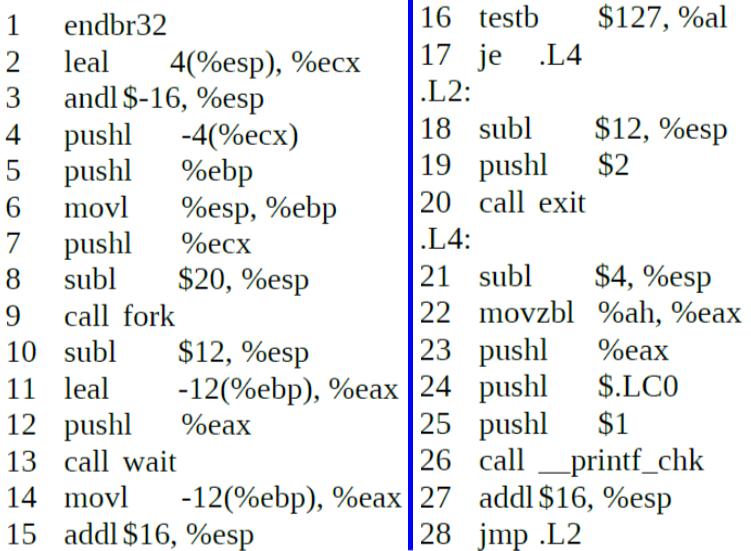
Lista 5 – Felipe Melo – Thalles Nonato

DRE Felipe: 119093752

DRE Thalles: 119058809

**Questão 1) a) O código de montagem referente ao C, que não precisa ser comentado, a priori, segue:**



**Questão 1) a1) Para testar que houve um término normal do processo filho, alguns bits são testados. Quais são esses bits e qual o estado dos bits que indica que houve um término normal do processo filho? É preciso mostrar as linhas pertinentes no código de montagem que justificam a resposta:**

Na linha 16 há uma instrução testb $127, %al, responsável por realizar um AND bit a bit entre o binário 11111112 e o byte menos significativo de %eax. Na linha seguinte, há um je .L4 que realiza um salto para .L4 caso %al seja zero. Assim, podemos verificar que o estado dos bits que indica um término normal do processo filho é qualquer estado diferente de zero. Uma outra maneira de verificar isso é atentando para o fado de que já que a macro WIFEXITED() retorna um valor diferente de zero caso o processo filho tenha terminado com êxito, o código de montagem nos mostra um desvio para .L4 (label responsável por imprimir a string na tela) se %al for zero, indicando término normal do processo filho.

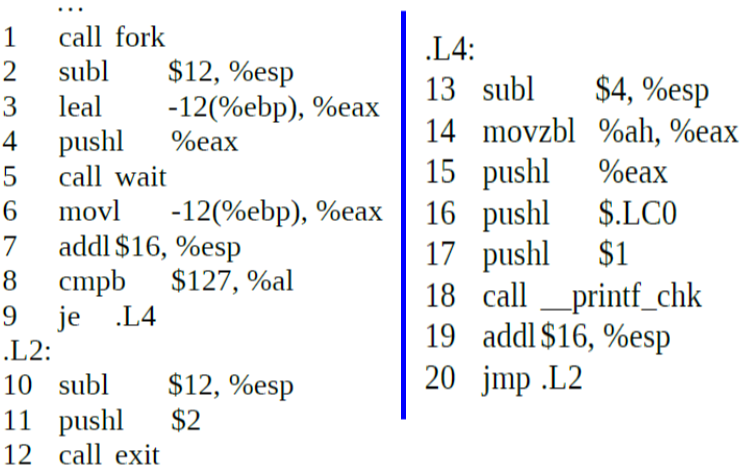
**Questão 1) a2) Onde o valor de término do processo filho fica armazenado? Justifique pelo código de montagem. É preciso explicar as linhas pertinentes no código de montagem que justificam a resposta:**

Fica armazenado em %eax. Verificamos isso das linhas 22 a 25 do código de montagem. Isso porque a rotina main imprime WEXITSTATUS() e analisando essas instruções vemos que são empilhados a string (em .LC0), o valor de término, armazenado em %eax, que serão impressos pela função \_\_printf\_chk, além do valor 1 que verifica êxito dessa função.

