平成14年度オペレーティングシステム試験解答例

実施: 2003年2月10日

1

(1) 実際に，最もありそうなエラーは次の二通り

- **(解答例 1)** 不正なメモリアクセス (Memory Protection Error, Segmentation Fault, など)
  OS によって，通常のユーザプロセスには読み書きの禁じられたアドレスに読み書きした際に生ずるエラー
- **(解答例 2)** 不正な命令の実行 (Illegal Instruction)
  特権命令を，ユーザモードで実行した際に生ずるエラー

下線部 (2) によくつながるのは後者である．しかし，大作が実際にどのようなプログラムを書いたかは不明であるし，I/O のための命令を実行する以前に，OS カーネル内の不正なメモリ参照を行い，そこで前者のエラーが発生する可能性も高いので，両者を正解とする．

講評　問題にある程度の曖昧さがあるために正解例は何通りもありうる．結局,「特権モードとユーザモードの存在」,「ユーザモードでは実行できない (実行するとトラップを発生する) 命令の存在」,「ハードディスクに直接アクセスを試みる命令が特権命令である」ということがわかっており，そこからありそうなエラーを書いたとみなされるものは正解としてある．例えば以下のような例は正解としてある

- ユーザモードで特権命令を出してはいけません
- スーパバイザモード以外ではハードディスクにアクセスできない

実際には，特権命令を実行したという例外から，それを「ハードディスクにアクセスしようとした」と推測することは簡単ではない (し，OS がそれをやる意味もない) ので，後者のようなエラーメッセージが出ることは現実問題としてはありえないのだが，正解としてある．

一方，単に「アクセスが拒否されました」「権限がありません」といった類の，問題の文脈からしたら「そりゃそうでしょう」としか，言いようのない解答はさすがに正解とはできない．この問題は，アクセスは拒否されるのだけれども，それがどの瞬間におきるのかがわかっているか，つまりもう一段下のレベルの仕組みがわかっているかどうかを試しているのである．それがつまりは保護の「仕組み」ということである．

(2) CPU には，特権モードとユーザモードという二つのモードがある．ユーザプロセスはユーザモードで実行され，ハードディスクに直接アクセスするような命令 (特権命令) を実行しようとするとトラップが生ずる．OS はそのトラップに対してその命令を実行させないよう，処理をする (典型的にはそのプロセスを終了させる).

講評　一番正解率が高かった.

正解とできない解答例は,

- スーパーバイザコールを通してしかハードウェアにアクセスできないようにする
- ハードディスクなどを OS の管理下におくことによりハードディスクへの直接アクセスを禁じる

といった，あまり情報のない答案で，まさにそのための「仕組み」を聞いているのである．

(3) OS は，仮想記憶ハードウェアを利用して，OS 内のメモリ領域を,「特権モードでのみ実行可能」であるように保護する．よって，OS 内の命令にジャンプ命令によって制御を写しても，この保護によって「不正なメモリアクセス」例外が生ずる

講評　上のように解答している答案は少なかった．厳しくは正解とはできないが今回は正解とした，非常に多かった解答は以下のようなものだった．

> アドレス変換の仕組みによって，各プロセスがアクセスできるメモリ空間は独立 (分離) している．したがってユーザが扱える論理アドレスをいくら変えてみても，別のプロセスのメモリにはアクセスできない

これは，プロセス A がプロセス B のメモリにアクセスできないことの説明としてはまったく正しい．だが，厳密にはここでは，プロセス A がカーネル (OS 内部) のメモリにアクセスできない仕組みを聞いている．もちろんカーネルの利用するデータとプロセスの利用するデータがアドレス空間として分離しており，カーネルデータはプロセス A のアドレス空間内には存在しないという OS の設計もありえなくはないが，実際にはそうなっていない．かわりにカーネルのデータは「すべての」プロセスのアドレス空間内の同じ場所に存在しているという設計が典型的である．それは，授業の最終回の豊嶋君の発表などにも出てきた，アドレス空間配置というもので決まっている．では，すべてのプロセスのアドレス空間内に存在しているカーネルデータを保護する仕組みは何かというと，仮想記憶ハードウェア (TLB やページテーブル) 中に保持されるページの保護属性の中にある,「ユーザモードで利用可能であるかどうかを示すビット」である．つまり，全てのプロセスにとって，カーネルデータは，アドレス空間内に存在はするが読み書きすることはできない．

