Sugestões para a resolução das QUESTOES-FSO-Autoavaliação – parte 1

José C. Cunha, DI-FCT/UNL

O objectivo destas notas é apenas dar algumas indicações e sugestões no sentido das soluções para as questões apresentadas em QUESTOES-FSO-Autoavaliacao, documento presente na seccao Documentacao de apoio no CLIP. Estas notas não substituem a necessidade de os alunos tentarem resolver as questões por si próprios.

Questão 1. Diga quais as diferenças entre os pares de conceitos indicados em cada uma das seguintes alíneas:
a) "***file descriptor***" comparado com´´***inode***´´ no sistema Unix**.**
b) "***ficheiro normal*****"** comparado com ´´***directoria***`` no sistema Unix**.**
c) "***processo*****"** no sistema Unix, comparado com ´´***programa***´.

→ a) File descriptor – denotado por um número inteiro não negativo, indica o número de um canal de entrada/saída, ligado a um ficheiro. Tem associada uma entrada na Tabela de Canais privada de cada processo, sendo essa entrada reservada quando o processo invoca a chamada ao sistema fd = open( ) sendo o valor retornado fd igual ao número do canal. Essa entrada contém um apontador para a TabelaGlobaldeFicheirosAbertos do SO, havendo nesta tabela uma entrada por cada canal aberto pelos processos, na qual se guarda o cursor de byte corrente (byte offset), o modo de abertura do canal (read, write ou read/write) e um apontador para a Tabela-de-I-nodes em memória, a qual contém uma entrada por cada ficheiro correntemente a ser utilizado. Nesta última tabela cada entrada é um descritor do ficheiro, chamado i-node no Unix. O descritor contém informação sobre os atributos do ficheiro, tais como o utilizador-dono (em geral, quem criou o ficheiro), os direitos de acesso do dono, grupo e outros utilizadores, o tipo de ficheiro (directoria, ficheiro normal em disco ou ficheiro especial), a localização dos blocos do ficheiro em disco ou então, no caso de ficheiros especiais, a referência para os respectivos Device Drivers (conjuntos de rotinas de controlo do periférico ao qual o ficheiro especial está associado), e as datas de acesso e manipulação do ficheiro.
Assim, em resumo, o file descriptor representa um canal aberto para um ficheiro, e o i-node representa o descritor de um ficheiro.
→ b) Ficheiro normal em disco é representado no Unix como uma sequência de n bytes, de 0 a n-1, sobre a qual se podem abrir canais (via operacao open()) e depois fazer acesso sequencial a partir da posicao corrente do cursor de byte (via operações read e write) ou acesso directo a partir de qualquer posicao do cursor (após ajustado este com a operação lseek()).
Directoria é um ficheiro contendo uma lista de pares, sendo cada par formado pelo nome simbólico de um ficheiro e pelo número de i-node que lhe está associado.
→ c) Programa é um conjunto de instruções numa dada linguagem. No caso dos programas em forma executável por um computador, por exemplo, contido num ficheiro dito executável, as instruções do programa são as instruções da máquina (note que também as instruções de chamada de subrotina, pois o call-subroutine é uma instrução máquina, bem como as instruções de chamadas ao SO, pois estas também são invocadas com uma instrução máquina do tipo <int n>, como nos processadores Intel). Um programa tem uma natureza estática, isto é, apenas um conjunto de instruções e especificação dos dados sobre os quais estas operam. Mas, para as instruções serem executadas, é preciso que o programa seja activado. A activação de um programa é uma das funções essenciais que um SO deve suportar. Quando um utilizador (através de um comando no teclado, ou da selecção de uma opção de

um menu, ou da acção de click num ícon visível no écran) pede ao SO a activação de um programa, o SO primeiro tenta localizar o ficheiro que contém o programa, depois abre este ficheiro, consulta a informação (num cabeçalho) que descreve o conteúdo do ficheiro, tal como as secções de código e de dados, quantos bytes têm, quais os dados a inicializar ou só reservar em memória, qual o start-address do corpo do programa (main), quais os argumentos que lhe devem ser passados, aquando da activação, etc. Após ter reservado zonas adequadas de memória (RAM) para carregar aquelas secções de código e de dados, o SO reserva também espaço para as zonas de heap e de stack, construindo assim o que se chama o Mapa de Memória de um processo.

