Gestão do Processador

1. A modelação do ambiente de multiprogramação através da activação e desactivação de um conjunto de processadores virtuais, cada um deles associado a um processo particular, supõe que dois factos essenciais relativos ao comportamento dos processos envolvidos sejam garantidos. Quais são eles?

É preciso garantir que a execução dos processos não é afectada pelo instante, ou local no código onde ocorre a comutação. É também preciso garantir que não são impostas quaisquer restrições relativamente aos tempos de execução, totais ou parciais, dos processos.

2. Qual é a importância da tabela de controlo de processos (*PCT*) na operacionalização de um ambiente de multiprogramação? Que tipo de campos devem existir em cada entrada da tabela?

Esta tabela é responsável por manter a ordem dos processos e por guardar toda a informação relativa a cada processo. É usada intensivamente pelo scheduler para fazer gestão do processador e de outros recursos.

Entradas da tabela:

- identificador do processo, do processo pai e do utilizador
- localização na memoria (RAM/SWAP) de acordo com a organização da mesma
- contexto que guarda todos os registos internos no momento em que se deu a comutação do processo
- contexto I/O que guarda a informação sobre os buffers e canais de comunicação
- informação sobre o estado e de scheduling de acordo com o diagrama de estados.

3. O que é o *scheduling* do processador? Que critérios devem ser satisfeitos pelos algoritmos que o põem em prática? Quais são os mais importantes num sistema multiutilizador de uso geral, num sistema de tipo *batch* e num sistema de tempo real?

Scheduling é a parte integrante do núcleo central e é responsável pelo tratamento de interrupções e por agendar a atribuição do processador e de muitos outros recursos do sistema computacional. Os algoritmos que põem em pratica este sistema devem ser justos, previsíveis, ter um alto throughput, minimizar o tempo de resposta, minimizar o tempo de turnaround, respeitar ao máximo as deadlines, eficientes.

Num sistema de multiutilizador de uso geral, o mais importante é a justiça, pois todos os utilizadores têm que ter direito ao mesmo tempo de processamento. Num sistema do tipo batch é reduzir é reduzir os tempos de turnaround dos jobs que constituem a fila de processamento. Por fim num sistema de tempo real é muito importante a previsibilidade pois pode ser necessário tratamento de processos com condições de alarme ou de acções cuja execução efectiva tem que ser mantida dentro de intervalos temporais bem definidos.

4. Descreva o diagrama de estados do *scheduling* do processador em três níveis. Qual é o papel desempenhado por cada nível? Num sistema de tipo *batch* multiprogramado fará sentido a existência de três níveis de *scheduling*?

O nível do processador (baixo nível) permite fazer uma melhor gestão do processador de modo a que todos os processos sejam atendidos ao mesmo tempo. O nível de memória principal (nível médio) permite fazer a gestão de memória para cada processo de modo a permitir mais processos em memória. Por ultimo, o nível de ambiente de multiprogramação (alto nível) é responsável por criar os processos e terminá-los caso o batch seja muito complexo, isto é, se invocar muitos processos.

Num sistema do tipo batch multiprogramado faz de facto sentido a existência de três níveis de scheduling para garantir um equilíbrio adequado entre o serviço aos processos que coexistem e a ocupação do processador.

5. Os estados *READY-TO-RUN* e *BLOCKED*, entre outros, têm associadas filas de espera de processos que se encontram nesses estados. Conceptualmente, porém, existe apenas uma fila de espera associada ao estado *READY-TO-RUN*, mas filas de espera múltiplas associadas ao estado *BLOCKED.* Em princípio, uma por cada dispositivo ou recurso. Porque é que é assim?

Isto acontece porque um processo só abandona o estado BLOCKED quando ocorre um acontecimento externo (que pode ser o acesso a um recurso, complemento de uma operação de entrada/saída, etc). Assim, são necessárias varias filas de espera para se saber qual foi o acontecimento externo que causou a transição do estado BLOCKED para o READY-TO-RUN.

6. Indique quais são as funções principais desempenhadas pelo *kernel* de um sistema de operação. Neste sentido, explique porque é que a sua operação pode ser considerada como um *serviço de excepções*.

