Lista de Exercícios -Sistemas Operacionais

Entrega 13/06

Threads

O livro base para responder aos exercícios é:

SILBERSCHATZ, Abraham; Peter Baer Galvin, Greg Gagne. Fundamentos de Sistemas Operacionais. 9 ed. Rio de Janeiro: LTC, 2015.

Aluno: JUAN CARLOS RIBEIRO FIUZA

Matricula: UC23102826

1. Forneça dois exemplos de programação em que a criação de múltiplos threads proporciona melhor desempenho do que uma solução com um único thread.

- **Servidor Web:** Um servidor web precisa atender a múltiplos pedidos de clientes simultaneamente. Com uma abordagem multithread, o servidor pode criar um thread separado para cada requisição. Enquanto um thread está bloqueado aguardando uma operação de I/O (entrada/saída), como ler um arquivo do disco para enviar a um cliente, outros threads podem continuar executando e atendendo a outras requisições. Em uma solução de thread único, cada requisição teria que esperar a anterior ser completamente finalizada, resultando em longos tempos de espera para os clientes.
- **Aplicações de Interface Gráfica (GUI):** Em uma aplicação com interface gráfica, é crucial que a interface permaneça responsiva aos comandos do usuário (cliques de mouse, digitação). Se uma tarefa demorada (como um cálculo complexo, renderização de uma imagem ou acesso a um banco de dados) for executada no mesmo thread da interface, a aplicação inteira irá "congelar" até que a tarefa seja concluída. Utilizando múltiplos threads, a tarefa demorada pode ser executada em um thread de "background" (plano de fundo), enquanto o thread principal continua a gerenciar a interface do usuário, garantindo uma experiência fluida e responsiva.

2. Cite duas diferenças entre threads de nível de usuário e threads de nível de kernel.

1. **Gerenciamento e Reconhecimento pelo Kernel:**
   - **Threads de Nível de Usuário (ULTs):** São gerenciadas por uma biblioteca de threads no espaço do usuário, sem o conhecimento direto do kernel do sistema operacional. Para o kernel, o processo que contém esses threads é visto como uma única entidade de execução.

- **Threads de Nível de Kernel (KLTs):** São gerenciadas diretamente pelo kernel do sistema operacional. O kernel está ciente de cada thread e é responsável por seu escalonamento, criação e sincronização.

2. **Impacto de Chamadas de Sistema Bloqueantes:**
   - **Threads de Nível de Usuário:** Se um thread de nível de usuário realiza uma chamada de sistema que causa um bloqueio (como uma operação de I/O), todo o processo é bloqueado pelo kernel, impedindo que qualquer outro thread do mesmo processo continue a execução.
   - **Threads de Nível de Kernel:** Se um thread de nível de kernel realiza uma chamada de sistema bloqueante, o kernel pode escalonar outro thread do mesmo processo para execução (se houver um disponível e o sistema tiver múltiplos processadores ou tempo de CPU ocioso), pois ele gerencia os threads individualmente.

3. Sob que circunstâncias um tipo é melhor do que o outro?

- **Threads de Nível de Usuário são melhores quando:**
  - A troca de contexto entre threads precisa ser extremamente rápida, pois não requer uma chamada ao kernel (uma mudança de modo privilegiado), sendo realizada por uma biblioteca em espaço de usuário.
  - O sistema operacional não oferece suporte nativo a threads de nível de kernel.
  - A aplicação não realiza um número significativo de chamadas de sistema bloqueantes.
- **Threads de Nível de Kernel são melhores quando:**
  - A aplicação necessita tirar proveito de arquiteturas multiprocessadas ou multi-core, pois o kernel pode agendar diferentes threads do mesmo processo para executar em paralelo em diferentes processadores.
  - É esperado que os threads realizem operações de I/O ou outras chamadas de sistema bloqueantes com frequência. A capacidade do kernel de escalonar outro thread enquanto um está bloqueado é crucial para o desempenho.
  - É necessária uma gestão de prioridades mais robusta e controlada pelo sistema operacional.

4. Descreva as ações executadas por um kernel para mudar o contexto entre threads de nível de kernel.

A mudança de contexto entre threads de nível de kernel é um processo mais leve do que a mudança de contexto entre processos, mas envolve etapas críticas gerenciadas pelo kernel:

1. **Salvar o Contexto do Thread Atual:** O kernel salva o estado do thread que está atualmente em execução. Isso inclui o conteúdo dos registradores da CPU (como o contador de programa, ponteiro de pilha e registradores de uso geral) na estrutura de dados do thread, conhecida como Bloco de Controle de Thread (TCB - Thread Control Block).

2. **Selecionar o Próximo Thread:** O escalonador do kernel escolhe o próximo thread a ser executado da fila de threads prontos, com base em seu algoritmo de escalonamento (por exemplo, prioridade, round-robin, etc.).
3. **Restaurar o Contexto do Novo Thread:** O kernel carrega o estado do novo thread selecionado a partir de seu TCB para os registradores da CPU. O contador de programa é carregado com o endereço da instrução onde o novo thread foi interrompido anteriormente, e o ponteiro de pilha é atualizado para apontar para a pilha correta. Após esta etapa, o novo thread retoma sua execução.

