- Módulo 1 -

GERÊNCIA DE PROCESSOS

Antes de abordarmos os temas específicos relacionados a Sistemas Operacionais Abertos é necessário revisarmos a teoria da disciplina Sistemas Operacionais, onde o Universitário poderá firmar seus conceitos.

1. GERÊNCIA DE PROCESSOS

Um sistema operacional é quem gerencia os recursos computacionais num sistema. Isto é, organiza para que todos os recursos não tenham conflitos uns com os outros. Este controle é realizado por um módulo responsável pela supervisão e execução dos programas, que aloca os recursos necessários através do administrador conveniente, e controla a utilização da UCP pelos diversos processos.

A gerência de processo é responsável pelas seguintes áreas do sistema operacional, dentre outras:

- Algoritmos e estruturas de dados para implementar processos e abstrações de recursos;
- Escalonamento de processos;
- Sincronização de processos;
- Estratégia para tratamento de bloqueios perpétuos (*deadlocks*);
- Parte dos mecanismos de proteção e segurança.

1.1. Processo

Um Processo é um programa em execução, e possui:

- Seção de texto (código)
- Contador de instruções
- Pilha
- Seção de dados

ATENÇÃO:

Processo é diferente de programa, visto que um programa pode ser repetidamente executado gerando vários processos.

Os processos podem ser classificados em:

- ***I/O-bound***: delimitados pelo tempo de I/O. Gasta mais tempo fazendo I/O do que computações, muitas vezes pequenas rajadas(*bursts*) de CPU.

- ***CPU-bound***: delimitados pelo tempo e CPU. Gasta a maior parte do tempo fazendo computações. Realiza poucas rajadas longas de CPU.

Para que o sistema operacional possa gerenciar os processos é necessário um **descritor de processo**, também chamado de bloco de controle de processo (PCB – *Process Control Block*). Esta é a estrutura de dados utilizada para monitorar e controlar a execução do processo.

A maioria dos sistemas cria um identificador de processo que direta ou indiretamente referencia o descritor do processo.

Nele estão contidas as seguintes informações:

- Estado do processo
- Valor do PC (apontador de instruções)
- Área para guardar valor dos registradores
- Informação para escalonamento do processo
- Informação para gerenciamento de memória
- Informação para contabilidade do processo
- Status das operações de I/O (ex.: arquivos usados).

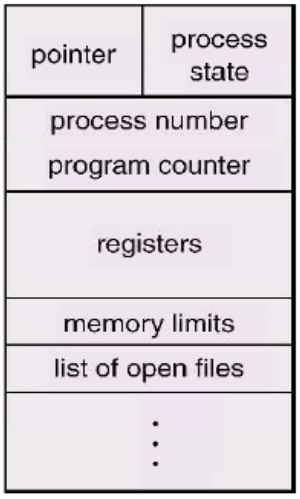


Figura 1 - *Process Control Block*

1.2. Espaço de Endereçamento

O espaço de endereçamento é um conjunto de localizações utilizado pelo processo para referenciar posições de memória primária, serviços do sistema operacional e recursos. Boa parte do espaço de endereçamento corresponde a localizações de memória primária.

Em outras palavras o espaço de endereçamento é a área de memória do processo onde o programa será executado, além do espaço para os dados utilizados por ele.

