Teste de Sistemas Operativos
Correcção do Grupo I

fsm

12 Junho 2015

Resumo

Pretende-se com este documento ajudar a compreender a forma correcta de responder ao Grupo I do teste de Sistemas Operativos (SO), permitindo uma melhor preparação para os alunos que terão de se apresentar em exame. A intenção não é dar aqui soluções completas, "replicáveis"às cegas em qualquer outra prova de avaliação. A ideia é ficar a perceber os caminhos errados, quer de interpretação de enunciado quer de estratégia de resposta. Quando chegar ao fim do texto deverá ser capaz de, por si só, chegar à resposta correcta.

1 Antes de começar...

Se tivesse de caracterizar SO numa única palavra, que palavra escolheria? Em sua opinião, qual é o contributo desta Unidade Curricular (UC) para a sua formação em Informática (seja em Engenharia ou Ciências da Computação)? O que é que se aprende (ou devia aprender) aqui que seja diferente em relação às UCs anteriores? Se não está a ver a resposta, pense no programa da UC. Quais são so capítulos principais? O que é que nos ocupa mais, bem mais de 50% das aulas? Teóricas e práticas? Que boneco/diagrama apareceu em praticamente todas as aulas teóricas e foi referido em muitas das práticas? Já percebeu?

Esta introdução serve para recordar que nos momentos de avaliação, o importante é mostrar resultados da aprendizagem. Resultados. Não é mostrar que sabe muito do que não lhe perguntaram. É aplicar os resultados do que se aprendeu em SO e reolver o caso concreto da questão que lhe foi colocada. Não é responder a outra que saiu no exame anterior. Não é tentar desesperadamente responder armado só com maior ou menor dose de intuição ou do que já foi resultado de aprendizagem de outra UC. Por outras palavras, uma resposta ou programa que pode ter tido elevada cotação em UCs anteriores, em particular as de programação imperativa, por si só pouco ou nada garante aqui. É preciso completar a resposta com a parte específica de SO.

Já percebeu onde terá falhado no teste? Por exemplo, quando se diz que uma *espera activa* é proibida em SO, qual é a verdadeira razão para esta "proibição"? A resposta deve ser óbvia: é porque os SOs vivem da concorrência, ela própria uma forma de atingir um objectivo específico (qual?) e uma das novidades (requisitos?) em SO é precisamente saber substituir esperas activas por esperas passivas no controle dessa concorrência. Por isso, uma bela função de inserção/remoção em listas ligadas ou a pesquisa num array para ver onde há lugares livres serão basicamente perda de tempo se o resto, a parte de SO, não existir ou não tiver sido programada correctamente com as ferramentas, técnicas e chamadas ao sistema pedidas na pergunta.

Conclusão: comece sempre pela parte de SO e depois complete-a com o necessário para a resposta ficar completa. Se for pergunta teórica, comece pelo essencial de SO e não fuja muito daí. Se for prática, primeiro pense no que aprendeu em SO (que arquitectura terá a agora sua solução?) e só depois acrescenta "o C"suficiente para o programa funcionar. Não desate a escrever uma função de inserção e outra de remoção

numa lista só porque o enunciado menciona filas de espera. Isso é "primeiro ano"... Não é SO. Vale o mesmo, aliás vale menos, do que inventar o nome de uma função, especificar a sua assinatura e semântica e limitar-se a usá-la com os argumentos e resultados correctos. De facto, a probabilidade de nem sequer precisar de codificar inserções e remoções é muito grande em SO! Na secção seguinte irá ver um exemplo disso.

