Quarta aula de FSO

José A. Cardoso e Cunha
DI-FCT/UNL

Este texto resume o conteúdo da aula teórica.

1 Objectivo

O objectivo da aula foi o estudo dos sistemas de multiprogramação, em particular a sua utilização nos sistemas interactivos de múltiplos utilizadores, em regime de time-sharing. Inicia-se o estudo do ambiente de execução de processos num sistema como o Unix. Inicia-se o estudo do sistema de ficheiros e as chamadas ao sistema de ficheiros do Unix

2 Rendimento de utilização dos periféricos e do CPU

Na figura 1, ilustra-se a evolução temporal das actividades de *leitura de cartões* (Pi), *processamento* (Pc) e *impressão de resultados* (Po), num sistema de monoprogramação, sem actividade de SPOOL.

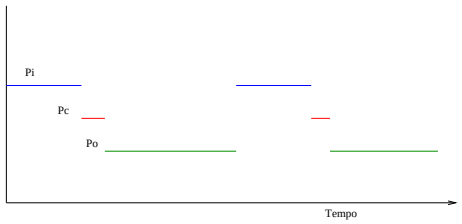

Figura 1: Processamento sequencial de tarefas

Observa-se o seguinte:

- enquanto o primeiro cartão não é lido para memória, o programa não pode iniciar-se;

- enquanto o programa não produzir resultados, a actividade de impressão não se pode iniciar;
- enquanto a actividade de impressão não terminar, as outras actividades não são (re)activadas, embora tal pudesse acontecer;
- a cada momento, só se executa uma actividade no sistema, ainda que haja múltiplos dispositivos físicos: leitor de cartões, processador central e impressora.

Com base no mecanismo de interrupções de programa, é possível executar diversas actividades de forma concorrente, tal como se ilustra na figura 2.

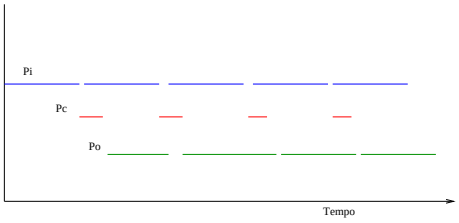

Figura 2: Processamento concorrente de tarefas

Observa-se agora que a leitura de cartões se tornou uma actividade regular, com a vantagem de melhorar a utilização do leitor de cartões que agora se mantem sempre ocupado (enquanto houver cartões para ler e espaço livre de memória para os armazenar). Isto permite ao sistema ir-se antecipando na leitura de cartões para *buffers* em memória e, deste modo, quando o programa necessitar de ler dados, já pode lê-los a partir de memória, não tendo assim tanto tempo de espera. Só quando esses *buffers* de entrada estiverem vazios, é que o programa terá de aguardar, até que sejam lidos mais dados do leitor de cartões. Os intervalos em que o programa aguarda correspondem às zonas 'interrompidas' do traço vermelho da actividade Pc. Durante este tempo, o CPU fica inactivo.

Também a impressão de resultados se tornou uma actividade regular. Para isso, basta que o programa vá escrevendo os seus resultados em *buffers* de memória, em vez de directamente na impressora. Enquanto houver dados nesses *buffers* de saída, a actividade de impressão pode prosseguir.

Como se pode observar pela figura, durante a maior parte do tempo, temos agora três actividades em progressão simultânea: leitor a ler, CPU a executar e impressora a imprimir.

A figura 3 ilustra o tipo de interacções envolvidas no processamento concorrente das entradas, execução do programa e saídas.