Processo é um conceito em que o SO se baseia para controlar a activação e execução de programas, garantindo que os seus mapas de memória são isolados (protegidos) uns dos outros, controlando a ordem de execução dos processos pelo CPU (pois que em geral há mais processos, isto é aplicações activadas, do que CPUs reais para as executar), e controlando as operações de entrada e saída, de modo a que quando um processo tiver de esperar pelo fim das operações dos periféricos, não tenha de o fazer em ciclos de espera activa, mas sim, bloqueado em filas de espera do SO, e dando oportunidade para o SO activar outros programas entretanto, de modo a manter uma elevada taxa de utilização do CPU (o que é conseguido através do mecanismo de multiprogramação). Portanto, enquanto um programa é uma entidade passiva e estática, um processo é uma entidade activa (que suporta a execução das instruções de um programa) e dinâmica, que passa por uma sucessão de estados ao longo do tempo, desde que é criado até que termina.

Em resumo, um programa é um conjunto de instruções e um processo é um programa em execução, com todo o contexto necessário para que o SO controle o progresso dessa execução (mapa de memória, estado actual de execução, valores dos registadores do CPU quando o processo se bloqueia, tabela de canais abertos, etc).

Questão 2. Explique, detalhadamente, as acções das seguintes chamadas ao sistema Unix, explique o seu efeito sobre as estruturas de dados internas do núcleo do sistema Unix que ache mais relevantes e indique o significado dos seus argumentos:

a) ***open***
b) ***link***
c) fork
d) ***close***
e) ***unlink***

→ Estas acções estão descritas nos acetatos e apontamentos.

Questão 3. Considere que, no sistema Unix, um processo foi criando processos filhos, por diversas invocações da chamada ao SO fork.

a) Usando chamadas ao sistema Unix, escreva, em C, o pseudo-código de uma funcao
int wait_any_child() que bloqueia o processo invocador até que um qualquer dos seus filhos termine a sua execução, devolvendo, como valor de retorno, o identificador do processo filho que tenha terminado.

b) Usando chamadas ao sistema Unix, escreva, em C, o pseudo-código de uma funcao
int wait_all_child() que bloqueia o processo invocador até que todos os seus filhos tenham terminado a sua execução, devolvendo, como valor de retorno, o identificador do processo filho que tenha terminado.

→ Uma vez que a chamada ao SO wait() não vem para o 1º teste, esta questão não vem para o 1º teste.

Questão 4. Considere um Sistema de Operação (SO) da família Unix.
a) Desenhe o diagrama de estados lógicos de um processo e todas as transições entre estados, durante a execução do processo num sistema Unix, desde que é criado até que é terminado. Indique os nomes dos diversos estados, defina as situações a que corresponde cada estado e, para cada transição do diagrama, indique as condições que a desencadeiam.

→ a) O diagrama de estados de um processo está nos acetatos e nos apontamentos. Indicam-se aqui apenas as transições entre estados.

