2020 年度オペレーティングシステム期末試験

2020 年 1 月 19 日 (火)

- **問題は 4 問**
- **この冊子は，表紙が 1 ページ (このページ), 問題が 2-9 ページからなる**
- 解答用紙の Submit は何度行っても良く, 最後に Submit したものだけが受け付けられるので, **試験終了や, その間際になるのを待たずに各自のタイミングで行うこと**
- 試験終了後に Submit するのは不正行為となる (Submit はできてしまうが, Submit した時刻が記録されており, 後からそれが発覚することになるので, 決して行わないよう注意せよ)

1

以下の会話を読んで, その下の問に答えよ.

駒場 (以下 K): 俺のおかんがね, 好きなシステムコールがあるらしいんやけど, その名前をちょっと忘れたらしくてね

内海 (以下 U): ほな俺がね, おかんの好きなシステムコール一緒に考えてあげるから, どんな特徴言うてたかって教えてみてよ〜

K: なんでもおかんが言うにはね, そのシステムコールは, 一つのシステムコールでファイルを読むことにも書くことにも, 使えるいうねんな

U: ほ〜, [(a)] やないか. その特徴はもう完全に [(a)] やがな. すぐわかったよこんなもん〜

K: ちょっとわからへんのよな. 俺も [(a)] と思ったんやけどな, おかんが言うにはな, OS の初心者でも man 見れば簡単に使えるやろ言うねん

U: ほ〜, ほな [(a)] と違うなあ. 何しろ [(a)] は引数の順番とか絶対覚えられへんし, OS の初心者は man 見てもなんのこっちゃかようわからんよ. ほなもうちょっと詳しく教えてくれる?

K: おかんが言うにはな, ファイルの読み書きとは無関係にメモリを割り当てることにも使えるいうねん

U: [(a)] やないかい! [(a)] は実はメモリ割り当てのための C のライブラリ関数 [(b)] の中でも普通に呼び出されとんのやから. [(a)] がなんでもやるせいで, メモリを割り当てる Unix の古くからのシステムコール [(c)] はもういらん思われてて, 肩身せもーてしゃーないんやから

K: 俺も [(a)] と思ったんやけどな, おかんが言うにはな, 引数とかも単純で, すぐに使える言うねん

U: [(a)] ちゃうやないかい! いくら OS に慣れた人でも, あの引数の個数と, 意味不明な並び順は絶対に man からコピペせな使えんのよ. [(a)] ちゃうよ. 他になんか言ってなかった?

K: 便利なだけやのうて, 効率よくファイルを読み書きするのにもいいらしいねんな. 物理メモリより大きいくらいのファイルの場合でもシステムコールの呼び出し自身は一瞬で終わる言うねん

U: [(a)] やないかい! [(a)] は OS がやってるメモリ管理の仕組みと一体的に動いていて, 大きなファイルを [(a)] しても,

(2) そのときには実際のファイルとの入出力はやらずに終わるんやから〜. その特徴を生かして, 大きなファイルの [(d)] を [(e)] に読み書きするときなんかに, 特に有用やねんから〜. 実はその特徴を生かして,

(3) コマンド実行するたびに [(a)] はいっぱい使われてんねんから〜

K: でもようわからんのよ. おかんが言うにはな, 名前だけ見ればなにしおるシステムコールかすぐにわかる言うねん

U: そら [(a)] ちゃうよ! Unix のシステムコールは, 文字数減らすために, 初めての人へのわかりやすさとか考えてへんからね! ファイルを作るシステムコールが, `create_file` とかやのーて, [(f)] やからね! 一文字削りたいがために, ちゃんとした英単語になってへんからね!

K: 他にも, OS がメモリ管理に使っているテクニックで, プロセスを作るシステムコールの

(4) [(g)] を高速化するのにも使われている, [(h)] っていうのがあるらしいねんけどな, [(a)] の実装でもそのテクを使うてるらしいねんな. そのおかげで, 大きなファイルを

(5) [(i)] が読み書きした場合にも, メモリが効率的に使えるらしいねん. あと使い方によって,

(6) プロセスをまたがって [(j)] することもできるんやって.

U: 完全に [(a)] よそれは!

