平成25年度オペレーティングシステム期末試験
Operating Systems: the Final Exam

2015年2月10日 (火)
February 10th 2015

- **問題は 3 問**

  There are three problems.

- **この冊子は，表紙 1 ページ (このページ)，日本語の問題 2-7 ページ，その英訳が 8-13 ページから成る．**

  This booklet consists of a cover page (this page), problems in Japanese (2-7 pages), and their English translations (8-13 pages).

- 解答用紙は 1 枚. **おもてとうらの両面あるので注意すること．**

  There is an answer sheet. **It is printed both sides.**

- 各問題の解答は所定の解答欄に書くこと.

  Write your answers to each problem in the designated box in the answer sheet.

1

次の会話を読んで後の, (1) から (7) の問いに答えよ (イラスト: Piro. シス管系女子より. 問題の内容とは関係ありません).

みんとちゃん (以下 M): う〜ん, 不思議だなぁ.

大野桜子先輩 (以下 O): みんとちゃん, どうしたの?

M: あ, 大野先輩! お疲れ様です. 今日大学の授業で，自分の PC で立ち上がっているプロセスやスレッドを調べる方法を習ったんです. [(a)] というコマンドラインでできるって. そしたらなんとプロセスが 195 個も立ち上がっていたんです!

O: ふ〜ん，そうかもね. でも，何が不思議なの?

M: はい, こんなに周りにプロセスが立ち上がっていたら，普段私が授業や演習で作っているプログラムって，本来のコンピュータの速度の，1/196 の速度でしか動いていないんじゃないかと思って. で，余分なプロセスがほとんど走っていないサーバにログインして同じプログラムを走らせたんですけど，ほとんど変わらないんです.

O: そりゃそうだよ，195 個プロセス—てことは少なくとも 195 個のスレッド—がいると言っても，そのほとんどは，中断中という状態になっていて，CPU は割り当てられないのよ.

M: あ，そういえば授業でもそんなことを言っていたような. でも OS はどうやって，中断中のスレッドとそうでないスレッドを区別するんでしょうか?

O: [(b)]

M: なるほど〜. で，実行可能なスレッドが複数いる場合は，当然それらの間で CPU を分け合うんですよね.

O: そうね.

M: 実行可能なスレッドの中には，私が演習で作ったみたいに，ひたすら計算だけをやってシステムコールなんて一切呼ばないものも有りますね．システムコールを呼べば OS がそのタイミングで他のスレッドに切り替えるとかできそうですけど，そうでないスレッドの場合，どうやってそのスレッドから CPU を奪って，他のスレッドに切り替えるのでしょうか?

O: (c)

M: なるほどですね〜．OS は，実行可能なスレッドを代わりばんこに実行しているんでしょうか?

O: ん〜，ま，大雑把にはそう思っていてもいいけどもう少し賢いことをしているんだな．まず,「代わりばんこ」っていうのは，もう少しちゃんとアルゴリズムとして定式化するとどうなるかしら?

M: そうですね，まず,「実行可能なスレッド」のキュー (run_queue) というのがあって，

(i) 実行中スレッドが他のスレッドに CPU を奪われる時は，もともと実行中だったスレッドを run_queue の末尾に入れ，run_queue の先頭にあるスレッドを次に実行する．

(ii) 実行中のスレッドが中断するときは，run_queue から外され，run_queue の先頭にあるスレッドを次に実行する．

(iii) 中断中のスレッドが復帰する (実行可能になる) ときは run_queue の末尾に入る

そんなところでしょうか?

O: そうね, たしかにこれだと，常時実行可能なスレッドが複数あったら，きれいに代わりばんこに実行されて,「公平性」っていう意味ではいいよね．でも，これだけだと，特にデスクトップ OS なんかで重要な，ある目標が達成されないんだな．その目標は,「(d)」というものなの．

M: たしかにそうですね．あ，じゃあ，こうしたらどうでしょうか?

