

# 平成30年度オペレーティングシステム期末試験

2019年1月22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2-9ページからなる
- 解答用紙は5ページ(両面).
- 各問題の解答は所定の解答欄に書くこと.

# 1

以下の会話を読んでその後の問いに答えよ。

チコちゃん (以下 T): ね～え岡村あ、この中で一番コンピュータに詳しい人はだ～れ～?

岡村 (以下 O): え～, やっぱりオレかなあ

T: じゃあ岡村に聞くけど, 同じコンピュータをみんなで同時に使っても安全なのはな～ぜ?

O: え!? あ, 安全!? それは, ...

T: 例えば, 他の人と同じコンピュータを使っているのに, 他の人に勝手にファイルを見たり, 書き換えられたりする心配はないの?

O: え!? あ～, それは, あれや, ちゃんと OS がチェックしてくれてるんとちゃうか?

T: でも, ファイルは所詮, ハードディスクとか SSD に入っているわけだし, OS といえども所詮, CPU の命令を実行してるだけだから, OS がハードディスクや SSD のデータを読んだり書いたりするのと同じ命令を発行すれば, 直接その中のデータが読めたり書けたりしちゃうじゃない?

O: え～, まあ～, 世の中そこまでするやつおらんちゃうの ...?

T: ボーっと生きてんじゃねーよ!

シュワシュワシュワ～

ナレーション: どうやって OS が複数のユーザを安全にサポートしているかも知らないで, やれこれからはクラウドだの, いやマルチテナントだのと, のんきにサーバを使っている日本人のなんと多いことか (後略)

(1) 下線部について, これができないようになっている仕組みを説明せよ。

以下は Linux を含めた, Unix 系の OS に関する設問である。

- (2) システムコールの中でファイルの読み書き権限を検査する際, 使われるパラメータのひとつがファイルにつけられた許可属性 (permission) と, 所有者 (owner) である。ファイルシステムで, ファイルにつけられた許可属性と所有者を変更するシステムコールの名前をそれぞれ答えよ (ヒント: 同名のコマンドがある)。また, それを用いて許可属性や所有者を変更できるのは誰かを, それぞれ答えよ。
- (3) システムコールの中でファイルの読み書き権限を検査する際, 使われるもうひとつのパラメータが, それを試みたプロセスの実効ユーザ ID (effective user ID, 以下 `eid`) である。あるファイルを, 読める (書ける) ユーザと読めない (書けない) ユーザがいるのも結局はそれを行うプロセスの `eid` が異なることから生じている。  
あるサーバにログインする際, パスワードや公開鍵などで認証をして, 認証されたユーザ ID を `eid` とするプロセスが実行を開始するまでの過程を, 鍵となるシステムコールに言及しながら説明せよ。
- (4) `sudo` や `su` のような, 管理者ユーザ (root) 権限でコマンドを実行できるシステムコールがある。root 権限でコマンドを実行できるということは, `sudo` や `su` は, root を `eid` として動いているということになる。しかし `sudo` や `su` は当然のことながら, root ではないユーザを `eid` として動いていたプロセスが呼び出すこともできる。ということは Unix においては, プロセスの `eid` を root 以外のユーザから, root に変更できる仕組みがあることになる。それはどのような仕組みか?

## 2

以下で定義される  $N$  次元のベクトルを  $M$  個格納したデータ構造 `vectors_t` を考える.

```
1 typedef struct {  
2     long a[M][N];  
3 } vectors_t;
```

このデータ構造に対して以下の二つの操作を考える.

- `int search(vectors_t * u, long q[N]);`  $u$  内に  $N$  次元ベクトル  $q$  と一致するベクトルが含まれていれば 1, なければ 0 を返す.
- `void swap(vectors_t * u, long i, long j);`  $u$  の  $i$  番目のベクトルと  $j$  番目のベクトルを入れ替える.

以下では, これらの関数を複数のスレッドが呼び出しても正しく動作するように実装することを考える. なお, 「正しく動作する」とは以下のようなことである.

初期状態で  $u$  に含まれるベクトルを  $p_0, \dots, p_{M-1}$  とするとき, 複数のスレッドが `swap` や `search` を並行に呼び出しても,  $q$  がどれかの  $p_i$  に一致したときおよびそのときのみ, `search( $u$ ,  $q$ )` が 1 を返す.

以下は, 1 つのスレッドしかこれらの関数を呼び出さないという前提で正しく動作する実装 (以下, 「逐次実装」と呼ぶ) である.

```
1 int search(vectors_t * u, long q[N]) {  
2     int found = 0;  
3     for (long i = 0; i < M; i++) {  
4         if (vec_eq(u->a[i], q)) found = 1;  
5         if (found) break;  
6     }  
7     return found;  
8 }
```

```
1 void swap(vectors_t * u, long i, long j) {  
2     if (i == j) return;  
3     long * p = u->a[i];  
4     long * q = u->a[j];  
5     for (long k = 0; k < N; k++) {  
6         long t = p[k];  
7         p[k] = q[k];  
8         q[k] = t;  
9     }  
10 }
```

`vec_eq( $p$ ,  $q$ )` は 2 つのベクトルが等しければ 1, さもないと 0 を返す関数で, 以下で定義される.

```
1 int vec_eq(long p[N], long q[N]) {  
2     for (long k = 0; k < N; k++) {  
3         if (p[k] != q[k]) return 0;  
4     }  
5     return 1;  
6 }
```

以下の問いに答えよ.

