DEFINIÇÃO

Definição de processos e threads e da forma como esses elementos devem ser estruturados para a construção de sistemas eficientes.

PROPÓSITO

Atualmente, até os mais simples dispositivos contam com a capacidade de multiprocessamento. Os sistemas operacionais, acompanhando o rápido desenvolvimento da tecnologia, fornecem mecanismos que possibilitam a construção de sistemas concorrentes que buscam tirar o máximo proveito desta capacidade. Saber explorar esta possibilidade é fundamental para a formação de profissionais habilitados a resolverem os desafios demandados por sistemas que buscam alto desempenho.

PREPARAÇÃO

Antes de iniciar o conteúdo deste tema, é desejável ter acesso a um computador (ou máquina virtual) com Linux instalado. Para os exemplos deste tema, foi utilizado o Ubuntu Desktop 20.04 LTS.

OBJETIVOS

MÓDULO 1

Descrever os conceitos de processos

MÓDULO 2

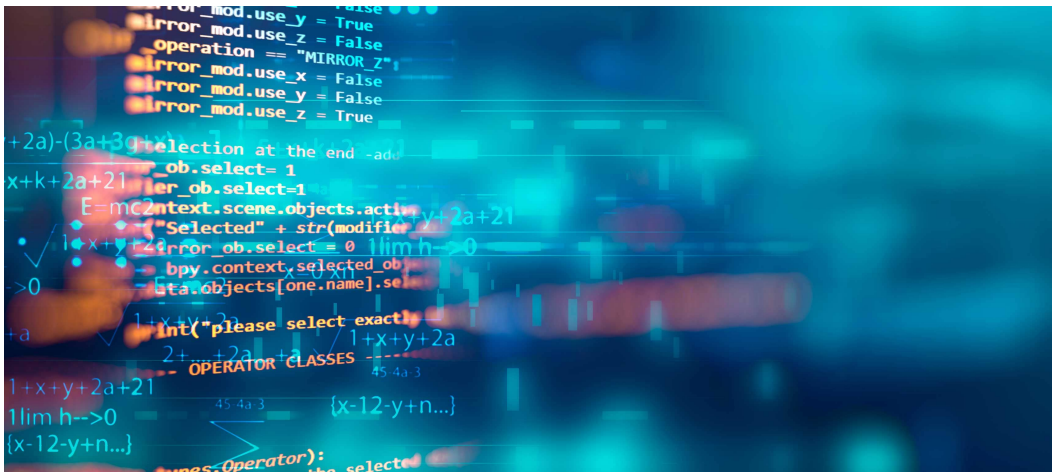
Compreender como ocorre a construção de programas concorrentes

MÓDULO 3

Identificar o mecanismo de comunicação entre processos

MÓDULO 4

Comparar as diferentes formas de escalonamento



INTRODUÇÃO

Neste tema, estudaremos os conceitos de processos e como eles são utilizados para tirar proveito da capacidade de multiprocessamento dos equipamentos atuais. Para isso, veremos a criação de subprocessos e como eles se comunicam. Estudaremos também o funcionamento dos threads, que são linhas de execução concorrentes dentro de um processo, fornecendo ao programador/desenvolvedor os mecanismos necessários para permitir a concorrência dentro de um processo.

Por fim, destacaremos os cuidados que devemos ter ao desenvolver sistemas que acessam dados compartilhados e os mecanismos que permitem o desenvolvimento seguro de tais aplicações.

MÓDULO 1

- **Descrever os conceitos de processos**

CONCEITOS

ANTES DE INICIAR

Neste tema, você verá alguns exemplos de funcionamento de programas e comandos. Para o desenvolvimento dos exemplos, foi utilizado o Sistema Operacional Ubuntu Desktop 20.04 LTS, porém você pode utilizar o Linux de sua preferência. O esperado é que não haja diferença de utilização entre as várias distribuições Linux.

Os programas utilizados como exemplo foram desenvolvidos em Linguagem C, uma linguagem clássica para o desenvolvimento de sistemas operacionais. O objetivo dos exemplos é permitir que você possa ver a utilização prática dos conceitos estudados no tema.

Para instalar o compilador de Linguagem C no Ubuntu, assim como em qualquer distribuição Linux derivada do Debian, basta executar no shell os comandos:

```
sudo apt-get update
sudo apt-get install gcc
```

Para compilar um programa chamado **prog.c** basta entrar no shell, no diretório onde se encontra o programa, e executar o comando:

```
gcc prog.c
```

Será criado um arquivo executável de nome **a.out**.

Para compilar um programa e escolher o nome do arquivo executável que será gerado, utilize o parâmetro **-o**. Por exemplo, para compilar o programa **prog.c** e gerar como saída um arquivo executável **prog**, utilize o comando:

```
gcc prog.c -o prog
```

Para executar o programa **prog** que acabou de ser compilado, execute o comando:

```
./prog
```

MODELO DE PROCESSO

Os primeiros sistemas permitiam a execução de apenas um programa de cada vez, que deveria ter o controle completo do sistema e acesso a todos os seus recursos. Os sistemas atuais permitem que diversos programas sejam carregados na memória e executados simultaneamente.

Essa evolução tornou necessário um controle maior na divisão de tarefas dos vários programas, resultando na noção de processo.

Em um **sistema multiprogramável**, a unidade central de processamento (UCP) alterna entre processos, dedicando um pouco de seu tempo a cada um, dando a ilusão de paralelismo. Este esquema costuma ser chamado de pseudoparalelismo.

SISTEMA MULTIPROGRAMÁVEL

Sistema que permite a execução de mais de um programa ao mesmo tempo.

Neste modelo, todo software executado no computador é organizado em **processos sequenciais**, também chamado de **processos**. O modelo de processos foi desenvolvido para tornar o paralelismo mais fácil de tratar.

Um processo é um programa em execução, incluindo os valores atuais dos registradores e variáveis, assim como seu **espaço de endereçamento**. Um programa por si só não é um processo, mas uma entidade passiva. Um processo é uma entidade ativa, com um **contador de instruções** e um conjunto de registradores a ele associado.

ESPAÇO DE ENDEREÇAMENTO

Conjunto de endereços de memória que um processo pode acessar.

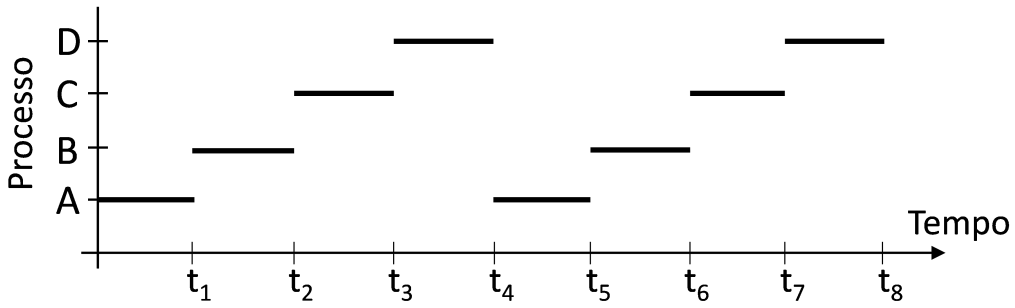
CONTADOR DE INSTRUÇÕES

Registrador de uma unidade central de processamento que indica qual é a posição atual na sequência de execução de um processo. Dependendo dos detalhes da arquitetura, ele armazena o endereço da instrução que está sendo executada ou o endereço da próxima instrução.

📢 ATENÇÃO

Embora dois processos possam estar associados a um mesmo programa, são duas sequências de execução distintas.

Conceitualmente, cada processo tem sua própria UCP. Com a UCP alternando entre os processos, a velocidade com que um processo executa não será uniforme.



📷 Exemplo de sistema de tempo compartilhado com 4 processos concorrentes.

Na figura anterior, vemos um exemplo de um sistema no qual são executados os processos A, B, C e D. O processo A é executado até o instante t_1, quando para, e o processo B é colocado em execução até o instante t_2. Os processos alternam suas execuções até que, no instante t_4, o processo D para de executar e o processo A retoma sua execução.

Como os processos alternam suas execuções de forma muito rápida, o usuário tem a ilusão de que os processos estão realmente sendo executados ao mesmo tempo.

CRIAÇÃO E ENCERRAMENTO DE PROCESSOS

Sistemas operacionais precisam criar processos durante sua operação.

Quatro eventos principais fazem com que processos sejam criados:

Inicialização do sistema.

Execução de uma chamada de sistema de criação de processo por um processo em execução.

Solicitação de um usuário para criar um processo.

Início de uma tarefa em lote.

Quando um sistema operacional é inicializado, uma série de processos são criados. Alguns são processos de primeiro plano (interagem com usuários), enquanto outros operam no segundo plano (background) e não estão associados a usuários em particular. Processos que ficam em segundo plano

para lidar com algumas atividades, como e-mail, páginas da web, notícias, impressão, são chamados de **daemons**.

Além dos processos criados durante a inicialização do sistema, outros também podem ser criados. Muitas vezes, um processo em execução emitirá chamadas de sistema para criar um ou mais processos para ajudá-lo em seu trabalho. Criar processos novos é particularmente útil quando o trabalho a ser feito pode ser facilmente formulado em termos de vários processos relacionados. Em um **multiprocessador**, por exemplo, cada processo pode ser executado em uma UCP, fazendo com que a tarefa seja realizada mais rapidamente.

Em sistemas interativos, os usuários podem começar um programa digitando um comando ou clicando duas vezes sobre um ícone. Cada uma dessas ações inicia um novo processo e executa o programa selecionado.

Tarefas em lote costumam ser executadas em grandes sistemas. As tarefas são submetidas ao sistema e, quando o sistema operacional tem os recursos necessários para executar outra tarefa, cria um processo e executa a próxima tarefa a partir da fila de entrada.

MULTIPROCESSADOR

Sistema com mais de um processador nos quais os processadores compartilham uma memória comum.

📢 ATENÇÃO

Em todos esses casos, um novo processo é criado por outro já existente, executando uma chamada de sistema de criação de processo. O que esse processo faz é executar uma chamada de sistema para criar o processo.

No Linux, a chamada de sistema mais comum para a criação de processos é a **fork()**. Essa chamada cria um processo idêntico ao processo que a chamou. Após a **fork()**, os dois processos, o pai e o filho, têm a mesma imagem de memória, as mesmas variáveis de ambiente e os mesmos arquivos abertos.