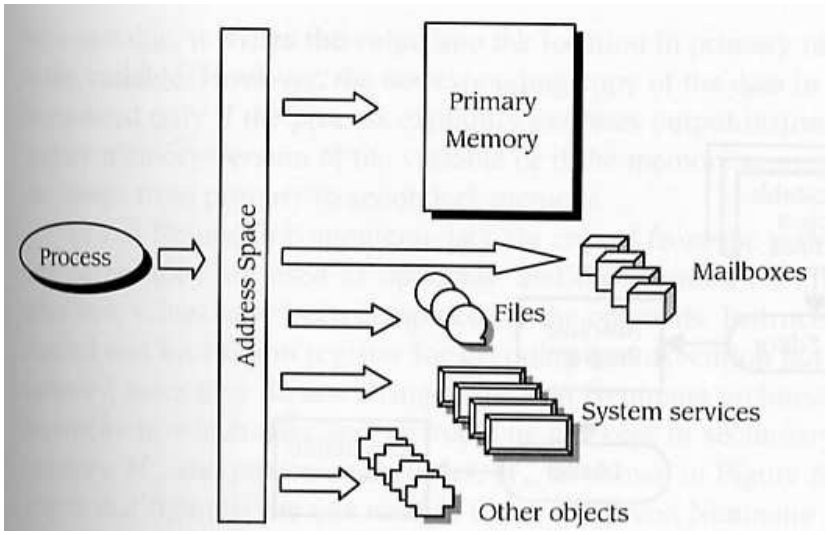


Figura 2 - Espaço de endereçamento

1.3. Estados do processo

Basicamente, existem três estados em que um processo pode se encontrar no sistema:

- **Execução**: quando está sendo processado

- **Pronto**: quando aguarda uma oportunidade para executar

- **Espera**/Bloqueado: quando aguarda algum evento externo ou por algum recurso para prosseguir seu processamento.

Figura 3 - Estados do processo

1.3.1. MUDANÇA DE ESTADOS

- Pronto para Execução: Quando um processo é criado, o sistema o coloca em uma lista de processos no estado de pronto, onde aguarda uma oportunidade para ser executado.

- Execução para Espera: Um processo passa para o estado de espera devido a eventos gerados pelo próprio processo, como uma operação de entrada/saída.

- Espera para Pronto: Um processo em espera passa para o estado de pronto quando a operação solicitada é atendida ou o recurso esperado é concedido.

- Execução para Pronto: Um processo em execução passa para o estado de pronto devido a eventos gerados pelo próprio sistema, como o fim do tempo que o processo possui para sua execução.

1.4. Threads

Até agora vimos um processo como um programa que possui apenas uma linha de controle, só realizando uma única tarefa por vez. No entanto, sistemas operacionais modernos permitem que um único processo possua várias linhas de controle (threads).

Isto permite, por exemplo, a um editor de textos realizar uma verificação ortográfica ao mesmo tempo em que o usuário digita caracteres.

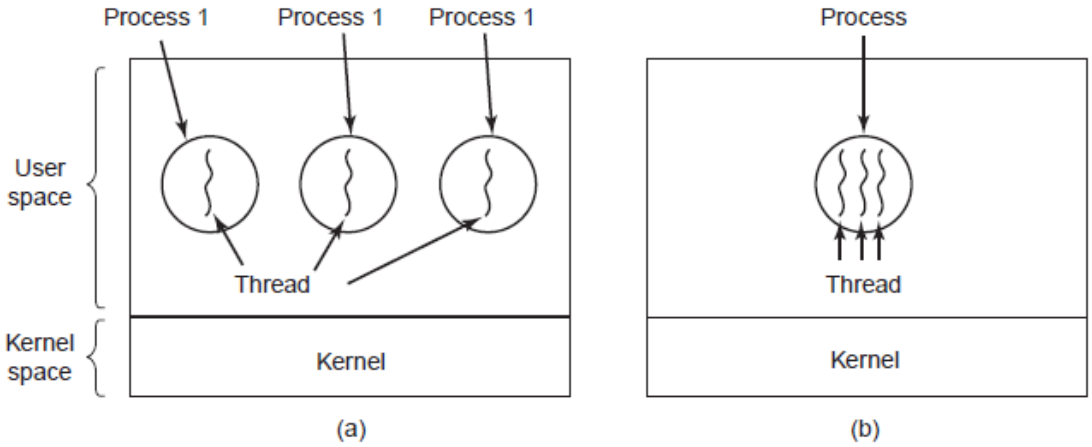


Figura 4 - (a) Três processos, cada um com um thread. (b) Um processo com três

Uma thread (ou *lightweight process*) é uma unidade básica de utilização de CPU. Uma thread consiste de:

- Apontador de Instruções (PC)
- Conjunto de registradores
- Espaço de pilha

Em um ambiente *multithread*, um processo é a unidade de alocação e proteção de recursos com um espaço de endereçamento virtual (espaço de endereçamento) que mantém a imagem do processo. Neste cenário obtém acesso controlado a outros processos, outros processadores, arquivos e outros recursos.