- (1) 逐次実装を複数のスレッドが呼び出した時に, 正しく動作しないことがあることを, 具体的な実行の系列で示せ.

(2) 前問のような問題や、それがおきる状況のことを総称してなんと呼ぶか?

(3) 排他制御 (Pthread の mutex) を使って正しく動作するよう、`vectors_t` のデータ定義、`search`、`swap` を修正せよ。解答用紙には逐次実装が書いてある。適切に行を挿入・削除・変更して示せ。`search`、`swap` が使われる前にデータ構造の初期化が必要であれば、解答欄の `mk_vectors` 内に書け。

なお以下に Pthread の mutex 関連の API を示す (`pthread_mutex_init` の第 2 引数には 0 を渡せばよい)。

```
1 int pthread_mutex_init(pthread_mutex_t * m, pthread_mutexattr_t * a);
2 int pthread_mutex_lock(pthread_mutex_t *m);
3 int pthread_mutex_unlock(pthread_mutex_t *m);
```

さて、排他制御をすると全ての `search`、`swap` が逐次的に実行されるため、性能上の問題が生ずる。そこで、read-write lock というものを使うことを考える。read-write lock は、以下のインタフェースを持ち、

```
1 void rwlock_init(rwlock_t * l); /* 初期化 */
2 void rwlock_rdlock(rwlock_t * l); /* read lock を取得 */
3 void rwlock_wrlock(rwlock_t * l); /* write lock を取得 */
4 void rwlock_unlock(rwlock_t * l); /* read (write) lock を解放 */
```

いかなる時点においても、以下が成り立つ。

1 つの read-write lock の write lock が同時に 2 つ以上のスレッドに保持されることはなく、write lock と read lock が同時に保持されることもない (複数のスレッドが read lock を同時に保持することはあり得る)。

言い換えると、ある read-write lock の、write lock を保持しているスレッド数を  $W$  個、read lock を保持しているスレッド数を  $R$  個とすると、常に

$(W = 0)$  または  $(W = 1 \text{ かつ } R = 0)$

が成り立つ。なお、あるスレッド ( $T$ ) が read-write lock ( $l$  とする) の write lock (または read lock) を「保持している」とは、 $T$  が呼び出した `rwlock_wrlock(l)` (または `rwlock_rdlock(l)`) が終了 (return) して、その後 `rwlock_unlock(l)` を呼び出していない状態のことである。

(4) 正しく動作するプログラムを、今度は read-write lock を使って書け。

(5) read-write lock を、mutex と条件変数で実現したい。Pthread の条件変数の API を以下に示す。

```
1 int pthread_cond_init(pthread_cond_t * cond, pthread_condattr_t * attr);
2 int pthread_cond_wait(pthread_cond_t * cond, pthread_mutex_t * mutex);
3 int pthread_cond_broadcast(pthread_cond_t *cond);
```

以下がそのデータ構造である。ただし、フィールド `rw` は、下位 1 ビットで write lock を保持しているスレッド数 (上記の  $W$ )、残りの 31 ビットで、read lock を保持しているスレッド数 (上記の  $R$ ) を維持している (スレッドの数は高々  $2^{31}$  個未満と仮定してよい)。

```
1 typedef struct {
2     int rw;
3     pthread_mutex_t m;
4     pthread_cond_t c;
5 } rwlock_t;
```

以下が初期化 (他の関数が呼び出される前に一度だけ呼び出される) 関数と, `rwlock_rdlock`, `rwlock_wrlock`, `rwlock_unlock` の実装の一部である. `□` になっている部分を埋める形で, 実装を完成させよ. ただし `rwlock_unlock` は, write lock または read lock を保持しているスレッドが呼び出すと仮定して良い (そうでなかった場合の動作は気にしないで良い).

注: `□` の個数や大きさは, 必ずしも, 埋めるべき式や文の個数を指定・示唆するものではない.

```
1 void rwlock_init(rwlock_t * l) {
2     l->rw = □;
3     □
4 }
```

```
1 void rwlock_rdlock(rwlock_t * l) {
2     □
3     while (1) {
4         int rw = l->rw;
5         if (□) {
6             □
7         } else {
8             l->rw = □;
9             break;
10        }
11    }
12    □
13 }
```

```
1 void rwlock_wrlock(rwlock_t * l) {
2     □
3     while (1) {
4         int rw = l->rw;
5         if (□) {
6             □
7         } else {
8             l->rw = □;
9             break;
10        }
11    }
12    □
13 }
```

```
1 void rwlock_unlock(rwlock_t * l) {
2     □
3     int rw = l->rw;
4     if (□) {
5         l->rw = □;
6     } else {
7         l->rw = □;
8     }
9     □
10 }
```

さて以下では, `pthread_mutex_t` がどう実装されているかを考える.

