Sistemas Operativos — Gestão de Processos [1]

Francisco Soares de Moura
Universidade do Minho/INESC,
Departamento de Informática,
Campus de Gualtar,
4700 Braga,
PORTUGAL
fsm@di.uminho.pt

Outubro 1995

Contents

Chapter 2

Gestão de Processos

Resumo

Este documento contém uma versão de trabalho do capítulo sobre a *Gestão de Processos* da futura sebenta de **Sistemas Operativos**. Destina-se a apoiar as disciplinas de *Sistemas Operativos* (3º ano da LESI e LMCC), e *Sistemas Operativos, Compiladores e Ambientes de Execução* (3º ano LIG) e *Opção* (5º ano LEI). Embora este capítulo esteja parcialmente imcompleto, a parte em falta encontra-se em qualquer um dos livros recomendados. Não é permitida a utilização e/ou reprodução deste texto para outros fins.

2.1 Introdução

De um modo geral, um processo representa qualquer actividade em execução num sistema, tenha ela sido criada pelo sistema operativo ou pelos utilizadores. Estas actividades competem pelo CPU e portanto surge a necessidade de fazer o seu escalonamento (*process scheduling*). Neste capítulo discutem-se os critérios e estratégias de seriação dos processos para efeito de atribuição de tempo de CPU. No entanto, é de salientar que os conceitos aqui apresentados são perfeitamente genéricos. São válidos no caso de **competição por recursos**, sejam estes informáticos (CPU, impressora, disco, registo de base de dados, etc.) ou mesmo não informáticos (alocação de tarefas a uma máquina ferramenta, escolha do primeiro ministro, etc.).

São várias as razões que determinam a criação de actividades concorrentes. A primeira é logo por uma questão de estruturação. Há inúmeros casos em que um algoritmo se exprime facilmente à custa da composição (concorrente) de várias actividades sequenciais: por exemplo, num sistema de gestão de stocks em ambiente multi-utilizador seria impensável desenvolver um programa que fizesse o *polling* dos vários terminais e tivesse de manter o estado de cada um. É muito mais naturar encarar cada terminal como uma actividade separada, que nada tem a ver com as outras excepto quando acede a recursos comuns. Ou seja, primeiro desenvolvem-se isoladamente as várias actividades (até aqui supostamente sequenciais), e depois introduz-se o controlo de concorrência se for caso disso.

Outro exemplo clássico é a construção da interface com o utilizador num sistema de janelas. É conveniente associar um processo a cada janela, pois deste modo o utilizador pode executar várias tarefas em "simultâneo" e comutar entre elas conforme entender. No entanto, cada uma das tarefas continua a ser sequencial — se a meio da leitura do correio electrónico tiver necessidade de abrir outra janela e executar novo comando, se regressar ainda estará no mesmo ponto.

De forma análoga, num sistema distribuído é altamente desejável que os servidores possam dedicar uma tarefa a cada sessão associada a um cliente remoto. A ideia é sempre a mesma: manter o modelo síncrono, a invocação sequencial de programas/procedimentos a que os programadores estão habituados, mas sem que estes sintam "remorsos" ao invocarem uma operação demorada que vai bloquear todo o serviço. Lá por baixo o sistema operativo se encarrega de atribuir o CPU a outra tarefa desse servidor.

No outro extremo, os sistemas operativos utilizam concorrência ao efectuarem pedidos de I/O sobre periféricos lentos: enquanto que uma tarefa se bloqueia à espera do fim do pedido, outras tarefas podem utilizar o processador. Isto acontece a todos os níveis, desde os *device drivers* até aos *spoolers* de impressoras e servidores de ficheiros remotos. Os próprios utilizadores podem querer lançar em *background* operações demoradas, e prosseguir a tarefa principal.

Os utilizadores também podem criar vários processos numa tentativa de explorar a existência de multiplos CPUs, seja em sistemas de multiprocessamento[1] com memória partilhada, seja em multicomputadores de memória distribuída. Aqui já não se trata apenas de concorrência mas sim de paralelismo real. O desenvolvimento de algoritmos paralelos adequados a essas máquinas pertence à chamada Computação Paralela, que está fora do âmbito desta disciplina[2].

2.2 Objectivos a Atingir

A importância relativa de cada um é sempre um bocado subjectiva, mas mesmo assim podemos relacioná-los com os dois grandes objectivos de um sistema operativo:

Conveniência

- Justiça
- Reduzir Tempos de Resposta
- Previsibilidade
- ...