2 Primeira Questão

2.1 Enunciado

Num sistema de atendimento ao público com fila única os utentes têm de esperar por ordem de chegada até que um dos balcões esteja livre, dirigindo-se então para esse balcão. Não há nenhum funcionário a controlar, são os próprios utentes que garantem a sincronização entre si. O esquema geral da solução será então:

```
shared  int a, b, c[];   // vars partilhadas
main()
{
     // Se necessário, aguarda por um balcão livre, x
     balcao (x)  ; utente é atendido no balcao x
     // ...
}
```

Complete este programa, assumindo a existência de memória partilhada entre processos (declarada com `shared`*) e operações* `cria_semaforo`*,* `P` *e* `V`*. Procure resolver a questão para um número $N > 5$ de balcões de atendimento. Em caso de dificuldade, resolva para $N = 1$[1].*

2.2 Discussão

Que programa é este que me pedem para completar? Quem o vai executar? Quem são os "actores" desta simulação? Reparou que o enunciado apenas refere um programa? Um... E que diz que não há nenhum funcionário a controlar os utentes do serviço imaginário com N balcões (serviços académicos, finanças, ctt, etc.). Reparou que dentro do `main()` há a invocação de uma função que simula a ida ao balcão passado como argumento?

```
main()
{
      ...
      balcao (x)  ; utente é atendido no balcao x
      ...
}
```

Tudo isto mostra que é o próprio utente quem executa o programa. É só um programa, todos os utentes executam o mesmo programa. Só um, o que lhe foi pedido. Falta a sincronização e esta TEM DE SER implementada com os mecanismos pedidos no teste (que correspondem a cerca de 1/4 das aulas teóricas). Note que a declaração de variáveis como `shared` é só uma ajuda para não obrigar ninguém a recordar-se dos argumentos da system call `mmap` vista nas aulas teóricas. Declara-se o que se quer como `shared` e através de alguma magia (que será convidado a desvendar na questão seguinte) essas variáveis ficam visíveis a todos os utentes.

- Não há mais nada, nem outros programas (excepto talvez a inicialização do estado de arranque da simulação) nem outros processos. Não há `fork()`, logo não há `wait()`, nem `WEXITSTATUS`, nada.

[1] Nesse caso a cotação será consideravelmente inferior.

- Se são os próprios utentes que garantem a sincronização entre si, não há cá ninguém a coordenar, não há nenhum gestor. É só este programa, o dos utentes.
- Se só há utentes, também não há cliente-servidor, nem `pipes`, nem código tirado dos guiões ou do trabalho prático. Nada disso.
- Se o utente se dirige para o balcão `x`, parece evidente que o programa vai ter de descobrir um balcão livre, ie. atribuir a `x` um valor. Se só tem um balcão, é fácil...

Então como fica o programa. Vejamos primeiro como não fica:

- As variáveis

  ```
  shared int a, b, c[]; // vars partilhadas
  ```

  eram apenas exemplos! Se precisasse de mais ou menos, devia acrescentar ou retirar!
- Já se disse acima que não há criação explícita de processos, é o interpretador de comandos quem cria o processo de cada utente processos que de seguida faz o `exec` deste seu programa. Ou seja, o programa passa a ser um comando do sistema.
- É asneira grossa tentar encaixar aqui o trabalho prático. Aí havia clientes e um servidor, havia quem fizesse pedidos e aguardasse pela resposta. Aqui claro que não. Há um utente que vê se há balcão livre, se houver indica (não avisa, tipo sms... indica, deixa escrito em variáveis que todos os processos podem ver quando quiserem procurar um balcão livre) que o balcão deixou de estar livre porque agora está ocupado por ele próprio. Não tente reproduzir o que "aprendeu". Puxe pela cabeça, use o que é a matéria de SO e avalie a sua própria solução. Se não está a ver como se faz, não perca tempo, passe a outra pergunta e volte aqui mais tarde, quando tiver garantido várias respostas correctas.
- É asneira ainda mais grossa cada utente criar `N` filhos e mandar cada filho para um balcão diferente! Acha mesmo que se quiser ir aos serviços académicos leva 5 amigos/filhos e manda cada um para um balcão, atropelando qualquer pessoa que por acaso naquele momento esteja a ser atendida num desses balcões?
- E acha que, por criar filhos e o pai esperar que eles terminem (com `wait()`), cada filho já não precisaria de esperar por um balcão livre? Leia o enunciado com muita atenção! Sincronizar é atrasar, esperar por um evento. Não basta despejar o padrão `fork/wait` só porque o enunciado fala em espera.
- Então há filas de espera e não há inserções nem remoções? Não? Pois não... Nas aulas teóricas explicou-se o conceito e a implementação das operações sobre semáforos e viu-se bem que as filas existem mas *dentro* do sistema operativo. Aqui basta usar correctamente os `P()` e `V()` e o sistema operativo faz o resto. Tira processos de RUN e coloca-os em ESPERA nesse semáforo e vice-versa. Viu? Não precisa do código para inserção e remoção.
- Já que estou a falar de `P()` e `V()`, tenho de repetir que estas são OPERAÇÕES (leia-se system calls) sobre semáforos. Não há um semáforo chamado P e outro V. Há semáforos que são criados com um determinado valor inicial e depois há operações `P()` e `V()` sobre esse semáforo. Se tiver

  ```
  mutex = cria\_semaforo (1); // valor inicial é 1
  ```