以下は `pthread_mutex.lock` と `pthread_mutex.unlock` の, 間違った実装である.

```

1 typedef struct {
2     int locked;
3 } pthread_mutex_t;
4
5 /* pthread_mutex_lock の間違った実装 */
6 int pthread_mutex_lock(pthread_mutex_t * l) {
7     if (l->locked == 0) {
8         l->locked = 1;
9     } else {
10         block(&l->locked);
11     }
12     return 0;
13 }
14
15 /* pthread_mutex_unlock の間違った実装 */
16 int pthread_mutex_unlock(pthread_mutex_t * l) {
17     l->locked = 0;
18     wake_all(&l->locked);
19     return 0;
20 }

```

ただし, `block(p)` は呼び出したスレッドをブロックさせる関数, `wake_all(p)` は, `p` に対して `block(p)` を呼び出してブロックしているスレッドを (全て) 起こす関数である.

(6) 上記の実装は間違っており, 以下の 2 つの間違いが起きうる. それぞれどのような実行系列で起きうるかを答えよ.

(a) 2 つのスレッドが同時に lock を取得できてしまう.

(b) lock が解放された状態であるにもかかわらず, lock が取得できずにスレッドがブロックしてしまう.

(7) 以下の 3 つの関数が使えらるとして, 上記の実装を正しく修正せよ.

- `cas(int * p, int a, int b)` (compare-and-swap): アドレス `p` に格納されている値 (`*p`) が `a` であれば, そこに `b` を代入し 1 を返す. `a` でなければ何もせずに 0 を返す. ここで `*p` が `a` であるかをチェックしてから代入するまでが atomic に行われる.
- `cab(int * p, int a)` (compare-and-block): アドレス `p` に格納されている値 (`*p`) が `a` であれば, 呼び出したスレッドをブロックさせる. さもないとブロックせずに return する. ここで `*p` が `a` であるかをチェックしてからブロックするまでが atomic に行われる.
- `wake_all(int * p)`: 上記の説明通り

(8) 問題 (4) では mutex を用いて read-write lock を実装した. 今度は, mutex を用いず, 上記 3 つの関数を用いて, read-write lock を実装せよ.

### 3

2 分探索木は, 要素を高速に検索するためのデータ構造で, 以下を保ちながら動作する.

- 各ノードは, キーと対応する値 (ともに long 型とする) を持つ
- 各ノードは, 左の子と右の子を持つ場合がある
- あるノード ( $n$ ) が, 左の子 ( $l$ ) を持つ場合,  $l$  のキー  $<$   $n$  のキー (\*)
- あるノード ( $n$ ) が, 右の子 ( $r$ ) を持つ場合,  $n$  のキー  $<$   $r$  のキー (\*)

以下ではファイルに格納された 2 分探索木を検索をするときの性能について考える. ファイルには全てのノードが隙間なく連続して格納されており (つまりファイルにはノードの配列がそのまま格納されていると思えば良い), 左の子や右の子は, それらのノードが格納された配列の添字 (整数) で示すことにする.

以下はそれに基づいたノードのデータ構造である. なおこの node 一つのサイズ (sizeof(node)) は 32 バイトである.

```
1 typedef struct {
2     long key;           /* キー */
3     long val;           /* 値 */
4     long l;             /* 左の子 (いないときは-1) */
5     long r;             /* 右の子 (いないときは-1) */
6 } node;
```

l (r) が左 (右) の子の添字であり, いない場合は-1を入れる. 図 1 は 2 分探索木の例と, それを node の配列として表現したものである

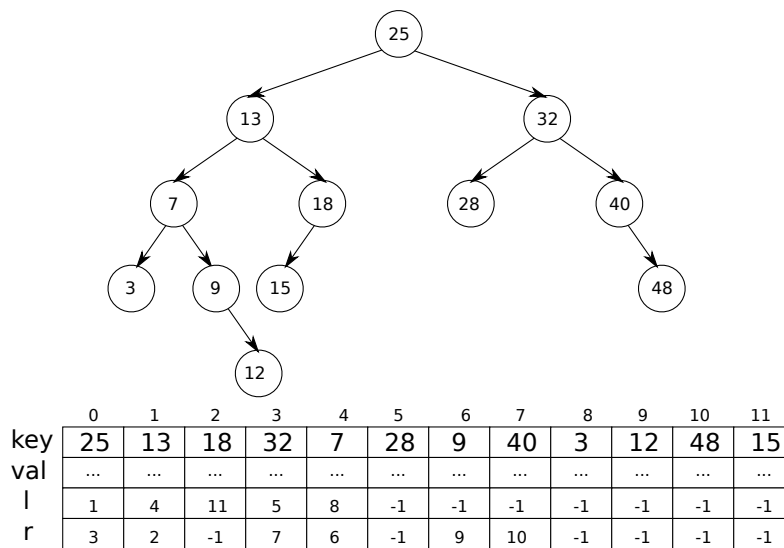


図 1: 2 分探索木 (上) とその配列表現

2 分探索木全体は全ノードを格納した配列 (nodes), 要素数 (n) から成り, 定義は以下である.

```
1 typedef struct {
2     node * nodes;           /* 全node の配列 */
3     long n;                 /* 要素数 */
4 } bstree_t;
```

