけ) と仮定して良い.

(1) barrier_t 構造体の中に, barrier(b) を呼出したスレッドの個数を数えるフィールド x を用意し, それが N になるまで待つ」という方針で以下のようなコードを書いた.

構造体 barrier_t の定義:

```
typedef struct {
  volatile long x;
  volatile long next;
  long N;
} barrier_t;
```

初期化関数 barrier_init の定義:

```
void barrier_init(barrier_t * b, long N) {
  b->x = 0;
  b->next = N;
  b->N = N;
}
```

同期関数 barrier の定義:

```
void barrier(barrier_t * b) {
  long next = b->next;
  long x = b->x;
  b->x = x + 1;
  while (b->x < next) /* 何もしない */  ;
}
```

この元で N 個のスレッドが以下の通り実行する.

```
int main(int argc, char ** argv) {
  int N = omp_get_max_threads(); /* スレッド数を得る */
  barrier_t b[1];
  barrier_init(b, N);
  /* 動作チェック用配列 */
```

```
  long a[N];
  for (long j = 0; j < N; j++) a[j] = -1;
#pragma omp parallel
  {
    int idx = omp_get_thread_num();
    /* 繰り返しなし */
    a[idx] = idx;               /* 動作チェック用 */
    barrier(b);
    for (long j = 0; j < N; j++) {
      assert(a[j] == j);        /* 動作チェック */
    }
  }
  printf("OK\n");
  return 0;
}
```

なお,

- 2 行目の `omp_get_max_threads()` は 9〜16 行目の並列領域を実行するスレッド数 (つまり N) を返す
- 10 行目の `omp_get_thread_num()` は呼び出したスレッドの ID (0 以上 N 未満) を返す
- 配列 `a` はバリア同期が正しく動作しているかの検査用で, 11 行目で書いた値が 14 行目で正しく読まれる (つまり, どのスレッドの 14 行目も, どのスレッドの 11 行目より後に実行されている) ことを確かめる

このコードで , 以下の (a), (b), (c) 3 つの問題が生じ得るかそれとも生じ得ないか, それぞれ答えよ. 生じ得る場合, それが生ずる理由を, 実行例を具体的に示しながら述べよ.

(a) まだ `barrier(b)` を呼んでいないスレッドがいるにもかかわらず, `barrier(b)` がリターンしてしまう

(b) 全員が `barrier(b)` を呼んだにもかかわらず, 決して `barrier(b)` からリターンしない

(c) 運よく (b) の問題がおきない場合でも, スレッド数が多いと非常に性能が悪くなる

(2) このコードに修正を加え , (a), (b), (c) どの問題もおきないよう, 排他制御, 条件変数などを用いてプログラムを修正せよ. 前述したとおり, `barrier(b)` は各スレッドがプログラム中でただ一度だけ呼ぶものと仮定してよい .

(3) それぞれのスレッドが (一度だけ `barrier_init` で初期化したあと), 何度でも `barrier(b)` を呼んでも良いように, `barrier` 関数を変更せよ. 具体的には以下のコードが正しく実行されるようにする. なぜその変更で正しく動作するのかの簡単な説明も記せ.

```
int main(int argc, char ** argv) {
  /* 繰り返す場合 */
  long n = (argc > 1 ? atol(argv[1]) : 10);
  int N = omp_get_max_threads(); /* スレッド数を得る */
  barrier_t b[1];
  barrier_init(b, N);
  /* 動作チェック用配列 */
  long a[N];
  for (long j = 0; j < N; j++) a[j] = -1;
#pragma omp parallel
  {
    int idx = omp_get_thread_num();
    /* 繰り返す場合 */
    for (long i = 0; i < n; i++) {
      a[idx] = idx + i * N;     /* 動作チェック用 */
```

```
16        barrier(b);
17        for (long j = 0; j < N; j++) {
18          assert(a[j] == j + i * N); /* 動作チェック */
19        }
20        barrier(b);
21      }
22    }
23    printf("OK\n");
24    return 0;
25  }
```

3

以下の, 配列の 2 分探索を行うプログラム binsearch(a, n, k) を考える. これは, record 型の要素 n 個を, キーの昇順に整列された状態で保持している配列 a と , 探索したいキーの値 k が与えられ, k をキーに持つレコードを探索するアルゴリズムである. また, a 中の要素のキーはすべて異なるとする.

正確には , binsearch(a, n, k) は以下の値を返す.

$$\texttt{binsearch}(a,n,k) = \begin{cases} -1 & (k < a[0].\texttt{key} \text{ のとき}) \\ n-1 & (k \geq a[n-1].\texttt{key} \text{ のとき}) \\ a[x].\texttt{key} \leq k < a[x+1].\texttt{key} \text{ を満たす } x & (\text{上記以外のとき}) \end{cases}$$

```
long binsearch(record * a, long n, long key) {
  long l = 0;
  long r = n;
  while (l + 1 < r) {
    long c = (l + r) / 2;
    if (a[c].key <= key) {
      l = c;
    } else {
      r = c;
    }
  }
  if (a[l].key <= key) {
    return l;
  } else {
    return -1;
  }
}
```

簡単のため, record 一要素の大きさは仮想記憶の一ページ分の大きさ (P とする) になっているとする. それ以外の詳細は重要ではないが, record の定義は以下のようなものである.

```
typedef struct {
  long key;
  char data[P-sizeof(long)];
} record;
```

以下の問に答えよ.