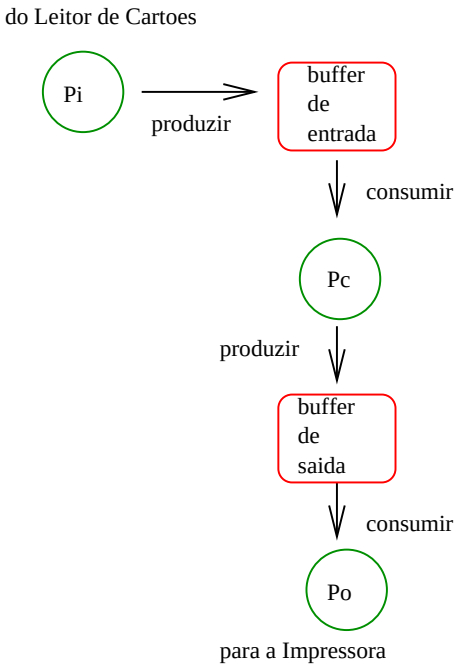

Figura 3: *Buffers* de entrada e de saída

Admita-se um regime de processamento de trabalhos em lotes, segundo o qual se execura cada trabalho sequencialmente, isto é, em regime de monoprogramação. Podem fazer-se as seguintes observações:

1. se o tempo total para a execução de um trabalho é dominado pelas actividades de entrada e saída, em comparação com o tempo de processamento, então haverá intervalos de tempo significativos em que o CPU estará inactivo; por exemplo, sendo os tempos de leitura e de escrita que caracterizam os periféricos, tipicamente muito superiores (da ordem dos segundos, milisegundos, ou microsegundos) aos tempos que caracterizam o ciclo do relógio do CPU (da ordem dos nanosegundos), o processo Pi 'demorará' muito tempo a preencher o *buffer* de entrada, pelo que o processo Pc terá de aguardar, enquanto aquele *buffer* estiver vazio; do mesmo modo, o processo Pc, muito 'rápido' comparado com o processo Po (pois este está limitado pelo tempo típico de impressão), encherá

rapidamente o *buffer* de saída, devendo aguardar até que Po consuma os dados desse *buffer*;

2. se, pelo contrário, o tempo de processamento domina os tempos de entrada e de saída, então haverá intervalos significativos em que os periféricos ficarão inactivos; de facto, sendo agora Pc muito 'lento', demorará muito tempo a consumir os dados do *buffer* de entrada, pelo que Pi não poderá prosseguir a leitura de mais dados; do mesmo modo, Pc produzirá resultados muito lentamente, pelo que Po ficará sem trabalho, enquanto o *buffer* de saída estiver vazio.

Um programa que, durante a sua execução efectua muitas leituras e escritas, tende a exibir um comportamento como o do primeiro caso. Este comportamento diz-se *limitado pelas entradas e saídas* ou *input/output bound*. Um programa que, pelo contrário efectue poucas operações de entrada e de saída, mas tenha longos períodos de cálculo, sobre estruturas de dados em memória, exibirá um comportamento como o do segundo caso, dito *limitado pelo processamento* ou *CPU bound*.

Um perfil *input/output bound* favorece a utilização dos periféricos, mas provoca tempos de desocupação do CPU. Por exemplo, um programa que interaja muito com o utilizador, através de um terminal (teclado e écran), passa a maior parte do tempo a aguardar dados ou a visualizar as respostas.

Um perfil *CPU bound* favorece a utilização do CPU, mas deixa os periféricos inactivos por períodos significativos. Por exemplo, um programa de cálculo científico, para a simulação de um modelo matemático por computador, processa matrizes de grandes dimensão, e tem enormes ciclos de iterações sobre esses dados. Se os dados cabem em memória, tal programa exibe um comportamento *CPU bound*.

A dificuldade é que, em geral, um programa passa, durante a sua execução, por umas fases em que o seu comportamento é *input/output bound* e por outras fases em que é *CPU bound*. Sendo assim, pergunta-se como é que o SO conseguirá melhorar o rendimento de utilização dos periféricos **e** do CPU, se os programas têm um comportamento dinâmico e tão imprevisível? Nos antigos sistemas de monoprogramação, tal não se conseguia.

3 Como garantir uma boa utilização dos periféricos?

Pretende-se garantir um fluxo o mais regular possível de dados de/para os periféricos, o mais independente possível das irregularidades do comportamento dos programas. Deve recorrer-se a *buffers* de entrada e de saída, devem

controlar-se os periféricos pelo mecanismo de interrupções e, dentro do possível, devem explorar-se as capacidades dos processadores dedicados ao controlo das entradas e saídas (ou simplesmente as do DMA, na falta de melhor).

