**UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE**
**INSTITUTO METRÓPOLE DIGITAL**
**BACHARELADO EM TECNOLOGIA DA INFORMAÇÃO**

Emanoel Dantas Pereira
Matheus Rangel de Melo

# RELATÓRIO DO TRABALHO 1.1 DE PROJETO DE SISTEMAS OPERACIONAIS (DIM0615)

Natal-RN, 20 de agosto de 2018

# 1. INTRODUÇÃO

O trabalho 1.1 da disciplina de Projeto de Sistemas Operacionais (código DIM0615), ministrada pelo professor doutor Edgard de faria Corrêa no primeiro semestre do ano 2018, tratava-se da implementação de alguns programas. Tais programas deveriam prevenir o travamento do sistema operacional diante da execução de processos fork bomb (processos que geram processos filhos infinitamente), imprimir o total de processos do sistema divididos ou não por usuários, gerar a árvore de processos dado o PID de um processo e fornecer alguma opção para salvar esta árvore em um arquivo. Assim, seguindo a especificação disponibilizada na turma virtual da disciplina, tais programas foram implementados na linguagem c++ e veremos adiante a solução encontrada para cada um destes. É importante lembrar que os programas foram compilados e executados no sistema operacional Ubuntu.

# 2. DESENVOLVIMENTO

Segue abaixo a descrição da implementação de cada programa do trabalho. O código dos programas encontra-se em https://github.com/Matheus-Rangel/PSO.

## 2.1. PREVENÇÃO DE FORK BOMB

### 2.1.1. SOLUÇÃO ENCONTRADA

Limitamos o número máximo de processos que os usuários comuns podem criar, primeiro o programa verifica quais usuários existentes no sistema e para limitar foi utilizada a função prlimit em todos os processos desses usuários.

### 2.1.2. COMO COMPILAR E EXECUTAR

Para compilar o programa, basta digitar no terminal, na pasta onde está o arquivo com o código, o comando g++ -std=c++11 limitador.cpp -o limitador. Para executar, digite ./limitador <num_maximo_de_processos> no mesmo diretório que digitou o comando para compilação.

### 2.1.3. RESULTADO

Todos os usuários comuns que estão com algum programa em execução ficarão com o número máximo de processos limitados,

impedindo assim o travamento do sistema quando algum deles tentar rodar um fork bomb.

#### 2.1.4. IMPLEMENTAÇÕES ADICIONAIS

Exibir um alerta para quando algum usuário atingir o limite de processos, mostrando os programas com o maior número de subprocessos e com a opção de excluí-los.

### 2.2. TOTAL DE PROCESSOS

#### 2.2.1. SOLUÇÃO ENCONTRADA

Para este programa, foi utilizada a função system(char*) da biblioteca stdlib.h. Ela recebe como parâmetro um char* com comandos que são executados no terminal. Assim, usamos a função sempre que precisamos executar algum desses comandos.

Primeiro foi usado o comando “cat /etc/passwd | cut -d: -f1 > usuario.txt” que salva todos os usuários no arquivo usuario.txt. Depois esses usuários são lidos deste arquivo e para cada um deles é executado o comando “ps -f -u nome_do_usuario | wc -l” que exibe o total de processos do usuário nome_do_usuario. Por fim, é executado o comando “ps aux | wc -l” que exibe o total de processos do sistema.

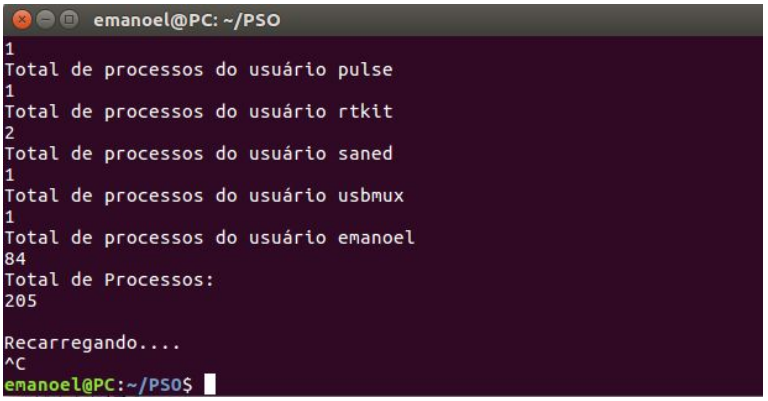
Para imprimir periodicamente a saída gerada pelo programa, foi colocado dentro de um loop infinito o código descrito acima e a função sleep(int) da biblioteca unistd.h com parâmetro 5. Assim, a cada 5 segundos o programa calcula os totais de processos novamente.