2分探索木の探索は、根ノードから始めて、上記の条件(\*)を利用してノードの左または右の子を追跡する。途中で値が見つかるか、追跡する子ノードがなくなったところで探索が終了する。

なお空でない2分探索木の根ノードは配列の0番目に格納されている。

```
1 long search_rec(long n, long k, bstree_t * t, long * np) {
2     node * p = &t->nodes[n];
3     *np = *np + 1;
4     if (k == p->key) {
5         return p->val;
6     } else if (k < p->key) {
7         if (p->l == -1) { /* 左の子がない */
8             return -1; /* not found */
9         } else {
10            return search_rec(p->l, k, t, np);
11        }
12    } else {
13        if (p->r == -1) { /* 右の子がない */
14            return -1; /* not found */
15        } else {
16            return search_rec(p->r, k, t, np);
17        }
18    }
19 }
20
21 long bstree_search(bstree_t * t, long k, long *np) {
22     if (t->n == 0) { /* 空 */
23         return -1; /* not found */
24     } else {
25         return search_rec(0, k, t, np);
26     }
27 }
```

以上の準備のもとで以下のような実験をする。

**ステップ 1:** 空の2分探索木を作り、 $n$ 個のキーを乱数で発生させて順に挿入し、ファイル( $F$ )に格納する。

**ステップ 2:**  $F$ をファイルキャッシュから追い出す。

**ステップ 3:** 「起動したら $F$ を読み込んで、乱数を $m$ 個発生させて順に探索する」プロセスを走らせる。このプロセス $m$ 個の探索にかかる時間 $T$ を計測する。つまり以下のような計測を行う。

```
1 T0 = 現在時刻;
2 bstree_t t[1];
3 t->n = n;
4 t->nodes = read_nodes(F, n); /* node 配列を読み込む */
5 for (long i = 0; i < m; i++) {
6     long k = 乱数;
7     bstree_search(t, k);
8 }
9 T1 = 現在時刻;
10 T = T1 - T0;
```

上記の `read_nodes` を実現するために以下の二つの方法を使う。

**方法 1:**  $n$  個分の領域を `malloc` で確保して、`fread` などの関数ですべて読み込む

**方法 2:** `mmap` を用いる



実験を行ったマシンは1GB ( $\approx 10^9$  バイト) の主記憶を搭載しており、この実験以外で主記憶はほとんど使われていないものとする。また以下では  $N = 32n$  (つまり node  $n$  要素に必要な主記憶が  $N$  バイト) と書くことにする。

- (1) 方法1, 方法2の実装を示せ。両者の概形はともに以下の通り (ここでも空欄の大きさや行数は答えの大きさを指示・示唆するものではない)。

```

1 node * read_nodes(char * F, long n) {
2     [ ]
3     node * nodes = [ ]
4     [ ]
5     return nodes;
6 }

```

関連する関数のインターフェースを以下に示す (使わないものがあったもよい)。

```

1 void *malloc(size_t size);
2 int open(const char *pathname, int flags);
3 ssize_t read(int fd, void *buf, size_t count);
4 FILE *fopen(const char *pathname, const char *mode);
5 size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
6 void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);

```

使うべき定数の名前などがうろ覚えなときは適当に書いた上で適宜、その意図をコメントせよ。

- (2) 以下のそれぞれの場合において、方法1, 方法2それぞれに対し、 $T$  のおおよその値を  $m, n$  の式で書いた上で、グラフの概形を書け。そのようになる理由も説明せよ。グラフには要点となる点の値や傾きなどを適宜記入せよ。

それぞれのケースで、方法1と方法2の違いがわかるよう、同じグラフ上に重ねて書くこと。

計算に当たっては、挿入されているキー、検索のためのキーとも、大きな範囲に一樣に散らばっている、また、2分探索木もよくバランスされている (完全2分木に近い) と仮定して良い。

式を書くにあたり、必要に応じて以下を定数として用いよ。他に必要な定数があれば適宜定義した上で用いよ。

- $P$  : ページサイズ (4096 バイト)
- $A$  : 1 ページを単独で2次記憶から読み出す時間。1 ページを2次記憶に追い出して代わりに別のページを読み出す時間もほぼ同じとしてよい。
- $a$  : 多数のページを逐次的に2次記憶から読み出す際の、1 ページあたりの時間 ( $A$  よりも小さいことが期待される)。