(1) binsearch(a, n, k) 一回の実行において, a 中のいくつの要素がアクセスされるか?

今, 0 以上 M 未満の相異なる整数を $n(< M)$ 個, 一様な確率でランダムに生成し, それらをキーとして, キーの昇順に整列された record の配列 a を作り, その配列をファイルに格納した.

指定されたファイル中の先頭 n 要素の範囲から, 指定されたキー key を探索する以下のプログラム binsearch_file を考える .

```
long binsearch_file(char * filename, long n, long key) {
  int fd = open(filename, O_RDONLY);
  long sz = n * sizeof(record);
  record * R = malloc(sz);
```

```
5   read(fd, R, sz);
6   long x = binsearch(R, n, key);
7   free(R);
8   close(fd);
9   return x;
10 }
```

(2) 横軸を配列 a の要素数 n, 縦軸を `binsearch_file`(*filename*, n, k) 一回の実行にかかる時間としたグラフを描き, なぜそうなるのかの簡単な説明を加えよ. ただし,

- 実行前, ファイルキャッシュは空の状態であるとする
- 実行したコンピュータは約 256MB ほどの主記憶を搭載していて, ほとんど使われていない (空き領域) 状態で実行した
- 横軸は, a の大きさが主記憶より十分大きくなるところ (384MB 程度) まで描くこと

(3) `binsearch_file`(*filename*, n, k) を, `read` 関数を使う代わりに `mmap` 関数を使ったものに書き換えよ. 関数名を `binsearch_file_mmap` とする. 解答欄のプログラムの不要なところを線で消して, 必要な行を書き足せ.

(4) `binsearch_file_mmap` に対して, (2) と同じ測定を行ったときのグラフを描け. ここでも実行前, ファイルキャッシュは空の状態であるとする.

(5) `binsearch_file_mmap` と `binsearch_file` の違いがわかるよう, (2), (4) の答えの両方を同じグラフに書け.

問題は以上である

配点・解答例・解説・補足

- ※ 採点した解答用紙を事務で受け取れるようにしています
- ただし, 各問について◯か△かなどだけが書いており, 点数などは書いていません. 点数を知りたければ以下の配点を見て計算してください (△の点数はだいたい配点 $\times 0.5$ のことが多いですが, 0.8 のときもあります).
- 満点は 100 点. レポート (60 点程度) と総合して成績をつけています.
- 試験の最高点は 95 点, 最低点は 12 点, 平均点は 63.46 点

学生証番号		氏名		

1	(1)	(a)	論理アドレス	(b)	物理アドレス	(c)	ページテーブル
		(d)	TLB	(e)	ページフォルト	(f)	MMU
		(g)	セグメンテーションフォルト	(h)	シグナル	(i)	シグナルハンドラ
		(j)	物理ページ	(k)	ゼロで埋め	(l)	読み戻し
		(m)	マイナーフォルト	(n)	メジャーフォルト	(o)	スラッシング
		(p)	要求時ページング				
	(2)	異なるプロセスが用いる物理アドレスが重ならないようにして, あるプロセスの挙動が他のプロセスのメモリを盗み見たり, 破壊したりしないようにしている					
	(3)	機能の名称　例 mmap, プロセス間共有メモリ, copy-on-write, copy-on-writeによるfork, etc.					
		実現方式の概要 本文参照					

配点と正答率:

(1) (a)〜(p) 各 1 点. 正答率 (a) 99% (b) 97% (c) 87% (d) 84% (e) 97% (f) 90% (g) 80% (h) 49% (i) 38% (j) 81% (k) 37% (l) 63% (m) 89% (n) 89% (o) 53% (p) 74%

(2) 6 点 (66%)

(3) 名称 2 点 (84%). 実現方式についての説明 6 点 (77%)

解説・講評:

(1) 省略

(2) 問題で問うていることは, ここで説明されている仕組み (論理アドレス → 物理アドレスの変換) を, OS がどのように安全性の向上に使っているかということで, もっとも普通の解答例は

> 異なるプロセスの物理アドレスがかぶらないようにして, プロセスごとに分離されたメモリ空間を提供している (だから, プロセスがどんな論理アドレスを用いてメモリアクセスしても, 他のプロセスのデータを呼んだり壊したり出来ない)

というもの.

上記で直接説明はしていないが, OS (カーネル) のデータをユーザから守る手段について説明したものも OK. 例えば,

> CPU は user モードまたは superviser モードのいずれかにあり, 各ページには, user モードでアクセス可か否かの属性が付けられている. アドレス変換の過程で CPU のモードとこの属性がチェックされ, 違反していれば例外が起きる. OS は, 通常のプログラムは user モードで走らせ, OS のデータを, user モードでアクセス不可に設定する. これにより, user が OS のデータを見たり壊したり出来ないようにしている.

(3) 名称 2 点. 実現方式についての説明 6 点. 例はいくつもあり, 授業でたくさん述べている. 例えば以下.

mmap: ファイルをメモリ内の領域に写像する API. mmap システムコール呼び出し時は, 当該領域に対する物理ページを割り当てず, ページテーブル上で物理ページ不在に設定する. その後, 物理ページ不在なページへのアクセス時に発生するページフォルトをきっかけとして, 必要な領域への I/O を行う.