O código a seguir, em Linguagem C, exemplifica a criação de um novo processo com a chamada de sistema **fork()**.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
```

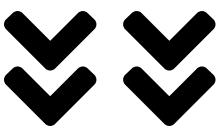
```
int main() {
int resultado, pid, ppid;
resultado = fork();
if (resultado < 0)
printf("Algo deu errado!!!\n");
pid = getpid();
if (resultado == 0) {
ppid = getppid();
printf("Eu sou o processo filho, meu PID é %d e meu pai tem PID=%d.\n", pid, ppid);
}
if (resultado > 0) {
printf("Eu sou o processo pai, meu PID é %d e meu filho tem PID=%d.\n", pid, resultado);
waitpid(resultado, NULL, 0);
}
}
```

Quando **fork()** é executada, o resultado do processamento é armazenado na variável “resultado”. O processo pai (que fez a chamada) receberá na variável “resultado” o **PID** do processo filho, enquanto o processo filho receberá na variável “resultado” o valor 0. Assim, para saber quem é o processo pai e quem é o processo filho, é necessário verificar o valor em “resultado”.

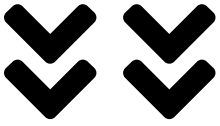
PID

Abreviação de *Process ID* (identificação do processo). É um valor que identifica um processo de forma única no sistema operacional.

De início, verifica-se a ocorrência de algum erro na criação do processo filho. Caso tenha ocorrido um erro, “resultado” receberá um valor negativo.

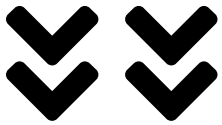


A seguir, o comando “pid = getpid();” obtém o PID do processo corrente.



Então, o processo verifica o valor de “resultado”. Se for zero, é o processo filho que está executando e, nesse, caso o processo executa “ppid = getppid();” para obter o PID do processo pai. Se o valor de

“resultado” for maior que zero, significa que quem está executando é o processo pai. Ambos (pai e filho) mostram na tela os respectivos PID.



Por fim, o processo pai executa a chamada de sistema “waitpid(resultado, NULL, 0);” para, antes de terminar, aguardar o término da execução do filho.

Quando o **fork()** é utilizado para a criação de um processo que executará o código de outro programa, a chamada de sistema **execve()** deve ser utilizada para que o processo filho mude sua imagem de memória e execute o novo programa.

O código a seguir ilustra a utilização de **execve()**. Nele, o processo pai cria três processos filhos em sequência. O primeiro coloca em execução uma calculadora, o segundo, o editor de textos **gedit**, e o terceiro, o utilitário **xeyes**.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char **argv, char* envp[]) {
int pid, i;

for (i=1; i<=3; i++) {
pid = fork();
if (pid < 0) {
printf("Algo deu errado!!!\n");
return 0;
}
if (pid == 0) { // Processo filho
if (i == 1)
execve("/usr/bin/xcalc", argv, envp);
if (i == 2)
execve("/usr/bin/gedit", argv, envp);
if (i == 3)
execve("/usr/bin/xeyes", argv, envp);
}
else // Processo pai
waitpid(pid, NULL, 0)
```

```
}
}
```

Após ter executado sua tarefa, o novo processo terminará devido a uma das seguintes condições:

Saída normal (voluntária)

A maioria dos processos termina por terem realizado o seu trabalho. Quando isto ocorre, o processo executa uma chamada para dizer ao sistema operacional que ele terminou e para que o sistema possa tomar as providências necessárias ao seu encerramento.

Erro fatal (involuntário)

O processo também pode terminar quando descobre um erro fatal. Ele pode solicitar ao usuário uma nova entrada de dados ou simplesmente encerrar sua execução de maneira similar à saída normal.

Saída por erro (voluntária)

Outra razão para o término é um erro causado pelo processo, muitas vezes decorrente de um erro de programa. Exemplos incluem executar uma instrução ilegal, referenciar uma memória não existente ou dividir por zero.

Morto por outro processo (involuntário)

A quarta razão pela qual um processo pode ser finalizado é a execução de uma chamada de sistema dizendo ao sistema operacional para *matar* outro processo. Para um processo *matar* outro, é necessário autorização.

Em alguns sistemas, quando um processo é finalizado, todos os processos que ele criou são finalizados também. Nem o Linux nem o Windows funcionam desta forma.

HIERARQUIA DE PROCESSOS

Em alguns sistemas, quando um processo cria outro, o processo pai e o processo filho continuam associados. O processo filho pode criar mais processos, formando uma hierarquia de processos. No Linux, um processo e todos os seus filhos e demais descendentes formam um grupo de processos.

No Linux, um processo especial chamado **systemd** (ou **init**, dependendo da versão) está presente na imagem de inicialização do sistema. É o primeiro processo a ser executado e é responsável por iniciar a execução dos demais processos do sistema operacional. Cada um desses processos pode iniciar mais processos. Desse modo, todos os processos no Linux pertencem a uma única árvore, com o **systemd** (ou **init**) em sua raiz.

A imagem a seguir exemplifica a árvore de processos em um sistema Linux por meio da execução do comando “**pstree**”.

```
systemd-+-ModemManager---2*[{ModemManager}]
|-NetworkManager---2*[{NetworkManager}]
|-VBoxService---8*[{VBoxService}]
|-accounts-daemon---2*[{accounts-daemon}]
|-acpid
|-avahi-daemon---avahi-daemon
|-colord---2*[{colord}]
|-cron
|-cups-browsed---2*[{cups-browsed}]
|-cupsd
|-dbus-daemon
|-gdm3-+-gdm-session-wor-+-gdm-wayland-ses-+-gnome-session-b---3*[{gnome-session-b}]
| | | `-2*[{gdm-wayland-ses}]
| | `-2*[{gdm-session-wor}]
| `-2*[{gdm3}]
|-2*[kerneloops]
|-login---bash---pstree
|-networkd-dispat
|-polkitd---2*[{polkitd}]
|-rsyslogd---3*[{rsyslogd}]
|-rtkit-daemon---2*[{rtkit-daemon}]
|-snapd---8*[{snapd}]
|-switcheroo-cont---2*[{switcheroo-cont}]
|-systemd---(sd-pam)
|-systemd-+-(sd-pam)
| |-at-spi-bus-laun-+-dbus-daemon
```

```
| | `-3*[{at-spi-bus-laun}]
| |-at-spi2-registr---2*[{at-spi2-registr}]
| |-dbus-daemon
| |-gjs---4*[{gjs}]
| |-gnome-keyring-d---3*[{gnome-keyring-d}]
| |-gnome-session-b---3*[{gnome-session-b}]
| |-gnome-session-c---{gnome-session-c}
| |-gnome-shell-+-Xwayland
| | |-ibus-daemon-+-ibus-engine-sim---2*[{ibus-engine-sim}]
| | | |-ibus-memconf---2*[{ibus-memconf}]
| | | `-2*[{ibus-daemon}]
| | `-6*[{gnome-shell}]
| |-goa-daemon---3*[{goa-daemon}]
| |-goa-identity-se---2*[{goa-identity-se}]
| |-gsd-a11y-settin---3*[{gsd-a11y-settin}]
| |-gsd-color---3*[{gsd-color}]
| |-gsd-keyboard---3*[{gsd-keyboard}]
| |-gsd-media-keys---3*[{gsd-media-keys}]
| |-gsd-power---3*[{gsd-power}]
| |-gsd-print-notif---2*[{gsd-print-notif}]
| |-gsd-printer---2*[{gsd-printer}]
| |-gsd-rfkill---2*[{gsd-rfkill}]
| |-gsd-smartcard---4*[{gsd-smartcard}]
| |-gsd-sound---3*[{gsd-sound}]
| |-gsd-usb-protect---3*[{gsd-usb-protect}]
| |-gsd-wacom---3*[{gsd-wacom}]
| |-gsd-wwan---3*[{gsd-wwan}]
| |-gsd-xsettings---3*[{gsd-xsettings}]
| |-gvfs-afc-volume---3*[{gvfs-afc-volume}]
| |-gvfs-goa-volume---2*[{gvfs-goa-volume}]
| |-gvfs-gphoto2-vo---2*[{gvfs-gphoto2-vo}]
| |-gvfs-mtp-volume---2*[{gvfs-mtp-volume}]
| |-gvfs-udisks2-vo---3*[{gvfs-udisks2-vo}]
| |-gvfsd---2*[{gvfsd}]
| |-gvfsd-fuse---5*[{gvfsd-fuse}]
| |-ibus-portal---2*[{ibus-portal}]
| |-ibus-x11---2*[{ibus-x11}]
| |-pulseaudio---3*[{pulseaudio}]
| |-tracker-miner-f---4*[{tracker-miner-f}]
```

| `-xdg-permission----2*[{xdg-permission-}]

|-systemd-journal

|-systemd-logind

|-systemd-resolve

|-systemd-timesyn---{systemd-timesyn}

|-systemd-udevd

|-udisksd---4*[{udisksd}]

|-unattended-upgr---{unattended-upgr}

|-upowerd---2*[{upowerd}]

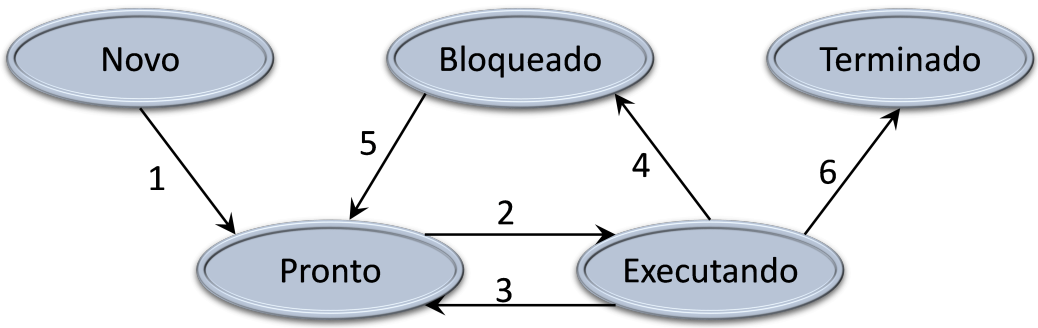
|-whoopsie---2*[{whoopsie}]

`-wpa_supplicant

ESTADOS DE PROCESSOS

Eventualmente, um processo que está em execução necessita parar momentaneamente sua execução por estar aguardando uma informação ainda não disponível ou porque já foi executado por muito tempo e precisa liberar a UCP para outro processo.

Um processo pode transitar por diferentes estados, que dependem do sistema operacional. De forma geral, podemos dizer que um processo pode estar nos estados **novo, executando, pronto, bloqueado ou terminado**.



Quando um processo é criado, ele se inicia no estado **novo**. O processo já existe, mas ainda precisam ser tomadas algumas providências pelo sistema operacional para que o processo possa iniciar sua execução. Quando tudo está pronto, o processo faz a transição 1 e muda para o estado pronto.

O processo reúne todas as condições para ser executado quando está no estado **pronto**, faltando apenas que o processador fique disponível para sua execução. Quando um processador fica disponível e o processo é selecionado para execução, ele faz a transição 2 e vai para o estado **executando**.

No estado **executando**, o processo tem suas instruções executadas pelo processador e três coisas podem acontecer:

1. A primeira delas é o processo ser executado por muito tempo. Então, o sistema operacional interrompe momentaneamente a execução do processo para colocar outro em seu lugar, ocorrendo a transição 3, que leva o processo de volta ao estado **pronto**, no qual aguardará nova oportunidade de execução.