-- Não_existente -> Criado -> Pronto: quando o SO cria um processo e prepara o seu mapa de memória, cria uma entrada numa TabelaGlobaldeProcessos com informação sobre os atributos do processo, obtém um identificador do processo criado (process identifier, pid) e insere-o numa fila global de processos no estado Pronto (os processos que, embora estivessem em condições de começar a executar de imediato, têm de ficar à espera de vez para executar, visto que em geral há mais processos do que CPUs)
-- Pronto -> Executando: quando o SO selecciona um processo (em geral o mais prioritário) da fila de processos prontos, para execução, indo então carregar os registadores do CPU com os valores correspondentes às zonas de dados, de stack (Stack Pointer), flags e código (Program Counter) que apontam para o mapa de memória do programa que esse processo deve começar a executar. O SO neste ponto também passa o CPU para modo utilizador e activa um temporizador com um intervalo de tempo (Time-slice, da ordem de 20 a 50 milisegundos), que permitirá ao SO monitorizar o tempo de execução do processo.
-- Durante o estado Executando, o processo executa as instruções do programa.
-- Executando -> Terminado: o processo chega ao fim do programa, nesse caso encontrando uma instrucao que chama a funcao exit(), a chamada ao SO que desencadeia a terminacao do processo; ou o processo termina de forma anormal, devido a um erro (instrucao ilegal em modo utilizador, violacao das proteccoes de memoria, etc) e nesse caso tambem termina mas com indicacao da condicao de erro.
-- Executando -> Bloqueado: o processo invocou uma chamada ao SO que não pode ser completada de imediato pelo que o SO decide colocar o processo num estado de bloqueio lógico até que as condições para completar aquela chamada ao SO se verifiquem e o processo possa continuar a execução. Diversas chamadas ao SO podem desencadear o bloqueio de um processo, havendo entao possiveis causas diferentes de bloqueio. Uma das mais frequentes ocorre quando um processo invoca a chamada ao SO read() e os buffers do SO associados ao correspondente canal (file descriptor, aberto no qual read foi invocada), está correntemente vazios. Por cada causa de bloqueio o SO marca-a, no descritor do processo na TabelaGlobal de processos, e coloca também informação para saber quando essa condição se deixar de verificar. Por exemplo, ao fazer uma operacao de read que obrigue o acesso aos blocos de um ficheiro em disco, havera indicacao que ligue o bloqueio aos buffers onde o device driver do disco irá mais tarde colocar a informacao (blocos) lida de disco, para que o SO saiba, nessa altura, que deve desbloquear o processo. Para implementar o bloqueio de um processo, o SO precisa de salvaguardar, no descritor do processo na TabelaGlobal de processos, todos os valores dos registadores do CPU, correspondentes ao ponto da execucao corrente do processo, quando se bloqueia, para que mais tarde, o SO possa retomar a execucao daquele processo a partir do ponto onde ia.
-- Note que, sempre um processo passa de Executando → Bloqueado, o CPU fica disponível para seleccionar outro processo para execução, pelo que irá seleccionar um dos processos que esteja à espera de vez na fila de processos prontos. Assim, uma transicao de estado Executando -> Bloqueado desencadeia sempre uma outra transicao de outro processo de Pronto -> Executando. Note que o mesmo sucede quando Executando -> Terminacao um processo termina a execucao, pois aqui o CPU tambem fica livre.

