平成15年度オペレーティングシステム試験 解答例と解説

2004年2月16日 実施

1

注: 問題文のプログラム中22行目と23行目の行番号が逆であった．以下ではあくまで問題文につけられた行番号で示している．したがって以下で23ととあるのは，あくまで`printf("sum = %d\n", sum);`のある行を指している．

(1) [**配点 16**] 0, 10, 20, 30の可能性がある．それぞれ，たとえば次のような順序で実行されたときにおきる．数字は行番号を示す．

0になる場合: 7, 8, 14, 15, 21, 23, $\cdots$

10になる場合: 7, 8, 9, 14, 15, 21, 23, $\cdots$

20になる場合: 7, 8, 14, 15, 16, 21, 23, $\cdots$

30になる場合: 7, 8, 9, 14, 15, 16, 21, 23, $\cdots$

つまり，変数doneが更新された後，sumがF, Gによって(各1度)更新される前に運の悪いタイミングでスレッドCに実行が移ると，while文を抜ける条件が成立してしまう．10となるのはFの更新後，Gの更新前にCに移った場合，20となるのはFの更新前，Gの更新後にCに移った場合，0となるのはF, Gの更新前にCに移った場合である．

また，上とは本質的に異なったストーリとして，10, 20になるのに以下のようなストーリもある．

10になる場合: 7, 8, 9(1), 14, 15, 16, 9(2), $\cdots$

20になる場合: 14, 15, 16(1), 7, 8, 9, 16(2), $\cdots$

ただし，9(1)は9行目のうち，`sum`を読み出す部分まで，9(2)は残りの部分を指す．16(1), 16(2)も同様である．つまり，`sum`への加算が不可分に行われないために，F, G どちらかのスレッドによる加算が失われた場合である．

このほかに，永遠に条件が成立しない場合もある．それは，

ケース 1: 7, 8(1), 14, 15, 16, 8(2), 9, $\cdots$

ケース 2: 14, 15(1), 7, 8, 9, 15(2), 16, $\cdots$

のようにして，doneへの加算のひとつが失われた場合である．

注: もちろん解答としては，ストーリをひとつだけ例示してあればよい．最後の「永遠に成立しない場合」は指摘していなくても良い．

(2) [**配点 10**]

- 大域変数としてmutexロック(`m`)を用意する．

– 8 行目と 9 行目を入れ替える．
– 15 行目と 16 行目を入れ替える．
– 8-9 行目と，15-16 行目を`m` の lock/unlock で囲む．

以下は pthread API を用いた．追加の行の番号を XXX:とした．入れ替えた行の番号には ’ をつけた．

```
 1: 大域変数:
XXX: pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
 2: int sum = 0;          /* f(x) と g(x) の和 */
 3: int done = 0;         /* 終了したスレッド数 (0, 1, 2 のどれか) */
 4:
 5: スレッド F:
 6: {
 7:   vf = f(x);
XXX:   pthread_mutex_lock(&m);
 9’:   sum = sum + vf;
 8’:   done = done + 1;
XXX:   pthread_mutex_unlock(&m);
10: }
11:
12: スレッド G:
13: {
14:   vg = g(x);
XXX:   pthread_mutex_lock(&m);
16’:   sum = sum + vg;
15’:   done = done + 1;
XXX:   pthread_mutex_unlock(&m);
17: }
18:
19: スレッド C:
20: {
21:   while (done < 2) { }
23:   printf("sum = %d\n", sum);
22: }
```

(3) [**配点 14**] 同期条件 (done ≥ 2) が不成立の間も，スレッド C は OS からは実行可能なスレッドとして認識され，CPU 時間が F や G と同様に割り当てられる．たとえば他にスレッドがおらず，F, G もずっと実行可能であれば 1/3 の CPU 時間が割り当てられ，これが無駄である．