(i) さっきと同じ

(ii) さっきと同じ

(iii) 中断中の状態から復帰する (実行可能になる) ときは，(e)

O: それはいいかもしれないけど，でもそうすると今度は基本的な目標である「公平性」を損なうことになるんじゃない? 例えば，(f) こんなプログラム を書いたら，そのスレッドは，不当に多くの CPU 時間を得ることができてしまうよね．

実際の OS ではこの (g)「公平性」と(d)を両立 すべく，スケジューリングアルゴリズムが作られているのよ．

M: ふえ〜，たかが CPU を割り当てるだけなのに，結構頑張ってるんですねぇ．

(1) [(a)] には全てのプロセスを列挙する Linux のコマンドが入る．それを (必要ならばオプションも含めて) 書け．

(2) [(b)] にはどのような事象に対して OS がスレッドを，中断中にするかの説明が入る．そのような事象を 3 つ以上あげながら，適切な文章を書け．

(3) [(c)] には，長時間システムコールを発行せずに計算だけをするスレッドから，OS がどのように CPU を奪うかの説明が入る．適切な文章を書け．

(4) [(d)] には，OS がスケジューリングの目標のひとつとしている項目が入る．適切な言葉を書け．

(5) [(e)] には，みんとちゃんが当初示したアルゴリズムに対する単純な変更が入る．適切な文を書け．

(6) 下線部 (f) はどんなプログラムか．概要を書け．

(7) 下線部 (g) の，両方の目標を達成するようなスケジューリングアルゴリズムを一つ考え，その概要を示せ．

2

N バイトのファイルを配列に読み出し，その配列中のランダムな W バイトを読み出す様々なプログラムを考える．以下の (1)〜(5) の各プログラムについて，**物理メモリ**の使用量はどのくらいになるか，答えよ．いずれの場合も，OS カーネルおよび関わるプロセス全体での合計使用量を答えよ．ファイルと配列を読み出すのに必要な以外の物理メモリは考えなくて良い．

ページサイズを P とする．解答は，以下の項，その定数倍 ($2N$, $2W$ など)，およびそれらの和 ($N+2W$, $N+CN$ など) で表わせ．また，$CWP \ll N$ を仮定して良い．

$$N,\ W,\ CN,\ CW,\ NP,\ WP,\ CNP,\ CWP.$$

(1) malloc で N バイトを確保し，read を用いて読みだす．プログラム片は以下．

```
char * a = malloc(N);
int fd = open(filename, O_RDONLY);
read(fd, a, N);
random_access(a, N, W);
```

ただし，random_access(a, N, W) は，アドレス $[a, a+N)$ から乱数で W 個のアドレスを選び，そこをアクセスする，以下のような関数とする．

```
void random_access(char * a, long N, long W) {
  char s = 0;
  long i;
  for (i = 0; i < W; i++) {
    long k = 0以上N 未満の乱数;
    s += a[k];
  }
  printf("%d\n", s);
}
```

(2) mmap でファイルを N バイト分，マップする．プログラム片は以下．

```
int fd = open(filename, O_RDONLY);
char * a = mmap(0, N, PROT_READ, MAP_PRIVATE, fd, 0);
random_access(a, N, W);
```

以下では，C 個の子プロセスを起動し，子プロセスがファイル・配列をアクセスする．ただし，random_access 中の乱数はプロセスごとに異なる列を返す．

(3) 親プロセスが malloc で N バイトを確保し，各子プロセスが read を用いて読みだす．

```
char * a = malloc(N);
long i;
for (i = 0; i < C; i++) {
  if (fork() == 0) {
    int fd = open(filename, O_RDONLY);
```

```
    read(fd, a, N);
    random_access(a, N, W);
    exit(0);
  }
}
```

(4) 親プロセスが malloc, read を行い，子プロセスが配列をアクセスする．

```
char * a = malloc(N);
int fd = open(filename, O_RDONLY);
read(fd, a, N);
for (i = 0; i < C; i++) {
  if (fork() == 0) {
    random_access(a, N, W);
    exit(0);
  }
}
```

(5) 各子プロセスが mmap を行う．

```
for (i = 0; i < C; i++) {
  if (fork() == 0) {
    int fd = open(filename, O_RDONLY);
    char * a = mmap(0, N, PROT_READ, MAP_PRIVATE, fd, 0);
    random_access(a, N, W);
    exit(0);
  }
}
```

3

次のような動作をする二つの関数を作りたい.

