ATIVIDADE AVALIATIVA – RA3

Esta atividade avaliativa pode ser realizada em grupos de até 4 alunos. **Grupos com mais de 4 alunos irão provocar a anulação da atividade. Esta atividade tem peso no cálculo da média conforme explicitado no Plano de Ensino.** Você deve ler todo documento antes de começar e considerar o seguinte código de ética: *você poderá discutir todas as questões com seus colegas de classe, professores e amigos. Poderá também consultar os livros de referência da disciplina, livros na biblioteca virtual ou não, e qualquer ferramenta de busca ou inteligência artificial disponível na internet, de forma geral e abrangente nos idiomas que desejar. Contudo, o trabalho é seu e deverá ser realizado por você.* ***Cópias, ou realização por sistemas de inteligência artificial ensejarão a anulação do trabalho****.*

Lembre-se: os computadores dos laboratórios possuem restrições para a instalação de aplicativos e bibliotecas. Se não for usar o seu próprio notebook para a prova de autoria, deve testar seu projeto no laboratório antes da prova de autoria.

Proposta

Pesquisar e praticar. Pesquisar os conteúdos que irão complementar o material apresentado em sala, ou nos livros sugeridos na ementa, e praticar estes mesmos conceitos. Esta é uma oportunidade para aprimorar sua formação e conseguir uma vantagem competitiva para se destacar no mercado. Uma avaliação com oportunidade de crescimento acadêmico e profissional.

1. CONTEXTO E MOTIVAÇÃO

Containers revolucionaram a forma como desenvolvemos e implantamos software, mas sua eficiência depende fundamentalmente de mecanismos do *kernel Linux*: **namespaces** (para isolamento) e **control groups** (para limitação de recursos). Compreender profundamente como esses mecanismos funcionam é essencial para qualquer engenheiro que trabalhe com infraestrutura moderna.

Esta atividade propõe o desenvolvimento de um **sistema de *profiling* e análise** que permita monitorar, limitar e analisar o uso de recursos por processos e containers, explorando as primitivas do kernel Linux que tornam a containerização possível.

2. OBJETIVOS DE APRENDIZAGEM

Ao final desta atividade, espera-se que os alunos sejam capazes de:

1. **Compreender profundamente** as interfaces ***/proc, /sys/fs/cgroup*** e as ***syscalls*** relacionadas a ***namespaces***
2. **Implementar** coletores de métricas de sistema em linguagem C/C++23
3. **Analisar** o *overhead* de diferentes mecanismos de isolamento e limitação

4. **Avaliar** *trade-offs* entre isolamento, performance e complexidade
5. **Correlacionar** métricas de diferentes camadas (processo, ***namespace***, ***cgroup***)
6. **Documentar** experimentos de forma científica e reproduzível
7. **Trabalhar colaborativamente** em projeto de engenharia de sistemas

3. DESCRIÇÃO DA ATIVIDADE

Vocês desenvolverão um **sistema de monitoramento e análise de recursos** composto por três componentes principais:

Componente 1: Resource Profiler

Ferramenta que coleta e reporta métricas detalhadas de processos:

- CPU (user time, system time, context switches, threads)
- Memória (RSS, VSZ, page faults, swap)
- I/O (bytes read/write, syscalls, disk operations)
- Rede (bytes rx/tx, packets, connections)

Componente 2: Namespace Analyzer

Ferramenta que analisa e reporta isolamento via namespaces:

- Identificar todos os namespaces ativos no sistema
- Mapear processos por namespace
- Comparar namespaces entre processos
- Medir overhead de criação de namespaces
- Gerar relatórios de isolamento

Componente 3: Control Group Manager

Ferramenta que analisa e manipula control groups:

- Ler métricas de todos os controladores (CPU, Memory, BlkIO, PIDs)
- Criar cgroups experimentais
- Aplicar limites de recursos
- Medir precisão de throttling
- Gerar relatórios de utilização vs limites

