SISTEMAS OPERACIONAIS

CAPÍTULO 2 - É POSSÍVEL CONTROLAR O SISTEMA OPERACIONAL DEFININDO TAREFAS E PRIORIDADES?

Osvaldo de Souza

Introdução

O sistema operacional é um *software* extremamente complexo. Possui muitas interfaces que o conectam com o usuário, com o meio externo, por intermédio das redes de comunicação de dados, como as conexões discadas e as sem fio, que permitem a qualquer microcomputador se manter conectado a outros equipamentos pela internet.

Consideramos a internet como um grande repositório de dados e informações, como imagens, filmes, textos e games. Fazemos uso de todos esses recursos e temos a tendência de nos esquecermos de que por trás de todo esse conjunto enorme de recursos e facilidades há uma enorme quantidade de microcomputadores que funcionam graças a seus respectivos sistemas operacionais.

Embora seja comum ouvir que a internet é uma grande rede de computadores, não haveria nada de errado - na verdade, seria muito justo - dizermos que ela é uma grande rede de sistemas operacionais, pois sem eles não haveria nada. Pois isso nos faz pensar: sendo o SO tão relevante e essencial, como controlá-lo e estabelecer as prioridades de execução? Como um usuário pode interagir diretamente com o SO? Aliás, isso é possível?

Neste capítulo, vamos acompanhar de perto o funcionamento do SO, inclusive os detalhes do mecanismo utilizado para criar e gerenciar os processos. Detalhes estes que chegam ao nível da linguagem de programação utilizada para "falar" com o SO no sentido de criar tais processos. Veremos também a estrutura e os detalhes da fila de processos e vamos abordar o complexo ambiente de funcionamento do escalonador de tarefas nas gerências dos processos. Assim, vamos compreender mais sobre a complexa troca de contexto durante o "vai-e-vem" de diferentes processos na CPU do microcomputador.

Ainda, vamos examinar com detalhes o algoritmo escalonador de tarefas conhecido por SJF-preemptivo, compreendendo seu funcionamento e a importância da escolha de um bom algoritmo para essa função. Por fim, vamos adentrar no complexo mundo da comunicação que ocorre entre os processos e o sistema operacional, analisar as opções para o controle dessa comunicação e a real importância em todo o funcionamento do SO, na solução de problemas de competição por recursos (concorrência) e o avançadíssimo conceito de programação concorrente.

Acompanhe a leitura e bons estudos!

2.1 A linguagem de programação e o SO

Obviamente, os sistemas operacionais não se comunicam com os processos, tampouco com o usuário, utilizando-se da linguagem humana (DEITEL; DEITEL, 2011). De fato, a capacidade de fala humana é tão complexa que ainda hoje existem vários estudos e pesquisadores com a finalidade de desenvolver habilidades de compreensão e fala da linguagem humana para microcomputadores. Tais estudos estão no campo do processamento da linguagem natural e são muito interessantes.

VOCÊ QUER LER?

O processamento de linguagem natural é um instigante tópico de pesquisa e que tem aplicações em diversas áreas da computação, em especial na interação homem *versus* máquina. O objetivo de tais pesquisas é de dotar os aplicativos, e até mesmo funções relacionadas aos serviços do SO, a comunicarem-se com o usuário através da linguagem natural humana. E não apenas para receber comandos, mas também para interagir em plena conversação com o usuário. Um texto interessante para ler sobre o assunto é o capítulo "Processamento da linguagem natural – algumas noções para aprimorar o tratamento de estruturas com predicado secundário" (CONTERRATTO, 2015) do livro "Fundamentos Linguísticos e Computação" (IBAÑOS; PAIL, 2015).

Seria muito interessante utilizar um sistema operacional que permitisse ao usuário falar com o SO, não acha? De forma que fosse menos necessário o uso de mouse e de teclado, por exemplo, para a criação de textos. Mas enquanto não podemos falar com os sistemas operacionais, como podemos nos comunicar com eles? De que forma podemos definir conjuntos de ações que devam ser realizados periodicamente pelo sistema operacional? Há uma maneira formal e processual para criar aplicativos para serem executados no microcomputador pelo sistema operacional? As respostas para todas essas perguntas podem ser resumidas em: por meio do uso de linguagens de programação para computadores.

2.1.1 Linguagem de programação

Linguagem de programação para computadores (ou simplesmente linguagem de programação) é um conjunto de palavras-chave (instruções) que são utilizadas para compor sequências de comandos, que uma vez executados na ordem definida, produzem um resultado planejado. Existem tipos de linguagens de programação de baixo nível e de alto nível. Em Deitel e Deitel (2011), por exemplo, podemos ver uma ampla discussão em torno da linguagem de programação C, que pode ser utilizada tanto em baixo nível, como em alto nível.
Mas vamos definir o que são essas linguagens. Linguagens de baixo nível são aquelas que lidam diretamente com questões associadas ao *hardware*. Tanenbaum e Bos (2016) nos dizem que:

> é claro que nenhum programador são iria querer lidar com esse disco em nível de *hardware* (...). Por essa razão, todos os sistemas operacionais fornecem mais um nível de abstração para se utilizarem discos: arquivos. Usando essa abstração, os programas podem criar, escrever e ler arquivos, sem ter de lidar com os detalhes complexos de como o hardware realmente funciona (TANENBAUM; BOS, 2016, p. 3).

Assim fica claro que a linguagem de baixo nível está associada a questões tecnicistas e programas feitos nessas linguagens em geral são para um *hardware* específico. Se você usa uma linguagem de baixo nível para escrever um programa que permitirá que o usuário digite um texto no teclado do fabricante X, esse programa não funcionará se o teclado for da marca Y.
Desse modo, usando linguagem de baixo nível produzimos os *drivers* e o próprio sistema operacional. Alguns utilitários são feitos com linguagens de baixo nível, como as linguagens *Assembly* e a C.
Já as linguagens de alto nível são mais apropriadas para a construção de aplicativos diretamente usados pelo usuário, ou para aquelas que não lidam com questões de *hardware*. Um programa para internet, como um

navegador *web*, é feito em linguagem de alto nível. Mas mesmo os aplicativos (e as linguagens de alto nível) do usuário utilizam o *hardware*. Então, como é possível que uma linguagem de alto nível faça acesso ao *hardware*, visto que esse tipo de linguagem não é próprio para tal feito? A resposta para essa pergunta chama-se abstração. Na Figura a seguir, você pode ver um esquema para o conceito de abstração.