2. Pode ocorrer ainda de o processo solicitar uma operação de entrada e saída e ter de aguardar a conclusão da operação, que costuma ser demorada quando comparada à capacidade de processamento do processador de um computador. Nesse caso, ocorre a transição 4, e o processo vai para o estado **bloqueado**.

3. Por último, o processo pode encerrar sua execução, realizando a transição 6 e indo para o estado **terminado**.

O processo que vai para o estado **bloqueado** permanece nele até que seja concluída a operação que aguardava. Quando isso ocorre, o processo passa pela transição 5 e vai para o estado **pronto** até ser novamente selecionado para execução.

📢 ATENÇÃO

O estado terminado é para os processos que não serão mais executados. Quando está neste estado, o sistema operacional deve providenciar a desalocação dos recursos que ainda estejam alocados ao processo. Somente após a desalocação de todos os recursos, o processo deixa de existir no sistema.

Para implementar o modelo de processos, o Linux mantém uma tabela de processos, com uma entrada por processo. Esta entrada é chamada de **Bloco de Controle de Processo** – BCP (*Process Control Block* – PCB) e contém todas as informações do processo. Algumas entradas do BCP são:

Estado do processo

Prioridade do processo

Número do processo

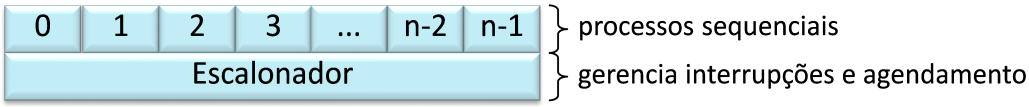
Registradores da UCP

Informações relativas ao gerenciamento de memória

Informações de contabilidade

Informações sobre operações de E/S

Essa visão dá origem ao seguinte modelo:



O nível mais baixo do sistema operacional é o **escalonador** (também conhecido como **agendador**). Ele cuida do gerenciamento de interrupções e dos detalhes de como iniciar e parar processos. Também costuma ser muito pequeno.

Um processo passa pelas várias filas de seleção durante sua execução. Cabe ao escalonador selecionar processos destas filas e decidir qual será o próximo a ser executado.

O escalonador é chamado com muita frequência. Um processo pode ser executado por apenas alguns milissegundos (ms) e ter de esperar por ter feito uma requisição de E/S. O escalonador costuma ser chamado pelo menos uma vez a cada 100ms para realizar a troca de processos. Devido ao pequeno intervalo de tempo entre as chamadas ao escalonador, sua execução deve ser bastante rápida, para que não se gaste muito tempo de UCP com trabalho de gerência.

MUDANÇA DE CONTEXTO

Para transferir o controle da UCP de um processo a outro, é necessário guardar o estado do processo em execução e carregar o estado do processo a entrar em execução. Esta tarefa é conhecida como **mudança de contexto** (ou **troca de contexto**).

O tempo gasto na mudança de contexto varia, dependendo de fatores como velocidade da memória, quantidade de registradores e existência de instruções especiais. Este tempo costuma variar de 1 a 1000 microssegundos.

O contexto de um processo pode ser dividido em três elementos básicos:

CONTEXTO DE HARDWARE

O **contexto de hardware** constitui-se basicamente do conteúdo dos registradores. No momento em que o processo perde a UCP, o sistema salva suas informações. Ele é fundamental para a implementação dos sistemas multiprogramáveis.

CONTEXTO DE SOFTWARE

O **contexto de software** especifica características do processo que influenciarão na execução de um programa. Ele define basicamente três grupos de informações sobre um processo: identificação, quotas e privilégios. A identificação define o processo para o sistema de forma única, através de seu PID, UID e GID. Quotas são os limites de cada recurso que o sistema operacional pode alocar, como número de

arquivos abertos, quantidade de memória, quantidade de subprocessos que podem ser criados etc. Privilégio é o que o processo pode ou não fazer em relação ao sistema e outros processos.

ESPAÇO DE ENDEREÇAMENTO

O **espaço de endereçamento** é a área de memória do processo em que o programa será executado e a área de memória onde os dados do processo serão armazenados. Cada processo possui seu próprio espaço de endereçamento, que deve ser protegido dos demais.

PROCESSOS NO LINUX

Os processos no Linux comportam-se como processos sequenciais tradicionais. É um sistema operacional multiusuário/multitarefa, que permite a execução simultânea de diversos processos, que podem pertencer a diferentes usuários. Mesmo que haja apenas um usuário logado no sistema, é comum a existência de diversos daemons em execução.

📢 ATENÇÃO

Um exemplo é o daemon de impressão, responsável por fazer a alocação da impressora e controlar o envio de trabalhos de impressão. É uma parte importante do sistema, pois permite o compartilhamento da impressora por diversos processos, possivelmente pertencentes a diferentes usuários.

A forma usual para criação de processos no Linux ocorre por meio da chamada de sistema **fork()**, conforme você já estudou neste tema.

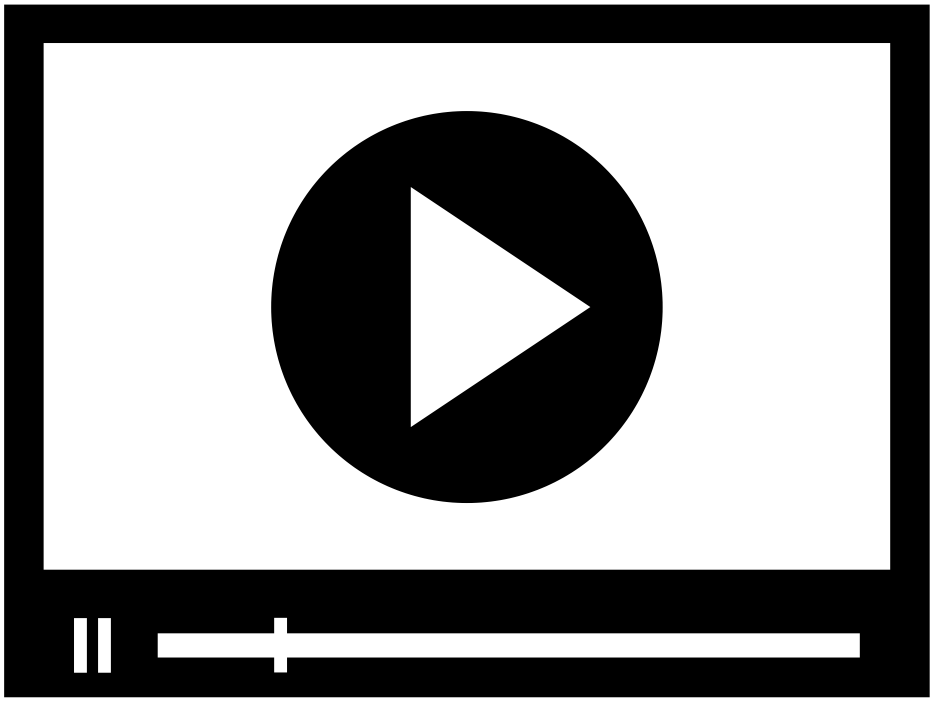
O processo filho recebe uma cópia exata do espaço de endereçamento do processo pai. Todos os valores de variáveis e demais objetos em memória serão idênticos, porém alterações realizadas por um dos processos em qualquer conteúdo de memória não afetarão o outro.

Arquivos que foram abertos antes da chamada **fork()** permanecem abertos para o processo pai e para o processo filho. As alterações realizadas por qualquer um destes processos se tornam imediatamente disponíveis para o outro. Processos são identificados por um número inteiro conhecido como PID (identificação do processo).

Algumas chamadas de sistema do Linux para o gerenciamento de processos são:

Chamada de sistema	Descrição
fork()	Cria um processo filho idêntico ao processo pai. Para o processo pai, retorna o PID do processo filho e, para o processo filho, retorna o valor 0.
waitpid()	Espera até que o processo filho passado como parâmetro termine sua execução.
execve()	Substitui a imagem de execução de um processo, fazendo com que, no lugar do processo corrente, seja executado o código do programa passado como parâmetro.
exit()	Termina a execução do processo e retorna como *status* o valor passado como parâmetro.

Faça o download do documento **Comandos do Shell**, com os principais comandos para Linux.



PROCESSOS NO LINUX.

VERIFICANDO O APRENDIZADO

MÓDULO 2

⊙ **Compreender como ocorre a construção de programas concorrentes**

SUBPROCESSO

Quando um processo (processo pai) cria um outro processo, ele é conhecido como **subprocesso** ou **processo filho**. O subprocesso, por sua vez, pode criar outros subprocessos.

A utilização de subprocessos permite dividir uma aplicação em partes que podem trabalhar de forma concorrente. Imagine, por exemplo:

★ EXEMPLO

Um servidor web que aceite requisições de clientes da internet e coloque as requisições em uma fila. Uma forma simples de implementar este servidor seria criar um processo que pegue a primeira requisição da fila, processe a requisição e devolva o resultado do processamento ao cliente que a solicitou. Após isso, ele pegaria a próxima requisição e faria o mesmo trabalho.

O problema com essa solução é que ela não aproveita a capacidade de multiprocessamento dos sistemas atuais. Como existe apenas um processo em execução, somente um dos processadores do sistema é utilizado para atendimento das requisições. Além disso, se houver várias requisições complexas e demoradas e uma requisição simples, como uma pequena página HTML, entrar no final da fila, esta requisição mais simples, que poderia ser respondida rapidamente, será atendida somente depois que todas as demais forem processadas.

A utilização de subprocessos resolve bem estes problemas. Se o servidor, no lugar de responder sequencialmente a cada requisição, criar um subprocesso para cada uma delas, tirará proveito da capacidade de multiprocessamento do sistema. Como **cada requisição será tratada por um processo diferente**, as requisições serão espalhadas pelos processadores do sistema, aproveitando sua capacidade de multiprocessamento. Além disso, como as requisições serão tratadas por diferentes processos, elas serão executadas concorrentemente.

Dessa forma, uma requisição simples, como a solicitação de uma página HTML, poderá ser iniciada e respondida rapidamente, ainda que existam outras requisições complexas solicitadas anteriormente e que elas ainda estejam sendo processadas.

O trecho de código abaixo em Linguagem C exemplifica a parte de um servidor web simples que cria subprocessos para atendimento das requisições:

CADA REQUISIÇÃO SERÁ TRATADA POR UM PROCESSO DIFERENTE

Um subprocesso é um processo completo. Então, cada subprocesso do servidor é, na realidade, um processo independente.

```
while (1) { // Loop infinito
req = pega_proxima_requisicao();
pid = fork();
if (pid == 0) { // Proceso filho
processa_requisicao(req);
exit(0);
}
}
```

Entenda o significado de cada um deles:

“PEGA_PROXIMA_REQUISICAO()”

Neste código, a rotina “pega_proxima_requisicao()” é responsável por verificar se existe alguma requisição enfileirada para atendimento. Se não houver, o processo pai fica bloqueado até que chegue uma nova requisição, sem impactar no desempenho do sistema. Enquanto isso, seus processos filhos continuam atendendo às requisições em andamento.

