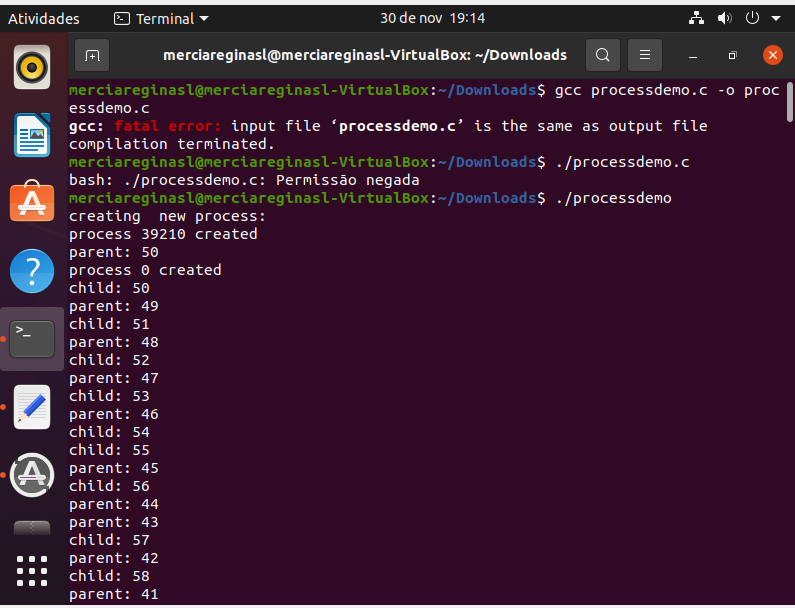
IFPB

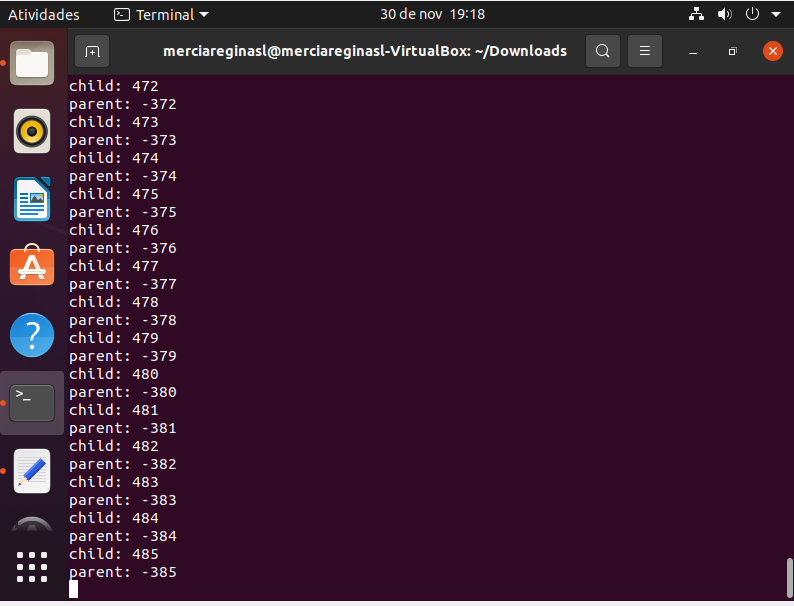
ALUNA: MÉRCIA REGINA DA SILVA

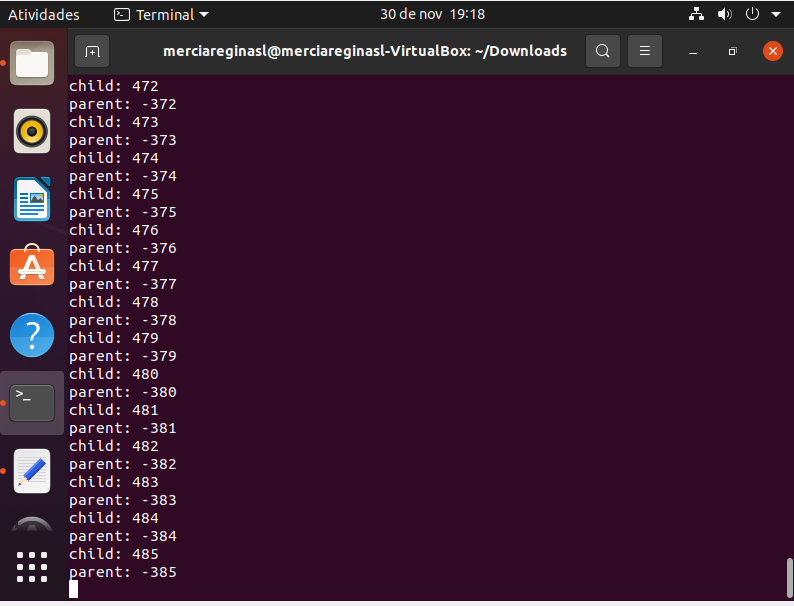
ATIVIDADE PRÁTICA I

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

Ao executar o programa no terminal, a função ***fork()*** cria processo filho idêntico ao pai, exceto por alguns atributos e recursos, o ***fork*** duplica ou clona o processo executa a chamada, donde o processo filho é uma cópia fiel do pai. O processo pai e filho continuam a sua execução na instrução chamada ***c = fork().***