As funções principais do kernel são: gestão do processador, gestão de memória e gestão do ambiente de multiprogramação. Podemos considerar a sua operação como um serviço a excepção pois toda a comutação de processos pode ser vista como uma rotina de serviço à excepção, com uma pequena alteração: normalmente a instrução que vai ser executada após o serviço da excepção, é diferente daquela cujo endereço foi salvaguardado antes de se iniciar o processamento da excepção.

7. O que é uma *comutação de contexto*? Descreva detalhadamente as operações mais importantes que são realizadas quando há uma comutação de contexto.

Uma comutação de contexto é no fundo uma comutação de processos. As operações mais importantes quando é realizada uma comutação de contexto são: salvaguarda do PC e do PSW na stack do sistema; carga no PC do endereço de rotina de serviço à excepção; salvaguarda do conteúdo dos registos internos do processador que vão ser usados; processamento da excepção; restauro do conteúdo dos registos internos do processador que foram usados; restauro do PSW e do PC a partir da stack do sistema.

8. Classifique os critérios que devem ser satisfeitos pelos algoritmos de *scheduling* segundo as perspectivas sistémica e comportamental, e respectivas subclasses. Justifique devidamente as suas opções.

Scheduling sob a perspectiva sistémica deve satisfazer critérios orientados aos utilizadores, pois estão relacionados com o comportamento do sistema de operação na perspectiva dos processos e dos utilizadores; deve também satisfazer critérios orientados ao sistema, pois estes estão relacionados com o uso eficiente dos recursos do sistema de computação. Já o scheduling sob a perspectiva comportamental deve satisfazer critérios orientados ao desempenho e mais alguns tipos de critérios.

9. Distinga disciplinas de prioridade estática das de prioridade dinâmica. Dê exemplos de cada uma delas.

Disciplinas de prioridade estática são aquelas que o seu método de definição é determinístico (como o utilizado em sistema de tempo real). Prioridades dinâmicas são aquelas que o seu método de definição depende da historia passada de execução do processo (como o usado em sistemas operacionais interactivos).

10. Num sistema de operação multiutilizador de uso geral, há razões diversas que conduzem ao estabelecimento de diferentes classes de processos com direitos de acesso ao processador diferenciados. Explique porquê.

Para evitar que haja um adiamento indefinido na calendarização de execução de um processo.

11. Entre as políticas de *scheduling preemptive* e *non-preemptive*, ou uma combinação das duas, qual delas escolheria para um sistema de tempo real? Justifique claramente as razões da sua opção.

Para um sistema de tempo real, uma politica de scheduling preemptive é mais adequada, porque num sistema de tempo real existe a necessidade de fazer constantes comutações de processos sempre que se executa um processo mais prioritário.

12. Foi referido nas aulas que os sistemas de operação de tipo *batch* usam principalmente uma política de *scheduling non-preemptive*. Será, porém, uma política pura, ou admite excepções?

É uma política pura pois este ordena os jobs por tempo (sendo o que irá ocupar menos tempo o primeiro a ser executado), atribuindo-lhes o processador sem nunca o retirar do processo que no momento o detém, tendo que aguardar que o processador seja libertado para o processo seguinte quando o anterior for executado até ao fim.

13. Justifique claramente se a disciplina de *scheduling* usada em Linux para a classe *SCHED_OTHER* é uma política de prioridade estática ou dinâmica?

A classe SCHED_OTHER usa uma disciplina de scheduling dinâmica, pois o algoritmo de scheduling combina a historia passada de execução do processo e a sua prioridade, e maximiza o tempo de resposta dos processos I/O - intensivos sem produzir adiantamento indefinido para os processos CPU – intensivos. O algoritmo usado por esta classe baseia-se em créditos.

14. O que é o *aging* dos processos? Dê exemplos de duas disciplinas de *scheduling* com esta característica, mostrando como ela é implementada em cada caso.

