# Segunda 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 a apresentação de alguns conceitos dos sistemas de operação, ligados à evolução histórica dos sistemas de computadores.

## 2 Características dos sistemas de computadores das décadas de 1940-50

Os sistemas de computadores digitais, cujas primeiras realizações com base na tecnologia dos tubos de vácuo, começaram a surgir no dealbar e durante a Segunda Guerra Mundial (1939–45), eram sobretudo utilizados para aplicações militares (cálculos de balística e algoritmos de decifragem de mensagens), ou pelo menos essa foi uma das principais forças que impulsionaram o desenvolvimento desses primeiros sistemas (para uma evolução histórica dos sistemas de computadores, pode consultar o capítulo 1, do livro *Structured Computer Organization*, de A. Tanenbaum, Prentice-Hall).

### 2.1 Máquina Dedicada

Esses primeiros computadores eram utilizados em regime de máquina dedicada: cada utilizador agendava um período de utilização, durante o qual era 'dono e 'senhor' da máquina. Como não havia quaisquer programas de apoio, o utilizador tinha de programar em linguagem binária da máquina e tinha de incluir no seu programa, todo o código de rotinas auxiliares, tais como as rotinas de entrada e saída e as rotinas de carregamento do programa em memória central. Nos primeiros sistemas, na fase inicial dos trabalhos, o utilizador introduzia, em binário, os seus programas directamente em memória, e controlava a sua execução, através de uma consola ou painel

de interruptores e indicadores luminosos, associados às linhas do bus ou aos registadores do processador. Para melhorar um pouco esta fase inicial, surgiram os primeiros programas carregadores, capazes de lerem directamente de periféricos de entrada, tais como os *leitores de fita perfurada* ou os *cartões perfurados*. Esses programas carregadores, escritos em binário e contendo uma dezena de instruções máquina, tinham primeiro de serem carregados em memória, eles próprios. Uma vez carregados, eram postos em execução, para lerem, a partir dos leitores de fita ou dos cartões perfurados. Visando automatizar a inicialização dos primeiros carregadores, surgiu o conceito de *bootstrap loader*, um pequeno programa carregador que, quando se inicializa o computador, é automaticamente posto em execução, com a função de, tipicamente a partir de um periférico pré-definido (unidade de *diskette* ou disco), carregar em memória os programas iniciais do sistema de operação.

***Assemblers* e Compiladores.** A evolução dos computadores, nas décadas de 1940 e 50, conduziu aos primeiros *assemblers*, uma razoável melhoria na programação, por permitirem o uso de uma linguagem simbólica com mnemónicas, ainda que praticamente correspondente às instruções máquina. O próximo passo foi o desenvolvimento das primeiras linguagem de programação de mais alto nível, como o FORTRAN.

Apesar das ligeiras melhorias, o desenvolvimento de programas continuava a ser árduo para o utilizador. A necessidade de muitos procedimentos de inicialização (instalação de fitas e cartões nos periféricos, carregamento e activação dos programas) tornava também muito baixo, o rendimento de utilização da própria instalação do computador. Convém notar que as primeiras realizações dos computadores eram muito caras, pelo que foi dada especial atenção à necessidade de melhorar o rendimento de utilização dos *recursos* do computador. Nos primeiros sistemas, tratava-se de *recursos físicos*, em particular, *tempo de utilização do processador (CPU*, *espaço ocupado em memória* e *tempo de operação dos periféricos*. De forma a optimizar estes parâmetros, tentou automatizar-se o maior número possível de tarefas, ligadas à inicialização, arranque e terminação de programas.

Os aspectos críticos dos primeiros sistemas eram:

- muitos procedimentos manuais de inicialização de periféricos, tais como, por exemplo, a instalação do rolo de fita perfurada num leitor de fita;
- periféricos muito lentos, com tempos típicos da ordem do minuto ou centenas de segundos, quando comparados com o tempo de execução de instruções do processador, já na escala do microsegundo);

- completo controlo da máquina, pelo programa carregado em memória; qualquer operação de leitura ou de escrita de dados, envolvendo interacção com os periféricos lentos (leitor de fita/cartão, unidade de banda magnética, perfurador de fita, ou impressora) deixava o processador desocupado, enquanto aquela operação não se completasse.

