2022年度オペレーティングシステム期末試験

2023年1月31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 2-12 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

次の関数 search_mem_for_string は, アドレスの範囲 [a, a + L) の中で文字列 s を検索し, その出現回数を返り値として返す.

```
#define _GNU_SOURCE
#include <string.h>

/* [a, a + L) 中で s の出現回数を返す */
long search_mem_for_string(char * a, long L, char * s) {
  char * a_end = a + L;
  char * p = a;
  long n_founds = 0;
  while (1) {
    p = memmem(p, a_end - p, s, strlen(s));
    if (!p) break;
    p++;
    n_founds++;
  }
  return n_founds;
}
```

次の関数 search_file_for_string は, ファイル filename の先頭 L バイト中の文字列 s を検索し, その出現回数を返り値として返す.

```
#include <err.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
long search_mem_for_string(char * a, long L, char * s);

/* filename の先頭L バイトにある s の出現回数を数える */
long search_file_for_string(char * filename, long L, char * s) {
    FILE * fp = fopen(filename, "r");
    if (!fp) err(1, "fopen(%s)", filename);
    char * a = malloc(L);
    if (!a) err(1, "malloc(%ld)", L);
    size_t rd = fread(a, 1, L, fp);
    if (rd != (size_t)L) err(1, "fread");
    fclose(fp);
    long c = search_mem_for_string(a, L, s);
    free(a);
    return c;
}
```

次の関数 search_file_for_strings は, 複数の文字列を検索するもので, ファイル filename の先頭 L バイト中に, 文字列の配列 S 中の各文字列 S[0], S[1], ..., S[n-1] それぞれが何回現れるかを表示する.

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

long search_file_for_string(char * filename, long L, char * s);

```

```
/* filename の先頭L バイトにある S[0], S[1], ..., S[n-1]
   の出現回数を数え, 表示する */
void search_file_for_strings(char * filename, long L, char ** S, int n) {
  for (int i = 0; i < n; i++) {
    long c = search_file_for_string(filename, L, S[i]);
    printf("%s : %ld\n", S[i], c);
  }
}
```

これらを使った以下のプログラムは,

$$./\texttt{search\_file}\ \textit{filename}\ \ L\ \ w_0\ \ w_1\ \ \cdots\ \ w_{n-1}$$

として起動すると, *filename* の先頭 L バイトに, $w_0\ w_1 \cdots w_{n-1}$ がそれぞれ何回現れるかを表示する.

```
#include <err.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include <unistd.h>

void search_file_for_strings(char * filename, long L, char ** S, int n);

/* ファイルのサイズを求める */
long get_file_size(char * filename) {
  struct stat sb[1];
  if (stat(filename, sb) == -1) {
    err(1, "stat(%s)", filename);
  }
  return sb->st_size;
}

/* 時刻 (ns 単位)を得る */
long cur_time_ns() {
  struct timespec ts[1];
  clock_gettime(CLOCK_REALTIME, ts);
  return ts->tv_sec * 1000 * 1000 * 1000 + ts->tv_nsec;
}

int main(int argc, char ** argv) {
  int i = 1;
  char * filename = (i < argc ? argv[i] : "test1.txt"); i++;
  long L          = (i < argc ? atol(argv[i]) : -1); i++;
  if (L == -1) {
    L = get_file_size(filename);
  }
  int n_words = (argc - i < 0 ? 0 : argc - i);
  long t0 = cur_time_ns();
  search_file_for_strings(filename, L, argv + i, n_words);
  long t1 = cur_time_ns();
  printf("took %.6f sec to search %s for %d words\n",
         (t1 - t0) * 1e-9, filename, n_words);
```

