平成22年度オペレーティングシステム期末試験: 解答と講評

2011年2月8日

注意事項

- **問題は3問, 8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**

1

次の文章を読んで, 後の (1)〜(8) の問いに答えよ.

美希: うわー, OS の試験が明日だ〜〜!

池上彰: ちゃんと勉強してますか?

美希: いいところへ来ました. 教えてください. OS の授業で先生が mmap について力説してたんだけど, 正直なんのことかわかりませんでした.

池上彰: どんな風に力説していたんですか?

美希: なんでも, OS の真髄である仮想記憶の仕組みを学んだ人には, mmap はとても自然に思えるとかなんとか... でも所詮はファイルを読むためにあるんだから, 「読めりゃええやん, ファイルなんやし」って思うんです.

池上彰: いい質問ですね! それではまず, mmap というのがどんなシステムコールなのか, おさらいしてみましょう.

美希: はい. じゃあマニュアル (5 ページ参照) を見て, と. 引数が色々あってちょとややこしいですけど...

池上彰: そうですね. じゃあ例題として, `foo` というファイルに, 文字 `A` が何文字含まれているかを表示するプログラムを, mmap を使って書いてみてください. 簡単のため, ファイルのサイズはわかっていて, 10000 バイトとしておきましょう.

美希: はい.

美希: ...(しばらくして) ととのいました!

```
int main() {
  int fd; char * a;
  ┌──────────┐
  │   (1)    │
  └──────────┘
  int i; int c = 0;
  for (i = 0; i < 10000; i++) {
    if (a[i] == 'A') c++;
  }
  printf("foo has %d 'A's\n", c);
}
```

池上彰: どれどれ, そうですね. よく出来ました.

美希: でも, 何が嬉しいのかわかりません. たしかに配列 a を読み書きするだけでファイルを読み書きで来ていますが, 同じことは, mmap なんか使わなくても, (2) 普通にメモリを malloc で割り当てて, read で読めばできるのではないでしょうか?

池上彰: いい質問ですね! それに答えるためにまず, mmap の仕組みについておさらいしてみましょう. まず, mmap システムコールを呼んだとき, OS はその中でどんなことをしていると思いますか?

美希: んー, その時にファイルを読んできているのではないでしょうか?

池上彰: 違いますねぇ.

美希: ええっと, あ, そういえば授業でそんなことを力説していたような... あそうだ, (3) . どうでしょうか?

池上彰: 正解! じゃあ, それでなぜ, 配列 a にアクセスするだけで, 結果的にファイルが読めるのでしょうか?

美希: 確かに... あ, だんだん思い出してきました. (4) . どうでしょうか?

池上彰: 正解!

美希: なるほど... でも先生, これで OS が, 何かうまいことやっているというのはわかりましたけど, この例では結局ファイルを全部読むわけですから, やっぱり malloc と read を使っても同じじゃないという気がするのですが...

池上彰: いい質問ですね! 確かに美希君が言うように, 遅かれ早かれ全部のファイルを読むようなアプリケーションでは, 大差がないことが多いです. mmap が一番有効なのは, ファイルの (5) ようなプログラムです.

美希: 確かに, ファイルの (5) ようなプログラムでは, read ですべてを読み出すのは無駄ですし, かといって lseek を使って必要なところだけを読むのでは, プログラムがややこしくなりそうですね.

池上彰: その通り. 実際には中身がディスクから読まれているわけではないのに, あたかもファイルがすべて読み込まれているような状態を作り出している, というところが味噌ですね.

美希: ファイルの (5) ようなプログラムって, 実際の応用プログラムとしてはどんなものがあるんですか?

池上彰: いい質問ですね! どんなのがあると思いますか?

美希: そうですねぇ... (6) なんてどうでしょうか?

池上彰: そうですね. ところで先程ファイルの (5) ようなプログラムに対して mmap が有効だと言いましたが, それ以外にも, 有効な場面があります. それは, 多数のプロセスが同じファイルを読もうとしたときに顕著に表れるのですが, わかりますか?

美希: えーとえーと,

池上彰: じゃあヒントを出しましょう. 先程の, mmap の仕組みを考えてみてください. そして同じファイルをたくさんのプロセスが read を使って読んだとき, mmap を使って読んだとき, を比べてみてください.

美希: あ, そうか. (7) .