--Bloqueado -> Pronto: esta transicao ocorre logo que a condicao (por exemplo completou-se uma leitura de disco para os buffers onde uma operacao read há-de ir ler os bytes) de que um processo esperava, no estado bloqueado, fica satisfeita. Assim, o processo que estava bloqueado passa de novo ao estado Pronto, voltando o seu identificador a ser colocado na fila de processos prontos, e o SO vai agora reconsiderar qual o processo mais prioritario que deve ser seleccionado para execucao. Assim, se o processo que acabou de ser desbloqueado for o mais prioritario neste momento no sistema, entao passara de imediato ao estado executando, ou seja verifica-se nesse caso duas transicoes seguidas: Bloqueado -> Pronto -> Executando. E o processo que tinha estado em execucao durante o tempo em que o outro estava bloqueado, terá de voltar ao estado Pronto (passando de Executando -> Pronto) por ter menos prioridade do que o outro. Diz-se, neste caso, que o CPU foi apreendido (à força, o termo em inglês, preemption) ao processo que estava em execução. Se, pelo contrário, o processo correntemente em execução continua a ser o mais prioritário no sistema, quando comparado com o que foi desbloqueado, então este último ficará simplesmente na fila Prontos, à espera de vez.
-- Executando -> Pronto: esta transição já vimos que pode ocorrer na situação vista acima. Mas também pode ocorrer quando um processo executa um programa numa fase de tipo CPU-bound, correspondente a ser dominado por instrucoes aritmeticas e logicas e de referencia de memoria, tendo portanto tendencia a ocupar o CPU (e a nao fazer chamadas ao SO que o pudessem bloquear) durante todo o Time-slice que lhe foi atribuído ao ser activado (veja acima, a transicao Pronto -> Executando). Neste caso, o temporizador activado nessa altura, chega ao fim do tempo, gera um pedido de interrupçao ao CPU, e a rotina de tratamento dessa interrupcao (que faz parte do codigo do SO), indica o fim do Time-slice, pelo que o SO irá apreender o CPU a esse processo, passando de Executando -> Pronto, de modo a dar a vez a outro processo. Este é um mecanismo que se destina a evitar que uma aplicacao de tipo CPU-bound monopolizasse todo o tempo de CPU, não dando às outras oportundade de execução. Note que, tal como na situação anterior, a transicao Executando -> Pronto do processo que expirou o Time-slice, desencadeia uma outra transicao Pronto -> Executando, de um dos outros processos da fila de processos prontos. Note também que o processo que expirou o seu Time-slice, irá aguardar uma nova oportunidade, quando se tornar de novo o mais prioritario de entre os que estao na fila de processos prontos. Assim, um processo CPU-bound irá sofrer possivelmente diversas comutações de contexto deste tipo, antes de conseguir terminar, mas é um preco que tem de pagar por haver múltiplas aplicações em regime de multiprogramação, partilhando o mesmo CPU. É uma questão de equilibrio ou atendimento justo entre os multiplos processos. Claro que a aplicaçao CPU-bound se completaria mais depressa se estivesse sozinha em execucao no computador...

b) Para cada uma das chamadas ao sistema Unix indicadas em **b1)** e **b2),** descreva os seus argumentos e os resultados da sua invocação, diga quais as acções internas que desencadeia a nível hardware e SO e quais as transições de estados desencadeadas no diagrama que apresentou na alínea **a)**.
b1) *wait*
b2) *fork*

-> A chamada wait nao vem para o 1º teste.
-> A chamada fork está descrita nos acetatos / apontamentos.

Questão 5. Considere a execução concorrente de um programa (P1) com comportamento *CPU-bound* e de outro programa (P2) com comportamento *IO-bound*, num sistema Unix para um computador que só tem um *CPU*.
a) Explique as características do comportamento *IO-bound*
b) Explique as características do comportamento *CPU-bound*
c) Num SO como o Unix, em que pontos da sua execução pode um programa perder o controlo do CPU? Para cada caso explique as correspondentes acções desencadeadas pelo hardware e/ou pelo sistema de operação.

-> a) e b) Foram mencionadas acima. Veja as caracteristicas tambem nos acetatos/apontamentos.
-> c) No Unix um processo pode perder o controlo do CPU quando:
--- termina (normalmente ou por erro)
--- se bloqueia explicitamente porque invocou uma chamada ao SO que não se pode bloquear no momento (por exmplo um read quando buffers estao vazios) e que tem uma semântica bloqueante
-- expira o intervalo de tempo do seu Time-slice
-- um outro processo que estava Bloqueado, foi passado a Pronto (porque a sua condição de bloqueio foi satisfeita) e quando esse processo tiver maior prioridade do que o processo em execucao.

