

SISTEMAS OPERACIONAIS

CAPÍTULO 1 - AFINAL, PARA QUE SERVE O SISTEMA OPERACIONAL?

Oswaldo de Souza

Introdução

Os microcomputadores e outros tipos de equipamentos digitais, como celulares, *tablets*, *smartphones* estão completamente integrados em nossas vidas. Apesar de serem diferentes, em relação a suas respectivas finalidades, tamanho e formato, há muita semelhança dentro deles. Todos funcionam por meio do controle de um sistema operacional (SO).

Os principais SOs encontrados em dispositivos móveis, são o SO Android, que pertence ao Google, o IOS, da Apple, e o Windows Phone, da Microsoft, mas a lista abrange vários outros SOs. Se você tem um telefone celular, é provável que utilize alguns desses SOs citados. Neste capítulo, vamos começar a compreender um pouco mais sobre o funcionamento do seu aparelho.

Mas, afinal, para que serve o sistema operacional? Entenda que um equipamento digital é composto de muitas partes, cada uma precisa funcionar em sintonia com as demais. É neste ponto que o SO aparece. Além de integrar todas as partes físicas do equipamento, o SO também é o primeiro responsável pela interação com o usuário.

Se uma aplicação precisa que o usuário confirme uma ação, é por meio do SO que isso ocorre. Quando uma aplicação tem alguma informação para exibir ao usuário, também é o SO que torna isso possível. Na busca por esse conhecimento, temos muitas perguntas, vamos lá! Como um SO é construído? De quantas partes é feito? Todos os SOs são iguais? Pois neste capítulo, vamos chegar às respostas que você espera.

Acompanhe a leitura e bons estudos!

1.1 Conceito e evolução de Sistemas Operacionais (SOs)

Vamos discutir a evolução dos sistemas operacionais, desde um pouco antes de eles começarem a existir. Para isso, vamos entender a evolução do computador, até chegar aos atuais microcomputadores. Acreditamos que, desta forma, você terá um panorama mais completo para compreender a real importância de um sistema operacional.

Talvez você não saiba, mas muito tempo já se passou desde a criação do primeiro computador, em 1642, por um cientista francês chamado Blaise Pascal (TANENBAUM; BOS, 2016). O invento era muito diferente do que vemos nas máquinas atuais, pois era basicamente uma máquina de calcular, mesmo assim, já era incrível. Essa é considerada a geração zero dos computadores.

Somente em 1943 foi criada a primeira máquina eletrônica, batizada de Colossus, com o objetivo principal de decifrar mensagens codificadas por um equipamento chamado Enigma (TANENBAUM; BOS, 2016). Alan Turing esteve envolvido na criação desta máquina eletrônica, considerada como o primeiro computador. Infelizmente não houve evolução, visto que foi mantido em segredo por mais de 30 anos, vindo ao conhecimento do público apenas após este período.

VOCÊ O CONHECE?



Alan Mathison Turing (1912-1954) foi um brilhante cientista, matemático e criptoanalista e teve participação decisiva na Segunda Guerra Mundial. Graças a seu trabalho e de sua equipe foi possível compreender mensagens cifradas pela máquina Enigma e, desta forma, contribuir para o fim da guerra. É importante lembrar que naquela época todo tipo de cálculo complexo tinha de ser feito manualmente. O trabalho realizado nessas condições era altamente sujeito a falhas e frequentemente tinha de ser revisto, o que acarretava em altos custos de retrabalho e de tempo demasiadamente longo, um problema, em tempos de guerra.

O Colossus foi seguido pelas máquinas de segunda geração, que funcionavam com transistores e inventadas nos laboratórios Bells por volta de 1948. Este trabalho rendeu a John Bardeen, Walter Brattain e William Shockley o Prêmio Nobel de Física em 1956.

A primeira máquina da segunda geração foi o TX-0, que não fez muito sucesso. Não se acreditava no sucesso dos computadores naquela época.

Em 1958, surgem os circuitos impressos, nos quais os componentes eletrônicos são integrados, de onde vem a terminologia de circuito integrado, e com isso temos a terceira geração de computadores. Já era possível construir máquinas menores, mais rápidas e baratas.

A evolução não parou por aí e surgiram os circuitos impressos de integração em escala muito grande, os chamados chips *Very Large Scale Integration* (VLSI). Assim, iniciou a quarta geração de computadores.

Nesta quarta geração, o avanço foi tão grande que os computadores começaram a ser chamados de minicomputadores, eram tão mais baratos que até mesmo uma pessoa física podia comprar um deles. Nascia neste momento o computador pessoal (microcomputador). Você comprava e recebia um kit com muitas peças para montar, o que não era fácil. Você teria de ser mais do que um mero usuário, era necessário entender como montar equipamentos eletrônicos / digitais.

VOCÊ QUER VER?



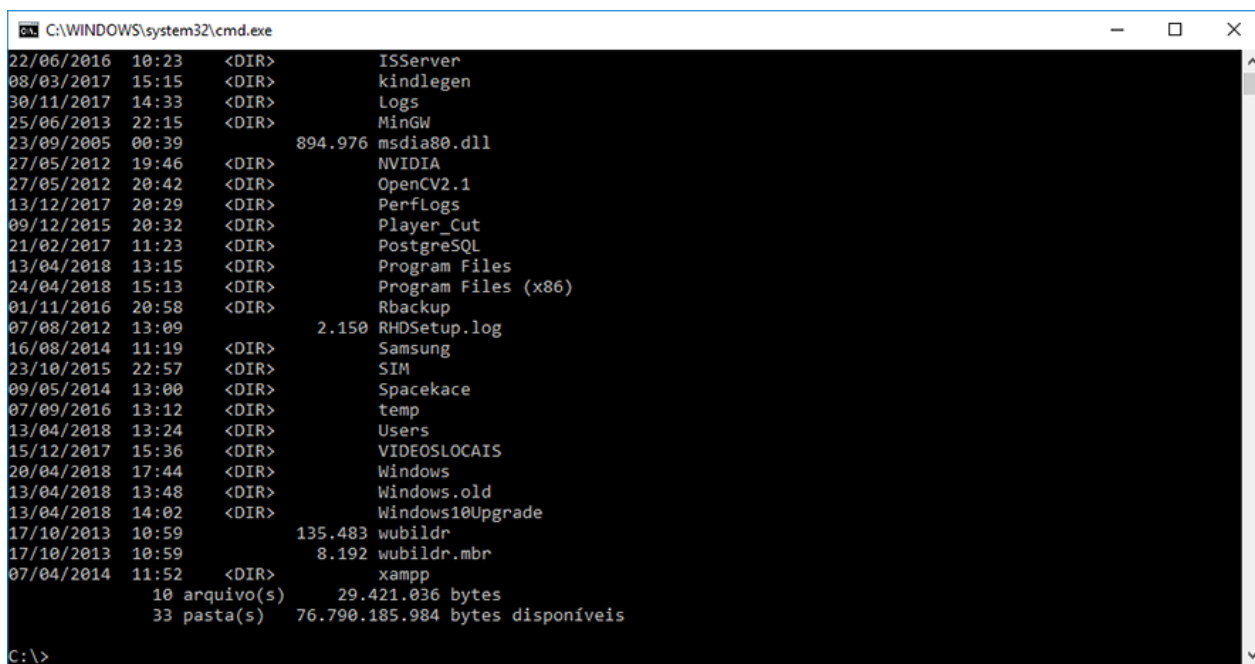
O filme *Os Piratas da Informática: Piratas do Vale do Silício* (BURKE, 1999) retrata a época do nascimento dos sistemas operacionais e do microcomputador. É possível conhecer a história do fundador da Microsoft e da Apple e entender como tudo começou. Na narrativa, você verá como foi o início das duas grandes companhias e como começou a competição entre elas. Também compreenderá como era o cenário tecnológico na época e quais eram as empresas dominantes do mercado antes do fenômeno do microcomputador e dos poderosos sistemas operacionais.