Mas, se os programas presentes no sistema forem todos *CPU bound* torna-se impossível garantir aquele objectivo.

4 Como garantir uma boa utilização do processador?

Para além de 'libertar' o CPU das tarefas de controlo das entradas e saídas, o mais possível, preferindo, por exemplo, o controlo por interrupções, em vez do controlo por espera activa, com teste das 'flags' de estado das interfaces dos periféricos, se só houver programas *input/output bound*, não se conseguem evitar períodos de desocupação do CPU.

5 Como garantir boa utilização dos periféricos e do processador?

A resposta é muito simples: basta manter em memória, isto é, disponíveis para execução, as imagens de **múltiplos** programas, em vez de apenas a de um programa. A imagem de um programa corresponde a um mapa de memória, com regiões (ou 'segmentos') de código, dados e pilha. O objectivo é manter disponível uma 'boa mistura' de programas, de perfis diversos (tanto *input/output bound* como *CPU bound*), capaz de manter ocupados, tanto os periféricos, como o CPU. Este é o objectivo dos sistemas de *multiprogramação*.

Multiprogramação não significa execução simultânea, pois por enquanto estamos a admitir que só existe um CPU no computador: a cada momento só pode estar a executar as instruções de um único programa. Multiprogramação significa que há múltiplos programas presentes no sistema, competindo ao SO decidir, a cada momento, qual deles irá ser executado pelo CPU. Na figura 4 ilustra-se

Quando um programa en execução(na figura denotado por 'processo P1') atinge um ponto em que deve esperar por dados de entrada ou não consegue produzir resultados para um *buffer* de saída que esteja cheio, então o SO decide pôr outro programa em execução (o 'processo P2', na figura), enquanto o primeiro não dispuser de condições para prosseguir. Na figura, cada linha horizontal representa a evolução temporal de um *processo*, isto é, a sequência de eventos determinada pela execução de um programa.

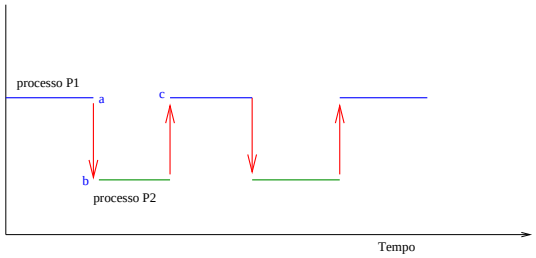

Figura 4: Multiprogramação

Considere um programa em linguagem máquina de um certo processador (por exemplo, o 8086). Para se iniciar a execução desse programa é necessário, como se sabe, inicializar um determinado conjunto de registadores do CPU e estruturas em memória, nomeadamente:

- carregar, em memória, as instruções do programa que se pretende executar, por exemplo, indicadas num segmento de código;
- inicializar, em memória, as zonas de dados especificadas, por exemplo, num segmento de dados;
- inicializar e reservar, em memória, uma zona de trabalho para a pilha de execução;
- inicializar certos registadores do CPU com valores bem definidos, por exemplo, no 8086:
  1. os registadores DS e SS com os valores correspondentes aos endereços reais da base dos segmentos de dados e de pilha em memória;
  2. o registador de estado do CPU, *Flags Register*, com os valores iniciais das flags;
  3. o registador CS com um valor correspondente ao endereço real da base do segmento de código em memória; o registador IP com o valor do deslocamento da primeira instrução, onde se inicia o programa em memória

Uma das *flags* de estado que deve ser inicializada também é a do *modo de operação do CPU*, a qual deve ser posta a indicar o *modo utilizador*. Note que esta flag não existe no 8086, que não dispõe de modos de protecção do

CPU utilizador/supervisor, pelo que esse processador não é adequado para suportar SO com multiprogramação.

A nível dos componentes do computador, o *contexto de execução* de um processo é, inicialmente, definido pelo conjunto dos valores acima indicados, isto é os valores das células de memória de código, dados e pilha, e os valores dos registadores do CPU. Esses valores são determinados pelo SO, com base em informações encontradas no ficheiro executável que se pretende executar, e também com base no modo como o SO inicializa a memória que reserva para aquele programa.

