

# 平成27年度オペレーティングシステム期末試験

2016年1月19日(火)

- 問題は3問
- この冊子は、表紙1ページ(このページ)、問題2-8ページからなる
- 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。
- 各問題の解答は所定の解答欄に書くこと。

# 1

巨大なメモリを搭載するコンピュータ上で、以下の A0～A4 までの 5 種類プログラムを走らせることを考える。

## A0:

```
1 int main() {  
2     char * a = malloc(A);  
3     /* --- */  
4 }
```

## A1:

```
1 int main() {  
2     int fd = open(file, O_RDONLY);  
3     char * a = malloc(A);  
4     ssize_t r = read(fd, a, A);  
5     read_array(a, B); /* 中身は後述 */  
6     /* --- */  
7 }
```

## A2:

```
1 int main() {  
2     int fd = open(file, O_RDONLY);  
3     char * a = mmap(0, A, PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, 0);  
4     read_array(a, B); /* 中身は後述 */  
5     /* --- */  
6 }
```

## A3:

```
1 int main() {  
2     int fd = open(file, O_RDONLY);  
3     char * a = mmap(0, A, PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, 0);  
4     write_array(a, B); /* 中身は後述 */  
5     /* --- */  
6 }
```

## A4:

```
1 int main() {  
2     int fd = open(file, O_RDWR);  
3     char * a = mmap(0, A, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);  
4     write_array(a, B); /* 中身は後述 */  
5     /* --- */  
6 }
```

ここで `read_array(a, m)` および `write_array(a, m)` は、それぞれ `a` から始まる `m` バイトを読み出すまたは書き込む関数で、以下に相当する動作をする。

```

1 void read_array(char * a, ssize_t m) {
2     for (ssize_t i = 0; i < m; i++) {
3         a[i];
4     }
5 }
6 void write_array(char * a, ssize_t m) {
7     for (ssize_t i = 0; i < m; i++) {
8         a[i] = 'x';
9     }
10 }

```

- (1) A0 を走らせた時消費する物理メモリの量 (おおよその値) を答えよ。答えはプログラム中のパラメータ  $A$ ,  $B$  を用いた式で表せ。ただし  $A$  や  $B$  はある程度 (少なくとも数十 M) の大きな値である。A1~A4 についても同様に答えよ。

以下を仮定してよい。

- システム関数呼び出し (open, malloc, mmap, read) は全て成功する。
- read は要求したバイト数分のデータを実際に読み出す。
- 関数呼び出しやメモリアクセスが実際の計算には不要だからといって、コンパイラの最適化で除去されることはない (プログラムはあくまで書かれた通りのことを実行する)。
- 上記プログラム中で明示的に行われるメモリ割当量に比べて、それ以外に必要な (例えばプログラムのコードやスレッドのスタックのための) メモリ量は小さいと考え、無視する (0 とする)。

なお、消費されるメモリ量は、走らせているプログラムが /\* --- \*/ の行に達した時点での消費量とする。

- (2) A0 を  $P$  個同時に立ち上げるとどうか? 答えは  $A$ ,  $B$ ,  $P$  を用いた式で表せ。消費されるメモリ量は、 $P$  個全てのプロセスが /\* --- \*/ に達し (そこでしばらく停止し) たところでの消費量とする。A1~A4 についても同様に答えよ。
- (3) 以下は OS がメモリ管理のために用いてる技法であるが、それぞれどのようなもので、どのような時にどのような利点があるか、簡潔に説明せよ。
- (a) 要求時ページング
  - (b) Copy on write
- (4) (2) の A2 および A3 の物理メモリ使用量の結果を、要求時ページング, copy on write という言葉を、関連があれば適宜使いながら、それぞれ説明せよ。

## 2

以下の関数 `f0`~`f3` はいずれも、あるメモリ領域に  $N$  回 1 を足し、終了後にその値を表示するという関数であり、対象となるメモリ領域だけが異なる (for 文以降はほとんど共通; 重要部に下線が引いてある).