Vale registrar que ao comprar um desses kits, você recebia somente o equipamento, não tinha nenhum *software* incluído. Se você quisesse que o seu computador pessoal fizesse algo, então tinha que escrever o programa que precisava. Foi neste ponto que Gary Kildall (NEMETH; SNYDER; HEIN, 2007) teve a ideia de dar uma ajuda e vender um *software* especial que já fazia a maior parte do trabalho necessário para controlar um computador

peçoal. Assim, ele criou o CP/M, ancestral de todos os Sistemas Operacionais. A seguir, vamos conhecer sobre os diferentes tipos de SO, com destaque para interface, formas de obtenção e memória.

1.1.1 Conceitos de Sistemas Operacionais e classificação

Não foram apenas os equipamentos (*hardware*) que evoluíram, com o surgimento do sistema operacional algumas evoluções levaram a diferentes tipos de SOs. Temos o SO que funciona apenas em modo texto (NEMETH; SNYDER; HEIN, 2007). Mas antes de continuar, o que é modo texto? Pois saiba que se trata de uma interface de estilo muito antiga, sem funcionalidades visuais ou interativas, como você vê na Figura a seguir.



```
C:\WINDOWS\system32\cmd.exe
22/06/2016 10:23 <DIR>      ISServer
08/03/2017 15:15 <DIR>      kindlegen
30/11/2017 14:33 <DIR>      Logs
25/06/2013 22:15 <DIR>      MinGW
23/09/2005 00:39      894.976 msdia80.dll
27/05/2012 19:46 <DIR>      NVIDIA
27/05/2012 20:42 <DIR>      OpenCV2.1
13/12/2017 20:29 <DIR>      PerfLogs
09/12/2015 20:32 <DIR>      Player_Cut
21/02/2017 11:23 <DIR>      PostgreSQL
13/04/2018 13:15 <DIR>      Program Files
24/04/2018 15:13 <DIR>      Program Files (x86)
01/11/2016 20:58 <DIR>      Rbackup
07/08/2012 13:09      2.150 RHDSetup.log
16/08/2014 11:19 <DIR>      Samsung
23/10/2015 22:57 <DIR>      SIM
09/05/2014 13:00 <DIR>      Spacekace
07/09/2016 13:12 <DIR>      temp
13/04/2018 13:24 <DIR>      Users
15/12/2017 15:36 <DIR>      VIDEOSLOCAIS
20/04/2018 17:44 <DIR>      Windows
13/04/2018 13:48 <DIR>      Windows.old
13/04/2018 14:02 <DIR>      Windows10Upgrade
17/10/2013 10:59      135.483 wubldr
17/10/2013 10:59      8.192 wubldr.mbr
07/04/2014 11:52 <DIR>      xampp
      10 arquivo(s) 29.421.036 bytes
      33 pasta(s) 76.790.185.984 bytes disponíveis
C:\>
```

Figura 1 - Exemplo de interface em modo texto que ilustra uma interface textual, muito comum nos SOs antigos, mas que ainda se encontra presente nos modernos.

Fonte: Printscreen do Microsoft Command, 2018.

O acesso e uso a esse tipo de SO é feito por comandos, cada um com função específica, digitados no teclado, como: dir, del, cls, mkdir. Outro tipo de SO é aquele com interface gráfica, como o ilustrado na Figura a seguir, que, no caso, trata-se do Microsoft Windows.

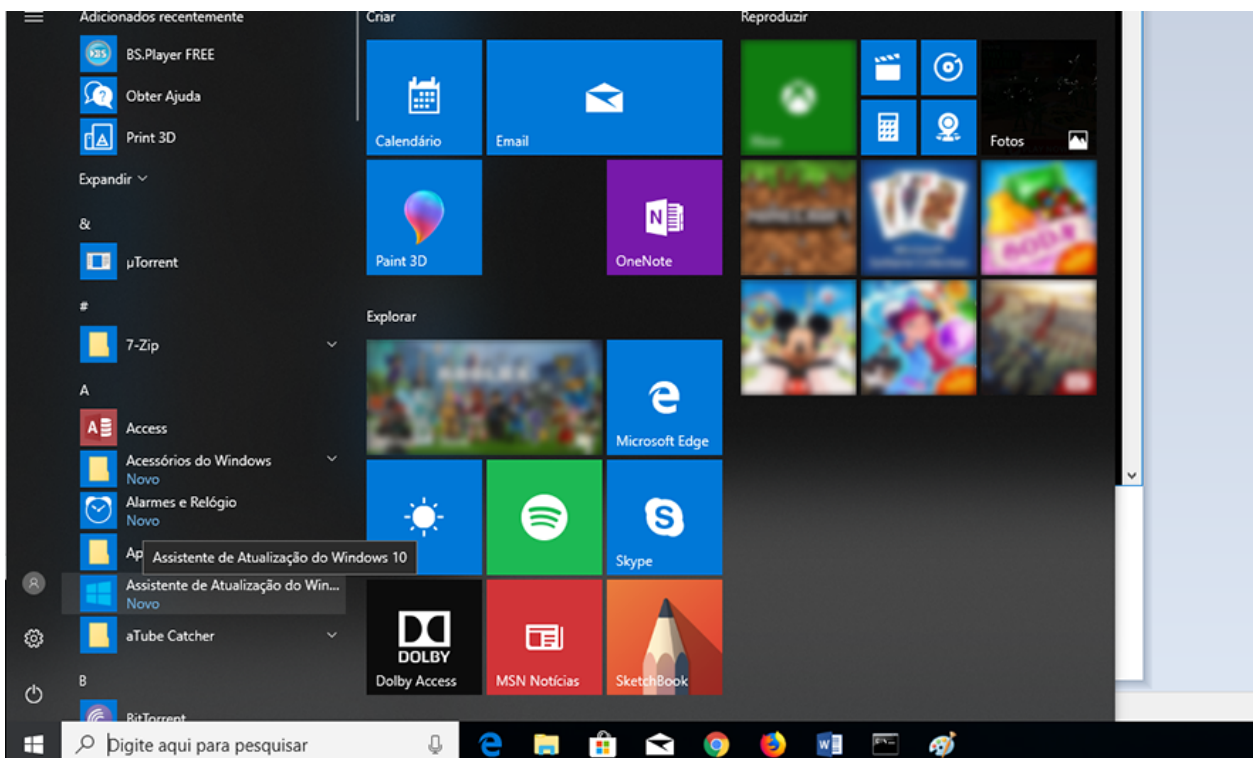


Figura 2 - Exemplo de interface em modo gráfico, à direita há um tipo de menu de opções, e à direita, os itens destacados.

Fonte: Printscreen da interface principal do Windows.

Perceba que o segundo tipo tem características visuais atrativas e também oferece muitas funcionalidades em relação a do modo texto. Atualmente, todos os SOs existentes têm suporte ao modo gráfico, sendo, na verdade, praticamente o padrão para todos eles, todavia, o modo texto ainda está presente mesmo nos SOs mais modernos.

Não é apenas pelo tipo de interface que os sistemas operacionais podem ser classificados, mas também segundo a forma como obtemos esses sistemas operacionais. Nesta classificação, podem ser comprados (pagos) e são chamados de sistemas operacionais proprietários, como, por exemplo, o Microsoft Windows. Ou, então, são de domínio público, gratuito, geralmente referido por *software livre*, como o Linux (NEMETH; SNYDER; HEIN, 2007). Os sistemas operacionais também podem ser classificados quanto a como gerenciam a memória. Existem os que gerenciam apenas memória física e tiveram uso nos primórdios do desenvolvimento dos computadores e também dos SOs, como o CP/M e o MSDos, ambos aposentados hoje em dia.

E há os que gerenciam memória física e virtual, e são todos os SOs modernos, tais como Linux (NEMETH; SNYDER; HEIN, 2007), Windows, Mac OS (TANENBAUM; BOS, 2016). Existem várias versões do Linux, do Windows e do Mac OS, que surgiram com o passar do tempo e com o lançamento de novas versões que sempre traziam diferenças das anteriores. Além disso, você também deve saber sobre a virtualização de memória e as categorias de estruturação do SO, temas que vamos abordar no próximo item.