池上彰: 素晴らしい!

美希: でも, 多数のプロセスが同じファイルを読むなんてことが, そんなに頻繁にあるんですか?

池上彰: いい質問ですね! もちろん Web サーバみたいに, たくさんのプロセスが, リクエストで指定されたファイルを読む, なんていう場合, そういう状況が発生し得ますね. それ以外に, 普通のデスクトップやノート PC でも頻繁に読まれているファイルとして, (8) がありますね. このファイルは, 自分では読んでいるつもりはないかもしれませんけどね.

美希: なるほど. コンピュータが効率的に動くのは, そうやって OS がメモリを効率的に管理しているからなんですね.

池上彰: そうですね. そしてそれを支えているのが仮想記憶ですね. 元々は, プロセス間のメモリの分離を目的として発明されたものですが, 結果的には驚くほど豊かな使い道があったわけですね.

以下の問いに答えよ.

(1) [(1)]には, fd と a に適切な値を代入し, a を通じてファイル foo の中身が読み出せるようにする一〜数行のコードが入る. それを書け.

(2) 下線部 (2) で主張している方法は具体的にはどのようなものか? [(1)] に当てはまるコードとして書け.

(3) [(3)] には, mmap システムコールが呼び出されたときに OS が行うことを説明する文章が入る. 適切な文章を書け.

(4) [(4)]には, 上記のプログラムの for 文によって, 結果的にファイル foo の中身が読める理由を説明する文章が入る. 適切な文章を書け.

(5) [(5)]に当てはまる, mmap が有効となるアクセスパターンを簡潔に述べよ.

(6) [(6)] に当てはまる具体的なアプリケーションの例を, 自分なりに考えて述べよ.

(7) [(7)]には, 多数のプロセスが同じファイルを読む際, mmap を用いた方法が, read を用いた方法に比べてどのような点が優れているのか, およびその理由が書かれている. 適切な文章を書け.

(8) [(8)]に当てはまる適切な言葉を書け.

参考: mmap システムコールマニュアル (抜粋)

```
MMAP(2)                    Linux Programmer's Manual                    MMAP(2)

NAME
       mmap, munmap - map or unmap files or devices into memory

SYNOPSIS
       #include <sys/mman.h>
       void *mmap(void *addr, size_t length, int prot, int flags,
                  int fd, off_t offset);

DESCRIPTION
        ... <中略> ...

       If addr is NULL, then the kernel chooses the address at which to create
       the mapping; this is the most portable method of creating a new map-
       ping.  If addr is not NULL, then the kernel takes it as a hint about
       where to place the mapping; on Linux, the mapping will be created at a
       nearby page boundary.  The address of the new mapping is returned as
       the result of the call.

        ... <中略> ...

       The prot argument describes the desired memory protection of the map-
       ping (and must not conflict with the open mode of the file). It is
       either PROT_NONE or the bitwise OR of one or more of the following
       flags:

       PROT_EXEC  Pages may be executed.

       PROT_READ  Pages may be read.

       PROT_WRITE Pages may be written.

       PROT_NONE  Pages may not be accessed.

       The flags argument determines whether updates to the mapping are visi-
       ble to other processes mapping the same region, and whether updates are
       carried through to the underlying file.  This behavior is determined by
       including exactly one of the following values in flags:

       MAP_SHARED Share this mapping.  Updates to the mapping are visible to
                  other processes that map this file, and are carried through
                  to the underlying file.  The file may not actually be
                  updated until msync(2) or munmap() is called.

       MAP_PRIVATE
                  Create a private copy-on-write mapping.  Updates to the map-
                  ping are not visible to other processes mapping the same
                  file, and are not carried through to the underlying file.
                  It is unspecified whether changes made to the file after the
                  mmap() call are visible in the mapped region.

       Both of these flags are described in POSIX.1-2001.

       In addition, zero or more of the following values can be ORed in flags:

        ... <中略> ...
RETURN VALUE
       On success, mmap() returns a pointer to the mapped area.  On error, the
       value MAP_FAILED (that is, (void *) -1) is returned, and errno is set
       appropriately.
```

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, N 個のスレッドが barrier() という関数を呼び出すことで, バリア同期が実現されることを目指す. つまり barrier() 関数は, N 個すべてのスレッドがそれを呼出したとき, およびその時に限り, リターンする. 各スレッドには 0 から $N-1$ の一意な ID がふられており, thread_id() という関数でそれが得られるものとする.