もちろんここでは，必ずしも上述したような細かい区別を知っていることを要求していたわけではないので，上のような解答も全て正解とした．ただ，上で解説したような仕組みはユーザとカーネルの間でデータのやり取りを行おうと思ったらかなり必然的なものであるということは，自分で考えて納得してみて欲しい．例えばユーザがシステムコールで渡したデータをカーネルが読もうと思えば，ユーザのアドレス空間とカーネルのアドレス空間が完全に分離されていたら不便だと想像がつく．

(4) トラップ，割り込み，ソフトウェア割り込み，など

講評　割り込みという言葉は通常,「外部からの割り込み」たとえばタイマ割り込みや，デバイスからの割り込みを指すのに使われるが，ユーザプロセスが特権モードに移行するために発行する命令も，結局外部割込みと同じ効果をもたらすため,「割り込み命令」と呼ばれる．

(5) 割り込みベクタ

講評　「割り込みベクタレジスタ」という答案が複数あった．あまり正解とはいえないが，正解としてある．割り込みベクタはメモリ上の「割り込み時に制御が移行するアドレスが記入してある一連の領域」のことである．割り込みベクタレジスタはそもそもその一連の領域の (先頭) アドレスを保持するレジスタのことである．したがって (5) に当てはまるのは，割り込みベクタであって，割り込みベクタレジスタではない．しかし，おまけの正解としてある．

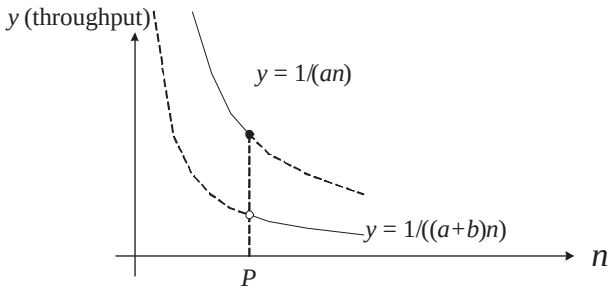


図 1:

2

(1) 要素 $A[i * \texttt{PAGESIZE}]$ を含むページを a_i と書くと，このスレッドは n 個のページ $(a_1, \cdots, a_n)$ を循環的に (つまり，$a_1; a_2; \cdots; a_n; a_1; a_2; \cdots; a_n; \cdots$ のように) アクセスする．よって，$n \leq P$ のときこれらのページはすべて物理メモリに収まり，ページフォルトは一切おきない．一方 $n > P$ のとき，全てのアクセスはページフォルトを起こす $(*)$.

よって，

- $n \leq P$ のとき，スループットは $1/an$
- $n > P$ のとき，スループットは $1/(a+b)n$

グラフは図 1 のようになる．

注 $(*)$ の理由: たとえば a_n をアクセスするときを考える．LRU 置換により，ある時点で物理メモリ上に存在するページは「最近アクセスされた P 個のページであり」これは (配列 A 以外へのアクセスを無視すれば), $a_{n-1}, a_{n-2}, \cdots, a_{n-P+1}$ のことである (注: $n - P + 1 \geq 1$)．この中に a_n は含まれない．他のページに対しても同様である．