Eficiência

[1] Não confundir *multiprocessamento*, computadores com vários CPUs ($\Rightarrow$ hardware), com *múltiplos processos* ou sistemas operativos ditos *multitarefa* ($\Rightarrow$ $software$).

[2] No entanto, o Grupo de Sistemas Distribuídos tem estado envolvido em vários projectos de investigação nessa área, e possui mesmo máquinas paralelas de vários tipos. Os interessados são encorajados a contactar os docentes desta disciplina.

- *Throughput*
- Maximizar a utilização do CPU (e outros recursos)
- Favorecer processos "bem comportados"
- ...

Infelizmente, é frequente a falta de coincidência entre os pontos de vista de utilizadores e administradores. Há assim que encontrar um compromisso entre a conveniência desejada pelos utilizadores e a eficiência procurada pelos administradores.

2.3 Critérios de Escalonamento

Para se atingirem os objectivos enunciados anteriormente, as estratégias de escalonamento a incorporar no gestor de processos *scheduler* poderão ter em consideração diversos critérios. Em primeiro lugar, faz sentido olhar para características dos processos, nomeadamente

- tipo *I/O-bound* ou *CPU-bound*
- Interactivo ou *batch*
- Urgência de resposta (eg. tempo real)
- Comportamento anterior (eg. memória virtual, *burst* CPU)
- Necessidade de periféricos especiais (eg. banda magnética)
- etc...

Note que as características *anunciadas* pelos utilizadores na altura da criação dos processos poderão não ser de confiança. Um exemplo típico é a submissão (inadvertida ou deliberada) de *jobs* longos para a fila dos processos interactivos. Mesmo estes podem entrar subitamente numa fase diferente (eg. uma pesquisa mais demorada numa base de dados, uma ordenação de ficheiros, a *garbage collection* num interpretador de Lisp, etc.).

Isto aponta para a necessidade de classificação dinâmica dos processos, se possível à custa de mecanismos que valorizem quer o passado recente de cada processo ($\Rightarrow$ capacidade de reacção rápida) quer o comportamento "médio" desses mesmos processos. Por exemplo, um editor de texto como o `emacs` é um exemplo de um programa altamente interactivo mas que ocasionalmente tem *bursts* de CPU longos. Não é por isso que ele deve ser considerado CPU-bound.

Além disso, as estratégias de escalonamento deverão ter em atenção outros critérios que tem apenas a ver com o próprio sistema, e não com as características dos processos, incluindo:

- Prioridades adquiridas (eg. pagas!)
- Serviço recebido até ao momento (eg. tempo de CPU acumulado, se foi ou não "ultrapassado por outros processos")

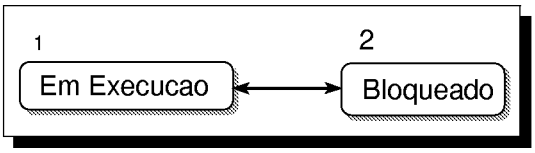

Figure 2.1: Diagrama de Estados Simplificado

- Contribuição para os objectivos (eficiência, tempos de resposta, etc.)
- etc...

2.4 Principais Decisões

Num sentido lato, um processo encontra-se "em execução" desde o instante em que foi criado até que termina. No entanto, durante este tempo de vida o processo acaba por passar por dois estados fundamentais, o da execução propriamente dita, e um outro estado em que o processo não pode prosseguir a sua actividade porque está à espera de um determinado evento.

Conforme indicado na Figura 2.1, um processo que foi escalonado pelo sistema operativo encontra-se no estado de execução. A transição inversa ($1 \rightarrow 2$) pode ser causada por chamadas ao sistema *bloqueantes* (por exemplo leitura de buffer vazio, ou pausa durante x segundos), ou por acontecimentos relacionadas com o processo (típicamente falta de página que tem de ser carregada do disco).

Num sistema monoprogramado que suporte apenas um utilizador, o programa pode ficar em espera activa (*polling*) a testar continuamente se o evento pretendido já ocorreu. Não há qualquer problema nisso, como só existe um programa no sistema, não há mais nada a fazer. É o que acontece por exemplo em MS-DOS[3].

