Trabalho Prático - SO1

CC-ICET-CUA | Semestre: 2025/1 | Prof. Sandino Jardim

Visão geral

Neste trabalho, você desenvolverá um servidor web concorrente. Para simplificar este projeto, será fornecido o código de um servidor web não concorrente (mas funcional). Este servidor web básico opera com apenas uma única thread; seu trabalho será tornar o servidor web multi-threaded para que ele possa lidar com várias solicitações ao mesmo tempo.

Os objetivos deste projeto são:

- Aprender a arquitetura básica de um servidor web simples
- Aprender a adicionar concorrência a um sistema não concorrente
- Aprender a ler e modificar uma base de código existente de forma eficaz

Os objetivos quanto aos conceitos de Sistemas Operacionais aqui exigidos serão:

- **Compreender a concorrência:** Aprender a criar e gerenciar múltiplos threads de execução (`pthread_create`).
- **Dominar a sincronização:** Utilizar variáveis de condição e mutexes (`pthread_mutex_t` , `pthread_cond_t`) para evitar condições de corrida em recursos compartilhados (o buffer de requisições).
- **Implementar agendamento de processos:** Entender como diferentes políticas de agendamento (FIFO, SFF) afetam o desempenho e o comportamento de um sistema.
- **Gerenciar recursos:** Lidar com a contenção de recursos, garantindo que o buffer de requisições não transborde (quando cheio) ou não seja acessado vazio.
- **Otimizar o desempenho:** Analisar os trade-offs entre a criação de threads por requisição versus um pool de threads fixo, um conceito central em sistemas concorrentes.

Leituras úteis do Livro-texto incluem:

- Introdução a threads
- Usando locks
- Relações produtor-consumidor
- Arquitetura de concorrência de servidor

Contexto de HTTP

Antes de descrever o que você irá implementar neste projeto, forneceremos uma breve visão geral de como um servidor web clássico funciona e o protocolo HTTP (versão 1.0) usado para se comunicar com ele, embora navegadores e servidores web tenham evoluído muito ao longo dos anos.

Um navegador se conecta a um servidor web e envia uma requisição HTTP para um arquivo. O servidor processa a requisição, envia a resposta (com o conteúdo do arquivo) e fecha a conexão. A seguir estão listados os principais conceitos relacionados ao tema:

- **HTTP:** O protocolo de texto usado para comunicação.
- **TCP/IP:** O conjunto de protocolos de rede que o HTTP utiliza para garantir a entrega confiável das mensagens.
- **Conteúdo Estático e Dinâmico:** O servidor lida com arquivos estáticos (arquivos no disco) e conteúdo dinâmico (a saída de programas executados no servidor, conhecidos como programas CGI).
- **URLs e Portas:** Arquivos são identificados por URLs (`http://host:porta/caminho/arquivo`). O número da porta especifica em qual serviço o servidor web está rodando.
- **Requisição e Resposta:** Uma requisição HTTP inclui o método (`GET`), o URI e a versão do protocolo. A resposta contém o status (`200 OK`, `404 Not Found`), cabeçalhos (`Content-Type`) e o corpo do conteúdo.

A Solicitação HTTP

A comunicação entre o navegador (cliente) e o servidor é definida pelo protocolo HTTP:

A Requisição HTTP

Uma requisição HTTP, enviada do navegador para o servidor, consiste em:

- Uma linha de requisição no formato `método uri versão`.
  - O **método** mais comum é `GET`.
  - O **URI** é o nome do arquivo.
  - A **versão** indica a versão do protocolo HTTP (ex: HTTP/1.0).
- Zero ou mais cabeçalhos de requisição.
- Uma linha de texto vazia.

A Resposta HTTP

A resposta HTTP do servidor para o navegador é semelhante, contendo:

- Uma linha de resposta no formato `versão status mensagem`.
  - O **status** é um código numérico (ex: `200 OK`, `404 Not Found`).
- Zero ou mais cabeçalhos de resposta, incluindo `Content-Type` (tipo de conteúdo) e `Content-Length` (tamanho do arquivo).
- Uma linha de texto vazia.
- O corpo da resposta, que contém o conteúdo solicitado.