1.1.2 Virtualização e estruturação do SO

Você sabe o que é virtualização de memória? É capacidade do SO (em conjunto com o equipamento) de gerenciar mais memória do que a máquina tem de fato. Funciona assim: parte do SO e parte das aplicações (programas do usuário), que estão sendo usadas ou executadas, são mantidas na memória física (real) do microcomputador, as outras partes são guardadas no disco rígido da máquina.

Quando uma parte que não está na memória (do SO ou da aplicação) é necessária, o próprio SO escolhe uma região da memória e a desocupa. Para isso, grava o conteúdo dessa região no disco rígido, e então carrega para a memória a parte que passou a ser necessária.

Esse processo de salvar - carregar - salvar é o que define a virtualização e por meio disso a memória do microcomputador só acaba quando esgotam a memória física e a virtual (em disco rígido). É uma maneira do SO, junto com o equipamento, realizarem coisas além do que é possível com a memória física que possuem.

Quanto à necessidade de carregar uma parte do SO ou da aplicação que está no disco, isso acontece, em geral, quando o usuário requisita alguma ação (pelo teclado ou *mouse*) como, por exemplo, carregar um aplicativo para acessar a internet. Posteriormente, quando esse aplicativo estiver funcionando pode pedir mais memória para carregar uma página web ou um filme. Em resposta a essa solicitação, ocorre o processo de virtualização da memória.

Para iniciar uma requisição ao SO ou aplicativo deve-se usar um conjunto de procedimentos, as chamadas de sistema (ou chamadas de SO), que é o mecanismo através do qual os aplicativos conversam com o SO. Um usuário nunca usa diretamente uma chamada de sistema, ele usa um aplicativo que fará o uso.

Esse conceito de virtualização é tão poderoso que é possível (e de fato ocorre) que um SO execute outro SO. Quando fazemos uso desse tipo de estratégia estamos usando a virtualização de SO, muito utilizada em grandes instalações de provedores de serviço de hospedagem de aplicativos, ou mesmo por questões de segurança de dados.

Quanto à estruturação do SO, normalmente temos níveis de responsabilidade dispersos nas seguintes categorias:

- o núcleo (*kernel*) no qual estão os trechos relativos às funções primordiais do SO, sem as quais não haveria nenhuma condição para que o SO funcionasse. Existem SOs que possuem núcleos enormes e outros com núcleos menores. Quanto maior for o núcleo, maior será o consumo de memória que o núcleo irá requerer para manter-se em funcionamento no microcomputador. Também, à priori, quanto maior for o núcleo, maior será o conjunto de funcionalidades que estarão prontas para serem usadas pelos processos sem demandar o envolvimento de memória virtual, ou mesmo da troca de conteúdo de memória. Todavia, quanto maior o núcleo, maior será sua complexidade e maiores serão as possibilidades de problemas. Já os sistemas operacionais com núcleos menores geralmente adotam a estratégia de código transiente complementar, códigos que são anexados (adicionados) ao núcleo apenas quando (e se) necessários, e uma vez que não estejam mais em uso são removidos do núcleo e, portanto, da memória do microcomputador. Nessa abordagem, pode-se ter uma vantagem relacionada ao sistema operacional poder controlar um dispositivo novo, recém-criado, para isso basta o SO receber o acréscimo do código correspondente a esse novo dispositivo;
- os códigos adicionados quando necessários normalmente são denominados de *drivers* (controladores) de dispositivo, ou simplesmente *drivers*. Sua finalidade é permitir que o SO consiga estabelecer comunicação com um novo equipamento que tenha sido conectado ao microcomputador como, por exemplo, um *pendrive*, ou uma câmera de vídeo. A criação desses controladores por vezes é feita pelos fabricantes do dispositivo ou pelos criadores dos SOs. Se você for um fabricante de dispositivos vai querer que todos os SOs possam usar o seu dispositivo, pois é vantajoso para a sua empresa produzir o próprio controlador e disponibilizá-lo junto ao equipamento na hora da venda. Assim, mesmo versões mais antigas dos SOs poderão fazer uso do seu equipamento porque você está fornecendo o controlador necessário para o SO;
- programas utilitários e executores de comandos são programas com finalidades bem específicas e permitem que o SO realize funções altamente especializadas como, por exemplo, interagir com o usuário. Pense nos utilitários como a caixa de ferramentas que permite que você realize tarefas de manutenção do SO. Remover lixo e manter o SO limpo, ou instalar um utilitário de proteção do SO contra vírus são bons exemplos do potencial desses utilitários;
- códigos de carregamento e inicialização são utilizados quando o microcomputador é ligado e tem por finalidade carregar o SO para a memória e, então, entregar o controle do microcomputador ao SO.

Também é importante conhecer como os SOs são estruturados, saber de quantas e quais partes são compostos. Tipicamente, os SOs são compostos dos seguintes módulos:

- **gerenciador de processos:** cuida da criação, execução e controle de todas as tarefas no microcomputador. Faz parte do gerenciador de processos manter o controle dos estados que um processo pode assumir. Ele é fundamental no SO, pois o tempo de processamento de um microcomputador precisa ser compartilhado entre as diversas aplicações (processos) que estiverem sendo executadas. A distribuição desse tempo disponível é feita de acordo com critérios estabelecidos previamente, que podem levar em conta o tamanho de uma tarefa a ser realizada, ou a prioridade do processo.
- **gerenciador de memória:** permite ao SO ter controle do uso e da distribuição da memória disponível no microcomputador, inclusive com o uso de memória virtual quando necessário. Como todo recurso é finito, e a memória não é exceção, torna-se fundamental o controle. É preciso compreender que não é possível manter todos os aplicativos simultaneamente em execução na memória (e no processador do microcomputador), tornando-se, portanto, necessário algum mecanismo de controle que permita que os processos requisitem e liberem memória para atender as suas necessidades;
- **gerenciador de dispositivos:** atua no estabelecimento das condições necessárias para que os diversos dispositivos do microcomputador possam ser utilizados pelos diversos processos que estejam em execução e que necessitem desses recursos. Há dispositivos que podem ser usados de modo compartilhado, como o teclado ou o dispositivo de acesso à rede ethernet, por exemplo, e há aqueles que têm uso restrito a apenas um processo por vez, como o modem ou *mouse*. A responsabilidade do gerenciador de dispositivos limita-se a estabelecer as condições para que os processos consigam comunicar-se com o dispositivo instalador. Para isso, torna-se necessário que o gerenciador de dispositivos tenha conhecimento sobre todos os dispositivos existentes e os que ainda serão criados. Obviamente é impossível preparar-se para controlar um dispositivo que ainda não existe, então foi necessário criar um mecanismo complementar ao gerenciador de dispositivos, trata-se dos *drivers* de dispositivo (ou controladores de dispositivos), que são trechos de código, em geral específicos para cada dispositivo. Assim, se em seu microcomputador tiver um teclado da marca X, usará o *driver* de teclado da marca X, mas se estiver usando o teclado da marca Y, então o *driver* usado será o da marca Y, e assim por diante para todos os demais tipos de dispositivos instalados no microcomputador;
- **gerenciador de arquivos:** embora esteja fortemente associado ao gerenciador de dispositivos, trata com especificidade do mecanismo necessário à organização, criação, leitura e escrita de arquivos nos dispositivos disponíveis no microcomputador. Quando um processo faz uso exclusivo, por exemplo, de um arquivo em disco, é tarefa do gerenciador de dispositivos não permitir que outro aplicativo tenha acesso ao arquivo. O cenário torna-se ainda mais complexo quando se considera que um arquivo pode ser aberto para leitura, escrita ou ambas simultaneamente. Assim, deve-se considerar que para a leitura vários processos podem abrir o arquivo ao mesmo tempo, todavia, para a escrita torna-se um problema se mais de um processo estiver escrevendo ao mesmo tempo no mesmo arquivo, pois pode ocorrer perda de dados ou o arquivo pode ser corrompido. Para complicar ainda mais o cenário, é preciso entender que um disco rígido, por exemplo, pode ser dividido em várias partes (partições) e cada uma destas partições pode ser organizada (e codificada) de maneira diferente. Discos (ou partições) codificados diferentemente irão requerer diferentes esquemas de acesso. Vamos a um exemplo: se você estiver usando o sistema operacional Windows, essa codificação pode ser do tipo *fat*, *fat32* ou *ntfs* (*New Technology File System*), entre outras. É preciso, então, primeiro conhecer e controlar esses diferentes tipos de codificação e depois controlar os arquivos e pastas que existam dentro dos discos ou partições.