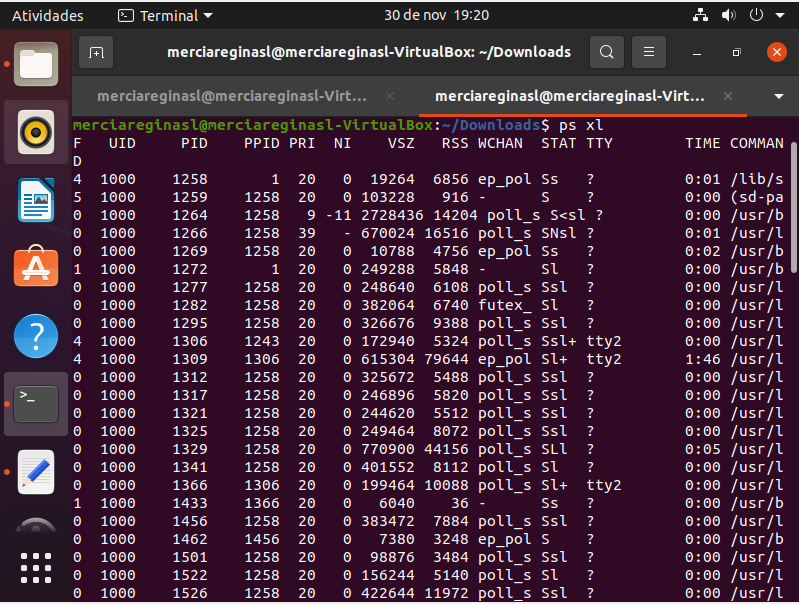
*Executando o programa*

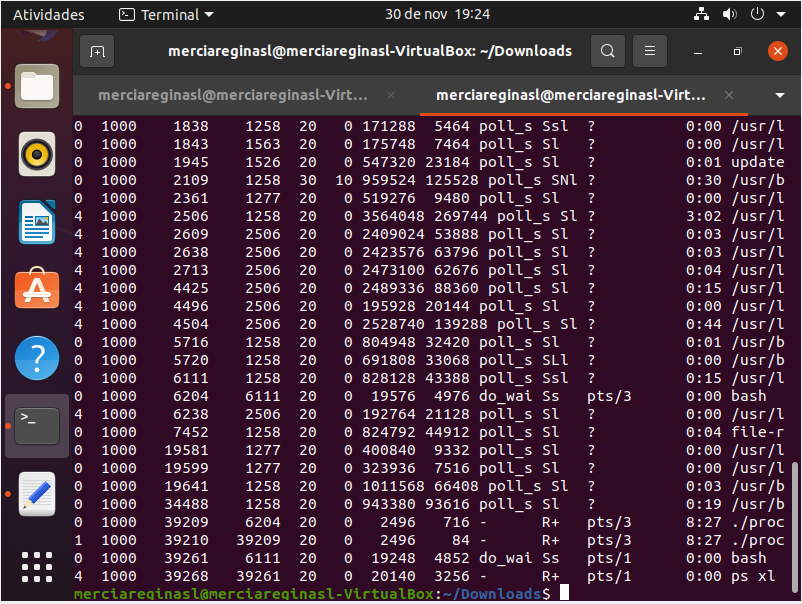




Os processos pai e filho partilham o mesmo código. O segmento de dados do usuário do novo processo (filho) é uma cópia exata do segmento correspondente ao processo antigo (pai). Por outro lado, a cópia do segmento de dados do filho do sistema pode diferir do segmento do pai em alguns atributos específicos (como por exemplo, o PID, o tempo de execução, etc.). Os filhos herdam uma duplicata de todos os descritores dos arquivos abertos do pai (se o filho fecha um deles, a cópia do pai não será modificada).

Realizando 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.