K: でもおかんが言うには [(a)] やないっていうんねん

U: ほしたら [(a)] ちゃうよそれは! どないなっとんねん〜

K: いや, おとんが言うには, close ちゃうか言うねん

U: いや絶対ちゃうやろ, もうええわ, ありがとうございました〜

(1) (a)〜(j) の空欄に当てはまる単語や短い語句を書け

(2) 下線部 (2) において, 「そのときには実際のファイルとの入出力はやらずに終わる」とあるがでは実際のファイルとの入出力はいつどういうきっかけで行われるのか?

(3) 下線部 (3) 「コマンド実行するたびに [(a)] はいっぱい使われて」いるとあるが, 何を目的としていっぱい使われているか, そして, なぜそこで [(a)] を使うことが効果的なのか述べよ

(4) 下線部 (4) で述べられている [(g)] を高速化する仕組みはどのようなものか, [(h)] との関係がわかるように説明せよ

(5) 下線部 (5) 「 [(i)] が読み書きした場合にも, メモリが効率的に使える」のはなぜか. 特に, [(a)] を使わずにそれをした場合と対比させた上で答えよ

(6) 下線部 (6) 「プロセスをまたがって [(j)] する」の仕組みを説明せよ

2

スレッドに CPU を割り当てる (スケジューリングする) 際の目標や, それを達成するための仕組みについて考える.

一般に OS がスレッドをスケジューリングする際,

利用効率: CPU を必要とするスレッドに CPU を割り当てる,

公平性: CPU を必要なスレッド間で CPU 割当時間を均一に近くする,

応答性: 対話的アプリケーションが CPU を必要とした際に短い遅延時間で CPU を割り当てる,

ことなどを目標とする.

現在の Linux のスケジューラでは, 各スレッドの vruntime という値を記録, 更新することでこれを達成している. 大雑把に言えば vruntime は各スレッドが使用した CPU 時間ということで, CPU をスレッドに割り当てる際, vruntime が最小のスレッドに割り当てることで公平性を達成する.

しかし文字通り, 「スレッドの vruntime = そのスレッドが使用した CPU 時間」としてしまうと不都合がある.

以下の問に答えよ.

(1) どのような不都合か? あげた目標のどれが達成できなくなるかを具体的に説明せよ

(2) 通常の環境でプロセスは 100 個以上立ち上がっていることが珍しくないが, だからといって計算をするプログラムを実行したときに, その実行速度が, 他のプロセスがない場合と比べて 1/100 になる (CPU が 1/100 しか割り当てられない) わけではない. それは OS が「CPU を必要とするスレッドに CPU を割り当てる」からなのだが, その具体的な仕組みについて説明せよ

(3) 無限ループするプログラムを実行してもマウスは動くし, 端末ソフトやエディタも動作を続ける. つまり, プログラム自身はずっと CPU を使い続けるように書かれていても, CPU がそのスレッドに独占されるわけではない. その具体的な仕組みについて説明せよ

(4) vruntime が文字通り, 「スレッドの vruntime = そのスレッドが使用した CPU 時間」というわけではないと述べたが, Linux OS は実際にスレッドの vruntime どのように維持, 更新しているのか, 説明せよ.

3

以下の問に答えよ.

(1) ファイル名 F と整数 $n \geq 0$ が与えられ, F の最初の n 行 (F が n 行に満たないときは F の全て) を, 標準出力に出力する関数, `head(`F`,` n`)` を書け.

(2) ファイル名 F と整数 $n \geq 0$ が与えられ, F の最後の n 行 (F が n 行に満たないときは F の全て) を, 標準出力に出力する関数, `tail(`F`,` n`)` を書け.

注 1: ただしどちらも大きなファイルに対して効率的に動くよう, 入出力の量がファイルの大きさではなく, 実際に出力される部分の大きさに比例するような実装をせよ (端的に言えば, 余計な部分を大量に読み込まないように, ということ).

注 2: ここで F の中の「行」とは, F 中の空でない部分文字列で, 以下の 3 つを全て満たすものである.

1. その最後の文字は, 改行文字 (`'\n'`) であるかまたは, ファイルの最後の文字である
2. その最初の文字は, ファイルの先頭の文字であるかまたは, 改行文字の直後の文字である
3. 最後の文字以外は改行文字ではない

形式的には, ファイルが n 文字から成り, それらを先頭から $c_0, c_1, \cdots, c_{n-1}$ と書くことにすると, i 文字目で始まり $j(\geq i)$ 文字目で終わる部分文字列 $c_i, c_{i+1}, \cdots, c_j$ が「行」であるとは, 以下を満たすときを言う.

