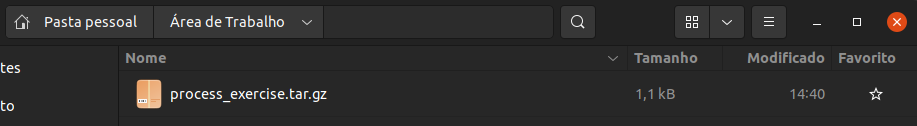
**Engenharia de Computação - Sistemas Operacionais - 4°P**

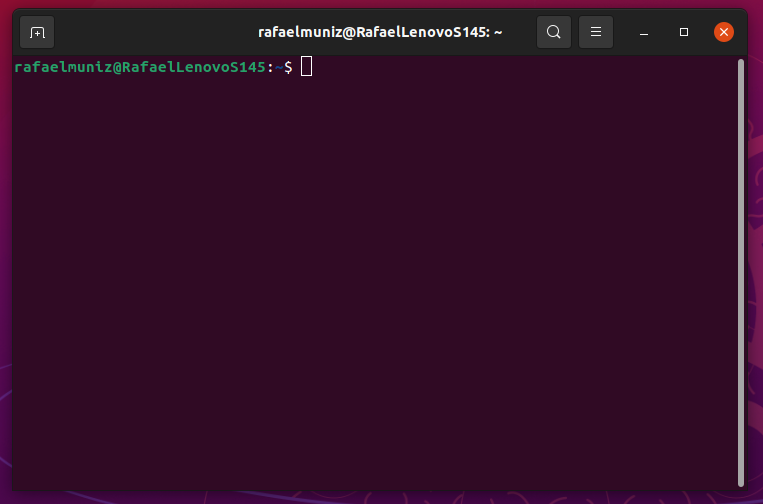
**Aluno: Rafael Victor Cordeiro Muniz**

**Roteiro:**

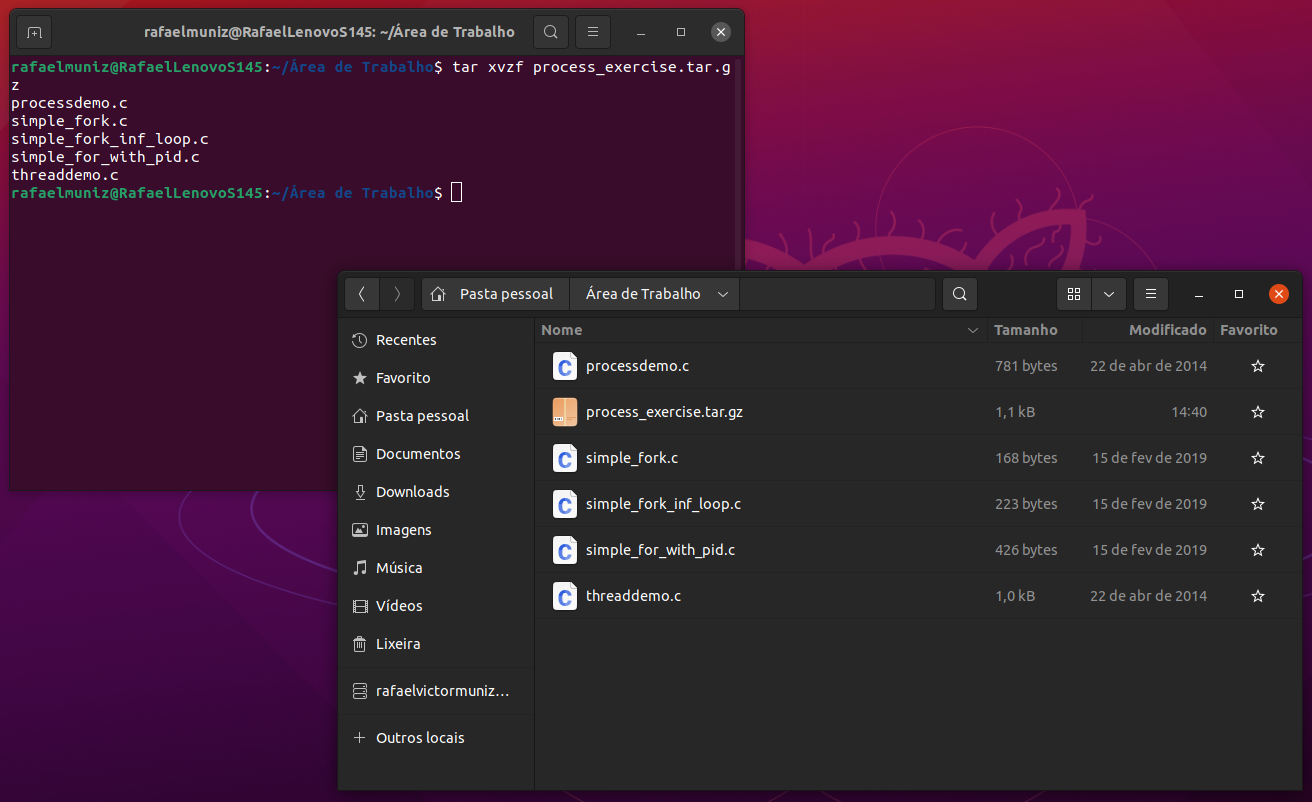
**1. Faça download do arquivo para a prática**