**Questão 1) a3) Quais os valores possíveis de término de um processo filho, em geral? No exemplo, o filho retornará 1. Queremos saber quais os valores possíveis para retorno. Justifique e explique usando as linhas pertinentes do código de montagem:**

Na linha 22 há uma instrução responsável por copiar %ah para %eax (valor de término do processo filho). %ah é o 2º byte menos significativo de %eax, isto é, o maior valor que pode ser representado nessa faixa de bits 28 – 1. Portanto, todos os números de 0 a 255 são possíveis para retorno.

**Questão 1) b) A parte relevante do código de montagem é:**



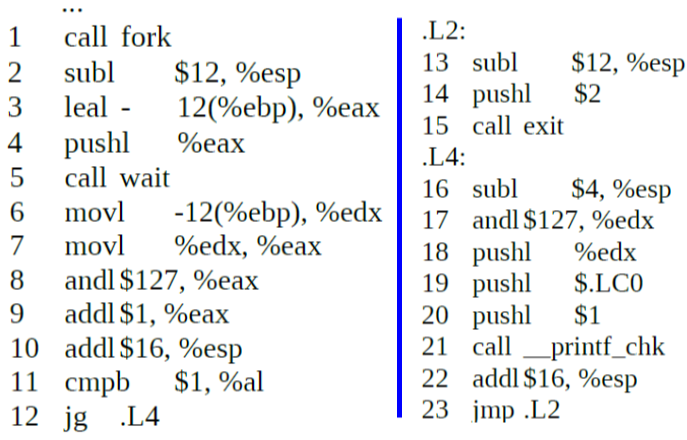
**Questão 1) b1) Para identificar o sinal que causou a parada do processo filho, alguns bits da variável status são testados. Quais são esses bits e qual o estado desses bits indica que o processo filho está parado? É preciso mostrar as linhas no código de montagem que justificam sua resposta:**

Na linha 8 há uma instrução cmpb $127, %al, responsável por comparar efetuando a subtração do byte menos significativo de %eax com o binário 11111112. Na linha seguinte, há um je .L4, que realiza um salto para .L4 (label responsável por imprimir a string na tela) caso %al tenha todos os bits em 1. Assim, podemos verificar que o estado dos bits que indica que o processo filho está parado é qualquer estado diferente de zero.

**Questão 1) b2) Onde o valor do sinal que causou a parada está armazenado e quais os possíveis valores para esse sinal? Justifique e explique usando as linhas pertinentes do código de montagem:**

Fica armazenado em %eax. Verificamos isso das linhas 14 a 18 do código de montagem. Isso porque a rotina main imprime WSTOPSIG() e analisando essas instruções vemos que são empilhados a string (em .LC0), o valor de término, armazenado em %eax, que serão impressos pela função \_\_printf\_chk, além do valor 1 que verifica êxito dessa função.

**Questão 1) c) A parte relevante do código de montagem gerado é:**



**Questão 1) c1) Para identificar que o processo filho foi interrompido por um sinal, alguns bits da variável status são testados. Quais são esses bits e qual o estado desses bits indica que o processo filho sofreu uma interrupção? É preciso mostrar as linhas pertinentes no código de montagem que justificam sua resposta:**

Na linha 11 é responsável por comparar %al com o valor 1, efetuando a subtração do byte menos significativo de %eax com 1. Na linha seguinte, há um jg .L4, que realiza um salto para .L4 (if(WIFSIGNALED(status)). Caso ocorra esse desvio para .L4, significa que o processo foi interrompido por um sinal. Os números dentro dos limites de %al (8 bits) que fazem com que essa condição seja válida variam de 2 a 255.

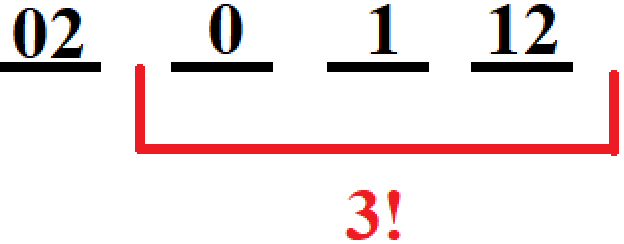
**Questão 1) c2) Onde o valor do sinal que interrompeu está armazenado e quais os valores possíveis desse sinal? Justifique e explique usando as linhas pertinentes do código de montagem:**

O valor está armazenado na variável status, em (%ebp - 12). Os possíveis valores estão no intervalo de 2 a 255.

