Recuperação de arquivos apagados em sistemas de arquivos ext2/3/4

Objetivo

O objetivo deste projeto é estudar a estrutura interna dos sistemas de arquivos da família ext (ext2, ext3 e ext4) e desenvolver uma ferramenta capaz de recuperar arquivos apagados, total ou parcialmente, de partições formatadas com esses sistemas de arquivos.

Descrição

Ao apagar um arquivo em sistemas de arquivos do tipo ext2/3/4, o conteúdo do arquivo geralmente não é imediatamente sobrescrito, o que possibilita a sua recuperação parcial ou total. No entanto, os metadados do arquivo (como entradas no diretório e inodes) são marcados como livres, dificultando o processo.

O projeto consiste em analisar discos ou imagens de disco contendo partições ext2, ext3 ou ext4 e identificar arquivos recentemente apagados que ainda não foram sobrescritos. A ferramenta deverá ser capaz de:

- Ler a partição à procura de inodes não referenciados.
- Identificar arquivos apagados com base em características do sistema de arquivos.
- Restaurar o conteúdo do arquivo (ou parte dele) para um local seguro.
- Registrar o sucesso ou falha da operação de recuperação.

Requisitos

- O projeto pode ser desenvolvido em C.
- O aluno deve utilizar imagens de disco reais ou criadas com ferramentas como `dd` e `mkfs.ext4`.
- A ferramenta deve funcionar, pelo menos, para o sistema de arquivos ext2, ext3 ou ext4.
- O uso de bibliotecas como `libext2fs` (do pacote `e2fsprogs`) **não** é permitido.
- Documentar o funcionamento interno do sistema de arquivos e justificar as abordagens adotadas para a recuperação.

Implementação de um escalonador em modo usuário

Objetivo

O objetivo deste projeto é implementar um escalonador de tarefas em modo usuário, com foco na troca de contexto entre tarefas (threads) de forma manual, sem depender de bibliotecas além da biblioteca padrão. O projeto introduz os conceitos de escalonamento cooperativo e, como extensão, escalonamento preemptivo com uso de sinais ou temporizadores.

Descrição

Sistemas operacionais modernos são responsáveis por gerenciar múltiplas tarefas concorrentes. Este projeto propõe simular esse comportamento em modo usuário, criando um pequeno ambiente multitarefa no qual várias funções (representando tarefas) compartilham a CPU de maneira intercalada.

Devem ser implementadas:

- Uma infraestrutura de criação e gerenciamento de tarefas (threads) em espaço de usuário.
- Um mecanismo de troca de contexto (salvando e restaurando registradores e pilhas).
- Um escalonador cooperativo, no qual as tarefas explicitamente cedem a CPU.
- (Extra) Um escalonador preemptivo, usando timers e sinais para forçar a troca de tarefas após um tempo fixo.

Requisitos

- A implementação deve ser feita em C.
- Não utilizar bibliotecas externas como `pthreads`, `boost`, etc.
- Utilizar apenas funções do sistema operacional como `malloc`, `sigaction`, `setjmp/longjmp` (ou `sigsetjmp/siglongjmp`), `setitimer`, `mmap`, entre outras permitidas.
- As tarefas devem ser executadas de forma concorrente, em pseudo-multitarefa.

Detecção de deadlocks e violação da ordem de aquisição de travas

Objetivo

O objetivo deste projeto é estender uma implementação didática de mutexes feita em sala, adicionando mecanismos de detecção de deadlocks e/ou verificação da ordem de aquisição de travas, inspirando-se em abordagens reais como o `lockdep` do Linux.

Descrição

Mutexes garantem exclusão mútua em regiões críticas, mas quando múltiplas threads acessam múltiplos mutexes simultaneamente, podem surgir situações de deadlock — geralmente causadas por espera circular entre threads. Este projeto propõe duas abordagens complementares:

1. Detecção de deadlocks via grafo de espera**:**
    - Monitorar quais threads possuem e quais esperam por cada mutex.
    - Construir dinamicamente um grafo de espera por recursos.
    - Detectar ciclos no grafo, sinalizando um deadlock real.

2. Detecção de violação da ordem de aquisição de travas (estilo lockdep):
    - Manter o histórico da ordem em que mutexes são adquiridos por cada thread.
    - Detectar quando uma thread tenta adquirir mutexes em uma ordem incompatível com a previamente observada, antes que o deadlock ocorra.
    - Exibir alertas sobre inversões de ordem (e.g., thread A adquire A → B, e thread B tenta B → A).

Requisitos

- Utilizar a implementação de mutex em C discutida em sala como base.
- Estender o mutex com estruturas auxiliares para rastrear:
    1. Dono atual.
    2. Fila de espera.
    3. Ordem de aquisição por thread.

- Implementar pelo menos uma das duas abordagens a seguir (vale pontos extras implementar ambas):
    1. Grafo de espera com detecção de ciclos (usando DFS ou algoritmo equivalente).
    2. Verificação de ordem de aquisição com comparação entre pares de mutexes adquiridos por diferentes threads.