Num sistema multiprogramado, a espera activa é inaceitável pois não existem tantos CPUs quantos os processos (não bloqueados). Por isso, sempre que um processo se bloqueia o CPU é-lhe retirado e atribuído a outro processo que esteja "pronto a executar" (Figura 2.2). A transição $2 \rightarrow 3$ dá-se quando surge o evento aguardado pelo processo (fim de transferência de disco, etc.). O **escalonamento de processos** diz então respeito ao conjunto de decisões relacionadas com a transição $3 \rightarrow 1$, e eventualmente $1 \rightarrow 3$.

Em princípio, a decisão mais importante é a escolha do próximo processo a executar, seleccionado entre os que estão no estado 3. Logo à partida esta é uma decisão difícil, pois, havendo uma lista de espera, escolher um significa *atrasar os outros...*! É necessário muito cuidado com a definição das "prioridades"!

No entanto, não basta escolher o próximo processo. Há também que definir QUANDO é que essa decisão entra em vigor. Por exemplo, suponha que um processo de alta prioridade se bloqueia à espera de uma transferência de disco, e que o sistema operativo comuta para um processo de baixa prioridade. Quando

[3]Na prática não bem assim, pois os programas `TSR` (*Terminate and Stay Resident*) também competem por tempo de CPU.

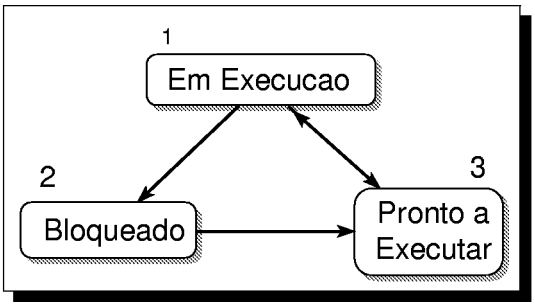

Figure 2.2: Diagrama de Estados Revisto

a transferência terminar, deverá comutar novamente para o processo anterior ($\Rightarrow$ *overhead*, possível interferência com a gestão de memória, etc.), ou deverá deixar ficar assim?

Além disso, há que considerar que a escolha do processo a executar é tomada num dado instante, possivelmente com base no contexto existente naquele instante, e talvez no comportamento anterior do processo, ou naquilo que o utilizador disse (eg. é processo interactivo, portanto prioritário).

Mas, ... será que esse contexto e comportamento se mantém? Dito de outra forma, QUANTO tempo vai o processo executar? Tendo escolhido um processo, terei ou não possibilidade de reconsiderar a minha escolha se as condições se alterarem significativamente? E se o processo "parecia" ser interactivo e afinal mostra ser *CPU-bound*? Compare com a escolha de um governo ...

Em resumo, escalonamento tem a ver com decisões sobre:

- QUAL o próximo processo a executar
- QUANDO começa a executar
- Durante QUANTO tempo executa

A primeira decisão implica a noção de prioridade, as outras tem a ver com o facto de se usar ou não **desafectação forçada** (em inglês *preemption*). A decisão sobre utilização ou não de estratégias de escalonamento com desafectação forçada afecta particularmente os tempos de resposta sentidos pelos processos, razão pela qual este conceito será tratado na secção seguinte.

2.5 Desafectação Forçada

Uma estratégia de escalonamento sem desafectação forçada (*non-preemptive scheduling*) permite que um processo, uma vez escolhido, execute até que este liberte voluntariamente o CPU (transição $1 \rightarrow 2$ na figura 2.2) ou termine a execução. Pelo contrário, se se utilizar uma estratégia com desafectação, o processo em execução pode ser removido em qualquer altura (transição $1 \rightarrow 3$).

Apesar do *overhead* causado pela comutação de processos, a desafectação traz enormes vantagens quando a carga do sistema não é estável ou é mesmo

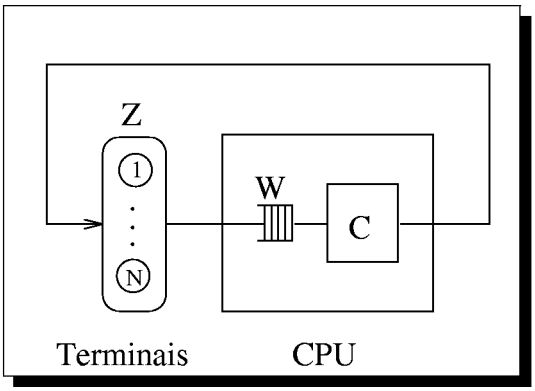

Figure 2.3: Modelo de Sistema Interactivo

desconhecida. Em particular, permite favorecer processos prioritários, designadamente os de tempo real ou altamente interactivos, melhorando assim os seus tempos de resposta, embora obviamente à custa dos outros...