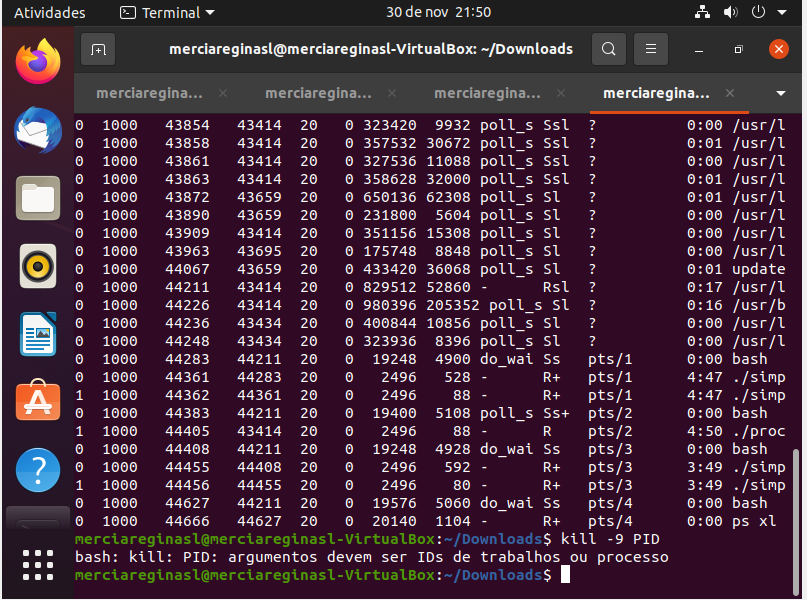
***PID (Process IDentification)*** é o número de Identificação (ID) de um processo (3º coluna) recebendo um número exclusivo chamado ID do processo PID que identifica esse processo do sistema. Neste caso, o PID seria o filho.

***PPID (Parent Process IDentification)*** é o número de identificação do processo-pai. Quando um processo inicia outro processo, passa a existir uma relação de pai->filho dentro do sistema, o processo que cria um outro processo é o processo-pai; o processo criado é o processo filho.

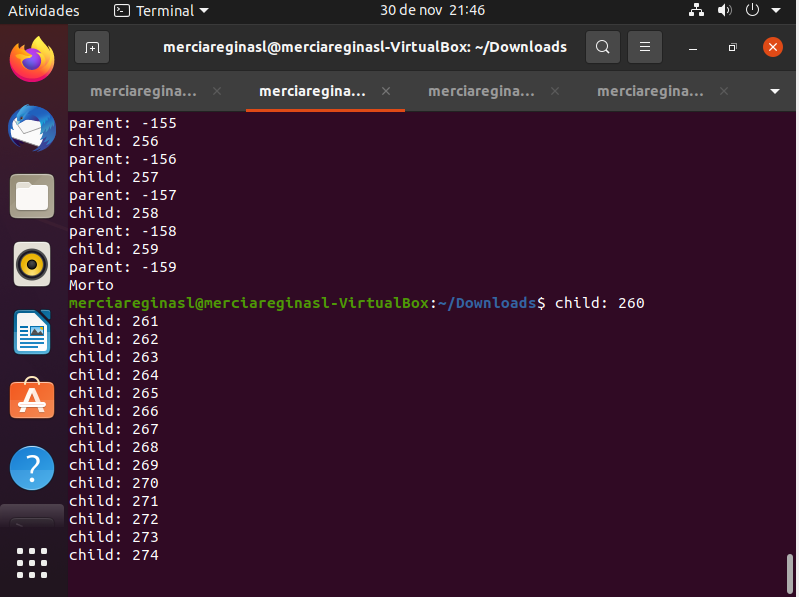
12. Use o comando "kill -9 PID" para matar o processo filho. O que aconteceu?