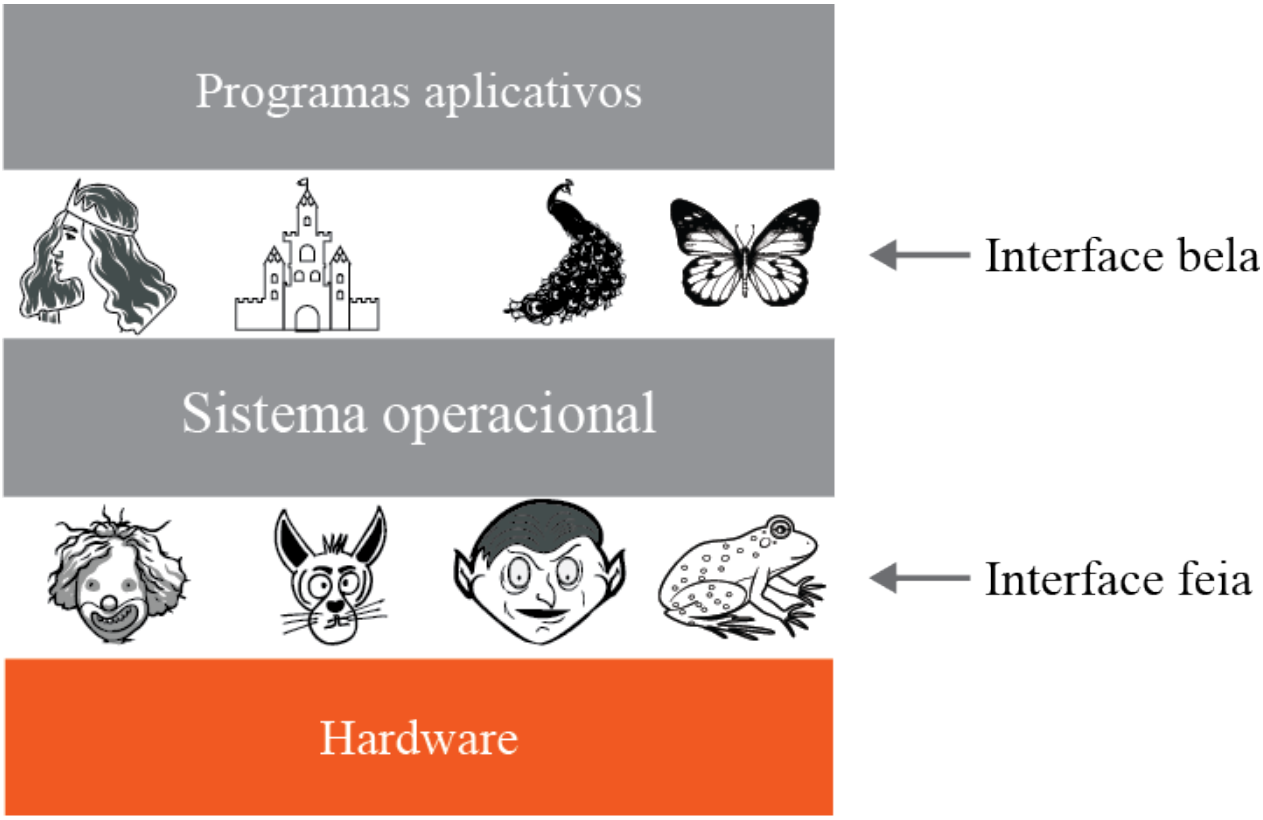

Figura 1 - O conceito de abstração na visão de Tanenbaum e Bos é um processo que esconde o hardware feio para uma forma de apresentação e uso bonitas.
Fonte: TANENBAUM; BOS, 2016, p. 3.

Na metáfora expressa na Figura anterior, compreendemos que a partir da abstração é possível esconder os detalhes técnicos de um *hardware* e tornar seu uso mais fácil, ao migrar de uma interface feia para uma interface bela. O conceito de feio e belo usado por Tanenbaum e Bos (2016) diz respeito ao feio ser difícil e técnico, e o belo ser fácil e pouco técnico.

A abstração é, portanto, uma maneira que nos permite esconder detalhes técnicos que são requeridos, mas os quais não deveríamos nos preocupar, e deviam ser feitos automaticamente, para assim nos concentramos no que realmente importa.

Vamos a outro exemplo. Ao dirigir um automóvel ou uma motocicleta, você usa o acelerador para que o veículo vá mais rápido ou mais devagar, mas você não se preocupa com a quantidade de combustível (gasolina) e de comburente (oxigênio) que precisa injetar no motor para que ele produza o resultado que você deseja.

Figura 2 - A tecnicidade do abastecimento de combustível e comburente em veículos é um exemplo do processo de abstração, isto é, da maneira de esconder detalhes que, embora necessários, podem ser resolvidos de outra forma.
Fonte: Shutterstock, 2018.

Assim, temos que as questões técnicas sobre o equilíbrio entre combustível e comburente, que são feias e técnicas, estão escondidas (abstraídas) de você em uma interface bonita e amigável, o acelerador.
Compreendido o processo de abstração, agora podemos concentrar nosso foco em como o SO mantém comunicação com os aplicativos ou, em outras palavras, como usamos a linguagem de programação para criar aplicativos que possam interagir com o SO.

VOCÊ QUER VER?

Talvez você já tenha ouvido falar do carro autônomo da empresa Tesla. Não ouviu? Pois bem, trata-se de um veículo que promete dirigir por você, mas, na verdade, é o SO quem dirige. Basta indicar onde você deseja ir e ele fará todo o trabalho de enfrentar o trânsito. Vários depoimentos de pessoas que já passaram pela experiência de usar este veículo afirmaram que é muito bom e transmite uma enorme confiança ao passageiro. Confira no vídeo <https://www.tesla.com/autopilot> (TESLA, 2018) a experiência de ver como um carro conduzido por um SO se comporta em diversas situações.

O sistema operacional produz uma abstração enorme que esconde o *hardware* e o disponibiliza aos programadores de computador por meio de um conjunto de chamadas de sistema, o qual é utilizado pelas linguagens na produção de APIs, que são as interfaces programáveis usadas pelos aplicativos. Na Tabela a seguir, podemos ver uma parte da relação de APIs em um comparativo entre o Windows e sistemas Unix.

Unix	Win32	Descrição
fork	createProcess	Cria um novo processo
Waitpid	Waitforsingleobject	Pode esperar que um processo termine
Execve	(nenhum)	CreateProcess = fork + execve
Exit	Exitprocess	Conclui a execução
Open	Createfile	Cria um arquivo ou abre um arquivo existente
Close	Closehandle	Fecha um arquivo
Read	Readfile	Lê dados a partir de um arquivo
Write	Writefile	Escreve dados em um arquivo
Lseek	setfilePointer	Move o ponteiro do arquivo
Stat	getfileattributesEx	Obtém vários atributos do arquivo
Mkdir	createDirectory	Cria um diretório (pasta de arquivos)
Link	(nenhuma)	Win32 não dá suporte a ligações
Unlink	DelefeFile	Destrói um arquivo existente
Mount	(nenhum)	Win32 não dá suporte a mount
Umount	(nenhum)	Win32 não dá suporte
Chdir	setcurrentDirectory	Altera o diretório de trabalho atual
Kill	(nenhum)	Win32 não dá suporte a sinais
Time	Getlocaltime	Obtém o tempo atual

Tabela 1 - Com o comparativo entre APIs é possível notar que há grande similaridade entre as APIs providas pelo Win32 e Unix.
Fonte: TANENBAUM; BOS, 2016, p. 43.

Quando um programador deseja criar um aplicativo que, por exemplo, crie uma nova pasta no disco rígido, ele teria de produzir algo parecido como a seguinte sequência de comandos da linguagem C++:

```
#include<iostream>
int main()
{
    system("mkdir \"pasta\documentos\"");
    return 0;
}
```

Todavia, se a linguagem utilizada fosse Java, ficaria algo como:

```
public class NewClass {

    public static void main(String[] args){
File diretorio = new    File("\\pasta\\documentos");
            diretorio.mkdir();
    }
}
```

O acesso às outras interfaces de programação previstas na API do SO é chamado de maneiras semelhantes aos dois trechos de linguagem de programação apresentados. E em alguns casos o uso é até mais fácil do que o modo demonstrado nos exemplos.

2.1.2 Threads

Um aplicativo normalmente é desenvolvido como uma sequência de comandos que são executados para realizarem uma determinada tarefa. Estamos falando de algo como o que ilustra a Figura a seguir.