プロセス間共有メモリ: 同一ファイルの同一領域に対する共有マッピングに対しては, 異なるプロセスであっても同一の物理メモリを用いるように, ページテーブルを設定する. それによりプロセス間の共有メモリを実現する

mmap された領域の copy-on-write による物理メモリ共有: 複数のプロセスが同一ファイルの同一領域をプライベートマッピングしている場合, 意味的にはそれぞれに異なる物理メモリを割り当てる必要があるが, どれかのプロセスによる書き込みが行われるまでは, 同一の物理メモリを用いるように, ページテーブルを設定する. この際ページテーブル上で当該ページ群を書き込み不可にしておくことにより書き込みを検出し, 実際に書き込みがあったページに対して (のみ) コピーを作る. これによりプライベートマッピングの意味を保ったまま, 物理メモリの使用量を削減する.

copy-on-write による高速 fork: fork は, それを呼び出したプロセスのアドレス空間がまるごとコピーされた新しいプロセスを作る, という意味を持つ. この際, 実際に物理メモリをコピーするのではなく, 親プロセス・子プロセスが同じ物理メモリを用いるようにページテーブルを設定する. ページテーブル上で当該ページ群を書き込み不可にしておくことにより書き込みを検出し, 実際に書き込みがあったページに対して (のみ) コピーを作る. それにより物理メモリの使用量や, fork にかかる時間を削減する.

2 (1) (a)

生じ得る (生じ得ない) (どちらかを丸で囲む)

生じうる場合は具体的な実行系列

(b)

(生じ得る) 生じ得ない (どちらかを丸で囲む)

生じうる場合は具体的な実行系列 2スレッドで実行

スレッド1:	スレッド2:
...	
(1) long x = b->x;	
(2)	long x = b->x;
(3)	b->x = x + 1;
	(4) b->x = x + 1;
	...

とすると b->xは1となりbarrier(b)がリターンしない

(c)

(生じ得る) 生じ得ない (どちらかを丸で囲む)

頻忙待機によってOSがスレッドに割り当てる時間一回分(通常 数ms)の時間がかかる

生じうる場合は具体的な実行系列

スレッド1:	スレッド2:
while (b->x < next) ;	
コンテクストスイッチ	
	b->x = x + 1;

(2)

```
typedef struct {
  volatile long x;
  volatile long next;
  long N;
  pthread_mutex_t m
  pthread_cond_t co

} barrier_t;
```

```
void barrier_init(barrier_t * b, long N) {

  b->x = 0;

  b->next = N;

  b->N = N;
  pthread_mutex_init(b->mu, 0);
  pthread_cond_init(b->co, 0);
}
```

```
void barrier(barrier_t * b) {
  pthread_mutex_lock(b->mu);
  long next = b->next;


  long x = b->x;


  b->x = x + 1;
  while (b->x < next)
    pthread_cond_wait(b->co, b->mu);
  if (x == next - 1) {
    pthread_cond_broadcast(b->co);
  }
  pthread_mutex_unlock(b->mu);

}
```

(3)

```
typedef struct {
  volatile long x;
  volatile long next;
  long N;
  pthread_mutex_t m
  pthread_cond_t co

} barrier_t;
```

```
void barrier_init(barrier_t * b, long N) {

  b->x = 0;

  b->next = N;

  b->N = N;
  pthread_mutex_init(b->mu, 0);
  pthread_cond_init(b->co, 0);
}
```

```
void barrier(barrier_t * b) {
  pthread_mutex_lock(b->mu);
  long next = b->next;


  long x = b->x;


  b->x = x + 1;
  while (b->x < next)
    pthread_cond_wait(b->co, b->mu);
  if (x == next - 1) {
    b->next = next + b->N;
    pthread_cond_broadcast(b->co);
  }
  pthread_mutex_unlock(b->mu);
}
```

正しいことの説明

i (最初が i = 1) 回目の同期を行っている間, b->next が i * N になっており, b->x は (i - 1) * N <= b->x < i * N になっている

配点と正答率:

(1) (a) 2 点 (99%) (b) 生じるの選択 2 点 (100%). 生じることの説明 6 点 (97%). (c) 生じるの選択 2 点 (97%). 生じることの説明 6 点 (79%).

(2) 8 点 (43%)

(3) プログラム 8 点 (15%), 正しいことの説明 4 点 (18%)

解説・講評:

(1) 正答率高かった

(2) これは 条件変数を用いて同期を実現するための基本型と言って良いものなのだが, 思いの外, 正答率が低かった. 基本フォームは以下.

```
ロックする;
while (! 待ってる「条件」)
  アンロック+寝る;
「条件」が成立したら寝てる人を起こす;
アンロックする;
```

データをいじったり, (他の人がいじってるかも知れない状況で) 読んだりするにはロックを取るのが基本だから,

```
ロックする;
???
???
アンロックする;
```

までは, 条件変数以前の話である. これに加えて「条件」が成り立つまで待ちたいというのだから, 最初に示した一般型は, 極自然なものと見えるはずである.

