Universidade Federal de Roraima
Departamento de Ciência da Computação
Sistemas Operacionais – DCC403

PROJETO FINAL - Prazo de Entrega: 26/06/2023

ALUNO(A):_____NOTA:_____

ATENÇÃO: Descrever as soluções com o máximo de detalhes possível, no caso de programas, inclusive a forma como os testes foram feitos. Todos os artefatos (relatório, código fonte de programas, e outros) gerados para este trabalho devem ser adicionados em um repositório no site `github.com`, com o seguinte formato:

`Aluno1Aluno2Aluno3_FinalProject_OS_RR_2023`.

[DESCRIÇÃO - 1] ***Compressed Network Communication***

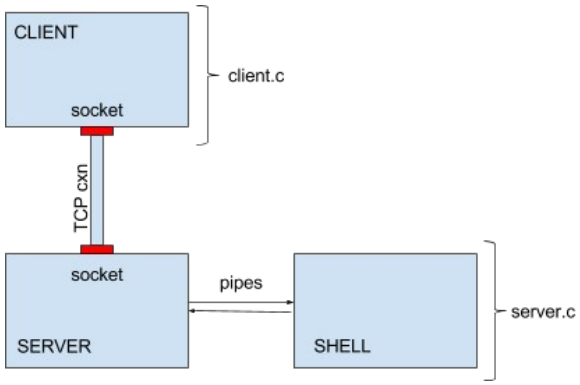
Quando uma aplicação (por exemplo, shell) é para ser usada remotamente, geralmente não é suficiente simplesmente substituir E/S de dispositivo local (por exemplo, entre o terminal do usuário e o shell) com a E/S de rede. Sessões remotas e protocolos de rede adicionam opções e comportamentos que não existiam quando o aplicativo estava sendo usado localmente. Para manipular esse processamento adicional sem fazer nenhuma alteração no aplicativo, é comum criar agentes do lado do cliente e do lado do servidor que manipulem a comunicação de rede e protejam o aplicativo das complexidades dos protocolos de acesso remoto. Esses agentes intermediários podem implementar recursos valiosos (por exemplo, criptografar o tráfego para aumentar a segurança ou compactar o tráfego para melhorar o desempenho e reduzir o custo da comunicação em escala de WAN), de forma totalmente transparente para o usuário e o aplicativo.

Neste projeto, você criará um cliente e um servidor de telnet que pode ser dividido em duas etapas principais:

- Passar a entrada e saída através de um soquete TCP; e
- Compactar a comunicação entre o cliente e o servidor.

Passar a entrada e saída através de um soquete TCP

Seu projeto geral do programa será semelhante a figura abaixo. Você terá um processo de cliente, um processo de servidor e um processo de shell. Os dois primeiros são conectados via conexão TCP, e os dois últimos são conectados via pipes.



O programa cliente

- O programa cliente abrirá uma conexão com um servidor (a porta deverá ser especificada com o parâmetro de linha de comando `--port`) em vez de enviá-lo diretamente para um shell. O cliente deve então enviar a entrada do teclado para o soquete (enquanto ecoa para o monitor), e a entrada do soquete para o monitor;
- Inclua, no cliente, uma opção `--log=filename`, que mantém um registro dos dados enviados pelo soquete;
- Adicione uma opção `--compress` ao cliente;

Para todos os testes, o cliente e o servidor estarão executando no mesmo host (localhost), mas se você quiser adicionar um parâmetro `--host=name`, poderá acessar uma sessão de shell de outros computadores.

O programa do servidor

- O programa do servidor irá se conectar com o cliente, receber os comandos do cliente e enviá-los para o shell e "servir" ao cliente as saídas desses comandos;
- O programa do servidor escutará em um soquete de rede (porta especificada com o parâmetro de linha de comando `--port` = obrigatório);
- Aceite uma conexão quando ela for feita;
- Uma vez que uma conexão é feita, crie um *fork* do processo filho, que executará um shell para processar os comandos recebidos. O processo do servidor deve se comunicar com o shell por meio de pipes;
- Redirecione o stdin/stdout/stderr do processo do shell para as extremidades apropriadas do pipe;
- Entrada recebida através do soquete de rede deve ser encaminhada através do pipe para o shell. Como o servidor cria um *fork* do shell (e conhece seu ID de processo), o processamento de ^C (transformando-o em um SIGINT no shell) deve ser feito no servidor. Da mesma forma, quando o servidor encontra um ^D, ele deve fechar o lado da gravação do pipe para o shell;
- A entrada recebida dos pipes do shell (que recebem stdout e stderr do shell) deve ser encaminhada para o soquete da rede.

