平成17年度オペレーティングシステム期末試験

2006年2月13日

問題は3問，8ページある．

1

Linux, FreeBSD, Solarisなどの，Unix系のオペレーティングシステムではプログラムを起動する際にシステムコールforkとexecve (または類似) を組み合わせて行う．たとえば以下は “ls” コマンドを起動するプログラムの断片である．参考までにforkとexecveシステムコールの動作について説明したマニュアルページの一部を付録としてつけておく．

```
int pid = fork();
if (pid == 0) {
  char * argv[] = { "/bin/ls", 0 };
  char * envp[] = { 0 };
  execve("/bin/ls", argv, envp);
} else {
  ...
}
```

Unixオペレーティングシステムは，(1) プログラムの起動を高速に行う, (2) 限られたメモリで多数のプロセスを起動できるようにする，ためにどのようなことをしているか? もちろん,「様々な工夫や最適化をしている」などという抽象的な説明を期待しているのではない．forkやexecveの意味を実現するための方法として，考えられる安直な方法と比較しながら説明せよ．

fork マニュアルページ (抜粋)

```
NAME
       fork - create a child process

SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>

       pid_t fork(void);

DESCRIPTION
       fork creates a child process that differs from the parent process only in its PID and PPID,
       and in the fact that resource utilizations are set to 0.  File locks and pend- ing signals
       are not inherited.

RETURN VALUE
       On success, the PID of the child process is returned in the parent's thread of execution,
       and a 0 is returned in the child's thread of execution.  On failure, a -1 will be returned
       in the parent's context, no child process will be created, and errno will be set
       appropriately.
```

execve マニュアルページ (抜粋)

```
NAME
       execve - execute program

SYNOPSIS
       #include <unistd.h>

       int execve(const char *filename, char *const argv [], char *const envp[]);

DESCRIPTION
       execve() executes the program pointed to by filename.  filename must be either a binary
       executable, or a script starting with a line of the form "#!  interpreter [arg]".

       argv is an array of argument strings passed to the new program.  envp is an array of
       strings, conventionally of the form key=value, which are passed as environment to the new
       program.  Both, argv and envp must be terminated by a null pointer.  The argument vector and
       environment can be accessed by the called program's main function, when it is defined as int
       main(int argc, char *argv[], char *envp[]).

       execve() does not return on success, and the text, data, bss, and stack of the calling
       process are overwritten by that of the program loaded.  The program invoked inherits the
       calling process's PID, and any open file descriptors that are not set to close on exec.

RETURN VALUE
       On success, execve() does not return, on error -1 is returned, and errno is set
       appropriately.
```

2

きめられた一定個数の要素を格納できる領域 (**有限バッファ**) を介して，データをスレッド間で受け渡しするプログラムを考える．生産者スレッドと呼ばれるスレッドは以下の producer という関数を実行する．それは，無限ループを回りながら乱数を生成し続け，それら有限バッファに順次格納する (putdata 関数)．消費者スレッドと呼ばれるスレッドは以下の consumer という関数を実行する．それは，無限ループを回りながら有限バッファからデータを取り出し続ける (getdata 関数)．以下ではこの putdata/getdata 関数を正しく書くことが話題である．以下で，random() は乱数 (整数) をひとつ発生させる関数である．ひとつの乱数の生成には非常に短い時間 (100ns 以下) しかかからない．生産者スレッドは複数走っているかもしれず，消費者スレッドはひとつのみ走っていると仮定する．consumer 関数中で 100,000 要素ごとに一度，何個の要素を取り出したかを表示して，プロセスの進捗を把握できるようにしてある．

```
/* 生産者スレッド (複数いる) */
void producer() {
  int i;
  for (i = 0; /* true */; i++) {
    putdata(random());
  }
}
```

```
/* 消費者スレッド */
void consumer() {
  int i;
  for (i = 0; /* true */; i++) {
    getdata();
    if (i % 100000 == 0) {
      fprintf(stderr, "progress: i = %d\n", i);
    }
  }
}
```

有限バッファは N 要素の配列 a であらわされている．a の一要素に一個のデータを格納することが出来る．それ以外に，これまでにバッファから取り出されたデータの個数を保持する変数 p と，これまでにバッファへ格納されたデータの個数 (すでに取り出されたもの，まだ取り出されていないもの両者の合計) を保持する変数 q を保持して実現されている．これら 4 つの変数は大域変数で，a, N は適切に初期化され，その後変化することはない．p, q は 0 に初期化され，決して減少することはない．p, q がオーバーフローすることはないと仮定してよい．

```
/* 大域変数 */
int * a;                    /* a = 配列の先頭アドレス */
int N;                      /* 配列の容量 */
volatile int p;             /* これまでに取り出された個数 */
volatile int q;             /* これまでに格納された個数 */
```