Quanto em execução compartilha com outras *threads* pares (*peers*):

- A seção de código
- A seção de dados
- Os recursos do Sistema Operacional

De forma semelhando ao processo, cada *thread* dispões de um bloco de controle da *thread* (TCB).

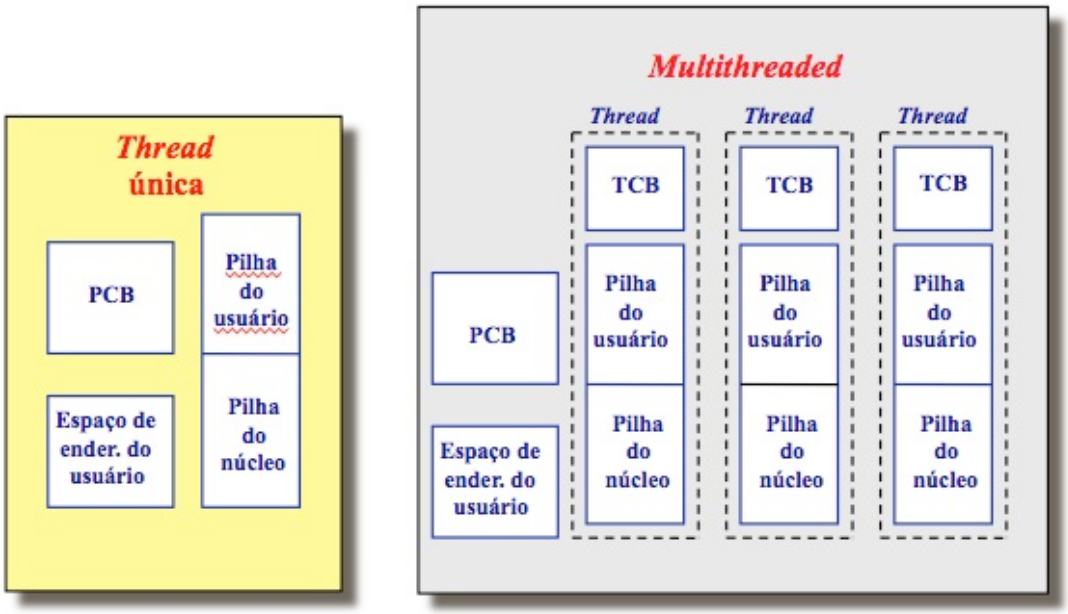


Figura 5 – Ambiente *Multithread*

Coletivamente, tudo isso é conhecido como tarefa ou *task*. Um processo tradicional é igual a uma tarefa com uma thread.

Em uma tarefa com múltiplas *threads*, enquanto uma *thread* está bloqueada e esperando, um segunda *thread* na mesma tarefa pode executar. Com isso, há cooperação entre muitas *threads* no mesmo *job*, conferindo maior vazão (*throughput*) e melhoria de desempenho. Com isso, as aplicações que requerem o compartilhamento de dados se beneficiam ao utilizá-las.

As *threads* provêm um mecanismo que possibilita a um processo sequencial fazer uma chamada bloqueante ao Sistema Operacional e ao mesmo tempo obter paralelismo no processo.

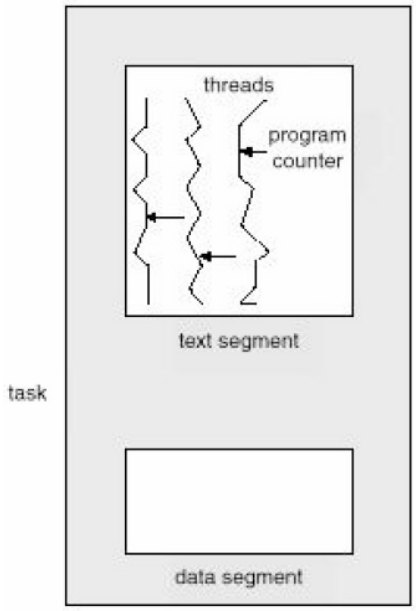


Figura 6 - Múltiplas *threads* dentro de uma tarefa

2. ESCALONAMENTO DE CPU

Para que a CPU não fique muito tempo sem executar tarefa alguma, os sistemas operacionais utilizam técnicas para escalonar os processos que estão em execução ao mesmo tempo na maquina.

