平成29年度オペレーティングシステム期末試験

2018年1月16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 2-10 ページからなる**
- 解答用紙は 2 枚. **1 枚目は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

以下の会話を読んでその後の問いに答えよ. [1]

利奈みんとちゃん (以下 M): 大野先輩, 今年はお正月早々, Meltdown だの Spectre だの, 恐ろしい名前の脆弱性が報告されて大変ですね.

大野桜子先輩 (以下 O): そうなのよ, それも単にボンクラな間違いがあったというのではなく, CPU の高速化の基本原理に根ざしたものなので, その影響が広範であるということで話題になっているのよ.

M: Meltdown と Spectre はどう違うのですか?

O: 攻撃の方法自体は非常に似ているのよ. 原論文 (どちらも `https://meltdownattack.com/` から入手可能) では両者は異なる攻撃方法を説明しているけど, 本質的にはどちらも同じと言っていいと思うわ.[2]攻撃の対象が違っていて, Meltdown は,

> "Meltdown breaks the most fundamental isolation between user applications and the operating system."

と言っているとおり, それを用いて OS 内部のデータを漏洩できることを示しているのよ. Spetre は, 同じプロセス内にあるデータなんだけど, 例えば javascript みたいな高級言語で, 言語仕様上は許可されていないはずのデータを読み出せてしまう, ということを示しているのよ. でも攻撃の方法も, その結果起きることもほとんど同じと言っていいと思うよ.

M: ふ〜ん, で, Meltdown が破っている, "the most fundamental isolation between user applications and the operating system" てのは一体何のことなのでしょうか?

O: そうね, まずはその前に, OS の基本である, 異なるプロセス間の分離 (isolation between different user applications), つまり, **あるプロセス A が別のプロセス B のデータを漏洩させようと思ってできない仕組み**, はどういうものであったかしら?

M: あ〜, そうですね, そういえば OS の授業で, それぞれのプロセスを別々のアドレス空間におくんだ, って習って気がします.

O: まあでもそういう抽象的な, ゆるふわな言い方じゃなくて, もう少し, CPU がどういう仕組みを持っていて, OS がそれをどう使って, それが実現されているのかを説明してくれる?

M: は, はい, じゃ, [(a)] くらいでどうでしょうか?

O: ま, 合格点ね. じゃ, 次に同じようなこととして, ユーザプロセスと OS の分離 (isolation between user applications and the operating system), つまり, **普通のプロセスが OS 内部のデータを漏洩させようと思ってもできない仕組み**, を説明してくれる? それが Meltdown によって破られてしまったんだけどね

M: あ, はい. う〜ん, それってさっきと同じじゃないんですか?

O: たしかに原理的にはほぼ同じ仕組みで守ることもできるよ. でも普通はそうしない, つまり, 普通のプロセスは OS と同じアドレス空間の中におくんだよ.

M: そうなんですか.

O: さて, それでも OS 内部のデータを漏洩できない仕組みはな〜んだ?

---

[1] この話のキャラクタは，シス管系女子（原作 Piro）のキャラクタを利用させていただいています．http://system-admin-girl.com

[2] 田浦の解釈

M: あ〜そういえば, そのための仕組みがありましたね. (b) . そんなところでどうでしょうか?

O: 正解. それをおさえたところで, Meltdown が何なのかを説明するよ. ちなみにさっきも言ったけど実際には Spetre も方法自体はほとんど同じで, 攻撃対象がたまたま OS じゃなかっただけ. Meltdown は, 任意のアドレス, たとえば OS カーネル内のアドレスみたいに, 本来読め出せないアドレス (たとえば a) であっても, その中身 (C 言語でいうところの$*a$) を高い確率でデータを推測できてしまうという, というものなの. そのために, ある命令が, 本来の意味では実行されていなくても—つまり, プログラムから直接観察できるような変化を引き起こさなくても— **投機的実行**によって, CPU 内部で部分的に実行され得るってことを利用しているのよ.

M: 投機的実行...

O: 例えば以下のコードを見て.

```
1 if (never_true()) {
2   secret = *a;
3 }
```

図 1:

ここで `never_true()` は 0 (偽) を返す関数であるとするよ.

M: なんの変哲もないコードですね. しかも, `never_true()` は 0 を返すんだから, then 節は実行されません. 要するに, 何もしない (noop) コードですね.

