

EEL770 - Sistemas Operacionais

Notas de Aula

Pedro Cruz *

¹Universidade Federal do Rio de Janeiro - PEE/COPPE/GTA

cruz@gta.ufrj.br

1. Introdução

Este texto contém notas de aula sobre a disciplina EEL770 - Sistemas Operacionais, ministrada no ano de 2018 na Universidade Federal do Rio de Janeiro. O texto pode e deve ser utilizado como material de consulta complementar, mas não deve substituir a bibliografia recomendada.


2. Processos

Os processos são uma abstração para os programas rodando em um computador. Do ponto de vista do Sistema Operacional (SO), os processos devem conter todas as informações que um programa qualquer deve ter para poder ser executado de maneira harmoniosa com os outros processos. O SO é responsável por gerenciar a execução, a utilização de memória e o compartilhamento de recursos pelos processos. Isso significa que o SO deve atrelar a cada processo todas as informações relativas ao estado de execução, permissões, relacionamentos com outros processos e outras propriedades que um programa qualquer deve ter para poder ser executado e conviver com os outros processos. Alguns desses atributos são armazenados dentro do espaço do processo, enquanto outros são armazenados na tabela de processos.

A tabela de processos é uma estrutura de dados mantida pelo SO que armazena dados de processos. Nela, cada linha da tabela representa um processo, ao passo que cada processo possui um número de identificação (*id*), um usuário que iniciou o processo, um processo pai, um tempo de início de execução, um ponteiro para uma lista de permissões (que deve conter os recursos que podem ser acessados pelo processo), entre outros atributos. A tabela de processos é melhor discutida na Seção 2.2.

Organização de um processo na memória Em termos de memória, cada processo é dividido pelo SO em três partes, ilustradas na Figura 1 e descritas a seguir:

- **Código:** é o trecho da memória que contém o código que é executado pelo processador ao longo da execução do processo. O contador de programa (*Program Counter* - PC) do processador aponta sempre para algum endereço de memória desse trecho. O trecho apontado pelo PC é executado pelo processador. Geralmente, é protegido para somente leitura pelo processo do usuário.

*O presente texto está sob uma licença Atribuição-NãoComercial-CompartilhaIgual CC BY-NC-SA . Uma cópia dessa licença pode ser obtida em <https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode>

- **Pilha de execução (*stack*):** estrutura do tipo último a entrar, primeiro a sair (*Last In, First Out* - LIFO). Contém as variáveis locais de cada função e os parâmetros que são passados a elas. Cada vez que uma função é chamada, suas variáveis locais e seus parâmetros são empilhados, juntamente com o endereço de retorno (ou seja, a linha de código que realizou a chamada da função). Quando a função retorna, as variáveis locais e os parâmetros são removidos da pilha e a execução é retomada a partir do endereço de retorno. Assim, a chamada de uma função qualquer aumenta o tamanho da pilha, ao passo que o retorno de uma função reduz o tamanho da pilha.
- **Dados (*heap*):** espaço reservado para a alocação dinâmica. O trecho de dados abriga a memória pedida pelo processo durante a execução. Os dados abrigados no *heap* não são apagados da memória até que o programa termine ou o que programa do usuário realize comandos específicos.

Os endereços de memória referenciados por um processo são endereços virtuais. O mapeamento entre endereços físicos da memória e os endereços referenciados pelos processos é um assunto extenso, a ser discutido em outro momento.

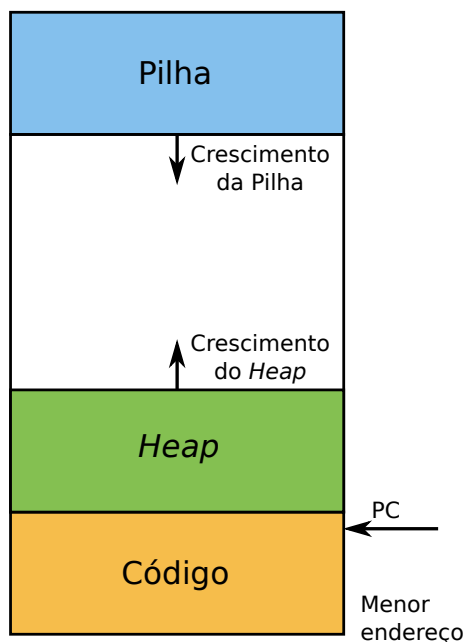


Figura 1. Organização típica de um processo na memória.