以下はこの課題を解いている二人の学生の会話である．これを読んで問に答えよ．

央 (ひさし): こんなの簡単じゃん．データを入れるほうはデータを入れて q を増やして，データを出すほうはデータを出して p を増やしゃいいんだろ．そらできた．

```
void putdata(int c) {
  a[q % N] = c;
  q++;
}
```

```
int getdata() {
  int c;
  c = a[p % N];
  p++;
  return c;
}
```

女友達: まったく基本がなってないわね．バッファが空のときにデータを取り出せるわけはないしバッファが満杯のときにデータを突っ込めるわけはないんだから，ちゃんとそれぞれ同期のためのコードを入れなくちゃ．

央: あーそういえば．まあでもそのくらいは想定の範囲内．同期のための API は色々あるらしいけど，授業聞いててもよくわかんなかったら無視無視．そんなもの使わなくたってこうすりゃいいじゃん．バッファが空ってのは $q-p=0$ ってことで，バッファが満杯てのは $q-p=N$ ってことだから，そらできた．

```
void putdata(int c) {
  while (q - p >= N) /* do nothing */ ;
  a[q % N] = c;
  q++;
}
```

```
int getdata() {
  int c;
  while (q - p == 0) /* do nothing */ ;
  c = a[p % N];
  p++;
  return c;
}
```

女友達: まだだめね．生産者スレッドが複数走ってたらどうなるか考えて．

央: あーそういえば，[(a)] なんていうことがおこる可能性もあるね．まあこれも想定の範囲内だけど．

女友達: そうそう．複数のスレッドが同じデータを読み書きしているときは，常にそういうことに気をつけておかなくてはならないんだな．こういう状態を，なんていうんだったかしら?

央: あー，[(b)] ね．俺はそんなチマチマした[(b)] なんかより，むしろレース**クイーン**に興味があるんだけどな．

女友達: しょうもないこと言ってないで，どうやってなおしたらいいのか答えなさい!

央: はいはい，そういうときは「排他制御」をすればいいんだったよね．これも想定内，想定内，と．しかたがないから排他制御 API (lock/unlock) を使ってと，そらできた．

```
/* 排他制御用データ構造体 M(大域変数) を初期化 */
pthread_mutex_t M = PTHREAD_MUTEX_INITIALIZER;
void putdata(int c) {
  while (q - p >= N) /* do nothing */ ;
  pthread_mutex_lock(&M);
  a[q % N] = c;
  q++;
  pthread_mutex_unlock(&M);
}
```

```
int getdata() {
  int c;
  while (q - p == 0) /* do nothing */ ;
  pthread_mutex_lock(&M);
  c = a[p % N];
  p++;
  pthread_mutex_unlock(&M);
  return c;
}
```

女友達: ダメ．(c)まだ[(b)]が残ってる．

央: あー，えーっと，そうか，ここも排他制御してあげなくちゃね．まだまだ想定の範囲内です．

```
void putdata(int c) {
  pthread_mutex_lock(&M);
  while (q - p >= N) /* do nothing */ ;
  a[q % N] = c;
  q++;
  pthread_mutex_unlock(&M);
}
```

```
int getdata() {
  int c;
  pthread_mutex_lock(&M);
  while (q - p == 0) /* do nothing */ ;
  c = a[p % N];
  p++;
  pthread_mutex_unlock(&M);
  return c;
}
```

女友達: ほーらやった．ありがちなミス．[(b)] はなくなったけど，今度は[(d)] するわ．走らせてごらん．

央: (走らせて見て) あ，ほんとだ．progress: i = 0 以降，ぜんぜん消費者スレッドの進捗が表示がされないや．

女友達: もう一度 OS の教科書でもよく読んでみたら?

央: (話は聞かずに) . . . あーそうかそうか，ここで排他制御を lock したまま同期のためのループを回るのがいけないんだね．じゃ，ええっと，少し同期のコードを書きかえて，相手のスレッドを待つときは排他制御を unlock してあげて，と．これでどうだ．ここで，while 文を抜けるときは lock を握りっぱなしで抜けるところがミソだね．だから (b) は生じないっと．あ，ちなみにこれも想定の範囲内ね．

```
void putdata(int c) {
  while (1) {
    pthread_mutex_lock(&M);
    if (q - p < N) break;
    pthread_mutex_unlock(&M);
  }
  a[q % N] = c;
  q++;
  pthread_mutex_unlock(&M);
}
```