O: そう, だからこのプログラムを実行しても, 普通の意味での変化は何も起きない. 例えば変数 `secret` に`*a` の値が読み込まれたりはしないよね. でも, CPU の投機的実行により, CPU 内部でその一部が実行されることがある. この例では, if 文の条件部分の真偽が定まる前に, CPU が分岐方向を予測し, その予測に基づいて以降の命令を先行して実行することがあるのよ.

M: あー, 分岐予測ですね. アーキテクチャの授業で習った気がします.

O: もちろん後に分岐の方向が判明し, 予測が間違っていたとわかったところで実行の影響は破棄され, プログラムから直接観測されるような変更—例えば変数 `secret` の値の更新とか— はされないようになっているよ.

だけどここで, 投機的実行の影響 (プロセッサ内部に及ぼした, プログラムからは本来見えないはずの副作用) を**間接的**に観測することが可能な場合があるのよ.

M: う〜ん, 具体的にはどんなことがあるんでしょうか?

O: 例えば投機的に実行されている命令が, あるアドレスをアクセスした結果, そのアドレスの中身は CPU のキャッシュに残る, とかね.

M: はい, ちなみにここでのキャッシュは OS がファイルの中身をメモリに保存しておくときのキャッシュじゃなくて, CPU が, メモリの内容をプロセッサ内部に保存しておくときの, キャッシュですよね.OS の試験だから一応ことわっとこ...

O: うん. で, その結果, **どのアドレスの値がキャッシュにあるか否かを測定することができれば**, 間接的にアクセスされたアドレスがなんであったかを推測することが可能になってしまうのよ.

M: それってどうやってやるんですか? どのアドレスがキャッシュにあるとかないとかなんて, わかるんですか?

O: それを直接教えてくれるなんて言う命令はないけど, アクセスしてみて時間をはかってみればいいのよ. キャッシュにある場合とない場合で時間が違うからね.

M: げげ, なるほど, 実際にそのアドレスを盗み見たわけではないけど, どのアドレスをアクセスしたかはキャッシュの状態から, 間接的にわかってしまうわけですね.

O: うん, こういう, 本来の経路で情報を流さずに情報を流出させる経路のことを, side channel と言って, 至るところに存在するのよ. で, これに基づいて実際に構成した meltdown の基本構成部品は以下のようなもの. 実は以下は原論文によるものとは違うんだけど, 本質的なアイデアは同じだし, 説明がしやすくなるので以下を使うよ. 実際著者は別の節で, 「こんなやり方もある」と言及しているわ. 詳しくは原論文を参照してね.

```
1  #define STRIDE 4096
2  char array[256 * STRIDE];
3
4  uint8_t peek(uint8_t * a) {
5    wipe_array_from_cache();
6    if (never_true()) {
7      /* 実際にはこちらに分岐することはない */
8      uint8_t secret = *a;
9      array[secret * STRIDE];  // secret は 0..255の整数
10   }
11   for (long i = 0; i < 256; i++) {
12     if (access_time(&array[i * STRIDE]) < cache_miss_latency) {
13       /* (高確率で) secret == i  */
14       return i;
15     }
16   }
17   return 0;                      /* あきらめ */
18 }
```

O: この関数 `peek(`a`)` は, アドレス a におかれている 1 バイトの値 (0〜255 のいずれか)($*a$) を推測します. 仮に a が本来アクセス不能な—例えば OS 内部の—アドレスであっても.

M: ちなみに `uint8_t` は 8 bit の整数のことですよね. 6-10 行目は図 1 に示したものとほとんど同じですね. 唯一違うのは, a から読み出した値 (`secret`) を使って `array[secret * STRIDE]` をアクセスすること (9 行目) ですね.

O: そう, で, その結果, `array[secret * STRIDE]`, つまり, `array[0 * STRIDE]`, `array[1 * STRIDE]`, ..., `array[255 * STRIDE]` のどれか, がキャッシュに格納される. これは, 実際に then 節への分岐がおきなくても, プロセッサが分岐予測を間違えて then 節の実行を予測するだけでも起きるのよ.

M: あとはキャシュに格納されたのがどれだったかを当てればいいわけですね. なんか, さっきの side channel 攻撃が使えそう.

O: そう, 11 行目から 16 行目で, どの値がキャッシュに格納されたのかを, 各要素へのアクセス時間を測定することで推測する. ちなみに最初から `array` の一部がキャッシュにあるかもしれないからそれは 5 行目で, 全部キャッシュから追い出してから, 6-11 行目を実行します. この単純なコードではまだダメな理由がいくつかある— 例えば CPU が実際に分岐予測を間違えるようにするとか— んだけど, 詳細は省略ね.