Para facilidade de exposição, suponha-se em primeiro lugar que um sistema interactivo está sugeito a uma carga absolutamente homogénea: todos os processos seguem um padrão rigorosamente igual, consistindo num ciclo infinito em que pedem C unidades de tempo de CPU, e a seguir bloqueiam-se durante Z unidades de tempo. Este ciclo está representado na Figura 2.3, e pretende reproduzir o comportamento de um utilizador ao terminal: requisita CPU, pensa na interacção seguinte, submete novo pedido, etc. Será possível desenvolver um modelo matemático que nos forneça algumas indicações quanto aos tempos de resposta prováveis para um número variável de terminais? No caso geral, o tempo de resposta é dado por

$$R = W + C$$

onde W representa o *tempo de espera*, e C representa o *tempo de serviço*. Mas há casos particulares...

Quando o número de utilizadores (ie. terminais) é 1, o tempo de resposta R é obviamente C: o sistema está sempre vazio quando este pedido é submetido, não havendo portanto tempo de espera. No entanto, para 2 ou mais utilizadores, qualquer estimativa já se torna mais difícil pois tudo depende do *instante* em que chegarem os novos pedidos.

Se o segundo pedido não se sobrepuser ao primeiro (isto é, chegar enquanto o primeiro utilizador está a "pensar" — no tal intervalo de Z unidades de tempo), então o sistema já está novamente vazio e o tempo de resposta para $N = 2$ ainda é C. Este raciocínio pode ser generalizado para $N = 3, \ldots$, numa atitude assaz optimista (e arriscada), definindo assim a recta

$$R = C$$

ilustrada na Figura 2.4. Esta recta deve ser vista como assímptota inferior dos tempos de resposta.

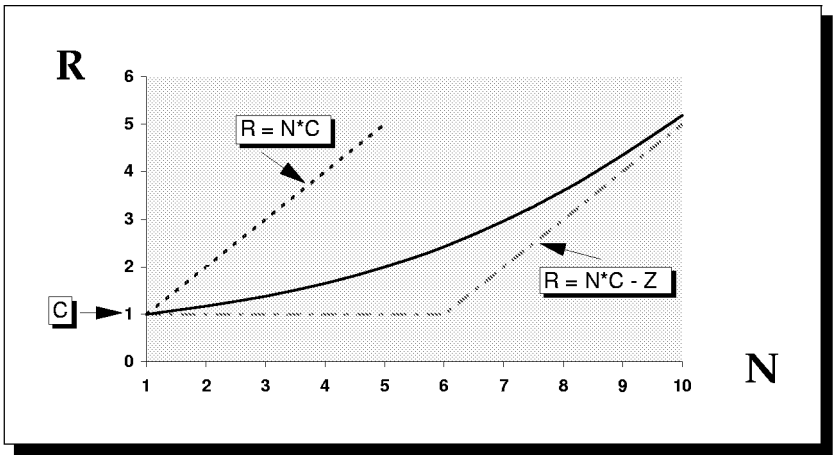

Figure 2.4: Tempo de Resposta: Carga homogénea

No entanto, há um ponto a partir do qual a *capacidade* do sistema se esgota e este fica *saturado*. Quando o número de terminais for tal que o intervalo de tempo Z fique totalmente preenchido pelos outros pedidos, é claro que, mesmo com todo o optimismo, há a certeza que se se acrescentar mais um terminal, o tempo de resposta vai subir para $2 * C$: o tempo de serviço C mais um tempo de espera C correspondente ao tempo de serviço do processo que ainda está a começar a ser executado quando o meu pedido é submetido. E sucessivamente, se acrescentar outro terminal, o tempo de resposta sentido por *todos* os processos passará para $3 * C$. Num sistema saturado, a assímptota dos tempos de resposta já é dada por

$$R = N * C - Z$$

conforme se pode ver na Figura 2.4, onde foi usado o valor de $Z = 5$.