Aging de processos é a alteração da prioridade dos processos tendo em conta o seu "tempo de vida". O algoritmo de total fairness vai utilizar o processo de aging pois à medida que um processo espera o seu tempo de espera vai sendo incrementado e sempre que ele é calendarizado para execução o seu tempo de espera é decrementado sendo depois executado o processo com a maior diferença entre tempo de espera e tempo de execução virtual. Outro algoritmo que utiliza o processo de aging é o algoritmo de calculo de prioridade dinâmica que procura estimar a fracção da ocupação da janela de execução seguinte em termos de ocupação de janelas passadas, atribuindo se o processador para o processo com menor estimativa.

15. Distinga *threads* de processos. Assumindo que pretende desenvolver uma aplicação concorrente usando um dos paradigmas, descreva o modo como cada um afecta o desenho da arquitectura dos programas associados.

Um processo dedica-se a agrupar um conjunto de recursos enquanto as threads constituem as entidades executáveis independentes mesmo dentro do contexto do processo. Ao escolher uma aplicação usando o paradigma de threads, ao envolver menos recursos por parte do sistema de operação, possibilita que operações como a sua criação e destruição e mudança de contexto, se tornem menos pesadas e portanto menos eficientes; alem disso em multiprocessadores simétricos torna-se possível calendarizar para execução em paralelo múltiplos threads da mesma aplicação, aumentando assim a sua velocidade de execução.

16. Indique, justificadamente, em que situações, um ambiente *multithreaded,* pode ser vantajoso.

O uso de um ambiente multithreaded é vantajoso quando vamos ter programas que envolvem múltiplas actividades e atendimentos a múltiplas solicitações; quando existe uma partilha do espaço de endereçamento e do contexto de I/O entre as threads em que uma aplicação é dividida, torna-se mais simples a gestão de memória principal e o acesso aos dispositivos de entrada e saída de uma maneira eficaz

17. Que tipo de alternativas pode o sistema de operação fornecer à implementação de um ambiente *multi-threaded*? Em que condições é que num multiprocessador simétrico os diferentes *threads* de uma mesma aplicação podem ser executados em paralelo?

Em alternativa à implementação de um ambiente multithreaded podemos apenas ter no nosso sistema computacional threads kernel visto que estas podem ser executadas paralelamente num sistema multiprocessador. Um multiprocessador simétrico possibilita a calendarização de várias threads da mesma aplicação para execução paralela o que é possível se o sistema computacional permitir multithreading.

18. Explique como é que as *threads* são implementadas em Linux.

Em Linux existe o comando fork que cria um novo processo a partir de um já existente por copia integral do seu contexto alargado, existe também o clone, que cria um novo processo a partir de um já existente por copia apenas do seu contexto restrito, partilhando o espaço de endereçamento e o contexto de I/O e iniciando a sua execução pela invocação de uma função que é passada como parâmetro. Desta forma, não há distinção efectiva entre processos e threads, que o Linux designa por tasks, e elas são tratadas pelo kernel da mesma maneira.

19. O principal problema da implementação de *threads*, a partir de uma biblioteca que fornece primitivas para a sua criação, gestão e *scheduling* no nível utilizador, é que quando uma *thread* particular executa uma *chamada ao sistema* bloqueante, todo o processo é bloqueado, mesmo que existam *threads* que estão prontas a serem executados. Será que este problema não pode ser minimizado?

Este problema pode ser minimizado implementando threads de kernel. Estas threads são implementadas directamente ao nível kernel que providencia as operações de criação, gestão e scheduling de threads. Esta implementação é menos eficiente que no caso das threads de utilizador, mas o bloqueio de uma thread particular não afecta a calendarização para a execução das restantes threads.

20. Num ambiente *multithreaded* em que as *threads* são implementadas no nível utilizador, vai haver um par de *stacks* (sistema / utilizador) por *thread*, ou um único par comum a todo o processo? E se os *threads* forem implementados ao nível do *kernel*? Justifique.

Num ambiente em que as threads são implementadas no nível do utilizador vai existir um par de stacks comum a todo o processo visto que threads ao nível utilizador não suportam vistas pelo kernel como uma única thread, tendo assim um só par porque são tratadas como uma única thread. Já nas threads implementadas ao nível do kernel vamos ter uma stack por thread, pois como as threads são lançadas pelo kernel cada uma destas threads vai ter um par de stacks.