M: Meltdown 攻撃を防ぐ方法はわかっているのでしょうか?

O: ソフトウェアで防ぐか, または起きにくくする方法はいくつか知られているよ. その中の一つ KPTI (Kernel Page Table Isolation) は, 本質的には, OS のデータを, ユーザプロセスと同じアドレス空間に置かず, 他のユーザプロセス同様に, 分離されたアドレス空間におくというもの.

M: そうですか. すぐにその対策が世界中でとられるといいですね.

O: そうね, でも, KPTI によって(c) 性能低下が引き起こされる可能性があって, さまざまな報告がなされているのよ. 一番大きいのは 30%とかね. もっとも, ほとんど影響を受けないアプリケーションもあるし, 人工的な例を作ればもっと影響の大きいものは作れそうだけど. だいぶ, アプリ依存というところがあって, それをどう報告するかは, 報告する人のバイアスがかかってしまいがちで難しいね.

(1) (a) に入る適切な説明を書け.

(2) (b) に入る適切な説明を書け.

(3) 下線部 (c) 「性能低下が引き起こされる」のはなぜか説明せよ. その上で, どのようなアプリケーションが影響を受けやすいか, あるいは受けにくいか, を述べよ.

2

内部に 10 要素の long 型の配列を持ち, そこに対して値を 1 つ追加する操作と, 値を 1 つ取り出す操作を持つデータ構造 (有限バッファ) の実装を考える. ただし, 追加される値は 0 以上の整数とする.

以下はそのための構造体 bb_t の定義である. volatile というキーワードの意味がわからなくても気にしなくて良い.

```
1  typedef long T;
2  enum { N = 10 };
3
4  typedef struct {
5    volatile T a[N];
6    volatile long w;
7    volatile long r;
8  } bb_t;
```

フィールド w, r はそれぞれ, これまでに追加された個数, 取り出された個数を数えているもので, 初期値は 0 である. 配列 a の初期状態は不定である. それを含め, bb_t を初期化する関数は以下で, 使う前に一度呼ばれる.

```
1  void bb_init(bb_t * bb) {
2    bb->r = bb->w = 0;
3  }
```

それを踏まえて, 追加 (bb_put) および取り出し (bb_get) の実装は以下のようになる.

```
1  int bb_put(bb_t * bb, T x) {
2    long r = bb->r;
3    long w = bb->w;
4    int ok;
5    if (w - r >= N) {
6      ok = 0;
7    } else {
8      bb->a[w % N] = x;
9      bb->w = w + 1;
10     ok = 1;
11   }
12   return ok;
13 }
14
15 T bb_get(bb_t * bb) {
16   T x;
17   long r = bb->r;
18   long w = bb->w;
19   if (w - r <= 0) {
20     x = -1;
21   } else {
22     bb->r = r + 1;
23     x = bb->a[r % N];
24   }
25   return x;
26 }
```

ただし以下に注意せよ.

(FULL) bb_put が, すでに N 要素格納されている (満杯な) 有限バッファに対して呼ばれたら, 要素を追加せずに 0 を返す (6 行目). そうでなければ要素を追加して 1 を返す (10 行目).

(EMPTY) `bb_get` が, 空の有限バッファに対して呼ばれたら, 要素を取り出さずに -1 を返す (20 行目). そうでなければ要素を 1 つ取り出してその要素 (≥ 0) を返す (23 行目).

以下ではこの実装を元に, スレッドセーフな (複数のスレッドから呼ばれても正常な動作をする) 有限バッファの実装を作る.

(1) この実装のまま, 複数のスレッドが `bb_put` または `bb_get` を繰り返し呼び出したときに, 以下の間違いが起きうるかを答えよ. 起きうる場合, どのようなタイミングでスレッドが実行されると起きるか, 具体的に述べよ.

(a) 追加されたはずの値 (`bb_put` に渡され, かつその呼び出しが 1 を返した値) が, 取り出されない

(b) 追加していない値が取り出される

(c) 1 回しか追加していない値が複数回取り出される

(2) 排他制御 (mutex) を用いて, それらの間違いが起きないようにせよ. 解答欄に上記のプログラムが載せてあるのでそれを修正して示せ. 修正は, 行間にコードを追加, 不要な行を消す, 行の一部を修正するなどをしてよい.