2.1. Pseudoparalelismo de processos

Um dos objetivos de um SO é garantir que um ou mais usuários sejam capazes de interagir com diversos processos simultaneamente. Porém, um computador com apenas uma CPU de núcleo único não é capaz de executar mais de um processo por vez. A estratégia utilizada pelos SOs é chavear os processos que são executados pela CPU, de forma que ao usuário exista a ilusão de que todos são executados simultaneamente.

Grande parte do esforço de um SO está em garantir que seja possível interagir com diversos processos, uma vez que chavear a CPU para diferentes processos não é trivial. Essa tarefa é realizada por uma parte do SO denominada escalonador. O escalonador

decide quanto tempo de CPU é oferecido a cada processo, além de ser responsável pela troca dos processos de fato.

A troca de processos deve ocorrer de forma que um processo removido da CPU possa ser colocado de volta exatamente da maneira em que foi retirado. O Bloco de Controle de Processos ajuda nessa tarefa.

2.2. Bloco de Controle de Processos

O Bloco de Controle de Processos é uma entrada na Tabela de Processos. Cada bloco armazena o identificador, o PC, os ponteiros inicial e final para cada região de memória relativa ao processo, o valor dos registradores e da Palavra de Estado do Programa (*Program Status Word* - PSW) da CPU e outros. Esses dados são utilizados para garantir que, quando a execução de um processo for interrompida pelo escalonador, é possível retomar a execução como se o processo nunca tivesse sido interrompido.

2.3. Estado de execução de um processo

Cada processo pode estar nos seguintes estados:

- **Novo:** o processo está sendo criado. O SO está criando as estruturas de dados relativas ao processo;
- **Pronto:** o processo está pronto para ser executado e está aguardando ser chamado pelo escalonador;
- **Executando:** o processo está sendo executado pelo processador;
- **Bloqueado:** o processo está aguardando algum evento, não podendo ser executado;
- **Suspensão:** o processo foi removido para a memória secundária a fim de ceder espaço na memória principal para outros processos. Pode estar **suspensão/bloqueado** ou **suspensão/pronto**.

A Figura 2 ilustra as transições de estados que um processo pode sofrer.