O escalonamento de processos é uma tarefa complicada, pois nenhum algoritmo é totalmente eficiente e a prova de falhas, principalmente em se tratando de sistemas interativos, como o Windows, pois a interação com o usuário é fundamental para este sistema onde quem o utiliza procura respostas rápidas e a todo o momento processos são interrompidos pelo usuário.

O escalonador do sistema operacional utiliza alguns critérios de escalonamento, como:

- A taxa de utilização de CPU, que é a fração de tempo durante a qual ela está sendo ocupada;

- Throughput que são números de processos terminados por unidade de tempo;

- Turnaround que é o tempo transcorrido desde o momento em que o software entra e o instante em que termina sua execução;

- Tempo de resposta: intervalo entre a chegada ao sistema e inicio de sua execução;

- Tempo de espera: soma dos períodos em que o programa estava no seu estado pronto.

Para compreender corretamente o mecanismo de escalonamento é necessário conhecer os seguintes conceitos:

- **Multiprogramação**: vários processos estão na memória ao mesmo tempo. Quando um processo espera, o sistema operacional coloca outro processo para executar.

- **Escalonamento preemptivo**: o escalonamento da CPU é preemptivo quando o sistema operacional pode interromper um processo em execução para que outro processo utilize o processador.

- **Dispatcher**: ou escalonador de curto prazo é uma rotina encarregada de escolher qual dos processos no estado de Pronto para rodar deve receber a UCP, de acordo com suas prioridades.

- **Fila de prontos**: ou fila da UCP é uma lista ordenada, de acordo com as prioridades de cada processo.

- **Fila de espera**: é uma lista de processos suspensos (ou BLOQUEADOS),esperando por alguma condição. Por exemplo, a espera de uma operação de E/S, a espera de memória, etc.

- **Scheduler/Escalonador**: Além de manipular essas filas, ele também opera com o relógio de intervalos, estabelecendo, a cada novo intervalo de tempo, qual o valor da fatia de tempo destinada ao processo escolhido para ganhar a UCP. A transferência do controle da UCP ao processo escolhido é feita através da reposição do contexto desse processo via seu BCP (Bloco de Controle de Processo).

Escalonamentos podem ser catalogados por:

- Prioridade de uso
- Tempo de uso
- Seqüência organizada

Independente da condição necessária do escalonamento, o fim é ter uma otimização do fluxo de ordens. Na ausência de escalonamentos, obviamente, haverá um engarrafamento de processos e dependendo do quanto seja critico pode chegar a bloquear o andamento dos processos e consequentemente do sistema operacional.

2.1. Políticas de Escalonamento

Segue abaixo algumas das políticas de escalonamento de processos:

2.1.1. *FIRST-IN FITST-OUT (FIFO)*

O algoritmo FIFO não é preemptivo. Uma vez que a CPU seja alocada para um processo, ele fica com ela até o final ou até que peça alguma operação de E/S. Para sistemas de tempo compartilhado, este algoritmo não se aplica.

2.1.2. *SHORTEST JOB FIRST (SJF)*

É um algoritmo não preemptivo no qual o processo na fila de espera com o menor tempo total estimado de processamento é executado em seguida. O SJF reduz o tempo médio de espera sobre o algoritmo FIFO e favorece processos pequenos em prejuízo dos processos maiores.

2.1.3. ESCALONAMENTO POR PRIORIDADE

A cada processo é associada uma prioridade e a CPU é alocada para o processo com a mais alta prioridade. Processos com prioridades iguais são escalonados segundo a política FIFO. Este escalonamento pode ser tanto preemptivo quanto não preemptivo. O maior problema é o *starvation* (um processo de baixa prioridade pode ser indefinidamente postergado). Um método para prevenir *starvation* é o *aging*, que consiste em aumentar gradualmente a prioridade de um processo, à medida que ele fica esperando.