****

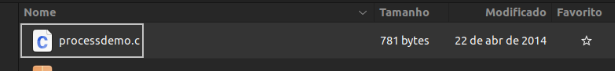
**2. Abra um terminal**

****

**3. Descompacte o arquivo. No terminal digite: "tar xvzf process\_exercise.tar.gz"**

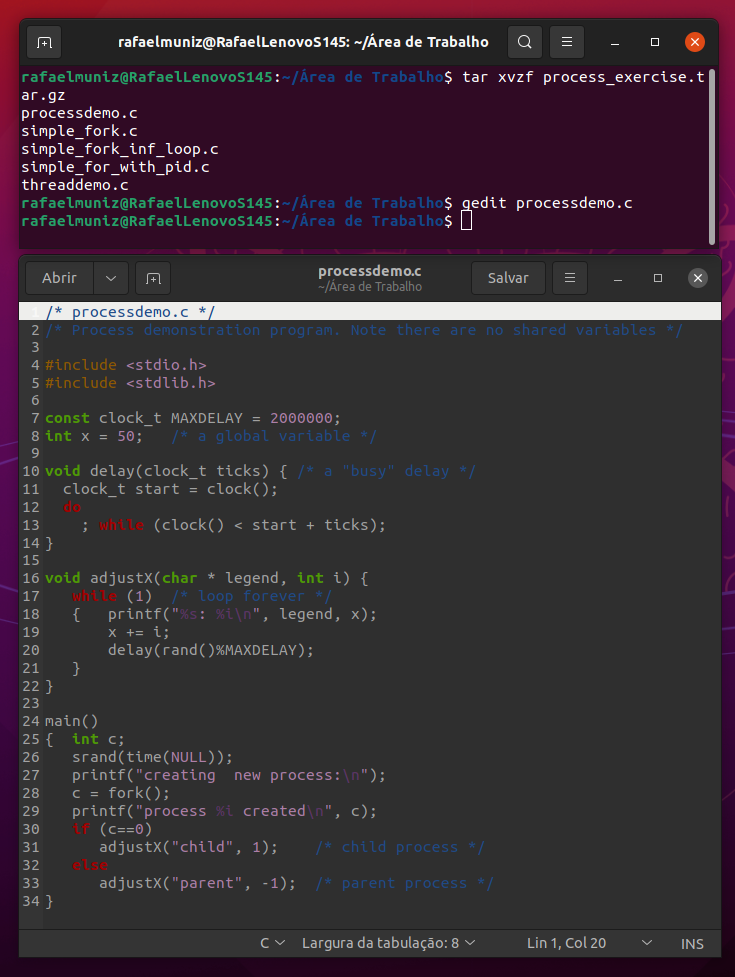
****

**4. Encontre o arquivo processdemo.c dentro da pasta descompactada. O programa usa a chamada fork() para criar um processo filho. Os processos pai e filho são executados separadamente e cada um chama a função adjustX() com parâmetros diferentes em cada processo**