## 2.2 Processamento de trabalhos em *batch*

A inicialização e terminação de programas envolvia manipulações de periféricos, pelo que se pensou deixá-las a cargo de especialistas, os *operadores* do sistema. O utilizador deixou de ter acesso à máquina, contrariamente ao regime de máquina dedicada, atrás referido.

A figura 1 ilustra, esquematicamente, a organização dos sistemas de processamento de lotes (*batch*) de trabalhos.

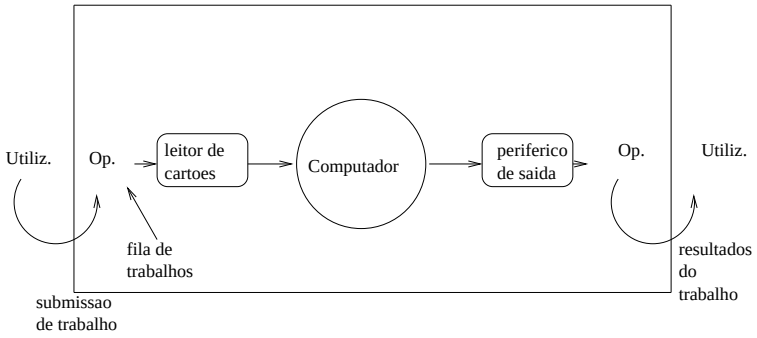


Figura 1: Sistema de processamento *batch*

Cada trabalho (*job*) podia corresponder a uma sequência de passos (*job steps*). Cada passo poderia envolver a compilação de um programa numa linguagem (*assembly* ou FORTRAN) ou a execução de um programa já compilado. O utilizador era responsável por especificar a descrição de cada trabalho, com base numa linguagem de controlo, devendo tipicamente codificar essa descrição sob a forma de cartões perfurados, editados numa máquina perfuradora de cartões. Assim, a descrição do trabalho pedido por cada utilizador constituia uma pilha de cartões, que eram depois entregues ao operador, responsável pela supervisão da execução dos trabalhos. Cartões especiais de controlo indicavam, na descrição do trabalho, informações tais como 'início de cartões de um trabalho', 'compilador de FORTRAN', 'início de sequência de cartões do programa utilizador', 'início de sequência de cartões de dados para o trabalho', etc.

O operador do sistema era responsável por escalonar a ordem segunda a qual os diversos trabalhos submetidos iriam ser executados. Uma vez decidida essa ordem, colocava as pilhas de cartões na 'calha' de leitura de um periférico de entrada: o leitor de cartões perfurados. Os cartões de cada trabalho iriam ser lidos para memória e interpretados por um programa que antes tinha sido carregado em memória (este programa era chamado *monitor control program*) e constituia o embrião dos actuais interpretadores de comandos do terminal (na designação anglo-saxónica *shell*).

Este sistema de processamento de trabalhos era estritamente *sequencial*, isto é, cada trabalho era executado completamente, até terminar e passar o controlo ao monitor de controlo, para este processar o próximo trabalho do lote. Durante a execução, o programa era responsável também pelas operações de entrada (leitura de cartões de dados) e de saída (escrita dos resultados numa impressora). Uma evolução introduzida nos primeiros sistemas foi a de disponibilizar as rotinas de controlo das entradas e saídas, como rotinas auxiliares, que residiam em memória, de modo a poupar o trabalho (e diminuir a probabilidade de ocorrência de erros) do programador. Estas rotinas de entrada/saída, mais o interpretador de cartões de controlo, formavam o que se designava por monitor de controlo ou programa supervisor de trabalhos. Este programa residia numa zona de memória central, conforme ilustrado na figura 2.

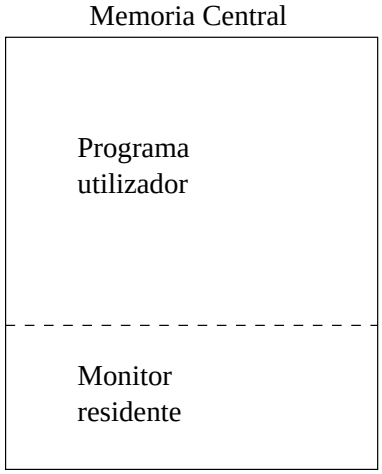


Figura 2: Ocupação de memória

Estes sistemas não ofereciam qualquer mecanismo de protecção contra

erros imprevistos dos programas utilizadores:

1. se o programa gerasse endereços que referissem a zona de memória do monitor, poderia corromper o seu código ou dados;

2. o programa de um utilizador, em situações de erro, podia começar a ler os cartões de entrada, do leitor de cartões, mas correspondentes aos utilizadores seguintes, no lote de trabalhos submetidos;

3. se o programa tivesse um ciclo infinito, nunca terminaria e o controlo nunca retornaria ao monitor de controlo, ficando o sistema completamente bloqueado

Para resolver os dois primeiros tipos de problemas, surgiram mecanismos de protecção, suportados pelo hardware dos próprios processadores (CPU). No registador de *flags* de estado do processador passou a haver um *bit* adicional de controlo, indicando o *modo de operação* corrente do processador.