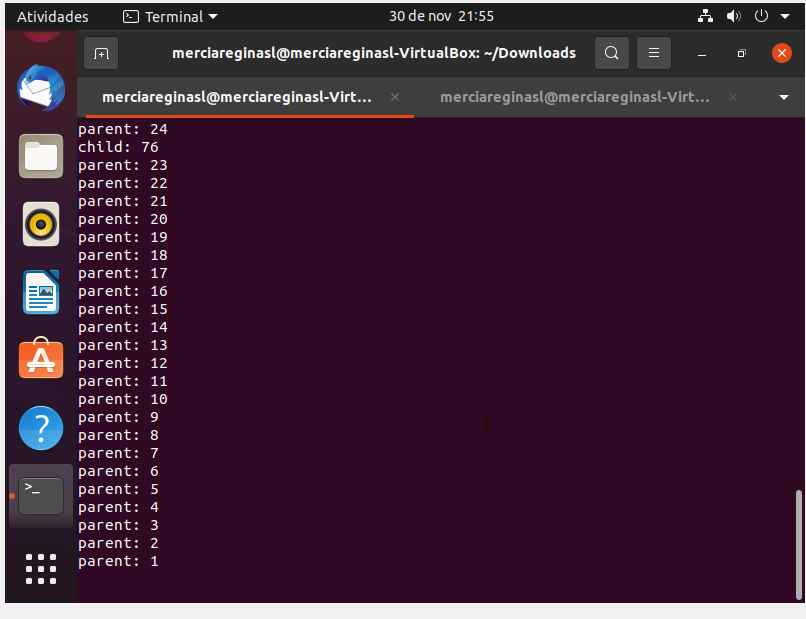
13. Use o comando "kill -9 PID" para matar o processo pai. O que aconteceu?

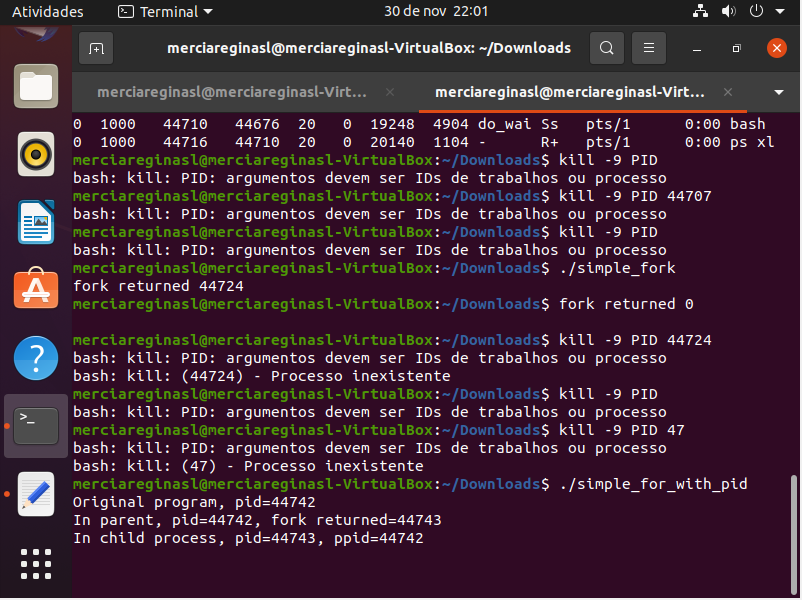
Ao realizar o comando **kill -9 PID** ocorreu:



Digitei um ID do PID







Ao realizar o comando kill foi informado morto – child e logo após outro comando lista apenas o pai (parents).