- **Prioridade estática:** não muda durante a vida do processo.

- **Prioridade dinâmica:** prioridade ajustada de acordo com o tipo de processamento e/ou carga do sistema.

2.1.4. *ROUND ROBIN* OU CIRCULAR

Algoritmo preemptivo, especialmente útil para sistemas de tempo compartilhado. Cada processo ganha um tempo limite para sua execução. Após esse tempo ele é interrompido e colocado no fim da fila de prontos. Este tempo é chamado de fatia de tempo, *time-slice* ou *quantum.*

Geralmente se situa entre 100 e 300ms. O tempo médio de espera é geralmente longo. O desempenho do algoritmo depende bastante da escolha do *quantum*. Se for grande, se parecerá com a política FIFO. Se for muito pequeno, o Round Robin é chamado Processador Compartilhado, pois parece, do ponto de vista dos processos, que cada um dos N processos "possui" uma CPU com velocidade 1/N do processador real.

2.1.5. ESCALONAMENTO POR PRAZOS

No escalonamento com prazos certos processos são escalonados para serem completados até certa data ou hora, ou um prazo.

Esses processos podem ter alta importância se entregues em tempo,ou podem não ter utilidade alguma se terminarem de ser processados além do tempo previsto no prazo. Este tipo de escalonamento é complexo por muitas razões:

- O usuário deve fornecer previamente todos os requisitos do processo;

- O sistema deve rodar o processo com prazo sem degradar o serviço para os outros usuários;

- O sistema deve cuidadosamente planejar seus requisitos de recursos durante o prazo. Isto pode ser difícil porque novos processos podem chegar e adicionar uma demanda imprevisível no sistema;

- O intensivo gerenciamento de recursos requerido pelo escalonamento com prazos pode gerar um *overhead* substancial, degradando o sistema.

3. DEADLOCK

Os sistemas computacionais estão repletos de recursos que podem ser usados por um processo por vez.

Exemplo: CD-ROM, Driver de Fita Dat, etc.

Ter dois processos simultaneamente gravando na impressora, por exemplo, resulta em uma confusão. Todos os SO têm a capacidade de temporariamente conceder acesso exclusivo a certos recursos para um processo.

Para muitos aplicativos, um processo requer acesso exclusivo não a um, mas a vários recursos. Imagine uma gráfica que precisa plotar um banner, cujas informações vêm em um CD.

Um processo A solicita a leitura do CD; um momento mais tarde outro processo B solicita a impressão de uma imagem. Agora o processo A solicita a plotadora e a bloqueia, esperando por ela. O processo B também solicita o CD e também bloqueia. Neste ponto os dois processos estão bloqueados e assim permaneceram eternamente. Essa situação é chamada de **Deadlock** ou **impasse.**

Simplificando:

Deadlock é um impasse gerado quando vários processos estão em disputa por recursos e acabaram gerando uma dependência circular.

Figura 5 - Impasse no trânsito

Os deadlocks podem ocorrer em outras situações além dessas de solicitar dispositivos dedicados de E/S. Em banco de dados, um programa pode precisar travar vários registros que ele está utilizando para evitar condição de corrida. Se o processo A trava o registro R1 e o processo B

trava o registro R2, e cada processo tenta travar o registro do outro também ocorre um deadlock. Em suma, os deadlocks ocorrem tanto em hardware como em software.

Em geral, deadlocks envolvem recursos não-preemptíveis. A seqüência de eventos requerida para utilizar um recurso é:

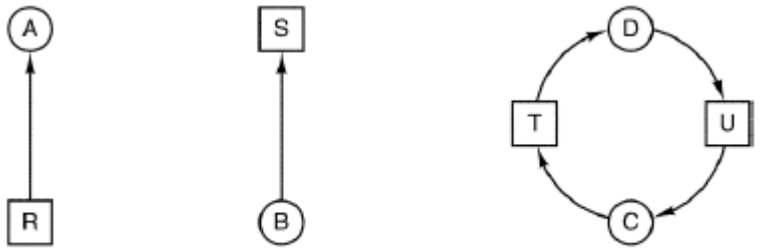
- Solicitar o recurso;
- Utilizar o recurso;
- Liberar o recurso.