(3) スレッド間でデータを受け渡すために用いる場合, 満杯のときに `bb_put` が呼ばれたり, 空のときに `bb_get` が呼ばれたりしたら, それぞれ, 満杯または空でなくなるまで return しない, という動作が望ましい. なおその際に, 満杯のときの `bb_put` や空のときの `bb_get` が無駄に CPU を用いないようにする必要がある. この動作を, 排他制御及び条件変数を用いて実現し, 前問と同じ要領で示せ.

3

大きなファイルがあり, 中には n 個の倍精度浮動小数点数 (C の `double` 型) が格納されている. 倍精度浮動小数点数ひとつの大きさ (C の `sizeof(double)`) は 8 バイトであり, ファイルにはそれ以外のデータは格納されていない (つまりファイルの中身は, n 要素の浮動小数点数が格納された配列である). 以降, $N = 8n$ と書く (つまり小文字の n は要素数, 大文字の N はバイト数).

このファイル中の数の分布を調べたいのだが, ファイルが大きいので, 内容すべてを読み取る代わりに, ファイル中から $m(< n)$ 個の要素を取り出す (サンプリングする) ことにした. $M = 8m$ と書く (ここでも小文字は要素数, 大文字はバイト数).

具体的には以下の関数を書いた.

```
1 void choose(char * filename, long n, long m, double * a);
```

これは, n 個の浮動小数点数が格納されているファイル *filename* から m $(< n)$ 要素をサンプリングし, それを配列 a $(a[0], a[1], \cdots, a[m-1])$ に格納する, という動作をする.

この関数を以下の 3 通りの方法で実現し, 実行時間を測定した. なお実行時間は全て, **ファイルがキャッシュに存在しない状態で測定**している.

方法 (あ): n 要素の配列を作り, 全 n 要素を read で読み込む. そこから m 個の要素を乱数で選ぶ.

```
1 void choose(char * filename, long n, long m, double * a) {
2   int fd = open(filename, O_RDONLY);
3   double * b = (double *)malloc(sizeof(double) * n);
4   read_all(fd, b, sizeof(double) * n);
5   for (long i = 0; i < m; i++) {
6     long k = rand_long(0, n);
7     a[i] = b[k];
8   }
9 }
```

関数 `read_all(`*fd*`,` *b*`,` *sz*`)` は, ファイルディスクリプタ *fd* で指定されるファイルから, *sz* バイトを配列 *b* に読み込む関数である.

また, `rand_long(`a`,` b`)` は, a 以上 b 未満の整数を等確率で選ぶ乱数である.

方法 (い): ファイルを mmap で読み込む. そこから m 個の要素を乱数で選ぶ.

```
1 void choose(const char * filename, long n, long m, double * a) {
2   int fd = open(filename, O_RDONLY);
3   double * b = mmap(0, sizeof(double) * n, PROT_READ, MAP_PRIVATE, fd, 0);
4   for (long i = 0; i < m; i++) {
5     long k = rand_long(0, n);
6     a[i] = b[k];
7   }
8 }
```

方法 (う): ファイルの先頭から 1 要素ずつ読み込んで, それをある確率で配列 a に格納する. 具体的には, i 番目の要素は, 最初の m 要素までであれば無条件に $a[i]$ に格納する (6-7 行目). それ以降は, 確率 $m/(i+1)$ で, 配列のどこかの要素に格納 (8 行目) する. どの要素に格納するかは当確率 (9 行目) で選ぶ.

```
void choose(const char * filename, long n, long m, double * a) {
  FILE * fp = fopen(filename, "rb");
  for (long i = 0; i < n; i++) {
    double x;
    size_t rd = fread(&x, sizeof(double), 1, fp);
    if (i < m) {
      a[i] = x;
    } else if (rand_double(0.0, 1.0) < m / (double)(i + 1)) {
      long k = rand_long(0, m);
      a[k] = x;
    }
  }
}
```

ここで `rand_double(a, b)` は, a 以上 b 未満の `double` 型の浮動小数点数を, 一様な確率で返す乱数である.

これらを踏まえ, 以下の問いに答えよ. ただし,

- 実験を行った計算機のメモリ搭載量を P (バイト) とする. 少なくとも 1GB (2^{30} バイト) はあるとしてよい.
- この計算機のページサイズは 4KB とする. OS が物理メモリを割り当てたり, 削除したりする際の最小単位がページであることに注意せよ.