Se este bit indicar o modo *supervisor ou protegido*, então todas as acções são permitidas na execução das instruções máquina, incluindo o acesso a toda a memória, sem restrições, o acesso aos registadores de controlo do processador, incluindo os que controlam o acesso às tabelas de páginas ou de segmentos dos programas, bem como o acesso às interfaces dos periféricos e o controlo do mecanismo de interrupções. Tipicamente, os programas que constituem o sistema de operação, irão ser executados no modo supervisor.

O sistema de operação, antes de passar o controlo da execução para um programa utilizador, modifica o bit de modo para indicar o modo *utilizador*. Neste modo, o programa só pode gerar endereços que refiram as zonas de memória às quais lhe foi dado acesso, por exemplo, através da sua tabela de páginas (inicializada pelo SO). Qualquer referência de memória fora dessas zonas é detectada pelo hardware do processador, no processo de transformação do endereço virtual, o que, em modo utilizador, origina uma interrupção do programa, por violação de memória. O mesmo controlo é exercido, de modo a impedir o programa em modo utilizador de aceder às portas de interface dos periféricos, executar instruções máquina de entrada e saída e efectuar acções de controlo do processador, por exemplo, para o controlo de interrupções. A figura 3 ilustra, esquematicamente, as acções permitidas e as proibidas.

Como aquelas acções ficam proibidas e são impostas por hardware, os erros dos dois primeiros tipos podem ser controlados pelo SO.

Surge, entretanto, uma dúvida. Se, em modo utilizador, o programa não pode executar instruções de entrada e saída, como é que comunica com o exterior, isto é, recebe dados e produz resultados, envolvendo os dispositivos