“PROCESSA_REQUISICAO()”

A rotina “processa_requisicao()” é executada somente pelo processo filho, uma vez que o comando if anterior faz essa verificação. Essa rotina fica encarregada do atendimento à requisição que foi enviada ao servidor web. Depois do processamento, a chamada de sistema “exit(0)” encerra o processo filho.

“WHILE (1) {…}”

Todo o código fica contido em um loop infinito “while (1) {…}”. Esse loop repetidamente pega a próxima requisição da fila, cria um processo filho para atendimento da requisição e retorna a aguardar uma nova requisição, enquanto o processo filho faz o processamento da requisição que acabou de chegar.

O uso de subprocessos demanda consumo de diversos recursos do sistema. Sempre que um novo processo é criado, o sistema deve alocar recursos (contexto de hardware, contexto de software e espaço de endereçamento) para ele, além de consumir tempo de UCP. Além disso, cada processo possui seu próprio BCP.

THREADS

Na tentativa de diminuir o tempo gasto na criação/eliminação de processos, bem como economizar recursos do sistema, foi introduzido o conceito de thread. Em um ambiente com múltiplos threads, não é necessário haver vários processos para implementar aplicações concorrentes.

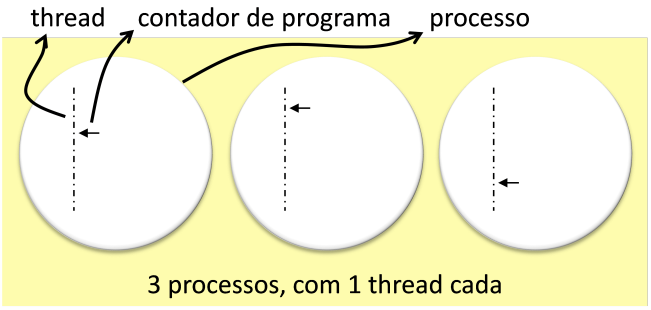
Threads compartilham o processador da mesma maneira que um processo. Cada thread possui seu próprio conjunto de registradores (contexto de hardware), porém compartilha o mesmo espaço de endereçamento com os demais threads do processo. No momento em que um thread perde a utilização do processador, o sistema salva suas informações. Threads passam pelos mesmos estados que um processo.

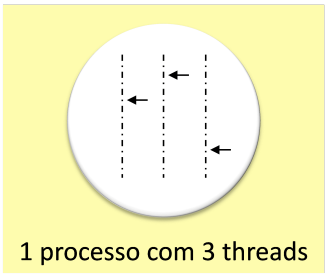
A grande diferença entre subprocessos e threads é em relação ao espaço de endereçamento.

Subprocessos possuem, cada um, espaços independentes e protegidos.

Threads, por outro lado, compartilham o mesmo espaço de endereçamento do processo, sem nenhuma proteção, permitindo que um thread possa alterar dados de outro thread. São desenvolvidos para trabalharem de forma cooperativa, voltados para desempenhar uma tarefa em conjunto, e são conhecidos como **processos leves**.

A mudança de contexto entre threads em um mesmo processo exige uma alteração de um conjunto de registradores, mas não é necessário nenhum outro trabalho, como gerenciamento de memória, por exemplo, tornando-a mais leve que a mudança de contexto entre processos.





Quando múltiplos threads estão presentes no mesmo processo, alguns campos da tabela de processos não ocorrem por processo, mas por thread.

Em alguns sistemas, os threads são gerenciados no espaço do usuário, sem o conhecimento do sistema operacional. É o caso do pacote P-threads (POSIX). A comutação de threads é muito mais rápida quando é feita no espaço do usuário pelo fato de não precisar fazer uma chamada ao kernel. Porém, quando os threads são executados no espaço do usuário e um thread bloqueia, todo o processo é bloqueado pelo kernel. Threads no nível do usuário também fazem com que o tempo dedicado a threads de diferentes processos não seja distribuído de forma justa.

📢 ATENÇÃO

Threads são muito úteis em sistemas com múltiplas UCP, nos quais o paralelismo real é possível. Elas permitem que ocorram múltiplas execuções no mesmo ambiente, fazendo com que várias partes de um mesmo processo estejam em execução concorrente em diferentes processadores.

O termo **multithread** também é usado para descrever a permissão de múltiplos threads no mesmo processo. Algumas UCP oferecem suporte de hardware direto para multithread e permitem que chaveamentos de threads aconteçam em uma escala de tempo de nanossegundos.

Processos com apenas um thread são conhecidos como **monothread**.

Além de compartilhar o espaço de endereçamento, os threads podem compartilhar o mesmo conjunto de arquivos abertos, processos filhos, alarmes e sinais.

📢 ATENÇÃO

É importante perceber que cada thread tem a sua própria pilha, que contém uma estrutura para cada rotina chamada, mas ainda não retornada. Essa estrutura contém as variáveis locais da rotina e o endereço de

retorno para serem usados quando a chamada de rotina for encerrada.

No Linux, os threads são criados com a chamada de sistema **clone()**. Sua sintaxe é:

INT CLONE(INT (*FN)(VOID *), VOID *STACK, INT FLAGS, VOID *ARG)

Os flags mais comuns são:

FLAG	COMPORTAMENTO	
	Quando utilizado	***Quando não utilizado***
CLONE_VM	Cria um thread.	Cria um processo.
CLONE_FS	Compartilha as informações sobre o sistema de arquivos.	Não compartilha informações sobre o sistema de arquivos.
CLONE_FILES	Compartilha os descritores de arquivos.	Copia os descritores de arquivos.
CLONE_SIGHAND	Compartilha a tabela do tratador de sinais.	Copia a tabela do tratador de sinais.
CLONE_PARENT	O novo thread tem o mesmo pai que o chamador.	O chamador é o pai do novo thread.
SIGCHLD	O thread envia o sinal SIGCHLD ao pai quando termina.	O thread não envia o sinal SIGCHLD ao pai quando termina.

Segue um exemplo de programa em Linguagem C que exemplifica a utilização de threads. Nele, são criados três threads dentro de um processo. O thread principal termina sua execução somente quando o último thread finaliza sua execução.

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sched.h>
#include <sys/wait.h>

#define TAMANHO_PILHA 65536
#define _1SEGUNDO 1000000

int global = 0; // Variável global alterada pelos threads

static int funcaoThread(void *arg) {
int id = *((int *) arg); // Identificação de cada thread
int i;

printf("Iniciou thread [%d]\n", id);
for (i=0; i<3; i++) { // Loop no qual o thread altera a variável global
   printf("Thread [%d] incrementou \"global\" para %d.\n", id, ++global);
   usleep(_1SEGUNDO * (1+id/10.0));
 }
printf("Saindo do thread [%d]\n", id);
}

int main () {
void *pilha;
int i, pid[3];
int id[3] = {1,2,3}; // Identificação a ser passada para cada thread

for (i=0; i<3; i++) { // Alocando espaço para a pilha de cada thread
if ((pilha = malloc(TAMANHO_PILHA)) == 0) {
perror("Erro na alocação da pilha.");
exit(1);
}
pid[i] = clone(funcaoThread,
pilha + TAMANHO_PILHA,
CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND | SIGCHLD,
&(id[i])); // Criação de cada thread
}
```

```
printf("Thread principal aguardando demais threads terminarem.\n");
for (i=0; i < 3; i++)
if (waitpid(pid[i], 0, 0) == -1) { // Aguarda até o término do thread
perror("waitpid")
exit(2);
}
printf("Thread principal terminando.\n\n");
}
```

Quando executado, o programa fornece a seguinte saída:

```
Thread principal aguardando demais threads terminarem.
Iniciou thread [3]
Thread [3] incrementou "global" para 1.
Iniciou thread [2]
Thread [2] incrementou "global" para 2.
Iniciou thread [1]
Thread [1] incrementou "global" para 3.
Thread [1] incrementou "global" para 4.
Thread [2] incrementou "global" para 5.
Thread [3] incrementou "global" para 6.
Thread [1] incrementou "global" para 7.
Thread [2] incrementou "global" para 8.
Thread [3] incrementou "global" para 9.
Saindo do thread [1]
Saindo do thread [2]
Saindo do thread [3]
Thread principal terminando.
```

Pela execução, você pode perceber que os valores das variáveis locais de cada thread não são influenciados pela execução dos demais threads. Porém, quando a variável “global” é modificada por um thread, esta alteração é vista imediatamente por todos os demais threads.

TIPOS DE PROCESSOS

Existem basicamente dois tipos de processos relacionados ao tipo de processamento que executam. Nos cards, a seguir, entenda cada um deles:

Os processos do tipo **CPU-bound** passam a maior parte do tempo no estado executando, realizando poucas operações de E/S. Costumam ser encontrados em aplicações científicas.

Os processos do tipo **I/O-bound** passam a maior parte do tempo no estado bloqueado, por realizar elevado número de operações de E/S. Costumam ser encontrados em aplicações comerciais. Processos interativos também são exemplos deste tipo de processo.

PROCESSOS E THREADS NO LINUX

O Linux pode duplicar um processo por meio da chamada de sistema fork(), mas também pode criar threads pela chamada de sistema clone(). No entanto, ele não faz distinção entre processos e threads. Toda entidade em execução, processo ou thread, será considerada como uma **tarefa (task)**. Um processo com um único thread será considerado como uma tarefa, e um processo com n threads terá n estruturas de tarefas.

📢 ATENÇÃO

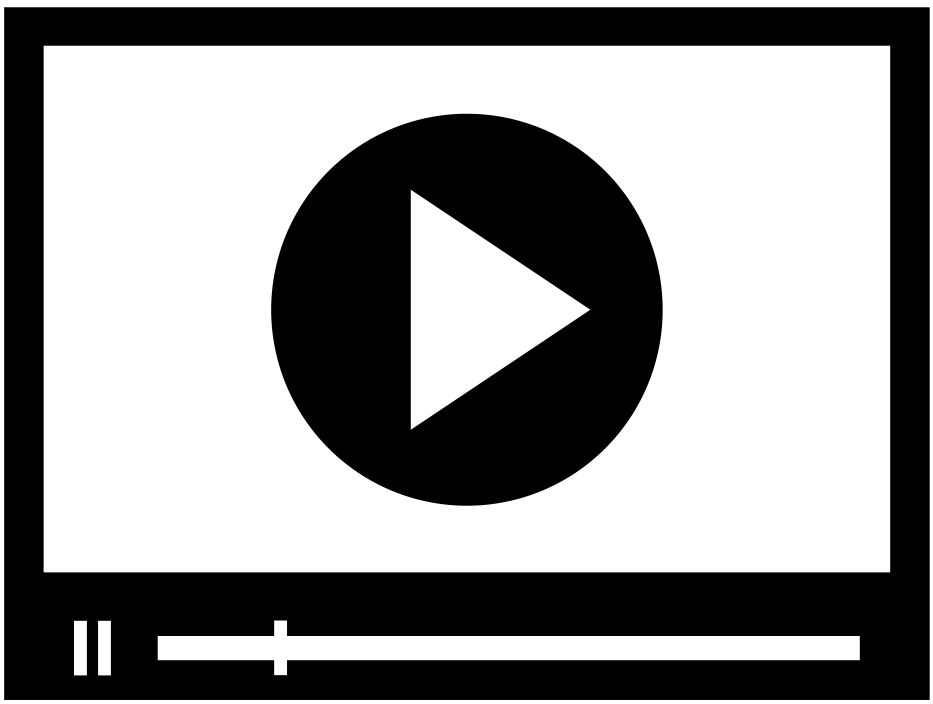
As chamas de sistema fork() e clone() são bastante semelhantes. De fato, se clone() for invocada sem nenhum flag passado como parâmetro, seu comportamento será idêntico ao fork().

O contexto de uma tarefa do Linux não é mantido em uma única estrutura, mas são criadas várias estruturas independentes para cada contexto. Os dados de uma tarefa são acessados por meio de ponteiros que apontam para cada estrutura de cada contexto. Dessa forma, o compartilhamento de informações entre as tarefas fica facilitado, bastando que as tarefas apontem para as mesmas estruturas em memória.

A fim de manter a compatibilidade com demais sistemas UNIX, o Linux associa um PID diferente a cada tarefa. Desta forma, diferentes threads de um processo terão diferentes PID.

Quando um subprocesso é criado, a princípio, o Linux deveria alocar memória para os diferentes processos e copiar o conteúdo dos segmentos de memória do processo pai. Porém, copiar segmentos de memória requer processamento e toma tempo. Então, para ser mais eficiente, o Linux simplesmente faz com que ambos (pai e filho) compartilhem o mesmo segmento de memória.

Se os processos se limitarem à operação de consulta nos segmentos, não haverá problema, mas, se algum dos processos tentar escrever em um dos segmentos, neste momento, haverá a cópia do segmento compartilhado (mecanismo conhecido como **cópia na escrita – copy on write**). Além de aumentar o desempenho do sistema, esse mecanismo ajuda a diminuir o consumo de memória física.



PROCESSOS E THREADS NO LINUX.

VERIFICANDO O APRENDIZADO

MÓDULO 3

- **Identificar o mecanismo de comunicação entre processos**

PROCESSOS DE APLICAÇÕES CONCORRENTES

O surgimento dos sistemas multiprogramáveis tornou possível criar aplicações nas quais diferentes partes do código de um programa pudessem executar de forma concorrente. Tais aplicações são conhecidas como **aplicações concorrentes**.

É comum que processos de aplicações concorrentes compartilhem recursos do sistema. Não importam quais recursos são compartilhados, os problemas decorrentes serão os mesmos. O compartilhamento de recursos entre processos pode gerar situações indesejáveis capazes de comprometer o sistema.

Se dois processos concorrentes trocam informações através de um buffer, um deles só poderá gravar dados caso o buffer não esteja cheio, enquanto um processo só poderá ler dados caso o buffer não esteja vazio. Em qualquer caso, os processos deverão aguardar até que o buffer esteja pronto para as operações de E/S.

As considerações anteriores levam a três questões:

Como um processo passa informações a outro?

Como garantir que dois processos não se interfiram?

Como realizar o sequenciamento quando um processo depende do outro?

É bastante comum que aplicações concorrentes necessitem trocar informações. Tal comunicação pode se dar por meio de variáveis compartilhadas (memória compartilhada) ou por troca de mensagens. Porém, independentemente do mecanismo de comunicação utilizado, é preciso que os processos possam se manter sincronizados.

📢 ATENÇÃO

Os mecanismos que garantem a comunicação entre processos concorrentes e o acesso a recursos compartilhados são chamados **mecanismos de sincronização**. Os mesmos mecanismos se aplicam a threads.

CONDIÇÃO DE CORRIDA

Em alguns sistemas, processos que estão trabalhando em conjunto, muitas vezes, utilizam uma memória comum, onde cada processo pode ler ou escrever. Este armazenamento compartilhado pode ser feito na memória principal ou em um arquivo em disco.

★ EXEMPLO

Imagine um programa que atualize o saldo de um cliente após o lançamento de um débito ou um crédito em um registro.

O trecho do programa que faz a atualização poderia ser:

```
void atualiza_saldo(double valor, int conta) {
Registro registro;
registro = le_registro(conta);
registro.saldo = registro.saldo + valor;
grava_registro(registro, conta);
}
```

★ EXEMPLO

Suponha que os funcionários Carlos e Orlando, caixas do banco, solicitem a atualização de uma conta cujo saldo é 500. Carlos faz uma operação de depósito de 100 e Orlando uma operação de saque de 200. Pode acontecer de o processo de Carlos ler o saldo da conta (500) e, ao mesmo tempo, o processo de Orlando ler o mesmo valor. Então, o processo de Carlos soma 100 e grava o registro com o valor 600. No entanto, o processo de Orlando já tem o valor de saldo de 500, subtrai 200 e fica com saldo de 300. Como o processo de Orlando grava o registro por último, o depósito não é computado, levando a uma inconsistência. O valor final das operações deveria ser 400, mas ficou registrado 300.

Problemas como esses são conhecidos como **condição de corrida**, que ocorre quando dois ou mais processos estão acessando dados compartilhados e o resultado do processamento depende de quem executa e quando é executado.

REGIÃO CRÍTICA

Como evitar as condições de corrida?

A forma mais simples é impedir que dois ou mais processos acessem um mesmo recurso no mesmo instante, impedindo que eles acessem o recurso compartilhado simultaneamente. Quando um processo estiver acessando o recurso, os demais deverão esperar. A esta exclusividade de acesso dá-se o nome de **exclusão mútua**, uma questão importante para o desenvolvimento de um sistema operacional.

A parte do programa que acessa a memória compartilhada é denominada **seção crítica** ou **região crítica (RC)**. Quando um processo é executado dentro de sua região crítica, nenhum outro processo pode entrar lá.

```
{
.....
Entra_Regiao_Critica();
Executa_Regiao_Critica();
Sai_Regiao_Critica();
.....
}
```

Não é suficiente evitar que o processo seja interrompido dentro da região crítica. São quatro as condições para uma boa solução:

1. Não pode haver mais de um processo simultaneamente dentro de suas regiões críticas.

2. Nenhuma suposição pode ser feita sobre a velocidade ou o número de UCP.

3. Nenhum processo que execute fora de sua região crítica pode bloquear outro processo.

4. Nenhum processo deve ter de esperar eternamente para entrar em sua região crítica (starvation).

Para garantir a implementação da exclusão mútua, os processos envolvidos devem fazer acesso aos recursos compartilhados de forma sincronizada.

SEMÁFOROS

Um semáforo é uma variável inteira que conta sinais enviados a ela. Associadas aos semáforos, existem duas operações especiais: up e down.

SEMÁFOROS

Semáforos e monitores são recursos para resolver o problema da condição de corrida.

A operação **down** decrementa o valor do semáforo se ele for maior que 0, senão o processo é bloqueado.

A operação **up** incrementa o valor do semáforo caso não haja processos que tenham sido bloqueados pela operação down, senão um processo é desbloqueado.

No caso da exclusão mútua, as instruções down e up funcionam como protocolos para que um processo possa entrar e sair de sua região crítica. O semáforo fica associado a um recurso compartilhado, indicando quando o recurso está sendo acessado por um dos processos concorrentes. Se seu valor for maior que 0, nenhum processo está utilizando o recurso. Caso contrário, o processo fica impedido de acessar o recurso.

Sempre que deseja entrar na sua região crítica, um processo executa uma instrução down. Se o semáforo for maior que 0, ele é decrementado de 1, e o processo que solicitou a operação pode executar sua região crítica. Se uma instrução down é executada em um semáforo cujo valor seja 0, o processo que solicitou a operação ficará no estado bloqueado em uma fila associada ao semáforo.

Quando o processo que está acessando o recurso sai de sua região crítica, ele executa uma instrução up, incrementando o semáforo de 1 e liberando o acesso ao recurso. Se um ou mais processos estiverem esperando, o sistema escolhe um processo na fila de espera e muda seu estado para pronto.

As operações up e down são realizadas pelo sistema operacional, que deve garantir que elas sejam executadas atomicamente.

A correção para o caso anterior de atualização do saldo de uma conta, com a utilização de semáforos, ficaria:

```
void atualiza_saldo(double valor, int conta) {
Registro registro;
down(mutex);
registro = le_registro(conta);
registro.saldo = registro.saldo + valor;
grava_registro(registro, conta);
up(mutex);
}
```

Com a utilização de semáforos, somente um dos processos estará dentro da região crítica em determinado instante. Voltemos ao caso dos caixas Carlos e Orlando:

★ EXEMPLO

Ambos solicitaram a atualização de uma conta cujo saldo é 500. Com a utilização de semáforos, antes de o processo de Carlos ler o saldo, é realizada a operação **down(mutex)**. Supondo que não haja outro processo na região crítica, o semáforo mutex permitirá a entrada do processo de Carlos. Quando o processo de Orlando tenta fazer sua operação, também se chamará **down(mutex)**, mas, como o processo de Carlos está na região crítica, o processo de Orlando será bloqueado pelo semáforo. O processo de Carlos segue normalmente, faz o depósito de 100 e atualiza o saldo para 600. Ao sair da região crítica, o processo de Carlos faz **up(mutex)**, liberando o processo de Orlando, que faz a leitura do saldo já atualizado (600). Assim, o processo faz o saque de 200 e atualiza o saldo para 400. Ao final, o processo de Orlando faz **up(mutex)**, liberando novamente a região crítica. Vemos que, com a utilização de semáforos, não ocorre a inconsistência vista anteriormente.

Para conhecer os programas em Linguagem C para Linux que fazem operações concorrentes de saque e depósito, consulte o documento **Utilização de semáforo na solução de condição de corrida**.

MONITORES

O uso de semáforos exige do programador muito cuidado, pois qualquer engano pode levar a problemas de sincronização imprevisíveis e difíceis de reproduzir. Monitores são mecanismos de sincronização de alto nível que tentam tornar mais fáceis o desenvolvimento e a correção de programas concorrentes.

Um monitor é uma coleção de variáveis, procedimentos e estruturas de dados que são agrupados em um pacote. Em determinado instante, somente um processo pode estar ativo em um monitor. Toda vez que algum processo chama um procedimento do monitor, ele verifica se já existe outro processo executando qualquer procedimento do monitor. Caso exista, o processo ficará aguardando a sua vez, até que tenha permissão para executar.

As variáveis globais do monitor são visíveis apenas a ele e a seus procedimentos. O bloco de comandos do monitor é responsável por inicializar essas variáveis, sendo executado apenas uma vez na ativação do programa onde está declarado o monitor.

Cabe ao compilador implementar as exclusões mútuas em entradas de monitor. A cláusula synchronized da linguagem Java é um exemplo de implementação de monitores. O programador transforma regiões críticas em procedimentos de monitor, colocando todas as regiões críticas em forma de procedimentos no monitor. Assim, o desenvolvimento de programas concorrentes fica mais fácil.

SINCRONIZAÇÃO NO LINUX

O Linux oferece suporte à utilização de semáforos para sincronização de tarefas por meio da chamada de sistema sem_wait(), que realiza a operação up(), e da chamada de sistema sem_post(), que realiza a

operação down().

Quando sem_wait() é invocada e o valor do semáforo é zero, a tarefa que faz a invocação é bloqueada até que o semáforo seja liberado por meio da função sem_post(). Porém, em algumas situações, a tarefa pode querer apenas verificar se é possível prosseguir, mas não é interessante bloquear, caso não possa prosseguir.

📢 ATENÇÃO

Para esses casos, pode ser utilizada a chamada de sistema **sem_trywait()**, que tem funcionamento parecido com a sem_wait(), exceto pelo fato de que, se o decremento não puder ser executado imediatamente, a chamada de sistema retorna um erro sem bloqueio.

Além da utilização de semáforos para sincronização de processos, o Linux permite a comunicação entre processos por meio de troca de mensagens e de sinais.

A **troca de mensagens** pode ser implementada pelo mecanismo de **pipe**. O pipe (|) é um mecanismo especial de redirecionamento utilizado para conectar a saída padrão de um processo à entrada padrão de outro processo. Por exemplo, os comandos a seguir juntam todos os arquivos com extensão “.txt”, ordenando suas linhas e retirando as duplicadas.