Uma vez inicializado o contexto de execução do processo P1, na figura, o CPU começa a executar as instruções máquina contidas na região de memória onde está o código do processo. Por cada instrução máquina executada, o CPU, como se sabe, pode modificar o valor de alguma célula de memória e/ou os valores de alguns registadores do CPU, incluindo as flags de estado. Por exemplo, o valor de IP é incrementado para apontar a instrução seguinte. Assim, é teoricamente possível seguir todos os passos de execução de um programa, indo consultando os valores do seu contexto de execução. Na prática isto é o que costumamos fazer quando se executa o programa passo a passo, sob controlo de um depurador (*debugger*).

Assim, quando um processo atinge, no seu programa, uma chamada ao SO READ, para ler dados, se não houver dados disponíveis (por um *buffer* de entrada estar vazio), é possível parar a execução deste programa, até que os dados estejam disponíveis. Isto é ilustrado no ponto **a** da figura 4. Como o contexto de execução do processo P1 é bem definido no ponto **a**, o SO pode simplesmente salvaguardar (fazendo cópias em zonas de memória próprias do SO) os valores dos registadores do CPU, correspondentes a esse ponto, bem como as zonas de memória do programa. Isto permitirá que, mais tarde, o SO possa simplesmente carregar nos registadores do CPU, os valores salvaguardados, voltando a apontar as regiões de memória do processo, que assim poderá retomar a execução, a partir do ponto onde tinha parado, mas agora com aqueles dados já disponíveis.

Os pontos a, b e c, indicados na figura, são pontos onde se dá a *comutação* entre os *contextos de execução* dos processos. Uma vez salvaguardado o estado da máquina correspondente ao contexto de execução de P1. o SO escolhe outro processo para execução (o processo P2). Para activar P2, basta carregar os registadores do CPU com os valores correspondentes ao seu contexto de execução inicial, devidamente guardados em zonas de memória do SO. E assim sucessivamente.

6 Exemplos de situações de comutação entre processos

6.1 Leitura de dados com espera activa

Considere a figura 5.

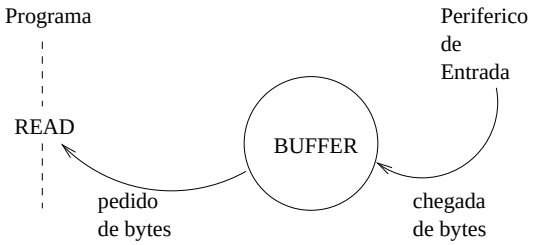

Figura 5: Operação de leitura.

O programa, ao atingir a invocação de READ, deve aguardar até que haja *bytes* para ler, do *buffer*. Os procedimentos de acesso ao *buffer* são os habituais e estão resumidos na figura 6.

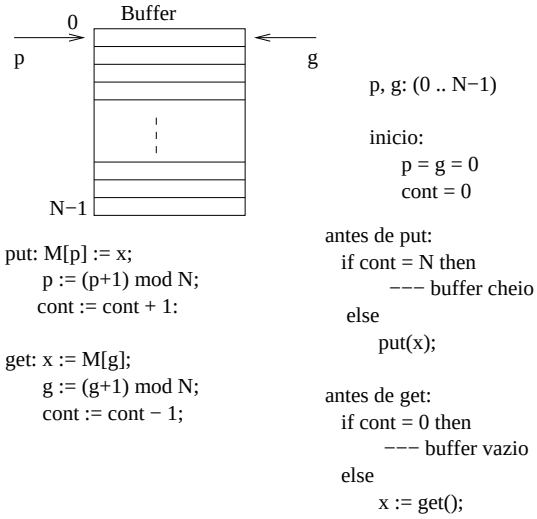

Figura 6: Um buffer e as acções de leitura.

Quando o *buffer* está vazio, função *get* deve fazer o processo que invocou READ, aguardar até que venham *bytes* do periférico. Isto pode ser conseguido

através de um ciclo de espera activa, dentro da rotina READ, conforme ilustrado na figura 7.