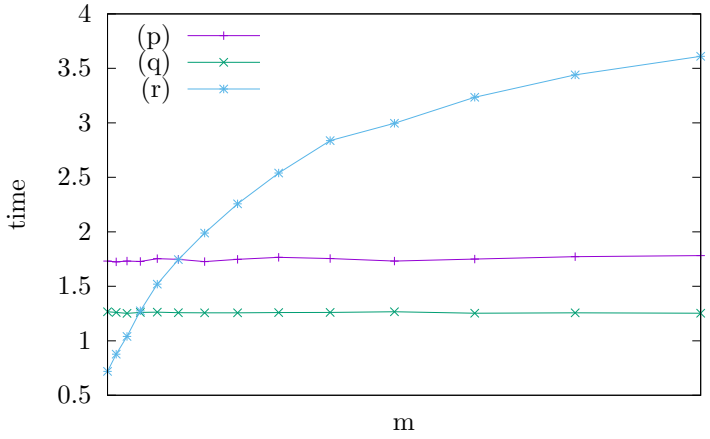
(1) mmap の挙動 (アプリケーションから見た挙動) と, その仕組みの概要について簡単に説明せよ.

(2) 前問を踏まえ, (い) の方法を使って 1000 個の要素をサンプリングする (つまり $m = 1000$) とき, ディスクからメモリに読み出されるデータ量 (トラフィック) は, およそいくらになるか?

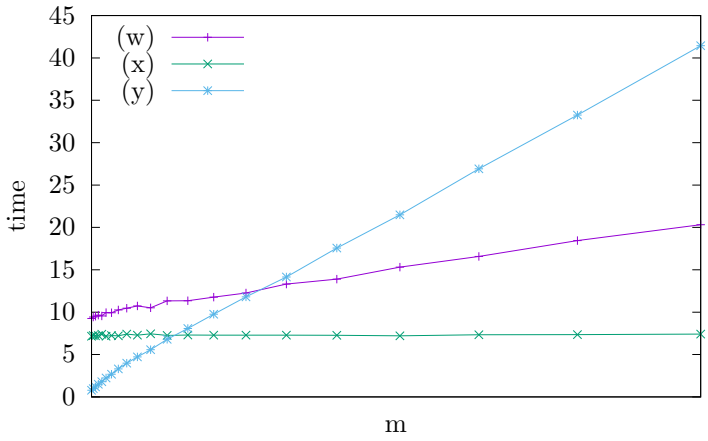
(3) $m = 1000$ に固定して, ファイルの大きさ (要素数) を, $n = 10000$ 程度から, N が P を優に超えるまで増やしていった.

このとき, それぞれの方法の実行時間 (choose 関数を開始してから終了までにかかる時間) がどうなるか, n を横軸に, 実行時間を縦軸にとって概形を描き, なぜそうなるのかを説明せよ.

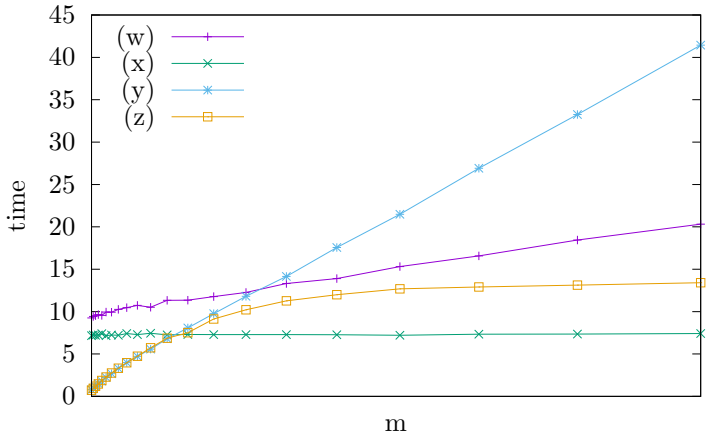
(4) ファイル全体がメモリに収まる状況, 例えば $N = P/2$ 程度の状況を考える. この状態で, m を, $m = 1000$ 程度から増やして行き, m を横軸に, 実行時間を縦軸にとって概形を描いたところ, 下図のようになった. グラフ中の (p), (q), (r) それぞれが方法 (あ), (い), (う) のどれか, またそう判断した理由を説明せよ.



(5) ファイルの大きさが優にメモリの大きさを超える状況 (例えば $N > 2P$) を考える. この状態で, m を, $m = 1000$ 程度から増やして行き, m を横軸に, 実行時間を縦軸にとって概形を描いたところ, 下図のようになった. グラフ中の (w), (x), (y) それぞれが方法 (あ), (い), (う) のどれか, またそう判断した理由を説明せよ.