**f0:**

```
1 int x = 0;
2 void * f0(void * _) {
3     for (int i = 0; i < N; i++) {
4         x = x + 1;
5     }
6     printf("it's %d\n", x);
7     return 0;
8 }
```

**f1:**

```
1 void * f1(void * _) {
2     int * p = (int *)calloc(1, sizeof(int));
3     for (int i = 0; i < N; i++) {
4         *p = *p + 1;
5     }
6     printf("it's %d\n", *p);
7     return 0;
8 }
```

**f2:**

```
1 void * f2(void * arg) {
2     int * p = ((arg_t *)arg)->m;
3     for (int i = 0; i < N; i++) {
4         *p = *p + 1;
5     }
6     printf("it's %d\n", *p);
7     return 0;
8 }
```

**f3:**

```
1 void * f3(void * _) {
2     int fd = open("/tmp/zero_file", O_RDWR);
3     int * p = mmap(0, sizeof(int), PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
4     for (int i = 0; i < N; i++) {
5         *p = *p + 1;
6     }
7     printf("it's %d\n", *p);
8     return 0;
9 }
```

ただし、f2における arg\_t は以下のような構造体である。

```
1 typedef struct {  
2     int * m;  
3 } arg_t;
```

また、f3における/tmp/zero\_file は事前に作られ、中身はすべて0で埋まった4 (sizeof(int)) バイトのファイルであるとする。

f0~f3それぞれの関数を、 $P$  個のスレッドもしくは  $P$  個のプロセスで並行に走らせる。例えば f0 を  $P$  個のスレッドで走らせる場合、

```
1 void run_threads() {  
2     arg_t args[P];  
3     int * m = (int *)calloc(1, sizeof(int));  
4     for (int i = 0; i < P; i++) {  
5         args[i].m = m;  
6     }  
7     pthread_t tids[P];  
8     for (int i = 0; i < P; i++) {  
9         pthread_create(&tids[i], 0, f0, &args[i]);  
10    }  
11    for (int i = 0; i < P; i++) {  
12        pthread_join(tids[i], 0);  
13    }  
14 }
```

のようにする。f0 を  $P$  個のプロセスで走らせる場合、

```
1 void run_procs() {  
2     arg_t args[P];  
3     int * m = (int *)calloc(1, sizeof(int));  
4     for (int i = 0; i < P; i++) {  
5         args[i].m = m;  
6     }  
7     pid_t pids[P];  
8     for (int i = 0; i < P; i++) {  
9         pids[i] = fork();  
10        if (pids[i] == 0) {  
11            f0(&args[i]);  
12            exit(0);  
13        }  
14    }  
15    for (int i = 0; i < P; i++) {  
16        waitpid(pids[i], 0, 0);  
17    }  
18 }
```

のようにする (2-6 行目は run\_threads と run\_procs で共通である)。

$f1 \sim f3$  を走らせる場合は、それぞれの中の  $f0$  と書かれた部分 (`run_threads` の 9 行目および `run_procs` の 11 行目) を、適切な関数名に置き換えるだけである。これで都合  $2(\text{run\_threads}, \text{run\_procs}) \times 4(f0 \sim f3) = 8$  通りのプログラムができたことになる。各スレッド (ないしプロセス) が, “it’s  $x$ ” という表示をするので, いずれのプログラムもそのような行を  $P$  個表示することになる。

以下では  $N = 10000000, P = 2$  とする。以下の問いに答えよ。

- (1) 8 通りのプログラムそれぞれについて, その出力としてあり得るものを, 以下の中から全て選び, 記号で答えよ。

(ア)

```
1 it's 8193611
2 it's 10000000
```

(イ)

```
1 it's 8193611
2 it's 10242445
```

(ウ)

```
1 it's 10000000
2 it's 10000000
```

(エ)

```
1 it's 10000000
2 it's 10242445
```

(オ)

```
1 it's 10193611
2 it's 10242445
```

(カ)

```
1 it's 19706152
2 it's 20000000
```

(キ)

```
1 it's 19706152
2 it's 20000001
```

(ク)

```
1 it's 20000000
2 it's 20000000
```

解答は、解答用紙の表で、あり得るケースにチェック (✓) を入れて答えること。

(例)

		(ア)	(イ)	(ウ)	(エ)	(オ)	(カ)	(キ)	(ク)
run_threads	f0	✓	✓	✓	✓				
run_threads	f1		✓	✓		✓			✓
⋮	⋮								