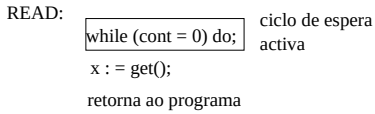

Figura 7: Ler com espera activa.

O esquema apresentado na figura 7 só funcionará nos casos em que o periférico do qual se pretende ler, seja activado por uma acção exterior ao computador. Por exemplo, no caso de um teclado, a pressão de uma tecla desencadeia o envio do código de um carácter para a interface do teclado. Assim, ter-se-ia, do lado da recepção de caracteres vindos do teclado, a rotina de serviço de interrupções, que se apresenta esquematicamente na figura 8.

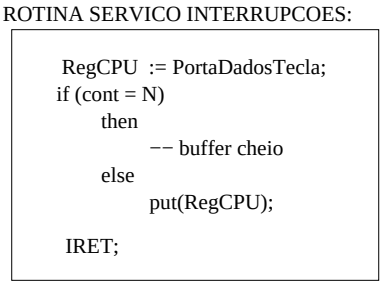

Figura 8: Ler do Teclado.

Para periféricos de entrada que exijam um comando de activação vindo do computador (por exemplo, para ler blocos de um ficheiro de disco exige-se um comando para o controlador de disco posicionar numa pista determinada do disco), tal terá de ser emitido, pela rotina READ, antes do ciclo de espera activa (figura 7).

Este esquema, em que existe um ciclo de espera activa na rotina READ, só é aceitável se o computador estiver dedicado à execução de um único programa. Caso contrário, o CPU ficará ocupado, de forma inútil, na execução das instruções máquina que realizam esse ciclo de espera activa. Dada a grande diferença entre os tempos de finalização das acções dos periféricos e os tempos de execução de instruções pelo CPU, o ciclo de espera activa representa um enorme desperdício de tempo de CPU.

6.2 Leitura de dados em regime de multiprogramação

Imagine-se que o esquema anteriormente ilustrado nas figuras 7 e 8 é executado num ambiente de multiprogramação. Como é que o problema acima mencionado, relacionado com o desperdício de tempo de CPU, ficará resolvido?

Quando o programa invoca a chamada ao SO READ, através de uma instrução especial *chamada ao supervisor*, gera-se um pedido de interrupção, vectorizado para uma rotina de serviço, do SO, que tratará de executar a operação pedida (READ). Note-se que o hardware do CPU, na ocorrência do pedido de interrupção, desliga automaticamente o atendimento de interrupções, empilha o conteúdo do *program counter* e do registador de estado do CPU (*flags*), muda o modo de operação do CPU de *utilizador* para *supervisor* e, finalmente, salta para a rotina de serviço. Esta rotina está descrita, esquematicamente, na figura 9).

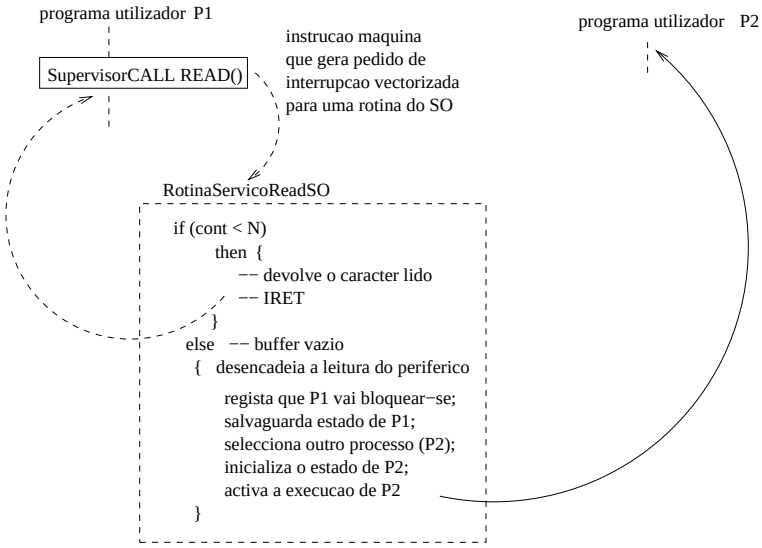

Figura 9: Leitura sem espera activa.