(6) 方法 (y) を元に, 方法 (z) を考案し, 前問と同じ状況で性能を測定すると, 下図のようになった. どのような修正を施したのか示せ. どうして (y) と異なる動作になるのかの理由も書け.



問題は以上である

学生証番号		氏名	

1	(1)	CPUはメモリアクセスのたびにページテーブルを参照して, プログラムが指定した仮想アドレスを物理アドレスに変換する. OSは異なるプロセス(アドレス空間)のユーザアドレスは同一の物理アドレスに対応しないよう, ページテーブルを管理する.
	(2)	ページテーブルは, 各ページの属性として, ユーザモードでもアクセス可能であるかを保持しており, OSは, OS内部のページはスーパバイザモードでのみアクセス可能なように設定している.
	(3)	システムコールを呼び出すたびに, カーネルのアドレス空間を参照するよう, ページテーブルを切り替えなくてはならず, そのオーバーヘッドが発生する. CPUによっては, TLBが複数のアドレス空間を同時に保持することができず, その場合システムコールを呼び出すたびにTLBをフラッシュするオーバーヘッドが発生する. Read/writeなどデータを受け渡すシステムコールのために, ある程度の共有されたバッファが必要で, データはそのバッファを経由して(コピーされて)OSとやり取りされるのでそのオーバーヘッドも発生する. システムコールをほとんど呼び出さない(計算中心の)アプリケーションはほとんど影響を受けないがシステムコール, 典型的にはI/Oを多用するアプリケーションは大きく影響を受ける.

(1) 採点基準

- ページテーブル, TLB など, CPU が参照する機構 (MMU でも可)
- メモリアクセスごとにページテーブルが参照され, 論理アドレス → 物理アドレスへの変換が行われる
- OS は異なるプロセスに割り当てられる物理アドレスが重ならないようにする

3 つめの項目について, 「OS はプロセスごとに異なる論理アドレス空間を割り当てているため」あるプロセスが他のプロセスのメモリを参照できない, という答えが多かったが, これは文中にあった「ゆるふわな説明」というものであって, あまり仕組みの説明になっていない.

ここで問うているのはその「異なる論理アドレス空間」というのが一体どうやって実現されているのか, ということである. それは, OS が (意識的に), 「異なるプロセスには同じ物理アドレスを割り当てない」ということによって, 「結果的に」実現されていることに注意. OS が異なる論理アドレス空間のアドレスを同一の物理アドレスに変換することもできるし, 実際プロセス間の共有メモリなどはそうやって実現されている. あくまで「どの物理アドレスを割り当てるか」を OS が決めており, その割り当て方によって「異なる論理アドレス空間は分離されている」という状態が結果的に実現されているという関係を, しっかり理解しておいて下さい.

(2) 採点基準

- ユーザモード・特権モードの存在
- 各ページの属性として, ユーザモードでアクセス可能かという属性の存在

(3) 採点基準

- 遅くなる理由が書かれている. 中身は甘めに見ている
- 影響を受けやすいものは, 「システムコールを多用する」アプリケーションであること, または逆に, 影響を受けにくいのは, 「システムコールをほとんど使わない」アプリケーションであると書かれている

(4) コメント

- ほとんどのシステムコールでは, ユーザプログラムからは触れないデータに触る必要がある (さもなければシステムコールである必要がない). それを実現するためには, システムコールが呼ばれた際に, OS のアドレス空間に, アドレス空間を切り替える必要がある. これが遅くなる一義的な理由である. 一旦アドレス空間を切り替えてしまえば, OS のデータに触るたびにいちいち特別な処理をするわけではない.
- 解答を見ていると, 「OS のデータにアクセスするのに時間がかかるようになるから」みたいな答案が多いがさすがにこれは何も説明したことになっていない (「なぜ」時間がかかるようになるからを説明してもらわないといけない).
- 「システムコール内で OS のデータにアクセスするたびにアドレス変換が必要になるので遅くなる」みたいな解答も多かった. この「アドレス変換」が CPU が行うアドレス変換のことを意味しているのであれば, それは常に起こっていることであってそれが理由で遅くなる, というのは違う.

2

(1) (a) 起き得る・起き得ない (どちらかを○で囲む). 起きうる場合具体的な実行例: putをするスレッドが2つ(P, Q)あり, 初期状態(bb->r == bb->w == 0)から以下のタイミングで進捗した場合:
P: long r = bb->r; long w = bb->w; (r = 0; w = 0;)
Q: long r = bb->r; long w = bb->w; (r = 0; w = 0;)
この後は, 両者が bb->a[0] = x; を実行することになり, どちらかの値が失われる.

