平成20年度オペレーティングシステム期末試験

2009年2月3日

問題は3問，4ページある．

1

オペレーティングシステムのスレッド (プロセス) スケジューラは, 一般に以下の条件を満たすようにスレッドをスケジューリングする.

1. CPU を無駄にしない—実行可能なスレッドがいる限り CPU をそれらのスレッドのうちのどれかに割り当てる
2. 公平である—実行可能なスレッドに割り当てられる時間が極端に偏らない
3. (a)

(1) (a) には, デスクトップやラップトップ PC において特に重要な, ある条件が入る. どのような条件か述べよ.

(2) これらすべてを両立させることは簡単ではない. 以下のような方針の単純なスケジューラは, どのような点が潜在的な欠点となるか, **上記の条件のうち, どれがどう損なわれるのかをはっきりさせながら**, 答えよ.

スケジューラ A:
- 実行可能スレッドのキューを管理する.
- 現在実行中のスレッドはキューの先頭にいる.
- タイマ割り込み時にキューの先頭にいるスレッドをキューの最後尾に回し, その後でキューの先頭にいるスレッドを実行する.
- 実行中のスレッドがブロックしたら, キューから取り外して先頭のスレッドを実行する.
- ブロックしていた (実行可能でなかった) スレッドが実行可能になったらキューの**末尾**に加える.

スケジューラ B: スケジューラ A とほぼ同じだが, ブロックしていた (実行可能でなかった) スレッドが実行可能になったらキューの**先頭**に加え, そのスレッドを即座に実行する.

解答例:

(1) 対話的プログラムの応答性をよくする

(2) スケジューラ A は, 条件 3 (対話的プログラムの応答性をよくする) を損なう可能性がある.

例えば実行可能なスレッドが多数, 対話的なスレッド (ほとんどの時間ブロックしているスレッド) が少数混在しているときに, 対話的スレッドの応答時間— 実行可能になってから実際にスケジュールされるまでの時間— が実行可能なスレッド数に比例して長くなる.

スケジューラ B は, 公平さを損なう可能性がある.

例えば, ブロックから目覚めた後, 次のタイマ割り込みが来るより少し前の時間まで走っては再びブロックすることを繰り返すようなスレッドに対して, 全体の半分くらいの実行時間を割り当ててしまう. (そのようなスレッドが二つあれば, その二つだけでほぼすべての CPU 時間を使い果たす).

コメント:

スケジューラ A に対しても, 「公平さを損なう」と解答し, その理由が, 「対話的スレッドがキューの末尾に回るのでなかなか実行されない」というものもあった. なかなか実行されないというのはその通りかもしれないが, このスケジューラに関しては, それはどのスレッドも同じ事情であるし, 基本的に「全員後ろに並ぶ」というのは不公平ではない.

対話的スレッドはそもそもほとんど CPU 時間を使っていないのに, それすら考慮せずに後ろに回るのは可哀想過ぎる (故に「不公平」である) という解答もあった. 例えば 10ms ごとにタイマ割り込みが起きるとして, 0.1ms 走ってブロックした人が 1 秒後に目覚めたときに最後尾に回ったとすると, 他の人は 10ms 使ったら「並びなおし」, この一だけは 0.1ms 使っただけで「並びなおし」となるのは「不公平」だということである.

これは考慮に値するのだが, そもそも 0.1ms 使っただけでブロックしたのは, その人が勝手にやったことである. その分を後から「埋め合わせ」しないことを, 即「不公平」と言えるのかは議論を要する.

コメント:

また, 「今年度の問題ではあの, バカな人が何人か登場してバカな会話を繰り広げる問題はないようだが?」というご指摘をいただいた.

真相は, 問題を考える時間がなかなかとれず, よいストーリーがが思いつかなかった, という, ただそれだけことであった (反省).

2

ページ置換アルゴリズムについて考察するため, 以下のような設定を考える.

- n 個のページ $1, 2, \cdots, n$ があり, それらのうち, 常に $m(< n)$ 個だけが同時に物理メモリ上に存在する.
- プロセスが順番にページにアクセスをする.
- プロセスが, 物理メモリ上に存在していないページに対するアクセスを行うとページフォルトが発生する. その際, 物理メモリ上からどれかのページを削除し, たった今アクセスされたページを物理メモリ上に収容する. これを**ページ置換**と呼ぶ.