```
40    return 0;
41  }
```

以下の問いに答えよ.

(1) search_file_for_strings のみを変更して, $w_0, w_1, \cdots, w_{n-1}$ を別々のプロセスで並列に検索するようにせよ.

解答用紙にあるセルを変更してプログラムを記述し, その下のセルを実行してコンパイル, その結果を実行すること (以下, プログラムを書く問題は全て同様).

(2) そのように変更したプログラムで上記を実行したときに, 全プロセス合計で消費する**仮想**メモリ量 (走らせてから終了するまでの間の最大値) はおよそいくらか? 理由の説明とともに, L, n などを用いて示せ. ただし L は十分大きな数 (例えば 10^7 以上) とし, ファイルの読み込み以外に消費する ($L = 0$ のときにも消費する) メモリ量は, L に比べて十分小さいとして無視してよい.

(3) **物理**メモリ量についてはどうか? 同様に答えよ. なお, 計算機は十分な量のメモリを搭載しているものとし, 物理メモリは OS が (このプログラムを実行した結果) 消費する量も含めよ.

(4) (1) で得た search_file_for_strings を用いた上で, さらに search_file_for_string だけを変更して, 物理メモリの消費量を大幅に削減するにはどうしたらよいか, 概要, それで物理メモリがどのくらいに削減されるか, その理由, を述べよ.

(5) 前問の概要に沿って search_file_for_string を実際に変更せよ.

(6) もともとのプログラムでは, $w_0, w_1, \cdots, w_{n-1}$ の出現回数が, この順に出力されるのに対して, (1) や (5) で作ったプログラムでは, その表示順が入れ替わることがある. (1) で得た search_file_for_strings を少し変更して, $w_0, w_1, \cdots, w_{n-1}$ を別々のプロセスが検索しつつ, $w_0, w_1, \cdots, w_{n-1}$ の出現回数がこの順に表示されるようにせよ.

2

英単語の長さの分布 (色々な文字数の英単語が現れる確率) を調べたい. そのために英語が書かれた大きなテキストファイル — 例えば Wikipedia の英語ダンプファイル (解凍したもの) — を用意し, このファイル中に出現する英単語の長さ (文字数) の分布を調べることでそれを得る. もちろんそれが実際の世の中で使われている英単語長の分布と一致する保証はないが, 今はそれで満足することにする.

ここで,「ファイル中に出現する英単語」とは, そのファイルの部分文字列 (w とする) であって, 以下を満たすもののこととする. 以下で w はファイル中の a 文字目から始まり, $(b-1)$ 文字目で終わっているとする. ファイルに含まれる全文字列を S で表すことにし, ファイルの大きさを L バイトとする.

- w はアルファベット (大文字, 小文字) および数字のみからなる. つまり,

$$\forall i \in [a, b)\ \mathrm{isalnum}(S[i]) \quad (*)$$

  ただし, $S[i]$ は文字列 S の i 文字目 (最初の文字は 0 文字目) を意味し, $\mathrm{isalnum}(c)$ は, 文字 c がアルファベットまたは数字のときに真となる述語とする.

- w は前後どちらにも, 性質 $(*)$ を満たしながらこれ以上伸ばせない. 言い換えると,
  - w はファイル先頭の文字で始まっている $(a = 0)$ か, もしくは w の 1 つ前の文字はアルファベットや数字ではない $(\neg\texttt{isalnum}(S[a-1]))$.
  - w はファイル最後の文字で終わっている $(b = L)$ か, もしくは w の 1 つ後の文字はアルファベットや数字ではない $(\neg\texttt{isalnum}(S[b]))$.

例えば以下に示す中身を持つファイルの英単語の出現は, `1`, `22`, `333`, ..., `mmmmmmmmmmmmmmmmmmmmm`22 の 22 個である.

```
1
22
333
4444
55555
666666
7777777
88888888
999999999
aaaaaaaa10
bbbbbbbbb11
ccccccccc12
ddddddddddd13
eeeeeeeeeee14
fffffffffffff15
ggggggggggggggg16
hhhhhhhhhhhhhhh17
iiiiiiiiiiiiiiii18
jjjjjjjjjjjjjjjjj19
kkkkkkkkkkkkkkkkk20
lllllllllllllllllll21
mmmmmmmmmmmmmmmmmmmmm22
```

さて, ある文字数 (例えば i 文字) の英単語が現れる確率を, 以下のこととする. まず適当な M を定める. i 文字の英単語が現れる確率とは, ファイル中に現れる i 文字の英単語の出現回数を c_i, $C = c_1 + c_2 + \cdots + c_{M-1}$ としたときの, c_i/C のことであるとする. 本来 M は, あり得る単語長の最大値 $+1$ とすべきであろうが, 本問では $M = 21$ とする. 言い換えれば 21 文字以上の英単語は稀だとして無視することにする. 上記の例であれば, 1 文字から 22 文字の単語が一度ずつ出現しているため, $c_1 = c_2 = \cdots = c_{22} = 1$, $C = c_1 + \cdots + c_{20} = 20$ となり, i 文字の英単語が現れる確率は $1/20$ $(1 \leq i \leq 20)$ ということになる.

ファイルをすべて読み取れば c_i を求めることは簡単だが, ここではそれをより高速に調べるために, サンプリングを行うことでこれを達成する.

具体的には以下を行う.

1. 長さ M の配列 h を用意し全要素を 0 に初期化する. $h[i]$ $(i < M)$ は長さ i の英単語の出現回数を格納する (よって, $h[0]$ はずっと 0 のままのはずである).
2. 以下を多数 (n) 回繰り返す
   - 2-1. 0 以上 L 未満の (一様) 乱数を発生させる. それを x とする.
   - 2-2. ファイルの x バイト目の文字がアルファベット (大文字, 小文字) または数字でなければ何もしない
   - 2-3. アルファベット (大文字, 小文字) または数字であったら, x バイト目を含む英単語の開始位置, 終了位置を求め, そこから長さ (l とする) を求める. ただし長さが M 以上だとわかったら $l = M$ としておく.
   - 2-4. $l < M$ ならば, 長さ l の英単語が出現したことを, h に反映する
3. 配列 h に記録された値を元に, 求める確率の近似値を計算し, `double` 型の配列 `p` に格納する.

乱数を使ったサンプルで計算した値のため, `p[`i`]` が求める確率に一致している必要はないが, 乱数が実際に一様であれば, $n \to \infty$ のときに `p[`i`]`$\to$ 求める確率 となる必要がある.

以下は乱数を発生させるためのデータ構造 (`prng_t`) と, それを初期化する関数 `prng_init`, 0 以上 `L` 未満の整数をランダムに発生させる関数 (`prng_rand_int`) である.