以下の (1)〜(4) の問いに答えよ. (1)〜(3) までは, barrier() 関数は各スレッドが一度だけ呼ぶ (複数回呼ぶことはない) ものと仮定して良い.

(1) 「barrier() を呼出したスレッドの個数を数える大域変数 c 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.

```
volatile int c = 0; /* グローバル変数 */
void barrier() {
  c++;
  while (c < N) /* 何もしない */ ;
}
```

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() を呼んでいないスレッドがいるにもかかわらず, barrier() がリターンしてしまう

(b) 全員が barrier() を呼んだにもかかわらず, 決して barrier() からリターンしない

(c) 運よく (b) の問題がおきなかったとしても, スレッド数が多いと非常に性能が悪くなる

(2) 「それぞれのスレッドが barrier() を呼出したか否かを記録する配列を作る」という方針で以下のコードを書いた.

```
volatile int a[N] = { 0, 0, ..., 0 }; /* グローバル配列 */
void barrier() {
  int i;
  a[thread_id()] = 1;
  for (i = 0; i < N; i++) {
    while (a[i] == 0) /* 何もしない */ ;
  }
}
```

(a), (b), (c) それぞれの問題がおきるかどうか, (1) と同様に答えよ.

(3) どちらかのコードを元に, (a), (b), (c) どの問題もおきないよう, 適宜スレッドプログラミングの API を用いてプログラムを修正せよ. 前述したとおり, barrier() 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.

(4) それらのスレッドが複数回 barrier() を呼んでも良いように, barrier() 関数を拡張することを考える. そこで「同期が成立した後, 次の呼び出しに備えて a の要素を 0 にする」という方針で以下のようなコードを書いた.

```
volatile int a[N] = { 0, 0, ..., 0 }; /* グローバル配列 */
void barrier() {
  int i;
  a[thread_id()] = 1;
  for (i = 0; i < N; i++) {
    while (a[i] == 0);
  }
  a[thread_id()] = 0;
}
```

同様に (a), (b), (c) それぞれの問題がおきるかどうか答えよ.

3

OS が果たしている役割でもっとも重要なもののひとつは, 複数のプログラムを安全に一つのコンピュータ上で実行することである. その仕組みについて以下の (1), (2) の問いに答えよ.

(1) ファイル入出力, ネットワーク通信など多くの処理が, OS に対して「システムコール」を発行することでのみ実行可能になっている. そもそも通常のユーザプログラムがディスクから直接データを読んだり, ネットワーク機器を直接操作して通信をしたりすることができないようになっていて, システムコールを呼び出すとそれが可能になっている仕組みを述べよ. 以下が明確になるように記述せよ

- CPU や, 場合によっては周辺ハードウェアが提供している機能
- ユーザプログラムを実行するのと OS のプログラムを実行するのとで, 何が違うのか? なぜ OS がディスクやネットワークへのアクセスを実現しているコードを, ユーザプログラムが真似して実行しても, ユーザプログラムはディスクやネットワークへ直接アクセスできないのか?
- ユーザプログラムが「システムコール」を呼び出すと何が起きるのか? 普通の関数呼出しと何が違うのか?

(2) 一つのスレッドが CPU を独占しようとしても, それができないようになっている仕組みを述べよ. 以下が明確になるように記述せよ.

- CPU や, 場合によっては周辺ハードウェアが提供している機能
- OS はそれをどのように利用しているか? 例えば無限ループに陥ったプログラムがあっても, 他のプログラムが実行できるための仕組みはどのようなものか?
- より一般に, 多数のスレッドに公平に CPU が割り当てられるようにする仕組みとしては, 例えばどのような仕組みが用いられているか?

問題は以上である

1 : 解答

(1) 5 点

解答例:

```
fd = open("foo", O_RDONLY);
a = mmap(NULL, 10000, PROT_READ, MAP_PRIVATE, fd, 0);
```

コメント:

- MAP_PRIVATE は MAP_SHARED でも可.
- fd, a それぞれについて正しく書けていれば 2.5 点ずつ.
- O_RDONLY など細かい定数名についてはそれらしく書けていれば不問.