#### 2.2.2. COMO COMPILAR E EXECUTAR

Para compilar o programa, basta digitar no terminal, na pasta onde está o arquivo com o código, o comando g++ total_processos.cpp -o total_processos. Para executar, digite ./total_processos no mesmo diretório que digitou o comando para compilação.

#### 2.2.3. RESULTADO

A figura abaixo mostra a saída gerada pelo programa no terminal, vale lembrar que o total de processos do usuário root é exibido no topo da saída e o total geral de processos no final.




### 2.2.4. IMPLEMENTAÇÕES ADICIONAIS

Uma melhoria que poderia ser feita no código para deixar a saída mais resumida era não exibir na lista usuários que tivessem apenas 1 processo.

## 2.3. ÁRVORE DE PROCESSOS

### 2.3.1. SOLUÇÃO ENCONTRADA

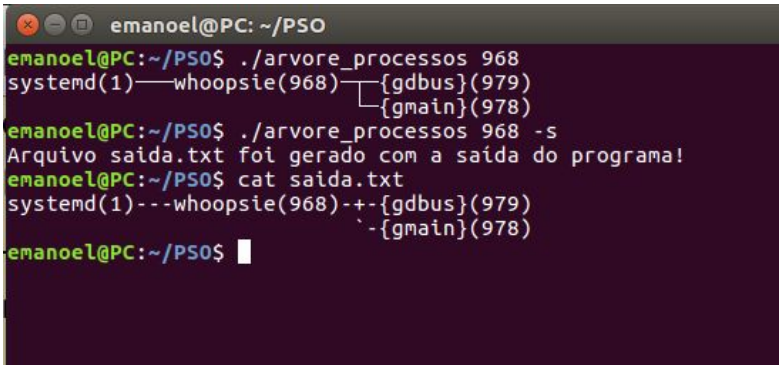
A solução deste algoritmo foi implementada usando também a função system(char*) da biblioteca stdlib.h. Ela consiste em pegar o PID do processo passado por linha de comando e executar “pstree -p -s PID_do_processo” no terminal. Adicionalmente, se o parâmetro -s for passado, o programa executa o comando “pstree -p -s PID_do_processo > saida.txt” gerando o arquivo saida.txt com a saída que seria exibida no terminal.

### 2.3.2. COMO COMPILAR E EXECUTAR

Para compilar o programa, basta digitar no terminal, na pasta onde está o arquivo com o código, o comando g++ arvore_processos.cpp -o arvore_processos. Para executar, digite ./arvore_processos <Pid_do_processo> -s no mesmo diretório que digitou o comando para compilação. Note que -s é parâmetro opcional.

### 2.3.3. RESULTADO

A figura abaixo mostra no terminal a árvore de processos para o processo com PID 968. Note que, quando o parâmetro -s foi passado na linha de comando, o programa gerou a saída no arquivo saida.txt.




### 2.3.4. IMPLEMENTAÇÕES ADICIONAIS

Uma versão adicional foi implementada e está disponível no link que contém todos os códigos do trabalho. Esta versão gera a árvore de processos em XML. Para compilar o código basta digitar g++ -std=c++11 arvore_processos_XML.cpp -o arvore_processos-XML. Para executar, ./arvore_processos-XML <PID_do_processo>, com isso é gerada a saída no arquivo processTree.xml. No entanto, esta implementação contém alguns pontos a serem melhorados, como uma melhor identificação dos campos xml do arquivo, e melhorar o desempenho para geração da árvore, pois pode demorar bastante caso o número de processos em execução seja muito elevado, devido a alta complexidade da solução implementada. A figura abaixo mostra parte da árvore gerada para o processo de PID 1 (init).