Se a quinta coluna não for 1, significa que ele possui um processo pai. Se o seu PPID = 1, pense em quais outros dispositivos ou processos podem estar relacionados a ele.

14. Rode o programa novamente. Identifique e mate o processo pai primeiro em seguida o filho. O que aconteceu?

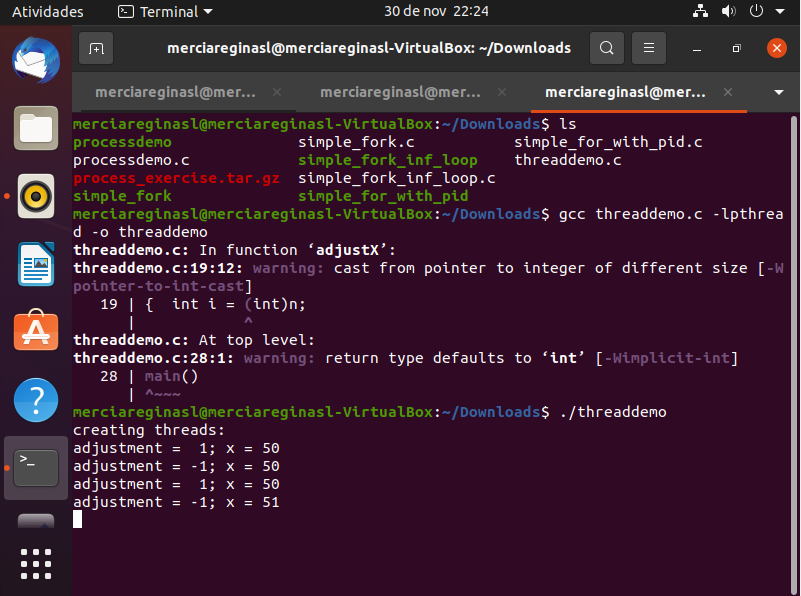
15. Faz diferença matar o pai ou o filho antes?

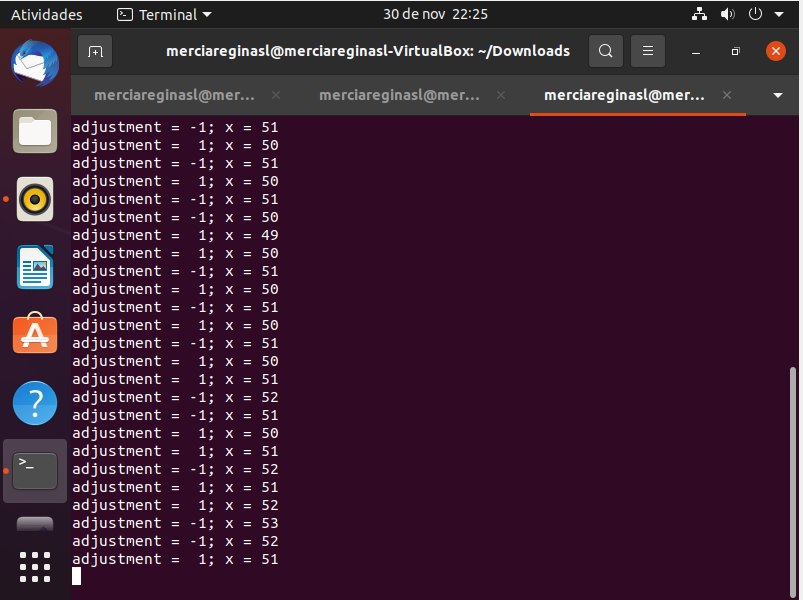
Notei que se o pai e o filho vivem, uma interrupção com o comando ***kill*** irá destruir os dois processos. Entretanto, se um filho vive enquanto seu pai está morto, uma interrupção pode não matá-lo.