具体的な API にすると,

- ロックする: `pthread_mutex_lock`
- ロックを開放+寝る: `pthread_cond_wait`
- 他の人を起こす: `pthread_cond_broadcast`
- アンロックする: `pthread_mutex_unlock`

なので最も自然な正解は以下.

```
/* 解答例 1 */
void barrier(barrier_t * b) {
  pthread_mutex_lock(b->mu);    /* ロックする */
  long next = b->next;
  long x = b->x;
  b->x = x + 1;
  while (b->x < next) {         /* b->x == next になるまで */
    pthread_cond_wait(b->co, b->mu); /* アンロック+寝る */
  }
  if (x == next - 1) {          /*「条件」が成立したら */
    pthread_cond_broadcast(b->co); /* 寝てる人を起こす */
  }
  pthread_mutex_unlock(b->mu);  /* アンロックする */
}
```

もっともこの問題に関しては while 文を挟む必要もない. というのも, 最後の一人が pthread_cond_broadcast を呼んだ (従って寝てる人が起こされた) 暁には, 「条件」が成り立っているに決まっているからである. なので以下も正解.

```
/* 解答例 2 */
void barrier(barrier_t * b) {
  pthread_mutex_lock(b->mu);
  long next = b->next;
  long x = b->x;
  b->x = x + 1;
  if (b->x < next) {
    pthread_cond_wait(b->co, b->mu);
  } else {
    pthread_cond_broadcast(b->co);
  }
  pthread_mutex_unlock(b->mu);
}
```

だが一般には, 「一度でも起こされたら『条件』が成り立っている」とは限らないので, while が必要な場合は多い. また, if の代わりに while を使ったからと言って, while を一回で抜ければ, if 文よりも遅い心配もほとんどない. なによりも, 「条件」が成り立つのを待つのに,

```
  while (!「条件」) ...
```

ほど正しさに確信の持てるコードはないので, 常にそうするものだと思っておいても差し支えはない.

さてこの問題に多かった誤答はどういうわけか, pthread_mutex_unlock を早く呼び出してしまうというもの.

```
void barrier(barrier_t * b) {
  pthread_mutex_lock(b->mu);
  long next = b->next;
  long x = b->x;
  b->x = x + 1;
  pthread_mutex_unlock(b->mu); /* !!!!!!!!!!!!!!!! */
  while (b->x < next) {
    pthread_cond_wait(b->co, b->mu);
  }
  if (x == next - 1) {
    pthread_cond_broadcast(b->co);
  }
}
```

これがなぜ間違いか? まず純粋な, 規約上の間違いから言うと,

> pthread_cond_wait(*co*, *mu*) は, mutex *mu* が取得されていることを前提として動作する (その前提で, *mu* を開放した上で呼び出したスレッドを寝かせる, という動作をする).

よって, ロックがかかっていない mutex を渡すのは規約上, すでに間違いである.

単なる規約上の違反ならば, 「じゃあもう少し気を利かせて, pthread_cond_wait(*co*, *mu*) は, mutex mu が取得されていなければそれを無視して, 呼び出したスレッドを寝かせる, という動作をすればいいじゃなイカ」と思うかも知れないが, そういう表面的な問題ではない.

```
  while (b->x < next) {
    pthread_cond_wait(b->co, b->mu);
  }
```

というループに, b->mu のロックが確保されていない状態で突入すると, b->x < next が真, という判定をして, pthread_cond_wait(b->co, b->mu); を呼ぼうとしたその刹那に, 最後の (N 個目のスレッドが) barrier(b) を呼び出し (そのままリターンまで実行し) てしまうことがある. するとこのスレッドは寝たまま起こされることなくブロックしてしまう. 簡単のためスレッド数を 2 個として以下のようなシーケンスでそれが発生する.

スレッド 1		スレッド 2	
`while (b->x < next)`	`/* b->x < next -> 真 */`		
		`pthread_mutex_lock(b->mu);`	
		`long next = b->next;`	`/* next <- 2 */`
		`long x = b->x;`	`/* x <- 1 */`
		`b->x = x + 1;`	`/* b->x <- 2 */`
		`pthread_mutex_unlock(b->mu);`	
		`while (b->x < next)`	`/* b->x < next -> 偽 */`
		`if (x == next - 1)`	`/* x == next - 1 -> 真 */`
		`pthread_cond_broadcast(b->co);`	`/* 誰も寝ていない */`
`pthread_cond_wait(b->co, b->mu);`			

比喩的に言うと, 条件不成立を見て寝ようと決心した人と, 条件を成立させた人の「すれ違い」ということで, しばしば, “lost wake up” と呼ばれる.

pthread_cond_wait はまさにこの, lost wake up が起きないように設計されている API であって, 寝る方は

- ロックする
- 条件をチェックする
- アンロック+寝る (pthread_cond_wait)

起こす方は

- ロックする
- 条件を成立させる
- 起こす (pthread_cond_broadcast)
- アンロックする

とすることで, 「寝る」「起こす」双方が, 共通のロックを握った上で呼び出される. だから, 条件をチェックしてから寝るまでの間に条件が書き換わる, などの心配をする必要がない.

ここまで書くと鋭い人は, pthread_cond_wait の「アンロック+寝る」の, 「アンロック」と「寝る」の間にスレッド 2 が入ってきて条件が書き換わったら, やはり同じ問題 (lost wake up) が起きるのではないか, と思うかも知れない. 非常に良い疑問で, そういうことがおきない, つまり, 「アンロック+寝る」は「不可分に」行われる, ということもちゃんと仕様に書かれている (pthread_cond_wait の man ページの引用. 下線部筆者).

> These functions atomically release mutex and cause the calling thread to block on the condition variable cond.

この不可分性をどう実現するかは自明ではなく, 興味深い課題なのだが, 長くなるので省略する.

