UNIVERSIDADE DO VALE DO ITAJAÍ
CAMPUS ITAJAÍ
CIÊNCIA DA COMPUTAÇÃO
DISCIPLINA SISTEMAS OPERACIONAIS
PROFESSOR FELIPE VIEL
PROCESSOS, THREADS, CONCORRÊNCIA E PARALELISMO

Eduardo Savian de Oliveira, Marcos Augusto Fehlauer Pereira, Yuri Rodrigues

Itajaí

01/04/2023

Sumário

Introdução

Uma Indústria de Alimentos de Santa Catarina chamada FoodSec S.A. possui a tarefa de escanear alimentos por meio de câmeras e verificar se os mesmos estão corretos. Os alimentos podem passar por uma das duas esteiras disponíveis. As duas esteiras são controladas por um por um único computador centralizado. Esse computador recebe dados de um sensor em cada uma das esteiras que captura a contagem de itens que são identificados como seguros.

A contagem é exibida em um display perto das esteiras (todos os displays são controlados pela mesma função, é apenas uma replicação). Além disso, os pesos dos itens que passam por cada uma das esteiras são armazenados em um único vetor de dados. A cada 500 unidades de produtos, das duas esteiras, é necessário atualizar o peso total de itens processados. Sendo assim, a empresa aceita uma pausa na quantidade de itens sendo contados e pesados para realizar a pesagem total, mas ela necessita saber quanto tempo é necessário para isso.

A empresa também fornece uma análise das 2 esteiras:

• Esteira 1: produtos de maior peso (5 Kg) – passa 1 item a cada 2 segundos pelo sensor.

• Esteira 2: produtos de peso médio (2 Kg) – passa 1 item a cada 1 segundo pelo sensor.

• A contagem de peso deve interromper todo o sistema e deve acontecer ininterruptamente.

• A exibição no display atualiza a cada 2 segundos para os operadores poderem acompanhar.

A proposta é implementar três versões da solução e comparar qual é a que apresenta o melhor desempenho quanto ao tempo de computação (tempo de processamento).

A primeira solução é usado IPC do tipo memória compartilhada. A segunda solução é considerada IPC via pipe e a terceira solução é implementada via multi-thread.

Considerando que todos os processos/threads funcionam na mesma máquina (não havendo necessidade de comunicação entre computadores) e se faz simulação da contagem com um simples incremento de variável.

Desenvolvimento

1. Multi-thread:

1.1. push_weight(f64 w) é responsável por adicionar um peso (em kg) ao buffer weigh_in_buffer. Se o buffer estiver cheio, a função zera o buffer e adiciona a soma dos pesos no display.val.

1.2.1 struct Belt é uma estrutura que representa uma esteira da fábrica de alimentos. Cada esteira tem um peso de produto e um atraso para empurrar o produto para o buffer.

1.2.2 A função run de Belt é responsável por empurrar o produto para o buffer weigh_in_buffer a cada delay milissegundos, um certo número de vezes (iter).

1.3 void* belt_run_wrapper(void* belt) é um wrapper que serve para passar a estrutura Belt como argumento para um thread do sistema operacional. Quando o thread começa a ser executado, a função run da Belt é chamada.

1.4 void* display_run_wrapper(void* display) é um wrapper que serve para passar a estrutura Display como argumento para um thread do sistema operacional. Quando o thread começa a ser executado, a função run da Display é chamada.

1.5 int main(int argc, const char** argv) recebe como argumento a quantidade de iterações (G::iterations) que as esteiras devem realizar.

Em seguida, a função cria três threads: uma para a esteira belt0, outra para a esteira belt1 e uma terceira para o display G::display. Cada thread é criado usando a função pthread_create.

Depois de criar os threads, a função espera que as esteiras terminem a execução com a função pthread_join. E, por fim, ela libera a memória alocada para as estruturas Belt e retorna 0.

2. IPC do tipo memória compartilhada:

2.1 show: função que recebe um valor de ponto flutuante de dupla precisão e o imprime na saída padrão, formatado para exibir apenas duas casas decimais.

2.2 Belt: classe que define uma esteira (ou "belt", em inglês) com um certo atraso na inserção de novos elementos. Possui atributos como o atraso (em milissegundos), o peso dos elementos que são adicionados na esteira, um buffer para armazenar os elementos, um acumulador para a soma dos elementos e a capacidade máxima do

buffer. Possui também métodos para adicionar novos elementos na esteira e executar um certo número de iterações, adicionando elementos com base no peso definido e parando por um tempo determinado pelo atraso.

2.3.1 A função main é a função principal do programa e é onde a execução do programa começa. No início da função, o programa verifica se foi fornecido um argumento de linha de comando contendo o número de iterações que o programa deve executar. Se não houver argumentos suficientes, o programa emite uma mensagem de uso na saída de erro e termina a execução. Em seguida, o programa cria dois acumuladores acc0 e acc1 usando a função mmap, que aloca uma região de memória compartilhada entre processos. Esses acumuladores são usados para armazenar a soma dos valores de cada Belt à medida que são processados. Em seguida, o programa cria um novo processo usando a função fork, que cria um novo processo idêntico ao processo pai.