今, n 個のページを $1, 2, \ldots, n, 1, 2, \ldots, n, \ldots$ のように循環的に (永遠に) アクセスするプロセスを考える. 簡単のためこれ以外のプロセスはないものとする.

1. ページフォルト時に, 物理メモリ上に存在するページの中で, 最後にアクセスされたのがもっとも遠い過去であるようなページを削除するアルゴリズム (Least Recently Used 置換) を考える.

   十分時間がたったあとページフォルト率 (十分長い連続したアクセスに対するページフォルトの発生割合) はいくらになるか?

2. この特定のアクセスパターンについて, ページフォルト率を最小にするページ置換では, ページフォルト率はいくらになるか?

3. 実際のオペレーティングシステムに於いては, Least Recently Used 置換またはそれを近似したようなアルゴリズムが使われることが多い. これはどのような考え方に基づくものか述べよ.

解答例:

(1) 1.

ページ i へのアクセスでページフォルトが生じたとする．この時, もしページ $(i+1) \bmod n$ がページ上にあれば, それが LRU ページであるからこれが追い出される．したがってページ i へのアクセス後, ページ $(i+1) \bmod n$ は物理メモリ上になく, したがって直後のアクセスもまたページフォルトを起こす.

つまり, 一度ページフォルトが起きれば, 以降のすべてのアクセスにおいてページフォルトが起こる．今, $m < n$ だったから, 最初の $(m+1)$ アクセス以内に最低一回はページフォルトがおき, 以降はずっとページフォルトが起きる. つまりページフォルト率は 1 である.

(2) $(n-m)/(n-1)$

これに関する説明は以下でコメントを混えながら述べる.

コメント:

未来のアクセス系列が分かっている場合のアルゴリズム (**オフラインアルゴリズム**という) では, 常に最適であるものが知られており, それは「次に使われるのが最も遠い未来であるようなページを取り除くアルゴリズムである」, ということは授業で述べた. そのようなページを Furthest Future Used (FFU) ページという.

小さな n, m に対して実際にそのアルゴリズムを適用して, 系の状態が一回りするまでの様子を観察すれば, 結果自体の予想はつく.

なお, 非常に多かった解答として, ページフォルト率を

$$(n-m+1)/n$$

であるとし, その理由は以下, というのがあった.

> この場合の最適アルゴリズムは, 物理メモリ m ページのうち, $(m-1)$ ページは常に固定しておき, 残りの 1 ページを, 置き換え用に使う, というものである. 故に, 連続した n ページへのアクセス中, $(m-1)$ ページはページフォルトを起こさず, 残りの $(n-m+1)$ ページでページフォルトが起きる.

この例に関しては一見正しいような気もするし, 最適値とは, 数値的にはあまり差がない. しかし, ($m > 1$ の場合) 最適でないことには違いはない. 念のため:

$$\frac{n-m+1}{n} - \frac{n-m}{n-1} = \cdots = \frac{m-1}{n(n-1)} > 0$$

差があるといってもせいぜい $1/n$ 程度の差だから数値的には似たり寄ったりだが, 何が最適であるのかは授業で述べたのだから, そのアルゴリズムのページフォルト率を求めよう, というところまではやってほしかった (ただし上記にも部分点は与えている).

FFU アルゴリズムのページフォルト率を厳密に計算するにあたっては, どこからどこまでを 1 周期と見れば考えやすくなるのかを発見することがポイント. n 個のページをアクセスしているのに, 周期が $(n-1)$ であるところは多少気づきにくい.

以下は厳密な証明で, 答案としてはここまでは要求していない.

> **補題:** ページ i へアクセスする直前のある時点での物理メモリ上のページ集合を R (ただし, $i-1 \in R$ $(*)$ を仮定する) とし, そこから始まる $(n-1)$ 回の連続したアクセス $A = i, i+1, \ldots, i+n-2$ を考える. このとき,
>
> $$p \in R \iff A \text{ 中で}, p \text{ へのアクセスによるページフォルトが起きない}$$
>
> が成り立つ.

証明:

- $\Rightarrow$) $p \in R$ とする. $p = i-1$ のとき, 主張が成り立つことは自明である ($i-1$ は A でアクセスされないから, 当然それへのアクセスによるページフォルトは起きない). 以下 $p \neq i-1$ とする. すると, $i, \ldots, p-1$ までのアクセスが行われる間, p が FFU になることはない. 実際, ページ i がアクセスされる直前には $i-1$ が FFU であり ($i-1 \in R$ を仮定している事に注意), その後ページ j がアクセスされた直後は, 明らかに j が FFU である. よって, p がアクセスされる際, p は物理メモリ上にありページフォルトを起こさない.