```
cat *.txt | sort | uniq
```

Entenda o significado de cada um deles:

O comando “cat *.txt” joga na saída padrão (tela) o conteúdo de todos os arquivos presentes no diretório e que possuem extensão “.txt”.

O comando “sort” recebe linhas por sua entrada padrão (inicialmente, o teclado), ordena as linhas e envia as linhas devidamente ordenadas para a saída padrão.

O comando “uniq” recebe linhas por sua entrada padrão e elimina as linhas duplicadas, enviando para a saída padrão somente as linhas que não estão em duplicidade.

Quando é utilizada a barra vertical (|) entre os comandos, é introduzido um pipe entre as tarefas, ou seja, as tarefas são colocadas em execução concorrente, e a saída padrão da tarefa à esquerda do pipe é conectada à entrada padrão da tarefa à direita do pipe. Assim, no exemplo, a saída do comando “cat *.txt” é enviada para a entrada do comando “sort”, que faz a ordenação, e sua saída é enviada à entrada do comando “uniq”, que elimina as linhas repetidas e, finalmente, joga o resultado de seu processamento para a saída padrão (o monitor, por padrão).

Outra forma de comunicação entre processos no Linux ocorre por meio de envio de sinais, que são enviados na ocorrência de eventos, como a chegada de uma informação pela rede, o fim de um temporizador, o término da execução de um processo filho etc. São eventos que chegam ao processo e precisam ser tratados de alguma forma. Para isso, é necessário definir uma rotina para o tratamento do evento.

São exemplos de sinais do Linux:

Sinal	Ação padrão	Comentário
SIGHUP	Terminar	Gerado pelo fim do terminal controlador.
SIGTERM	Terminar	Informa que deve parar a execução.
SIGINT	Terminar	Recebeu uma interrupção + pelo terminal controlador.
SIGKILL	Terminal	Força a finalização do processo.
SIGTSTP	Suspender	Processo deve ser suspenso (+).
SIGSTOP	Suspender	Processo deve ser suspenso. Semelhante ao SIGTSTP, mas não pode ser sobrescrito.
SIGCONT		Retornar à execução se estiver suspenso.
SIGCHLD	Ignorar	Informa ao processo pai que um processo filho terminou ou foi suspenso.
SIGALRM	Terminar	Fim de temporizador.
SIGURG	Ignorar	Condição urgente no socket. Normalmente, aviso de chegada de pacote de rede.

SIGUSR1	Terminar	Sinal definido pelo usuário.
SIGUSR2	Terminar	Sinal definido pelo usuário.

Quando um sinal chega ao processo, ele pode:

Receber o tratamento padrão definido pelo kernel.

Ser capturado, sendo, então, tratado por uma função definida pelo usuário.

Ser ignorado.

Para os sinais SIGKILL e SIGSTOP, sempre é executado o tratamento padrão. Eles não podem ser capturados nem ignorados.

Sinais podem ser gerados por:

Exceções de hardware.

Condições de software.

Pelo shell com o comando kill.

Por outro processo, utilizando a chamada de sistema kill().

Por uma combinação de teclas, como, por exemplo, < Ctrl> + < C>.

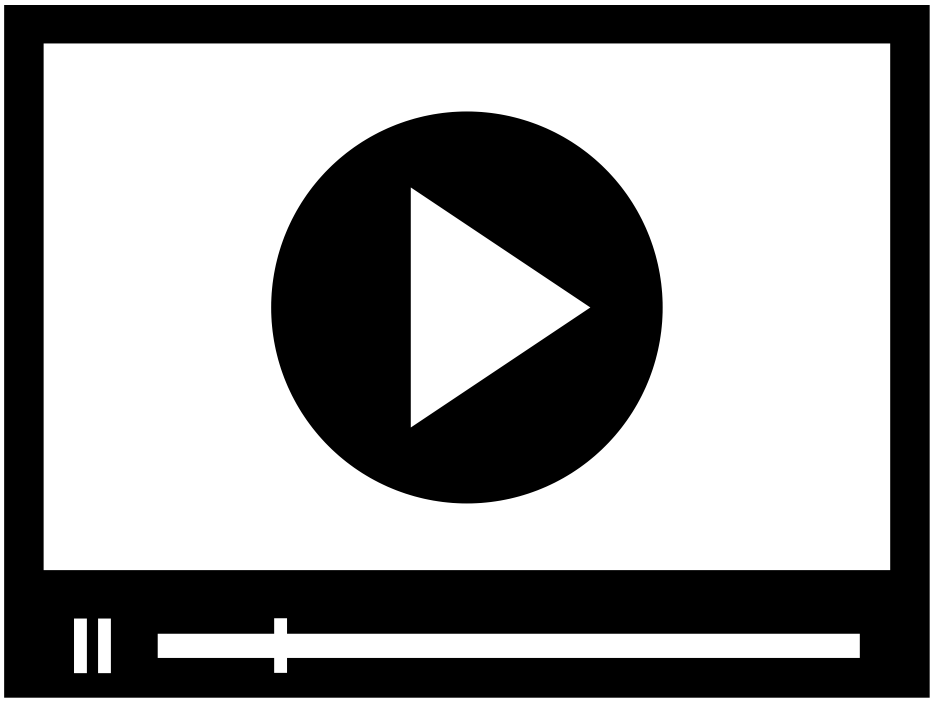
Controle de processos.

As principais chamadas de sistema relativas ao tratamento de sinais são:

Chamada de sistema	Descrição
signal()	Instala rotina para tratamento do sinal.
sigaction()	Define a ação a ser tomada nos sinais.
sigreturn()	Retorna de um sinal.

sigpending()	Obtém o conjunto de sinais bloqueados.
kill ()	Envia um sinal para um processo.
alarm()	Ajusta o alarme do relógio para envio de um sinal.
pause()	Suspende o chamador até o próximo sinal.

Faça o download do documento **Tratamento de sinais em Linux**, que mostra como fazer o tratamento de sinais em Linux.

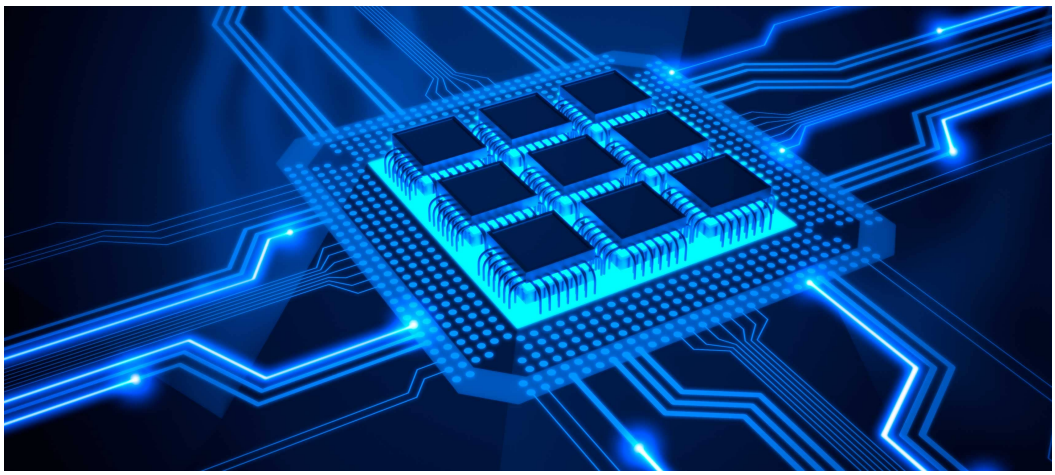


SINCRONIZAÇÃO DE PROCESSOS NO LINUX.

VERIFICANDO O APRENDIZADO

MÓDULO 4

- **Comparar as diferentes formas de escalonamento**



ESCALONAMENTO

A **multiprogramação** tem como objetivo permitir que, a todo instante, haja algum processo para maximizar a utilização da UCP.

O conceito que possibilitou a implementação de sistemas multiprogramáveis foi a possibilidade de a UCP ser compartilhada entre diversos processos. Portanto, deve existir um critério para determinar a ordem da escolha dos processos para execução dentre os vários que concorrem pela UCP.

O procedimento de seleção é conhecido como **escalonamento (scheduling)**. A parte do sistema operacional responsável pelo escalonamento é o **escalonador (scheduler)**, às vezes chamado de **agendador**. Sempre que a UCP se torna ociosa, o escalonador seleciona um processo dentre aqueles que estão na memória prontos para serem executados (na fila de processos no estado pronto) e aloca a UCP para que ele possa ser executado.

A principal função de um **algoritmo de escalonamento** é decidir qual dos processos prontos deve ser alocado à UCP.

O algoritmo de escalonamento não é o único responsável pelo tempo de execução de um processo. Ele afeta somente o tempo de espera na fila de processos prontos, e pode ser classificado como preemptivo ou não preemptivo. Quando o sistema pode interromper um processo durante sua execução para colocá-lo no estado pronto e colocar outro processo no estado executando, tem-se um sistema **preemptivo**. Caso contrário, há um sistema não preemptivo.

Realizar o escalonamento preemptivo exige que uma interrupção de relógio ocorra ao fim do intervalo para devolver o controle da UCP ao escalonador.

A troca de um processo por outro na UCP (mudança de contexto) causada pela preempção gera um overhead ao sistema. Para não se tornar crítico, o sistema deve estabelecer corretamente os critérios de preempção.

PREEMPTIVO

O escalonamento preemptivo permite que o sistema dê atenção imediata a processos mais prioritários, além de proporcionar melhor tempo de resposta em sistemas de tempo compartilhado. Outro benefício decorrente é o compartilhamento do processador de maneira mais uniforme.

📢 ATENÇÃO

Sistemas que usam escalonamento preemptivo têm o problema da condição de corrida, o que não ocorre com sistemas que usam escalonamento não preemptivo. No escalonamento não preemptivo, quando um processo ganha o direito de utilizar a UCP, nenhum outro processo pode tirar dele esse recurso.

A seguir, conheça os tipos de escalonamento:

ESCALONAMENTO FIRST IN FIRST OUT (FIFO)

Algoritmo de escalonamento também conhecido como **primeiro a chegar, primeiro a ser servido (*first come, first served*)**.

A grande força deste algoritmo é que ele é fácil de aprender e de programar.

Nesse escalonamento, o processo que chegar primeiro é o primeiro a ser selecionado para execução. É preciso apenas uma fila, em que os processos que passam para o estado pronto entram no seu final. Quando um processo ganha a UCP, ele a utiliza sem ser interrompido, caracterizando-o como um **algoritmo não preemptivo**.

O problema do escalonamento FIFO é a impossibilidade de se prever quando um processo terá sua execução iniciada. Outro problema é a possibilidade de processos CPU-bound de menor importância prejudicarem processos I/O-bound mais prioritários.

Este algoritmo foi inicialmente implementado em **sistemas batch**.

ESCALONAMENTO SHORTEST JOB FIRST (SJF)

O algoritmo SJF (tarefa mais curta primeiro) associa a cada processo seu tempo de execução. Quando a UCP está livre, o processo no estado pronto que precisar de menos tempo para terminar é selecionado para execução.

O escalonamento SJF beneficia processos que necessitam de pouco processamento e reduzem o tempo médio de espera em relação ao FIFO. O problema é determinar quanto tempo de UCP cada processo necessita para terminar seu processamento. Em ambientes de produção, é possível estimar o tempo de execução, mas, em ambientes de desenvolvimento, é muito difícil.

Imagine quatro tarefas A, B, C e D com tempos de execução de 14, 8, 6 e 4 minutos, respectivamente. Ao executá-las nessa ordem, o tempo de retorno para A é 14 minutos, para B, 22 minutos, para C, 28 minutos e, para D, 32 minutos, resultando em uma média de 24 minutos. Agora, considere executar essas quatro tarefas usando o algoritmo da tarefa mais curta primeiro. Os tempos de retorno são agora 4, 10, 18 e 32 minutos, o que resulta em uma média de 16 minutos.

É um algoritmo de escalonamento **não preemptivo** e, assim como o FIFO, também foi utilizado nos primeiros sistemas operacionais com processamento batch.

ESCALONAMENTO SHORTEST REMAINING TIME NEXT (SRTN)

O SRTN **(tempo restante mais curto em seguida)** é uma versão **preemptiva** do SJF. Nele, o escalonador escolhe o processo cujo tempo de execução restante é o mais curto. Para o seu funcionamento, o tempo de execução precisa ser conhecido antecipadamente.

Quando uma nova tarefa chega, seu tempo total é comparado ao tempo restante do processo atual. Se a nova tarefa precisar de menos tempo para terminar do que o processo atual, ela é suspensa, e a nova tarefa é iniciada. Esse esquema permite que novas tarefas curtas tenham um bom desempenho.