****

**5. Vamos examinar o código-fonte e tentar determinar o que ele faz. (Você pode**

**fazer isso usando o programa Gedit.)**

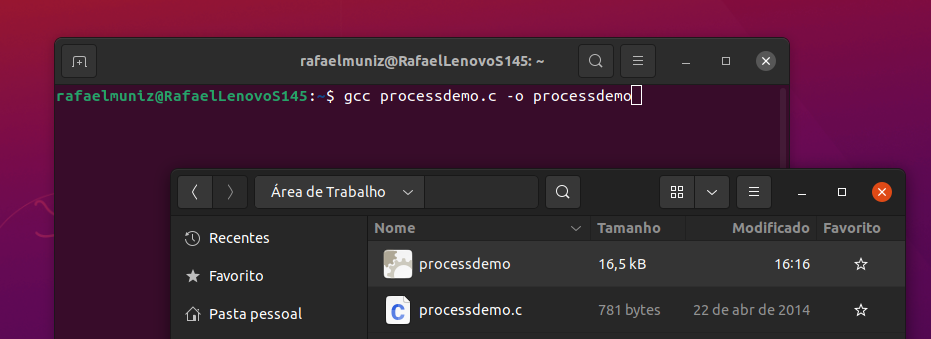
****

**6. Faça um exercício de tentar prever o que o código vai fazer**

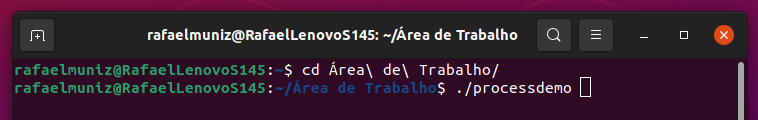
Aparentemente o código criará um novos processos, esses processos serão pai e filho.

**7. Compile o código-fonte. No terminal digite:**

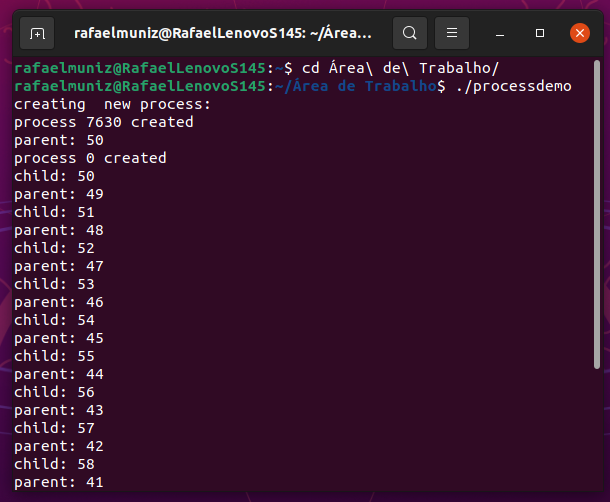
**gcc processdemo.c -o processdemo**

****

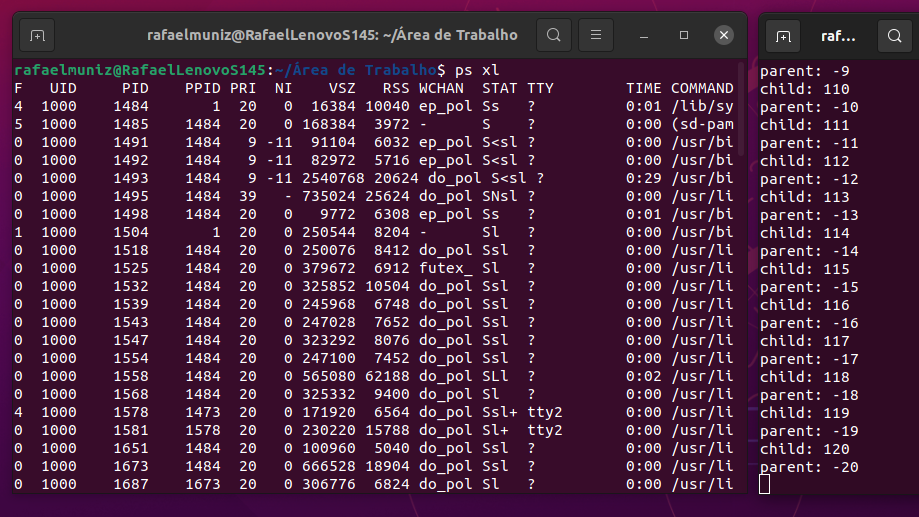
**8. Rode o programa. No terminal digite: ./processdemo**