Se o recurso requerido não está disponível, duas situações podem ocorrer:

- Processo que requisitou o recurso fica bloqueado até que o recurso seja liberado, ou;
- Processo que requisitou o recurso falha, e depois de certo tempo tenta novamente requisitar o recurso;

Então, deadlock pode ser definido formalmente da seguinte forma:

> Um conjunto de processos está em situação de deadlock se todo processo pertencente ao conjunto estiver esperando por um evento que somente outro processo desse mesmo conjunto poderá fazer acontecer.



a) recurso R está alocado ao processo A;

b) processo B está solicitando/esperando pelo recurso S;

c) processos C e D estão em deadlock sobre recursos T e U;

3.1. Condições para Deadlock:

1. Condição de exclusão mútua: Um recurso só pode estar alocado para um processo em um determinado momento;

2. Condição de posse e espera (*hold and wait*): Processos que já possuem algum recurso podem solicitar novos recursos;

3. Condição de não preempção: Recursos já alocados não podem ser retirados do processo que os alocou; Somente o processo que alocou o recurso pode liberá-lo;

4. Condição de espera circular: Deve ser uma cadeia circular de dois ou mais processos; Cada um está à espera de recurso retido pelo membro seguinte dessa cadeia;

4. PROCESSOS LINUX

Quando falamos em processos, podemos interpretar de maneira mais simples este conceito dizendo que, processos nada mais são do que execuções de tarefas por parte do próprio sistema operacional (tarefas internas ao sistema operacional) como tarefas solicitadas pelo usuário (execução de aplicativos, impressões...). Durante a execução de um processo, este utiliza diversos recursos de sistema como: processador, memória... para que se tenha uma divisão igualitária dos recursos entre os processos concorrentes o LINUX deve saber exatamente o que cada um dos processos está fazendo no sistema, ou seja, ele precisa saber qual seu estado de execução, informação esta que o LINUX encontra junto ao seu descritor de processos - *task_struct*, (estrutura complexa que possui como componentes uma série de campos como ponteiros para áreas de memória, informações de escalonamento temporizadores e contexto de processos...) usado pelo *scheduler* (escalonador) para racionalizar o uso de recursos do sistema.

Do ponto de vista do kernel, um processo é uma entrada na tabela de processos. Nada mais. A tabela de processos, então, é uma das mais importantes estruturas de dados no sistema, conjuntamente com a tabela de gerenciamento de memória e o *buffer cache*. O item individual na tabela de processos é a estrutura *task_struct*, definida em include/linux/sched.h. Com a *task_struct*, tanto informações de baixo quanto de alto nível, são mantidas.

Depois do boot, o kernel está sempre trabalhando em um dos processos, e a variável global "*current*", um ponteiro para cada item da *task_struct*, é usado para guardar o processo que está rodando.

Então no Linux, cada processo é formado por uma estrutura de dado chamada *task_struct*.

Quando um processo é criado, ele tem um apontador para ele incluso no vetor de apontadores de processo, para conseguir acessar cada processo e decidir que ação tomar. Então, quando um processo é criado, uma nova *task_struct* é alocada do sistema de memória e adicionada no arranjo de ponteiros (vetor do processo). Esse vetor contém 512 apontadores. Isto quer dizer que o Linux suporta no máximo 512 processos rodando ao mesmo tempo no sistema operacional.

O Linux suporta dois tipos de processos: os normais e os de tempo real. Os processos de tempo real são aqueles que necessitam estar rodando constantemente e necessitam de estado de espera menores que os outros processos.

A estrutura *task_struc* é apontada pelo apontador do vetor de processos do Linux. Esta estrutura contém áreas especificas para separar as regiões de um processo. Essas áreas servem tanto para guardar informações sobre o processo quanto para armazenar informações que serão úteis para realização de tarefas pelo sistema operacional.