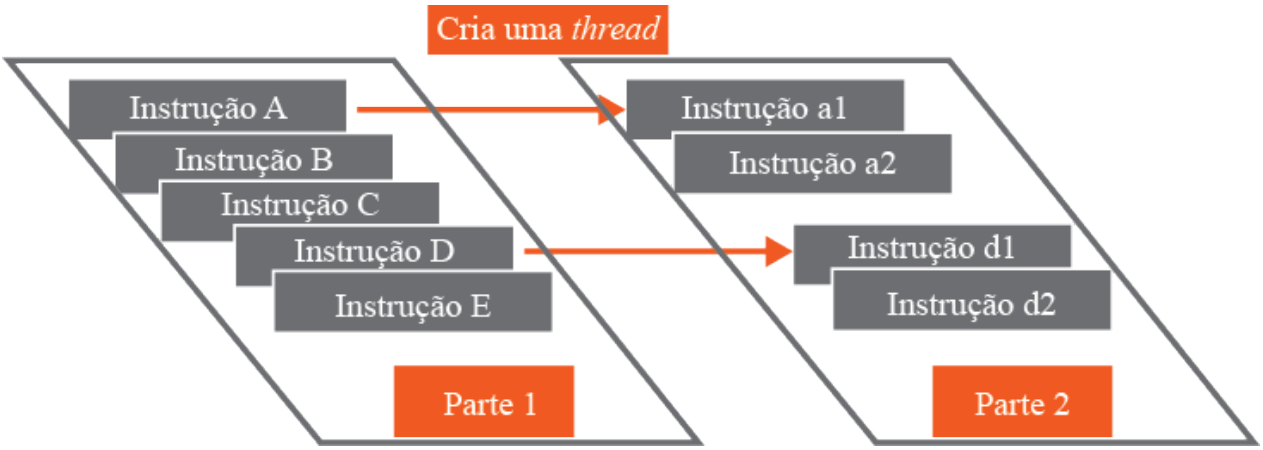

Figura 3 - Sequência de execução de um programa (1) e as threads (2), em que na Parte 1 é ilustrado um programa sendo executado sequencialmente (A->E) e ao lado, as threads.
Fonte: Elaborado pelo autor, 2018.

Pela Figura anterior, observe que na Parte 1 temos um programa composto de 5 instruções (A, B, C, D, E) que são executadas sequencialmente, isto é, primeiro A, depois B, e assim sucessivamente. Ocorre que, eventualmente, pode-se usar um recurso de programação chamado de *thread*, que constitui no primeiro passo em direção à

programação concorrente e utilizada quando desejamos que um programa tenha duas ou mais linhas de execução simultâneas.

Uma linha de execução corresponde a uma sequência de comandos que estão executados, sendo que a linha principal é aquela que deu início ao programa e é a tarefa principal para o SO. As *threads* são ramais da linha principal, utilizadas, normalmente, para realizar alguma atividade que seja complementar ao processamento que está sendo executado na linha principal.

Ainda sobre a Figura anterior, observe que a linha principal é dividida em duas secundárias (*threads*), uma no momento em que está sendo executada a instrução A, e outra quando está sendo executada a instrução B. Quando uma *thread* chega ao fim, a linha principal não é encerrada, ela continua a ser executada normalmente.

VOCÊ QUER LER?

As *threads* abrem caminho para a construção de programas que podem ser executados mais rapidamente, pois um programa pode ser subdividido em várias CPUs. Todavia, o melhor uso desse potencial é conseguido quando o programa é desenvolvido pensando no uso dessa abordagem. Para que isso seja possível é necessário que o programador entenda e compreenda a problemática da programação concorrente ou paralelizada. Um ponto de partida para essa abordagem pode ser encontrado no texto "Programação Paralela e Distribuída em Java" (BARCELLOS, 2002) que apresenta a problemática associada ao uso da linguagem de programação Java. Confira o texto em: <http://www.lbd.dcc.ufmg.br/colecoes/erad-rs/2002/007.pdf>.

Threads podem ser usadas para "apressar" o processamento, quando o *hardware* no qual o programa está sendo executado possui mais de uma CPU. Nesse caso é possível indicar à CPU principal que as *threads* devam ser executadas em diferentes CPUs do microcomputador, estabelecendo-se o conceito de paralelismo, no qual a execução de um processo é fragmentada (de maneira planejada e controlada) em diversos segmentos que são executados simultaneamente nas diferentes CPUs do microcomputador.

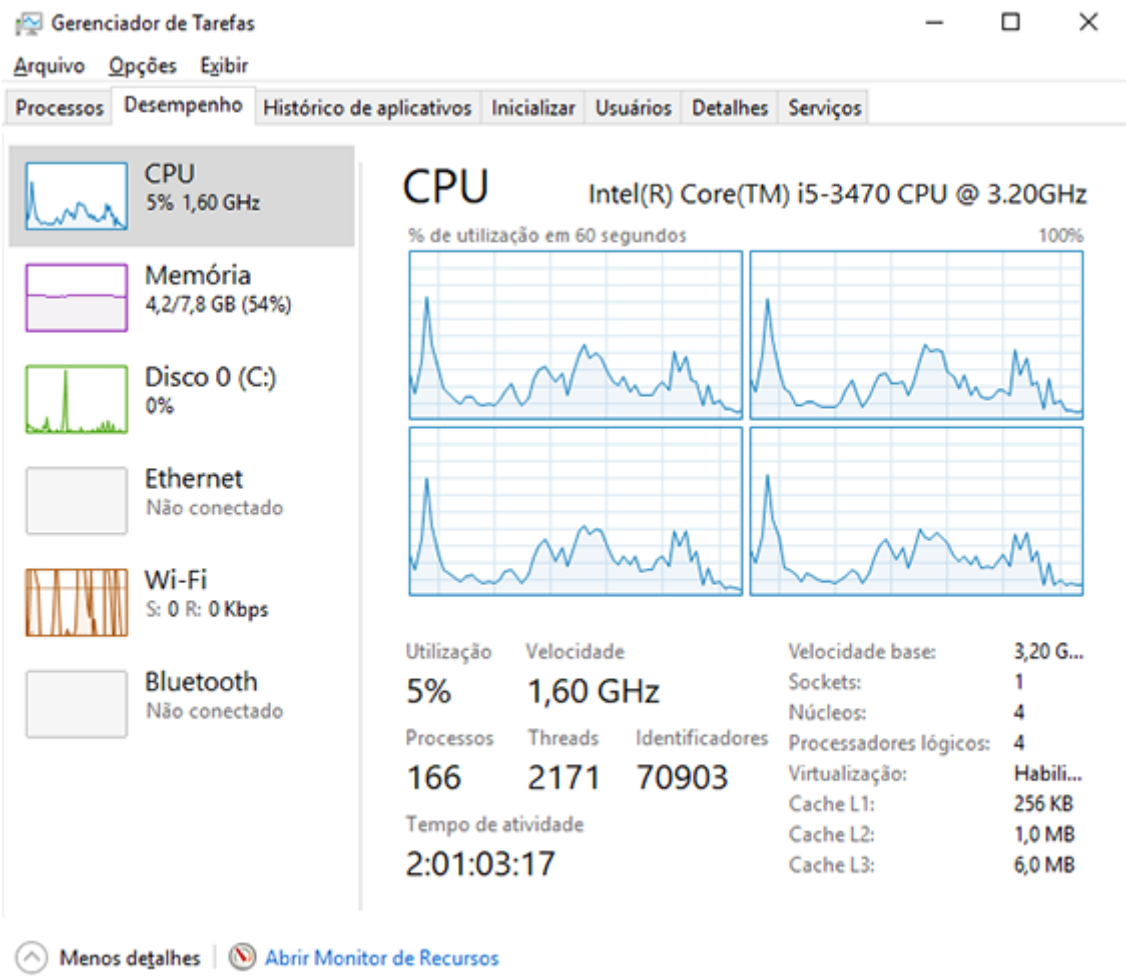

Figura 4 - Visão das CPUs no monitor de recursos do Windows apresenta quatro gráficos, correspondentes a cada uma das CPUs do microcomputador.
Fonte: Printscreen do Microsoft Windows, 2018.