よくある間違いに open を fopen としたものがある. やや細かい話にはなるが, fopen を呼んだ結果かえってくるものは, FILE という構造体へのポインタで, mmap などへ渡せるもの (ファイルディスクリプタ) ではないので, 実践的にはその違いは重要.

(2) 7.5 点

解答例:

```
a = malloc(10000)
fd = open("foo", O_RDONLY);
read(fd, a, 10000);
```

コメント:

- a, fd, および read の呼び出しそれぞれについて正しく書けていれば 2.5 点ずつ

(3) 5 点

解答例: ... あ, そうだ, プロセスのアドレス空間から, 10000 バイト分の, 連続して開いている論理アドレスを探してきて, その範囲がファイル"foo" にマップされている事を, OS が管理するデータ構造, アドレス空間記述表とか言ったかな, に記録する.

コメント: 論理アドレス空間上で, 10000 バイト分の領域を, 「割り当てられた」と記録しておく「だけ」というところがポイントで, その時に実際に物理メモリを探してくるわけではない (demand ページング).

なお, それが可能なのは, 実際にその割り当てられたところにアクセスがあったときにページフォルトが発生して, それを OS が捕まえてその時になって (こっそり), 実際に物理メモリを割り当てることができるからである. ページフォルトが発生する理由は, ページテーブルに, 割り当てた範囲のページ対応する物理アドレスが「不在」となっているからである.

(4) 2.5 点

実際にその範囲のアドレスにプロセスがアクセスすると, ページフォルトが発生して, その時に OS が物理ページを割り当て, データをファイルから持ってくるんだったけな.

コメント: ページフォルトがおきてそれを OS が処理しているというストーリが明確に分かっているかどうかがポイントです.

(5) 2.5 点
一部を, ランダムに アクセスする

コメント: 「一部をアクセスする」という解答が多かったが, やはり後者の「ランダムに」(単純に逐次的ではなく) アクセスするという点も重要である. 例えば 1000000 バイトあるファイルの, 3000 バイト目から, 5000 バイト目を逐次的にアクセスするのであれば, 3000 バイト目に seek して 2000 バイト読めば良いだけの話である. あちらこちらから少しずつ読む (ランダム) という挙動が重要である.

(6) 2.5 点
辞書, データベースなど.

コメント: もちろん解答は色々あり得る. 適切な説明がなされていれば名詞は問わない. 重要なことは,

- 全データは大きく, 一回アプリケーションが起動するたびに全データをファイルから読むのはコストが大きい
- アプリケーション一回の操作ではわずかのデータしかアクセスしない

ようなアプリケーションを探すことである. 辞書であれば, 通常ハッシュ表や木構造などを用いて単語を高速に検索できるようになっているが, それらはデータ構造の僅かな部分を読むだけで, 目的の単語を検索する仕組みである. 従ってひとつの単語をルックアップするたびに辞書全体をファイルから読むのは非常に効率が悪い. そして目的の単語にたどり着くまでのアクセスはランダム性が高い. 木構造の子供をたどっていったり, ハッシュ値に基づいてアクセスする場所が決まるなど, 単純な逐次アクセスではない.

なお, 非常に多かった答えに「動画のプレイヤー」というものがあった. こう解答した人が何を意図してこう書いたのかは正直分からないが, 田浦が何か見落としをしているのでない限り動画プレイヤーが上記のような性質を持っているとは考えにくい.

確かに一部だけを再生する場合に, 「ファイル全体を読まない」という挙動にはなるが, その場合でも動画を再生している間のアクセスというのは, 逐次的である. もしかして, 普段動画プレイヤーにお世話になってばかりいるので, それしか思い浮かばなかったとか (笑... えない)????

(7) 2.5 点
同じファイルをたくさんのプロセスが読んだ場合, mmap を使った場合はそれらのプロセスが物理メモリを共有することができるのに対し, read では, データを読み込む領域はプロセス毎に別々で, それぞれに別の物理メモリが必要になる, だから mmap の方が必要な物理メモリがうんと小さくなる, でどうでしょうか?

コメント: なぜこのような違いが出てくるのか—単に OS が「たまたま」mmap の実装をそうしており, read の方の実装をサボっているだけではないのか— を少し突っ込んで考えてみよう.