(1) (b) 起き得る・起き得ない (どちらかを○で囲む). 起きうる場合具体的な実行例:

(1) (c) 起き得る・起き得ない (どちらかを○で囲む). 起きうる場合具体的な実行例: bb_getをするスレッドが2つ(G, H)あり, bb_putが一度だけ行われた初期状態(bb->w == 1, bb->r == 0)から以下のタイミングで進捗:
G: long r = bb->r; long w = bb->w; (r = 0; w = 1;)
H: long r = bb->r; long w = bb->w; (r = 0; w = 1;)
この後は, 両者が x = bb->a[0]; を実行することになり, 同じ値が2度取り出される.

(2)

```
typedef struct {
  volatile T a[N];

  volatile long w;

  volatile long r;
  pthread_mutex_t m[1];
} bb_t;
```

```
int bb_put(bb_t * bb, T x) {
  pthread_mutex_lock(bb->m);
  long r = bb->r;

  long w = bb->w;

  int ok;

  if (w - r >= N) {

    ok = 0;

  } else {

    bb->a[w % N] = x;

    bb->w = w + 1;

    ok = 1;

  }
  pthread_mutex_unlock(bb->m);
  return ok;
}
```

```
T bb_get(bb_t * bb) {
  pthread_mutex_lock(bb->m);
  T x;

  long r = bb->r;

  long w = bb->w;

  if (w - r <= 0) {

    x = -1;

  } else {

    bb->r = r + 1;

    x = bb->a[r % N];

  }
  pthread_mutex_unlock(bb->m);
  return x;
}
```

(3)

```
typedef struct {
  volatile T a[N];

  volatile long w;

  volatile long r;
  pthread_mutex_t m[1];
  pthread_cond_t c[1];
} bb_t;
```

```
int bb_put(bb_t * bb, T x) {
  pthread_mutex_lock(bb->m);
  while (1) {
   long r = bb->r;
   long w = bb->w;
   int ok;

   if (w - r >= N) {
     pthread_cond_wait(bb->c, bb->m);
     ok = 0;

   } else {

     bb->a[w % N] = x;

     bb->w = w + 1;
     if (w - r == 0)
       pthread_cond_broadcast(bb->c);
     ok = 1;
     break;
   }
  }
  pthread_mutex_unlock(bb->m);
  return 1;
}
```

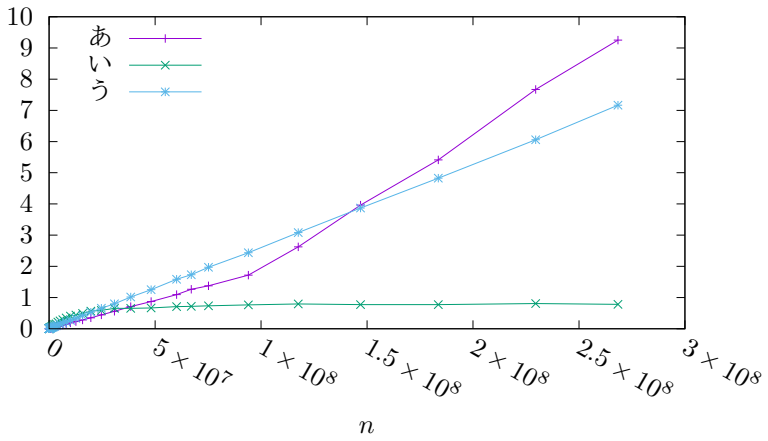
```
T bb_get(bb_t * bb) {
  pthread_mutex_lock(bb->m);
  T x;
  while (1) {
   long r = bb->r;
   long w = bb->w;
   if (w - r <= 0) {
     pthread_cond_wait(bb->c, bb->m);
     x = -1;

   } else {

     bb->r = r + 1;

     x = bb->a[r % N];
     if (w - r == N)
       pthread_cond_broadcast(bb->c);
     break;
   }
  }
  pthread_mutex_unlock(bb->m);
  return x;
}
```