(2) 各スレッドが n 個のページを循環的にアクセスする．

- $nt \leq P$ の時，(1) と同様の議論によりページフォルトは全くおきない．よって全体のスループットは，(1) と同様，$1/a$ である．
- $nt > P$ のとき，(1) と同様の議論により，全てのアクセスがページフォルトを起こす (注 $*$)．このとき全体のスループットは，CPU の処理能力から来る限界 ((1) と同様の $1/an$) と，1 スレッドのスループットの限界 $(1/(a+b)n)$ とで押さえられる．つまり，$\min\left(\frac{t}{a+b}, \frac{1}{a}\right)$ である (注 2)

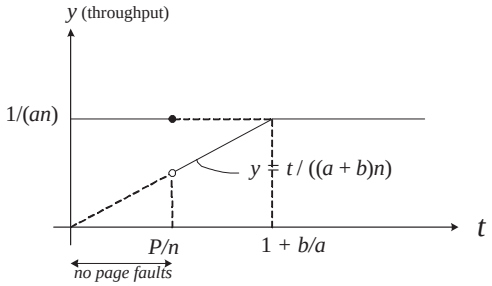
グラフの形は，$nt > P$ の領域で min をとる項が入れ替わるかどうか，つまり，$nt = P$ を満たす $t = P/n$ と，$\frac{t}{a+b} = \frac{1}{a}$ を満たす $t = 1 + b/a$ の大小で異なり，図 2 のようになる．

(注 $*$) 少しわかりにくいかもしれないが，図 3 が，$nt > P$ の場合にどのようにスレッドが実行されるのかを示している．横軸が時刻，実線部が compute を実行している時間，破線部がページフォルトによってブロックしている時間である．ただし，図では compute を実行中のタイマによるスレッドの切り替えは無視している．

スループットとはつまり，この絵をある決められた幅 (つまり単位時間) で切って取り出したときにいくつの「実線部 + 破線部」の組が入っているかということである．ただし，二つのスレッドが同時にひとつの CPU を使うことができないので，実線部同士は横軸に投影したときに互いに重なってはいけない．

このように考えれば，スレッド数が少ないうちはスレッド数を増やすにつれてスループットが比例して向上することがわかるだろう．しかしそれはやがて頭打ちになる．それは実線部同士が重なりだしたところである．その時点で結局 CPU は常にどれかのスレッドを実行していることになり，スループットはページフォルトがないときのそれと同じになる．

1 + *b*/*a* > *P* / *n*:



1 + *b*/*a* <= *P* / *n*:

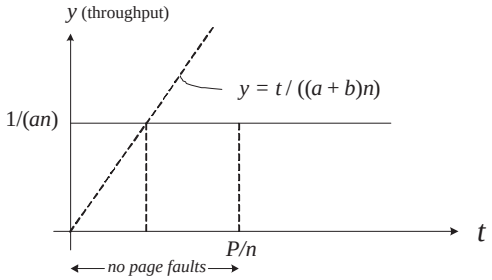


図 2:

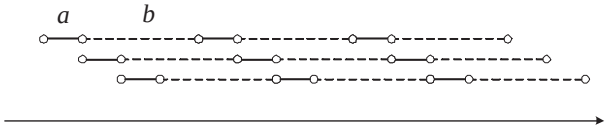


図 3:

講評　(1) で，$n \leq P$ の場合はかなりよくできたいた．$n > P$ のときは，全てのアクセスでページフォルトがおこるのだが，それをなぜか，n ページ中 $n - P$ 回のページフォルトとしている答案が非常に多かった．

(2) では，まず基本的な OS の仕組みとして，

> ページフォルト時には，ページフォルトを起こしたスレッドはブロックし，変わりに他のスレッドに CPU が割り当てられる

ということがわかっていないと話しにならない．そして，$nt > P$ のときは，あるスレッドが (最高でも)a [ms] CPU を消費しては次のスレッドに切り替わるということをずっと繰り返すということを理解しなくてはならない (もちろん実際には a [ms] 中にもスレッド切り替えによるスレッドの切り替えが起こることはありうる)．図 3 がイメージできれば下り坂のはずだが，やはり正確に求めるのは難しかったかもしれない．