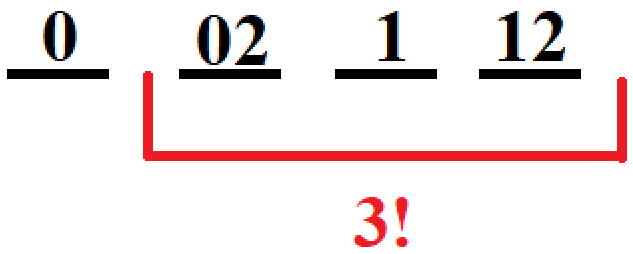
**Questão 2) A função atexit recebe como argumento um ponteiro para uma função. Esta função é acrescentada a uma lista de funções (inicialmente vazia) que serão executadas quando a função exit é chamada. Assuma que o sistema operacional executará a rotina end imediatamente com o término do processo, fazendo com que o 2 apareça sempre imediatamente após a impressão provocada por main. Apresente e justifique todas as saídas possíveis deste programa. Lembre-se que processos podem ser executados em qualquer ordem:**

Após analisar as possíveis ordens de execução dos processos, descobrimos que algumas saídas começam com [02], enquanto outras começam com [0]. Além disso, sempre existe a sequência [12] devido ao atexit() do primeiro fork(), que imprime 2 ao término de um dos processos. Também sabemos que devido ao else, 1 nunca pode ser impresso primeiro, já que há um wait(). Portanto, analisando todas as possibilidades, percebemos que há um comportamento que pode ser expresso em termos de fatoriais.

Começando com [02], temos:



Começando com [0]:

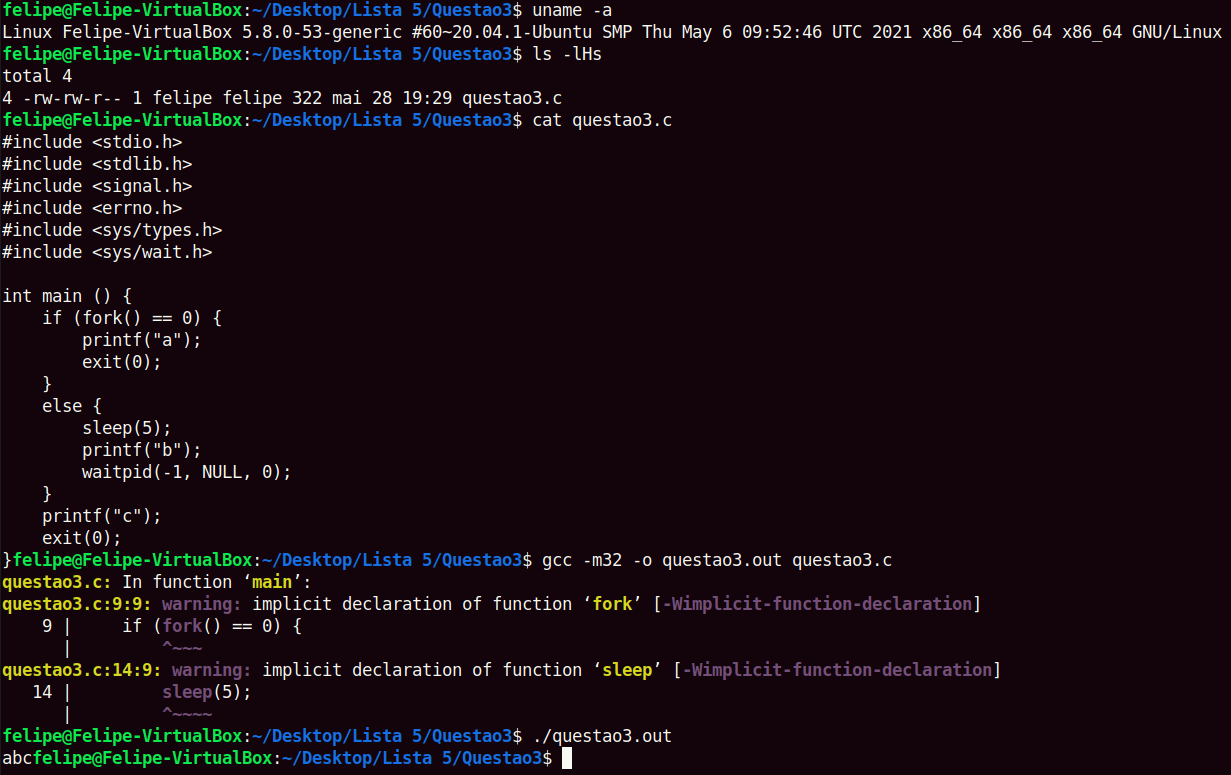


Desse modo, como pode ocorrer uma ou outra, somamos 3! + 3! = 12 possibilidades de impressão. São elas:

020112 021012 021120 020121 021201 021210

002112 010212 011202 002121 012021 012102

**Questão 3) Liste o sistema operacional em uso. Rode o programa abaixo e copie a tela do terminal com a impressão de saída:**

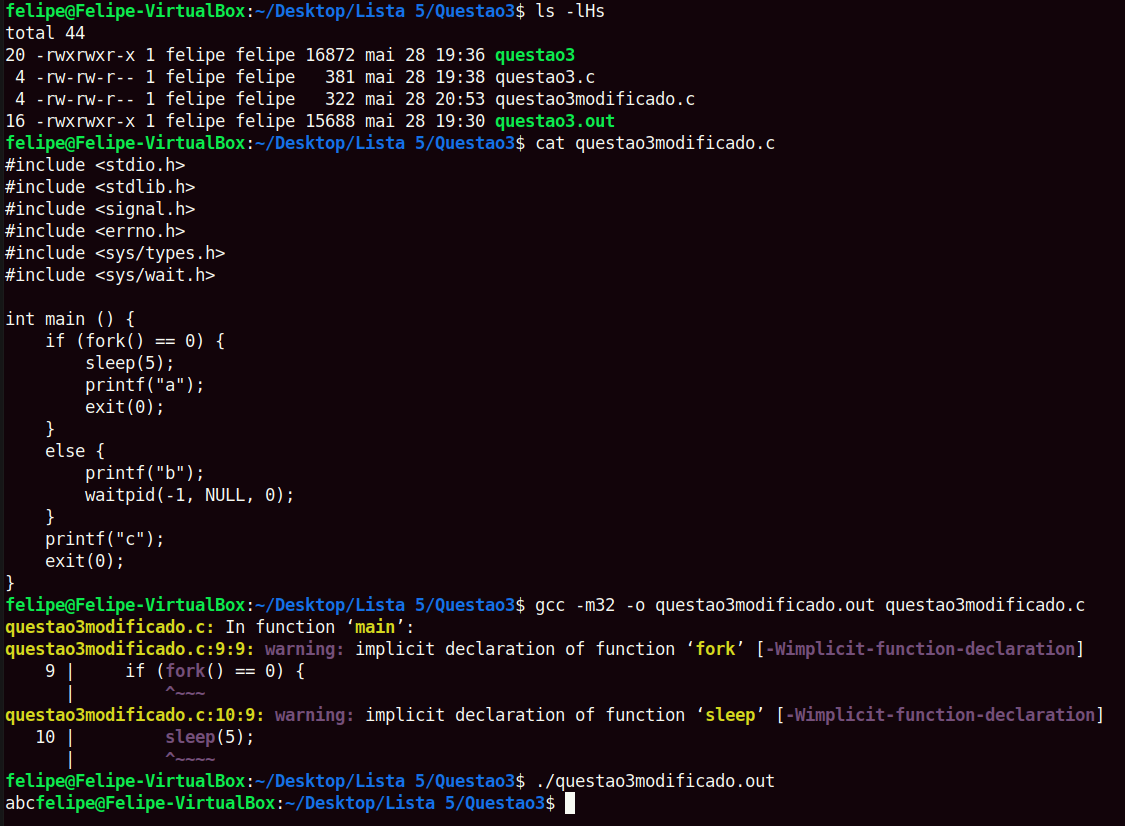


**Questão 3) a) Justifique a saída do programa em termos das chamadas de sistema utilizadas:**

“a” é impresso quando o processo filho é chamado. Após isso, esse processo é encerrado com a função exit(0) e o processo pai, após 5 segundos imprime “b”. Depois, há um waitpid(-1, NULL, 0), que bloqueia o acesso do processo pai até os processos filhos serem finalizados para imprimir “c”. No entanto, como o processo filho já foi encerrado, é impresso “c” logo assim que foi impresso “b”.