5. Que recursos são usados quando um thread é criado? Em que eles diferem daqueles usados quando um processo é criado?

- **Recursos Usados na Criação de um Thread:** A criação de um thread é considerada "leve". Os principais recursos alocados são:
  - Uma **pilha (stack)** para variáveis locais e chamadas de função.
  - Um conjunto de **registradores** para manter o estado da CPU.
  - Um **identificador de thread** e um **bloco de controle de thread (TCB)** para gerenciamento pelo sistema.
- **Diferença em Relação à Criação de um Processo:** A criação de um processo é uma operação muito mais "pesada" e consome mais recursos. Um novo processo requer a alocação de:
  - Um **espaço de endereçamento de memória completo e exclusivo**, que inclui as seções de código, dados e heap.
  - Um **Bloco de Controle de Processo (PCB)**, que contém informações mais extensas do que um TCB.
  - Descritores de arquivos, informações de estado, e outros recursos do sistema operacional.

A principal diferença é que **threads dentro do mesmo processo compartilham recursos**, como a memória do heap, variáveis globais e o segmento de código do processo pai. Processos, por outro lado, são isolados e não compartilham esses recursos por padrão.

6. Suponha que um sistema operacional mapeie threads de nível de usuário para o kernel, usando o modelo muitos-para-muitos e que o mapeamento seja feito por LWPs. Além disso, o sistema permite que os desenvolvedores criem threads de tempo real para uso em sistemas de tempo real. É necessário vincular um thread de tempo real a um LWP? Explique.

Sim, é necessário vincular um thread de tempo real a um LWP (Lightweight Process).

Explicação: No modelo muitos-para-muitos, múltiplos threads de nível de usuário são multiplexados em um número menor ou igual de LWPs. Um LWP é a "ponte" pela qual os threads de usuário acessam o kernel e são escalonados na CPU.

Sistemas de tempo real exigem que as tarefas sejam concluídas dentro de prazos estritos e previsíveis. Se um thread de tempo real não estiver vinculado (bound) a um LWP

específico, ele compete com outros threads de usuário pelo acesso a um LWP disponível. Isso pode introduzir um atraso não determinístico (latência), pois o thread de tempo real pode ter que esperar que um LWP seja liberado por outro thread antes de poder ser executado. Essa imprevisibilidade é inaceitável em um ambiente de tempo real.

Ao **vincular** o thread de tempo real a um LWP dedicado, garante-se que ele tenha um caminho direto e imediato para o kernel. Isso permite que o escalonador do sistema operacional trate esse LWP (e, por consequência, o thread de tempo real) com a prioridade necessária, garantindo que ele possa ser executado imediatamente quando necessário para cumprir seus prazos críticos, sem a contenção de outros threads de usuário.

7. Qual dos seguintes componentes de estado de um programa são compartilhados pelos threads em um processo com múltiplos threads? a. Valores do registrador b. Memória do heap c. Variáveis globais d. Memória da pilha

Resposta: Os componentes compartilhados são:

- **b. Memória do heap**
- **c. Variáveis globais**

Cada thread possui seus próprios valores de registrador e sua própria memória de pilha para manter seu estado de execução e chamadas de função independentes.

8. Uma solução com múltiplos threads usando múltiplos threads de nível de usuário pode obter melhor desempenho em um sistema multiprocessador do que em um sistema uniprocessador? Explique.

Não. Uma solução que usa *apenas* threads de nível de usuário (modelo muitos-para-um) não consegue obter melhor desempenho em um sistema multiprocessador.

Explicação: O kernel do sistema operacional não tem conhecimento da existência de threads de nível de usuário; ele apenas enxerga o processo que os contém como uma única thread de execução. Portanto, o kernel só pode escalonar esse processo inteiro para ser executado em um único núcleo (processador) de cada vez. Mesmo que o sistema tenha múltiplos processadores, os threads de nível de usuário dentro daquele processo só podem executar de forma concorrente (alternando rapidamente em um único núcleo), mas nunca em paralelo (simultaneamente em múltiplos núcleos). O paralelismo verdadeiro, que é a principal fonte de ganho de desempenho em sistemas multiprocessadores, não pode ser alcançado com este modelo.

9. É possível haver concorrência, mas não paralelismo? Explique.

Sim, é perfeitamente possível.

Explicação:

- **Concorrência:** Refere-se à capacidade de um sistema gerenciar múltiplas tarefas que progridem ao mesmo tempo. Em um sistema com um único núcleo de processamento, a concorrência é alcançada através da alternância rápida de execução entre as tarefas (context switching). As tarefas estão "em andamento" em períodos de tempo sobrepostos, mas não estão sendo executadas no mesmo instante.
- **Paralelismo:** Refere-se à capacidade de um sistema executar múltiplas tarefas literalmente ao mesmo tempo (simultaneamente). Isso requer hardware com múltiplos núcleos de processamento ou múltiplos processadores.