たくさんのプロセスが mmap を使った, つまり以下のような呼出しを行った場合,

```
fd = open("foo", O_RDONLY);
a = mmap(NULL, 10000, PROT_READ, MAP_PRIVATE, fd, 0);
```

OS にとっては [a, a+10000) というアドレスの範囲は, どのプロセスでも同じ中身を持つということが明らかである (書き込みがおきるまでは). それは mmap という API の「仕様」である. だから OS にとっては, それらに同じ物理ページを割り当てる—ページテーブルを使ってそれぞれのプロセスにおける [a, a+10000) のアドレス範囲を同じ物理メモリにマップする— のはごく自然にできることである.

一方, malloc と read を使った場合,

```
a = malloc(10000)
fd = open("foo", O_RDONLY);
read(fd, a, 10000);
```

a の中身がそれらのプロセスの間で一緒になるのは, それらが「あとになって」同じファイルの同じ領域を読み込んでいるからたまたまそうなるのであって, OS がメモリ割り当て要求 (malloc) を見たときにはそんな事 (これからその領域に同じ内容が書かれる事) はわかるはずがない. それらに同じ物理ページを割り当てようと思ったら, OS としては, read が行われた際 (もっと一般にはメモリ領域が書きかわった際) に, たまたま同じ内容を持つページがシステムのどこかにないかを見つけてくる, ということになる. この探索をオーバーヘッドを少なくやるのは困難であろう. 実は, そのオーバーヘッドをどうにか少なくできたとしても, さらに困難 (というよりも現在のハードウェアでは不可能) な理由がある. それは, 論理アドレスから物理アドレスへの変換はページを単位としてしか行えないということである. そして, 各プロセスで, a というアドレスがページ内のどのオフセットに位置するかわからないので, それらをすべて同じ物理アドレスに変換するようにページテーブルをセットするのは不可能なのである.

例を使って具体的に説明する. たとえばあるプロセス P では a = 0x10100 (16 進表記), 別のあるプロセス Q では a = 0x20300 (16 進表記) だったとする. それぞれ, a = 0x10000 というページ内でオフセット 100, それぞれ, a = 0x20000 というページ内でオフセット 300 である. P の 0x10000 と Q の 0x20000 を同じ物理ページ—たとえば 0x40000—にマップしても, 結果として P の a は, 0x40100 に, Q の a は 0x40300 にマップされることになるので, 目的は達せられないのである.

(8) 2.5 点

共有ライブラリ

2：解答

(1) (a) は 2.5 点. (b)(c) は実行例を含めて各 5 点 (計 12.5 点).

- (a) おこり得ない.
- (b) おこり得る. 実行例.

  (1) スレッド P が c から値 (x とする) を読む;

  (2) スレッド Q が c から値 (同じ x) を読む

  (3) スレッド P が c に $x+1$ を書く;

  (4) スレッド Q が c に $x+1$ を書く;

  結果として c はいつまでたっても N にならない.
- (c) おこり得る. c が N になるまでの間, while (c $< N$); といういわゆる「頻忙待機」をしており, 最後のスレッドになかなか実行の機会が与えられない. 実行例:

  1 CPU しかないシステムで実行しているとし, スレッド $0, \ldots, N-2$ までがこのループを実行しているとすると,

  (1) スレッド 0 が自分のタイムスライスを使い果たすまで実行する;

  (2) スレッド 1 が自分のタイムスライスを使い果たすまで実行する;

  (3) ...

  (4) スレッド $N-2$ が自分のタイムスライスを使い果たすまで実行する;

  という具合に, 最後のスレッドが c++を実行するまでの間に, タイムスライス $\times(N-2)$ 分だけの時間が必要になる.

(2) (a)(b) は 2.5 点. (c) は実行例を含めて各 5 点 (計 10 点).

- (a) おこり得ない
- (b) おこり得る
- (c) おこり得る. おきる状況は (1) の場合と全く同じである.

(3) 2.5 点

```
pthread_mutex_lock(&m);
int d = c;
c = d + 1;
while (c < N) pthread_cond_wait(&co, &m);
if (d == N - 1) pthread_cond_broadcast(&co);
pthread_mutex_unlock(&m);
```

(4) 実行例を含めて各 6 点 (計 18 点).

- (a) おこり得る. 実行例は以下の通り.

(1) スレッド P が barrier() 内のループを抜け, a[P] = 0; を実行する.

(2) (他のスレッドが一切動く前に) スレッド P が再び次の barrier() を呼び, a[P] = 1; を実行する.