  então já pode ter

  ```
  P(mutex) ... V(mutex)
  ```

  Se o semáforo se chama `s`, tem de ser `P(s)`. Usar semáforos que não foram criados nem se sabe qual o valor inicial, assim como semear `P()` e `V()` sem argumento, sem se saber sobre que semáforo actuam, é falhar completamente nos resultados de aprendizagem de SO. Cotação nula.

- Já agora, os semáforos criam-se e inicializam-se antes de serem usados. Várias respostas (curiosamente muito "iguais", sobretudo nos erros) tinham um algoritmo "do primeiro ano" e no fim, caso isso não servisse, então vamos lá criar um semáforo anónimo.
- Infelizmente ainda há quem confunda programa com função e escrever uma função com usar essa função. Vários alunos começaram por redefinir as operações `P()` e `V()`, "estragando-as". Sbem que as system calls, estão protegidas, o seu código não pode ser modificado, certo? Sabem que usar um `read()` para ler um caracter do teclado não é o mesmo que implementar a system call `read()`? Vamos procurar ler a avaliar as nossas próprias respostas?
- E se lhe pedirem para usar `P()` e `V()`, é com `P`s e `V`s que deve de responder. Não é com `wait()` nem `signal()`. Não faria grande diferença, desde que criasse correctamente os semáforos e usasse correctamente esses wait/signal com os respectivos semáforos. Foi isso que lhe disseram durante o teste, se não sabe com Ps e Vs, tente com wait/signal mas por sua conta e risco. Desde que não confunda: um `signal()` é equivalente ao `V()` e não "manda um sinal", altera o valor do semáforo e liberta processos se for apropriado. Não "manda" sinais, os do guião 8 das práticas. Se usasse `V()` talvez não dissesse essa asneira. E se usasse `P()`, talvez não caísse na tentação de criar filhos só para usar `wait()`. Percebe que o professor está a tentar ajudar? E quando pede para usar wait/signal, está muito provavelmente a desperdiçar a ajuda e a meter-se por caminhos errados?

Então como fica? Bem, quem foi às aulas teóricas tem a solução completa nos seus apontamentos. É o exercício do parque de estacionamento ou da sala de cinema sem lugares marcados. Dos outros esperar-se-ia que deduzissem que se só houver um balcão, então o acesso a esse balcão é a região crítica, pelo que bastaria acrescentar o semáforo `mutex` e respectivas operações

```
P(mutex); balcao(1); V(mutex);
```

Com vários balcões, só entraria na sala se houvesse um livre (semáforo de capacidade) e lá dentro teria uma região crítica para descobrir um balcão livre. Depois de ser atendido, nova região crítica para actualizar o estado do balcão, passando-o a livre.

Só isto. 3 a 6 linhas. Menos de 5 minutos?