- (2) 例えば (オ) のような出力が生じうるケースについて、それがどのようにして生じるのかを説明せよ。
- (3) pthread\_mutex\_t 型の大域変数 M を用意し、各繰り返しの中で 1 を足す操作の前後を pthread\_mutex\_lock(&M) と、pthread\_mutex\_unlock(&M) ではさむことにする。

具体的には、f0 であれば以下のように変更する。下線部 が変更した部分。

```

1 pthread_mutex_t M;
2 int x = 0;
3 void * f0(void * _) {
4     for (int i = 0; i < N; i++) {
5         pthread_mutex_lock(&M);
6         x = x + 1;
7         pthread_mutex_unlock(&M);
8     }
9     printf("it's %d\n", x);
10    return 0;
11 }

```

f1～f3 についてもほぼ同じ変更を施す (唯一の違いは、pthread\_mutex\_lock(&M), pthread\_mutex\_unlock(&M) の呼び出しではさむのが、\*p = \*p + 1; である点)。

このようにしてできたプログラム達の出力はどうなるか? (1) と同じ形式で答えよ。

- (4) 多くのプロセッサに備わる compare-and-swap 命令について、その動作を説明せよ。
- (5) compare-and-swap 命令を用いて、pthread\_mutex\_lock, pthread\_mutex\_unlock で達成したのと同じようなことを達成することができる。具体的にどうすればよいか述べよ。答え方としては、f0 の x = x + 1; または f1～f3 の \*p = \*p + 1; の行を、どのように変更すればよいかを記せ (どちらを使っても良い)。

### 3

少年は指導教員から、「自分のノート PC のハードディスクの性能を測れ」と言われた。そこで少年は、自分の PC (これはやや旧式で、SSD ではなくハードディスクを搭載している) 上で 1GB のファイルを作り、それを読み出すのにかかる時間を測定した。

```
1 # large という、1GB (1024 * 1024 * 1024 バイト)のファイルを作る
2 $ dd if=/dev/zero of=large bs=$((1024 * 1024)) count=1024
3 1024+0 レコード入力
4 1024+0 レコード出力
5 1073741824 バイト (1.1 GB) コピーされました、3.03388 秒、354 MB/秒
6 $ time cat large > /dev/null # cat で読み出し、時間を測る
7 real    0m0.149s
8 user    0m0.000s
9 sys 0m0.148s
```

少年はこの結果を持って、「1GB が 0.149 秒で読めたとあるから、ハードディスクの読み出し性能は 1GB/0.149 秒  $\approx$  6.7GB/秒くらいです、ついでに言うと、書き込み性能は dd の結果から、354MB/秒くらいですよね」と報告したところ、指導教員に (1) アホかと言われてしまった。そして、「ハードディスクの性能が測りたいなら、あることに気をつけて測定しないとダメだ」と言われた。そこで (2) きちんと気をつけて測定したところ、

```
1 $ time cat large > /dev/null
2 real    0m7.902s
3 user    0m0.007s
4 sys 0m0.212s
```

という結果が得られ、約 120MB/秒程度の性能となり、指導教員にはその数字ならあり得ると言われた。

次に、「1GB のファイルが 7.902 秒で読み出せることはわかった。じゃあ、1GB のファイル中の、指定された 1KB の部分 (だけ) を読めと言われたら、それにかかる時間はどのくらいか」と聞かれた。少年は調べてきますと言いつつ、「1GB が 7.902 秒だから 1KB にかかる時間は、実験などしなくても簡単に計算できる」と思い、

$$7.902 \text{ 秒} \times 1\text{KB}/1\text{GB} \approx 7.902 \times 10^{-6} \approx 8\mu\text{秒}$$

さも実験をしたようなふりをしつつ「約 8 マイクロ秒でした」と答えたところ、「ウソつけ」と言われしまった。(3) 「カタログに載っているこの数字を見れば、数ミリ秒より速いはずがないんだよ」とのことであった。

少年は、OS が行っているファイル読み書き性能が、いくつもの (4) ソフトウェア的な工夫によって達成されていることを学んだのであった。

以下の問いに答えよ。

- (1) 下線部 (1) について、少年の測定はなぜ間違いなのか、説明せよ。6.7GB/秒という数字がハードディスクの性能でないとしたら、いったいなぜこんな数字が出たのか?
- (2) 下線部 (2) について、正しく測定するための手順 (の例) を述べよ。
- (3) 下線部 (3) について、カタログに載っているどのような数値を元に、「数ミリ秒より速いはずがない」と判断できるのか?
- (4) 下線部 (4) について、仮に 1KB 読むのに数ミリ秒はかかるのだとしたら (仮に 1 ミリ秒かかるとする) これを比例させると、1GB 読むのには 1000 秒はかかる計算になる。それが実際には 7.9 秒で読み出せているのはどのような工夫を OS がしているからなのか?