ESCALONAMENTO COOPERATIVO

O SJF e o FIFO não são algoritmos de escalonamento aplicáveis a sistemas de tempo compartilhado, onde um tempo de resposta razoável deve ser garantido a usuários interativos.

No escalonamento cooperativo, quando um processo já está em execução por determinado tempo, ele voluntariamente libera a UCP, retornando para a fila de processos prontos.

Sua principal característica está no fato de a liberação da UCP ser uma tarefa realizada exclusivamente pelo processo em execução, que a libera para um outro processo. Não existe nenhuma intervenção do sistema operacional na execução do processo. Isto pode ocasionar sérios problemas na medida em que um programa pode não liberar a UCP ou um programa mal escrito pode entrar em loop, monopolizando a UCP.

É um algoritmo de escalonamento **não preemptivo**.

ESCALONAMENTO CIRCULAR (ROUND ROBIN)

Esse algoritmo é bem semelhante ao FIFO. Entretanto, quando um processo passa para o estado executando, existe um tempo limite (conhecido como **time-slice** ou **quantum**) para utilização da UCP de forma contínua. Quando esse tempo expira, o processo volta ao estado pronto, dando a vez a outro processo.

A fila de processos no estado pronto é tratada como uma fila circular. O escalonamento é realizado alocando a UCP para cada processo da fila no intervalo de tempo determinado pelo quantum.

Se o quantum for muito pequeno, gasta-se muito tempo de UCP com trabalho administrativo. Se o quantum for muito grande, a interatividade fica prejudicada, já que um processo que sai de execução pode demorar muito a voltar. Em geral, o quantum varia de 10 a 100ms.

Através do escalonamento circular, nenhum processo poderá monopolizar a UCP, caracterizando-o como um algoritmo de escalonamento **preemptivo**.

Um problema do escalonamento circular é que ele não oferece qualquer tratamento diferenciado a processos I/O-bound. Assim, processos CPU-bound terão a tendência de monopolizar a utilização da UCP, enquanto processos I/O-bound permanecem à espera.

ESCALONAMENTO POR PRIORIDADE

O escalonamento circular consegue melhorar a distribuição do tempo de UCP em relação aos escalonamentos não preemptivos, porém ainda não consegue implementar um compartilhamento equitativo entre os diferentes tipos de processos.

Para solucionar esse problema, os processos I/O-bound devem levar alguma vantagem no escalonamento, a fim de compensar o tempo excessivo gasto no estado bloqueado. Como alguns processos devem ser tratados de maneira diferente dos outros, é preciso associar a cada um deles uma prioridade de execução. Assim, processos de maior prioridade são escalonados preferencialmente.

A preempção por prioridade é implementada mediante um relógio que interrompe o processador periodicamente, para que a rotina de escalonamento reavalie prioridades e, possivelmente, escalone outro processo, caracterizando-o como um algoritmo de escalonamento **preemptivo**.

Todos os sistemas de tempo compartilhado implementam algum esquema de prioridade, que é uma característica do contexto de software de um processo, podendo ser **estática** ou **dinâmica**.

Tem-se a **prioridade estática** quando a prioridade não é modificada durante a existência do processo. Apesar da simplicidade de implementação, a prioridade estática pode ocasionar tempos de resposta

elevados.

Na **prioridade dinâmica**, a prioridade do processo pode ser ajustada de acordo com o tipo de processamento realizado pelo processo ou pela carga do sistema. Quando o processo sai do estado bloqueado, recebe um acréscimo à sua prioridade. Dessa forma, os processos I/O-bound terão mais chance de ser escalonados e de compensar o tempo que passam no estado bloqueado.

Para evitar que processos com maior prioridade sejam executados indefinidamente, a prioridade é diminuída com o passar do tempo.

Outra forma de obter prioridade dinâmica é fazer com que o quantum do processo seja inversamente proporcional à fração do último quantum utilizado.

Embora os sistemas de prioridade dinâmica sejam mais complexos e gerem um overhead maior, o tempo de resposta oferecido compensa.

ESCALONAMENTO POR MÚLTIPLAS FILAS

Como os processos de um sistema possuem diferentes características de processamento, é difícil que um único mecanismo de escalonamento seja adequado a todos. Uma boa política seria classificar os processos em função do tipo de processamento realizado e aplicar a cada grupo mecanismos de escalonamentos distintos.

O escalonamento por múltiplas filas implementa diversas filas de processo no estado pronto, onde cada processo é associado exclusivamente a uma delas. Cada fila possui um mecanismo próprio de escalonamento em função das características do processo. Nesse esquema, os processos devem ser classificados, previamente, em função do tipo de processamento para poderem ser encaminhados à determinada fila.

Cada fila possui uma prioridade associada, que estabelece quais são prioritárias em relação às outras. O sistema só pode escalonar processos de uma fila se todas as outras filas de prioridade maior estiverem vazias.

Para exemplificar esse escalonamento, considere que os processos, em função de suas características, sejam divididos em três grupos: sistema, interativo e batch. Os processos do sistema devem ser colocados em uma fila de prioridade superior aos outros processos, implementando um algoritmo de escalonamento baseado em prioridades. Os processos de usuários interativos devem estar em uma fila de prioridade intermediária, implementando, por exemplo, o escalonamento circular. O mesmo mecanismo de escalonamento pode ser utilizado na fila de processos batch, com a diferença de que esta fila deverá possuir uma prioridade mais baixa.

ESCALONAMENTO EM SISTEMAS DE TEMPO REAL

Um sistema de tempo real é aquele em que o tempo tem um papel essencial. Tipicamente, um ou mais dispositivos físicos externos ao computador geram estímulos. O computador tem de reagir em conformidade dentro de um montante de tempo fixo. Exemplos de sistemas de tempo real são

equipamentos que estão monitorando pacientes em uma UTI, o piloto automático de um avião e o controle de robôs em uma fábrica automatizada. Em todos esses casos, ter a resposta certa tarde demais é, muitas vezes, tão ruim quanto não tê-la.

Sistemas em tempo real geralmente são categorizados como **tempo real crítico**, significando que há prazos absolutos que devem ser cumpridos, e **tempo real não crítico**, significando que descumprir um prazo ocasional é indesejável, mas, mesmo assim, tolerável. Em ambos os casos, o comportamento em tempo real é conseguido com a divisão do programa em uma série de processos, cada um dos quais é previsível e conhecido antecipadamente. Esses processos geralmente têm vida curta e podem ser concluídos em curto espaço de tempo. Quando um evento externo é detectado, cabe ao escalonador programar os processos de maneira que todos os prazos sejam atendidos.

Os eventos a que um sistema de tempo real talvez tenha de responder podem ser categorizados ainda como **periódicos** (significando que eles ocorrem em intervalos regulares) ou **aperiódicos** (significando que eles ocorrem de maneira imprevisível). Dependendo de quanto tempo cada evento exige para o processamento, tratar de todos talvez não seja possível. Se há *m* eventos periódicos e o evento i ocorre com o período P_i e exige C_i segundos de tempo da UCP para lidar com cada evento, a carga só pode ser tratada se:

$$\sum_{i=1}^{i} \frac{C_i}{P_i} \leq 1$$

Diz-se de um sistema de tempo real que atende a esse critério que ele é escalonável, o que significa que ele realmente pode ser implementado. Um processo que não atende a esse critério não pode ser escalonado, pois o montante total de tempo de UCP que os processos querem coletivamente é maior do que a UCP pode proporcionar.

Algoritmos de escalonamento de tempo real podem ser **estáticos** ou **dinâmicos**. **Algoritmos estáticos** tomam suas decisões de escalonamento antes de o sistema começar a ser executado. **Algoritmos dinâmicos** tomam suas decisões no tempo de execução, após ela ter começado. O escalonamento estático funciona apenas quando há uma informação perfeita disponível antecipadamente sobre o trabalho a ser feito e sobre os prazos que precisam ser cumpridos. Algoritmos de escalonamento dinâmico não têm essas restrições.

ESCALONAMENTO DE THREADS