- `track_modifications(void * a, long n);`
- `is_modified(void * p);`

`track_modifications`(a, n) を呼び出すと，$[a, a+n)$ の範囲にあるのアドレス p へ，その呼び出し以降書き込みが起きたかどうかを，`is_modified`(p) を呼び出すことで知ることができる．より正確には，

- a および n はページサイズの倍数とする.
- p は $a \leq p < a+n$ を満たす.
- 簡単のため `track_modifications` は一度だけ呼ばれるとする.
- これらの条件を満たす時，`is_modified`(p) は，`track_modifications`(a, n) の呼び出し以降，p を含むページへの書き込みが起きていれば 1, 起きていなければ 0 を返す.

(1) これらの関数を，OS カーネルで (つまり，必要とあらば OS を変更して) 実現する方法の概要を記せ.

(2) これらの関数を，Unix に既に備わっているシステムコールを用いて，ユーザレベルで実現する方法の概要を記せ.

(3) これらの関数の応用を一つ記せ.

解答例

1

(1) ps ax (他にオプションとしては，-ef, aux, auxw, auxww など．とりあえず全てのプロセスが表示されれば何でも OK).

配点 6 点. ps だけでも 4 点. top など，全部のプロセスを表示するのが困難なツールは 2 点.

(2) raed や recv で入力がなくて入力待ちになった場合，pthread_mutex_lock や wait なんかで他のスレッドと同期を撮ろうと思った時にその同期が成立しなかった場合，sleep なんかで自主的にブロックする場合，メジャーページフォルトを起こして 2 次記憶からページを読み込まないといけなくなった時なんかに，OS はそのスレッドを中断させるのよ.

配点 12 点 (まっとうな要因をひとつ上げるごとに 4 点).

(3) 定期的に CPU にタイマ割り込みが発生するようになっていて，OS がその他イミングで制御を得ることができるのよ.

配点 6 点. 概ね,「タイマ」「割り込み」といったキーワードが入っているか否かで判定している.

(4) 対話的プログラムの応答性

配点 4 点.

(5) `run_queue` の先頭に入れる

配点 4 点. なぜか「2 番めに入れる」などの解答もあったが同様にマルにしている.

(6) (例) 次のタイマ割り込みが入る直前 (∗) まで走り続け，そこでわずかな時間 sleep するプログラム.

注: (∗) で，次のタイマ割り込みが起きるタイミングをどうやって知るのかという問題が残るが，もちろんここではそれを述べていなくても OK としている．実際には，時計を見ながらしばらく走り続ければ，他のスレッドに CPU を奪われ，再び制御が戻ってきた時を知ることができる (時刻が急に飛ぶ).「その時刻 + タイマ割り込み間隔」が，次のタイマ割り込みが起きるおおよその時刻である.

配点 6 点. なお，不当に多くの CPU を得るには,「ある程度」走って，中断して「すぐ」起きる，という組み合わせが必要で，単に,

```
while (1) {
  sleep(一瞬);
}
```

のようなものは，不当に CPU を多く得ることはない．上記のようなプログラムや，単に「中断と復帰を繰り返す」のような表現は正解とは出来ない.

sleep で CPU を手放せば，少なくとも他のスレッドがひとつ，タイマ割り込み分くらいは走ることになる．一方でこのスレッドはほとんど何もせずに sleep だけを繰り返したのでは,「不当に」多く CPU を得るには至らない.

(7) (例): 各スレッドに，CPU が割り当てらた累積時間 (vruntime) を管理する．スレッドを切り替える際は，vruntime が最も小さいものを選ぶ (∗)．ただし，中断したスレッドが復帰するときは，その

スレッドの vruntime $\geq$ 実行可能なスレッドの vruntime の最小値 $-$ 一定値 (例えば 20ms) となることを保証する (†).

なぜこれで目的が達成されるかの説明は，要求していないが，ポイントは以下の通り．

- (∗) により，CPU の割り当て時間は公平になる
- 対話的なスレッドは多くの場合 CPU を消費しない — よって vruntime が小さい — ため，(∗) により，応答性も良くなる．
- (†) のようにすることで，長時間中断したスレッドが再開しても，CPU を連続して専有できる時間には上限が設けられ，他のスレッドの応答性が損なわれることもない．

配点 12 点．もちろん上記は一例．他のやり方もある．文章を読み取った結果，上記と同じような意味での「公平性」が達成されていると判断したら 6 点.「応答性」が達成されていると判断したら 6 点.