2.3.2 O novo processo criado é chamado de "processo filho" e o processo original é chamado de "processo pai". O processo filho executa o código dentro do primeiro if enquanto o processo pai executa o código dentro do segundo eles. O processo filho cria outro processo filho e, em seguida, cria duas instâncias da classe Belt para cada um desses processos filhos. Cada instância de Belt é criada com um determinado atraso, peso e tamanho de buffer e os métodos add e run são chamados para cada instância para realizar a simulação. Enquanto isso, o processo pai executa um loop para imprimir o valor somado de acc0 e acc1 a cada iteração. Esse loop executa o método show para formatar a saída e, em seguida, pausa por 2 segundos usando a função microsleep. Após o loop for terminar, o programa termina a execução.

3. IPC via PIPE:

3.1 serialize_f64(byte* buf, f64 n) converte um valor de ponto flutuante de precisão dupla (f64) em um array de bytes (byte*), para que possa ser transmitido através do pipe (tubo) entre os processos. A função usa um ponteiro para n para acessar os bytes que representam o valor e, em seguida, copia esses bytes para o array buf.

3.2 struct Belt é uma estrutura que representa uma esteira transportadora. Possui três campos: delay (atraso), que é o tempo em milissegundos que a esteira leva para empurrar um objeto; weight (peso), que é o peso dos objetos transportados; e fifo_path, que é o caminho do pipe usado para enviar valores da esteira para o processo de exibição. A estrutura também possui um método run() que é usado para enviar continuamente valores da esteira pelo pipe.

3.3 void Belt::run() é usado para enviar continuamente valores da esteira pelo pipe. Ele cria um array de bytes para armazenar o valor atual da esteira, converte o valor para um array de bytes usando a função serialize_f64(), abre o pipe para escrita, grava o valor no pipe e, em seguida, fecha o pipe. O método também chama a

função microsleep() para atrasar a execução da esteira antes de enviar o próximo valor.

3.4 Função int main(int argc, const char** argv) do belt.cpp começa verificando se há argumentos suficientes para o programa (deve haver três), e, em seguida, converte os argumentos de string para os tipos de dados corretos (um inteiro sem sinal e um ponto flutuante de precisão dupla). A função então cria uma instância da estrutura Belt e chama o método run() para iniciar o envio de valores da esteira pelo pipe. Finalmente, a função retorna 0 para indicar que o programa foi executado com sucesso.

3.5 Função read_f64(int fd) é responsável por ler um número de ponto flutuante (double precision) de um arquivo descriptor (fd). Ela lê sizeof(f64) bytes do arquivo e converte em um número de ponto flutuante, retornando-o.

3.6 Função show(f64 val) imprime na tela o valor passado como argumento, formatando-o com duas casas decimais entre colchetes, como "[%.2f]".

3.7 Função main(int argc, const char** argv) do display.cpp é responsável por executar o programa principal. Ela recebe um argumento (iter) que determina o número de iterações que serão executadas.Na primeira parte do corpo da função, cria duas named pipes (caminhos FIFO) chamadas "belt0.fifo" e "belt1.fifo" com permissões de leitura e escrita para todos. Em seguida, há um loop que lê um número de ponto flutuante de cada uma das pipes (fd0 e fd1), usando a função read_f64(). Os valores lidos são armazenados em um buffer de tamanho 8 (G::weigh_in_buf). O buffer é usado para armazenar os pesos lidos em pares, de modo que cada índice par (0, 2, 4, ...) corresponde a um valor lido da pipe "belt0.fifo", e cada índice ímpar (1, 3, 5, ...) corresponde a um valor lido da pipe "belt1.fifo". Se o buffer estiver cheio (i.e., G::buf_index + 2 >= G::BUF_SIZE), o programa calcula a média dos valores armazenados no buffer, imprime o resultado na tela usando a função show(), limpa o buffer e reinicia o índice (G::buf_index). Após cada leitura, as pipes são fechadas e há um atraso de 2000 microssegundos (2 milissegundos) antes da próxima iteração. Depois que o loop é executado iter vezes, as named pipes são removidas do sistema e o programa retorna 0.

Conclusão

A escolha da melhor solução entre multi-thread, IPC FIFO e IPC com memória compartilhada depende das características específicas do problema em questão e dos requisitos de desempenho, escalabilidade, confiabilidade e complexidade.

Em geral, multi-threading é uma boa solução quando a tarefa é altamente dependente de operações de CPU e a memória compartilhada não é um problema.

Ele permite que os threads compartilham variáveis e sincronizem operações por meio de semáforos e mutexes, o que pode ser muito eficiente para muitos tipos de problemas.

IPC FIFO é uma boa solução quando a tarefa envolve comunicação entre processos e é necessário garantir a ordem em que as mensagens são enviadas e recebidas. Os FIFOs podem ser implementados com diferentes níveis de complexidade, dependendo das necessidades do problema, e geralmente são muito eficientes e confiáveis.

IPC com memória compartilhada é uma boa solução quando a tarefa envolve compartilhamento de grandes quantidades de dados entre processos ou threads. Ele pode ser mais eficiente do que a comunicação por meio de FIFOs, pois os dados podem ser acessados diretamente na memória compartilhada, sem a necessidade de cópias adicionais.

Na implementação o IPC FIFO obteve o pior resultado, depois memória compartilhada e com a melhor performance do multi-theading. Esse resultado do FIFO foi causado pelo fato do blocking io e a diferença entre os melhores foi causado pelo fato de que acessar a memória é mais custoso do que colocar cadeado no multi-theads.