- (a)  $N = 2\text{GB}$  程度に固定して、 $m$  を1から  $n/100$  程度まで変えて  $T$  を計測する (グラフの横軸は  $m$ , 縦軸は  $T$ )。
- (b)  $N = 300\text{MB}$  程度に固定して、 $m$  を1から  $n/100$  程度まで変えて  $T$  を計測する (グラフの横軸は  $m$ , 縦軸は  $T$ )。
- (c)  $m$  を100 ( $n$  に比べて小さな値) に固定して、 $N$  を32KB程度から2GB程度まで変えて  $T$  を計測する (グラフの横軸は  $N$ , 縦軸は  $T$ )。

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○かどうかだけが書いており、点数などは書いていません。点数を知りたいければ以下の配点を見て計算してください。
- 満点は 86 点 (無理矢理 100 点にする意味がないので)。レポートと総合して成績をつけています。
- 試験の最高点は 80 点 (93%), 最低点は 8 点 (9%), 平均点は 39.33 点 (45%)。

学生証番号		氏名	
-------	--	----	--

1		スーパーユーザモード, 特権命令	
	(1)	6点	
	(2)	許可属性を変更するシステムコール 2点 chmod	許可属性を変更できるユーザ 2点 所有者, root
		所有者を変更するシステムコール 2点 chown	所有者を変更できるユーザ 2点 root
	(3)	ログインを受け付けるプロセスがeuid = root として実行されており, 認証が済んだところで seteuid, setreuid で認証されたユーザに euid を変更する 6点	
	(4)	ファイルに1属性として setuid bit があり, それはexecされるとそのファイルの所有者に euid が変更される 6点	

## 解説・補足

- (1) (正解率 0.57)
- (2) (正解率 chmod 0.67, 所有者と root 0.63, chown 0.51, root 0.66)
- (3) (正解率 0.34)
- (4) (正解率 0.1) root でないユーザがどうやって, su や sudo などのコマンドで root 権限で実行されるのか, その仕組みを問うたもの. su や sudo がその答えであるかのような解答が多かったのだが, それは違う. sudo や su というシステムコールがあるわけではない (もしあったとすればおそらくそのシステムコール自身がパスワードを引数に要求するようなものになるだろうがそれは柔軟な仕組みとは言えない).

2	<p>正しく動作しない実行系列の説明</p> <p>swap(u, 0, 1)を実行しているスレッドをA, search(u, q)を実行しているスレッドをBとする。</p> <p>ただし q = u[0]だったとする。</p> <p>[1] Aが7行目までを実行する</p> <p>[2] Bがsearchを実行する</p>
(2)	2点 競合状態
(3)	<p>4点</p> <pre> typedef struct {     pthread_mutex_t l;     long a[M][N]; } vectors_t;  int search(vectors_t * u, long q[N]) {      int found = 0;     pthread_mutex_lock(&amp;u-&gt;l);     for (long i = 0; i &lt; M; i++) {          if (vec_eq(u-&gt;a[i], q)) found = 1;          if (found) break;      }     pthread_mutex_unlock(&amp;u-&gt;l);     return found; } </pre> <pre> vectors_t * mk_vectors() {      vectors_t * u = malloc(sizeof(vectors_t));     pthread_mutex_init(&amp;u-&gt;l);     /* u-&gt;a[*][*]を初期化(省略) */      return u; }  void swap(vectors_t * u, long i, long j) {      if (i == j) return;      long * p = u-&gt;a[i];      long * q = u-&gt;a[j];     pthread_mutex_lock(&amp;u-&gt;l);     for (long k = 0; k &lt; N; k++) {          long t = p[k];          p[k] = q[k];          q[k] = t;      }     pthread_mutex_unlock(&amp;u-&gt;l); } </pre>
(4)	<p>4点</p> <pre> typedef struct {     rwlock_t l;     long a[M][N]; } vectors_t;  vectors_t * mk_vectors() {      vectors_t * u = malloc(sizeof(vectors_t));     rwlock_init(&amp;u-&gt;l);     /* u-&gt;a[*][*]を初期化(省略) */      return u; } </pre>