Note-se que a rotina de serviço de interrupções, conforme encontre o *buffer* vazio ou não, decide:

- activar outro programa (P2) (após ter desencadeado a operação de leitura dos dados pedidos por P1 e de ter salvaguardado o estado de P1);
- ou retornar de imediato ao programa P1.

Deste modo, no primeiro caso, durante o tempo em que P1 está *bloqueado*, aguardando a transferência dos dados, a partir de um periférico, o CPU pode estar ocupado a executar as instruções do programa do processo P2.

Quando a operação de transferência de dados, do periférico, estiver completada, será gerado um pedido de interrupção externa, vindo da respectiva *interface*, conforme se esquematiza na figura 10).

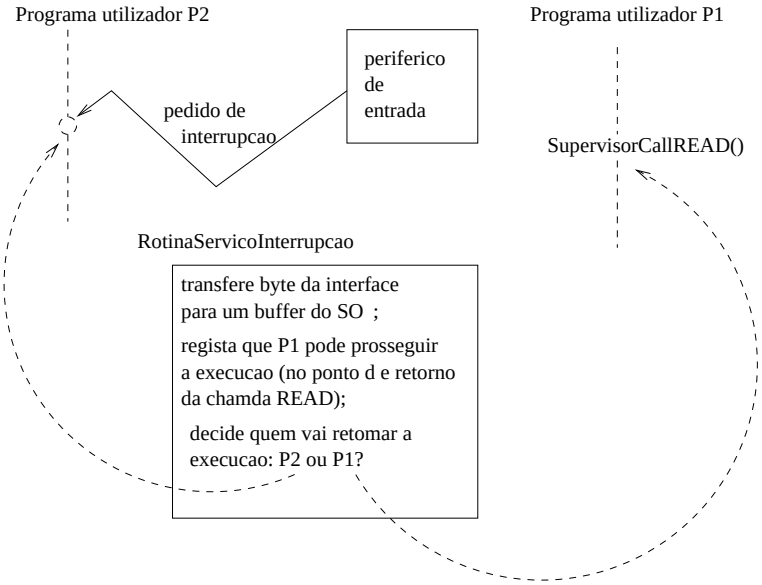

Figura 10: Desbloqueio de um processo.

O tratamento desta interrupção obriga a interromper a execução do programa P2. A rotina do SO que trata a interrupção encarrega-se de ler os *bytes* da interface do periférico, colocando-os num *buffer*, depois actualiza a informação (em zonas de memória internas ao SO) indicando que o processo P1 pode ser desbloqueado (pois já chegaram os dados que aguardava) e, finalmente, decide qual dos dois processos (P2 ou P1) deve prosseguir a execução. Esta última decisão depende da estratégia de *escalonamento* do SO. O escalonamento de processos (*process scheduling*)ăé a actividade do SO que trata de decidir, a cada momento, qual deve ser o processo que deve ser activado para execução. Tal decisão depende de diversos factores, desde uma prioridade que possa ser atribuída a cada processo no início, até factores que dependam do tipo de comportamento exibido por cada programa, tais como o espaço de memória exigido ou a frequência de operações de entrada e saída.

Uma vez decidido qual dos processos deve ser re-activado, o SO retorna ao P2 (executando simplesmente a instrução máquina IRET) ou retorna a P1, para o que necessita de salvaguardar primeiro todo o estado de P2 e depois carregar os registadores do CPU com os valores anteriormente salvaguardados, quando P1 foi bloqueado por ter invocado READ.

6.3 Estados de um processo num sistema de multiprogramação

Qualquer que seja a decisão do SO, isto é, executar P1 ou P2, o outro processo, isto é, P2 ou P1, terá de ficar numa situação em que aguarda uma oportunidade futura para execução. Repare que esta situação é diferente do estado em que P1 ficou quando invocou READ e o *buffer* estava vazio. Neste caso, P1 teria de ficar forçosamente bloqueado, pois os dados não estavam disponíveis. Quando os dados vêm, P1 deixa de estar logicamente bloqueado mas, se a decisão do SO, foi continuar a execução de P2 (por, por exemplo, ter maior prioridade), então P1 fica num estado lógico em que 'aguarda, ainda que esteja pronto para execução', sendo vulgar dizer-se que está *pronto (para execução)* (*ready (to run)*). Quando um processo é escolhido para execução, diz-se que fica no estado *executando*) (*running*). Estes estados lógicos que um processo atravessa, durante a sua execução, estão ilustrados na figura 11.