Observe na Figura anterior que no momento da captura da imagem do monitor de recursos do Windows, havia 166 processos em execução que geraram 2171 *threads*.

VOCÊ SABIA?

A primeira versão do sistema operacional da Microsoft não foi criada pela empresa, mas sim pela *Seattle Computer Products*, por 50 mil dólares, e o SO se chamava Q-DOS. A compra ocorreu devido a Microsoft ter firmado um acordo de fornecimento com a IBM, que pretendia lançar um novo microcomputador e ainda não possuía um sistema operacional. A solução envolveu realizar modificações para que o SO passasse a ser o conhecido MS-DOS da Microsoft.

Note que as *threads* são amplamente utilizadas e que os gráficos relativos ao trabalho realizado por cada uma das 4 CPUs do microcomputador, no qual foi realizada a captura para a imagem, embora sejam parecidos, há diferenças no histórico do processamento. Você consegue achar alguma diferença? E ainda sobre o assunto, você sabe como criar *threads*?

2.1.3 Modelos de Criação de Threads

A criação de *threads* de um processo no sistema operacional não é uma tarefa difícil, de fato em várias linguagens a tarefa é bem simples. O trecho de código-fonte, obtido na documentação do Java, listado a seguir, ilustra esse processo:

```
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Ola, estou rodando!");
    }
    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }
}
```

Já o correspondente na linguagem C seria algo assim:

```
pthread_t tid[2];
void* doSomeCode(void  *arg)
{
    unsigned long x = 0;
    pthread_t id = pthread_self();
    if(pthread_equal(id,tid[0])){
        printf("\n OK, a primeira thread estah rodando\n");
    }else{
        printf("\n OK de novo!, sinal de vida da segunda thread\n");
    }
    for(x=0; x<(0xFFFFFFF);x++); // apenas para aguardar.
    return NULL;
}
int main(void)
{
    int err;
    int j = 0;
    while(j < 2)    {
        err = pthread_create(&(tid[i]), NULL, &doSomeCode, NULL);
        if (err != 0)
            printf("\n Falha na criação da thread:[%s]", strerror(err));
        else
            printf("\n OK, Thread criada sem problemas\n");
        j++;
    }]
    sleep(4);
    return 0;
}
```

Generalizando os dois exemplos de código mostrados anteriormente, o modelo geral para criação de *threads* pode ser resumido como:

- criar o trecho de código de execução que representa a *thread*. Este trecho deve realizar a tarefa que é necessária ser "paralelizada". A execução irá ocorrer (em geral) independente do estado de execução da linha principal. Assim, a linha principal (o processo principal) pode estar em estado de "aguardando", enquanto uma de suas *threads* pode estar em "executando", outra em "encerrado";
- criar um trecho de código de execução que represente a linha principal. Em geral, pode ser resumido como um grande *loop* (laço de execução) que dá início as *threads*, e toma providências quando alguma delas é encerrada, ou surge alguma condição (ou solicitação) externa que faz com que uma nova *thread* seja necessária. Nesta situação, a linha principal dará início a uma nova *thread*.

Uma observação relevante se refere ao fato de que o processo é o ponto no qual os recursos são solicitados, portanto a problemática de solicitação de recursos cabe ao processo, enquanto a *thread* seria destinada exclusivamente ao trabalho que requer exclusivamente o tempo da CPU.

Uma observação relevante se refere ao fato de que o processo é o ponto no qual os recursos são solicitados, portanto a problemática de solicitação de recursos cabe ao processo, enquanto a *thread* seria destinada exclusivamente ao trabalho que requer exclusivamente o tempo da CPU. Se uma abordagem de *thread* for realizada em um microcomputador com uma única CPU, pode não haver ganho significativo no desempenho do processo, todavia, em máquinas com múltiplas CPUs esse ganho será significativo.

VOCÊ SABIA?

Existe a promessa de que os computadores quânticos irão romper com o problema de paralelismo, tornando os atuais computadores ou micros, obsoletos e lentos. Neill, et al. (2018) apresenta um cenário no qual demonstra a supremacia da computação quântica. A questão não é mais, se é possível e realizável, e sim quando ficará disponível. Com a computação quântica tornando-se realidade novos caminhos se abrem para o processamento de aplicativos, tornando as preocupações de tempo de processamento algo do passado, devido a absurda velocidade desses computadores.

Tanenbaum e Bos (2016), enfatizando as vantagens do uso de *threads*, fazem uma analogia da situação para uma aplicação de edição de texto. Se o processador for construído em duas partes, uma interativa e que mantém contato com o usuário, e a segunda, que faz o trabalho de formatação do texto, poderá haver grande benefício para o usuário. Caso o usuário faça uma alteração em uma página, em uma parte inicial de um livro, a *thread* que trabalha na formatação trabalharia para reformatar o livro. Assim, quando o usuário se mover pelas páginas do livro (as páginas que ficam depois da alteração), já vai encontrá-las corrigidas, tudo graças ao trabalho feito em paralelo pela *thread* interativa.

VOCÊ O CONHECE?

Na época do registro do primeiro programa de computador, coube à matemática e escritora inglesa Ada Lovelace a autoria do que é considerado o primeiro algoritmo para um programa de computador. O feito ocorreu em 1843 quando Ada se envolveu com o matemático britânico Charles Babbage e o trabalho de Luigi Frederico Menabrea (FUEGI; FRANCIS, 2003). Quer conhecer mais sobre Ada Lovelace? Leia em: <https://www.scss.tcd.ie/coghlan/repository/J_Byrne/A_Lovelace/J_Fuegi_&_J_Francis_2003.pdf>.

Diante do que vimos até o momento, pode surgir a pergunta: mas como definir qual processo será realizado primeiro? Existem prioridades para isso? Vamos entender isso melhor no tópico a seguir.

2.2 Definindo prioridades na fila de processos do SO

Nos SOs que utilizam estratégias de gerenciamento baseadas em prioridades, é possível modificar, arbitrariamente, a fila de prioridades. Essa possibilidade é atraente, pois o usuário pode desejar que um determinado processo relativo a um aplicativo iniciado por ele tenha uma prioridade diferenciada. De maneira geral, todas as estratégias de gerenciamento deveriam prever certos grupos de prioridades, caso contrário pode ocorrer a interrupção de tarefas.

CASO

João tem um trabalho escolar importante para realizar e embora já tenha começado com antecedência, ainda não conseguiu concluí-lo. Ele para com o trabalho, visto que já está bem cansado, faz um lanche, toma banho e decide assistir 15 minutos de um vídeo na internet. Ao todo, João ausentou-se do microcomputador por 45 minutos. Ao retornar para terminar o trabalho, qual foi sua (desagradável) surpresa que sem consultá-lo sobre a adequação do momento, o que sequer se era de seu interesse, o SO havia iniciado uma atualização do sistema. A atualização estava em cerca de 4% quando João retornou da pausa de descanso e pela demora até aquele ponto, ele estimou que teria de esperar, ao menos, mais 2 horas até que o microcomputador estivesse disponível novamente. Nesta situação, é obvio que o SO não priorizou a tarefa que João estava realizando, pois o correto seria pausar a atualização (ou mesmo cancelá-la), permitindo terminar a tarefa e somente quando João indicasse sua permissão a atualização deveria continuar. É um caso típico em que o SO erra na priorização das tarefas e ainda não permite que o usuário corrija o que ele entende como prioridade.