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?

É possível notar que o programa que utiliza ***threads*** é mais complicado do que o que não utiliza. No contexto de um processo, temos que ele possui pelo menos uma ***thread*** de execução.

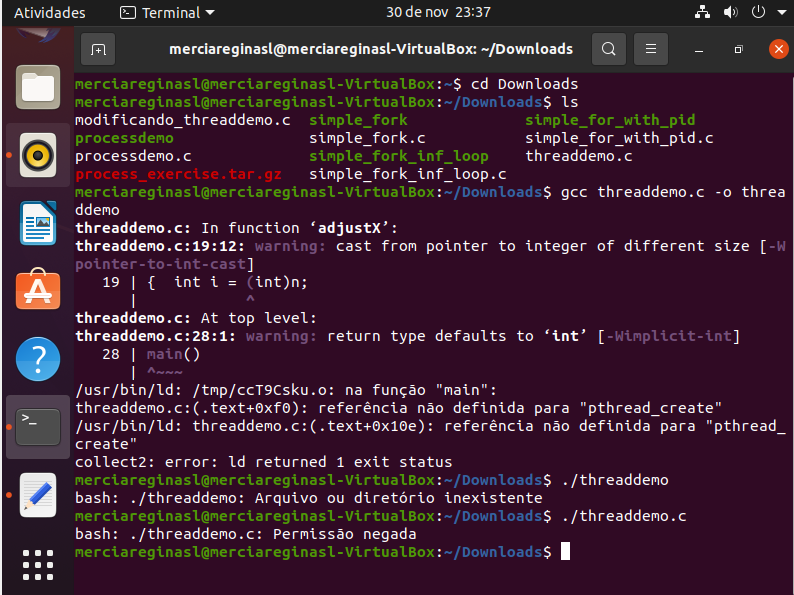
Quando mais de uma ***thread*** tenta acessar uma variável ao mesmo tempo temos uma condição de corrida. Ou seja, elas concorrem por algum recurso. O impacto dessa condição, é que o resultado da computação dessas ***threads***depende de quem executou primeiro. Não há como garantir que independente do número de execuções o resultado será o mesmo.





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

Ao realizar a modificação a compilação apresenta erro. Seus encadeamentos estão retornando à fila os respectivos loops de eventos, mas eles saem antes de realmente executar o loop de eventos, para que os retornos de chamada nunca sejam executados.



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





MODIFICANDO THREADDEMO.C

|  |
| --- |
| #include <stdio.h> |
|  |

|  |
| --- |
| #include <stdlib.h> |
|  |

|  |
| --- |
| #include <pthread.h> |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| const clock\_t MAXDELAY = 2000000; |
|  |

|  |
| --- |
| const char \*FIRST = "first thread"; |
|  |

|  |
| --- |
| const char \*SECOND = "second thread"; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| 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; |
|  |

|  |
| --- |
| int x = 50; |
|  |

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

|  |
| --- |
| { |
|  |

|  |
| --- |
| printf("%s: %i\n", i == 1 ? FIRST : SECOND, x); |
|  |

|  |
| --- |
| x += i; |
|  |

|  |
| --- |
| delay(rand()%MAXDELAY); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| return(n); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| main() |
|  |

|  |
| --- |
| { int a; |
|  |

|  |
| --- |
| srand(time(NULL)); |
|  |

|  |
| --- |
| pthread\_t up\_thread, dn\_thread; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| pthread\_attr\_t \*attr; /\* thread attribute variable \*/ |
|  |

|  |
| --- |
| attr=0; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| printf("creating threads:\n"); |
|  |

|  |
| --- |
| pthread\_create(&up\_thread,attr, adjustX, (void \*)1); |
|  |

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

|  |
| --- |
|  |
|  |

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

|  |
| --- |
| { ;} |
|  |

}

|  |
| --- |
|  |