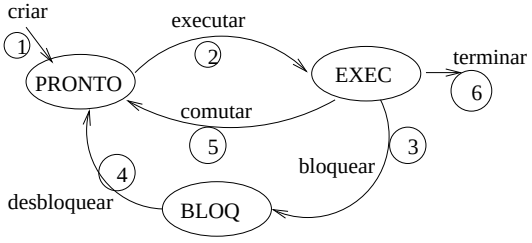

Figura 11: Estados de um processo.

6.3.1 Criação de um processo

Inicialmente o processo é criado (transição 1, na figura). Isto corresponde às acções iniciais do SO, segundo as quais define:

- o mapa de memória do processo, com as regiões de código, dados e pilha

- os valores iniciais de variáveis do ambiente (tais como, quais os argumentos a passar ao programa, quando iniciar a execução, qual a directoria corrente inicial, etc.)
- os valores iniciais dos registadores do CPU, quando o programa for posto em execução
- os canais iniciais para as entradas e saídas
- uma entrada numa tabela, interna ao SO, na qual descreve o estado de cada processo.

Uma vez efectuadas estas acções, podemos admitir que estão preparadas todas as condições para executar o novo processo. Excepto uma! Se só há um CPU no computador, o programa que está correntemente em execução, ou seja, o processo que acabou de criar um novo processo (mais adiante veremos que existe, em cada SO, uma chamada ao SO que efectua as acções de criação acima resumidas), continuará em execução. Só quando este processo 'pai' terminar ou se bloquear, por exemplo, aguardando dados, é que haverá oportunidade para o processo recém-criado ser posto em execução. Isto justifica o estado designado por *pronto*, quer dizer, o processo está 'pronto para execução', mas ainda não tem o CPU para ele!

A esse estado corresponde uma fila de espera, gerida pelo SO, pois pode haver mais do que um processo pronto, aguardando a vez de execução. A figura 12 ilustra esta situação.

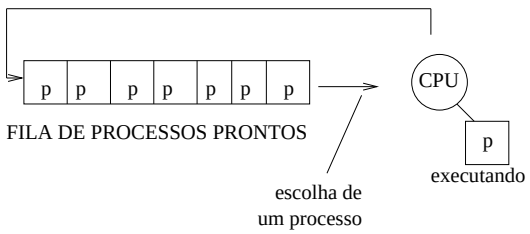

Figura 12: Fila de processos prontos.

6.3.2 Activação de um processo para execução

Quando é que o SO decide escolher um novo processo para execução (transição 2, na figura)? Podemos responder em três tempos:

1. quando o processo correntemente em execução terminar (transição 6) ou se bloquear (transição 3), por ter feito uma chamada ao SO que o obriga a esperar, por exemplo por dados;

2. quando o SO decidir que o processo correntemente em execução, por estar numa fase de tipo *CPU bound*, está a monopolizar o uso do CPU, sem dar oportunidades aos outros processos prontos, de poderem ser executados (transição 5);

3. quando, tendo acabado de haver uma transição de tipo 4, em que um processo bloqueado pode ser desbloqueado (isto é, passa ao estado pronto), o SO decide que este processo tem maior prioridade do que o processo que estava correntemente em execução; isto pode acontecer se, por exemplo, um processo de muito alta prioridade, a aguardar dados, vê esses dados chegarem: pode ser conveniente pôr esse processo em execução 'imediatamente', isto é, mesmo sem esperar que o processo corrente termine ou se bloqueie.