Existem outros cenários em que a concorrência pelo processamento acontece como consequência dos múltiplos aplicativos que estão sendo executados por ação do usuário, tais como, quando o usuário inicia um processador de textos e um aplicativo de música. Considere que em dado momento, o SO deve controlar a execução de um programa de acesso à internet, como um navegador *web*, e que o usuário está acessando um recurso multimídia, do tipo YouTube. Durante esse mesmo período, o usuário aciona outro aplicativo para a reprodução de música, também através da internet. Temos um cenário em que o SO terá de lidar com dois processos distintos que estão usando recursos de *software* e de *hardware* ligados ao acesso à rede, à reprodução de áudio, e à reprodução de vídeo. Perceba que há concorrência pelos mesmos recursos críticos para as duas aplicações (processos): a) a rede: para poder obter as sequências do fluxo de vídeo e de áudio; b) a reprodução de áudio: ambas as aplicações, navegador e reprodução de música, necessitam reproduzir áudio. A prioridade deve ser dada ao aplicativo de acesso ao YouTube ou aquele que está reproduzindo música?

Em todos os casos, independentemente de como a prioridade esteja sendo tratada, isto é, do algoritmo de escalonamento que foi escolhido, é necessário que o SO permita que o usuário altere a prioridade para certos processos de seu interesse (TANENBAUN; BOS, 2016).

No Windows, uma forma fácil para o usuário fazer isso diretamente é com a utilização do monitor de recursos, utilizando-se a aba “Detalhes”, como se vê na Figura a seguir.

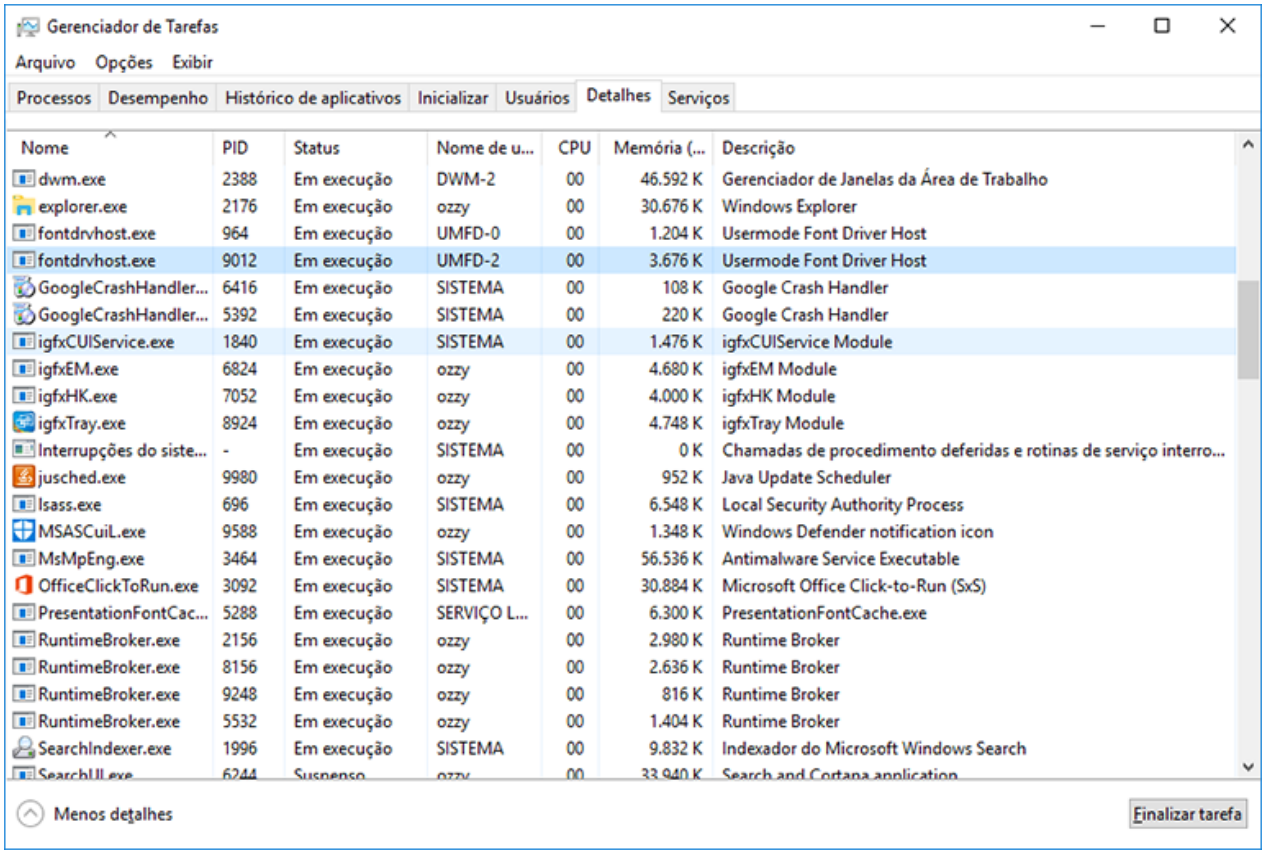

Figura 5 - Com a visão dos processos que estão em execução, o usuário pode usar este aplicativo para alterar arbitrariamente a prioridade de algum processo.
Fonte: Printscreen no Microsoft Windows, 2018.

No Linux, pode-se utilizar o utilitário *top*, que exibe informações semelhantes ao que podemos ver na Figura anterior (NEMETH; SNYDER; HEIN, 2007). Quando a alteração da prioridade dos processos não é feita utilizando tais tipos de aplicativos, é necessário realizar a ação por intermédio da programação, e neste caso, utiliza-se uma API para fazer a chamada de sistema correspondente à alteração de prioridade.

2.2.1 Escalonamento (FIFO e prioridades)

Até este ponto, você já percebeu o quanto o SO é relevante para o bom funcionamento e uso dos microcomputadores, não é mesmo? De fato, sem o SO, usar um microcomputador seria uma tarefa totalmente técnica. Lembrando-se sempre de que outros tipos de equipamentos, além dos microcomputadores, também têm o funcionamento facilitado pelo uso de algum tipo de SO. Dentre as diversas tarefas que o SO realiza, a responsabilidade de selecionar qual processo (dentre os muitos que possam existir) deve ser executado em primeiro, segundo etc. é certamente a mais relevante para a percepção do bom funcionamento por parte dos usuários do microcomputador.

A decisão sobre qual processo deve ser executado recebe o nome de escalonamento, existindo, portanto, os algoritmos de escalonamento. Mas qual o motivo para existirem diferentes algoritmos de escalonamento? Em primeiro lugar, é preciso esclarecer que alguns dos algoritmos de escalonamento têm função acadêmica, no contexto de SO, pois podem não ser boas escolhas para o tipo de tarefa que os microcomputadores possuem atualmente.

Antes, contudo, vamos rever um conceito importante chamado de preempção. Um dos significados da palavra é o de preferência, todavia, para os algoritmos de escalonamento, preemptivos são aqueles nos quais o processo que

está em execução pode ser suspenso a qualquer momento, já os não preemptivos, uma vez iniciados, serão executados até o fim. Perceba que pela existência da preempção, há motivos pelos quais o processo que está em execução possa ser suspenso. E quais seriam esses motivos?