3	(1)	ユーザから見るとmmapが返したアドレスから, 指定したサイズ分の領域に, ファイルの指定したオフセットからサイズ分が写像されたように見える. 仕組みは, 要求次ページングとほぼ同じで, mmapされた時点では写像されたアドレスに物理メモリは割り当てず, あるページに初めてアクセスが起きた際にページフォルトが起きるのでそこで対応する部分を(キャッシュされていなければ)ファイルから読み込んで対応する物理ページへ対応付ける.
	(2)	約 4KB x 1000 = 4MB程度 (注: 実際には同一ページが複数回アクセスされればこれより少なくなるが, ファイル全部合わせて最低でも 256K 程度のページがあるので1000のページはほぼすべて異なるとしてよい)
	(3)	
	(4)	(p) う　　(q) あ　　(r) い
		判断理由: (r)は, mが非常に小さいときにほとんど時間がかかっていないことからmmapを用いた(い)と判断できる((あ)(う)はn要素を全て読み込むのでその時間が支配的になる). (あ)(う)はどちらも実行時間がほとんどmによらないが, 一要素ごとの処理が(う)の方が大きいため, (p)が(う)と判断できる.
	(5)	(w) あ　　(x) う　　(y) い
		判断理由: (r)は, 前問同様(い)と判断できる. 今回の設定ではファイル全体が物理メモリに収まらないので, (あ)でファイル読み込み後に(7行目で)ランダムな要素をアクセスする際にメジャーページフォルトが起き得る. そのため前問と異なりmが大きくなるときの実行時間の増大も無視できなくなるし, ファイルを読み込んだ後の処理がない(う)よりも遅くなると予想される.

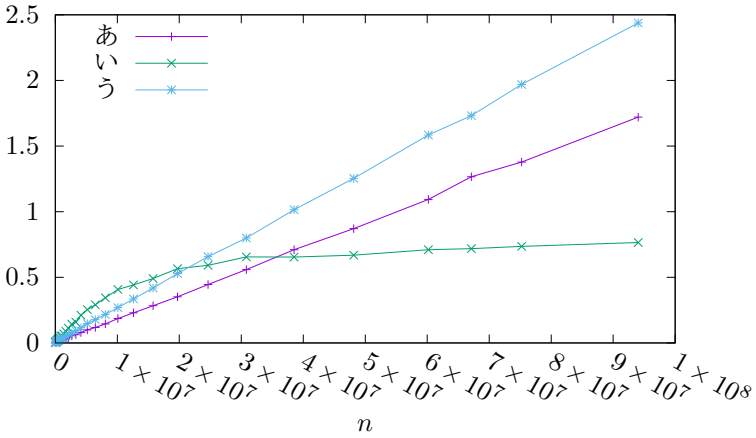
	(6)	void choose(const char * filename, long n, long m, double * a) {
		  int fd = open(filename, O_RDONLY);
		  double * b = mmap(0, sizeof(double) * n, PROT_READ, MAP_PRIVATE, fd, 0);
		  long * K = (long *)malloc(sizeof(long) * m);
		  for (long i = 0; i < m; i++) {
		  K[i] = **rand_long(0, n);**
		  }
		  qsort(K, m, sizeof(long), cmp_long);
		  for (long i = 0; i < m; i++) {
		  a[i] = b[K[i]];
		  }
		}

		(z)ではサンプリングされたデータが, 添字の小さい順にアクセスされている. そのため(y)と異なり, 同一のページ内のデータが時間的に連続してアクセスされ, 各ページに対して起きるページフォルトは一回だけとなる. よってmがある程度大きいときの実行時間は, 実質的に全てのページを読み出す(ほとんどmによらない)時間となる.

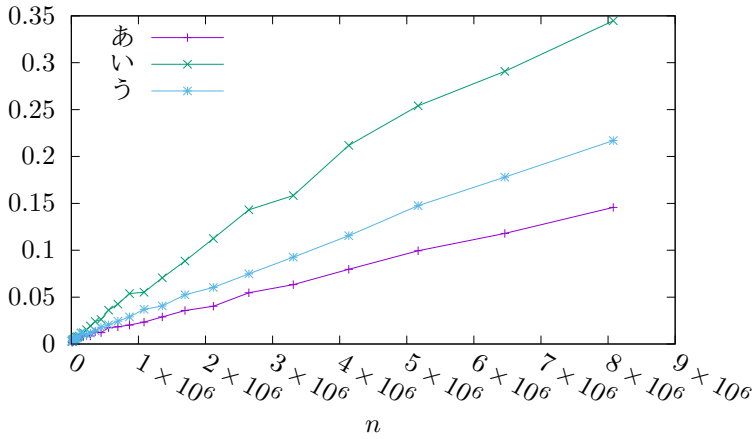
(3) の答え. 以下は同じグラフで, n をどの範囲で表示するかだけを変えたもの



- $0 \leq n < 3 \times 10^8$



- $0 \leq n < 10^8$



- $0 \leq n < 10^7$