Um Servidor Web Básico

Instruções e Funcionalidades

- **Compilar e Executar:** Para começar, você deve compilar o código usando o comando `make`. Para executar o servidor, você precisa especificar uma porta (exemplo: `./wserver 8000`). Para evitar conflitos, use portas maiores que 1023, pois as portas abaixo desse valor são reservadas.
- **Comunicação:** Para testar o servidor, você pode acessá-lo a partir de um navegador usando um URL como `http://localhost:8003/meu_arquivo.html`, onde `localhost` é o nome do host e `8003` é o número da porta que você especificou. O servidor deve estar rodando na mesma máquina onde você inicia o cliente.
- **Limitações do Código Base:** O código fornecido é mínimo e tem algumas limitações importantes que você deve conhecer:

    - **Métodos HTTP:** Ele lida apenas com requisições do tipo `GET`.
    - **Tipos de Conteúdo:** O servidor entende apenas alguns tipos de conteúdo.
    - **Robustez:** Ele não é muito robusto; uma falha de conexão do cliente pode causar a saída do servidor. Você não precisa corrigir esses problemas, pois o foco do projeto é outro.
- **Segurança:** A seção enfatiza a importância da segurança. Você deve sempre encerrar o servidor após os testes para não criar uma "porta traseira" no seu sistema. Além disso, é crucial garantir que o servidor não permita acesso a arquivos fora do diretório de trabalho inicial, rejeitando nomes de caminho que contenham `..`.
- **Ferramentas de Teste:** Para ajudar no desenvolvimento, são fornecidos outros arquivos:
    - `wclient.c`: Um cliente web simples que pode ser modificado para enviar requisições simultâneas ao seu servidor.
    - `spin.c`: Um programa CGI de exemplo que simula um processo de longa duração, útil para testar a concorrência.

Funcionalidades a serem implementdas

Neste projeto, você adicionará duas peças-chave de funcionalidade ao servidor web básico.

Primeiro, você tornará o servidor web multi-threaded.

Segundo, você implementará diferentes políticas de agendamento para que as solicitações sejam atendidas em ordens diferentes. Você também modificará como o servidor web é invocado para que ele possa lidar com novos parâmetros de entrada (por exemplo, o número de threads a serem criados).

Parte 1: Multi-threaded

O servidor web básico fornecido tem um único thread de controle. Servidores web de thread único sofrem de um problema de desempenho fundamental, pois apenas uma única solicitação HTTP pode ser atendida por vez. Assim, todos os outros clientes que estão acessando este servidor web devem esperar até que a solicitação HTTP atual tenha terminado; isso é especialmente

um problema se a solicitação atual for um programa CGI de longa duração ou estiver residente apenas em disco (ou seja, não está na memória). Assim, a extensão mais importante que você adicionará é tornar o servidor web básico multi-threaded.

A abordagem mais simples para construir um servidor multi-threaded é espalhar um novo thread para cada nova solicitação HTTP. O SO então agendará esses threads de acordo com sua própria política. A vantagem de criar esses threads é que agora solicitações curtas não precisarão esperar que uma solicitação longa seja concluída; além disso, quando um thread é bloqueado (ou seja, esperando a E/S do disco terminar), os outros threads podem continuar a lidar com outras solicitações. No entanto, a desvantagem da abordagem de uma-thread-por-solicitação é que o servidor web é penalizado com a sobrecarga de criar uma nova thread a cada solicitação.

Portanto, a abordagem geralmente preferida para um servidor multi-threaded é criar um pool de tamanho fixo de threads de trabalho quando o servidor web é iniciado pela primeira vez. Com a abordagem de pool de threads, cada thread é bloqueada até que haja uma solicitação HTTP para ela lidar. Então, se houver mais threads de trabalho do que solicitações ativas, algumas das threads serão bloqueadas, esperando que novas solicitações HTTP cheguem; se houver mais solicitações do que threads de trabalho, essas solicitações precisarão ser armazenadas em buffer até que haja uma thread pronta.

Em sua implementação, você deve ter uma thread mestre que começa criando um pool de threads de trabalho, cujo número é especificado na linha de comando. Sua thread mestre é então responsável por aceitar novas conexões HTTP pela rede e colocar o descritor para esta conexão em um buffer de tamanho fixo; em sua implementação básica, a thread mestre não deve ler desta conexão. O número de elementos no buffer também é especificado na linha de comando. Observe que o servidor web na versão atual tem uma única thread que aceita uma conexão e, em seguida, lida imediatamente com a conexão; em seu servidor web, esta thread deve colocar o descritor de conexão em um buffer de tamanho fixo e retornar a aceitar mais conexões.

Cada thread de trabalho é capaz de lidar com solicitações estáticas e dinâmicas. Uma thread de trabalho acorda quando há uma solicitação HTTP na fila; quando há várias solicitações HTTP disponíveis, qual solicitação é tratada depende da política de agendamento, conforme descrito na Parte 2. Uma vez que a thread de trabalho acorda, ela executa a leitura no descritor de rede, obtém o conteúdo especificado (seja lendo o arquivo estático ou executando o

processo CGI) e, em seguida, retorna o conteúdo ao cliente escrevendo no descritor. A thread de trabalho então espera por outra solicitação HTTP.