1. c_j が改行文字であるか, $j = n - 1$
2. $i = 0$ であるか, c_{i-1} が改行文字
3. $c_i, c_{i+1}, \cdots, c_{j-1}$ は改行文字でない

例えば, F の中身が

`a`, `b`, `c`, nl, `d`, `e`, `f`, `g`, nl, `h`, `i`

であった場合 (nl は改行文字を表すとする), このファイルは以下の 3 行

`a, b, c,` nl　`d, e, f, g,` nl　`h, i`

から成っている. F の中身が

`a`, `b`, `c`, nl, `d`, `e`, `f`, `g`, nl, `h`, `i`, nl

であった場合は,

`a, b, c,` nl　`d, e, f, g,` nl　`h, i,` nl

から成っている (この場合も 3 行).

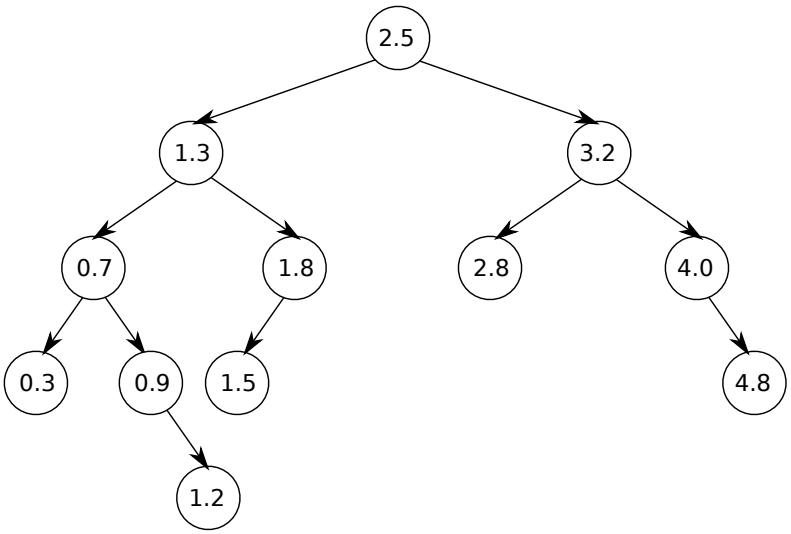
4

以下の問いに答えよ.

2 分探索木は, 以下の条件を満たすデータ構造である.

- 各ノードにひとつの値が格納されている. 以降, あるノード n に格納されている値のことを単に「n の値」と呼ぶことにする
- 各ノードには左の子, 右の子がいるかもしれない
- あるノード n に左の子 l がいる場合, l 及び l の子孫全てのノード m に対し, n の値 $> m$ の値である
- あるノード n に右の子 r がいる場合, r 及び r の子孫全てのノード m に対し, n の値 $< m$ の値である

下図は 2 分探索木の一例である.



以下は 2 分探索木を C の構造体として表したものである.

```c
//% file: bst_1.c
/* 2分探索木のノード */
typedef struct node {
  double val;                   /* ノードの値 */
  struct node * children[2];    /* 子ノード (いない場合は 0) */
} node_t;
/* 2分探索木 */
typedef struct {
  node_t * root;
} bst_t;
```

2 分探索木に対する値の挿入 `bst_insert` と, 検索 `bst_search` を考える.

- `bst_t * t = mk_bst()` : 空の 2 分探索木を作って返す.
- `bst_insert(bst_t * t, double x)` : 2 分探索木 t に x を挿入する
- `bst_search(bst_t * t, double x)` : 2 分探索木 t に x が挿入されていれば 1 を, なければ 0 を返す