Caso o processo pai inicie antes do processo filho, não ocorre nenhuma mudança, uma vez que ele “dorme” 5 segundos, tempo suficiente para término do processo filho.

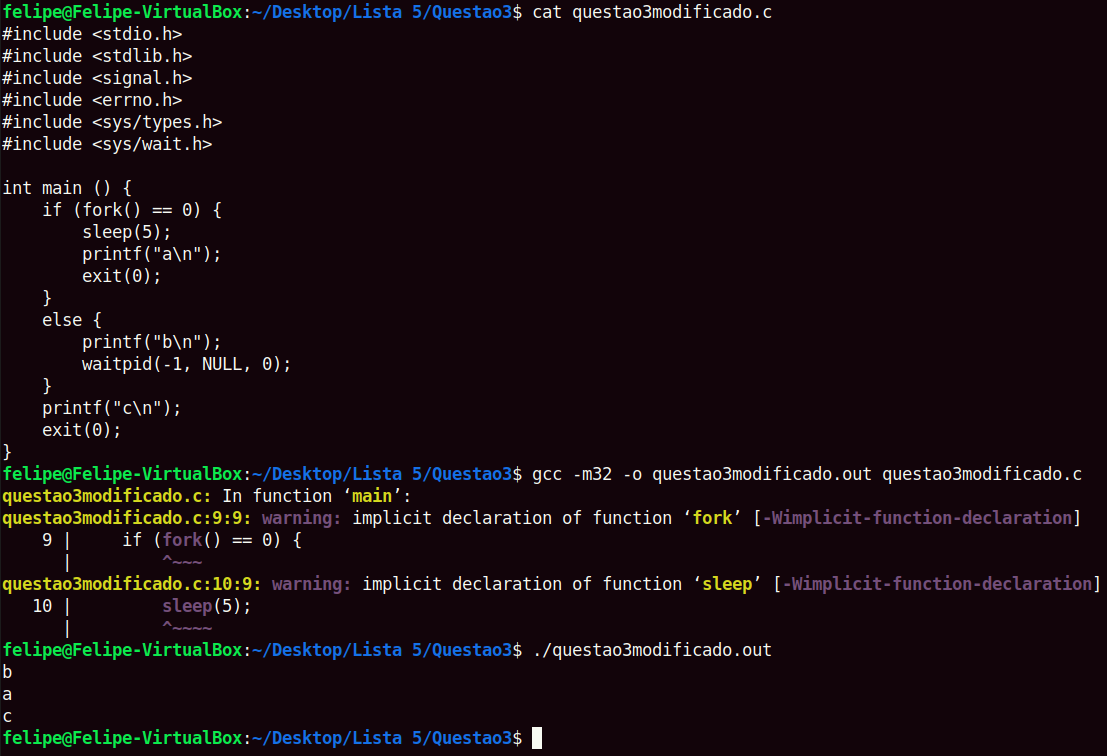
**Agora, rode o programa modificado e copie a tela do terminal com a impressão de saída:**