解決のためには同期条件が不成立の間，C がブロックするようにする．たとえば条件変数を用いる．

```
 1: 大域変数:
XXX: pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
YYY: pthread_cond_t c = PTHREAD_COND_INITIALIZER;
 2: int sum = 0;          /* f(x) と g(x) の和 */
 3: int done = 0;         /* 終了したスレッド数 (0, 1, 2 のどれか) */
```

```
 4:
 5: スレッド F:
 6: {
 7:   vf = f(x);
XXX:   pthread_mutex_lock(&m);
 9':   sum = sum + vf;
 8':   done = done + 1;
XXX:   pthread_mutex_unlock(&m);
YYY:   if (done == 2) pthread_cond_broadcast(&c);
10: }
11:
12: スレッド G:
13: {
14:   vg = g(x);
XXX:   pthread_mutex_lock(&m);
16':   sum = sum + vg;
15':   done = done + 1;
XXX:   pthread_mutex_unlock(&m);
YYY:   if (done == 2) pthread_cond_broadcast(&c);
17: }
18:
19: スレッド C:
20: {
YYY:   pthread_mutex_lock(&m);
YYY:   while (done < 2) { pthread_cond_wait(&c, &m); }
YYY:   pthread_mutex_unlock(&m);
23:   printf("sum = %d\n", sum);
22: }
```

注: この問題では実際には `while (done < 2)` のループが 2 度以上まわることはない．条件変数の使い方の一般的なフォームとしてあえて while 文を使っているに過ぎない．

別解 1 Semaphore を用いる．Semaphore のカウンタの初期値を 0 とし，F, G は終了時に Semaphore の値をインクリメントする (post/release 操作)．C は 2 回，その Semaphore に wait を行う．コードは省略．

別解 2 この問題であれば，単純に C がスレッド F,G の終了を待つ，というのでもよい．Pthread であれば，`pthread_join`, Windows であれば，スレッドのハンドルそれぞれに対して，`WaitForSingleObject` を (都合 2 回) 呼べばよい．コードは省略．

別解 3 Semaphore も条件変数も使わずに，while 文の本体に，sleep(0), thr_yield() などの「CPU を他のスレッドに譲る」API 呼び出しを入れるだけ，というのでも良い．この場合，条件が成立するまでに C が何度も呼び出されることには間違いがないので，これまでの解答よりも浪費される CPU は大きいが，単純な上，実用的にはこれで十分という場合もある．もちろん sleep(0) が他のスレッドに本当に CPU を譲るかどうかは OS/ライブラリ依存であるのだが．

解説　(1) は良くできていた．並行プログラミングではこのような注意が必要だということを知ってさえいれば，パズルとしてはやさしいだろう．ただし，解答に示したとおり，ストーリ (注意すべき点) が二つある．ひとつは，done と sum の更新の順番がまずいために，sum の更新以前に C がループを抜ける，というストーリ．もうひとつは，sum への加算が競合を起こして片方が失われるというストーリである．後者しか気づいていない人は 0 というケースがあることを見逃してしまった．

(2) では，

- done と sum の更新の入れ替え
- 加算を不可分に行うための lock

の**両方**が必要であることに気づいていた人はごく少数であった．片方だけではだめだということを良く考えてみて欲しい．

なお，lock は今回，8-9 行目，15-16 行目をひとまとめにして lock しているが，加算が失われないことだけが重要なので，8 行目単独，9 行目単独，15 行目単独，16 行目単独，でももちろん構わない．そして，lock を使わずに「不可分な足し算」を行うこともでき (授業のスライド参照)，それでももちろん正解である．

`vf = f(x)` や`vg = g(x)` を含めてすべてを lock で囲んでいる答案もあった．これはもちろん問題に対する解答としては OK なのだが，若干減点させてもらった．これをしてしまうとそもそも`f(x)` と`g(x)` を並行に計算していることにならない．これはもちろん「問題文外の，常識に基づいたお約束」に過ぎないので,「ああ，問題にもう少しはっきり書いておくべきだった」というのが正直なところなのだが，しかしまあ，若干の減点くらいは許して欲しい．

2

(1) [**配点 3**]

> (. . . 演習で) **B というファイルを作り (すでにあれば上書きし)，ファイル A の内容をそれにコピー** (するプログラムを. . .)

会話を自然にするために，単に**ファイルをコピー**でも正解.