O primeiro deles é quando ocorre uma interrupção. Uma interrupção no contexto de SO refere-se a uma sinalização de *hardware* ou de *software* (no caso o *software* será uma das partes do SO) que indicam à CPU que um evento de prioridade ocorreu em alguma parte do *hardware* do microcomputador, ou em alguma parte prioritária do código do SO. O termo técnico para uma interrupção é requisição de interrupção, do inglês *interrupt request* (IRQ). Quando uma interrupção é sinalizada, o processo que estava em execução é suspenso, a CPU irá então executar o trecho de código que irá atender a IRQ. Este trecho de código está no espaço do núcleo e foi posto na memória pelo SO e pertence ao SO. Quando o trecho de código que atende a IRQ for concluído, a execução retornará ao processo que estava sendo executado no momento em que ocorreu a IRQ.

O segundo momento em que o processo pode ter a execução suspensa está relacionado ao tipo de algoritmo de escalonamento. Dentre eles destacamos o FCFS, SJF Não Preemptivo, SJF Preemptivo, Prioridade Preemptivo e Round Robin. O FCFS objetiva atender o processo que chegar primeiro, dentro da política "primeiro a chegar, primeiro a ser servido", do inglês *first come, first served*. Já o SJF vem do inglês *short jobs first*, que em tradução livre significa "o menor trabalho primeiro", sendo que esse algoritmo pode ser preemptivo ou não. O algoritmo por prioridade preemptivo baseia-se na prioridade do processo, conceito de que pode variar de implementação para outra em diferentes SOs. Uma sugestão de leitura para aprofundar ou relembrar detalhes sobre esses algoritmos é a encontrada em Deitel (2005) e Tanenbaum e Bos (2016).

Neste capítulo, todavia, vamos abordar o algoritmo por prioridade, com tempo definido de execução e preemptivo e também o algoritmo *first in, first out* (FIFO), que é uma abordagem simplificada de escalonamento. Porém, antes vamos destacar um pouco sobre algoritmos em lote.

Um algoritmo em lote tipicamente é usado quando existe um conjunto conhecido e finito de processos que devem ser executados e para os quais não há necessidade de interação, durante o processamento, entre o SO e o usuário. O conjunto de processos tem a execução iniciada e somente após o seu término um novo conjunto poderá ser admitido. Todavia, dentro do lote, qual deve ser o primeiro processo a ser executado?

Bom, a execução em lote, a escolha do primeiro, segundo etc. processo a ser executado pode ser determinada pela ordem de inscrição no lote. Aqui a política será do tipo "quem entrar primeiro na fila, será atendido primeiro e sairá primeiro", o que a caracteriza com FIFO. Outra ordem possível de execução pode ser pelo critério de "menor trabalho", estimado pelo usuário e informado durante a criação do lote de processos. Neste tipo de algoritmo os processos de menor tamanho (menor trabalho) serão executados primeiros.

Entretanto, ocorre que nos sistemas operacionais modernos para microcomputadores não há aplicação prática desses tipos de algoritmos, visto que uma característica principal que encontramos nos SOs atuais é a interação com o usuário. De fato, a interação evoluiu tanto que hoje os processos interagem com o usuário via teclado, mouse, tela sensível ao toque, e até mesmo com o uso da voz e sons. Por fim, note que, embora, o conceito de escalonamento por lote possa não ser uma boa política para um SO moderno, não quer dizer que em outras situações não possa ser usado. Existem muitos servidores na internet, como os servidores *web*, que usam esse tipo de processamento em lote para gerenciar *pools* (conjunto) de portas de conexão de seus clientes.

Retornando aos tipos de estratégias que não são baseadas em lotes, vamos focar nosso estudo novamente na política FIFO de escalonamento, pois ela pode estar presente tanto nas estratégias em lote quanto nas não baseadas em lote.

Obviamente, FIFO é algo bem simples, em termos de decisão da escolha de quem será executado, pois, literalmente, o primeiro processo a entrar, será o primeiro a sair. Podemos interpretar isso da seguinte forma: o primeiro processo que for criado pelo SO será o primeiro a ser executado. Assim, o FIFO é como um algoritmo por prioridade simples, no qual a prioridade é dada ao primeiro processo criado. O segundo processo somente será prioridade quando o primeiro for encerrado.

Lembre-se de que esse é um conceito teórico, já que na prática, com os microcomputadores e SOs atuais, pode não ser muito desejável e eficiente implementar uma política FIFO, pois não faria sentido executar um processo do início ao fim, sem interrompê-lo, desprezando-se todas as possibilidades de interação. Além disso, você se

lembra das IRQs? Elas estão presentes em todos os microcomputadores e SOs modernos. Portanto, mesmo utilizando-se a política FIFO de escalonamento, os processos que estiverem sendo executados poderiam ser suspensos para o tratamento de uma IRQ, sendo o seu processamento retomado ao término da IRQ.
Considerando um cenário hipotético de um SO moderno que implementasse a política FIFO, imagine que você trabalha em um escritório e precisa realizar, por exemplo, os cálculos da folha de pagamento. Como você sabe que irá demorar umas duas horas de processamento, você aproveita e resolve assistir a um vídeo no microcomputador durante esse tempo. Veja que você inicia o vídeo e depois o processamento da folha de pagamento. Ao término do filme, ao ver como está o processamento da folha, acreditando já estar pronta, para sua (terrível) surpresa, o processamento só começou após o término do vídeo. Um SO que usasse exclusivamente FIFO teria um comportamento assim.
Como você pode perceber, uma política FIFO talvez não seja uma solução ideal quando temos processos e ambientes interativos entre o SO e o usuário. Surge então o conceito de execução dos processos de maneira compartilhada. Neste compartilhamento o tempo disponível para o processamento será dividido igualmente entre todos os processos. Essa parcela de tempo que cada processo recebe é denominada de *quantum*. Em uma abordagem simples, o primeiro processo a chegar seria a prioridade um. Caso surja outro processo, o primeiro processo continuará a ser a prioridade um, e será executado pelo seu *quantum* de tempo e, ao término desse tempo, será a vez da prioridade dois. Dessa forma, em algum momento, todos os processos serão prioridade e estarão em execução pelo seu *quantum* de tempo. Como cada processo é interrompido ao término do seu *quantum* de tempo, esse algoritmo por prioridade preemptivo.

2.2.2 Troca de contexto

Compreendemos que nos sistemas operacionais modernos ocorre constantemente um processo de substituição do processo que está em execução na CPU. Esse processo está associado diretamente à política de escalonamento que estiver em uso no SO, e junto com a decisão de substituir um processo por outro na CPU várias ações precisam ser tomadas pelo SO.
O primeiro passo que deve ocorrer quando o SO decide realizar a troca de processo na CPU é providenciar a proteção de todos os dados que estejam em memória e também a situação em relação a eventuais recursos que possam ter sido solicitados pelo processo, além de informações sobre a próxima instrução que deverá ser executada quando o processo retornar. Os dados, os recursos e as informações sobre a próxima instrução compõem o contexto de um processo.
O processo de proteção do contexto pode envolver simples remanejamento de dados de um ponto para outro na memória RAM ou deslocamento de dados da memória RAM para o disco, esta decisão é tomada em consideração ao estado de alocação da memória RAM no instante em que for ocorrer uma troca de contexto.
Para uma manipulação mais facilitada, as informações sobre o contexto do processo são reunidas em uma estrutura de bloco de controle. O bloco de controle de processo normalmente é denominado por PCB, do inglês *process control block* e estão registradas muitas informações sobre o processo, como, por exemplo:

- o ponteiro contador do programa, que tem por finalidade exclusiva indicar o endereço da próxima instrução que deverá ser executada para o processo. É por esse contador que a CPU saberá qual instrução deve ser executada, mesmo com várias trocas de contexto. Quando um processo é encerrado, ou entra em espera, um novo processo é escalado para a execução, e dentre as muitas coisas que precisam ocorrer, uma delas é carregar para a CPU a instrução que está contida neste contador;
- além do contador de programa, outros controles são necessários para que a CPU execute corretamente a diversidade de processos que podem estar carregados. Esses controles são denominados por registradores, compreendidos como variáveis que pertencem a CPU. Cada processo em execução terá valores diferentes de registradores, portanto quando ocorre a troca de contexto, esses registradores precisam ser corrigidos, processo a processo. O contador de programas é um desses registradores;

- registrador de base de memória, que também é um registrador, e demais informações associadas, dentre elas a tabela de páginas de memória, ou tabelas de segmento. Sem essas informações a CPU não saberia em que parte da memória os dados do processo estariam;
- informações de entrada e saída, relacionando os dispositivos que estão em uso pelo processo, inclusive a relação de arquivos que estão abertos, dentre outras informações necessárias.