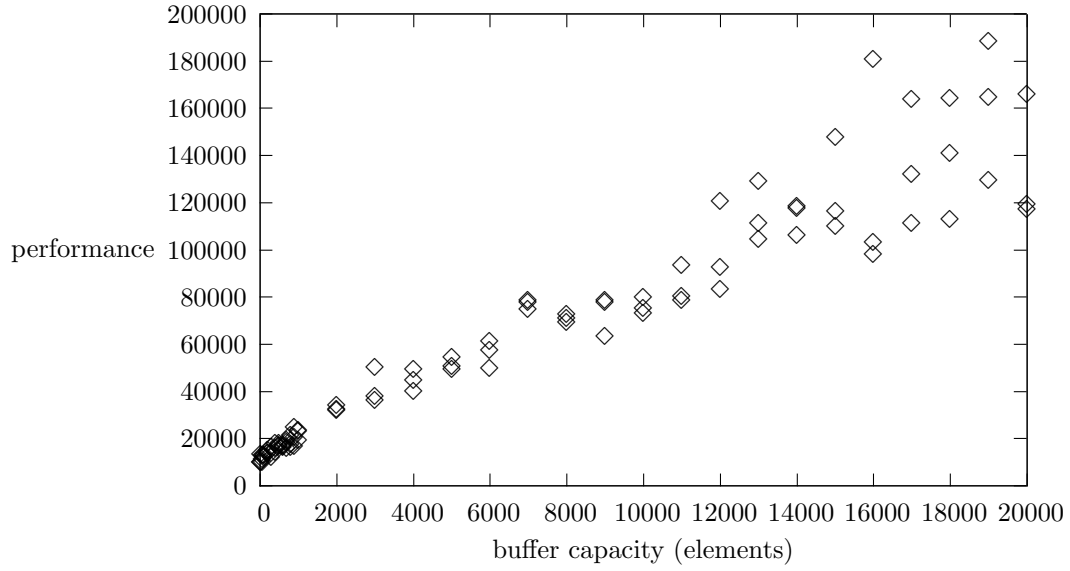
```
int getdata() {
  int c;
  while (1) {
    pthread_mutex_lock(&M);
    if (q - p > 0) break;
    pthread_mutex_unlock(&M);
  }
  c = a[p % N];
  p++;
  pthread_mutex_unlock(&M);
  return c;
}
```

女友達: んー，それっぽくはなってきたけど，まだ問題があるわ．それを見るために，バッファの大きさ N を色々変えて走らせてごらんなさい．

央: じゃあまずは $N = 1,000,000$ くらいにしてほらっ，そらできた．

女友達: 有限バッファっていうくらいだから，もうちょっと N を小さくしたり，色々変えてみなきゃ．

央は N を色々と変えながら性能を測り，プロットする．「性能」は，プログラムを 10 秒以上走らせ，消費者スレッドが 1 秒当たりに受け取ったデータの数で測られている．すると N と性能の関係として以下のようなデータが得られた．



央: たしかに，N が小さいと極端に性能が悪いね．

女友達: そう．なぜだかわかる?

央: んー，んー，(大分考えて) あ，そうか! (e)

女友達: ご名答．じゃあ，具体的にどういう風に書き換えればいいか，やってみて．

央: は，はい．ちなみにこれも想定の範囲. . . 内. . . です．

央，30 分後，やっとの思いでコードを書きあげる．

央: (f)で，できた.

女友達: はあ，やっとできたわね．央君はこれまでは,「すばらしい若者です」「わが弟です」とおもってたんだけど，この程度だったなんて，あなたの実力は粉飾されてたってことね!

央: ぐさーーっ，そんな手のひら返しはいくらなんでも想定外だった〜!

女友達: さてはこないだのテストが 100 点だったっていうのも，風説の流布だったのね!

央: その容疑は否認いたします...

女友達: あんたはもうおしまい．私の中でのあなたの株価は 100 日連続ストップ安よ!

問:

(a) には，このプログラムを走らせた結果起こりうる，誤った挙動についての説明が入る．そのような誤った挙動が起こる具体的なシナリオ (実行の履歴) と共に，どういう誤った挙動が起こるのかを説明せよ.

(b) に入る適切な用語を，**央の発言の残りの部分を踏まえて**書け.

(c) で残っている (b) とはどんなものか．(a) と同様の説明を背よ.

(d) に入る適当な用語を書け.

(e) には，このプログラムの性能が，N が小さいときに悪い理由を説明する文章が入る．特に，グラフを見ると N が小さいときの性能は大体 N に比例するようである．このことに対する定量的な説明を含めた文章を 5-10 行程度で書け.

ただし，二人が課題を実行しているコンピュータは 1 CPU (コア) を搭載している.