```
#pragma once
#include <stdint.h>

typedef struct  {
  uint64_t x;
} prng_t;

void prng_init(prng_t * rg, uint64_t seed);
uint64_t prng_rand_int(prng_t * rg, uint64_t L);
```

```
#include __PRNG_H__

void prng_init(prng_t * rg, uint64_t seed) {
  const uint64_t mask = (1UL << 48) - 1;
  rg->x = seed & mask;
}

/* 0 <= x < L なる乱数を返す */
```

```
uint64_t prng_rand_int(prng_t * rg, uint64_t L) {
  const uint64_t a = 0x5deece66dull;
  const uint64_t c = 0xb;
  const uint64_t mask = (1UL << 48) - 1;
  uint64_t x = rg->x;
  uint64_t next = (a * x + c) & mask;
  rg->x = next;
  return next % L;
}
```

例えば以下のようにして使う. seed は乱数の種 (64 bit 符号なし整数) である. 種が同じなら発生する乱数の系列は同じである.

```
prng_t rg[1];
prng_init(rg, seed); /* 初期化 */
for ( ... ) {
  uint64_t x = prng_rand_int(rg);
  ...
}
```

以下の関数 word_len は, 長さ L の文字列 S の, i ($< L$) バイト目を含む英単語の長さを求める関数である. ただしそれが M 以上の場合は M を返し, 文字列 S の i バイト目がアルファベットまたは数字でない場合は, 0 を返す.

```
#include <assert.h>
#include <ctype.h>

/* S[i]を含む単語の長さを返す. ただし,
   長さがM 以上 -> M を返す.
   S[i]が単語の一部でなければ (i.e., アルファベット･数字でない) -> 0を返す */
int word_len(char * S, long i, long L, long M) {
  /* 単語の一部でない */
  if (!isalnum(S[i])) return 0;
  /* INV: S[a:b]はすべてアルファベット･数字 */
  long a = i;
  long b = i + 1;
  /* 先頭を見つける. 長さ>=M と判明したら break */
  while (a > 0 && b - a < M && isalnum(S[a - 1])) {
    a--;
  }
  assert(a == 0 || b - a == M || !isalnum(S[a - 1]));
  /* 終わり+1を見つける. 長さ>=M と判明したら break */
  while (b < L && b - a < M && isalnum(S[b])) {
    b++;
  }
  assert(b == L || b - a == M || !isalnum(S[b]));
  assert(0 <= a);
  assert(0 < b - a);
  assert(b - a <= M);
  assert(b <= L);
  for (long j = a; j < b; j++) {
    assert(isalnum(S[j]));
  }
  return b - a;
```

```
}
```

以下の関数 sample_word_len は, ファイル (filename) に含まれる英単語長の確率分布の近似値を p に返す関数の一部である. ただし L は, filename の大きさ (バイト数) である. M 文字以上の単語は無視する. n 個の位置をサンプリングしてこれを求める.

```
#define _GNU_SOURCE
#include <err.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>

#include __PRNG_H__
int word_len(char * S, long i, long L, long M);

/* filename の最初の L バイト中に現れる単語長の分布を p
   に返す. M バイト以上の単語は無視する(なかったものとみなす).
   n 個のサンプルをとる. そのための乱数の種がseed */
void sample_word_len(char * filename, long L, long n, uint64_t seed, long M, double p[M]) {
  int fd = open(filename, O_RDONLY);
  if (fd == -1) err(1, "open(%s)", filename);
  ...;
  prng_t rg[1];
  prng_init(rg, seed);
  long h[M];
  for (int l = 0; l < M; l++) h[l] = 0;
  for (long i = 0; i < n; i++) {
    /* 0 <= x < L なる乱数発生 */
    long x = prng_rand_int(rg, L);
    /* x バイト目を含む単語の長さを求める */
    int l = ...;
    if (0 < l && l < M) {
      /* 長さl の単語の出現を記録 */
      h[l] += 1;
    }
  }
  close(fd);
  /* h を元に配列 p に答えを格納 */
  ...;
}
```

サンプリングする位置は, 乱数の種を seed にした prng_t 構造体 rg[1] に対し n 回, prng_rand_int(rg, L) を呼び出したときの返り値 $x_0, x_1, \cdots, x_{n-1}$ とする. 特に, seed が同じである限り全く同じ場所からサンプルを取り出さなくてはならず, したがって結果も同じにならなくてはならない.

これらを使った以下のプログラムは,