Questão 6. Explique detalhadamente o significado dos parâmetros e as acções efectuadas por cada uma das seguintes chamadas ao Unix, explicando o seu efeito sobre as estruturas relevantes, internas ao sistema Unix:
a) *close* (e o seu efeito nas tabelas de canais de ficheiros e de i-nodes)
b) *fork* (e o seu efeito no mapa de memória e na tabela de canais dos processos envolvidos)

-> Veja acetatos/apontamentos

Questão 7. Utilizando chamadas ao sistema Unix, pretende-se implementar a seguinte função
int get_characters(char *destino, int nc);
…
-> A solucao baseia-se nos mesmos principios que os de uma funcao put_character como estudado no 1o trabalho, mas agora para o caso da entrada.
Tambem, no caso do get, se aplicam os mesmos principios que descritos nos acetatos. Neste caso, sempre que houver bytes suficientes no buffer do programa, basta devolver esses bytes no vector argumento da funcao e actualizar simplesmente o contador de bytes do buffer e o apontador para o buffer. Se nao forem suficientes, basta ler directamente pedindo ao SO através de uma chamada ao SO read() que encha o buffer de novo, e depois fazer como anteriormente.
No inicio, a funcao get tem de chamar open() a 1ª vez, para registar o numero de canal (file descriptor) para o ficheiro, que será sempre utilizado a partir daí, sempre que tiver de fazer read().
No fim, uma vez obtido 0 como retorno de read() indica que chegou ao fim do ficheiro, pelo que isto deve ser registado pois chamadas sucessivas de get a partir daí deverão retornar sempre 0.

Questão 8. a) Defina os principais parâmetros que caracterizam a eficiência de um Sistema de Operação.
b) No caso de um sistema de multiprogramação com um regime *batch* de trabalhos não interactivos, explique quais as técnicas utilizadas pelo Sistema de Operação, para melhorar cada um dos parâmetros referidos em **a).**
c) No caso de um sistema de multiprogramação com um regime de múltiplos utilizadores interactivos, explique quais as técnicas utilizadas pelo Sistema de Operação, para melhorar cada um dos parâmetros referidos em **a).**

-> a) Veja os acetatos / apontamentos.
(Util) Taxa de utilização dos recursos: Tempo utilização / tempo total
(Deb) Débito de trabalhos: Trabalhos executados / unidade de tempo
(Resp) Tempo de resposta para o utilizador: Tempo que decorre desde a submissão do trabalho até obter os resultados isto é, Tempo_resposta – Tempo_submissao

-> b) A multiprogramação melhora Util do CPU pois aproveita os tempos em que um processo tem de bloquear-se e utiliza o CPU durante esse tempo para executar outro processo. Também melhora Util dos periféricos pois,

havendo mais aplicações activas, há mais pedidos de operações de I/O, o que contribui para manter os periféricos melhor ocupados. Em monoprogramação, só uma aplicação pode fazer pedidos de I/O, pelo que haverá tendência em manter os periféricos menos bem utilizados. Como os periféricos são mais lentos do que o CPU, convém indo mantê-los ocupados e, com o controlo dos mecanismos de interrupções, ir transferindo os bytes de/para buffers em memória, por forma a também reduzir os tempos de espera dos programas quando estes precisarem dos bytes ou de os enviar.
Como se vê, a actividade de multiprogramação, com o recurso aos mecanismos de interrupções e gestão de buffers pode contribuir para aumentar Util dos CPU e dos periféricos. Indirectamente contribuirá para melhorar Resp, pois o uso de buffers conduzirá a menores tempos de espera dos processos, pelo reduzirá o seu tempo de resposta.

Do mesmo modo, a utilização da técnica de SPOOL contribui para melhorar Util. A maior autonomia dos controladores dos periféricos, para além do mecanismo de interrupções, através do uso de DMA (Direct-Memory-Access) e de processadores dedicados ao controlo dos periféricos, contribui para reduzir o tempo em que o CPU tem de executar programas de controlo dos periféricos (e também reduz o número de interrupções feitas ao CPU), permitindo que os programas em execução pelo CPU sejam menos atrasados.