(2) [**配点 3**]

> (このコンピュータには) **どのくらいのメモリがつまれている** (の？)

(3) [**配点 4**]

とりあえず，**128MB** くらい，というのが最もありそうな答えであろう．根拠は，

> 100MB と 105MB で大きく実行時間が違っているのは，100MB ではほとんど発生していないページングが 105MB になったときに大量に発生している (そうなる理由は (4) の解答参照) ことから，このコンピュータには，100MB のファイルの中身がちょうど収まる程度のメモリが搭載されていると考えられる．ファイルのデータ以外にこのアプリケーションのほかのデータ，コード，OS 自身が使うメモリなどを考慮に入れ，この付近でメモリの大きさとして切りのいい，128MB が妥当な推論である．もちろん OS がメモリ食いであれば，192MB や 256MB ということもありうる．

(4) [**配点 10**]

> ファイル A の中身全体が物理メモリに収まらなかったわけ．だから，read システムコールがファイル A を読んでいる最中に，ページフォルトが発生しているわけ．そのときにはおそらくファイル A が，大体古く読まれた方からページアウトされているわけ．で，それが write システムコールの最中にまた読み込まれるわけ．結果として，非常に多くにページフォルトが発生して遅くなっている (というわけ).

(5) [**配点 10**]

ファイルの内容を一度に読み込むのではなく，小さなバッファを用意して，中身を順に読んでは B に書き込むことを繰り返す．変更された行の番号に' をつけ，追加された部分の行番号を XX とした．

```
 1: int main()
 2: {
 3:   int a = open("A", O_RDONLY);
 4:   int b = open("B", O_CREAT|O_WRONLY|O_TRUNC, 0777);
 5:   int sz = filesize("A");
6':   char * buf = (char *)malloc(BUFSZ);
XX    while (sz > BUFSZ) {
 7:     read(a, buf, BUFSZ);
 8:     write(b, buf, BUFSZ);
XX      sz -= BUFSZ;
      }
XX:   read(a, buf, sz);
XX:   write(b, buf, sz);
 9:   close(a);
10:   close(b);
11: }
```

解説　総じて良くできていた．今振り返ると，あまりわかっていなくても想像で解けてしまう，易しい問題だったかもしれない．

(3) は，いずれにせよ断定的なことはいえないので，解答例以外にも，その根拠が的を射ていれば，正解としている．OS やその他のプログラムが消費している領域によって答えは変わる．「100MB くらい」という答えも多かった．これは OS その他で消費する領域を考慮に入れていないという点で，センスとしては減点ものだが，なにしろ「(3)くらい」と，「くらい」がついているから，128MB も 100MB くらいだろといわれた日には減点できない．

以下に，採点をしていて笑えた，「お気に入りの解答・コメント集」をあげよう．

お気に入り解答集

- (4) 100MB のファイルを読み込んだときはそのままメモリに収まるから良かったんだけど，105MB のファイルになると収まりきらない部分が出てくるじゃない? そうなると，メモリに入っていない部分を読み込もうとしたときに，2 次記憶から読み込んで，メモリに入っているほかのデータと入れ替えなきいけなくなるわけ．これをページングって言うんだけど，これがけっこうバカにならない時間がかかんだよ．だから 100MB をこえるファイルを読み込むとページングが頻発に起こって時間が倍以上かかる（というわけ．原文どおり）

  田浦's コメント: 模範に近い解答だが，きちんと会話の「流れ」に乗っていて，この後の女のセリフと続けて呼んでいくと笑える．

- (4) もったいぶってなんかないよ〜．いっきに説明するから良く聞いてね（以下省略）

- (3) 256MB. OS を Windows XP と仮定すると，プログラムの消費するメモリと OS の消費するメモリの合計がちょうど 256MB 程度になると考えられるため．

- (4) 一度にたくさん read しすぎなんだよ (以上)

- (3) (前略) なお，256MB である理由は，私の自宅にあるパソコン (いまだに Windows Me だが) が，物理メモリのサイズが 256MB であるのに起動直後の状態で使用されていないメモリのサイズがせいぜい 110MB であるからである．
- (3) (前略) 他のアプリケーションがメモリを大量に消費していれば当然 256MB 以上のメモリがあっても遅くなることもあり得るだろうが，あまり現実的ではないので 128MB とした (こういう人は PC を買ったときのまま掃除もせず次々とソフトをインストールしてそうなのであり得るかもしれない).
- (3) (最優秀作品賞．全文掲載) 答え: 128MB. 理由: 100MB, 105MB 程度のファイルすべてメモリに展開に処理をし，105MB で遅くなるということは，OS，その他が 25MB-30MB とっているとするとこんなものか．また，食事を食べながらプログラムを実行しているとするところをみると，一般的に考えて，ここは情報処理演習室ではなく，よって女の持つ PC はノートである．大学入学，それより少し後くらいに両親に買ってもらったと仮定すると，128MB は妥当である．