3 Segunda Questão

3.1 Enunciado

Explique como, na questão anterior, se consegue partilhar um conjunto de endereços de memória por vários processos e simultaneamente garantir que cada processo mantém as suas próprias variáveis locais e globais.

3.2 Discussão

Vou também começar com uma pergunta. Várias. Qual a percentagem de alunos que acredita que a resposta a uma pergunta está no enunciado da pergunta anterior? Será idêntica dos que não conseguem interpretar um texto e confundem um exemplo com a resposta correcta ou uma pista para essa resposta? Tal como na primeira questão confundiram as variáveis `shared` e fizeram grande esforço para usarem as que foram dadas apenas como exemplo?

Se a pergunta tivesse sido *explique como se consegue partilhar um conjunto de endereços de memória por vários processos e simultaneamente garantir que cada processo mantém as suas próprias variáveis locais e globais*, as respostas teriam sido muito diferentes? Fará diferença ser na questão anterior? Se fosse com outros processos e

noutro algoritmo (por exemplo um produtor/consumidor, o do colocar/retirar copos do balcão) a resposta seria diferente?

Quais são as palavras chave da pergunta? Como se consegue X, em que X é *partilhar endereços de memória*? O professor é amigo, até diz o nome do capítulo onde está a resposta correcta. Endereços de memória não costumam ser estudados no escalonamento, nem na sincronização de processos, nem no input/output, etc. Deve ser memória. Tem de ser, mesmo que a system call que permite isso possa ser vista como pertencendo aos ficheiros, se houver ficheiros mapeados em memória (`mmap()`, lembra-se?). E faz diferença partilhar código ou partilhar dados? Faz muito pouco, é apenas uma questão de modo de acesso, read/write ou execute. E nas aulas falamos em partilhar bibliotecas, certo?

Então como se consegue ter uma ou mais zonas partilhadas, visíveis, acessíveis, mapeadas em... vários processos, quando se disse que em Unix e sistemas similares os processos têm espaços de endereçamentos distintos privados. Até vimos isso nas aulas práticas, o filho recebe uma cópia (embora por trás da cortina o código e muitos dados sejam partilhados, estes marcados como *copy-on-write*). Como se consegue não partilhar? E como se consegue partilhar código, por exemplo (lá está o exemplo), uma biblioteca, `libc` ou outra qualquer? Ou seja, proteger o espaço de endereçamento de cada processo e a mesmo tempo abrir parte dele só a alguns processos bem identificados?

A resposta está na implementação da memória virtual. O espaço de endereçamento que um processo pensa que tem, e os endereços que são emitidos pelos CPU ao executar instruções, são trocados antes de serem colocados no barramento de endereços da memória real. Como? Procurando numa tabela de segmentos que contém o modo de acesso permitido e o endereço real em memória, se estiver carregado em RAM. Claro que aqui seria muito mais fácil e rápido fazer um desenho. Bastavam 2 processos, duas tabelas de segmentos, algumas entradas dessas tabelas seriam privadas, apontariam para sítios diferentes de RAM, swap, etc, e uma entrada seria igual em dois processos (aliás em todos o utentes). Apontaria para o mesmo endereço físico em RAM. Acrescente-se o boneco da MMU para evitar a indexação em RAM, a fazer a validação do modo de acesso e tratar da *cache miss* (e outras tarefas irrelevantes para esta questão, como sejam saber o que foi usado recentemente para efeito de rejeição). Quem quisesse acrescentar paginação, tudo bem, tradução de endereços com 2 níveis.

Memória. Memória virtual. Mas acompanhando a descrição da *tradução de endereços* com um diagrama dos espaços de endereçamento virtuais e respectiva troca de bits para endereços reais, partilhados ou não. Não era despejar os benefícios da memória virtual nem dizer que com ela se consegue proteger. Ou partilhar. Isso já dizia o enunciado. A pergunta era *como* um sistema operativo faz essa magia.

Só isto. Outros 5 minutos?