- $\Leftarrow$) 対偶を言う. $p \notin R$ のとき, 明らかに p へのアクセスはページフォルトを起こす.

どんなページ集合も, 1 回アクセスを行えば, 仮定 ($*$) を満たす状態になるから, それ以降任意の $(n-1)$ アクセス中のページフォルトの回数は $n-1-(\#R-1) = n-m$ 回となる ($\#R-1$ の, -1 は, A 中でアクセスされないページ $i-1$ の分). すなわちページフォルト率は $(n-m)/(n-1)$ となる.

(3) 一般に, ページ置換では, 次に使われるのが最も遠い未来であるようなページ (未来において一切使われないページは, 使われるのがどのページよりも遠いと見なす) を置換するのが良いことが知られている. LRU は, アクセスの時間的局所性—最近使われたページはまたすぐに使われる傾向にある—を仮定して, 近い将来使われるのは, 近い過去において使われたページであろう, という考えに基づく.

3

ファイルに含まれる改行文字の数を数える以下の 3 つのプログラムを考える. F はファイル名, N はそのバイト数である.

1. read システムコールを 1 度だけ使ってファイルの中身をすべて配列に読み込む. 例えば以下.

```
1:  int newline_count_A() {
2:    int fd = open(F, O_RDONLY);
3:    int s = 0;
4:    int i;
5:    char * a = (char *)malloc(N);
6:    read(fd, a, N);
7:    for (i = 0; i < N; i++) {
8:      if (a[i] == '\n') s++;
9:    }
10:   close(fd);
11:   return s;
12: }
```

2. read システムコールを繰り返し使ってファイルの中身を 1 バイトずつ読み込む. 例えば以下.

```
1:  int newline_count_B() {
2:    int fd = open(F, O_RDONLY);
3:    int s = 0;
4:    int i;
5:    char a[1];
6:    for (i = 0; i < N; i++) {
7:      read(fd, a, 1);
8:      if (a[i] == '\n') s++;
9:    }
10:   close(fd);
11:   return s;
12: }
```

3. mmap システムコールを 1 度だけ使ってファイルの中身をメモリ上にマッピングする. 例えば以下.

```
1:  int newline_count_C() {
2:    int fd = open(F, O_RDONLY);
3:    int s = 0;
4:    int i;
5:    char * a = mmap(NULL, N, PROTO_READ, MAP_PRIVATE, fd, 0);
6:    for (i = 0; i < N; i++) {
7:      if (a[i] == '\n') s++;
8:    }
9:    close(fd);
10:   return s;
11: }
```

newline_count_A, newline_count_B, および newline_count_C の方式でファイルの改行数を数えるプログラムを作りそれぞれ, A, B, C と呼ぶことにする.

2GB 程度の主記憶を持つコンピュータの上で 300MB 程度のファイル F を用意して, A, B, C の経過時間測定を行った. 以下の問いに答えよ.

(1) まず以下の手順で計測を行った.

1. F が主記憶上のキャッシュにのっていない状態を作る.
2. A を一つだけ起動し, 経過時間を記録する.
3. F が主記憶上のキャッシュにのっていない状態を作る.
4. B を一つだけ起動し, 経過時間を記録する.
5. F が主記憶上のキャッシュにのっていない状態を作る.
6. C を一つだけ起動し, 経過時間を記録する.

A, B, C のうち一つだけ, 他の二つと比べて圧倒的に (桁違いに) 遅いものがあった. それはどれか? 理由とともに述べよ.

(2) それを簡単に改善する方法がある. 擬似コードを用いながらその方法を述べよ. こうして得られたプログラムを Z とする.

(3) (1) で圧倒的に遅かったものを除く二つのプログラム (仮に X, Y と書く) と, (2) で得た X に対して以下の手順で計測を行った.