2

(1) $2N$. OS が管理するキャッシュで N, プロセスのアドレス空間 (read が用意した buffer) に N かかるので．

配点 4 点．ただし，N という解答も 2 点与えている．

(2) WP. W 個のページをさわるため．

実際には，W 回のアクセスの中に，同じページが複数回出現するかもしれないが，条件 $CWP \ll N$ (特に，$WP \ll N$) より，その影響は無視できると考えて良い．

配点 4 点．

(3) $(C+1)N$

OS が管理するキャッシュで N, 各プロセスのアドレス空間 (read に渡された a) に**それぞれ** N かかるので．

配点 4 点．ただし，CN という解答も 2 点与えている．

(4) $2N$

OS が管理するキャッシュで N, 全プロセスのアドレス空間で共有された a に N かかるので．

前問との違いに注意．今度は親プロセスが read を呼んだ時点で a のために N バイトの物理メモリが割り当てられる．その後の fork() の際，a のための物理メモリはコピーされず，書きこみがおきるまでは共有されている．その後も書き込みは起こっていないから，a のために必要な物理メモリは全体で N バイト．

配点 4 点．ただし，N という解答も 2 点与えている．

(5) CWP. 各プロセスが W 個のページをさわるため．

実際には，CW 回のアクセスの中に，同じページが複数回出現するかもしれないが，条件 $CWP \ll N$ より，その影響は無視できると考えて良い．

配点 4 点．

3

(1) (解答例) `track_modifications(`a`, `n`)` では，$[a, a+n)$ を構成するページ群の dirty bit をクリアする．`is_modified(`p`)` は，p を含むページの dirty bit の値 (0 または 1) を返す (解答終わり)．

注: 以上程度が書けていれば正解とするが，実際には dirty bit はその他の目的 (OS のページ置換アルゴリズムが最近使われたページを把握するため) でも使われるため，それと干渉しないような注意も必要である．それを考慮した解答例は以下．

(長い解答例) OS のページ置換アルゴリズムが定期的にページの referece/dirty bit の値を保存し，クリアしていることを前提にする．

`track_modifications(`a`, `n`)` では，$[a, a+n)$ を構成するページ群に対応する bool 型の配列 `bool` $M[n/P]$ を用意する．`bool` $M[i]$ は，`track_modifications(`a`, `n`)` で 0 に初期化され，これ以降更新があったかどうかを記録する．また，当該ページ群のページテーブル上の dirty bit をクリアする．`is_modified(`p`)` は，$i = (p-a)/P$ として，

$$p \text{ を含むページの dirty bit} | M[i]$$

の値 (0 または 1) を返す．

ページ置換アルゴリズムが，$[a, a+n)$ に属するページの dirty bit をクリアする際は，各ページに対し

$$M[i] := M[i] | \text{ ページ } i \text{ の dirty bit}$$

として，$M[i]$ に望む情報 (`track_modifications(`a`, `n`)` 呼び出し時点からの更新の有無) が維持されるようにする．

また，ページ置換アルゴリズムが最近の更新の有無を正しく判断できるように，`track_modifications(`a`, `n`)` が dirty bit をクリアする前に，OS が最近の更新を記録するデータ構造もその時点での dirty bit を用いて更新する．例えば 1 秒おきに更新の有無を記録するアルゴリズムであれば，`track_modifications(`a`, `n`)` 呼び出し時の dirty bit をそれに反映させるなど (解答終わり)．

カーネル内部ではページテーブルに保持されて dirty bit, reference bit が参照できるから，それを使うのが自然である．ただし，ページを read only にしておいて例外 (保護違反) を検出する (その例外を補足することで書き込みがあったページを検出する)，という方法も可能である．そのような解答も，きちんとかけていればマルとしている．なお，ここで発生する例外は CPU が OS に通知する「保護違反」であり，ページフォルトとは似て非なるもの．ページフォルトは，対応する物理ページがない場合に発生する．「ページフォルト」という言葉を用いた解答が多かった．が，それも大目に見ている．

(別解) `track_modifications(`a`, `n`)` では，$[a, a+n)$ を構成するページ群を，MMU の機能を用いて書き込み不可にする．書き込みがあると保護違反例外が発生するため，それを補足することで，書き込みがあったページを検出できる (解答終わり)．