Alguns SOs podem ter mais módulos do que esses apresentados, ocorre que em algumas construções de SO os arquitetos preferem subdividir responsabilidades, na tentativa de deixar o código-fonte mais fácil de compreensão e manutenção. É comum encontrar um módulo que seja responsável exclusivamente pelo suporte a rede de dados, ou então um dedicado a parte gráfica do SO, ou ainda um responsável exclusivamente pela implementação das políticas de segurança.

Importante destacar que o SO não deve impor políticas, mas disponibilizar recursos (mecanismos) para a imposição de política. Com isso, enfatiza-se que não é uma função do SO permitir ou não que um usuário instale um aplicativo de jogo em seu microcomputador. O SO deve, sim, prover os meios para que o usuário faça as instalações que julgar conveniente, visto que a função do microcomputador e, portanto, do SO, é atender as necessidades dos usuários (quando possíveis), sejam elas quais forem.

Do ponto de vista da complexidade de construção dos sistemas operacionais, existem, de maneira geral, duas grandes vertentes de *design* arquitetural: a vertente de construção de um núcleo monolítico e a vertente que direciona para a construção de um núcleo em camadas.

1.1.3 Design arquitetural, espaço do núcleo e espaço do usuário

No núcleo de construção monolítica, todos os elementos (módulos), que compõem o SO, fazem parte de um único bloco, ainda que partes de bloco tenham responsabilidades diferentes. Cada parte tem uma finalidade e estabelece intercomunicação com as demais, sem necessitar de protocolos de comunicação, critérios de identidade e políticas de segurança. A velocidade de comunicação entre os módulos é a mais alta possível por tratar-se de uma estrutura na qual todos os módulos têm acesso a regiões de memória que são compartilhadas entre eles. Contra esse modelo de arquitetura de construção de núcleo de sistema operacional afirma-se que ele leva a uma programação (criação dos códigos-fonte) muito complexa e com forte dependência entre as diversas partes, de forma que a falha em uma delas leva o núcleo a degenerar rapidamente.

Na segunda abordagem, que direciona para um padrão arquitetural em camadas, as camadas mais inferiores provêm serviço para as superiores, de forma que quanto mais baixa for a camada, maior será o envolvimento com o *hardware* do microcomputador. Essa arquitetura permite que mudanças em uma camada sejam facilmente isoladas das camadas acima e abaixo, de forma que cada uma possa evoluir individualmente, sem que o todo possa configurar-se como um código extremamente complexo.

Entre as camadas é estabelecido um protocolo de comunicação que permite que serviços e recursos providos pelas camadas inferiores sejam acessados de maneira estável, ordeira e com fácil aplicação de políticas de segurança e de registro das atividades para fins de auditorias futuras.

Nesse padrão arquitetural também é possível que as diferentes camadas sejam dedicadas a diferentes grupos de funções como, por exemplo, a primeira camada (a mais baixa) teria a responsabilidade de comunicar-se e controlar os equipamentos, a segunda poderia estabelecer mecanismos de controle de acesso à primeira, permitindo a construção de políticas de acesso, de proteção e de segurança dos dados. Contra esse modelo que trabalha com ênfase na comunicação entre as camadas e na separação de funções, afirma-se que prejudica o tempo de comunicação entre as camadas, e que uma necessidade de recurso de um aplicativo do usuário possa demorar demais para chegar ao *hardware*, tendo em vista que entre as camadas ocorre uma intensa comunicação, e esta envolve regras bem rígidas, as quais estabelecem complexos protocolos de comunicação.

Vimos, até aqui, que o núcleo do SO é o que de fato controla o microcomputador, portanto, deve ter uma maneira de proteger o próprio SO para que um aplicativo do usuário, por exemplo, não consuma toda a memória do microcomputador e acabe destruindo o SO que estava na memória. Para essas situações foram criados dois espaços bastante distintos: o espaço do núcleo e o espaço do usuário. No espaço do núcleo, apenas códigos pertencentes ao SO podem ser executados, qualquer uso deste espaço por outros aplicativos resultará em um erro e o programa que tentar isso será encerrado pelo SO.

Caso um aplicativo precise de algum serviço do SO, ou de algum dado que esteja no espaço do SO, deverá acessar esses serviços ou dados utilizando-se das chamadas de sistema. A principal finalidade das chamadas de sistema é permitir a comunicação entre o SO e os aplicativos, mas, ao mesmo tempo, manter separados e seguros os espaços do núcleo e o do usuário.

Espaço do usuário é, portanto, a região da memória em que são livremente carregados e executados os aplicativos do usuário. Os aplicativos utilitários que acompanham o SO também são executados neste espaço. Pense nesse espaço como um enorme parque de diversões controlado pelo SO.

Mas se por algum motivo o usuário precisar que algum aplicativo funcione no espaço do núcleo? Nesta situação, não há como um aplicativo do usuário funcionar no espaço do núcleo, todavia, o usuário pode separar algumas partes do seu aplicativo e reescrevê-las como um *driver* e, então, pedir ao SO para carregar esse *driver*. Depois de carregado, as funções desse *driver* serão executadas no espaço do núcleo e poderão ter acesso direto às funcionalidades do núcleo, mas é uma região muito delicada, qualquer erro e o SO travará (ou reiniciará) e o usuário terá de reiniciar o microcomputador. Dependendo do tamanho do estrago causado, poderá ser necessário reinstalar todo o SO e, nesta situação, o usuário poderá ter grande perda de dados. Vamos agora examinar como o SO faz o gerenciamento dos recursos que estão disponíveis na máquina. Para começar, vamos aos conceitos básicos e depois examinar a aplicabilidade desse gerenciamento.

1.2 Gerenciamento de recursos

Nosso primeiro passo aqui é estabelecer o entendimento do que se refere à palavra recurso no contexto dos sistemas operacionais. Por recurso, devemos compreender toda funcionalidade ou capacidade que esteja presente no microcomputador. Alguns recursos são providos através de *hardware* e outros de *software*, portanto, há duas categorias de recursos. Como exemplo de distinção, vamos citar a capacidade de uso de interface gráfica. Tanto no Windows como no Linux, há um motor gráfico com o papel de prover funções primárias de manipulação de interfaces gráficas, tais como: criação de um espaço de interação, ou criação de um objeto de interação. Os espaços de interação devem ser compreendidos como todos os espaços em que o programador pode inserir conteúdo e outros elementos na tela. Quando você inicia um aplicativo em um ambiente gráfico, geralmente ele cria uma visualização gráfica que fica contida dentro de uma janela. Esta janela é um espaço de interação em que são criados novos objetos, normalmente com a finalidade de serem utilizados na interação entre o programa e o usuário, por isso são denominados de objetos de interação.

Estes elementos, espaços de interação e objetos de interação, são providos pelo motor gráfico (alguns o chamam de servidor gráfico). Mas esse motor gráfico não consegue, de fato, criar a visualização, pois requer que sejam dados comandos ao *hardware* do monitor, e estes comandos são emitidos pela placa gráfica do microcomputador. Então, o motor gráfico solicita os recursos da placa gráfica, que solicita os recursos do monitor.

Note que nesse cenário há claramente a colaboração entre dois recursos: o de *hardware*, representado pela placa gráfica e pelo monitor, e de *software*, que no exemplo é provido pelo motor gráfico.

Em geral, para quase todos os recursos que interagem com o usuário, é usada uma abordagem parecida. Há um recurso de *software* que gerencia as saídas de áudio, e há o *hardware* de áudio que produz o som. Agora que você compreendeu isso, aproveite para ouvir uma música tranquila enquanto lê sobre o gerenciamento de recursos.

1.2.1 Conceitos básicos e aplicabilidade de gerenciamento de recursos

Os recursos que estão disponíveis em um microcomputador são limitados: memória (mesmo virtualizada), espaço em disco rígido, capacidade de processamento. Todo elemento de *hardware* e *software* adicionado ao microcomputador torna-se um recurso que pode ser usado por um único aplicativo, ou compartilhado entre vários outros.