Leitor e extrator de arquivos do sistema de arquivos ISO 9660

Objetivo

Desenvolver um programa em C capaz de ler e interpretar imagens ISO 9660 (padrão de sistemas de arquivos para CDs/DVDs) e realizar listagens e extração de arquivos e diretórios, sem o uso de bibliotecas externas específicas para ISO.

Descrição

O sistema de arquivos ISO 9660 define a estrutura de armazenamento de arquivos em mídias ópticas. Esse projeto propõe a leitura direta de imagens `.iso`, realizando o parse manual da estrutura interna da imagem.

O programa deverá:

- Abrir e analisar uma imagem ISO 9660.
- Interpretar o Primary Volume Descriptor e localizar o diretório raiz.
- Listar o conteúdo de um diretório especificado por caminho (ex: `/docs/manuals`).
- Permitir a extração dos arquivos listados para o sistema de arquivos local.
- Preservar a hierarquia de diretórios durante a extração.

Requisitos

- A implementação deve ser feita em C.
- O código deve acessar diretamente os bytes da imagem `.iso`, usando funções como `fopen`, `fread`, `fseek`.
- É proibido usar bibliotecas prontas para ISO 9660, como `libiso9660`, `libcdio`, `fuseiso`, etc.
- O programa deve aceitar pelo menos os seguintes modos de operação por linha de comando:

```
./iso_reader lista imagem.iso /caminho/desejado
./iso_reader extrai imagem.iso /caminho/desejado
```

Funcionalidades Adicionais (opcional, vale pontos extras)

- Suporte parcial a **extensões Rock Ridge** (nomes longos, permissões Unix).
- Suporte a **Joliet** (Unicode/UTF-16 nos nomes).
- Listagem recursiva de diretórios.
- Filtragem de arquivos por extensão.

Implementação de semáforos e variáveis de condição com `futex`

Objetivo

Implementar primitivas clássicas de sincronização — semáforos e variáveis de condição — utilizando diretamente a chamada de sistema `futex` do Linux. O objetivo é compreender como primitivas de sincronização podem ser construídas com base em operações atômicas e suporte do kernel, explorando o modelo userspace fast path + kernel slow path.

Descrição

Sistemas operacionais modernos fornecem mecanismos eficientes de sincronização que evitam transições ao kernel sempre que possível. O `futex` é uma primitiva de baixo nível do Linux que permite a construção dessas estruturas de maneira eficiente: threads aguardam por uma variável inteira no espaço de usuário e são colocadas em espera apenas quando necessário.

Neste projeto, deve-se:

- Implementar um semáforo, com as operações clássicas `incrementar` e `decrementar`, diretamente com `futex`.
- Implementar uma variável de condição, com as operações `esperar`, `notificar_um` e `notificar_todos`, utilizando `futex` e o mutex implementado em sala.
- Utilizar instruções atômicas de comparação e troca para garantir consistência no espaço de usuário.

Requisitos

- A implementação deve ser feita em C.
- Não é permitido o uso de `pthread_mutex_t`, `pthread_cond_t`, ou qualquer API pronta de sincronização.
- A chamada de sistema `futex` deve ser invocada diretamente via `syscall(...)` ou por wrapper próprio.
- Espera ocupada deve ser evitada: o uso de `futex` deve garantir que o kernel bloqueia a thread quando necessário.
- É necessário implementar um exemplo de uso para semáforo e variável de condição.

Implementação de Tabela de Descritores

Objetivo

Implementar, em C, uma tabela de descritores de arquivos semelhante àquela usada em sistemas Unix/Linux, incluindo:

- alocação do menor índice disponível (como exige POSIX),
- expansão dinâmica com limite superior,
- duplicação e redirecionamento de descritores (`dup`, `dup2`),
- contagem de referência no recurso (não na tabela),
- saturação da contagem, onde o recurso se torna permanente,
- suporte a concorrência segura entre múltiplas threads.

Descrição Geral

Em sistemas Unix, descritores de arquivos são inteiros pequenos que referenciam recursos (arquivos, pipes, etc.) compartilhados. Esses recursos possuem contagem de referência: enquanto houver descritores apontando para eles, não são destruídos.

Neste projeto, deve-se implementar uma estrutura de tabela de descritores de arquivos com as seguintes propriedades.

Requisitos Funcionais

1. **Tabela de descritores:**
    - Deve ser inicializada com um tamanho inicial e um tamanho máximo
    - Cresce sob demanda, até o máximo permitido.
    - Cada entrada contém um ponteiro para um recurso

2. **Alocação do menor índice livre:**
    - Deve retornar o menor descritor disponível.

3. **Fechamento de descritor:**
    - Libera o descritor e tenta decrementar a referência do recurso.
    - Se a referência atingir zero, o recurso é destruído.

4. **Contagem de referência nos arquivos:**
    - Cada arquivo tem um contador de referências.
    - A contagem é incrementada quando um novo descritor aponta para ele.
    - Quando a contagem atinge um valor máximo, ela satura:
        - não pode mais ser incrementada;
        - não pode mais ser decrementada;
        - o recurso se torna "permanente" e nunca será destruído.

5. **Concorrência:**
    - A implementação deve funcionar corretamente em execução paralela.