Quando vários processos têm, cada um, múltiplos threads, há dois níveis de paralelismo presentes: processos e threads. O escalonamento nesses sistemas vai diferir, ainda, se são threads de usuário ou threads de núcleo.

Sendo threads de usuário, o núcleo não tem ciência da existência dos threads. Assim, ele opera como sempre fez, escolhendo um processo A e dando a A o controle de seu quantum. O escalonador de thread dentro de A decide qual thread executar (A1, por exemplo). Dado que não há interrupções para multiprogramar threads, esse thread pode continuar a ser executado por quanto tempo quiser. Se ele utilizar todo o quantum do processo, o núcleo selecionará outro processo para executar.

Quando o processo A executar novamente, o thread A1 retomará a execução. Ele continuará a consumir todo o tempo de A até que termine. No entanto, seu comportamento não afetará outros processos. Eles receberão o que quer que o escalonador considere sua fração apropriada, não importa o que estiver acontecendo dentro do processo A.

Agora, considere o caso em que os threads de A tenham relativamente pouco trabalho para fazer, por exemplo, 5ms de trabalho dentro de um quantum de 50ms. Em consequência, cada um é executado por um tempo. Então, cede a UCP de volta ao escalonador de threads. Isso pode levar à sequência A1, A2, A3, A1, A2, A3, A1, A2, A3, A1, antes que o núcleo chaveie para o processo B.

O algoritmo de escalonamento usado pelo sistema de tempo de execução pode ser qualquer um dos descritos anteriormente. A única restrição é a ausência de um relógio para interromper um thread que esteja sendo executado há tempo demais. Como os threads cooperam, isso normalmente não é um problema.

Vejamos agora a situação com threads de núcleo. Aqui, o núcleo escolhe um thread em particular para executar. Ele não precisa levar em conta a qual processo o thread pertence, mas pode fazer isso. O thread recebe um quantum e é suspenso compulsoriamente se o exceder. Com um quantum de 50ms e threads que são bloqueados após 5ms, a ordem do thread por algum período de 30ms pode ser A1, B1, A2, B2, A3, B3, algo que não é possível com esses parâmetros e threads de usuário.

Uma diferença importante entre threads de usuário e de núcleo é o desempenho. Realizar um chaveamento de thread com threads de usuário exige um punhado de instruções de máquina. Com threads de núcleo, é necessário um chaveamento de contexto completo, mudar o mapa de memória e invalidar o cache, o que representa uma demora bem maior. Por outro lado, com threads de núcleo, ter um bloqueio de thread na E/S não suspende todo o processo, como ocorre com threads de usuário.

Como o núcleo sabe que chavear de um thread no processo A para um thread no processo B é mais caro do que executar um segundo thread no processo A, ele pode levar essa informação em conta quando toma uma decisão.

Outro fator importante é que os threads de usuário podem empregar um escalonador de thread específico de uma aplicação.

SISTEMA BATCH

Tipo de processamento de dados em lote que não depende da interação com o usuário.

ESCALONAMENTO NO LINUX

O Linux suporta a **multitarefa preemptiva**, em que o escalonador decide que processo deve ser executado e quando executá-lo. Tomar essas decisões de modo que mantenha o equilíbrio entre justiça e desempenho para muitas cargas de trabalho diferentes é um dos desafios mais complicados dos sistemas operacionais modernos.

O Linux é baseado em tarefas do núcleo. São definidas três classes de tarefas:

FIFO em tempo real

Escalonamento circular em tempo real

Tempo compartilhado

Entre essas três classes, é utilizado um escalonamento por múltiplas filas, em que a classe FIFO em tempo real é a fila de maior prioridade e a classe tempo compartilhado é a classe de menor prioridade.

As tarefas FIFO em tempo real são escalonadas exclusivamente por prioridade, sem preempção. Sempre que houver uma tarefa FIFO em tempo real, ela utilizará o processador sem ser interrompida, a não ser que outra tarefa FIFO em tempo real de maior prioridade entre no estado pronto.

O escalonamento circular em tempo real funciona com **algoritmo de escalonamento circular** (Round Robin), com um quantum associado a cada tarefa. Sempre que uma tarefa excede seu quantum, ocorre uma preempção, e a tarefa é colocada no final da fila de processos prontos.

📢 ATENÇÃO

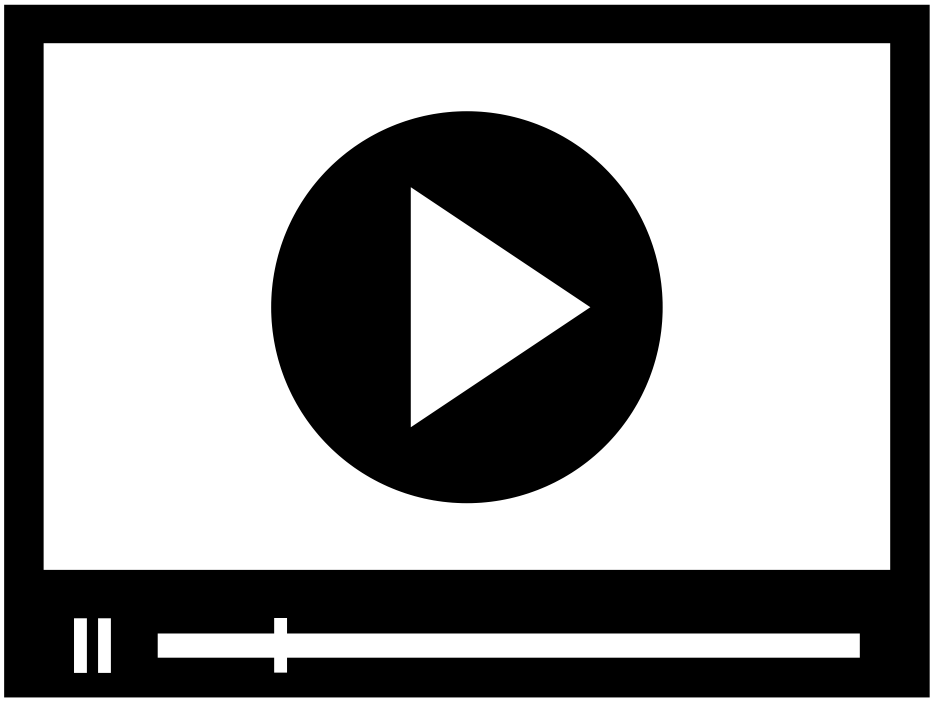
Apesar do nome “tempo real”, nenhuma dessas classes é realmente de tempo real, pois o sistema não tem como garantir os limites de tempo necessários ao funcionamento de um sistema de tempo real.

As tarefas de tempo real recebem prioridade de 0 a 99, sendo 0 o nível de prioridade mais alto e 99 o nível de prioridade mais baixo.

As tarefas de tempo compartilhado não competem com as tarefas de tempo real. Elas são escalonadas somente se não houver nenhuma tarefa de tempo real no estado pronto. As tarefas de tempo compartilhado recebem prioridade de 100 a 139, sendo 100 a prioridade mais alta e 139 a prioridade mais baixa desta fila. Dessa forma, o Linux possui um total de 140 níveis de prioridade.

Existe um utilitário de linha de comando chamado **nice**, que pode ser utilizado para alterar a prioridade de uma tarefa. O nice recebe como parâmetro um valor (de -20 a +19), que é somado à prioridade do thread. Apenas a conta root (administrator do Linux) pode fornecer um valor negativo como parâmetro

do nice. Isso significa que um usuário comum pode solicitar apenas diminuição na prioridade de suas tarefas, nunca aumento de prioridade.



ESCALONAMENTO DE PROCESSOS NO LINUX.

VERIFICANDO O APRENDIZADO

CONCLUSÃO

CONSIDERAÇÕES FINAIS

Iniciamos nosso estudo aprendendo sobre o modelo de processo, fundamental para implementação de sistemas de tempo compartilhado, em que o usuário do sistema pode executar, concorrentemente, vários programas, tendo a ilusão de que estão todos executando em paralelo.

Vimos ainda como a implementação de subprocessos e threads podem ser utilizados para o desenvolvimento de aplicações que façam uso da capacidade de multiprocessamento dos sistemas atuais. Tal técnica é de suma importância para o desenvolvimento de sistemas de alto desempenho.

Na parte de comunicação e sincronização entre processos, você pode perceber que, apesar do desempenho que pode ser obtido com a programação concorrente, cuidados devem ser tomados, principalmente no tocante à atualização de dados compartilhados. Para isso, podemos utilizar técnicas oferecidas pelo sistema operacional, como semáforos, que definem e controlam o acesso a regiões críticas, evitando as condições de corrida.

Por fim, estudamos os principais algoritmos de escalonamento que podem ser implementados pelos sistemas operacionais, vendo as principais caraterísticas de cada um deles. Como exemplo de algoritmos de escalonamento, vimos também como o Linux faz para selecionar o próximo processo a ser colocado em execução.

Concluímos que o conhecimento acerca da gerência do processador de um sistema operacional e de como se comportam processos, subprocessos e threads eleva o nível de conhecimento de um desenvolvedor/programador, permitindo que ele se torne apto ao desenvolvimento de sistemas melhores e de maior desempenho.

⚠ PODCAST

REFERÊNCIAS

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

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

NEMETH, E. *et al.* **Manual completo do Linux**: Guia do Administrador. 2. ed. São Paulo: Pearson Prentice Hall, 2007

UBUNTU. **Official Ubuntu Documentation**. Consultado em meio eletrônico em: 2 ago. 2020.

SILBERSCHATZ, A. **Fundamentos de Sistemas Operacionais**. 9. ed. Rio de Janeiro: LTC, 2013.

SILBERSCHATZ, A.; GALVIN, P. B. **Sistemas Operacionais – Conceitos**. 5. ed. São Paulo: Prentice Hall, 2000.

SILVA, G. M. **Guia Foca GNU/Linux**. Consultado em meio eletrônico em: 2 ago. 2020.

SOARES, L. F. G. **Redes de Computadores – Das LANs, MANs e WANS às Redes ATM**. 2. ed. Rio de Janeiro: Campus, 1995.

TANENBAUM, A. S.; BOS, H. **Sistemas Operacionais Modernos**. 4. ed. São Paulo: Pearson Educacional do Brasil, 2016.

TANENBAUM, A. S.; WOODHULL, A. S. **Sistemas Operacionais – Projeto e Implementação**. 3. ed. Porto Alegre: Bookman, 2008.

TANENBAUM, A. S. **Redes de Computadores**. 5. ed. São Paulo: Pearson, 2011.

TANENBAUM, A. S. **Structured Computer Organization**. 3. ed. Londres: Prentice Hall International, 1990.

EXPLORE+

Um contratempo que pode ocorrer com a criação de problemas concorrentes é o impasse entre processos. Aprenda sobre este assunto lendo o capítulo 6 do livro *Sistemas operacionais modernos*, de Andrew Stuart Tanenbaum e Herbert Bos.

CONTEUDISTA

Fábio Contarini Carneiro

🔗 CURRÍCULO LATTES