		<pre> int search(vectors_t * u, long q[N]) {      int found = 0;     rwlock_rdlock(&amp;u-&gt;l);     for (long i = 0; i &lt; M; i++) {          if (vec_eq(u-&gt;a[i], q)) found = 1;          if (found) break;      }     rwlock_unlock(&amp;u-&gt;l);     return found; } </pre>	<pre> void swap(vectors_t * u, long i, long j) {      if (i == j) return;      long * p = u-&gt;a[i];      long * q = u-&gt;a[j];     rwlock_wrlock(&amp;u-&gt;l);     for (long k = 0; k &lt; N; k++) {          long t = p[k];          p[k] = q[k];          q[k] = t;      }     rwlock_unlock(&amp;u-&gt;l); } </pre>
(5)	6点	<pre> void rwlock_init(rwlock_t * l) {     l-&gt;rw = 0;     pthread_mutex_init(&amp;l-&gt;m, 0);     pthread_cond_init(&amp;l-&gt;c, 0); }  void rwlock_wrlock(rwlock_t * l) {     pthread_mutex_lock(&amp;l-&gt;m);     while (1) {         int rw = l-&gt;rw;         if (rw) {             pthread_cond_wait(&amp;l-&gt;c, &amp;l-&gt;m);         } else {             l-&gt;rw = 1;             break;         }     }     pthread_mutex_unlock(&amp;l-&gt;m); } </pre>	<pre> void rwlock_rdlock(rwlock_t * l) {     pthread_mutex_lock(&amp;l-&gt;m);     while (1) {         int rw = l-&gt;rw;         if (rw &amp; 1) {             pthread_cond_wait(&amp;l-&gt;c, &amp;l-&gt;m);         } else {             l-&gt;rw = rw + 2;             break;         }     }     pthread_mutex_unlock(&amp;l-&gt;m); }  void rwlock_unlock(rwlock_t * l) {     pthread_mutex_lock(&amp;l-&gt;m);     int rw = l-&gt;rw;     if (rw == 1) {         l-&gt;rw = 0;     } else {         l-&gt;rw = rw - 2;     }     pthread_cond_broadcast(&amp;l-&gt;c);     pthread_mutex_unlock(&amp;l-&gt;m); } </pre>
(6)	(a) 4点	<p>2つのスレッドが同時にlockを取得できてしまう実行系列の説明</p> <p>pthread_mutex_lockを呼び出している二つのスレッドをA, Bとする</p> <p>[1] Aが7行目までを実行</p> <p>[2] Bが7行目までを実行</p>	
	(b) 4点	<p>lockが解放された状態であるにもかかわらず, lockが取得できずにスレッドがブロックしてしまう実行系列の説明</p> <p>lockを呼び出しているスレッドをA, unlockを呼び出しているスレッドをBとする.</p> <p>スレッドBがすでにロックを取得しているとする</p> <p>[1] Aが7行目までを実行(l-&gt;locked == 1を読み出す)</p> <p>[2] Bが19行目までを実行</p> <p>[3] Aが10行目を実行</p>	

(7) 6点	<pre> tint pthread_mutex_lock(pthread_mutex_t * l) {     int * p = &amp;l-&gt;locked;     while (1) {         int locked = *p;         if (locked == 0) {             if (cas(p, 0, 1)) {                 break;             }         } else {             cab(p, 1);         }     } } </pre>	<pre> tint pthread_mutex_unlock(pthread_mutex_t * l) {     int * p = &amp;l-&gt;locked;     int locked = *p;     *p = 0;     wake_all(p); } </pre>
(8) 6点	<pre> void rwlock_init(rwlock_t * l) {     l-&gt;rw = 0; }  void rwlock_wrllock(rwlock_t * l) {     while (1) {         int rw = l-&gt;rw;         if (rw) {             cab(&amp;l-&gt;rw, rw);         } else {             if (cas(&amp;l-&gt;rw, 0, 1)) {                 break;             }         }     } } </pre>	<pre> void rwlock_rdlock(rwlock_t * l) {     while (1) {         int rw = l-&gt;rw;         if (rw &amp; 1) {             cab(&amp;l-&gt;rw, rw);         } else {             if (cas(&amp;l-&gt;rw, rw, rw + 2)) {                 break;             }         }     } }  void rwlock_unlock(rwlock_t * l) {     int rw = l-&gt;rw;     if (rw == 1) {         l-&gt;rw = 0;         wake_all(&amp;l-&gt;rw);         break;     } else {         assert((rw &amp; 1) == 0);         if (cas(&amp;l-&gt;rw, rw, rw - 2)) {             wake_all(&amp;l-&gt;rw);             break;         }     } } </pre>

## 解説・補足

- (1) (正解率 0.71) 解答例以外にも様々な正解がある. 具体的な実行系列 (こういうタイミングで実行されると実際に search の結果が間違ふ) ということを示せという問題だが, それにあまり沿っているいえないような書き方でもたいがいのものは正解にしている

- (2) (正解率 0.66)

- (3) (正解率 0.66) 殆どの人ができていた. よくあった間違いは,

```
1 pthread_mutex_t * m;
```

のように mutex をポインタ変数で保持しておきながら, このポインタ変数に正しいアドレスを代入していないもの. そのような回答はほとんどが何故か,

```
1 pthread_mutex_init(&u->m);
```

のようにポインタ変数のアドレスを pthread\_mutex\_init に渡していた. なおこの組み合わせでも半分の点を与えている. その他細かいミスはたくさん大目に見ている.