O ponto central a ser compreendido aqui é que toda troca de processo a ser executada é precedida de uma atualização do PCB do processo em execução, atualizando-se os valores dos registradores da CPU, do ponteiro contador de programa, e das informações de entrada e saída. Em seguida, ocorre um salvamento de PCB do processo que estava em execução. Neste ponto ocorre o carregamento de um novo PCB e ajustes nos registradores da CPU para coincidirem com os dados do PCB do processo que irá ser executado. Após a realização dessa troca de contexto, o processo terá sua execução iniciada e/ou continuada na CPU, repetindo o ciclo de salvamento e carregamento enquanto o microcomputador estiver em uso.

2.3 Escalonamento de processos

Você deve ter percebido que a característica mais marcante no ambiente de funcionamento do SO é a constante troca de processos. De fato, é o principal fator de relevância de um SO, que pode ser bonito e atrativo visualmente, oferecer uma infinidade de funcionalidades, mas se for ruim na gestão dos processos, estará fadado ao fracasso.

A chave para o sucesso de um SO está intrinsecamente ligada à forma como gerencia os processos e, ainda mais especificamente, à forma como ele escolhe qual processo deve ser executado. Essa escolha pode envolver várias questões, tais como a idade do processo (há quanto tempo está sendo executado), ou a quantidade de tempo que ele está esperando, ou ainda o tamanho do trabalho a ser realizado pelo processo.

Cada uma dessas questões traz consigo diversas possibilidades de ação e é justamente a escolha tomada que torna o SO bom ou ruim. Em Tanenbaun e Bos (2016) e em Machado e Maia (2013), há uma extensa discussão sobre os algoritmos de gerenciamento de tarefas. Já em Deitel (2005), ocorre uma discussão com viés mais prático do que teórico. Na sequência, vamos analisar uma abordagem de escalonador de tarefas e ver os prós e contras da possível abordagem.

2.3.1 SJF-Preemptivo

Tipicamente o algoritmo SJF-preemptivo (lembre-se de que SJF significa “a tarefa menor primeiro”) (MELO NETO, 2014) associa duas abordagens em uma só: a primeira refere-se à escolha da menor tarefa; a segunda, ao uso de divisão de tempo com término arbitrário (preemptivo) ao término da divisão de tempo da tarefa (término do *quantum* de tempo). Sobre o SJF, Silberschatz, Galvin e Gagne (2015, p. 63) nos dizem que:

> o algoritmo SJF é um caso especial do algoritmo geral de *scheduling* por prioridades. Uma prioridade é associada a cada processo, e a CPU é alocada ao processo com a prioridade mais alta. Processos com prioridades iguais são organizados no *schedule* em ordem FCFS. O algoritmo SJF é simplesmente um algoritmo por prioridades em que a prioridade (p) é o inverso do próximo pico de CPU (previsto). Quanto maior o pico de CPU, menor a prioridade, e vice-versa (SILBERCHATZ; GALVIN; GAGNE, 2015, p. 63).

Sobre a escolha da tarefa de menor trabalho, há alguns problemas a serem superados. O primeiro dele diz respeito a como determinar o menor trabalho. Este de fato é o mais significativo problema envolvendo o algoritmo SJF, pois não há uma maneira definitiva de determinar essa informação. O tamanho de uma tarefa não pode ser determinado pelo tamanho do aplicativo, nem pela quantidade de memória que ele utiliza. Tampouco pela quantidade dos demais recursos que ele utiliza.

O segundo problema refere-se à possibilidade de que algum processo nunca venha a ser executado por ele ser "grande" e o SO sempre preferir escolher tarefas de menor tamanho. Em um cenário em que surjam vários processos de tamanho pequeno, os processos que forem maiores nunca serão executados.
De fato, o algoritmo SJF é aplicável a processos em lote, e para um cenário no qual os usuários possam informar qual é a duração estimada dos processos. Sem essa informação o SJF não pode ser utilizado. Tal abordagem poderia fazer sentido em um cenário diferente daquele propiciado pelo microcomputador, no qual por vezes a tarefa não tem duração determinada previamente. Quando o usuário decide iniciar um aplicativo para navegar na internet, qual deveria ser a duração informada para essa tarefa? Fica claro a partir dessa simples questão, que o SJF não tem lugar nos dias atuais. Durante o tempo em que um processo é executado ele necessita de recursos, seja a memória, o acesso a um disco para escrever em um arquivo etc. O fato é que os processos necessitam informar ao SO sobre suas necessidades, o que chamamos de comunicação.

2.4 Comunicação e sincronização de processos

A comunicação de um processo pode ocorrer entre o processo e o SO, ou então entre diferentes processos. Quando ocorre entre processos é denominada de comunicação entre processos. Há ocasiões em que um processo produz dados para serem usados (consumidos) por outros processos. Como exemplo, um navegador web, que não se comunica diretamente com o *hardware*, mas faz uso dos recursos de rede, através da comunicação entre processos. Essa comunicação também pode se dar por meio do uso de API e chamadas de sistema.
No caso típico de produtor-consumidor, damos o nome de *pipeline*, que caracteriza a comunicação entre processos, na qual a saída de um é a entrada de outro. Além da comunicação, tem de ocorrer uma sincronização entre os processos. O consumidor não deve tentar ler os dados antes que o produtor os disponibilize, portanto é preciso estabelecer uma estratégia em que o produtor possa informar ao consumidor que há dados disponíveis.
A comunicação e a sincronização também estão presentes quando dois processos, digamos A e B, desejam ter acesso ao mesmo recurso, embora quando A esteja usando o recurso, este deva ficar indisponível para B, então B poderá usar quando A não estiver usando, e assim sucessivamente. Aqui a sincronização deve auxiliar na solução do uso compartilhado do recurso.
Vamos a um exemplo. Um processo X, que produz dados que devem ser impressos, e Y é o processo que deve imprimir os dados. Y deve sincronizar seu trabalho com o processo X para que a impressão comece apenas depois de X ter disponibilizado dados para serem impressos.
Para o estabelecimento da comunicação entre processos (ou com o SO) existem as seguintes abordagens: a) utilizar API ou chamada de sistema; b) usar memória compartilhada; c) usar encadeamento de saída/entrada (*pipeline*).
Quando se utiliza a API, ou chamada de sistema, para o estabelecimento de comunicação se estabelece um cenário bastante previsível de comunicação, no qual o uso da API define os mecanismos envolvidos nessa comunicação, pois as API determinam quais dados devem ser fornecidos ao solicitar um serviço, bem como quais dados serão retornados como resultados da execução do código correspondente a API.
Por outro lado, caso o recurso de comunicação entre os processos seja uma abordagem baseada no compartilhamento de memória, então será necessário o estabelecimento de estratégias de apoio que todos deverão cordialmente seguir.
No compartilhamento de memória estabelece-se um cenário no qual dois ou mais aplicativos terão acesso à mesma região da memória, e será necessário estabelecer de que forma irão compartilhar essa memória. Este cenário em particular introduz uma problemática que é o controle de concorrência e região crítica, que veremos logo adiante.
Já na abordagem de *pipeline* simplesmente um processo, digamos X, inicia um processo Y, este produz dados e os disponibiliza através, por exemplo, da saída padrão (normalmente é o vídeo). Assim, enquanto Y estiver produzindo dados, X poderá ser executado normalmente. Não há necessidade de se estabelecer outros mecanismos de controle da comunicação.