Tome como exemplo uma placa de rede, que permite ao microcomputador conseguir comunicar-se com outros equipamentos mesmo que estejam do outro lado do mundo, o acesso a este recurso é compartilhado entre todos os aplicativos que estejam instalados no microcomputador.

A tarefa de gerenciamento desses recursos cabe ao SO e ele tem de lidar com as questões relacionadas ao controle de estado de um recurso (pronto, disponível, desligado etc.), ao controle de qual aplicativo (e usuário) tem acesso ao recurso, manter um histórico de quem faz uso dos recursos, alocar e desalocar os recursos e manter controle dessa variação de *status*.

Também é tarefa do sistema operacional controlar o quanto um usuário ou aplicativo pode ter de tempo e uso de um determinado recurso. Esse balanço entre quem usa e por quanto tempo é a chave de um gerenciamento de sucesso em qualquer SO. O tempo que um processo utiliza é o tempo em execução em alguma das unidades de processamento central (CPU) que o microcomputador possua. Atualmente, os microcomputadores possuem várias CPUs e quanto maior o número de CPU, maior a capacidade de processamento.

Para que seja possível ao sistema operacional controlar o uso dos recursos é necessário que cada aplicativo em execução, e cada recurso disponível, seja identificado por identificadores únicos. Quando um aplicativo é iniciado pelo sistema operacional é associado um número de processo a este aplicativo que, posteriormente, pode desdobrar-se em vários processos, para fins de múltiplas execuções em paralelo, por exemplo. Portanto, o conceito de processo é o modo com o qual o SO lida com os aplicativos.

Já os recursos que estão disponíveis no SO recebem também um identificador, um número de registro do recurso, ou um endereço de porto de entrada do recurso, dependendo de qual sistema operacional utilizado. De qualquer maneira, seja a primeira ou a segunda estratégia, os usuários não lidam diretamente com isso, quem lida com essa problemática são os aplicativos. E os aplicativos para fazerem uso dos recursos sempre usam as chamadas de sistema que são providas pelo SO. Muito bem, agora que vimos sobre gerenciamento de recursos do SO, vamos para outro importante componente, o gerenciamento de processos.

1.3 Gerenciamento de processos

O gerenciamento de processos trabalha sempre em um ritmo intenso, com a responsabilidade de manter a fila de processos de maneira que nenhum fique aguardando demasiadamente, bem como impedir que um único use tempo demais de processamento. Caso o gerenciador de processos não realize corretamente a distribuição de tempo de processamento da CPU, podem ocorrer travamentos que comprometam o funcionamento geral do SO, ou podem fazer com que algumas tarefas não se completem.

Quer um exemplo? Quando você pede a alguns SOs para carregarem um aplicativo, e enquanto isso tenta fazer algum outro uso, continuar a assistir a um vídeo que estava em pausa, por exemplo. Muitas vezes o vídeo não retorna, pois o SO estava ocupado demais carregando o aplicativo solicitado. Temos aqui um problema de escalonamento de tarefas ou de sobrecarga de um recurso. Todo e qualquer aplicativo que esteja em execução em um microcomputador controlado por um sistema operacional é encarado como um processo. Mas como podemos definir processo? Continue a leitura para chegar à resposta.

1.3.1 Definição de processo

Podemos entender como é o processo de um aplicativo sendo executado, com parte do seu código de execução (ou todo, se for pequeno) em memória real (ou virtual), além, também, de existir um contexto associado. Um contexto de um processo refere-se a todos os recursos que ele esteja usando, como memória, arquivos abertos no disco rígido, conexão com a impressora. Para que um processo seja corretamente executado, ele precisa do seu código de execução e de seu contexto. Em alguns sistemas operacionais, o termo usado para um processo pode ser tarefa ou *job*.

Durante o período em que um determinado processo está ativo ele pode ser encontrado em vários estágios:

- a) novo – quando o processo está em fase de criação;
- b) pronto - quando o processo está preparado e aguardando ser direcionado para a execução em uma CPU;
- c) esperando - quando o processo está esperando algum recurso ou evento, para que seja executado;
- d) executando – é o estado no qual o processo está em execução em alguma CPU;
- e) terminado – quando o processo não está mais em execução e o seu contexto já foi liberado.

Para você entender melhor o assunto, vamos estudar a seguir, os diagramas de execução e os estados do processo.

1.3.2 Diagrama de execução e estados do processo

Na Figura a seguir, podemos ver como os estados e a interação entre os estados que um processo pode ter. Observe que entre o início (novo) e o término (terminado) um processo percorrerá várias vezes os estados possíveis.

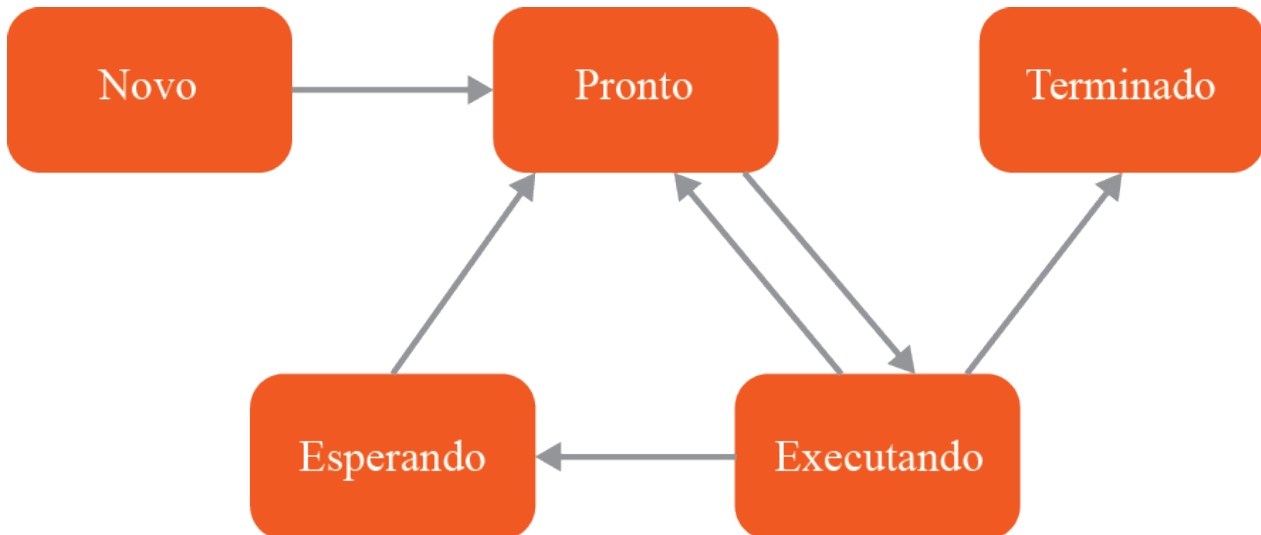


Figura 3 - Diagrama de Estados dos Processos mostra os estados que os processos podem assumir durante o ciclo de sua execução na CPU.

Fonte: Elaborada pelo autor, 2018.

A transição do estado de **novo** para **pronto** ocorre logo após a criação do processo, em seguida será escalonado (selecionado para a execução) para a primeira execução, e neste período fará a solicitação aos recursos que venha a necessitar. Perceba que o estado **pronto** se refere à situação na qual o processo tem reunida todas as condições para ser executado. A partir da solicitação de recursos, que ocorre na primeira execução ou ao longo do tempo em que o processo estiver ativo, ele poderá ter a transição para o estado de **esperando**, e ficará até que os recursos solicitados sejam fornecidos ao processo. Quando um processo está no estado pronto, ele aguardará apenas na fila para ser escalonado, e durante sua execução pode passar aos estados de esperando (no caso de solicitar novos recursos) ou pronto (no caso de terminar o seu *quantum* de execução e precisar aguardar por um novo período) e de terminado. O estado de **terminado** ocorre quando o processo chega ao fim, não restando mais nenhum processamento a ser feito. A mudança de estado de um processo ocorrerá sempre em função da prioridade, respeitando-se o tipo de política de escolha para a execução que estiver em uso pelo gerenciador de processos, que este processo possui (menor prioridade recebe menos tempo de execução), bem como da disponibilidade dos recursos que o processo vier a solicitar. Compreenda que a prioridade é um conceito adaptável, quando se está falando de gerenciador de processos, dependendo do algoritmo em uso, a prioridade será: do menor trabalho, do mais antigo, do que estiver esperando a mais tempo.