正しい組み合わせは,

```
1 pthread_mutex_t m;
```

と

```
1 pthread_mutex_init(&u->m);
```

または,

```
1 pthread_mutex_t m;
```

と

```
1 u->m = malloc(sizeof(pthread_mutex_t));
2 pthread_mutex_init(u->m);
```

の組み合わせ. 以下でも良い.

```
1 pthread_mutex_t m[1];
```

```
1 pthread_mutex_init(u->m);
```

- (4) (正解率 0.58) よくある間違いは (2) の間違いと同じ.

- (5) (正解率 0.32) 条件変数の使い方の基本を問うもの. 以下のテンプレートを覚えること.

```
1 lock(m);
2 /* 待つ必要があるなら */
3 while (!進める条件) {
4     cond_wait(c, m);
5 }
6 /* 待っている人がいるかもしれないなら */ cond_broadcast(c);
7 unlock(m);
```



- (6) (正解率 (a) 0.84, (b) 0.47) これも解答例以外にも様々な正解がある. (b) は少しこれまでと少し毛色の違う競合状態だが, block してはいけないのに block してしまうのはどういう場合かと考えれば, すぐに答えは出てくるはず.
- (7) (正解率 0.17) これは少し発展的問題. とはいえ, 条件変数の使い方が, 考え方として頭に入っていればそれと似ている.

```
1 while (1) {  
2     if (lock されている) {  
3         すれ違いで状態が変化していないことを確認しつつblock  
4     } else {  
5         すれ違いで状態が変化していないことを確認しつつlock を取得  
6     }  
7 }
```

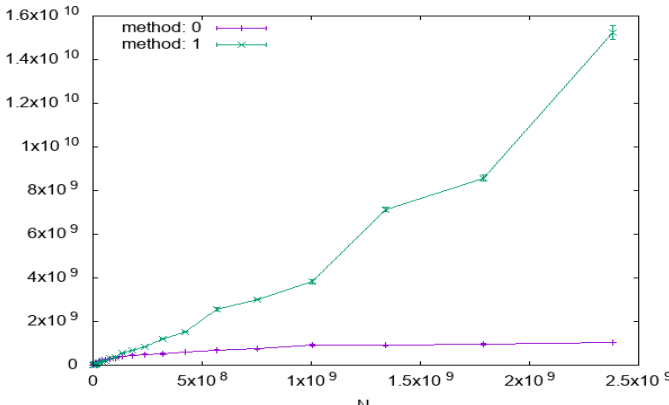
なお解答例よりも短く以下のような解答も正解.

```
1 int * p = &l->locked;  
2 while (1) {  
3     if (cas(p, 0, 1)) {  
4         break;  
5     } else {  
6         cab(p, 1);  
7     }  
8 }
```

- (8) (正解率 0.03) 同じことを read-write lock に対して行うもの. 少しややこしいが考え方は同じ. rdlock であれば

```
1 while (1) {  
2     if (wrlck されている) {  
3         すれ違いで状態が変化していないことを確認しつつblock  
4     } else {  
5         すれ違いで状態が変化していないことを確認しつつrdlock を取得(rw = rw + 2)  
6     }  
7 }
```

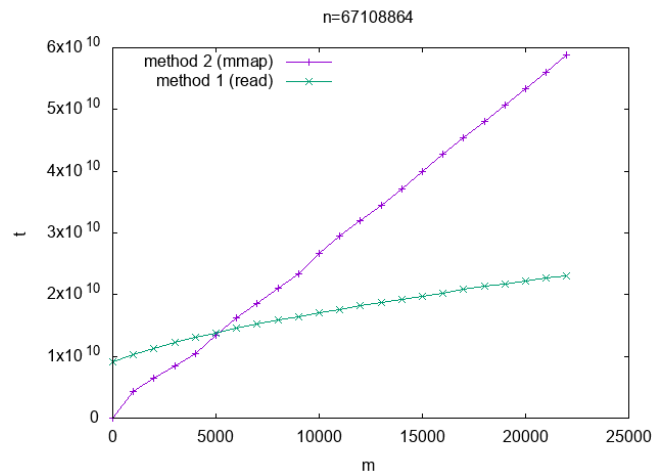
3	(1)	<div> <div> 方法1:  <pre> nnodes * read_nodes(char * F, long n) {     int fd = open(filename, O_RDWR);     nodes * nodes = malloc(t-&gt;sz * sizeof(node));     size_t rd = 0;     while (rd &lt; count) {         size_t x = read(fd, buf + rd, count - rd);         if (x == -1) { err(1, "read"); }         else if (x == 0) { err(1, "premature EOF"); }         else { rd += x; }     }     return nodes; } </pre> </div> <div> 方法2:  <pre> nodes * read_nodes(char * F, long n) {     int fd = open(filename, O_RDWR);     nodes * nodes = mmap(0, t-&gt;sz * sizeof(node),         PROT_READ PROT_WRITE, MAP_SHARED, fd, 0);      return nodes; } </pre> </div> </div>
(2)	(a)	<div> <div> 式: (方法1) <math>T = a N / P + A m \log(N)</math>  (なお方法1, 2の式が正しければ両者の直線は並行になる  はずだが実測結果はなっていない. 以下の注参照) </div> <div> 式: (方法2) <math>T = A m \log(N)</math> </div> </div> <div> <p>グラフ (実測値は右のようになったが, 以下の解説を参照)</p> <p>理由 (方法1) 全ページを一旦読み込む(aN/P)がその後の searchの際にも各ページへのアクセスでページフォルトが生ずる(一回の検索で約 log(N)ページに触るので A m log(N))</p> <p>(方法2) mmapの場合ページを一旦読み込む必要がないので</p> </div>
	(b)	<div> <div> 式: (方法1) <math>T = a N / P</math> </div> <div> 式: (方法2) <math>T = A \max(m \log(N), N / P)</math>  (実際はもう少し複雑. 以下のグラフを参照) </div> </div> <div> <p>グラフ (実測値は右のようになったが, 以下の解説を参照)</p> </div>