A opção `--log`

Para garantir que a compactação está sendo feita corretamente, pedimos que você adicione uma opção `--log=filename` ao seu cliente. Se essa opção for especificada, todos os dados gravados ou lidos do servidor deverão ser registrados no arquivo especificado. Prefixo cada entrada de log com `SENT # bytes:` ou `RECEIVED # bytes:` conforme apropriado. (Observe o dois pontos e o espaço entre a palavra bytes e o início dos dados). Cada uma dessas linhas deve ser seguida por um caractere de nova linha (mesmo se o último caractere da string relatada for uma nova linha).

Saída de formato de log de amostra:

```
ENVIADO  35 bytes: sendingsendingsendingsendingsending
RECEBIDO 18 bytes: receivingreceiving
```

Comunicação comprimida/compactada

O objetivo da compactação é reduzir a quantidade de dados de espaço ocupados. Nos cenários de comunicação, uma versão compactada de dados usará menos largura de banda para transmitir os mesmos dados que a forma não compactada usaria. Neste projeto, você só executará compactação, sem criptografia. Você usará uma biblioteca de compactação padrão para melhorar a eficiência de suas comunicações de soquete.

- Dependendo do sistema em que você desenvolve o seu projeto, você pode precisar instalar o pacote **zlib** para obter a biblioteca de compressão e sua documentação;
- Adicione uma opção de linha de comando `--compress` ao seu cliente e servidor que, se incluído, ativará a compactação (de todo o tráfego em ambas as direções);
- Modifique o aplicativo cliente e servidor para compactar o tráfego antes de enviá-lo pela rede e descompactá-lo após recebê-lo;
- A opção `--log` deve registrar os dados de saída após a compactação e a pré-descompactação de dados recebidos; e
- Inclua seus arquivos `--log` no seu envio.

Segue abaixo alguns links:
http://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html
https://www.zlib.net/zlib_how.html

[DESCRIÇÃO - 2] Desenvolvimento de um mini terminal Linux, ou seja, um interpretador de comandos, o qual deverá aceitar a maioria dos comandos convencionais do terminal Linux, além da criação de processos e tratamento dos mesmos.

Neste trabalho você deve implementar um interpretador de comandos (chamado no Unix/Linux de *shell*). Para isso, você deverá aprender como disparar processos e como encadear sua comunicação usando *pipes*. Além disso, você deverá utilizar pipes e manipulação de processos para criar um par produtor/consumidor por envio de mensagens que trabalhará dentro da *shell* para ``alimentar'' programas que você disparar com dados, ou para receber os dados de um programa.

Seu programa deverá ser implementado em C (não serão aceitos recursos de C++, como classes da biblioteca padrão ou de outras fontes). O programa, que deverá se chamar `shellso` deverá ser iniciado sem argumentos de linha de comando, ou com apenas um. Caso seja disparado sem argumentos, ele deverá escrever um *prompt* no início de cada linha na tela (um símbolo como “>>PROMPT ” ou uma mensagem como “Sim, mestre?”) e depois ler comandos da sua entrada padrão (normalmente, o teclado). Mensagens de erro e o resultado dos programas, salvo quando redirecionados pelos comandos fornecidos, devem ser exibidos na saída padrão (usualmente, a janela do terminal). Essa forma de operação é denominada interativa. Já se um parâmetro for fornecido, ele deve ser interpretado como o nome de um arquivo de onde comandos serão lidos. Nesse caso, apenas o resultado dos comandos deve ser exibido na saída padrão, sem a exibição de *prompts* nem o nome dos comandos executados.

Em ambos os modos de operação, sua *shell* termina ao encontrar o comando “`fim`” no início de uma linha ou ao encontrar o final do fluxo de bytes de entrada (ao fim do arquivo de entrada, ou se o usuário digita `Ctrl-D` no teclado). Cada linha deve conter um comando ou mais comandos para ser(em) executado(s). No caso de haver mais de um programa a ser executado na linha, eles devem obrigatoriamente ser encadeados por *pipes* (“|”), indicando que a saída do programa à esquerda deve ser associada à entrada do programa à direita.