(3) barrier を何度でも呼び出せるようにするというのはつまり, 何度でも barrier としての役割を果たせるようにするということである.

(2) の答えではなぜそうならないかと言うと, 全スレッドが一回ずつ barrier を呼び出した後は, 「b->x = スレッド数」となるため, それ以降の呼び出し時には,

```
1  while (b->x < next) {
2    pthread_cond_wait(b->co, b->mu);
3  }
```

が直ちに成立してしまうからである.

修正は簡単で, 上記で b->x がスレッド数になってしまうのが問題なのだから, barrier が成立したら b->x を 0 に戻してやれば良い, ...

```
void barrier(barrier_t * b) {
  pthread_mutex_lock(b->mu);
  long next = b->next;
  long x = b->x;
  b->x = x + 1;
  while (b->x < next) {
    pthread_cond_wait(b->co, b->mu);
  }
  if (x == next - 1) {
    b->x = 0; /* !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! */
    pthread_cond_broadcast(b->co);
  }
  pthread_mutex_unlock(b->mu);
}
```

... というのは問屋が降ろさない (!).

これだと,

```
  while (b->x < next) {
    pthread_cond_wait(b->co, b->mu);
  }
```

のループで pthread_cond_wait(b->co, b->mu) からリターンしたスレッドが再び条件 b->x < next をチェックしたときに, b->x が 0 になっているということで, ループから抜けられない. つまり, せっかく第 1 回目の barrier から抜けられる条件が成り立ったのに (b->x が 0 にリセットされたために), 抜けられないでいる, ということになってしまう. なお偶然にもこの問題は, while を使わないバージョン (解答例 2) では発生しない. したがって以下は正解.

```
void barrier(barrier_t * b) {
  pthread_mutex_lock(b->mu);
  long next = b->next;
  long x = b->x;
  b->x = x + 1;
  if (b->x < next) {
    pthread_cond_wait(b->co, b->mu);
  } else {
    b->x = 0;
    pthread_cond_broadcast(b->co);
  }
  pthread_mutex_unlock(b->mu);
}
```

ただこれには, pthread_cond_wait で起きたあとは条件をチェックせずにそのまま barrier を抜けられるという, 一般には通用しないかも知れない前提を使っているという欠点がある.

while 文を使った場合の直し方は簡単で, b->x < next の b->x の方はそのまま, 単調増加させていき, next の方もバリアの回数に合わせて追随させる. つまり, 1 回めの同期の際は, b->x == N で成立, 2 回目の同期は b->x == 2 * N で成立, ... ということだから,

```
void barrier(barrier_t * b) {
  pthread_mutex_lock(b->mu);
```

```
  long next = b->next;
  long x = b->x;
  b->x = x + 1;
  while (b->x < next) {
    pthread_cond_wait(b->co, b->mu);
  }
  if (x == next - 1) {
    b->next += b->N;
    pthread_cond_broadcast(b->co);
  }
  pthread_mutex_unlock(b->mu);
}
```

とすればよい.

3	(1)	(約) log (N)	
	(2)	グラフ概形 本文参照	なぜそのようなグラフになるのかの説明 readですべてのデータを読み出すのに n に比例する時間がかかる (検索そのものの時間はそれに比べてほとんど無視できる)

	(3)	long binsearch_file(char * filename, long n, long key) {
		int fd = open(filename, O_RDONLY);
		
		long sz = n * sizeof(record);
		
		~~record * R = malloc(sz);~~
		
		~~read(fd, R, sz);~~
		record * R = mmap(0, sz, PROT_READ, MAP_PRIVATE, fd, 0);
		long x = binsearch(R, n, key);
		
		~~free(R);~~
		munmap(R, sz);
		close(fd);
		
		return x;
		}	

	(4)	グラフ概形 本文参照	なぜそのようなグラフになるのかの説明 mmap の場合実際に (binsearch の 6 行目で) 触った配列要素(を含むページ. もしかすると周辺の数ページ)だけが読み込まれるため, 実行時間も触った要素数に比例するようになる
	(5)	グラフ概形 本文参照	なぜそのようなグラフになるのかの説明 log (n) と比べて n が圧倒的に大きいため

配点と正答率:

(1) 4 点 (92%)

(2) グラフの概形 4 点 (54%), 理由説明 4 点 (54%)

(3) 4 点 (40%)

(4) グラフの概形 4 点 (76%), 理由説明 4 点 (75%)

(5) グラフの概形 4 点 (32%), 理由説明 4 点 (32%)

解説・講評:

(1) 殆どの人が正解.

採点に当たっては, $\log n$, $\lfloor \log n \rfloor$, $\lceil \log n \rceil$, それに 1 を足したり引いたり... などという類のものはすべて正解としている.

なお, 何回かを n だけの式で書くことは出来ない. たとえば n が 3 のとき, 6 行目の if 文の結果がどちらに出るかによって, $r - l$ の値が, $3 \to 1$ と遷移する場合 (ループを 2 回回る) と, $3 \to 2 \to 1$ となる場合 (ループを 3 回回る) があるので, n の式だけで表現することは無理である. この時点で本当は, 「問題の不備だ〜」と叫ばれると半歩ほどあとずさりしてしまうのだが, そこはコンピュータを学ぶ人間ならば, 「だいたい $\log n$」だということを答えられればよい, と解釈して, さっと答える, というおおらかさが大事である.