****

**9. Descreva a saída e explique por que ela é dessa forma.**

****

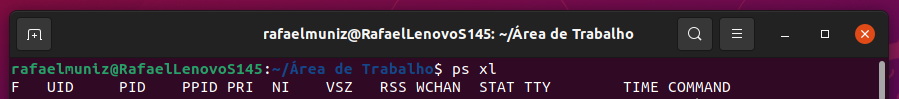
**10.Enquanto o programa roda, execute (em outro terminal) o comando ps xl**

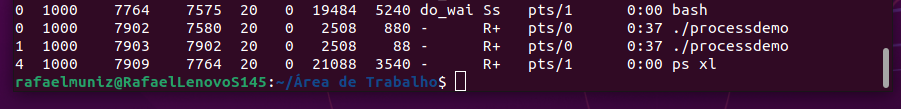
****

**11.Qual o processo pai e qual o processo filho? (Dica, verifique a coluna**

**PID e PPID. Se não souber o que é PID e PPID, procure no Google).**

**Justifique.**

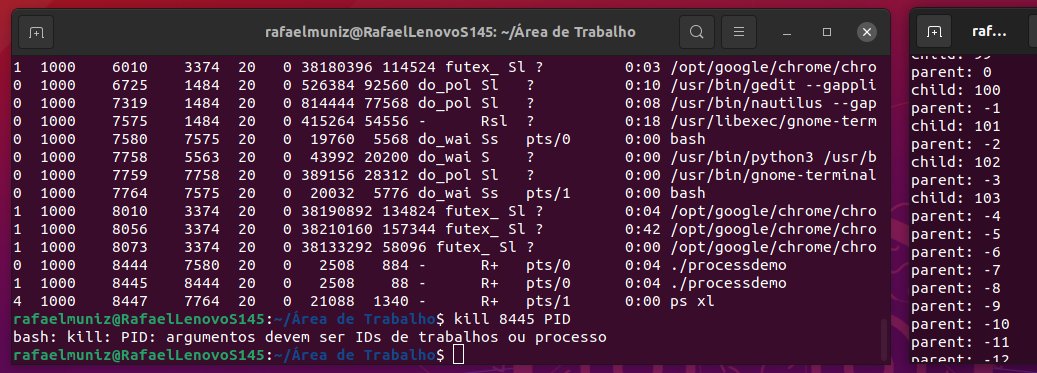
****

****

O processo pai tem o PID 7902, o PPID é o PID do pai do processo, no caso, o processo filho tem o PID 7903 e o PPID 7902. É um relacionamento pai-filho. Um único processo pai pode gerar vários processos filho, cada um com um PID exclusivo, mas todos compartilhando o mesmo PPID.

**12.Use o comando "kill -9 PID" para matar o processo filho. O que**

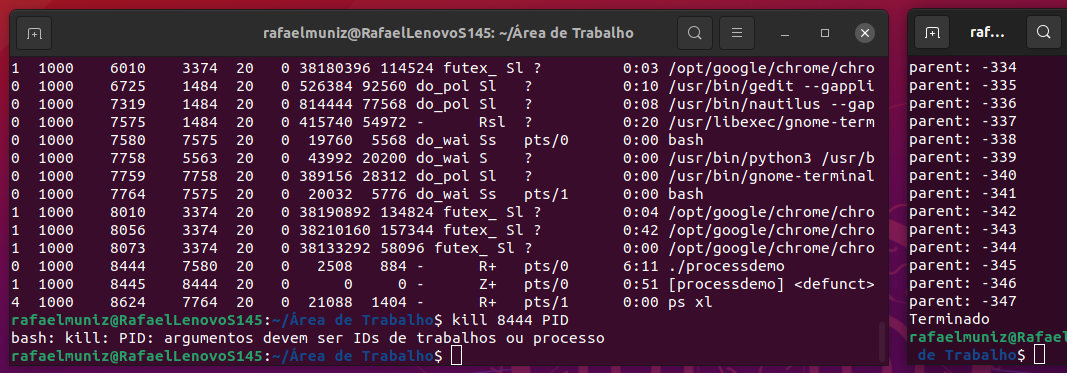
**aconteceu?**

****

Apenas o processo pai continuou o seu processo após matar o processo filho.