4. REQUISITOS TÉCNICOS

4.1 Requisitos Obrigatórios

Linguagem de Implementação:

- C ou C++ 2023 (obrigatório para os coletores principais)
- Python ou Shell Script (permitido para scripts auxiliares e visualização)

Funcionalidades Mínimas:

1. **Resource Profiler:**
   - Monitorar processo por PID com intervalo configurável
   - Coletar pelo menos: CPU, Memória, I/O
   - Calcular CPU% e taxas de I/O
   - Exportar dados em CSV ou JSON
   - Tratar erros (processo inexistente, permissões)

2. **Namespace Analyzer:**
   - Listar todos os namespaces de um processo
   - Encontrar processos em um namespace específico
   - Comparar namespaces entre dois processos
   - Gerar relatório de namespaces do sistema

3. **Control Group Manager:**
   - Ler métricas de CPU, Memory e BlkIO cgroups
   - Criar cgroup experimental
   - Mover processo para cgroup
   - Aplicar limites de CPU e Memória
   - Gerar relatório de utilização

4. **Qualidade de Código:**
   - Compilar sem *warnings* (-Wall -Wextra)
   - Código comentado e bem estruturado
   - Makefile funcional
   - README com instruções de compilação e uso

4.2 Funcionalidades Opcionais (Pontos Extras)

- Sem *memory leaks* (validar com *valgrind*)
- Interface *ncurses* para visualização em tempo real
- Suporte a *cgroup* v2 (*unified hierarchy*)
- **Dashboard web com visualização de métricas**
- Detecção automática de anomalias
- Suporte a monitoramento de múltiplos processos simultaneamente
- Comparação com ferramentas existentes (*docker stats*, *systemd-cgtop*)

5. ORGANIZAÇÃO DO TRABALHO

5.1 Divisão Sugerida para Grupos de 4 Alunos

Aluno 1: Resource Profiler + Integração

- Implementar coleta de CPU e Memory
- Implementar cálculos de percentuais e taxas

- Integrar os três componentes
- Criar Makefile geral

Aluno 2: Resource Profiler + Testes
- Implementar coleta de I/O e Network
- Criar programas de teste (CPU, Memory, I/O intensive)
- Validar precisão das medições
- Documentar metodologia de testes

Aluno 3: Namespace Analyzer + Experimentos
- Implementar análise de namespaces
- Criar experimentos com diferentes tipos de namespaces
- Medir overhead de isolamento
- Documentar resultados experimentais

Aluno 4: Control Group Manager + Análise
- Implementar coleta de métricas de *cgroups*
- Implementar criação e configuração de *cgroups*
- Conduzir experimentos de *throttling*
- Gerar relatórios e visualizações

6. EXPERIMENTOS OBRIGATÓRIOS

Cada grupo deve realizar e documentar os seguintes experimentos:

Experimento 1: Overhead de Monitoramento

Objetivo: Medir o impacto do próprio *profiler* no sistema

Procedimento:
1. Executar *workload* de referência sem monitoramento
2. Executar mesmo *workload* com monitoramento em diferentes intervalos
3. Medir diferenças em CPU *usage* e execution time

Métricas a reportar:
- Tempo de execução com e sem profiler
- CPU overhead (%)
- Latência de sampling

Experimento 2: Isolamento via Namespaces

Objetivo: Validar efetividade do isolamento

Procedimento:
1. Criar processo com diferentes combinações de namespaces
2. Verificar visibilidade de recursos (PIDs, rede, filesystems)

3. Medir tempo de criação de cada tipo de namespace

Métricas a reportar:

- Tabela de isolamento efetivo por tipo de namespace
- Overhead de criação (µs)
- Número de processos por namespace no sistema

Experimento 3: Throttling de CPU

Objetivo: Avaliar precisão de limitação de CPU via cgroups

Procedimento:

1. Executar processo *CPU-intensive* sem limite
2. Aplicar limites de 0.25, 0.5, 1.0 e 2.0 cores
3. Medir CPU *usage* real em cada configuração

