平成23年度オペレーティングシステム期末試験

2012年2月7日

注意事項

- **問題は 3 問, 6 ページある.**
- **1 枚の解答用紙に 1 問解答する (複数の問題の解答を 1 枚に混ぜたり, 1 問の解答を複数の用紙にまたがって書いたりしない) こと**
- **各解答用紙にはっきりと, どの問題に回答したのかを明記すること**
- **提出時は, 3 枚の答案を問題 1, 2, 3 の順に重ねてホチキスでとめること**

1

現在の汎用 CPU には, メモリ管理ユニット (MMU) が搭載され, 様々な目的に使われている.

(1) CPU がメモリアクセス命令を発行した際, MMU が何を行うか述べよ.

(2) OS はプロセス間のメモリを分離している, すなわち, あるプロセスが他のプロセスのデータを読んだり書いたりできないようにしている. このために OS が MMU をどのように利用しているかを述べよ.

(3) Unix OS が, プロセスを生成するシステムコール fork() を高速化するために, MMU をどのように利用しているかを述べよ.

(4) OS が, 多数のプログラムが利用するシステムライブラリのロードを高速化し, 使用メモリを節約するために, MMU をどのように利用しているかを述べよ.

解答例 (1)

- TLB, ページテーブルを参照して, アクセスされたアドレスのアクセス権を検査する
- 違反していれば保護例外を発生させる
- 違反していなければ仮想アドレスを物理アドレスに変換する
- 物理アドレスが不在となっている場合はページフォルトを発生させる

(2) 特に指定されない限り, プロセスごとに異なる物理メモリを割り当て, 各プロセスごとのページ表にそれらの (異なる) 物理メモリを登録する. 一部の領域に対しては, TBL/ページ表で書き込みを禁止しつつ物理メモリを共有する場合もある. 書き込みが起きたら, 実際に書き込み禁止であれば保護違反を発生させ, 実際には書き込みが許可されたページであれば書き込み時に物理ページをコピーすることにより, 必要に応じて物理ページを分離する.

(3) fork() のセマンティクス (仕様) では, 呼び出したプロセスのアドレス空間をコピーした子プロセスが生成されるが, その際ページの中身をコピーするのではなく, TLB・ページ表だけを, 保護属性を書き込み禁止にした上でコピーする. その後書き込みが起きたページだけを, 必要に応じてコピーしていく (コピーオンライト) ことで, fork() の際に生じるメモリコピーの量を減らして

(4) システムライブラリは mmap を用いてプロセスのアドレス空間にマップされる. その際, 全プロセスが同じ物理メモリを書き込み禁止で共有することにより, 使用メモリは節約される上, プロセスの起動ごとにファイルをプロセスのアドレス空間にコピーする必要もなくなる.

2

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, キーの昇順に整列された record の配列 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. 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. また, a 中の要素のキーはすべて異なる.

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

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

正確には, binsearch(a, n, k) は以下の値を返す.

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

(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, 一様な確率で抽出し, それらをキーとして a を作る. その配列 a をファイル A に格納する. 0 以上 M 未満のキーをひとつ一様な確率で選び, そのキーを A 中から探索する以下のプログラムを考える.

```
int binsearch_file(int n) {
  int fd = open(A, O_RDONLY);
  int sz = n * sizeof(record);
  record * R = malloc(sz);
  read(fd, R, sz);
```

```
  k = 0 以上 M 未満の一様乱数;
  return binsearch(R, n, k);
}
```

ただし簡単のため, エラーチェックなどは省略している.

(2) 横軸を配列 a の要素数 n, 縦軸を `binsearch_file(`n`)` 一回の実行にかかる時間としたグラフを描け. また, 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. 横軸はは, a の大きさ (バイト数; すなわち nP) が主記憶より大きくなるところまで描くこと.

(3) 上記のプログラムで malloc/read を mmap に置き換えた, 以下のプログラムに対して, (2) と同様のグラフを描け. (2) のグラフと対比できるよう, 同じグラフ中に書き入れよ.

```
int binsearch_file_mmap(int n) {
  int fd = open(A, O_RDONLY);
  int sz = n * sizeof(record);
  record * R = mmap(0, sz, PROT_READ, MAP_PRIVATE, fd, 0);
  k = 0 以上 M 未満の一様乱数;
  return binsearch(R, n, k);
}
```

解答 (1)

$\lfloor \log(n) \rfloor$ 以上 $\lceil \log(n) \rceil$ 以下.