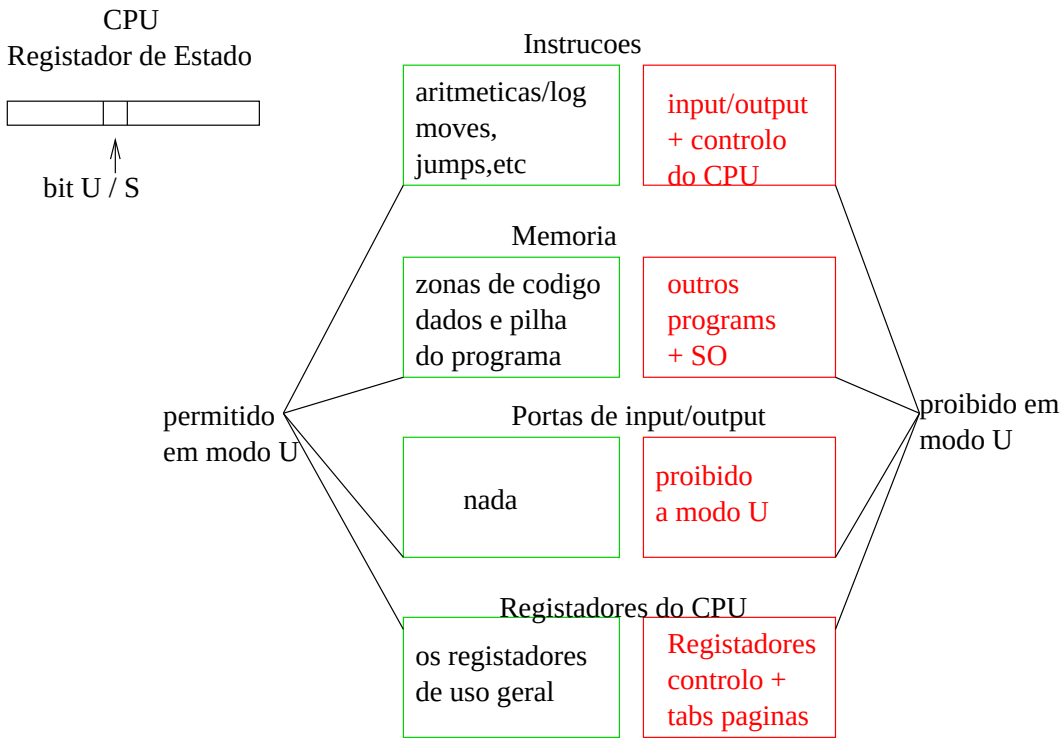


Figura 3: Modos de operação do processador

periféricos, sejam de entrada, de saída ou de arquivo de ficheiros? A resposta é: fazendo pedidos ao SO, através da invocação de *chamadas ao sistema*, conforme se explicou na primeira aula.

## 2.3 A instrução *chamada ao supervisor*

Uma chamada ao SO não pode ser feita com base na instrução máquina *CALL*, pois esta contém explicitamente o endereço de memória destinatário e tenta, como se sabe, afectar o *Program Counter, PC* do processador, com esse endereço. Ora, se este endereço refere uma zona de memória do SO, por corresponder a uma subrotina do SO (por exemplo, para ler dados de um ficheiro), a execução de tal instrução máquina, por um programa em modo *utilizador* daria um erro, por violação de memória. Por outro lado, usar essa instrução exigiria que o programa uutilizador conhecesse o endereço de destino da subrotina a chamar, o que também não é conveniente. De facto,

isto obrigaria a 'religar' os programas utilizadores de cada vez que se alterasse o código do SO.

Precisamos de um mecanismo que permita fazer o seguinte:

- 'saltar' (isto é, afectar o PC) para um ponto de entrada no código do SO, sem que o invocador saiba qual o endereço de destino do salto;
- alterar o modo de operação do processador, para *supervisor*, quando se efectua aquele salto, para que a subrotina chamada tenha todos os privilégios para executar a acção pedida;
- ao retornar ao programa utilizador, repor o valor do bit de estado para o modo *utilizador*

Este mecanismo é suportado por uma instrução máquina que faz o que, nalguns processadores, se designa por uma *chamada ao supervisor*, e tipicamente gera uma interrupção do programa, ou seja:

- empilha o PC e as Flags de estado do processador, põe a flag de interrupções do CPU a zero, põe a flag de modo a indicar *supervisor*;
- salta para uma rotina cujo endereço obtém a partir de uma entrada de uma tabela de vectores de interrupção, cujo índice é indicado como um argumento da própria instrução

Isto é semelhante ao efeito da instrução máquina *int n*, do processador 8086: gerar uma interrupção e saltar para uma rotina, vectorizada pelo código n (a diferença é que, no 8086, não há dois modos de operação do processador).

Para o retorno da execução da subrotina do SO e regresso ao programa utilizador, faz-se o equivalente a uma instrução de *return from interrupt*, o que repõe automaticamente o valor das flags de estado do programa, incluindo o modo *utilizador*.

Em conclusão, o mecanismo de *chamada ao supervisor* é essencial para suportar a protecção do SO face a erros dos programas utilizadores, sem impedir, contudo, que os programas utilizadores tenham acesso aos serviços de entradas e saídas, e outros, que sejam da responsabilidade do SO.

A figura 4 ilustra, esquematicamente, a interacção do programa com o SO:

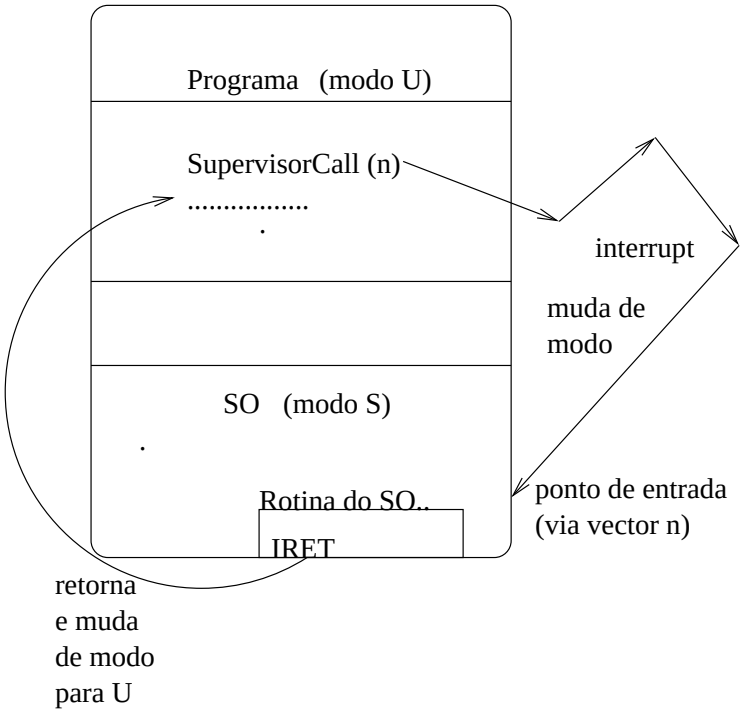


Figura 4: Invocação de chamadas ao SO

## 2.4 Ciclos infinitos

E o terceiro tipo de erros, os ciclos infinitos de um programa que, no sistema de processamento em *batch*, bloqueariam o sistema eternamente? Bem, nos primeiros sistemas, competia ao operador do sistema 'ir vendo' se a execução do programa não estaria, por acaso, a demorar mais tempo do que o 'normal' e, se assim, fosse, o operador tomava a iniciativa de abortar a execução, através dos interruptores da consola de operação do computador.

Uma melhor solução viria a ser suportada com o uso de contadores digitais programáveis, mais habitualmente chamados *temporizadores* (ou *timers*). Uma vez inicializado com um certo valor inteiro, o contador vai sendo decrementado a cada tique de um relógio interno do computador. Ao atingir o valor zero, é gerado um pedido de interrupção, como se o temporizador se comportasse como um dispositivo periférico, pedindo a atenção do processador. Sabendo o período do relógio (intervalo de tempo entre dois tiques sucessivos), pode-se programar o temporizador para um intervalo bem definido, ao fim do

qual é invocada, por hardware, uma rotina de serviço da interrupção gerada. Por exemplo, se o período do relógio fosse 1 microsegundo, um valor inicial de 1000 no temporizador, daria um intervalo de 1 milisegundo. Assim, ao iniciar a execução de cada programa, o SO pode inicializar o contador com um valor adequado, tendo a certeza de que, se o programa ainda não se tiver completado, ao fim daquele tempo, será forçosamente interrompido. Caso contrário, isto é, ao terminar-se o programa antes daquele o tempo, o SO reinicializa o contador.

Como definir o valor adequado para o intervalo de tempo a dar a cada programa? A resposta não é fácil, pois o comportamento de cada programa é frequentemente muito dependente dos dados que consulta, durante a execução. Nos sistemas de processamento em *batch*, havia diversas classes de trabalhos, consoante a duracção esperada, uns mais longos, outros mais curtos. O utilizador devia classficar o seu trabalho, especificando um cartão de controlo, que monitor do SO interpretava, para inicializar o temporizador. Muitas vezes, este processo de definição do valor adequado, ficava sujeito a várias tentativas e erros. Por exemplo, um valor muito baixo para o intervalo de tempo, poderia fazer abortar a execução do programa, quando este ainda estava a fazer trabalho útil e nem sequer estava em ciclo...

Este problema, ligado à detecção da terminação de programas, continua, nos sistemas de operação modernos, a ser resolvido com base neste mecanismo, em boa verdade *ad hoc*, baseado em temporizadores, que definem os chamados intervalos de *time out*. Num sistema interactivo, em que temos um utilizador ao terminal, este problema pode ser contornado, pois podemos deixar ao cuidado do utilizador, a decisão de quando abortar explicitamente a execução do seu programa, através de alguma tecla de controlo, logo que suspeite de qualquer comportamento erróneo... Mas quando se têm dois computadores a trocarem mensagens através de uma rede de comunicações, como é que se sabe se o outro parceiro ainda está activo? A resposta é: não se sabe, e por isso recorre também ao uso de temporizadores (matéria a estudar na disciplina de Redes de Computadores).

Em 1939, Alan Turing analisou este problema e demonstrou a impossibilidade de se conceber um programa X que, recebendo como entrada um outro programa P, decida se P pára ou não. Intuitivamente, o que acontece é que X irá emular P para ver se P pára ou não, donde, no caso em que P não pára (por estar em ciclo), também X não pára e assim não consegue dar a resposta (Esta matéria, na perspectiva da Teoria da Computação, é estudada na disciplina de Algoritmos e Estruturas de Dados 2).