Portanto, um sistema **uniprocessador** pode executar um programa multithread de forma **concorrente**, mas **não paralela**. Os threads alternam a execução no único núcleo disponível, dando a ilusão de simultaneidade, mas em qualquer instante de tempo, apenas um thread está de fato sendo executado.

10. Considere o segmento de código a seguir:

```
C
  pid_t pid;
  pid = fork();
  if (pid == 0) {/*processo-filho */ [cite: 13]
     fork();
     thread_create( … );
  }
  fork(); [cite: 14]
```

a. Quantos processos únicos são criados?

São criados **4 processos únicos**.

Análise:

1. **Processo Original (P0):** Inicia a execução.
2. **pid = fork(); (Primeiro fork):** P0 cria um processo filho, **P1**. Agora temos P0 e P1.
3. **if (pid == 0):** Este bloco é executado apenas pelo processo filho P1.
    - **fork(); (Segundo fork):** Dentro do if, P1 cria um novo filho, **P2**. Agora temos P0, P1 e P2.
4. **fork(); (Terceiro fork):** Esta linha está fora do bloco if e será executada tanto pelo processo original (P0) quanto pelo primeiro filho (P1).
    - P0 executa o fork() e cria um novo filho, **P3**.
    - P1 também executa este fork() e cria um novo filho, **P4**.

Os processos criados são P1, P2, P3 e P4. Total: **4 processos criados**. O número total de processos existentes ao final será 5 (o original + os 4 criados).

b. Quantos threads únicos são criados?

É criado **1 thread único**.

Análise: A função thread_create(...) é chamada apenas uma vez. Ela está localizada dentro do bloco if, que por sua vez é executado apenas pelo processo P1. Dentro desse bloco, a chamada thread_create() ocorre *depois* do segundo fork(), no fluxo de execução do processo P2 (o filho de P1). Portanto, apenas o processo P2 criará um novo thread.

11. Especifique como se dá a implementação de Threads no Windows, Posix e Java.

- **Windows:**
  - **Modelo:** O Windows utiliza um **modelo de mapeamento um-para-um (1:1)**. Cada thread criado por uma aplicação é mapeado diretamente para um thread de nível de kernel, que é gerenciado e escalonado pelo kernel do Windows.
  - **Implementação:** A API do Windows (Win32/Win64) fornece funções como CreateThread para criar threads. O kernel mantém estruturas de dados complexas para cada thread, incluindo o Bloco de Thread Executivo (ETHREAD), o Bloco de Thread do Kernel (KTHREAD) e o Bloco de Ambiente de Thread (TEB) no espaço do usuário. Esse modelo permite o verdadeiro paralelismo em sistemas multi-core e um controle granular sobre os threads, embora a criação de threads seja uma operação mais custosa do que em modelos de nível de usuário.
- **POSIX (Pthreads):**
  - **Modelo:** POSIX (Portable Operating System Interface) não é uma implementação, mas sim um **padrão de API (IEEE 1003.1c)** para a criação e manipulação de threads. As implementações podem variar. No entanto, a maioria dos sistemas operacionais modernos baseados em UNIX, como **Linux e macOS, implementa Pthreads usando um modelo um-para-um (1:1)**, semelhante ao Windows.
  - **Implementação:** Os programadores utilizam a biblioteca Pthreads (com funções como pthread_create e pthread_join) para escrever código multithread portátil. No Linux, a NPTL (Native POSIX Thread Library) mapeia cada pthread diretamente para uma "tarefa" (task) do kernel, que é a unidade de escalonamento do sistema. Isso permite que pthreads se beneficiem diretamente do escalonador do kernel e executem em paralelo em sistemas multiprocessados.
- **Java:**
  - **Modelo:** Os threads em Java são gerenciados pela **Máquina Virtual Java (JVM)**. O modelo de implementação depende da JVM e do sistema operacional subjacente. Nas primeiras versões do Java, era comum o uso de "Green Threads", um modelo **muitos-para-um (N:1)** onde a JVM gerenciava muitos threads de usuário em um único thread do sistema operacional.
  - **Implementação Atual:** Hoje, a grande maioria das implementações modernas da JVM (como a HotSpot) utiliza um **modelo um-para-um (1:1)**. Cada thread Java (java.lang.Thread) é mapeado diretamente para um thread nativo do sistema operacional hospedeiro (como um KLT no Windows ou uma task no Linux). A JVM gerencia o ciclo de vida do objeto Thread, mas o escalonamento real é delegado ao kernel do SO. Com o Java 21, foram introduzidos os "Threads Virtuais" (Projeto Loom), que implementam um

modelo muitos-para-muitos (M:N) para tarefas leves, mas os threads de plataforma tradicionais continuam seguindo o modelo 1:1.