**13.Use o comando "kill -9 PID" para matar o processo pai. O que**

**aconteceu?**

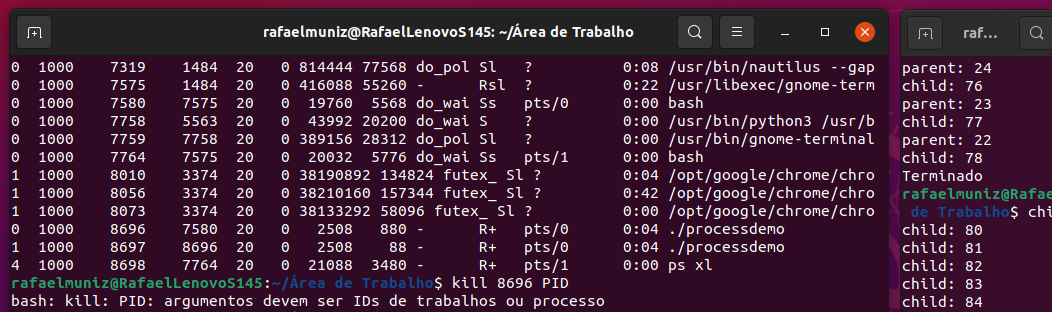
****

O processo pai foi encerrado e o programa terminado.

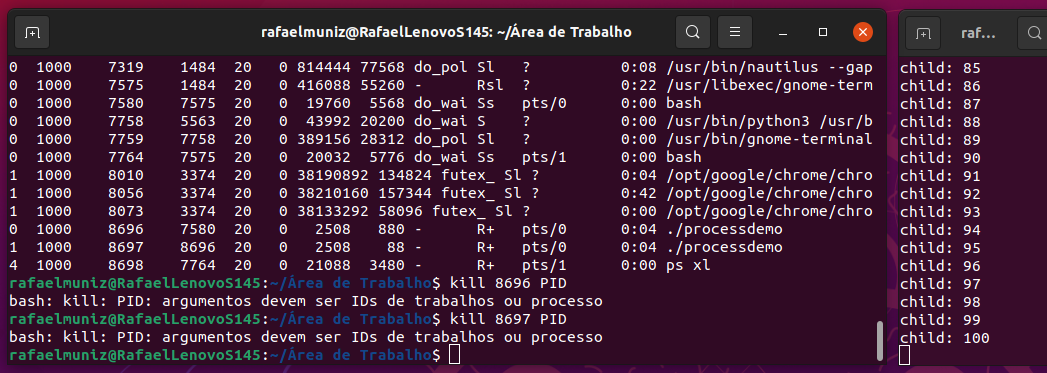
**14.Rode o programa novamente. Identifique e mate o processo pai primeiro**

**em seguida o filho. O que aconteceu?**

Matando processo pai:

****

Matando processo filho:

****

Após matar o processo pai, mesmo aparecendo a mensagem de que o programa havia terminado, o processo filho permaneceu executando, sendo necessário encerrá-lo individualmente para parar o processo.

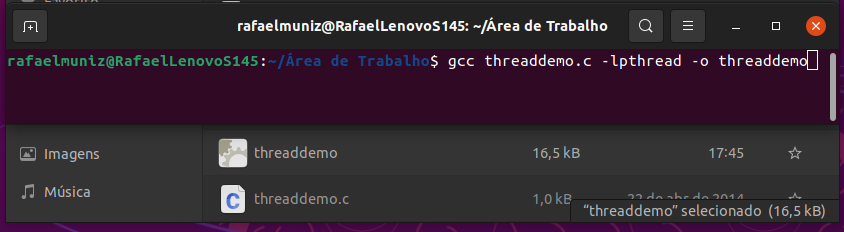
**15.Faz diferença matar o pai ou o filho antes?**

Sim, caso o pai seja morto antes, o filho se torna um processo com um novo pai, que no caso é o primeiro processo existente. Ele que passará a monitorar a sua execução.

Caso o filho seja morto antes, o pai ainda irá continuar executando seu processo da mesma forma.