Cada processo é representado por um bloco de controle de processos, o qual consiste em um conjunto de metadados sobre os processos, normalmente referenciado por *Process Control Block* (PCB). Um PCB contém informação sobre:

- o número do processo que o identifica;
- o estado atual do processo, dentre os vários estados que um processo pode assumir;
- um contador do programa que aponta o endereço de memória que contém a próxima instrução do código de execução do aplicativo que deverá ser executado quando o processo mudar para o estado de executando;

- os registradores da CPU, que são variáveis mantidas pelo SO e fundamentais para a execução do processo;
- dados que indicam a prioridade do processo dentro da política de escalonamento que estiver em uso no SO;
- a gerência de memória para o processo;
- o *status* de E/S que inclui dados dos recursos (dispositivos) de E/S utilizados pelo processo.

Por fim, inclui a contabilização do tempo de CPU, que o processo já consumiu e a quantidade de CPU alocada, dentre outras informações.

VOCÊ SABIA?



Eventualmente, pode ocorrer de dois processos bloquearem um ao outro? Suponha que existam dois processos: A e B. O processo A tem controle do disco, mas precisa de memória para poder carregar um conteúdo. Enquanto isso, o processo B tem controle da memória e deseja liberá-la salvando o conteúdo no disco, que está em uso pelo processo A. Nesta situação, A espera que B libere um recurso, enquanto B espera que A libere recursos. O resultado é um travamento mortal, bastante conhecido como *deadlock*.

Na Figura a seguir, podemos ver uma ilustração de como o PCB é gerenciado no momento em que o SO precisa trocar de contexto para salvar e carregar um novo processo para a execução em alguma CPU.

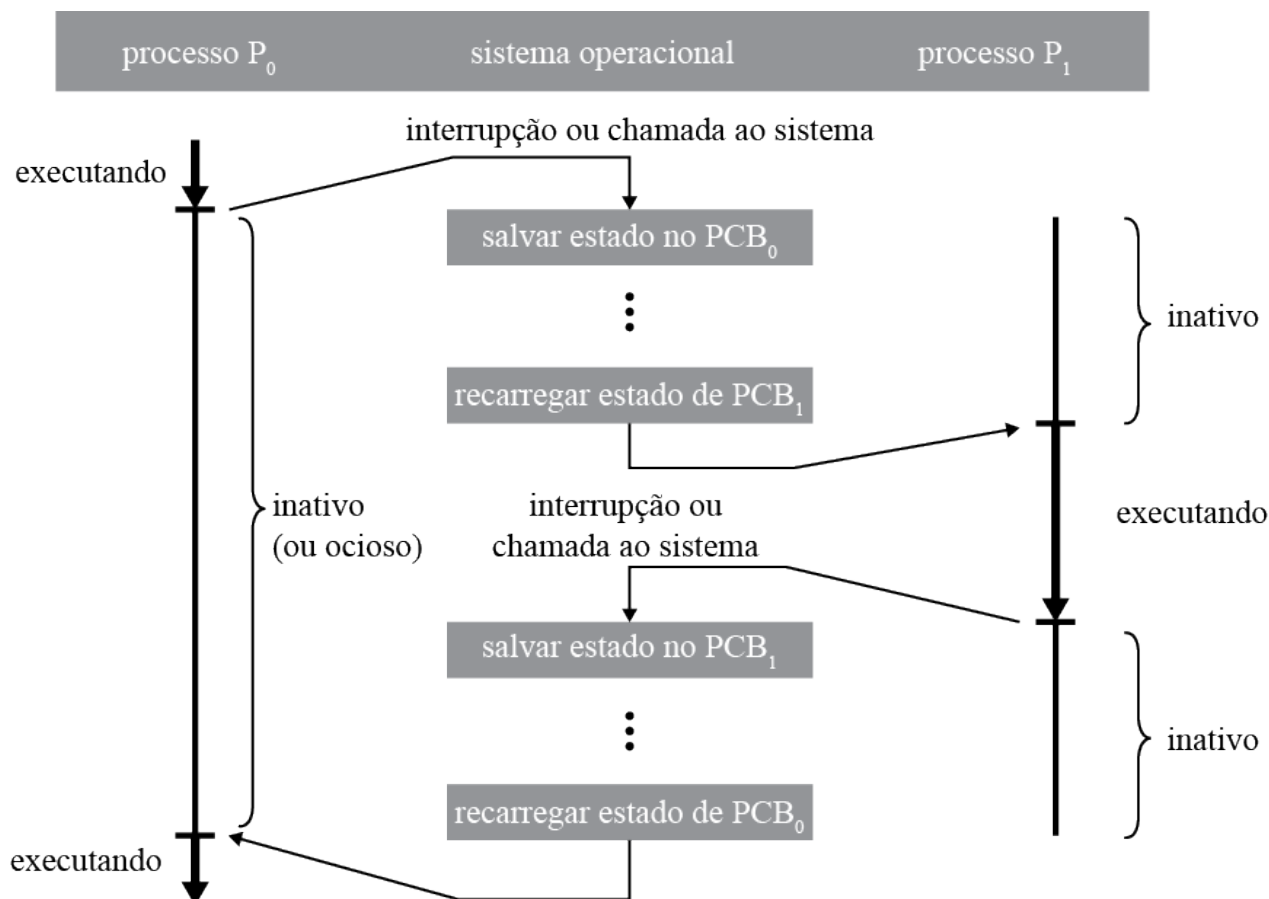


Figura 4 - Ilustração do Ciclo de troca de contexto para a troca processo na CPU, no caso entre PCB0 e PCB1.

Fonte: Silberschatz; Galvin; Gagne, 2004, p. 77.

Perceba que, quando vai ocorrer uma troca de processo na CPU (período inativo), o contexto (PCB) do processo que estava em execução é salvo em outra região da memória. Em seguida, o contexto do próximo processo a ser executado é carregado, e somente após essa carga é que o processo poderá ter sua execução iniciada.

VOCÊ SABIA?



Há um desafio bastante conhecido, quando se estuda o funcionamento dos sistemas operacionais: o dos filósofos glutões. Você já conhece? O desafio é assim: existem cinco filósofos sentados ao redor de uma mesa redonda. Eles estão prontos para o jantar. Ocorre que em frente a cada um deles há um prato para comer macarrão, e entre cada prato há um único *hashi* (palitinho de madeira). Como precisamos de dois *hashis* para comer, se todos pegarem os *hashis* ao mesmo tempo, ninguém vai conseguir comer, do contrário, apenas dois poderão comer por vez. Como você resolveria a situação de forma que todos pudessem comer, ainda que não ao mesmo tempo?

O trabalho de escolha de qual processo deve ir para o *status* de executando, bem como para os demais estados requer uma estratégia de escalonamento, também denominada de política de escalonamento. Dentre as várias escolhas possíveis, as mais comuns são: FCFS; SJF Não Preemptivo; SJF Preemptivo; Prioridade Preemptivo; Round Robin.

CASO



O trabalho de escolha de qual processo deve ir para o *status* de executando, bem como para os demais estados requer uma estratégia de escalonamento, também denominada de política de escalonamento. Dentre as várias escolhas possíveis, as mais comuns são: FCFS; SJF Não Preemptivo; SJF Preemptivo; Prioridade Preemptivo; Round Robin.