ところで，(2) の結果は，スレッド数を増やせば結果としていくらメモリを使ってもスループットを高く (ページフォルトがない場合と同じに) 保てるという結果である．つまり，CPU がディスクやネットワークをアクセスしている間，その空き時間を埋めるべく多数のスレッドを作っておけば，CPU を無駄にしない，つまり，ディスクアクセスやネットワークアクセスによる「空き時間」を隠蔽できるということを教えてくれている．

実際にはこの結果はもちろん，ディスクのバンド幅が無限という (非現実的な) 仮定の元で成り立っていた話である．実際には頻繁にページフォルトを起こせば今度は全体の性能がディスクとのアクセスによって律速される可能性がある．したがってページフォルトを起こしながら，その遅延を多数のスレッドで隠すというような技法を意図的にすることはないと思ってよい．ページフォルトは「起こさないように」メモリの使用量に気をつけるというのが基本である．

ただし，この多数スレッドを作って遅延を隠すという方法は有用な概念で，とくに，ネットワークからデータを取り寄せる時のような，(ページフォルトと違って) 避けられない遅延を隠す際に有用である．このような場合に適度な数の

スレッドを作れば有効に CPU を利用できる．例えば WWW ブラウザがページを表示するときに，いくつかの画像ファイルを一つのスレッドが順に取り寄せる代わりに，複数のスレッドに分けて取り出せばよいのである．ネットワークのバンド幅が十分なら，この問題のモデルに近い形でスループット (単位時間に表示できる画像の数) が改善する．

3

OS は各スレッド (またはプロセス; 以下いちいち断らない)t に対して，その時点におけるスケジューリング優先度 $G(t)$ を管理する．典型的には以下のようにして目的を達成する．

1. 定期的に発生するタイマ割り込み時: 現在実行中のスレッド t の優先度 $G(t)$ を一定値 (例えば 1) 減少させる．$G(t) = 0$ となった時点で再スケジュール (次に CPU を割り当てるスレッドの選択) を行う
2. ページフォルト, I/O, 自発的な一定時間の休眠 (sleep) などによるスレッドのブロック (中断) が生じた時: 直ちに再スケジュールを行う
3. ページフォルト処理，I/O，自発的な休眠が完了した時: 直ちに再スケジュールを行う．
4. 再スケジュール時: その時点で実行可能 (ブロックしていない)，G の値が正でかつ最も大きいスレッドを実行する．このとき，全てのスレッドの G の値が 0 であればすべてのスレッドの優先度を再計算する．

1. と 4. により，複数のスレッドが CPU を利用している場合に，ひとつのスレッドが CPU を独占することなく，CPU が公平に分配される．2. により，CPU を無駄にすることなく,「実行可能なスレッドがある限り」CPU がスレッドに割り当てられる．3. により，ページフォルトや I/O などによる中断から復帰したスレッドが，復帰し次第直ちにスケジュールされる機会を得る．これは I/O によって中断することが頻繁な，対話的プロセスの応答を高める効果がある．

さらに，対話的プロセスの応答を高めるために，以下のような優先度の計算を行う OS が典型的である．

- 上記 2. において，中断から復帰したスレッドの G の値を一定値増加させる (Priority Boost).
- 上記 4. において優先度の再計算する際，全スレッドの G をある一定値 (例えば 10) にリセットするだけの場合もあるが，$G(t) = G(t)/2 + 10$ のようにして，$G(t) > 0$ のままブロックしたスレッドの優先度をあげておく．これによって，このスレッドが後に中断から復帰した際に高い優先度を持ち，直ちにスケジュールされる

注: 前者は Windows, 後者は Linux などで用いられているようである．後者は授業で説明したが前者については特に話していない．が，非常に単純な仕組みであり，思いつくことは易しい．むしろこれによって公平性が損なわれないかの方が考察を要する．