配点 10 点. 実際には日本語として通じない言葉が多数並んでいたが，ともかくそれらしい言葉 (dirty bit, 書き込み不可，フォールト，etc.) に反応して点数を与えている．

(2) `track_modifications(`a`, `n`)` では，$[a, a+n)$ を構成するページ群に対応する bool 型の配列 `bool` $M[n/P]$ を用意する．`bool` $M[i]$ は，`track_modifications(`a`, `n`)` で 0 に初期化され，これ以降更新があったかどうかを記録する．

また，当該ページ群を，ユーザレベル仮想記憶プリミティブ (mprotect) を用いて，書き込み禁止にするとともに，保護違反発生 (書き込み発生) 時に呼ばれるシグナルハンドラをセットする．そのシグナルハンドラでは，書き込みが発生したページに対応する $M[i]$ をセットすると共に，当該ページの書き込みを mprotect を用いて可能にする．

is_modified(p) は，$i = (p - a)/P$ として，

$$p \text{ を含むページの dirty bit} | M[i]$$

の値 (0 または 1) を返す (解答終わり)．

配点 10 点．

出題時に想定していなかった解答として，track_modifications(a，n) 時に，$[a, a + n)$ をどこかへコピーしておき，is_modified(p) は，コピーと現在の値を比べる，というものがあった．この方法は，書き込みはあったが値は換わっていない (ないしは元に戻っている) という場合は検知できないから，もちろん技術的には正解とは言いがたい．また，is_modified(p) を実現するだけならなんとかなっても,「書き込みが在ったページを全部挙げる」といった目的にはひどく効率が悪い (100 万ページ中，数ページだけ更新されたというような場合でも，全部のページが更新されたかを調べなくてはならない)．このように「正解」とは出来ない理由はあるのだが，むげにバツとも出来ない．要は，こんな解答を想定していないが，たしかに出された問題に対する解答としては部分的にはあっている．ということでこの方針の解答には 2 点与えている．

(3) (解答例) 差分チェックポインティング．ネットワークページング，分散共有メモリ，インクリメンタル GC の際の書き込みバリア．

注: チェックポインティングは，アプリケーションデータを定期的にディスクに保存して，障害発生時に再開できるようにする手法．差分チェックポインティングは，その保存をする際に，最後のチェックポイント取得時から更新があったものだけを保存する手法．

ネットワークページングは，ページアウトのための領域を，ディスクではなくネットワーク越しのマシンとする方法．

その他はスライド参照 (授業では説明していない)．

配点 10 点．

なお，特に説明はしていなくても，正しい応用がかけていれば，マルとしている．

なお，これまた想定外の解答として,「LRU ページ置換の実装」というものがあった．ちなみにこれには 2 点与えた．

この解答も，間違いとは言い切れないが，ちょっと的を外していると言わざるを得ない解答．ここで聞いているのは，is_modified というような API で書き込みが判定できるとしたら，それによって新たに可能になる機能はなにか，ということである．

ページ置換は OS 内部にすでに組み込まれているもので，わざわざこんな API をユーザプログラムに提供して，それをあらためて使う，というものではない．

仮にそこを譲って，想像上の話として，まずこの API があって，それを (OS 内部で) 用いれば，LRU 置換の役に立つ，という議論があったとしても，LRU 置換の実現には，書き込まれかどうかだけではなく，参照されたかどうかの情報も必要であるから，LRU 置換をこの API の応用と位置づけるのはこの意味でも無理がある．

全体講評

受験者 107 人．最高点 83 点, 最低点 4 点．上位 20 人の得点

$$83, 80, 78, 76, 76, 74, 72, 72, 72, 70, 68, 64, 62, 62, 60, 58, 58, 56, 56, 54$$

なお，成績は試験 70%, 11/11-11/18 にかけて行われた演習の提出状況 30%でつけている．

後者は，試験が振るわなかったがきちんと出席して演習をやっていた人に対する救済という意味合いが大きい．とにかく，授業の出席人数と試験の数があまりにもかけ離れていたため，演習をマジメにやった人とそうでない人を一律に採点するのは忍びないということでこのようにした．