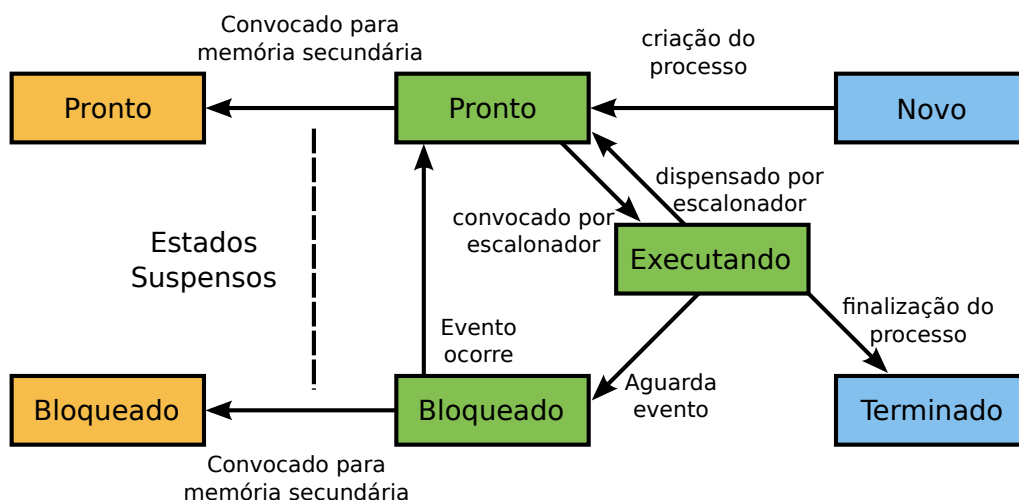


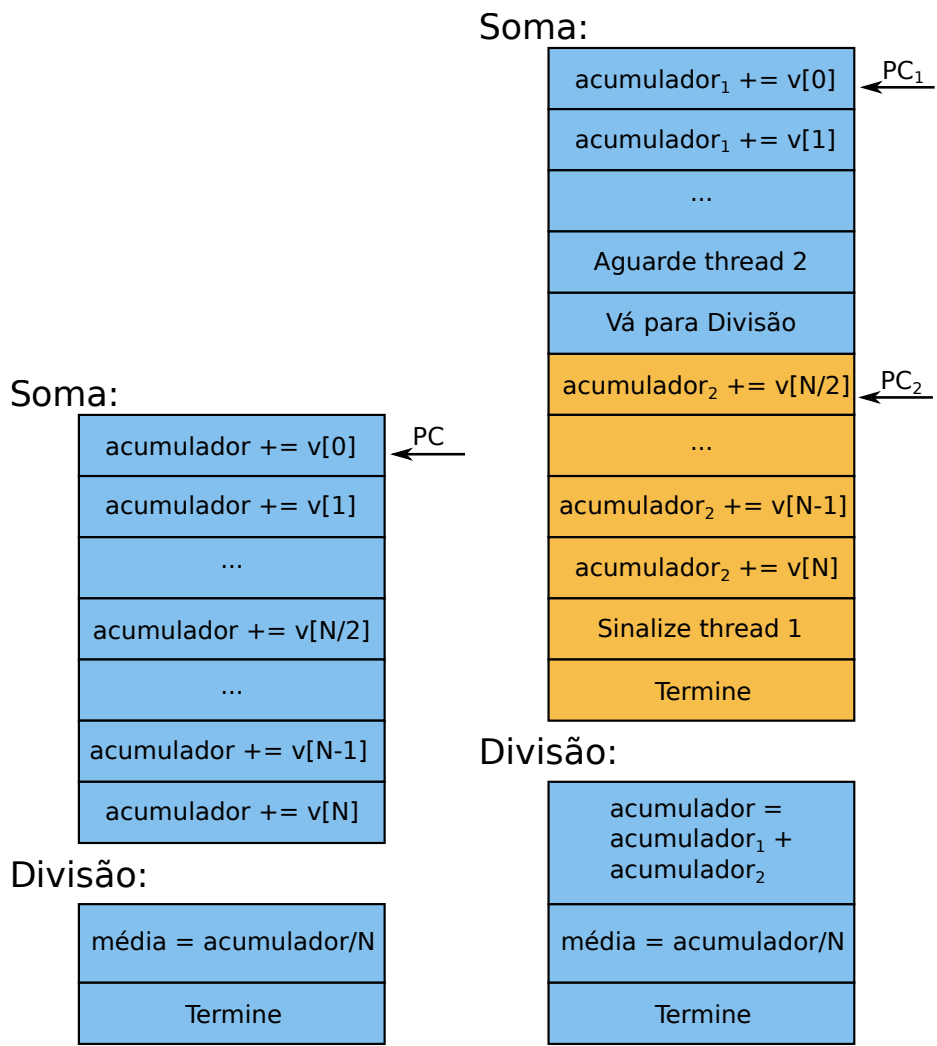
Figura 2. Transições de estados de um processo.

3. Threads

Um algoritmo é uma sequência de passos que devem ser executados para se obter um resultado. Quando o algoritmo é sequencial, existe uma ordem bem definida com a qual cada passo do algoritmo deve ser executado. Uma consequência direta disso é que não é possível executar os dois passos simultaneamente. Por exemplo, para calcular a média dos valores armazenados em um vetor, é necessário antes realizar a soma desses valores e depois efetuar a divisão dos valores pelo número de elementos no vetor. A Figura 3(a) ilustra a obtenção da média dos valores armazenados no vetor v utilizando uma única linha de execução. Assumimos que a variável *acumulador* está carregada com o valor zero. O trecho *Soma* acumula os valores, índice a índice, de cada elemento do vetor na variável *acumulador*. O trecho *Divisão* obtém a média dividindo o valor acumulado pelo número de elementos no vetor e termina o programa.

Existem algoritmos que permitem a execução fora de ordem de alguns trechos ou até mesmo a execução simultânea de tais trechos. Por exemplo, para calcular a média dos valores armazenados no vetor, não existe uma ordem para a qual as somas devam ser realizadas. Na Figura 3(b), o trecho *Soma* é executado em duas linhas de execução. A linha de execução apontada por PC_1 armazena a soma da primeira metade dos elementos do vetor na variável $acumulador_1$ e aguarda o término linha de execução apontada por PC_2 . A linha de execução apontada por PC_2 armazena a soma da segunda metade dos elementos do vetor na variável $acumulador_2$, sinaliza a primeira linha de execução e termina. A primeira linha de execução pode, então, obter a soma das duas metades do vetor e, então, calcular a média dos valores. Se tudo isso ocorrer no mesmo processo, cada uma dessas linhas de execução é denominada *thread*.

Para que seja possível executar essas linhas de execução, é necessário que cada uma delas possua sua própria pilha de chamadas, suas variáveis locais, seu contador de programa e um contexto.



(a) Cálculo da média dos valores em um vetor utilizando uma única *thread*. (b) Cálculo da média dos valores em um vetor utilizando duas *threads*.

Figura 3. Execução de um mesmo processamento utilizando uma ou duas *threads*.