2.4.1 Programação concorrente

Sempre que dois ou mais processos necessitam usar os mesmos recursos ocorrem condições de corrida. Suponha que o processo A deseja escrever dados em uma região da memória que está compartilhado com o processo B e com o processo C. Suponha que em um determinado momento, o valor da região da memória seja 20 (vinte), todos os processos A, B e C leem esse valor e começam o processamento. Supondo-se que o processamento que deva ocorrer seja somar 1 ao valor que estava na memória, o correto seria que ao término do trabalho encontrássemos o s novo valor na memória: 23 (vinte e três).

Mas supondo-se que todos os processos estejam trabalhando em prioridades diferentes, com o término do processo B ele vai escrever o novo valor, que será 21 (vinte e um). Neste instante, o processo B é suspenso e quem passa a ser executado é o processo A (que também leu o valor 20, lembra-se?). O processo A conclui seu processamento e escreve o valor que encontrou (20 + 1) que é igual a 21 (vinte e um). O mesmo ocorre com o processo C que também leu o valor 20, calculou o valor 21 e o escreveu.

O resultado desse processamento e da condição de corrida será que ao término do processamento de A, B e C o valor de memória será 21 e não 23, que deveria ser o valor correto. Este tipo de problema de concorrência pode ter graves consequências.

Para superar este tipo de problema uma estratégia bastante utilizada é o estabelecimento de regiões críticas e o uso de semáforos para controlar quem tem acesso à região, definindo-se assim apenas um por vez.

2.4.2 Problemas de compartilhamento de recursos

Da situação analisada nas condições de corrida, percebe-se que é necessário organizar solidamente o acesso a recursos compartilhados, pois os problemas de compartilhamento de recursos podem causar graves prejuízos de funcionamento aos processos com resultantes de perdas de dados para o usuário. A condição de corrida em relação aos recursos compartilhados torna a região crítica, também chamada de seção crítica.

VOCÊ QUER LER?

A implementação de semáforos na solução de problemas de compartilhamento de recursos é um problema que sempre preocupou os desenvolvedores de SO e os programadores que fazem uso de técnicas que envolvem o compartilhamento. A solução que envolve a implementação de semáforos tem sido amplamente utilizada e a partir disso são estabelecidas duas vertentes de soluções. Caso você tenha ficado curioso em saber mais, leia a seção “Semáforos” do capítulo “Sincronização de processos” em “Fundamentos de Sistemas Operacionais” (SILBERSHATZ; GALVIN; GAGNE, 2015).

Algumas estratégias de controle envolvem o estabelecimento de semáforos para o controle de acesso à região crítica. E o que vem a ser semáforos? Trata-se de uma variável de controle. O processo que conseguir “travar” a variável terá acesso à memória. Essa trava poderia ocorrer da seguinte forma: a variável terá inicialmente o valor zero, se um processo ao ler a variável, encontrar o valor zero então ele muda o valor para um, e a variável fica travada, pois os demais processos encontrarão o valor um, indicando que a variável está travada.

Mas essa solução que parece criativa e simples está errada, pois ela tem o mesmo problema que tenta resolver, portanto solução não resolve nada.

Tentativas de solucionar o problema do compartilhamento de recursos encontraram soluções de dois tipos: a espera ocupada e a espera não ocupada (dormir). O primeiro ocupa tempo de processamento da CPU, sendo,

portanto, um processo no estado de "executando", enquanto que o segundo não ocupa tempo de processamento da CPU, estando na maioria dos SOs modernos, no estado "esperando". Este estado em particular é usado vinculando-se o processo que se deseja que "durma" a alguma IRQ de *software*. Quando a IRQ ocorrer o processo será acordado.

Na espera ocupada, o processo que irá esperar pelo recurso ocupará tempo de processamento da CPU, o que é uma estratégia ruim. Já na outra abordagem, o processo que espera pelo recurso será colocado em espera e somente será acordado quando o recurso estiver disponível para ele.

Bom, você percebeu que o trabalho do gerenciador de tarefas é enorme e que se não for bem desenvolvido o SO inteiro será de qualidade inferior, ou até mesmo inútil. Sendo assim, de nada adiantará quaisquer outras funcionalidades e características fantásticas se o SO não conseguir lidar bem com o gerenciamento das tarefas.

Síntese

A administração dos processos que estejam sendo executados é uma tarefa crucial e dela depende o sucesso de um SO. Vimos que várias ações e decisões devem ser tomadas para que o projeto de um SO seja vencedor.

Neste capítulo, você teve a oportunidade de:

- compreender a relação entre as linguagens de programação e a comunicação com o SO;
- entender a problemática relativa à escolha dos processos por parte dos algoritmos de escalonamento de tarefas;
- identificar o processo de comunicação entre os processos e o SO, como fatores chaves no compartilhamento de recursos;
- compreender conceitos iniciais sobre os semáforos e a espera ocupada.

Bibliografia

BARCELLOS, M. P. Programação Paralela e Distribuída em Java. In: 2ª Escola Regional de Alto Desempenho - ERAD 2002. São Leopoldo. **Anais...** São Leopoldo, p. 181-192,2002. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/erad-rs/2002/007.pdf>. Acesso em: 31/05/2018.

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.

CONTERRATO, G. B. H. Processamento da linguagem natural – algumas noções para aprimorar o tratamento de estruturas com predicado secundário. In: IBAÑOS, A. T.; PAIL, D. B. (orgs). **Fundamentos Linguísticos e Computação**. Porto Alegre: EDIPUCRS, 2015.

DEITEL, H. **Sistemas Operacionais**. 3. ed. São. Paulo: Pearson, 2005.

_____; DEITEL, P. J. C: **Como programar**. 6. ed. São Paulo: Pearson Prentice Hall, 2011.

FUEGI, J.; FRANCIS, J. Lovelace & Babbage and the creation of the 1843 'Notes'. In: **IEEE Annals of the History of Computing**. IEEE Computer Society. v. 25, n. 4, p. 16-26, 2003. Disponível em:< https://www.scss.tcd.ie/coghlan/repository/J_Byrne/A_Lovelace/J_Fuegi_&_J_Francis_2003.pdf>. Acesso em: 5/6/2018.

MACHADO, F. B.; MAIA L. P. **Arquitetura de Sistemas Operacionais**. 5. ed. São Paulo: LTC, 2013.

MELO NETO, A. **Estrutura dos Sistemas Operacionais.** USP. 2014. Disponível em: <https://www.ime.usp.br/~adao/teso.pdf>. Acesso em: 17/05/2018.

NEILL, C. et al. Blueprint for demonstrating quantum supremacy with superconducting qubits. **Science**. v. 360, n. 6385. p. 195-199, abr 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. **Fundamentos de Sistemas Operacionais**. 9. ed. São Paulo: LTC, 2015.
TANENBAUM, A. S.; BOS, H. **Sistemas Operacionais Modernos.** 4. ed. São Paulo: Pearson Education do Brasil, 2016.
TESLA. **The Autopilot Car**. 2018. Disponível em: <https://www.tesla.com/autopilot>. Acesso em: 31/05/2018.