**16.Compile o programa threaddemo.c com o comando:**

**gcc threaddemo.c -lpthread -o threaddemo**

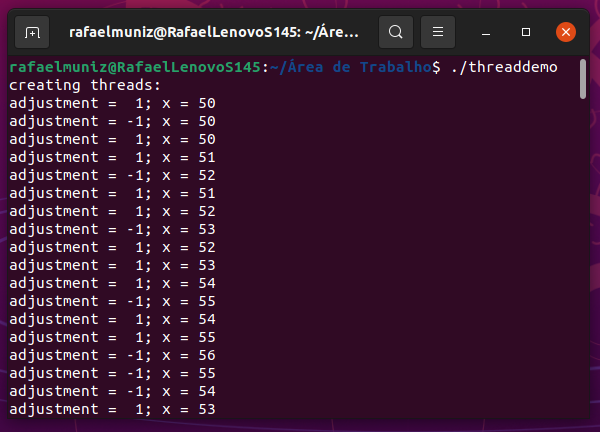
****

**17.Esse programa usa a biblioteca POSIX threads library e é muito similar a**

**processdemo.c, mas usa threads em vez de processos**

**18.Rode o programa. O que ele faz? Qual a diferença dele para o programa**

**processdemo.c?**

****

Ele cria um processo que demonstra a utilização de threads. O “processdemo” mostra como se dá a execução de um processo que utiliza de um filho para executar tarefas em conjunto. Já o “threaddemo” gera apenas um processo que se utiliza dos threads para executar suas tarefas.

**19.Qual a diferença de velocidade de saída (medido em linhas por**

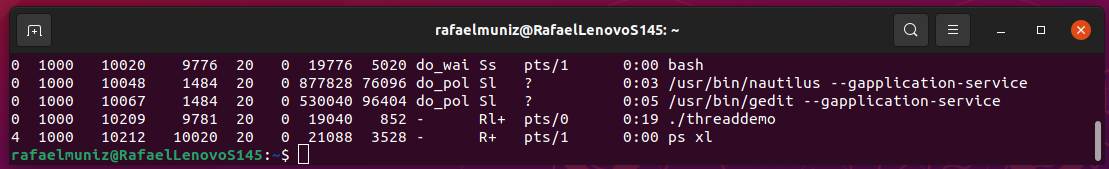
**segundo) comparado a processdemo? Quem é mais rápido? Você tem**

**uma ideia do porquê?**

O programa “threaddemo” é bem mais rápido que “processdemo”.

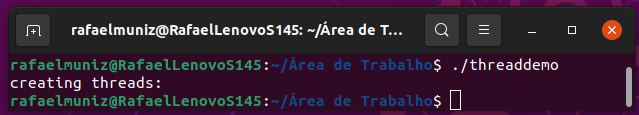
A diferença típica é que os threads (do mesmo processo) são executados em um espaço de memória compartilhado, enquanto os processos são executados em espaços de memória separados, dessa forma o thread consegue executar suas funções com mais velocidade.

**20.Use o comando ps xl para verificar que há apenas 1 processo threaddemo**

****

**21.Investigue o efeito de remover o loop infinito no fim do main(). O que**

**acontece? Por que?**

****

A execução não se torna mais um looping infinito. Ao remover o while (1) (true na linguagem C), o programa é encerrado quando chega ao fim. Como o uso de threads requer que o programa esteja em execução para que os processos permaneçam, ao encerrar o programa os processos não continuam sua execução.

**22.Modifique o programa threaddemo.c para ele fazer a mesma coisa que**

**processdemo.c**

****

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <pthread.h>**

**int x = 50; /\* a global (shared) variable \*/**

**const clock\_t MAXDELAY = 2000000;**

**void delay(clock\_t ticks) { /\* a "busy" delay \*/**

**clock\_t start = clock();**

**do**

**; while (clock() < start + ticks);**

**}**

**void \* adjustX(void \*n)**

**{ int i = (int)n;**

**while (1) /\* loop forever \*/**

**{ printf("adjustment = %2i; x = %i\n", i, x);**

**x += i;**

**delay(rand()%MAXDELAY);**

**}**

**return(n);**

**}**

**main()**

**{ int a;**

**srand(time(NULL));**

**pthread\_t up\_thread, dn\_thread;**

**printf("creating new process:\n");**

**c = fork();**

**if (c==0)**

**pthread\_create(&up\_thread, c, adjustX, (void \*)1);**

**else**

**pthread\_create(&dn\_thread, c, adjustX, (void \*)-1);**

**}**