Um novo *prompt* só deve ser exibido (se necessário) e um novo comando só deve ser lido quando o comando da linha anterior terminar sua execução, exceto caso a linha de comando termine com um "*&*", quando o programa deve ser executado em *background*. Em qualquer caso, o interpretador deve sempre receber o valor de saída dos programas executados -- isto é, ele não deve deixar zumbis no sistema (confira as chamadas de sistema `wait()` e `waitpid()` para esse fim). Para o caso dos programas executados em *background*, você deve se informar sobre o tratamento de sinais, em particular o sinal SIGCHLD, para tratar o término daqueles programas.

O seu interpretador não aceitará os comandos de redirecionamento de entrada e saída normalmente utilizados, "<" e ">". Ao invés disso, ele aceitará os símbolos "=>" e "<=", que indicarão entrada e saída por *pipes* para processos produtores/consumidores.

Com base nessa descrição, são comandos válidos (supondo que o *prompt* seja "Qual o seu comando?":

```
Qual o seu comando? ls -l
Qual o seu comando? ls -laR => arquivo
Qual o seu comando?
Qual o seu comando? wc -l <= arquivo &
Qual o seu comando? cat -n <= arquivo => arquivonumerado
Qual o seu comando? cat -n <= arquivo | wc -l => numerodelinhas
Qual o seu comando? cat -n <= arquivo | wc -l => numerodelinhas &
Qual o seu comando? fim
```

Com base na descrição apresentada, para o referido projeto deve ser apresentado os seguintes itens:

1. Descreva as principais funções do terminal;
2. Descrever as funções dos seguintes componentes do terminal: parser; executor; subsistemas do terminal;
3. Descrever a implementação para a criação de processos;
   - No seu interpretador crie pelo menos um novo processo para cada novo comando;
   - Para ler linhas da entrada, você pode querer olhar a função `fgets()`;
4. Descrever a implementação de pipe e redirecionamento no terminal;
5. Descrever a implementação de wildcards no terminal;
6. Codificação de caracteres no terminal; e
7. Lista de comandos disponível;

[DESCRIÇÃO – 3] Projete e implemente o seu próprio File System – Sistema de Arquivos

Com base na descrição apresentada, para o referido projeto deve ser apresentado os seguintes itens:

1) Leia o artigo **Creating Linux virtual filesystems** em https://lwn.net/Articles/57369/ e apresente uma descrição dos itens necessários para o desenvolvimento de um sistema de arquivos;
2) Teste os seguintes exemplos de sistemas de arquivos
   I. http://www.ic.unicamp.br/~islene/2s2013-mo806/vfs/gog-islenefs/
   II. http://www.ic.unicamp.br/~islene/2s2013-mo806/vfs/hellofs/

3) Projete e implemente o seu próprio *file system* (de brinquedo)
   I. Funcionalidades básicas: operações de leitura e escrita;
   II. Arquivos, estes não precisam ser armazenados em disco;
   III. Árvore de diretórios com suas funcionalidades.

[DESCRIÇÃO – 4] Simulador de memória virtual e física

Será implementado um simulador de memória virtual e física. Seu programa deverá ser implementado em C (não serão aceitos recursos de C++, como classes da biblioteca padrão ou de outras fontes).

O simulador receberá **como entrada um arquivo** que conterá a sequência de endereços de memória acessados por um programa real (na verdade, apenas uma parte da sequência total de acessos de um programa). Esses endereços estarão escritos como **números hexadecimais**, seguidos por uma letra R ou W, para indicar se o acesso foi de leitura ou escrita. Ao iniciar o programa, será definido o tamanho da memória (em quadros) para aquele programa e qual o algoritmo de substituição de páginas a ser utilizado. O programa deve, então, processar cada acesso à memória para atualizar os bits de controle de cada quadro, detectar falhas de páginas (*page faults*) e simular o processo de carga e substituição de páginas. Durante todo esse processo, estatísticas devem ser coletadas, para gerar um relatório curto ao final da execução.