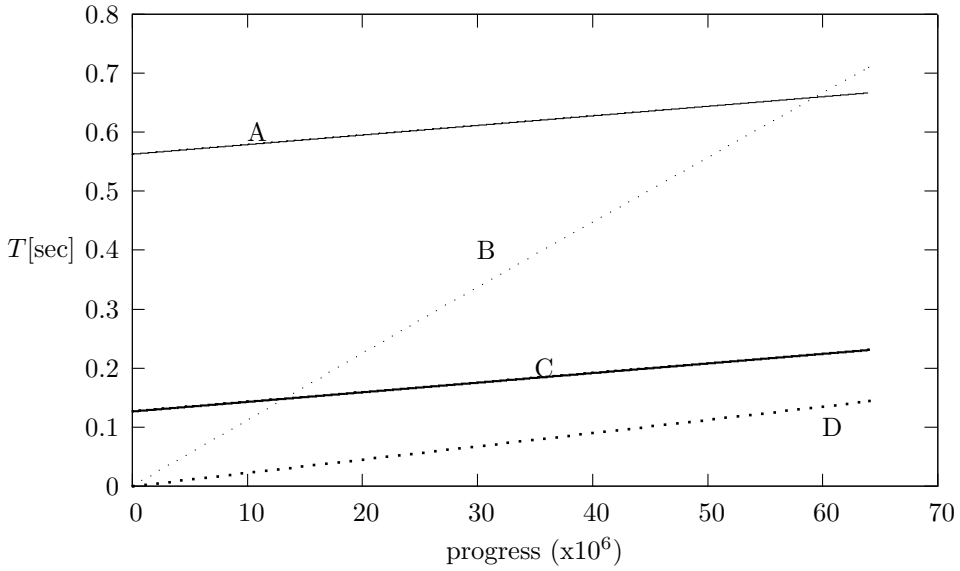
(f) 央にかわり，最終的に正しいプログラムを書いてみよ.

3

大きさ N バイトのファイル F に含まれる改行文字 ('\n') の数を数える，以下のようなプログラムを考える．簡単のためにエラー検査などは省略している．空欄 (P) の部分でポインタ a を設定しており，それを通じてファイルの全内容がアクセスできるようにあるシステコールが使われている．

```
int count_newlines() {
  int fd = open(F, O_RDONLY);
  char * a;
  int s = 0;
  int i;
  (P)
  for (i = 0; i < N; i++) {
    if (a[i] == '\n') s++;
  }
  return s;
}
```

次のグラフは，このプログラムの性能が (P) の部分で用いられている「ファイルの読み方」と，このプログラムが起動された時の「状況」に応じてどう変わるを示している．横軸は進捗，つまりプログラム中の for 文が回った回数を示している (単位は 10^6)．縦軸は，この手続き開始時からの経過時間を示している．それは (P) を実行する時間も含むことに注意せよ．



なお，ファイルの大きさは 64MB であり，この計算機は 512MB のメモリを搭載している．この測定を行った際は，対象プログラムをひとつだけ立ち上げており，他に活動しているプロセスは存在しなかった．

「ファイルの読み方」と「状況」はそれぞれ以下に示す 2 通りずつあり，都合 $2 \times 2 = 4$ つの場合がある．グラフの 4 つの線がそれぞれ，そのうちのどれかの場合に対応している．

「ファイルの読み方」は以下の 2 通り:

読み方 M: Unix の mmap システムコールや Windows の MapViewOfFile API を使って，ファイル全体をプロセスのアドレス空間にマッピングする．

読み方 R: ファイルの大きさと同じだけの領域を，malloc 関数を用いて確保し，そこへ Unix の read システムコールや Windows の ReadFile API を用いてファイルの内容を一度に全て読み込む．

「状況」は以下の 2 通り:

状況 C: コンピュータが起動して間もない状態にあり，そのファイルは起動してから一度も使用されたことがない．

状況 W: コンピュータが起動して以来，そのファイルは頻繁に使われており，つい最近も，たとえば cat コマンドのようなコマンドによって先頭から終わりまで読み出されている．

以下の問に答えよ．

(1) 状況 C と状況 W とでは，このプログラムの性能に違いが出るようだが，それはなぜか．それらでは，計算機の**状態**がどのように違うのかを，その違いをもたらす**オペレーティングシステムの仕組み**と共に説明せよ．

(2) 「ファイル全体をプロセスのアドレス空間にマッピングする」とは具体的にはどういうことか．この方法でこのプログラムを実行した際に，(P) の中，および実際に配列 a の要素が for ループ中でアクセスされていく過程でシステム内でおきている事や，オペレーティングシステムが行う事を説明せよ．

(3) 以上を総合して，グラフの A, B, C, D が，どの「ファイルの読み方」(M または R) と「状況」(C または W) を組み合わせた時の性能であるかを，根拠と共に答えよ．

問題は以上である