問題は以上である



所属学科		氏名
------	--	----

1	(1)	A0	0	(2)	A0	0	
		A1	2A (A Δ)			A1	A(P + 1) (O(AP) Δ)
		A2	B (2B Δ)			A2	B (2B Δ)
		A3	2B (B, A+B Δ)			A3	B (P + 1) (O(BP) Δ)
		A4	B (2B Δ)			A4	B (2B Δ)
	(a)	sbrkやmmapなどのメモリ割り当てAPI呼び出し時には物理メモリを割り当てず、その後実際にアクセスが行われた時点で、当該ページごとに物理メモリを割り当てる。メモリ割り当て時点での時間を短縮できる。またメモリ割り当て後実際には使われないページがある場合物理メモリの節約になる					
	(b)	複数プロセスがforkやPRIVATEなmmapの結果同一ページのコピーを持つことになった場合、実際に書き込みが行われるまで物理ページを共有し、書き込みが起きた時点でコピーを作る。それらのページが実際には余り変更されない場合、forkやmmapの高速化、物理メモリの節約になる					
	A2	要求時ページングの効果によりファイルの大きさによらず実際にreadした大きさ(配列Aのうち書き込まれた分)だけしか物理メモリは割り当てられない。またmmapした領域に書き込まないので、copy-on-writeの効果によりmmapされた領域の物理メモリは全プロセスで共有される					
	A3	要求時ページングの効果によりファイルの大きさによらず実際にreadした大きさ(配列Aのうち書き込まれた分)だけしか物理メモリは割り当てられない。各プロセスがmmapした領域に書き込むので、各プロセスごとにBだけの物理メモリが消費される					

[illegible]

3	<p>(1) ファイルを作った直後には, そのファイルはおそらく主記憶上にキャッシュされているため, ハードディスクからの転送性能ではなく, 単なるメモリ上のコピーの性能を測っている.</p>
	<p>(2) ファイルをキャッシュから追い出してから測定する. 具体的な手順の例としては, メインメモリよりも大きなメモリを割り当て実際にアクセスするプログラムを実行する. もしくはキャッシュからデータを追い出す明示的なコマンド(Linuxであれば <code>sysctl -w vm.drop_caches=3</code>) を実行する.</p>
	<p>(3) ハードディスクは回転する円盤がヘッドの下に来るのを待つて目標のデータにアクセスするため, アクセスが来たら読みだすまで, 平均して半回転相当の時間程度はかかる. カタログ値として回転数が載っており, 例えば毎分12000回転としても毎秒200回転で, 半回転に2.5ミリ秒程度かかる(実際のハードディスクの回転数は5000-10000回転程度).</p>
	<p>(4) ファイルを先頭から終わりまで順に読むようなアクセスパターンに対して, 近い将来アクセスが予測されるデータの入力を事前に発行しておく(先読み). また, ファイル上で連続した領域を, ディスク上でも連続した(ヘッドを動かさずに読める)領域に割り当て, 大きな単位でアクセスを発行する.</p>

# 解説

1

2

あり得る表示は,

$$\begin{aligned} N &\leq m \leq 2N, \\ 0 &\leq M - m \leq N - 1 \end{aligned} \tag{1}$$

を満たす  $(m, M)$  の組全て及びそれだけである.

まず上記を満たす任意の  $(m, M)$  の組に対して, 実際にそのような値が表示されうることを示す.

$0 \leq a \leq N - 1$  を満たす任意の  $a$  に対して, 以下のような実行の履歴があり得る.