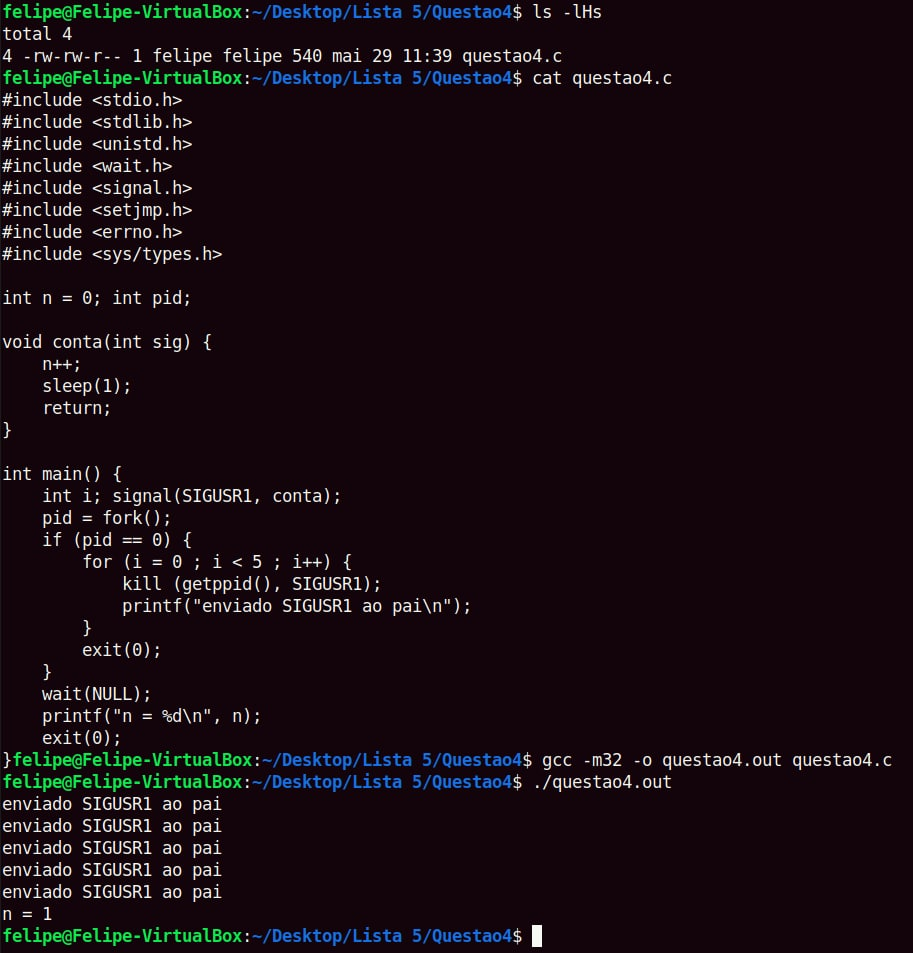
**Questão 3) b) Justifique a saída do programa em termos em termos das chamadas de sistema utilizadas. Encontre justificativa plausível. O programa deve ser executado como no enunciado, sem qualquer alteração, e sua saída deve ser explicada:**

Foi impresso o mesmo conteúdo de antes da modificação, em mesma ordem, mas em tempo distinto (após 5 segundos foi impresso “abc” de uma vez). Isso ocorreu devido à ausência de “\n” ao final de cada printf(). Isso ocorre pois, quando há criação de um novo processo com fork(), o processo filho herda tudo do processo pai, inclusive o buffer que não é limpo após a chamada de printf().

Caso haja “\n” em cada um dos 3 printf(), é impresso “bac”, como pode ser visto na imagem abaixo.

