Ao executar `exit()`, o sistema operacional realiza uma série de procedimentos relacionados ao processo que está terminando. Nesse instante, o sistema operacional libera todos os recursos que o processo alocou para si, liberando memória, encerrando arquivos abertos e tornando disponível o descritor de processos. Uma outra forma de terminar a execução de um processo é através da chamada de sistema `kill()`.

9.3.4 Uma palavra sobre threads

Um conceito atualmente bastante em voga em sistemas operacionais é o conceito de *threads*. Esse interesse por *threads* está associado com o advento de máquinas multiprocessadoreas (SMP) e com a facilidade de exprimir atividades concorrentes. Uma ***thread*** é usualmente definida como um fluxo de controle no interior de um processo.

Quando um novo processo é criado, como nós já vimos, o núcleo do Linux copia os atributos do processo corrente para o que está sendo criado. É o procedimento de `fork-exec`. O Linux, entretanto, prevê uma segunda forma de criação de processos: a clonagem. Um processo clone compartilha os recursos (arquivos abertos, memória virtual, etc.) com o processo original. Quando dois ou mais processos compartilham as mesmas estruturas, eles atuam como se fossem diferentes *threads* no interior de um único processo. O Linux não diferencia as estrutura de dados de *threads* e de processos, e, por conseqüência, ambos são tratados indistintivamente por todos os mecanismos de gerência do núcleo. Essa característica é mais visível no escalonamento: *threads* e processos são tratados da mesma forma. A vantagem de criar *threads* está associada ao seu custo de criação (tempo): elas são criadas mais rapidamente que processos pois não necessitam copiar os atributos do processo original, basta inicializar ponteiros de seu descritor de processos de forma que eles referenciem as áreas já existentes do processo que está sendo clonado. Como as *threads* do Linux são reconhecidas pelo núcleo, elas se enquadram no modelo de *threads* 1:1, conforme a classificação apresentada no capítulo sobre programação concorrente.

9.4 Escalonamento em Linux

O problema básico de escalonamento em sistemas operacionais é como satisfazer simultaneamente objetivos conflitantes: tempo de resposta rápido, bom *throughput* para processos *background*, evitar postergação indefinida, conciliar processos de alta prioridade com de baixa prioridade, etc. O conjunto de regras utilizado para determinar como, quando e qual processo deverá ser executado é conhecido como política de escalonamento. Nesta seção, nós estudaremos como o Linux implementa seu escalonador e qual a política empregada para determinar quais processos recebem o processador.

Tradicionalmente, os processos são divididos em três grandes classes: processos interativos, processos *batch* e processos tempo real. Em cada classe, os processos podem ser ainda subdivididos em *I/O bound* ou *CPU bound* de acordo com a proporção de tempo que ficam esperando por operações de entrada e saída ou utilizando o processador. O escalonador do Linux não distingue processos interativos de processos *batch*, diferenciando-os apenas dos processos tempo real. Como todos os outros escalonadores UNIX, o escalonador Linux privilegia os processos *I/O bound* em relação aos *CPU bound* de forma a oferecer um melhor tempo de resposta às aplicações interativas.

O escalonador do Linux é baseado em *time-sharing*, ou seja, o tempo do processador é dividido em fatias de tempo (*quantum*) as quais são alocadas aos processos. Se, durante a execução de um processo, o *quantum* é esgotado, um novo processo é selecionado para execução, provocando então uma troca de

contexto. Esse procedimento é completamente transparente ao processo e baseia-se em interrupções de tempo. Esse comportamento confere ao Linux um escalonamento do tipo preemptivo.

O algoritmo de escalonamento do Linux divide o tempo de processamento em épocas (*epochs*). Cada processo, no momento de sua criação, recebe um *quantum* calculado no início de uma época. Diferentes processos podem possuir diferentes valores de *quantum*. O valor do *quantum* corresponde à duração da época, e essa, por sua vez, é um múltiplo de 10 ms inferior a 100 ms.

Outra característica do escalonador Linux é a existência de prioridades dinâmicas. O escalonador do Linux monitora o comportamento de um processo e ajusta dinamicamente sua prioridade, visando a equalizar o uso do processador entre os processos. Processos que recentemente ocuparam o processador durante um período de tempo considerado "longo" têm sua prioridade reduzida. De forma análoga, aqueles que estão há muito tempo sem executar recebem um aumento na sua prioridade, sendo então beneficiados em novas operações de escalonamento.

Na realidade, o sistema Linux trabalha com dois tipos de prioridades: estática e dinâmica. As prioridades estáticas são utilizadas exclusivamente por processos de tempo real correspondendo a valores na faixa de 1-99. Nesse caso, a prioridade do processo tempo real é definida pelo usuário e não é modificada pelo escalonador. Somente usuários com privilégios especiais têm a capacidade de criar e definir processos tempo real. O esquema de prioridades dinâmicas é aplicado aos processos interativos e *batch*. Aqui, a prioridade é calculada, considerando-se a prioridade base do processo e a quantidade de tempo restante em seu *quantum*.