実際は, $\lfloor \log n \rfloor$ または $\lceil \log n \rceil$ またはそれらに 1 を足したもの (細かい計算は後述).

点を与えていない人のほとんどは, $O(\log n)$ という解答で, これは, (n が大の時) $\log n$ の定数倍で押さえられるあらゆる関数の集合, という意味であり, 大目に解釈したとしても, $2 \log n$ と思っているのか, $3 \log n$ と思っているのか, $100 \log n$ と思っているのかなどを答えていないことになる. この問題は正確に, (ほぼ) $\log n$ と答えられるものを聞いているのだから, O などをつけずに答えるべきもの. どちらかというと, O の意味わかって書いてんのか〜, ということで点をあげたくなくなるというのが本音.

約 $\log n$ で答えとしては十分 (として採点している) のだが, 出題者の責任として正確にいくらになるかについて書いておく (試験中に以下をやれという意味ではない).

まず 2 分探索の動きを理解した人にはすぐにわかる重要な手がかりは,

> while 文を一回回る度に, $r - l$ の値がほぼ半分になる, ということである.

ここからただちに, while 文を回る回数はおよそ $\log n$ 回とわかる. 「ほぼ」の意味は, $r - l$ が奇数だったら, if の結果, 7 行目と 9 行目のどちらに行くかによって, 端数の 0.5 が切り上げられたり切り捨てられたりする, ということ. もちろんこれは「約 $\log n$」という結果を得るにあたってはあまり気にしなくて良い. 以下はもう少し正確に議論したい場合.

重要なのは while 文を回る回数だから, それに対するもっともらしい仮説を立てる. それは,

- 命題 (a) 4 行目の時点で,
$$\mathtt{r} - \mathtt{l} \leq 2^k$$
が成り立っているならば, この while 文は**高々あと k 回**繰り返す.
- 命題 (b) 4 行目の時点で,
$$2^k \leq \mathtt{r} - \mathtt{l}$$
が成り立っているならば, この while 文は**最低あと k 回**繰り返す.

どちらの証明もほとんど同じなので命題 (a) について示す. 帰納法で示す.

- $k = 0$ のとき

  $$\mathtt{r} - \mathtt{l} \leq 2^k$$

  は

  $$\mathtt{r} - \mathtt{l} \leq 1,$$

  つまり, $\mathtt{r} - \mathtt{l} = 1$ ということで, この時プログラムから実際, この while 文はすぐに抜ける (0 回繰り返す) のでよい.

- ある k まで成り立つとすると, $k+1$ でも成り立つことを言う. 4 行目の時点で

  $$\mathtt{r} - \mathtt{l} \leq 2^{k+1} \quad (\dagger)$$

  が成り立っているとする (目標はこの時 while 文が高々$(k+1)$ 回しか繰り返されないことを示すこと). 5-10 行目が実行されたとする (されなければ直ちに命題が成立する). 5-10 行目を実行する前の $\mathtt{r}$, $\mathtt{l}$ の値をそれぞれ r, l, 実行した後の値をそれぞれ r', l' と書くことにする.

  - $r - l$ が偶数の場合: 容易に, $r' - l' = (r-l)/2$ とわかる ($/$ は普通の数学の (端数ありの) 割り算). よって,

    $$r' - l' = (r-l)/2 \leq 2^{k+1}/2 = 2^k$$

    したがって帰納法の仮定により (†) の状態から 5-10 行目を実行した後, while 文は高々k 回繰り返す. したがって (†) の状態からは while 文は高々$(k+1)$ 回繰り返す.
  - $r - l$ が奇数の場合: 6 行目の if 文の結果次第で, $r' - l' = (r-l+1)/2$ または $r' - l' = (r-l-1)/2$ だが, どちらにせよ

    $$r' - l' \leq (r-l+1)/2.$$

    ここで

    $$r - l \leq 2^{k+1}$$

    であったが $r - l$ は奇数なので実際は,

    $$r - l + 1 \leq 2^{k+1}$$

    である. よって

    $$r' - l' \leq (r-l+1)/2 \leq 2^{k+1}/2 = 2^k.$$

    だからこの場合も, (†) の状態から, while 文は高々 $(k+1)$ 回繰り返す.

命題 (b) の証明もほとんど同じ. 命題 (a) と (b) を合わせて以下が示された.

- $n \leq 2^k$ のとき, while 文は高々k 回繰り返す

  $\Rightarrow \log n \leq k$ ならば while 文の繰り返し回数 $\leq k$

  $\Rightarrow$ while 文の繰り返し回数 $\leq \lceil \log n \rceil$ $\cdots$ (a')

(b') $n \geq 2^k$ のとき, while 文は最低 k 回繰り返す

  $\Rightarrow \log n \geq k$ ならば while 文の繰り返し回数 $\geq k$

  $\Rightarrow$ while 文の繰り返し回数 $\geq \lfloor \log n \rfloor$ $\cdots$ (b')

(a')(b') を合わせて

$$\lfloor \log n \rfloor \leq \text{while 文の繰り返し回数} \leq \lceil \log n \rceil$$

つまり,

$$\text{while 文の繰り返し回数} = \lfloor \log n \rfloor \text{ または } \lceil \log n \rceil$$

が示された.