Métricas a reportar:

- CPU% medido vs limite configurado
- Desvio percentual
- *Throughput* (iterações/segundo) em cada configuração

Experimento 4: Limitação de Memória

Objetivo: Testar comportamento ao atingir limite de memória

Procedimento:

1. Criar *cgroup* com limite de 100MB
2. Tentar alocar memória incrementalmente
3. Observar comportamento (*OOM killer*, falhas de alocação)

Métricas a reportar:

- Quantidade máxima alocada
- Número de falhas (*memory.failcnt*)
- Comportamento do sistema ao atingir limite

Experimento 5: Limitação de I/O

Objetivo: Avaliar precisão de limitação de I/O

Procedimento:

1. Executar *workload* I/O-intensive sem limite
2. Aplicar limites de *read/write* BPS
3. Medir *throughput* real

Métricas a reportar:

- *Throughput* medido vs limite configurado
- Latência de I/O
- Impacto no tempo total de execução

7. ENTREGAS

7.1 Código-Fonte (40%)

Você deverá seguir as mesmas normas de entrega de código fonte que usamos nos trabalhos anteriores, incluindo a gestão do git/github, *commits e pull-requests*. Considere também que a divisão de tarefas propostas atende o equilíbrio de trabalho entre os integrantes do grupo.

O não cumprimento destas regras implica em deméritos severos na sua nota provisória.

Estrutura obrigatória:

```
resource-monitor/
├─── README.md
├--- Makefile
├--- docs/
|    └--- ARCHITECTURE.md
├--- include/
|    ├--- monitor.h
|    ├--- namespace.h
|    └--- cgroup.h
├--- src/
|    ├--- cpu_monitor.c
|    ├--- memory_monitor.c
|    ├--- io_monitor.c
|    ├--- namespace_analyzer.c
|    ├--- cgroup_manager.c
|    └--- main.c
├--- tests/
|    ├--- test_cpu.c
|    ├--- test_memory.c
|    └--- test_io.c
└--- scripts/
     ├--- visualize.py
     └--- compare_tools.sh
```

README.md deve conter:

- Descrição do projeto
- Requisitos e dependências
- Instruções de compilação
- Instruções de uso com exemplos
- Autores e sua contribuição

8. CRITÉRIOS DE AVALIAÇÃO

8.1 Código-Fonte (100 pontos)

Critério	Pontos	Descrição
Funcionalidade	45	Todos os requisitos obrigatórios implementados
Correção	25	Código funciona corretamente, sem bugs críticos
Qualidade	10	Código limpo, bem estruturado, seguindo boas práticas
Eficiência	10	Uso apropriado de recursos, algoritmos adequados
Robustez	10	Tratamento de erros, validação de entrada

Deméritos:

- Não compila: o trabalho será zerado
- Warnings de compilação: -10 pontos
- Algum erro não foi tratado: - 10 pontos

Bônus:

- Funcionalidades opcionais implementadas (+5 cada)
- Código excepcionalmente bem documentado (+10)
- Testes automatizados (+10)

8.2 Prova de Autoria

A Prova de Autoria será realizada conforme as regras definidas no Plano de Ensino. Caso o aluno sorteado do grupo não seja capaz de provar a autoria do trabalho, **a nota anteriormente atribuída ao grupo será reduzida em 35%**.

Atenção: todos os alunos do grupo devem ser capazes de explicar qualquer parte do trabalho. Isso significa, por exemplo, que o Aluno 1 (***Resource Profiler***) deve ser capaz de explicar a lógica do ***Control Group Manager*** do Aluno 4, e o Aluno 4 (***Control Group Manager***) deve entender como a ***Análise de Namespaces*** foi implementada pelo Aluno 3.

