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;
```

```
10
11
12    while (b->x < next) /* 何もしない */  ;
13
14
15
16
17  }
```

この元で N 個のスレッドが以下の通り実行する.

```
1  int main(int argc, char ** argv) {
2    int N = omp_get_max_threads(); /* スレッド数を得る */
3    barrier_t b[1];
4    barrier_init(b, N);
5    /* 動作チェック用配列 */
6    long a[N];
7    for (long j = 0; j < N; j++) a[j] = -1;
8  #pragma omp parallel
9    {
10     int idx = omp_get_thread_num();
11     a[idx] = idx;               /* 動作チェック用 */
12     barrier(b);
13     for (long j = 0; j < N; j++) {
14       assert(a[j] == j);        /* 動作チェック */
15     }
16   }
17   printf("OK\n");
18   return 0;
19 }
```

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) {
  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();
    for (long i = 0; i < n; i++) {
      a[idx] = idx + i * N;     /* 動作チェック用 */
      barrier(b);
      for (long j = 0; j < N; j++) {
        assert(a[j] == j + i * N); /* 動作チェック */
      }
      barrier(b);
    }
  }
  printf("OK\n");
  return 0;
}
```

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);
```

```
5
6     record * R = malloc(sz);
7
8     read(fd, R, sz);
9
10    long x = binsearch(R, n, key);
11
12    free(R);
13
14    close(fd);
15
16    return x;
17  }
```

(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

学生証番号		氏名		

1	(1)	(a)		(b)		(c)	
		(d)		(e)		(f)	
		(g)		(h)		(i)	
		(j)		(k)		(l)	
		(m)		(n)		(o)	
		(p)					
	(2)						
	(3)	機能の名称					
		実現方式の概要					

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		
		(c)	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)	typedef struct {
		  volatile long x;
		  volatile long next;
		  long N;
		
		} barrier_t;
			void barrier_init(barrier_t * b, long N) {
			
			  b->x = 0;
			
			  b->next = N;
			
			  b->N = N;
			
			}
				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 */ ;
				
				}	

	(3)	typedef struct {
		  volatile long x;
		  volatile long next;
		  long N;
		
		} barrier_t;
			void barrier_init(barrier_t * b, long N) {
			
			  b->x = 0;
			
			  b->next = N;
			
			  b->N = N;
			
			}
				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 */ ;
				
				}	

		正しいことの説明			

3	(1)		
	(2)	グラフ概形	なぜそのようなグラフになるのかの説明

	(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);
		
		long x = binsearch(R, n, key);
		
		free(R);
		
		close(fd);
		
		return x;
		}	

	(4)	グラフ概形	なぜそのようなグラフになるのかの説明
	(5)	グラフ概形	なぜそのようなグラフになるのかの説明