Infelizmente, a hipótese de que os pedidos chegam perfeitamente des-sincronizados é demasiado irrealista, e assim a curva *provável* andará muito próxima das assímptotas quando o sistema está quase vazio ou muito carregado, com tendência para se afastar destas rectas para valores intermédios de N. O valor *exacto*[4], depende da forma como se "distribuem" no tempo os tempos de serviço C e a taxa de chegada de novos pedidos aos sistema (relacionado com Z). Aqui poder-se-ia entrar no terreno da Simulação e Estatística, nos processos estocásticos, distribuições de Poisson, hiper-exponencial, etc., em busca do valor exacto da curva. Como o objectivo deste capítulo é apenas o estudo de técnicas de escalonamento, sendo o modelo usado apenas para introdução dos aspectos essenciais de *scheduling*, remete-se o leitor interessado para a literatura correspondente.

Há ainda um caso particular em que poderemos estar interessados em obter tempos máximos de resposta, em vez dos tradicionais tempos médios. É nitida-

[4]Convém aqui alertar para o facto de ser possível construir um modelo matemático pouco credível, mas que produz resultados matematicamente correctos (exactos), existindo também modelos que reproduzem fielmente o sistema (exactos), mas são matematicamente impossíveis de resolver... Qual dos casos prefere?

mente o caso dos sistemas de *tempo real*, onde de nada vale afirmar que x por cento dos pedidos de CPU terão um tempo de resposta inferior a y. Para provocar uma catástrofe, basta que um alarme dispare e o sistema não actue num dado tempo *limite*. Nestes sistemas as contas são feitas segundo uma óptica "pessimista", admitindo-se que os pedidos podem surgir todos no mesmo instante. Isto conduz à recta

$$R = N * C$$

e o sistema é projectado jogando com as prioridades dos eventos a tratar[5].

Veja-se agora o impacto que uma carga heterogénea pode ter nos tempos de resposta. Suponhamos inicialmente que o *scheduler* seleciona processos para execução apenas com base na ordem na ordem de chegada, e que não há desafectação forçada (ie. temos uma estratégia de escalonamento FIFO). Uma vez selecionado um processo longo, não há forma de o lhe retirar o CPU. Os restantes processos, mesmo que necessitem de pouco tempo de CPU, ficam à espera.

Suponha-se ainda que uma em cada 10 interações requer um tempo de CPU também 10 vezes superior ao habitual ($10 * C$). Este modelo pretende reproduzir com maior realismo o comportamento normal de um utilizador interactivo: a maioria das interações é curta (eg. edições de texto, listagem de directoria, etc.), mas pontualmente surgem pedidos mais longos, sejam compilações, processamento de texto, etc..

Inadvertidamente poder-se-á pensar que uma interação longa no meio de 10 interações curtas não irá afectar muito os tempos de resposta[6]. No entanto, basta recordar que a inclinação da assímptota da saturação é dada por $N * C$. Agora tem-se um tempo de serviço *médio* igual a

$$(9 * C + 1 * (10 * C))/10 = 1.9 * C$$

que é quase o dobro do anterior.

Sem desafectação forçada, e uma vez que não se alterou o tempo Z, a assímptota continua a cruzar o eixo dos YY no ponto $-Z$, e a sua inclinação sobe para $1.9 * C$. Vê-se perfeitamente na Figura 2.5 que se anteriormente este sistema suportava 8 utilizadores com um tempo de resposta médio inferior a $3 * C$ (*stretch factor* de 3), agora já só suporta metade. Se se mantiverem os mesmos o postos de trabalho, o tempo de resposta será elevadíssimo. E só porque uma em cada 10 interações é longa!

Como sempre, há duas soluções para um problema de utilização de um recurso escasso: ou se aumenta a capacidade desse recurso, ou se melhora a gestão do recurso[7].

Neste caso, a solução errada consiste em trocar o CPU por um mais rápido (1.9 vezes?). Mantém-se os tempos de resposta baixos, mas em 90% por cento das interações o sistema está sub-utilizado $\Rightarrow$ não consegue tirar partido do seu

[5] Há várias estratgias de escalonamento vocacionadas para sistemas de tempo real: prioridades estáticas, *deadline scheduling*, etc..

[6] Os leitores com formação prática em Electrónica Analógica certamente recordarão a forma como se ignorava uma resistência em paralelo com outra 10 vezes inferior...