4.1. Criação de processos

Nenhum processo é criado a partir do zero no Linux, a não ser o processo inicial chamado INIT. Os outros processos são todos clonados a partir desse processo inicial os quais serão responsáveis pela gerência do sistema.

A criação de um processo realiza-se da seguinte forma: um processo já existente se duplica através da chamada de sistema FORK(). O novo processo se chama processo filho e tem seu código substituído pelo código que deve ser executado através da chamada de sistema EXEC() (esta chamada permite ao processo filho assumir seu próprio conteúdo apagando de si o conteúdo do processo pai).

Depois de executar o FORK(), duas copias do mesmo programa estão rodando. Uma delas usualmente executa- EXEC() - outro programa. A chamada a sistema EXEC() deve localizar a imagem binária do arquivo executável, carregá-lo e executá-lo.

4.2. Processos em *Foreground* e *Background*

O Linux permite dois modos de execução de programas em modo normal:

- ***Foreground***: enquanto o processo especificado na linha de comando não termina, o *shell* não libera o *prompt* para o usuário, impedindo o disparo de novos processos.

- ***Background***: vários programas ou comandos podem ser executados, independente do término de cada um deles. Este modo de execução é especificado pelo símbolo & no final da linha de comando.

Exemplo:

$ netscape &

4.3. Término de Processos

Cada processo deve notificar o sistema sobre seu término invocando o processo EXIT(), que é chamado automaticamente quando do término da última linha de execução do processo ou ainda quando colocada explicitamente por programadores no fim de seu código. No momento do término, o sistema libera o descritor de processos e todos os recursos alocados para entrada de novo processo.

Outra forma de término de processo é pela chamada de sistema KILL().

O Linux permite que um processo seja suspenso temporariamente. Quando se congela um processo ocorre uma parada momentânea de sua execução. A suspensão de um processo pode ser feita por um usuário ou pelo sistema operacional. O Sistema pode suspender um processo quando, por exemplo, há coalescência ou uma sobrecarga. No Linux o comando para suspender um processo é o Control+Z.

4.4. Escalonador Linux

O escalonador LINUX é baseado em *time-sharing*, ou seja, atua na divisão do tempo de processador entre os processos e para tanto utiliza um *Scheduler*.

Scheduler é o programa encarregado de agendar os processos, isto é, ele deve escolher o próximo processo que vai rodar, deve decidir quando o tempo de um processo terminou o que fazer com um processo quando ele requisita I/O e assim por diante. Ele é chamado de vários pontos do programa, como após colocar o processo corrente em uma fila de espera, no final de uma chamada de sistema ou qualquer outro momento em que se faz necessário escalonar os processos. Ao ser chamado, o *scheduler* tem uma seqüência de ações que devem ser tomadas para que seu trabalho possa ser feito. Essas ações se dividem em:

- *Kernel Work*: o *scheduler* deve realizar uma série de rotinas específicas do *kernel* e deve tratar da fila de esperas de tarefas do *scheduler*.

- Seleção de processo: o *scheduler* deve escolher o processo que irá rodar. A prioridade é o meio pelo qual ele escolhe.

- Troca de processos: o *scheduler* salva as condições que o processo atual apresenta (contexto específico do processo) e carrega o contexto do novo processo que irá rodar.

Em ambientes multiprocesados (SMP - *Simultaneous Multi Processing*), cada processador tem um *scheduler* para tratar separadamente quais processos irão rodar nele. Dessa forma, cada processo guarda informação sobre o processador atual e o último processador em que rodou.

Processo que já tenham rodado em um processador tem preferência em relação aqueles que não tenham rodado ali ainda. Essa implementação permite um ligeiro acréscimo de ganho no desempenho do sistema.

4.4.1. PRIORIDADES

O LINUX trabalha com dois tipos de prioridade:

- **Estática**: exclusiva de processos em tempo real, neste caso a prioridade é definida pelo usuário e não é modificada pelo escalonador. Somente usuários com privilégios especiais no sistema podem definir processo de tempo real.