$T_0$	$T_1$	$x$ の値
	$R_0$	0
$R_0$		0
$\vdots$		$\vdots$
$W_{a-1}$		$a$
	$W_0$	1
$R_a$		1
$\vdots$		$\vdots$
$\dagger \quad W_{N-1}$		$N - a + 1$
	$R_1$	$N - a + 1$
	$\vdots$	$\vdots$
$*$	$W_{N-1}$	$2N - a$

よって特に,  $a = 2N - M$  とすれば,  $T_1$  が  $M$  を表示することになる. またこのとき  $T_0$  が最後の書き込みを終了した時点 ( $\dagger$  の行の終了後) での  $x$  の値は  $-N + M + 1$  で,  $T_0$  はここから  $*$  までの任意の時点で printf で表示する値を読みうるので,

$$-N + M + 1 \leq m \leq M,$$

つまり

$$0 \leq M - m \leq N - 1$$

を満たす任意の  $m$  に対して, その  $m$  が  $T_0$  によって表示されうることになる.

逆にどのような実行系列で表示される値も必ず式 (1) を満たすことを示そう. 実行している二つのスレッドを  $T_0, T_1$  としよう. まず両者とも以下のようなアクション  $R$  と  $W$  を交互に  $N$  回ずつ, 繰り返し実行していることを確認しておく.

$R: \quad t = x;$   
 $W: \quad t = t + 1; x = t; i = i + 1;$

ひとつのアクションは共有変数 ( $x$ ) へのアクセスと一度しか含んでいないので, 不可分に実行されるとみなして良い. 実際に起こりうる実行系列は, スレッド  $T_0$  が実行する  $R, W, R, W, \dots$  と  $T_1$  が実行する  $R, W, R, W, \dots$  を, 任意にインターリーブさせたものである.<sup>1</sup>

<sup>1</sup> $W$  の中に一件 3 つの命令が含まれるのに,  $W$  は不可分に実行しているという前提での議論に疑問を感じる向きもあるかも知れないが, 順番が入れ替わることによって結果が変わるのは, 共有メモリへのアクセスだけなのでこれでよい. 例えば,  $W$  の命令を  $X, Y, Z$  の 3 つに分解したとして, 二つのスレッドが  $X, X', Y', Y, Z', Z$  という順番で実行したものと,  $X', Y', Z', X, Y, Z$  という順番で実行したものは同じ結果になる. 実際に  $x$  へ書き込んでいる命令 ( $Y'$  と  $Y$ ) の順番さえ入れ替えなければ結果は変わらない. したがって  $X, Y, Z$  の 3 つは時間軸上隣り合って (すなわち不可分に) 実行されるとして構わない.

キーとなる不変条件は以下である．各繰り返しの先頭 ( $R$  実行直前および  $W$  実行直後) 以下が成り立つ．

$$\min(i_0, i_1) \leq x \leq i_0 + i_1 \quad (2)$$

ただし  $i_0$  ( $i_1$ ) は,  $T_0$  ( $T_1$ ) が持つ変数  $i$  の値である．

この条件が, あるスレッドが  $R; W$  を一回実行した時に保存されることを示そう．一般性を失うことなく,  $T_0$  が  $R, W$  を一度ずつ実行する系列に注目する．

```

R:   t = x;
      ⋮
       $T_1$  の任意の命令列
      ⋮
W:   t = t + 1; x = t; i = i + 1;

```

という一連のアクションの前後で, 式 (2) が保存されることを示す． $T_1$  の  $i$  の値の,  $R$  実行前時点での値を  $i_1$ ,  $W$  実行後時点での値を  $i'_1$  と書いて区別する．同様に  $x$  の値の,  $R$  実行前時点での値を  $x$ ,  $W$  実行後時点での値を  $x'$  と書いて区別する．

$$\min(i_0, i_1) \leq x \leq i_0 + i_1 \Rightarrow \min(i_0 + 1, i'_1) \leq x + 1 \leq i_0 + 1 + i'_1$$

$R$  と  $W$  の間に,  $T_1$  が  $W$  を  $n(\geq 0)$  回実行したとする．すなわち  $i'_1 = i_1 + n$ ．一般に,

$$\min(x + a, y + b) \leq \min(x, y) + \min(a, b)$$

に注意しておく．

左側の不等式は,

$$\min(i_0 + 1, i'_1) = \min(i_0 + 1, i_1 + n) \leq \min(i_0, i_1) + \min(1, n) \leq \min(i_0, i_1) + 1 \leq x + 1$$

だからよい．右側の不等式は,

$$x + 1 \leq i_0 + i_1 + 1 \leq i_0 + (i_1 + n) + 1$$

だからよい．

### 3