さて求めるべきは, 触られる `a[]` の要素数であった. 多くの場合,

$$\text{触られる a[] の要素数} = \text{while 文の繰り返し回数}$$

であるが, 12 行目の `a[l]` が while 文中で触ったことがない要素である場合,

$$\text{触られる a[] の要素数} = \text{while 文の繰り返し回数} + 1$$

になる. 具体的には l が 0 のまま while 文が終了した場合にそうなる (細かすぎる話).

以上より,

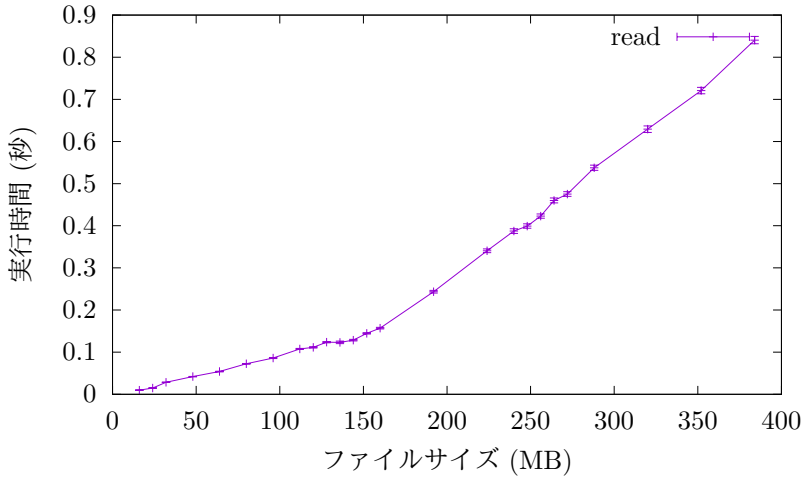
$$\text{触られる a[] の要素数} = \lfloor \log n \rfloor, \lceil \log n \rceil, \lfloor \log n \rfloor + 1, \lceil \log n \rceil + 1 \text{ のどれか}$$

実際には $n = 2^{\text{整数}}$ の形のときは, $\lfloor \log n \rfloor = \lceil \log n \rceil$, であり, それ以外のときは, $\lfloor \log n \rfloor + 1 = \lceil \log n \rceil$ であるから, $\lceil \log n \rceil$ はどの場合も冗長で,

$$\text{触られる a[] の要素数} = \lfloor \log n \rfloor, \lfloor \log n \rfloor + 1, \lceil \log n \rceil + 1 \text{ のどれか}$$

(2) グラフの概形 4 点 (54%), 理由説明 4 点 (54%)

説明よりまず, 実際の実験結果を見ていただこう. これは, メモリ容量 256MB の仮想マシンを作って, 最小限のソフトのみをインストールした環境で実験した結果である.



要するに二つの線分からなる折れ線で, 128MB 付近で傾きが少し変わる.

128MB 付近で少し傾きが変わるのは, 物理メモリ量 256MB と関係している. 一般に read で,

```
record * R = malloc(sz);
read(fd, R, sz);
```

のようにして, sz バイト読み出すと, OS がキャッシュ領域として sz バイト, ユーザプロセス内の領域 (R) のために sz バイトが必要だから, 都合 2sz バイトのメモリを消費する.

したがって sz > 128M の場合, 2sz バイトが物理メモリ (256MB) に収まらないので, sz バイト中の 128MB 以降を読み出すときに, 一部の領域をメモリから追い出し (ページアウトし) ながら, 読んでいくことになる. そこは, 単なる読み出しよりも時間がかかる.

採点基準にしているのは,

- 二本の線分からなる折れ線であること. 折れ線の折れる場所は, 正しくは 128MB 付近だが, 256MB 付近になっている答案 (非常に多かった) も, ○にしている. さらに寛大に, 折れ線になっていなくても (一本の線のようになっていても) ○としている.
- 線が原点付近から出発していること

この二つが表現されていないものは 0 点.

非常に多かった間違いは, 折れ線が原点付近から出発しておらず, **大きな y 切片を持つと誤解**している答案. この問題でファイルサイズが小さいときに起きることは,

(a) 小さいファイルを open

(b) 小さい領域を malloc で割当て

(c) 小さいファイルを read

ということであり, **大きなファイルを全部読んでその一部だけを探索しているわけではない.**

大きな y 切片のグラフを描いた人は, 授業中に見た別のグラフ (大きなファイルを**全部**読み込んで, そのあと読み込んだ領域を触っていくプログラム. x 軸が触った遼で, y 軸が (ファイルの読み込みを含んだ) 時間) と混同していると思われる.

もう一つ多かったタイプの誤答は, 途中まで直線で, それ以降は上に凸な (logっぽい) 曲線になる, というものである. 理由の方を読んで見るとどうやら, 256MB まではファイルを読み込む時間が支配的, 256MB を超えると, ファイルを読み込んだ後, binsearch が配列を検索する際に, ページアウトされた領域を再び触るようになる (そこで再び IO が発生する) から, という考察をしているらしかった. 上に凸の部分は, 直線に binsearch にかかる $\log n$ が足されている, というものである.

目の付け所は悪くないのだが, binsearch が触るページは高々$\log n$ ページ程度だから, binsearch を行っている間のページイン (IO) の数もせいぜい $\log n$ ページ程度となる. これと, ファイルを全部読み込む間の IO (n ページ) とは比べ物にならないほど後者が大きいので, それによる実行時間の増分は僅かである $n + \log n \approx n$.