De maneira geral, segundo Tanenbaum e Bos (2016) e Deitel (2005):

- o escalonamento FCFS tem por direcionamento atender o processo que chegar primeiro, resultante direta da tradução literal de *First come, first served*, para o qual temos que o primeiro a chegar, será o primeiro a ser servido sendo um algoritmo não preemptivo. O termo preemptivo refere-se à política de escalonamento na qual o processo é interrompido sempre que o SO julgar necessário. Dentre os motivos que justificam o SO interromper o processo está o término da fatia de tempo (*quantum*) que o processo possui;
- o SJF não preemptivo, no qual se escolhe o processo que represente o menor trabalho, introduz aqui um problema relativo a: como saber qual é o menor trabalho? SJF vem do inglês *Shortest Job First*, que significa o menor trabalho primeiro. Por ser não preemptivo este processo, uma vez iniciado na CPU, é executado até o seu término;
- o SJF preemptivo também escolhe o processo que represente o menor trabalho e interrompe o processo após o término do *quantum* de tempo;
- o escalonador por Prioridade Preemptivo faz o escalonamento baseado na simples prioridade dos processos e encerrando-os quando esses processos esgotarem os seus respectivos *quantum* de tempo;
- o algoritmo escalonador Round Robin é idêntico ao FCFS, porém é preemptivo, assim, trata-se de um sinônimo de escalonamento por revezamento, e é uma abordagem bastante comum.

Para o próximo tópico, vamos entender um pouco mais sobre como ocorre a interação do gerenciador de processos com o sistema.

1.4 Interagindo com o gerenciador de processos

O gerenciador de processos possui certos níveis de configuração, mas pode definir arbitrariamente a prioridade de um determinado processo. Apesar de termos a impressão de que um microcomputador executa vários processos ao mesmo tempo, isto não é fato. Ele executa apenas um processo por *quantum* de tempo, que nada mais é do que a menor fatia de tempo que um processo recebe para ser executado em uma CPU. A priori, todos os processos recebem o mesmo *quantum* de tempo, mas o usuário pode definir novas divisões deste tempo.

Caso o usuário decida que um processo é mais prioritário, será dada uma fatia maior de tempo para esse processo, portanto, o seu *quantum* de tempo será maior do que dos demais processos, fazendo com que seja executado mais rápido. Para que ocorra a interação dos aplicativos com o sistema operacional sempre será utilizado algum recurso de comunicação entre aplicativos. Vamos conhecer a linguagem de programação a seguir.

1.4.1 Usando linguagem de programação

A comunicação entre aplicativos ocorre através da *application programming interface* (API), que significa interface de programação de aplicativos, constituída de um conjunto de chamadas de sistema e ou chamadas de aplicativos. É com essas API que qualquer programa se comunica com o sistema operacional e por vezes com outros programas que estejam em execução no microcomputador. Em alguns sistemas operacionais, a comunicação entre aplicativos recebe o nome de *interprocess communication* (IPC).

Na Figura a seguir podemos ver o conjunto de API para acesso a memória no Windows. As APIs são utilizadas por meio de linguagens de programação, seja a linguagem Java, C, C++, dentre muitas outras.

Função de API	Significado
VirtualAlloc	Reserva ou compromete uma região
VirtualFree	Libera ou descompromete uma região
VirtualProtect	Altera a proteção ler/escrever/executar em uma região
VirtualQuery	Consulta o estado de uma região
VirtualLock	Transforma uma região da memória em residente, isto é, desabilita paginação para ela
VirtualUnlock	Torna uma região paginável do modo normal
CreateFileMapping	Cria um objeto de mapeamento de arquivo e lhe designa (opcionalmente) um nome
MapViewOfFile	Mapeia (parte de) um arquivo para o espaço de endereço
UnmapViewOfFile	Remove um arquivo mapeado do espaço de endereço
OpenFileMapping	Abre um objeto de mapeamento de arquivo previamente criado

Figura 5 - Exemplo de API para acesso à memória virtual no Windows, este conjunto parcial de API é provido pelo Windows e utilizado pelos aplicativos.

Fonte: Tanenbaum; Bos, 2016, p. 647.

Assim, percebe-se que o SO, muito além de gerenciar o microcomputador, também fornece funcionalidades que por vezes não estão presentes no *hardware*. Essas funcionalidades são relativas aos serviços que o SO fornece e que podem ou não estarem ligados ao *hardware*. Essas funcionalidades por vezes são referenciadas como **funcionalidades de software**, a exemplo podemos citar o uso por vários usuários simultâneos (multiusuário) via rede; os serviços de autenticação de usuário; os serviços de proteção de dados (uso de senha; encriptação; criação de pastas; criação de arquivos etc). Há um conjunto amplo dessas funcionalidades de *software* providas pelo SO, que não são relacionadas ao *hardware*, mas que de alguma forma melhoram o seu acesso, ou até incluem características diferentes de uso.

Dessa forma, podemos afirmar que um bom SO melhora o equipamento, enquanto um SO ruim irá piorar o desempenho da máquina.

Veja que, na Figura com o exemplo de API, as chamadas de sistema são acessadas usando as APIs, por meio da linguagem de programação de seus comandos próprios. Em C, por exemplo, utiliza-se *malloc* para solicitar memória. Internamente, na linguagem, usa-se uma das APIs do SO para solicitar memória. Por exemplo, no Windows tais APIs possuem nomes que são diferentes dos nomes usados no Linux. Assim, o encadeamento correto é: programa do usuário - comandos da linguagem de programação - APIs - chamada de sistema. Não se preocupe em decorar esse grande encadeamento, ele ficará cada vez mais claro ao longo da disciplina.

Já na próxima Figura, podemos ver um fragmento de código fonte de um aplicativo com finalidade de realizar a cópia de um arquivo. O fragmento está escrito na linguagem de programação C e é direcionado para ser

executado no SO Linux. Uma abrangente discussão sobre a programação na linguagem C pode ser obtida na obra de Deitel (2011).

```
/* Abre os descritores de arquivo. */
infd = open("data,0);
outfd = creat ("newf", ProtectionBits);

/* Laço de cópia. */
do {
    count = read(infd, buffer, bytes);
    if (count > 0) write(outfd, buffer, count);
} while (count > 0);

/*Fecha o arquivo. */
close(infd);
close(outfd);
```

Figura 6 - Exemplo de fragmento de código-fonte em linguagem C que representa a invocação de uma chamada de sistema.

Fonte: Tanenbaum; Bos, 2016, p. 388.

No exemplo, *open*, *creat*, *read*, *write* e *close* são comandos da linguagem C. Internamente na linguagem C, quando usados no SO Linux, por exemplo, serão redirecionados para a API *syscall()*. Esta API fará, enfim, a chamada de sistema correspondente, ficando parecida como “*syscall(0x5)*” estivesse solicitando a chamada de sistema de abertura de arquivo, cujo número de chamada no sistema de macros do Linux fosse 0x5 e apontará para o trecho de código que pertence ao *kernel* do SO. No Linux, a relação das chamadas de sistema pode ser vista no arquivo localizado em: */usr/include/sys/syscall.h*.

VOCÊ QUER LER?



O conjunto de chamadas de sistema disponíveis pelo SO é vasto e permite acesso a recursos físicos e de *software*. Uma ampla relação de chamadas de sistema para SOs baseados no UNIX pode ser visto e acessado em (MELO NETO, 2014): <<http://www.ime.usp.br/~adao/teso.pdf>>.

Não se preocupe se você achou muito técnica esta seção, o objetivo principal é que você compreenda os mecanismos envolvidos e não aprender a fazer uso deles imediatamente. Em seguida, vamos conhecer a simulação de processos.

1.4.2 Simulação de processos

Você sabe em que consiste uma simulação de processos? É um aplicativo que permitisse a simulação de execução de processos e experimentasse (e compreendesse) de perto a dinâmica de um gerenciador de processos. Para realizar essa tarefa é necessário que se estabeleçam as seguintes condições:

- a) um sistema que permita a criação de tarefas, associando necessidades de recursos para essas tarefas;
- b) um mecanismo de fornecimento de recursos para atender as tarefas;
- c) um escalonador que gerencie os estados das tarefas, alternando aqueles que estarão em execução e ou nos demais estados.

Claro que tudo isso como uma simulação.

VOCÊ QUER LER?



O gerenciador de processos é um componente muito sofisticado do SO. Uma leitura tanto didática quanto técnica pode ser encontrada no livro “Sistemas Operacionais com Java” (SILBERSCHATZ; GALVIN; GAGNE, 2004). Ele é voltado para a linguagem Java e repleto de exemplos de código-fonte.