- **Dinâmica**: aplicada aos demais, sendo sua prioridade calculada em função da prioridade base do processo e a quantidade de tempo que lhe resta para execução.

Os processos de prioridade estática recebem prioridade maior que os de dinâmica.

As faixas de prioridade variam numa escala de -20 a +20. A prioridade padrão de uma tarefa é 0, com -20 sendo a mais alta. Só o administrador pode reajustar a prioridade de um processo para ser menor que 0, mas os usuários normais podem ajustar prioridades no alcance positivo. Este é usando após o comando "renice", entretanto internamente o Linux usa um *quantum* contador de tempo registrado no *task_struct*.

Processos novos herdam a prioridade de seus pais.

4.4.2. ALGORITMOS DE ESCALONAMENTO

Primeira opção é o PREEMPTIVO: cada processo tem a sua prioridade (créditos iniciais). O escalonador escolhe o processo com o maior número de créditos e o executa até que fique com zero crédito.

Como segunda opção, o Linux implementa duas políticas de escalonamento: FIFO e ROUND ROBIN.

4.4.3. POLÍTICAS DE ESCALONAMENTO

Existem critérios para o escalonamento dos processos em Linux:

- ***Policy***: Pode haver duas políticas de escalonamento *round-robin* e *first-in-first-out* (FIFO).

- ***Priority***: A prioridade do processo é dada de acordo com o tempo que ele gastou para executar (em *jiffies*[1]).

- ***Real time priority***: Esse recurso é usado para processo de tempo real. Através disso, os processos de tempo real podem ter prioridade relativa dentro desse conjunto. A prioridade pode ser alterada através de chamadas do sistema.

- ***Counter***: É a quantidade de tempo (em *jiffies*) que os processos têm permissão para rodar. É setada a prioridade quando o processo é rodado pela primeira vez e decrementada a cada *tick* do *clock*.

4.5. Comandos no Linux para administração de processos

top: verifica processos, memória e processadores;

htop: idem top porém interativo com o usuário;

ps: mostra os processos em andamento;

kill, **killall**: mata um ou todos os processos;

renice: altera uma prioridade de funcionamento;

time: roda um programa e informa o tempo que usou para tal fim;

Strace: mostra as chamadas realizadas por um comando.

[1] *Jiffies* é uma variável que indica a quantidade de tempo que um processo pode ser executado, onde cada valor atribuído depende de cada máquina. Quanto maior o tempo em uma execução anterior, menor a prioridade do processo.

4.5.1. TOP

Permite que sejam visualizadas as informações dos processos e estados de memórias e processadores.

É uma ferramenta de tempo real e, portanto não precisa ser atualizada para ver os diferentes momentos de leitura.

Permite a verificação de trabalho de cada um dos processadores em forma opcional.

Os parâmetros mais importantes são:

-p id só visualiza os dados desse processo

-H mostra as threads vinculadas com tal processo

4.5.2. HTOP

Possui recursos similares aos do comando top.

O comando htop pode não estar disponível nas instalações padrão do Linux. O uso de aplicativos como apt-get (para distribuições Debian e derivadas) ou yum (para Red-hat ou derivadas) podem fazer a instalação do item em questão de poucos minutos.

4.5.3. PS

Permite a visualização de um ou todos os processos em andamento. A variedade de parâmetros faz impossível a reprodução dos mesmos aqui. Destacam-se:

-e para a visualização de todos os parâmetros;

-H para a visualização em forma de árvore hierárquica de processos;

-F para visualização de threads

4.5.4. KILL e KILLALL

São comandos que acabam com processos em andamento. O comando kill mata um único processo informado pelo id.

O comando killall matará todos os comandos que obedeçam a um nome determinado.

4.5.5. RENICE

Altera a prioridade de funcionamento de um determinado processo. Os processos podem ser individuais, de um usuário ou de um grupo.

4.5.6. STRACE

Este comando realiza o trace (roteamento) de todas as chamadas de sistema que são feitas num determinado comando.

A saída deste comando pode ser realizada a um arquivo. Os dados resultantes são a seqüência de chamadas de threads, com sua posição de memória, id de processos e um conjunto de dados vinculados.