($\lfloor x \rfloor$ は x 以下の最大の整数, $\lceil x \rceil$ は x 以上の最小の整数を表す)

注: あまり細かいところにはこだわらずに, $\log(n)$ などでも正解.

(2)

(1) 途中まである傾きで n に比例

(2) 主記憶/2 を超えたところで傾きが急激に大きくなる

(3)

(1) $\log(n)$ のままずーっと

(2) 実際に触る分しか主記憶を必要としない

3

スレッド間でデータ (簡単のため，int 型の整数とする) を受け渡しするキュー (Queue) を操作する二つの手続き get と put があるとする．

- get(q) はキュー q からデータを一つ取り出す．もちろん空のキューからデータを取り出すことはできない．その場合，データが put によって一つ挿入されるまで待つ．
- put(q，x) はキュー q に一つのデータ x を挿入する．キューには一定の容量 c があり，満杯のキューにデータを挿入することはできない．つまり，q にデータがすでに c 個格納されていれば，一つのデータが get によって取り出されるまで待つ．

簡単のために $c = 1$ とした上で，以下の構造体や関数定義の$\boxed{\cdots}$部分を補う形で，(容量 1 の) キューの実現方法を考える．ただし, 実現に当たっては任意個のスレッドが, 任意のタイミングで put/get を呼び出すことができる, 汎用的な実現方法を考える．

```
typedef struct
{
  int n;        /* 格納されているデータ数．0 または 1 */
  int data;     /* n=1 のとき，格納されているデータ */
  ···
} * Queue;

void put(Queue q, int x) {
  ···
}

int get(Queue q) {
  ···
}
```

以下の問いに答えよ．

(1) 大島さんは, 以下のようにすれば良いと考えた．

```
typedef struct
{
  int n;        /* 格納されているデータ数．0 または 1 */
  int data;     /* n=1 のとき，格納されているデータ */
} * Queue;

void put(Queue q, int x) {
  while (q->n != 0) ;
  q->data = x;
  q->n++;
}
```

```
int get(Queue q) {
  while (q->n == 0) ;
  q->n--;
  return q->data;
}
```

このプログラムの問題点について, 以下が当てはまっているか否かを答えよ. 当てはまっている場合, 問題となる具体的な実行例を示せ.

(a) 値が正しく受け渡されないことがある

(b) 値が正しく受け渡されたとしても, 性能が著しく低いことがある

(2) そこへ黒沢さんがやってきて正しいプログラムを教えてくれた. それを書け. スレッドの同期のための標準的な API (排他制御，条件変数など) を適宜用いよ．プログラムは C 言語風の擬似コードで書くことを想定しているが，厳密な文法や API の用法 (引数の数や順番など) にはこだわらないので，適宜文章で説明を補え.

解答 (1)

(a) 当てはまる. get を呼ぶスレッドが一つ, put を呼ぶスレッドが一つ動いているとする. 前者を G, 後者を P と呼ぶことにする. 例えば P が put(a), put(b) を続けて呼んだとすると, 以下のような実行順序で, G が a を受け取れないことがありうる.

```
        /* 初期状態:  q->n == 0 */
P  :  while (q->n != 0);
P  :  q->data = a;
P  :  q->n++; /* q->n = 1 */
G  :  while (q->n == 0);
G  :  q->n--; /* q->n = 0 */
P  :  while (q->n != 0);
P  :  q->data = b;
G  :  return q->data; /* return b */
```

(b) 当てはまる. たとえば 1 CPU しかない計算機で上記二つのスレッドを動かした場合,

```
        /* 初期状態:  q->n == 0 */
G  :  while (q->n == 0);
```

初期状態 (q->n == 0) から, 最初に G が上記 while ループに達すると, そこで OS がスレッドを切り替えるまで G はループを回り続ける. 一回の受け渡しに 1 time quantum 時間がかかることになり, 非常に遅くなる.

(2)

```
typedef struct Queue_
{
  int n;
  int data;
  pthread_mutex_t m;
  pthread_cond_t c;
} * Queue;

void put(Queue q, int x) {
  pthread_mutex_lock(&q->m);
  while (q->n != 0) pthread_cond_wait(&q->c, &q->m);
  q->n++;
  q->data = x;
  pthread_cond_broadcast(&q->c);
  pthread_mutex_unlock(&q->m);
}

int get(Queue q) {
  int x;
  pthread_mutex_lock(&q->m);
  while (q->n == 0) pthread_cond_wait(&q->c, &q->m);
  q->n--;
  x = q->data;
  pthread_cond_broadcast(&q->c);
  pthread_mutex_unlock(&q->m);
  return x;
```

```
}
```

問題は以上である