Observe que a thread mestre e as threads de trabalho estão em uma relação produtor-consumidor e exigem que seus acessos ao buffer compartilhado sejam sincronizados. Especificamente, a thread mestre deve bloquear e esperar até que o buffer esteja cheio; uma thread de trabalho deve esperar até que o buffer esteja vazio. Neste projeto, você é obrigado a usar variáveis de condição, conforme visto na Aula 7. Nota: se sua implementação executar qualquer espera ocupada (ou espera de rotação [spinning]) em vez disso, você será penalizado.

Observação: não se confunda com o fato de que o servidor web básico fornecido gera um novo processo para cada processo CGI que ele executa. Embora, em um sentido muito limitado, o servidor web use múltiplos processos, ele nunca lida com mais de uma única solicitação por vez; o processo pai no servidor web espera explicitamente que o processo CGI filho seja concluído antes de continuar e aceitar mais solicitações HTTP. Ao tornar seu servidor multi-threaded, você não deve modificar esta seção do código.

Parte 2: Políticas de Agendamento

Neste projeto, você implementará uma série de diferentes políticas de agendamento. Observe que quando seu servidor web tiver múltiplas threads de trabalho em execução (especificado na linha de comando), você não terá nenhum controle sobre qual thread é realmente agendada a qualquer momento pelo SO. Seu papel no agendamento é determinar qual solicitação HTTP deve ser tratada por cada um dos threads de trabalho esperando em seu servidor web.

A política de agendamento é determinada por um argumento de linha de comando quando o servidor web é iniciado e são as seguintes:

- First-in-First-out (FIFO): Quando uma thread de trabalho acorda, ele lida com a primeira solicitação (ou seja, a solicitação mais antiga) no buffer. Observe que as solicitações HTTP não necessariamente terminarão em ordem FIFO; a ordem em que as solicitações são concluídas dependerá de como o SO agenda as threads ativos.
- Smallest File First (SFF): Quando um thread de trabalho acorda, ele lida com a solicitação para o menor arquivo. Esta política aproxima-se do Shortest Job First na medida em que o tamanho do arquivo é uma boa

previsão de quanto tempo leva para atender a essa solicitação. Solicitações de conteúdo estático e dinâmico podem ser misturadas, dependendo dos tamanhos desses arquivos. Observe que este algoritmo pode levar à inanição de solicitações para arquivos grandes. Você também notará que a política SFF requer que algo seja conhecido sobre cada solicitação (por exemplo, o tamanho do arquivo) antes que as solicitações possam ser agendadas. Assim, para suportar esta política de agendamento, você precisará fazer algum processamento inicial da solicitação (dica: usando stat() no nome do arquivo) fora dos threads de trabalho; você provavelmente vai querer que a thread mestre execute este trabalho, o que exige que ela leia do descritor de rede.

Parâmetros da Linha de Comando

Seu programa C deve ser invocado exatamente da seguinte maneira:

```
prompt> ./wserver [-d basedir] [-p port] [-t threads] [-b buffers] [-s schedalg]
```

Os argumentos da linha de comando para seu servidor web devem ser interpretados da seguinte forma.

- `basedir`: este é o diretório raiz a partir do qual o servidor web deve operar. O servidor deve tentar garantir que os acessos a arquivos não acessem arquivos acima deste diretório na hierarquia do sistema de arquivos. Default: diretório de trabalho atual (por exemplo, '`.`').
- `port`: o número da porta em que o servidor web deve ouvir; o servidor web básico já lida com este argumento. Default: 10000.
- `threads`: o número de threads de trabalho que devem ser criados dentro do servidor web. Deve ser um inteiro positivo. Default: 1.
- `buffers`: o número de conexões de solicitação que podem ser aceitas de uma vez. Deve ser um inteiro positivo. Observe que não é um erro ter mais ou menos threads criados do que buffers. Default: 1.
- `schedalg`: o algoritmo de agendamento a ser executado. Deve ser um de FIFO ou SFF. Default: FIFO.

Por exemplo, você poderia executar seu programa como:

```
prompt> server -d . -p 8003 -t 8 -b 16 -s SFF
```

Neste caso, seu servidor web ouvirá a porta 8003, criará 8 threads de trabalho para lidar com solicitações HTTP, alocará 16 buffers para conexões que estão atualmente em andamento (ou esperando) e usará o agendamento SFF para solicitações que chegam.