8.5 Deméritos:

- Suspeita de Plágio. A suspeita de plágio será determinada por semelhança pura e simples. Uma única função exatamente igual e dois ou mais trabalhos ou a semelhança de código com nomes de funções e variáveis modificados implicam na nota zero.
- Não rodou na prova de autoria: o trabalho será zerado
- Histórico de *Commits* Desbalanceado (Ex: mais de 60% feito por 1 ou 2 alunos): Adicionar penalidade de −30 (Exceto para Trabalho Individual)
- Ausência de *Pull Requests* ou Uso Incorreto do Fluxo de Trabalho (*Commit* direto na main): Adicionar penalidade de −30 pontos (Exceto para Trabalho Individual)

- Documentação Individual Incompleta no README.md (Caso o README.md não reflita a contribuição de todos os alunos: Adicionar penalidade de −10 pontos por aluno (Exceto para Trabalho Individual)
- **A nota final não será inferior a zero.**

8.4 Nota Final

$$\boldsymbol{Nota} = \big((\boldsymbol{Código}) - (\boldsymbol{Deméritos})\big) \times \boldsymbol{Prova\ de\ Autoria}\ (\boldsymbol{ou}\ \mathbf{1}\ \boldsymbol{ou}\ \mathbf{0,65})$$

9. RECURSOS E REFERÊNCIAS

9.1 Documentação Oficial

Kernel Linux:

- /usr/src/linux/Documentation/filesystems/proc.txt
- /usr/src/linux/Documentation/cgroup-v1/
- /usr/src/linux/Documentation/cgroup-v2.txt
- Man pages: man 7 namespaces, man 7 cgroups

Online:

- Kernel.org Documentation
- LWN.net - Namespaces in operation
- Red Hat - Resource Management Guide

9.2 Livros Recomendados

- **"Understanding the Linux Kernel"** - Daniel P. Bovet, Marco Cesati: capítulos sobre Process Management e Memory Management
- **"Linux System Programming"** - Robert Love: capítulos sobre Process Management e Resource Limits
- **"The Linux Programming Interface"** - Michael Kerrisk: capítulos 28 (Process Creation), 35 (Process Priorities)
- **"Container Security"** - Liz Rice: capítulos sobre Namespaces e Cgroups

9.3 Artigos Científicos

- Soltesz et al. (2007). **Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors**
- Xavier et al. (2013). **Performance Evaluation of Container-based Virtualization for High Performance Computing Environments**

- Felter et al. (2015). **An Updated Performance Comparison of Virtual Machines and Linux Containers**

9.4 Ferramentas Úteis

Análise e Debug:

- *strace - trace system calls*
- *perf - performance analysis*
- *valgrind - memory debugging*
- *gdb - debugging*

Monitoramento:

- *htop - process monitoring*
- *iotop - I/O monitoring*
- *systemd-cgtop - cgroup monitoring*

Visualização:

- *gnuplot - plotting*
- *matplotlib (Python) - graphing*
- *graphviz - diagrams*

10. PERGUNTAS FREQUENTES

P: Precisa funcionar em qualquer distribuição Linux? R: Não. Documentem a distribuição e kernel usados. Foco em Ubuntu 24.04+ mas o grupo pode escolher a distribuição que desejar.

P: Precisa ter interface gráfica? R: Não é obrigatório. *Command-line* é suficiente. Interface gráfica (com *ncurses* ou não) é opcional para pontos extras.

P: E se não tiver acesso root? R: Algumas funcionalidades (criar *cgroups*) precisam de root. Usem VM ou container com privilégios.

P: Quantas linhas de código são esperadas? R: A qualidade é mais importante que a quantidade. Estimativa: 1500-3000 linhas de C ou C++ 23, bem documentadas.

P: Posso usar bibliotecas externas? R: Apenas libc e bibliotecas padrão do sistema. Bibliotecas externas não podem ser usadas. Exceções serão tratadas presencialmente.

P: E se o código não funcionar na apresentação? R: A nota será zerada.

P: Podemos fazer em Python? R: Componentes principais devem ser em C, ou C++ 23. Contudo, você pode usar o Python para scripts auxiliares e visualização.