[7] Se possui automóvel, e acha que este não lhe dá o rendimento esperado, o que faz: compra um automóvel mais potente, ou... leva o seu a uma garagem para ser afinado?

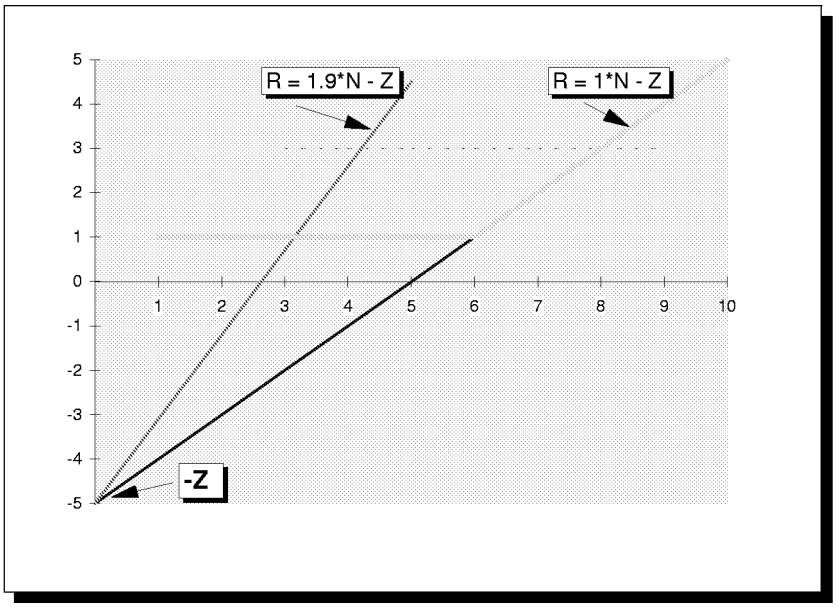

Figure 2.5: Tempo de Resposta: Carga heterogénea

investimento. A solução correcta baseia-se apenas em *software* (supostamente mais barato, para quem sabe). O que é preciso é uma estratégia de escalonamento que detecte interações longas e lhes retire o CPU sempre que absorvam C unidades de tempo. Assim, a assímptota regressa novamente à inclinação $1 * C$, e as interações curtas não são grandemente penalizadas pela intromissão de uma interação longa. Acabamos de descobrir a estratégia conhecida por *round-robin*, uma das mais simples, mas já com desafectação forçada.

2.6 Estratégias de Escalonamento

2.6.1 *First Come First Served*

2.6.2 *Round-Robin*

2.6.3 *Shortest Job Next*

2.6.4 *Shortest Remaining Time Next*

2.6.5 *Prioridade Estática*

2.6.6 *Multi-level Queues*

2.7 Níveis de Escalonamento

Até aqui estivemos a admitir que todos os processos no estado pronto a executar (*Ready*) são de facto candidatos à atribuição de CPU. Em qualquer das estratégias analisadas, desde que um processo não esteja bloqueado então em

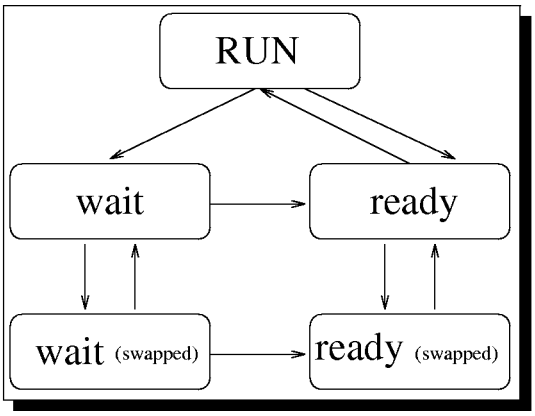

Figure 2.6: Diagrama de Estados com *Swap*

qualquer momento pode ser considerado o mais prioritário e passar ao estado de execução. Por outras palavras, até agora o nosso diagrama de estados só tem três estados (ver figura 2.2).

No entanto, um processo não compete apenas por CPU. Em particular, precisa de memória central para poder executar, e até já vimos na introdução que há sistemas operativos que "rejeitam" processos inteiros ou parte deles para a chamada Área de *Swap*. Apesar de só se tratar da gestão de memória virtual no capítulo seguinte, não custa a aceitar que atribuir o CPU a um processo que não está carregado em memória leva a que este se bloqueie imediatamente à espera de ser carregado[8]. O bom-senso diz-nos que seria preferível carregar primeiro o processo e só depois dar-lhe o CPU, conforme indicado na figura 2.6.