Visão Geral do Código Fonte

Recomenda-se que você entenda como o código fornecido funciona. São fornecidos os seguintes arquivos:

- `wserver.c`: Contém `main()` para o servidor web e o loop de serviço básico.
- `request.c`: Executa a maior parte do trabalho para lidar com solicitações no servidor web básico. Comece em `request_handle()` e trabalhe através da lógica a partir daí.
- `io_helper.h` e `io_helper.c`: Contêm funções invólucro para as chamadas de sistema invocadas pelo servidor web e cliente básicos. A convenção é adicionar `_or_die` a uma chamada existente para fornecer uma versão que ou tem sucesso ou sai. Por exemplo, a chamada de sistema `open()` é usada para abrir um arquivo, mas pode falhar por uma série de razões. O invólucro, `open_or_die()`, ou abre um arquivo com sucesso ou sai em caso de falha.
- `wclient.c`: Contém `main()` e as rotinas de suporte para o cliente web muito simples. Para testar seu servidor, você pode querer mudar este código para que ele possa enviar solicitações simultâneas para o seu servidor. Ao iniciar `wclient` várias vezes, você pode testar como seu servidor lida com solicitações concorrentes.
- `spin.c`: Um programa CGI simples. Basicamente, ele "gira" por uma quantidade fixa de tempo, o que pode ser útil para testar vários aspectos do seu servidor.
- `Makefile`: É fornecido um Makefile de exemplo que cria wserver, wclient e spin.cgi. Digite `make` para criar todos esses programas. Digite `make clean` para remover os arquivos de objeto e os executáveis. Digite `make server` para criar apenas o programa do servidor, etc. À medida que você cria novos arquivos, precisará adicioná-los ao Makefile.

A melhor maneira de aprender sobre o código é compilá-lo e executá-lo.

Execute o servidor básico no seu navegador web. Execute este servidor com o código de cliente fornecido. Você pode até mesmo fazer com que o código de

cliente fornecido entre em contato com qualquer outro servidor que fale HTTP. Faça pequenas alterações no código do servidor (por exemplo, faça com que ele imprima mais informações de depuração) para ver se você entende como ele funciona.

Leitura Adicional Útil

Você achará as seguintes rotinas úteis para criar e sincronizar threads: `pthread_create()`, `pthread_mutex_init()`, `pthread_mutex_lock()`, `pthread_mutex_unlock()`, `pthread_cond_init()`, `pthread_cond_wait()`, `pthread_cond_signal()`.

Avaliação

Parte 1: Implementação Multi-threaded (60 pontos)

- **Criação do Pool de Threads (15 pontos):** O servidor deve ser capaz de criar um número fixo de threads de trabalho, conforme especificado na linha de comando (`t`). A execução deve iniciar com a thread mestre criando este pool.
- **Buffer de Conexões (15 pontos):** A implementação deve incluir um buffer de tamanho fixo para armazenar os descritores de conexão, conforme o argumento de linha de comando (`b`).
- **Relação Produtor-Consumidor (15 pontos):** A sincronização entre o thread mestre (produtor) e os threads de trabalho (consumidores) deve ser correta. A mestre deve bloquear se o buffer estiver cheio, e os trabalhadores devem bloquear se o buffer estiver vazio.
- **Uso de Variáveis de Condição e Mutex (15 pontos):** A implementação deve usar **exclusivamente** `pthread_mutex_t` e `pthread_cond_t` para a sincronização. O uso de espera ocupada (busy-waiting) deve resultar em pontuação zero nesta parte.

Parte 2: Políticas de Agendamento (30 pontos)

- **Agendamento FIFO (15 pontos):** A política de agendamento padrão deve ser implementada corretamente. Quando uma thread de trabalho estiver livre, ela deve pegar a requisição mais antiga no buffer.

- **Agendamento SFF (15 pontos):** A implementação deve ser capaz de agendar a requisição para o menor arquivo. Isso exige que a thread mestre leia o descritor de rede e determine o tamanho do arquivo (usando `stat()`) antes de colocar a requisição no buffer.

Funcionalidades Adicionais e Qualidade do Código (10 pontos)

- **Tratamento de Parâmetros da Linha de Comando (5 pontos):** O programa deve suportar todos os parâmetros especificados no documento: `d`, `p`, `t`, `b` e `s`. Os valores padrão também devem ser respeitados.
- **Segurança (5 pontos):** O servidor deve rejeitar requisições que contenham `..` no nome do caminho para prevenir o acesso a arquivos fora do diretório base.

Referências

Adaptado de: https://github.com/remzi-arpacidusseau/ostep-projects/tree/master/concurrency-webserver