  田浦's コメント: その PC は，最近その男が買ったという可能性はないか?
- (3) (前略) 問題文に現れる女は計算機やそのアーキテクチャに関する深い知識はなさそうだから，自分のパソコンのメモリを増設したとは考えにくい.
- (3) (前略) でも有効数字 3 桁で答えておいて (3) くらい というのはおかしいか?
- (コメントとして) この問題はバレンタインデーの日に OS がクラッシュして，ひきこもって復旧していたら一日が終わったという非モテ系電気学生へのイヤガラセですか? そうですか?

  田浦's コメント: いえ，そうではなく，非モテ系電気学生が，行く場所を選ぶことにより，コンピュータに強く，優しく，ルックスも◎ (自称) であったために，容易に新しい彼女を GET することができた... という励ましのストーリ展開であることに注意されたい．つい 10 年前，インターネットが普及する前はコンピュータなどできたところで,「モテ度」には何の足しにもならないどころか，オタクとみなされてマイナスだった時代もあったのである．

3

[配点 30]
解答例:

> CPU にはメモリ管理ユニットがあり，そこではメモリアクセス命令で指定されたアドレス (論理アドレス) を物理アドレスに変換して，メモリをアクセスする．その変換の内容は，メモリ上のページテーブルや CPU 内の TLB に設定されている．OS は，プロセスが利用する論理アドレスに対する物理アドレスが，(通常は) 重ならないようにこの変換を設定して，各プロセスごとに分離された「論理アドレス空間」を実現する．そのため，プロセスがどのような論理アドレスをアクセスしても，それが他のプロセスのメモリをアクセスすることにはならない．また，論理アドレスではなく，物理アドレスを直接指定してメモリにアクセスする命令は，特権命令であり，カーネル外のアプリケーションプログラムが実行することはできない．また，ページテーブルや TLB の書き換えも，特権命令によって行われ，したがってユーザプログラムが設定を行うことはできない.

解説

CPU に備わる，関連した仕組み: については,「アドレス変換機構」すなわち，メモリアクセスの際にプログラムで指定されたアドレスが，物理アドレスに変換されてアクセスされること，についての言及が必要である．また，CPU

にユーザモード，特権モードという概念があり，この「変換」を，ユーザモードで指定することはできない，ということに対する言及が必要である．つまりたとえば，ページテーブルを更新したり，TLB を更新したりすることはユーザモードできない，ということである．

また，ユーザモードでは，物理アドレスを使ってメモリをアクセスすることはできない，ことへの言及も採点基準に取り入れている．

オペレーティングシステムが提供する概念: は,「論理アドレス空間」のことである．言葉はともかくとして,「プロセスがどのような論理アドレスにアクセスしようとも，アドレス変換機構が介在することによって，他のプロセスのアドレス空間をアクセスすることにはならない」ことに言及する必要がある．

その概念をオペレーティングシステムがどう実現しているか: については，OS がアドレス変換を「各プロセスの論理アドレス空間が写像される物理アドレスが重ならないように」設定していることに言及する必要がある．

この問題を採点していて気になったのは，いわゆる，関係の「ありそうな」ことを片っ端から書いていて，それらの記述にうそはないものの，この問題に対する解答としてはほとんどが的外れである，というような答案が多かったことである．こういうものを減点法 (的外れなことを書いたら減点していく) で採点するとどんどん点数が減りそうであるが，そうはしていない.「そもそも受け答えとして意味が通ってない」答案なのだから，大幅減点を食らっても文句は言えないのだが...

技術的な点で気になったのは,「メモリアクセス命令が特権命令」という記述が意外に多かったことである．そういう人には，普段自分が書いているプログラムはどうメモリをアクセスしているのか，と問うてみたい．おそらく，そう書いた人の多くは「物理メモリに直接アクセスする＝物理アドレスを使ってメモリをアクセスする」のが特権命令だと言いたかったのだろうと善意に解釈している．これは正しい．