Para já isto levanta algumas questões: carregar um processo para memória é em si mesmo um acto de escalonamento. Houve um processo que foi considerado prioritário relativamente a outros também transferidos para a Área de *Swap*. Então à pouco dissemos que o escalonamento só devia ser feito depois do processo ser carregado e afinal agora sempre há escalonamento? Ou será que o *scheduler* ao ver que o processo eleito ainda está na Área de *Swap* inicia a sua transferência para memória central? À primeira vista isso também não parece muito inteligente pois seria o próprio *scheduler* a bloquear-se à espera do fim da transferência de disco, arrastando atrás de si os outros processos...

Este último aspecto da potencial paragem do *scheduler* resolve-se com a solução habitual para operações de I/O sobre periféricos lentos: concorrência. Se o *scheduler* "mandar"outro processo transferir do disco o processo inicial, então este poderá ficar bloqueado mas o *scheduler* mantém-se operacional. Então porque não instruir esse escravo com a política pretendida, que ele próprio toma a iniciativa de proceder às transferências? Ou seja, porque não promover esse segundo processo, e resolver assim o problema do escalonamento? Já agora este processo também podia fazer o escalonamento inverso, ie. transferir processos para a Área de *Swap* segundo algum critério.

[8] Se calhar até se bloqueia antes disso, pois primeiro é preciso arranjar espaço em memória central.

Esta análise começa a levar-nos para a conclusão de que devemos ter mais do que um *scheduler* no sistema operativo. Dito de outra forma, devemos ter dois (ou mais) níveis de escalonamento. O de baixo-nível, conhecido por *dispatcher*, atribui tempo de CPU aos processos que estão de facto prontos a executar. O de alto-nível prepara os processos para execução, transferindo-os se necessário para memória central, e entrega-os ao nível inferior para "despacho"[9].

Há outras razões que sugerem o escalonamento com mais do que um nível. Se só tivermos um único nível, então a função que escolhe o próximo processo pode ser extremamente pesada[10]. Terá de verificar se o processo está em disco ou em memória, assim como os restantes critérios de escalonamento (eg. prioridade externa, previsão da fatia de tempo, número de vezes que foi ultrapassado por outros e respectivo tempo de espera, etc. etc.. Veremos a seguir que os sistemas operativos seguem uma mistura de políticas que pode redundar numa função de avaliação muito complexa. Com dois ou mais níveis, o nível inferior mantém-se muito simples e os níveis superiores executam as funções pesadas, embora muito menos vezes (típicamente uma vez por segundo, ou menos).

Um exemplo de *scheduling* com três níveis poderia ser:

- Alto-nível (*job scheduler*) — responsável por admitir ou não um *job* para execução. Por exemplo, podia recusar o `login` a mais utilizadores no caso de detectar que o sistema está saturado (cf. figura 2.5), ou podia só admitir determinados *jobs* a partir uma certa hora, tipicamente os *jobs* em *batch*.
- Médio-nível (*process scheduler*) — responsável pelo grau de multiprogramação, transfere processos entre disco e memória e vice-versa.
- Baixo-nível (*dispatcher*) — usa uma estratégia simples e rápida para atribuir o CPU aos processos multiprogramados em memória central.

Há vantagens em que os dois níveis de cima sejam concretizados sob a forma de processos, e não como funções. Mas se forem processos, quem lhes dá tempo de CPU? Aqui a resposta não é complicada: o *process scheduler* tem de ser residente para nunca se bloquear à espera dos seus próprios serviços...O *job scheduler* pode ser um processo "normal", e até é um bom candidato a ir parar à Área de *Swap* pois em princípio não será executado frequentemente.

2.8 Estudo de casos

2.8.1 Unix V

2.8.2 VMS

2.8.3 Windows NT

[9] Compare com o funcionamento de uma secretaria: o pessoal da secretaria prepara os documentos, o "chefe" dá uma vista de olhos e assina rapidamente...

[10] Recorde o que foi dito nas aulas: queremos a todo o custo reduzir o número de instruções executadas neste caminho pois estas funções de "despacho" são executadas centenas de vezes por segundo.