```
#include <err.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
```

```c
#include <unistd.h>
#include __PRNG_H__

enum { M = 21 };

void sample_word_len(char * filename, long L, long n, uint64_t seed, long M, double p[M]);

/* ファイルのサイズを求める */
long get_file_size(char * filename) {
  struct stat sb[1];
  if (stat(filename, sb) == -1) {
    err(1, "stat(%s)", filename);
  }
  return sb->st_size;
}

/* 時刻 (ns 単位)を得る */
long cur_time_ns() {
  struct timespec ts[1];
  clock_gettime(CLOCK_REALTIME, ts);
  return ts->tv_sec * 1000 * 1000 * 1000 + ts->tv_nsec;
}

int main(int argc, char ** argv) {
  int i = 1;
  char * filename = (i < argc ? argv[i] : "test2.txt"); i++;
  long n          = (i < argc ? atol(argv[i]) : 10000); i++;
  uint64_t seed   = (i < argc ? atol(argv[i]) : 1234567890); i++;
  long L          = get_file_size(filename);
  /* 実行 + 測定 */
  double p[M];
  long t0 = cur_time_ns();
  sample_word_len(filename, L, n, seed, M, p);
  long t1 = cur_time_ns();
  /* 結果の表示 */
  for (int l = 1; l < M; l++) {
    printf("%2d %.4f\n", l, p[l]);
  }
  printf("took %.6f sec to draw %ld samples from %s\n",
         (t1 - t0) * 1e-9, n, filename);
  return 0;
}
```

./sample_word_len *filename n seed*

として起動すると, ファイル名 *filename* から n 個の位置をサンプリングして, 単語長の分布を求める.
以下の問いに答えよ.

(1) 上記で示した sample_word_len を, 欠けているところを補って完成させよ (注: おそらく変更・加筆が必要であろう部分を, ... で示してあるが, 必ずしも他の場所を変更・加筆してはいけない・する必要はない, ということを意図していない).

(2) `prng_rand_int` を Pthread の mutex API を使ってスレッドセーフな関数にせよ. スレッドセーフにするとは `prng_rand_int` を, 複数のスレッドが呼び出しても 1 スレッドの場合と同じ動作をするという意味である. つまり, 全スレッドが合計で同じ回数呼び出したときに返される値の多重集合[1]が, スレッド数によらず同じであることを意味する. 必要ならば, `prng_t` 構造体も変更せよ.

(3) `prng_rand_int` を non-blocking なスレッドセーフ関数にしたい. ここで non-blocking とは, あるスレッドが運の悪いタイミングで実行を preempt や suspend されて長時間停止したとしても, 他のスレッドが進捗できる (乱数を取得できる) ことを言う. mutex を使った実装はなぜ non-blocking ではないのかを説明せよ.

(4) `prng_rand_int` を non-blocking なスレッドセーフ関数にせよ.

(5) `sample_word_len` 関数に必要な変更を行って, 複数のスレッドが並列にサンプリングを行うようにせよ. 同じ引数を与えれば, スレッド数によらず, 元のプログラムと全く同じ結果が出るようにすること.

(6) そのように変更したプログラムを, 4 個のプロセッサ (コア) を搭載したマシンで実行した. このときの性能について述べたものとして, 正しいものは以下のどれか? ただし, ファイルはキャッシュされていない状態であるとする.

   (a) スレッド数を 2 以上にしても, スレッド数 1 のときと比べて殆ど性能向上しない.

   (b) 4 スレッド程度まで性能向上するが, それ以上はほとんど性能向上しない.

   (c) 4 スレッドよりずっと多くのスレッド数まで性能向上する.

(7) 前問について, そうなる理由を述べよ.

(8) ファイルがキャッシュされていない状態で, スレッドを用いずにこのプログラムの性能を同様に向上させる方法がある. その方法と, その方法でなぜ性能が向上するのか述べよ.

[1] 各要素が現れる回数までを含んだ集合

3

Linux は vruntime という値を各スレッドに対して保持しながらスケジューリングを行っている.

(1) Linux は vruntime をどのように維持しているか (各スレッドの vruntime をどのように更新しているか) 述べよ.

(2) Linux は vruntime をスケジューリングにどのように利用しており, そしてそれによって, どのような望ましい性質を達成しているかを述べよ.

4

Unix OS がプログラムの起動を高速化するのに行っているいくつかの工夫を, プログラムの起動に必要な一連のステップに沿って述べよ.

問題は以上である