(3) (他のスレッドが一切動く前に) スレッド P が for 文に入り, すぐに抜ける

- (b) おこり得る. 実行例は以下の通り.

  (1) a[thread_id()] = 1; を実行した最後のスレッドを P とする. このスレッド P が a[P] = 1; を実行する.

  (2) スレッド Q が for 文を抜け, a[Q] = 0; を実行する.

  (3) スレッド P が for 文に入り, a のチェックを始める.

  すでに a[Q] = 0 が実行されているため, P は for 文 (正確には, `while (a[Q] == 0) ;`) を抜けることができない.

- (c) おこり得る. 実行例は (1)(2) と同じ.

コメント: ご覧の通り, 「競合状態」というのは非常にやっかいな問題である. 初めての場合, 細かいパズル的な思考を要求されるが, 実践的には, まともに同期を使わずに書かれたプログラムはきっと何か競合状態があるという「感覚」になれることかもしれない.

3：解答

(1) 15 点

解答例: CPU が提供する機能として, 特権モードとユーザモードという二つ (以上) の, 状態の区別がある. 一部の命令 (特権命令) は, ユーザモードでは実行できず, ネットワークなど外部との入出力を行う命令 (IO 命令) は, 特権命令である. OS は通常のプログラム (OS カーネルやデバイスドライバ以外のプログラム) を, ユーザモードで実行するため, 通常のプログラムが OS の真似をして IO 命令を発行しても実行できない.

CPU は, ユーザモードから特権モードへ移行するとともに, 割り込みベクタで指定されているアドレスへジャンプする命令 (「トラップ命令」) を提供している. システムコールはトラップ命令を用いて OS 内のシステムコール処理を開始するアドレスへ制御を移行する. それにより, 特権命令が発行可能になるとともに, ユーザが勝手なアドレスのコードを特権モードで実行することはできなくなっている.

コメント 以下が明確になっているか否かをチェックして, それぞれに 3 点程度ずつ与えている.

- CPU は 特権モードとユーザモード を提供している
- ネットワーク通信など, 外部とのやりとりを実現する命令 (IO 命令) は ユーザモードでは実行できない
- ユーザプログラムはユーザモード で実行されている
- CPU は トラップ命令(ユーザモードから特権モードへ移行すると共に特定の場所へジャンプする命令) を提供する
- システムコールはトラップ命令を発行する ことで呼び出される

(2) 12 点

解答例: CPU は「割り込み」を提供する. これは, 外部からの信号で, CPU に強制的に処理を特定場所に移行させる仕組みである. 周辺機器として, タイマコントローラがあり, CPU に定期的, ないし指定された時間後に割り込みを発生させる. CPU はタイマコントローラを用いて割り込みが定期的に発生するようにし, かつ割り込み時にスレッドの切り替えを行う必要があれば行う. これによりユーザプログラムが無限ループに陥っても, 定期的に OS に制御が戻り, かつその機会に他のユーザプログラムを実行することが可能となっている. OS は, CPU をスレッドに公平に割り当てるするために, 例えば以下のようなことをする.

タイムスライス (ひとつのスレッドが連続して走って良い時間) を設定し, OS が制御を得た際 (割り込み時など) に, 現在実行中のスレッドが走り出してからタイムスライス以上の時間が経過していないかどうか検査する. 経過していればそのスレッドから制御を奪い, 次のスレッドに CPU を割り当てる. そして, 実行可能なスレッドにかわりばんこ (ラウンドロビン) に CPU を割り当てる.

コメント: 以下が明確になっているかどうかをチェックし, 各項目に 3 点ずつ程度.

- CPU が 割り込み 機能を提供している
- 周辺機器 (タイマ) により, 定期的にタイマ割り込みが発生するようにする
- OS は, 割り込み時に, 場合によってはコンテクストスイッチする
- OS は, 各スレッドにタイムスライスを割りあてるなどして, 公平性を保証している

全体に対する補足

- 試験自身の素点 60 点以上が可, などの絶対的な線引きはしていないので, 自分の素点自体はあまり気にしなくて良い.
- 実際には点数の分布を見た上で線引きをして, 不可になりそうな答案に改めて目を通し, 「不可やむなし」と確認した上で, 不可は不可になっている.
- 上述した通りの細分化した配点をしていて, 集計は Excel でおこなっている.