		理由 (方法1)全ページを一旦読み込み(aN/P)その後はほぼページフォルトが生じない	(方法2) (a)と途中までは同じだが徐々にメモリ上にデータが埋まってくるのでほとんどページを新たに取得する必要がなくなる
(c)		式: (方法1) $T = a N / P$	(方法2) $T = A m \log (N)$
4点 x2	グラフ		
		理由: (方法1) 全ページを読み込むのにかかる時間そのもの	(方法2) mが小さいため一回の探索でいたいlog(N)の異なるページを触る

## 解説・補足

- (1) (正解率 read 0.77, mmap 0.49) mmap の方でよくあった間違いは, mmap の場合でもまず malloc でメモリを割り当てて mmap にそのアドレスを渡しているというもの. mmap はメモリ割当を兼ねている (すでに割当済みのアドレスを用いることはできない). なお, 引数の細かい記号 (PROT\_PRIVATE など) は不問にしている. read であれば, open, malloc, read (または fread) と書かれていればほとんど正解. mmap であれば, open, mmap と書かれていればほとんど正解.
- (2) (正解率 read 0.48, mmap 0.52) グラフは実際に実験をして得たものである.

ただし実際のところポイントはグラフの左端の方, つまり  $m$  が非常に小さい場合であって, そこだけ ( $m < n/3000$  の部分だけ) をズームしたものが以下 (その意味では問題としても,  $m = n/100$  とわずに,  $m = n/3000$  くらいにすべきだった).



採点基準としては, read の方が  $y$  切片の大きなほぼ直線になっていれば正解, mmap の方が  $y$  切片のほぼ 0 なるほぼ直線になっていれば正解としている.

mmap が  $y$  切片が 0 となる理由は, mmap の時点では実際の読み込みを行わず, 実際のアクセスがあってから読み込みが行われるからである. read が  $y$  切片が  $> 0$  となる理由は, read を発行した時点ですべてのデータが読み込まれるから.

そしてこの問題においては「データ量  $>$  物理メモリ量」であるために, 全てのデータが物理メモリに載ることはない. したがって探索のループが始まったあともページフォルトを起こし続ける.

なお両者の傾きが大きく異なっており, read の方が傾きが小さい. 結果として,  $m$  が極小さい場合を除けば mmap の方が遅くなっている.

これにはあまり必然的な理由がないと思われる. 理由については探究が必要で残念ながら今は明快な解答がない.

なおもちろん採点としては両者の傾きが同じであっても OK にしている.

- (3) (正解率 read 0.49, mmap 0.12) 基準としては, 前者がほぼ傾き 0 の直線になっていること. 後者は, 最初は急速に立ち上がり, やがてほぼ傾き 0 の直線状になっていること.

read の挙動は, すべてを read するのに一定時間がかかり, この時点でデータの全てがメモリ上に存在するので, 以降の検索にはほとんど時間がかからない (ほぼ  $m$  によらないとして良い時間. 実際にはグラフはもちろん僅かに正の傾きを持っているはずである).

mmap の方の挙動は興味深い。あるページを初めて触った際にそのページが主記憶上に読み込まれ、1 度読み込まれたあとはページアウトされることはないため、しばらくするとほぼすべてのページが主記憶に読み込まれた状態になり、以降はページフォルトが起きない状態になる。

なお、後者を  $m$  の式で表そうと思うとかなり難しく、式は不問にしている。

- (4) (正解率 read 0.24, mmap 0.31) ポイントは、read にかかる時間は  $N$  に比例、mmap は  $\log N$  に比例していることである。なお read の場合に、 $N$  が主記憶 (正確には  $N/2$ 。read の場合、OS のキャッシュと、ユーザが指定したバッファに同量の主記憶を必要とする) を越えたところで、ページを読み込むだけでなく、追い出すコストが発生するため性能が劣化する。したがって途中で傾きが大きくなる。