Globalmente o Tempo de resposta de um processo depende de diversos factores tais como:
- o número e tipo de instruções máquina do programa, bem como o tipo do programa, ou seja de que forma a execução do programa se reparte entre fases CPU bound e IO bound
- o tempo que o processo tem de esperar pelo completar de operações de I/O, através de chamadas ao SO bloqueantes
- o tempo em que o processo tem de esperar pela sua vez para passar do estado Pronto para Executando, o que depende da sua prioridade, depende também do conjunto de outros processos que sejam multiprogramados com aquele e que assim competem pelo CPU , e depende ainda das estratégias de escalonamento que o SO utiliza para seleccionar e activar os processos para execução.

O débito dos trabalhos está também indirectamente relacionado com os factores anteriores. Em particular num sistema que processa os trabalhos em batch de forma não interactiva, basicamente para cada processo o SO deve activar uma sequência de comandos (por exemplo a partir de um ficheiro de comandos, dito um guião ou script). Se na fila prontos houver processos de distintas características, o débito dos trabalhos poderá aumentar se o SO der prioridade aos trabalhos que se completem mais rapidamente em detrimento dos que sejam mais longos.

Em resumo, as técnicas do SO que podem influenciar aqueles factores de eficiência são: a multiprogramação, a gestão de buffers e o mecanismo de interrupções, a técnica de SPOOL, a gestão das prioridades dos processos em função do seu perfil de execução e as estratégias de escalonamento de processos para execução. A boa combinação destas técnicas para aumentar simultaneamente os três factores de eficiência indicados, é difícil de conseguir pois algumas soluções melhoram um factor e pioram outro, conforme o tipo de processo. Por exemplo, como se disse, aumentar a prioridade dos trabalhos mais rápidos contribui para melhorar o débito de trabalhos, pois o SO processa mais trabalhos por unidade de tempo, mas pode contribuir para aumentar o tempo de resposta dos trabalhos mais longos, uma vez que têm de esperar pelos mais curtos. E o resultado global dependerá de qual a mistura corrente de trabalhos de cada daqueles dois tipos, num dado momento, sendo isso que afectará os valores médios daqueles factores de eficiência.

-> c) Num sistema de multiprogramação interactivo em que o utilizador interage através de um terminal, geralmente as mesmas técnicas conforme acima referido, são aplicadas. Em particular, o SO monitoriza as fases de execução de cada processo, entre CPU-bound e IO-bound, por forma a ajustar dinamicamente as prioridades, por exemplo elevando a prioridade de um processo que nos momentos recentes tem efectuado mais operacoes de I/O. Pois que este processo tenderá a estar mais tempo bloqueado à espera de IO, libertando assim o CPU para outros processos (e contribuindo para melhorar o débito de trabalhos e a utilização do CPU).

Estes problemas serão abordados mais adiante, no capítulo final desta disciplina. Por agora é suficiente compreender saber quais são os prinicipais factores de eficiência e quais as técnicas que o SO pode utilizar para os tentar melhorar.

Questão 9. Utilizando chamadas ao sistema Unix, pretende-se implementar uma função, em C:
int put_*characters(char *origem, int nc);*
-> Veja os comentários feitos para a Questão 7, acima.

Questão 10.
a) Apresente, em pseudo-código, todas as acções desencadeadas quando um programa executado em modo utilizador, invoca uma instrução para fazer a chamada ao sistema de operação *fork()* no Unix. Deve indicar quais as acções suportadas automaticamente pelo *hardware* do computador e quais as acções suportadas pelo SO e o seu efeito nas estruturas relevantes do núcleo do sistema.
b) Apresente, em pseudo-código, todas as acções desencadeadas (seja por *hardware,* seja pelo SO), num sistema de multiprogramação, para efecutar a comutação de contextos entre dois processos P1 e P2, no caso em que o processo P1, quando em execução, atinge o limite do seu *Time-slice*, e sendo P2 o novo processo que o SO escolherá para execução. Desenhe, no diagrama de estados dos processos, quais as transições de estados sofridas por P1 e P2 neste caso.

-> Veja os acetatos/apontamentos e as sugestões anteriores, sobre as transições de estados dos processos (Questoes 4 e 5).