O segundo caso é realizado, na prática, recorrendo ao uso de temporizadores programáveis. Estes são contadores digitais que, uma vez activados, decrementam o valor inicial de um contador, de uma unidade a cada 'tique' do relógio, gerando um pedido de interrupção ao CPU, assim que o contador atinja o valor zero. O valor inicial do contador determina o intervalo de tempo que se quer impor, correspondendo tipicamente a 20 ou 50 milisegundos (o que permite ao CPU executar milhoões de instruções máquina!). Ao activar um processo para execução (transição 2 da figura 11), o SO inicializa o temporizador. Se o processo posto em execução, não terminar ou se bloquear entretanto, então, ao fim daquele tempo, é interrompido pelo temporizador. No tratamento deste pedido de interrupção, uma rotina do SO, verificando que o processo corrente expirou o tempo a que tinha direito, para um uso contínuo e prolongado do CPU, suspende a sua execução temporariamente, isto é:

- salvaguarda o estado dos registadores do CPU, correspondentes ao estado de execução do processo corrente, em zonas reservadas do SO;

- escolhe o processo mais prioritário da fila 'prontos';

- carrega os registadores do CPU com os valores que estejam salvaguardados em zonas do SO, correspondentes ao estado de execução do novo processo.

6.3.3 Bloqueio de um processo

Quando um processo invoca uma chamada ao SO que o obriga a esperar por alguma condição de que necessita para prosseguir, o processo passa ao estado bloqueado:

- o SO salvaguarda o estado dos registadores do CPU, correspondentes ao estado de execução do processo corrente, em zonas reservadas do SO;
- escolhe o processo mais prioritário da fila 'prontos';
- carrega os registadores do CPU com os valores que estejam salvaguardados em zonas do SO, correspondentes ao estado de execução do novo processo.

Exemplos de condições são:

- *buffer* não vazio, quando o processo quer ler;
- *buffer* não cheio, quando o processo quer escrever;
- não há mensagens pendentes, quando o processo quer receber

6.3.4 Desbloqueio de um processo

Exemplos de situações que podem satisfazer as condições de bloqueio são:

- chegam caracteres a um *buffer* que estava vazio;
- são removidos caracteres de um *buffer* que estava cheio;
- chegam mensagens.

Quando se verifica a condição pela qual aguardava um processo, que estava bloqueado, este passa ao estado pronto, ficando a aguardar a sua vez de ser escolhido, em função da decisão do SO.

A figura 13 ilustra de que modo um sistema com multiprogramação, permite ter simultâneamente em memória, diversos mapas de regiões de código, dados e pilha, correspondentes aos múltiplos processos presentes no sistema. A figura também sugere a existência de estruturas de dados, internas ao SO, que descrevem a informação relevante. Em particular, uma Tabela de processos, com uma entrada por cada processo criado, onde se registam as alterações do estado do processo (ver figura 11), bem como se salvaguardam os valores dos registadores do CPU, ou apontadores para onde se encontram outra informação sobre o contexto de execução do processo, por exemplo, os canais abertos para ficheiros.

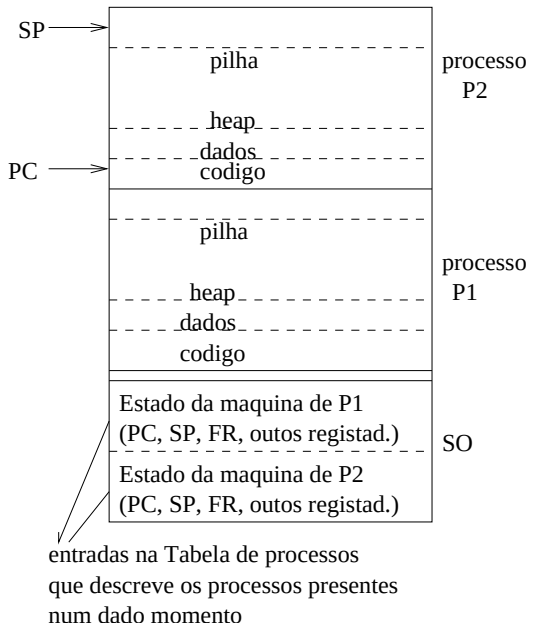

Figura 13: Mapas de memória de processos.