Existem muitos simuladores de processos, um deles é o que é apresentado em Carvalho et al. (2016), no qual é possível satisfazer parte das condições elencadas anteriormente, se tratando, portanto, de uma ferramenta mais simples que pode ser utilizada sem instalação (direto na *web*). Já um simulador completo que permite a escolha de alguns tipos de algoritmos de escalonamento, visualização do PCB, foi proposto por Machado, Maia e Pacheco (2005). Os autores, além de proporem a ferramenta, também criaram uma versão educacional para instalação em *desktop*, o SOsim.

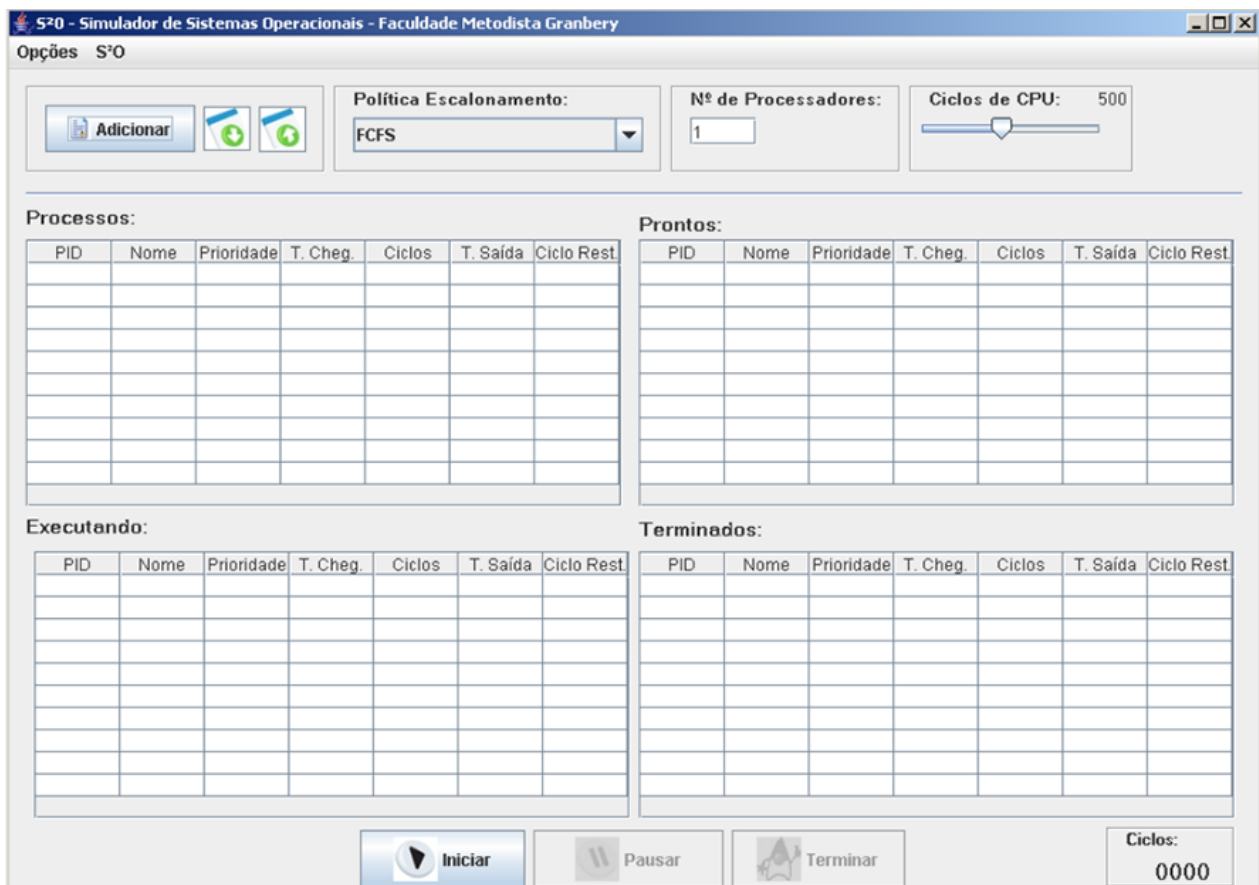


Figura 7 - Uma das interfaces do S20, simulador simples de processos, informando os processos e seus respectivos estados.

Fonte: Carvalho et al., 2016, p. 4.

Ao utilizar este simulador simples, é possível experimentar diversas situações nas quais a prioridade, dentre outros fatores, podem ser ajustadas para a simulação. Como vimos na Figura anterior, há três diferentes estados na simulação: prontos, executando e terminados. A fila de execução pode ser modificada pelo usuário, bem como as prioridades de cada processo. Note que a simulação de processos é uma boa ferramenta para termos uma percepção de como ocorre a dinâmica na qual o SO está profundamente envolvido por meio de seu escalonador de tarefas.

E com isso, você agora tem o conhecimento para compreender como os mecanismos que compõem o SO interagem, e como essa dinâmica afeta o resultado final para o usuário. Todos os conceitos apresentados neste capítulo são essenciais para compreensão do tema.

Síntese

Tivemos, neste capítulo, um contato inicial com o sistema operacional, conhecendo os conceitos mais importantes sobre o tema. Pudemos compreender o funcionamento das principais partes que compõem o SO e como elas interagem.

Neste capítulo, você teve a oportunidade de:

- conhecer os conceitos iniciais e fundamentais sobre o sistema operacional e sobre o seu funcionamento;
- conhecer as partes integrantes do SO e o modo como interagem;
- aprender a divisão de espaços de execução, uma para o núcleo e uma para o usuário;

- compreender que um SO é composto de várias partes associadas em um único bloco (monolítico) ou em camadas, compreender que os processos, durante o seu ciclo de vida, podem assumir diferentes estados, controlados pelo gerenciador de processos;
- conhecer o gerenciador de memória, responsável pelo controle da memória real e a memória virtual do microcomputador;
- compreender que para as aplicações terem acesso ao SO é necessário usar o conjunto de chamadas de sistema, acessado via linguagem de programação, na forma de APIs.

Bibliografia

- BURKE, M. **Os Piratas da Informática: Piratas do Vale do Silício**. Direção: Martyn Burke. Produção: Leanne Moore. EUA, 1999.
- CARVALHO, D. et al. Simulador para a prática de Sistemas Operacionais. **Revista Eletrônica da Faculdade Metodista Granbery**, Juiz de Fora, v. 1, 2006. Disponível em: <<http://re.granbery.edu.br/artigos/MjQx.pdf>>. Acesso em: 17/05/2018.
- DEITEL, H. **Sistemas Operacionais**. 3.ed. São. Paulo: Pearson, 2005.
- _____; DEITEL, P. J. C. **como programar**. 6. ed. São Paulo: Pearson Prentice Hall, 2011.
- MACHADO, F. B.; MAIA, L. P. **Fundamentos de Sistemas Operacionais**. Rio de Janeiro: LTC, 2011.
- _____; _____. PACHECO A. A constructivist framework for Operating Systems Education: a pedagogic proposal using the SOsim. In: **10th Annual Conference on Innovation and Technology in Computer Science Education (ITCSE)**, Universidade Nova de Lisboa, Portugal, p. 27-29, jul 2005.
- MAZIERO, C. A. **Sistemas Operacionais: Conceitos e Mecanismos**. Curitiba: UFPR, 2017. Disponível em: <<http://wiki.inf.ufpr.br/maziero/lib/exe/fetch.php?media=so:so-livro.pdf>>. Acesso em: 17/05/2018.
- MELO NETO, A. **Estrutura dos Sistemas Operacionais**. USP. 2014. Disponível em: <<https://www.ime.usp.br/~adao/teso.pdf>>. Acesso em: 17/05/2018.
- NEMETH, E.; SNYDER, G.; HEIN, T. R. **Manual Completo do Linux: Guia do Administrador**. 2. ed. São Paulo: Pearson, 2007.
- SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, R. **Sistemas Operacionais com Java**. 6. ed., Rio de Janeiro: Campus, 2004.
- TANENBAUM, A. S.; BOS, H. **Sistemas Operacionais Modernos**. 4. ed. São Paulo: Pearson Education do Brasil, 2016.