以下はそれらを C の関数として表したものである.

```c
//% file: bst_2.c
/* 2分探索木のノードを作る */
node_t * mk_node(double val) {
  node_t * n = malloc(sizeof(node_t));
  if (!n) err(1, "malloc");
  n->val = val;
  n->children[0] = n->children[1] = 0;
  return n;
}
/* 空の 2分探索木を作る */
bst_t * mk_bst() {
  bst_t * t = malloc(sizeof(bst_t));
  if (!t) err(1, "malloc");
  t->root = 0;
  return t;
}
/* t から始まるノードの下にノード n を挿入 */
int bst_insert_rec(node_t * t, node_t * n) {
  double x = n->val;
  /* 重複 (挿入しない) */
  if (x == t->val) return 0;
  /* val が小さい->左 (which_child == 0), 大きい->右 (which_child == 1) */
  int which_child = x > t->val;
  node_t * child = t->children[which_child];
  if (child == 0) {
    /* そっちに子供はいない -> n がその子供になる */
    t->children[which_child] = n;
    return 1;
  } else {
    /* そっちに子供はいる -> n をその子孫にする */
    return bst_insert_rec(child, n);
  }
}
/* 2分探索木への挿入 */
int bst_insert(bst_t * t, double val) {
  node_t * n = mk_node(val);
  node_t * root = t->root;
  if (root == 0) {
    /* 空の木 -> n が根に */
    t->root = n;
    return 1;
  } else {
    /* 空でない -> n を根の子孫に */
    return bst_insert_rec(root, n);
  }
}
/* t の下を探索 */
int bst_search_rec(node_t * t, double val) {
  /* 見つかった */
  if (val == t->val) return 1;
  /* val が小さい->左 (which_child == 0), 大きい->右 (which_child == 1) */
  int which_child = val > t->val;
  node_t * child = t->children[which_child];
  if (child == 0) {
    return 0;
  } else {
    return bst_search_rec(child, val);
  }
```

```
}
/* 2分探索木の探索 */
int bst_search(bst_t * t, double x) {
  node_t * root = t->root;
  if (root == 0) {
    /* 空の木 */
    return 0;
  } else {
    /* 空でない木 */
    return bst_search_rec(root, x);
  }
}
```

この 2 分探索木を, 複数のスレッドが並行にアクセスしても正しく動作するようにする (スレッドセーフにする) ことを考える. ここで「正しく動作する」とは, 挿入した値が見つからなかったり, 挿入していない値が見つかったりすることがないということである. **なお本問では, 値が削除されることはない.**

正しさの厳密な定義は面倒なので述べないがさしあたり以下の (複数のスレッドが並行して多数の値を挿入し, それらが全て終わってからそれらの値が見つかること, それら以外の値が見つからないことを確かめる) プログラムがエラーを起こさずに終了することでそれを確かめる.

```
//% file: bst_3.c
/* 現在時刻 */
double cur_time() {
  struct timespec ts[1];
  clock_gettime(CLOCK_REALTIME, ts);
  return ts->tv_sec + ts->tv_nsec * 1.0e-9;
}

int main(int argc, char ** argv) {
  int i = 1;
  long n     = (i < argc ? atol(argv[i]) : 1000000); i++;
  bst_t * t = mk_bst();
  double t0 = cur_time();
  /* 並行にn 個の値を挿入 */
#pragma omp parallel for
  for (long i = 0; i < n; i++) {
    if (bst_insert(t, sin(2 * i)) == 0) {
      fprintf(stderr,
              "warning: sin(%ld) = %f duplicates a previous value\n",
              2 * i, sin(2 * i));
    }
  }
  double t1 = cur_time();
  /* 2n 個の値を並行に検索 */
#pragma omp parallel for
  for (long i = 0; i < n; i++) {
    assert(bst_search(t, sin(2 * i)));
    assert(bst_search(t, sin(2 * i + 1)) == 0);
  }
  double t2 = cur_time();
  printf("OK\n");
  printf("%f sec to insert %ld elements\n", t1 - t0, n);
  printf("%f sec to search %ld elements\n", t2 - t1, 2 * n);
  return 0;
}
```

(1) 並行プログラミングにおける「競合状態」とは何かを説明せよ.

(2) そのままではこの 2 分探索木はスレッドセーフではない理由を, 上記の「正しさ」に基づいて説明せよ. 単に, 「競合状態があるから」などという答えを期待しているのではないので注意. どのような「正しくない」結果が, どのようにして生じうるのかを具体的に示すこと.

(3) 排他制御を用いてスレッドセーフな 2 分探索木を実現せよ. 解答欄の `mk_bst`, `bst_insert` の中だけを変更して, 呼び出し側のコードは変更せずそのまま動作するようにすること.

(4) 排他制御を用いずにスレッドセーフな 2 分探索木を実現せよ. 解答欄の `mk_bst`, `bst_insert` の中だけを変更して, 呼び出し側のコードは変更せずそのまま動作するようにすること.