O programa, que deverá ser iniciado com quatro argumentos, exemplo, `progvirtual <polisub> <arquivo.log> <sizePg> <sizeFis>.` Esses argumentos representam, pela ordem:

1. O `<polisub>` algoritmo de substituição a ser usado (`lru`, `fifo` ou `random`);
2. O `<arquivo.log>` arquivo contendo a sequência de endereços de memória acessados;
3. O `<sizePg>` é o tamanho de cada página/quadro de memória, em kilobytes -- faixa de valores razoáveis: de 2 a 64;
4. O `<sizeFis>` é o tamanho total da memória física disponível para o processo, também em kilobytes - faixa de valores razoáveis: de 128 a 16384 (16 MB).

Formato da saída do simulador. Ao final da simulação, quando a sequência de acessos à memória terminar, o programa deve gerar um pequeno relatório, contendo:

- a configuração utilizada (definida pelos quatro parâmetros);
- o número total de acessos à memória contidos no arquivo;
- o número de page faults (páginas lidas); e
- o número de páginas "sujas" que tiveram que ser escritas de volta no disco (lembrando-se que páginas sujas que existam no final da execução não precisam ser escritas).

Um exemplo de saída poderia ser da forma (valores completamente fictícios):

```
prompt> simvirtual lru arquivo.log 4 128
  Executando o simulador...
  Arquivo de entrada: arquivo.log
  Tamanho da memoria: 128 KB
  Tamanho das páginas: 4 KB
  Tecnica de reposicao: lru
```

```
Paginas lidas: 520
Paginas escritas: 352
```

Mais detalhes podem ser consultados em:
https://homepages.dcc.ufmg.br/~dorgival/cursos/so/tp3.html

[DESCRIÇÃO – 5] Virtualização de Sistemas Operacionais – VB e Containers

Neste projeto o seu objetivo é criar uma rede de computadores virtuais que se comuniquem usando Máquinas Virtuais (exemplo, Virtual Box) e com Containers (exemplo, Docker). Com base na descrição apresentada, para o referido projeto deve ser apresentado os seguintes itens:

- Processo para instalação de um software para a criação de máquinas virtuais e containers;
- Criar um cluster com máquinas virtuais;
- Criar um cluster de containers;
- Apresentar uma avaliação de vantagens e desvantagens entre máquinas virtuais e containers;
- Desenvolva um sistema web escalável usando containers, sendo que cada aspecto do sistema deve conter containers para objetivos específicos, exemplo, um container para o banco de dados, servidor web e outros que se façam necessário. Adicionalmente, aplique um software de sua escolha para a gerência dos containers; e
- Apresente 3 sistemas operacionais com o foco em virtualização de sistemas operacionais.

[DESCRIÇÃO – 6] Scheduling Policy to the Linux Kernel

O escalonador de processos é o núcleo do sistema operacional. É responsável por escolher um novo processo para ser executado sempre que a CPU estiver disponível. Você estudará partes do escalonador do Linux em profundidade e entenderá como ele decide qual processo executar. Você modificará o escalonador do Linux para implementar uma nova classe de escalonador, chamada de escalonador em background. Você avaliará o desempenho de sua nova classe de escalonador.

A nova política de agendamento chamada SCHED_BACKGROUND, projetada para suportar processos que só precisam ser executados quando o sistema não tem mais nada a fazer. Essa política de agendamento "em segundo plano" só executa processos quando não há processos nas classes SCHED_OTHER, SCHED_RR ou SCHED_FIFO a serem executados. Quando houver mais de um processo SCHED_BACKGROUND pronto para ser executado, eles devem competir pela CPU, assim como os processos SCHED_OTHER.

Neste projeto você irá testar políticas de escalonamento e implementar uma nova no kernel do linux. Com base na descrição apresentada, para o referido projeto deve ser apresentado os seguintes itens:

- Descrever o funcionamento de um escalonamento;
- Criar um tutorial com exemplos de códigos para a criação de uma política de escalonamento no linux;
- Apresente 4 políticas de escalonamento suportadas pelo kernel do linux; e

- Depois de implementar sua política de agendamento e depurá-la com cuidado, você avaliará seu desempenho executando combinações de processos intensivos de CPU em diferentes políticas de agendamento e medindo o tempo necessário;
  - Você pode usar o `gettimeofday()` para medir o tempo "wallclock" e `getrusage()` para registrar o tempo do usuário e do sistema. Sua análise deve incluir wallclock, sistema e tempo do usuário em milissegundos.

Segue abaixo alguns links:
https://helix979.github.io/jkoo/post/os-scheduler/