O escalonador do Linux executa os processos de prioridade dinâmica apenas quando não há processos de tempo real. Em outros termos, os processos de prioridade estática recebem uma prioridade maior que os processos de prioridade dinâmica. Para selecionar um processo para execução, o escalonador do Linux prevê três políticas diferentes :

- **SCHED_FIFO**: Essa política é valida apenas para os processos de tempo real. Na criação, o descritor do processo é inserido no final da fila correspondente à sua prioridade. Nessa política, quando um processo é alocado ao processador, ele executa até que uma de três situações ocorra: (i) um processo de tempo real de prioridade superior torna-se apto a executar; (ii) o processo libera espontaneamente o processador para processos de prioridade igual à sua; (iii) o processo termina, ou bloqueia-se, em uma operação de entrada e saída ou de sincronização.

- **SCHED_RR**: Na criação, o descritor do processo é inserido no final da fila correspondente à sua prioridade. Quando um processo é alocado ao processador, ele executa até que uma de quatro situações ocorra: (i) seu período de execução (*quantum*) tenha se esgotado nesse caso o processo é inserido no final de sua fila de prioridade; (ii) um processo de prioridade superior torna-se apto a executar; (iii) o processo libera espontaneamente o processador para processos de prioridade igual a sua; (iv) o processo termina, ou bloqueia-se, em uma operação de entrada e saída ou de sincronização. Essa política também só é válida para processos de tempo real.

- **SCHED_OTHER**: Corresponde a um esquema de filas multinível de prioridades dinâmicas com *timesharing*. Os processo interativos e *batch* recaem nessa categoria.

A criação de processos em Linux, como em todos os sistemas UNIX, é baseada em uma operação do tipo `fork-exec`, ou seja, um processo cria uma cópia sua (`fork`) e em seguida substitui o seu código por um outro (`exec`). No momento da criação, o processo pai (o que fez o `fork`) cede metade de seu

quantum restante ao processo filho. Esse procedimento é, na verdade, uma espécie de proteção que o sistema faz para evitar que um usuário, a partir de um processo pai, crie um processo filho que execute o mesmo código do pai. Sem essa proteção, a cada criação o filho receberia um novo *quantum* integral. Da mesma forma, o núcleo Linux previne-se contra o fato de um mesmo usuário lançar vários processos em *background*, ou executar diferentes sessões `shell`.

O escalonador do Linux é executado a partir de duas formas diferentes. A primeira é a forma direta através de uma chamada explícita à rotina que implementa o escalonador. Essa é a maneira utilizada pelo núcleo do Linux quando, por exemplo, detecta que um processo deverá ser bloqueado em decorrência de uma operação de entrada e saída ou de sincronização. A segunda forma, denominada de *lazy*, também é conseqüência do procedimento de escalonamento, ocorrendo tipicamente em uma de três situações.

A primeira dessas situações é a rotina de tratamento de interrupção de tempo que atualiza os temporizadores e realiza a contabilização de tempo por processo. Essa rotina, ao detectar que um processo esgotou seu *quantum* de execução aciona o escalonador para que seja efetuada uma troca de processo. A segunda situação ocorre quando um processo de mais alta prioridade é desbloqueado pela ocorrência do evento que esperava. A parte do código que efetua o desbloqueio, isto é, trata os eventos de sincronização e de entrada e saída, consulta a prioridade do processo atualmente em execução e compara-a com a do processo que será desbloqueado. Se o processo a ser desbloqueado possuir uma prioridade mais alta, o escalonador é acionado, e ocorre uma troca de processo. A terceira, e última forma de execução *lazy*, é quando um processo explicitamente invoca o escalonador através de uma chamada de sistema do tipo *yield*. Essa chamada de sistema permite a um processo "passar sua vez de execução" a outro processo, e, para isso, parece claro, é necessário executar o escalonador.

9.5 Gerência de Memória

O subsistema de gerência de memória constitui uma das partes mais importantes e críticas dos sistemas operacionais em geral. A necessidade de utilização de mais memória que a fisicamente disponível em uma máquina tornou-se uma constante no projeto de sistemas operacionais, e uma série de métodos foram criados para solucionar esse problema. A memória virtual é sem dúvida o melhor. Essa técnica baseia-se, essencialmente, na tradução de endereços lógicos em endereços físicos auxiliada por mecanismos implementados no hardware do próprio processador. Os dois mecanismos básicos de tradução são a paginação e a segmentação.

Embora alguns processadores, entre eles os da família Intel, suportem a segmentação, o Linux utiliza muito pouco. Entre as razões pelas quais o Linux não explora a segmentação, estão: a gerência da paginação é mais simples que a da segmentação; nem sempre o hardware dos processadores oferecem bom suporte à segmentação; e, além disso, normalmente é possível transformar a segmentação em paginação, mapeando-se todo o espaço virtual em um único segmento. Nas próximas seções, nós iremos analisar mais detalhadamente o modelo de memória empregado pelo Linux e seus mecanismos associados.

9.5.1 Memória virtual

Um processo UNIX possui um modelo de memória organizado em quatro partes: texto, dados não inicializados, dados inicializados e pilha. A área de texto corresponde ao código do programa. A área de dados – inicializados ou não - é composta pelo espaço de armazenamento necessário às variáveis alocadas estaticamente no programa. A área de pilha fornece o espaço de memória necessário às variáveis automáticas (locais), para a passagem de parâmetros, e ainda, para salvar e restaurar endereços de retorno dando suporte a sub-rotinas. As áreas de texto e de dados ocupam a parte baixa de memória, e a