(3) mmap を使えるかどうかを問うているだけの問題.

```
record * R = malloc(sz);
read(fd, R, sz);
```

を

```
record * R = mmap(0, sz, PROT_READ, MAP_PRIVATE, fd, 0);
```

に置き換え, それに応じて解放する方も,

```
free(R);
```

を

```
munmap(R, sz);
```

に置き換えるだけ. なお, 後者 (free の書き換え) は忘れていても不問としている.

mmap の引数の並びはとても覚えきれないので, 細かいことは不問にしている. 基準は以下が表現されているか.

- fd, ファイル中のオフセット (開始位置, つまり 0), サイズ (sz) の 3 つを渡していること
- 返り値が R に代入されているのであって, 引数に R を渡しているのではないこと

これらは mmap が何をする API なのかだけわかっていればわかるはずなので, それらができていないものは 0 点.

よくあった誤解は,

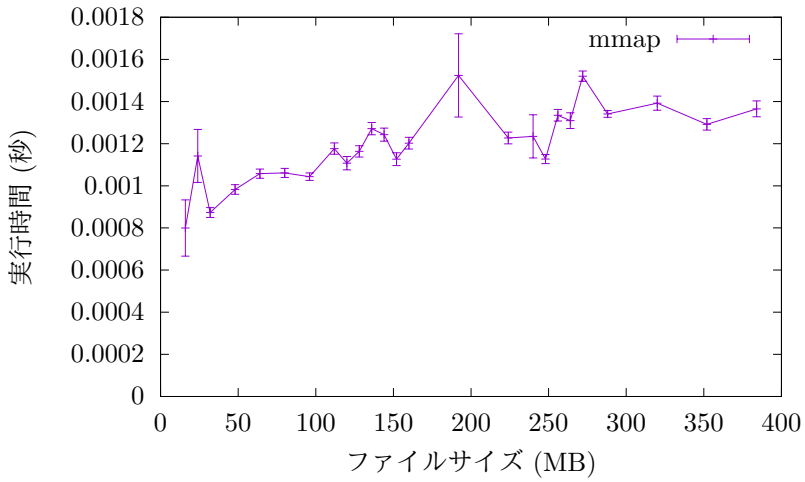
```
record * R = malloc(sz);
mmap(R, 0, sz, PROT_READ, MAP_PRIVATE, fd, 0);
```

や

```
record * R;
mmap(R, 0, sz, PROT_READ, MAP_PRIVATE, fd, 0);
```

みたいな答案. mmap はメモリ割当てを伴う操作である (malloc の効果を**兼ねる**) ことはよく覚えておくこと.

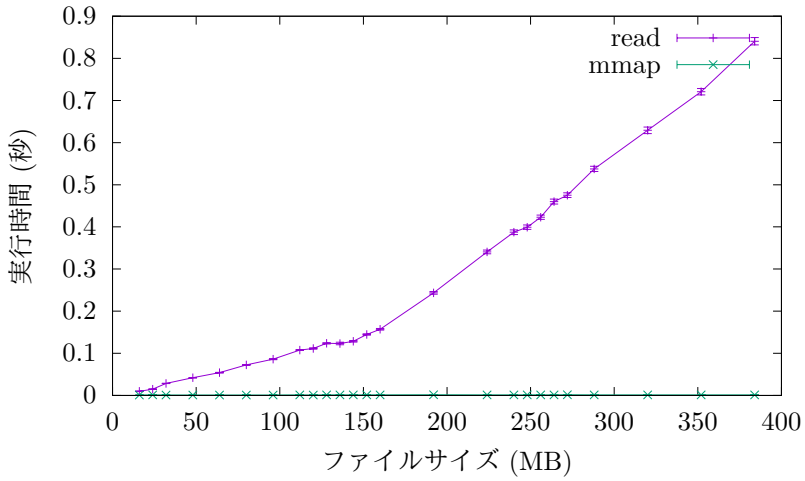
(4) こちらも実際の実験結果を見せる.



実行時間があまりに小さく, いろいろな不確定要素の影響を大きく受けるので, あまりきれいに現れないが, 要するに mmap では実際に binsearch 内で触った領域しか読み込まないため, $\log n$ のグラフに近いものになるはず, というところがポイント (上記はまあ, それに近いものになっている).

なお上記は, ランダムなキーでの検索を, (毎回キャッシュからファイルを追い出して)200 回行い, その平均と, 95% 信頼区間を表示したものである.

(5) 両者の「形」を問うたあとで, read と mmap の時間が相対的にどうなのか, を聞いている. 実際の結果は見ての通り.



要するに, mmap の方が「圧倒的に」速い. mmap の方はほとんど 0 にしか見えない. 数学で $n \gg \log n$ は習って知っていると思うが, それは要するにこういうこと.

聞きたかったのは, この違いが「圧倒的」であることを本当にわかっているでしょうか, ということで, 採点の際見ているのは,

- (2), (4) どちらとも, 少なくとも 0 点ではないこと (どちらかでも 0 点の場合, 両者を重ねて書くこと自体, ほとんど何を問うているのかわからなくなるため)
- グラフの右の方で少なくとも read > mmap であること

これが表現されていなければ 0 点.

あとは, グラフの右の方で read ≫ mmap であるか否かにより, ○となるか, △となるかを決めている. これはもうほとんど気持ちの問題.