1. F が主記憶上のキャッシュにのっていない状態を作る.
2. X を同時に p 個起動し, p 個のプロセスが終了するまでの経過時間を記録する.
3. F が主記憶上のキャッシュにのっていない状態を作る.
4. Y を同時に p 個起動し, p 個のプロセスが終了するまでの経過時間を記録する.
5. F が主記憶上のキャッシュにのっていない状態を作る.
6. Z を同時に p 個起動し, p 個のプロセスが終了するまでの経過時間を記録する.

p を小さな値から徐々に増やして上記の実験をしたところ, p がある値を越えるまではどれも大差のない時間であったが, ある値を越えたあたりから, 一つのプログラムが極端に遅くなり始めた.

そのプログラムとはどれのことか? 理由とともに述べよ. また, 「ある値」は大体どの程度の値と予想されるか.

(4) 「F が主記憶上のキャッシュにのっていない状態を作る」には, 具体的にはどのようにすればよいか? 簡単な方法を述べよ.

問題は以上である

解答例:

(1) B. 1 バイト読むごとにシステムコールを発行しているのでそのオーバーヘッドが大きい. これに対して A, C はシステムコールを 1 度ずつしか発行していない.

コメント:

ほとんどの人が正解だったがそういう人の中でも非常に多かった事実誤認は以下のようなものである. (事実誤認とはいえ, この問いに対する答えとしては的外れではないので, 相応の部分店は与えている).

> (誤解) read システムコールは, 発行する度に, ディスク I/O が発生する. したがってプログラム B は, 1 バイト読み込む度にディスク I/O が発生し, ヘッドの位置合わせなどを含めた時間のかかる操作が行われる.

これは誤解で, そもそもディスクは 1 バイト単位での読み書きなどできないので, いやでも実際のディスクとのやりとりは, 512 バイト, 1024 バイトなどのブロック単位となる. 仮にできたとしても, read のリクエストに対して, ある程度まとまった単位でディスクとのやりとりを行うのは OS にとっては自然なことである. また, この例のような, ファイルの先頭から終わりまでの逐次的な読み出しに対しては, 適当なところで「先読み」が発動され, ますますディスクとやりとりする部分の効率は, 良くなっていくのである.

ではなぜ B が遅いのか? その理由はもっと単純な事で, そもそも (ディスクとの IO が仮に発生しなくても), 「システムコールで OS を呼び出して, プロセスと OS の間でデータのやりとりする」こと自体のオーバーヘッドが非常に多数回発生しているからである.

(2)

解答例 1: 1 バイトずつ読むのではなく, 100 バイト, 1,000 バイトなどの適当な一定サイズごとに, read システムコールを発行する.

解答例 2: fread のような, バッファリングを行うライブラリを用いる. 実際には, fread 関数の中で, 上記と同じ事 (適当なバッファサイズで read システムを発行する).

コメント:

解答例 2 のパターンが多かったが, 元を正せばそれでよい理由は, 一度に OS にリクエストするデータのサイズを増やして, read システムコールの呼び出し回数を減らしているからであり, より明解なのは解答例 1 の方です.

(3) A. A. は, 各プロセスが, ファイルサイズと同じだけの物理メモリ (N バイト) を消費する. それ以外にファイルのキャッシュのためにファイルサイズと同じだけの物理メモリを消費する. したがっておおよそ, $(p+1)N$ が物理メモリを上回るかいなかで性能が不連続に変化する. 答えるべき境目の値としては, 6 か 7 程度.

(4)

解答例 1: 物理メモリと同程度か, それを上回るメモリを確保し, 確保した全ページに一度ずつ書き込みを行うプログラムを実行する.

解答例 2: 物理メモリと同程度かそれ以上の大きさのファイルを読み込む.

コメント:

私にとっては「珍回答」とうつるのだが, 非常に多かった解答は「再起動をする」というものであった. 確かにその通りで再起動後にはメモリに F はのっていないことだろう. だが, F を追い出すには他のファイルを

キャッシュに置いたり, プログラムのデータとしてメモリを使えば良いので, そちらの方がうんと話が早いし, かかる時間も短い. 再起動は数分かかるだろうが, これならせいぜい 10 秒かそこらだ.

他に思いつかなかったので, いわばユーモア, とんち, として書いたものだと解釈させていただいた.

しかしこれには考え込んでしまった. 「簡単な方法」と述べたつもりだったのだが, 言われてみれば確かに, プログラムを書いたりコマンドを実行することよりも, リセットボタンを押す方が「簡単」, という人もいるだろう.

それでも, この答えがとてもインテレクチャルなものではない, ということには同意していただけるものと仮定して, これは間違いとしている.