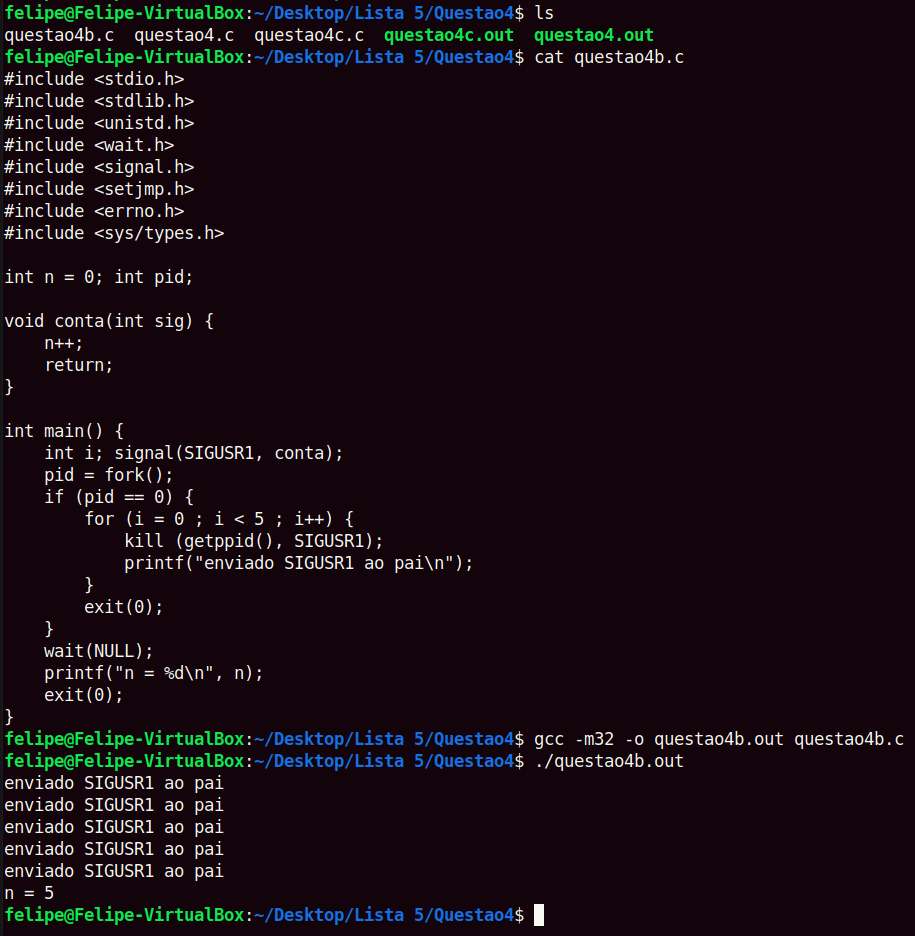


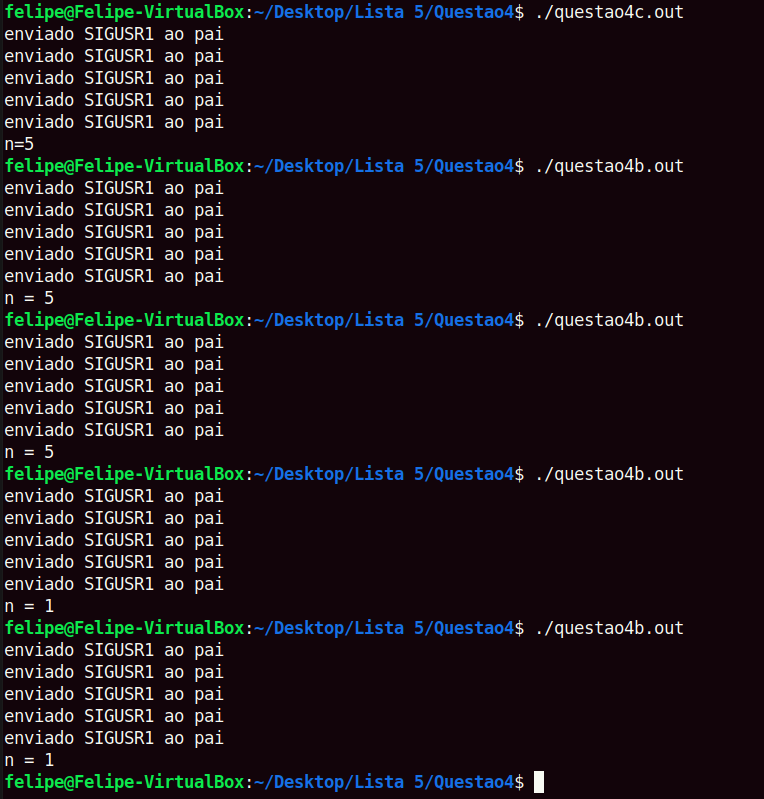
**Questão 4) a) Capture numa mesma tela de terminal a compilação e execução do programa. Analise o programa, explique a interação entre os processos e justifique a saída obtida:**



Há a tentativa de envio de um sinal, mas ocorre falha pois como as instruções ocorrem simultaneamente, ocorre erro.

**Questão 4) b) Elimine o comando sleep(1) da rotina conta e recompile. Rode algumas vezes e se houver diferentes saídas, capture a tela que demonstra isso:**

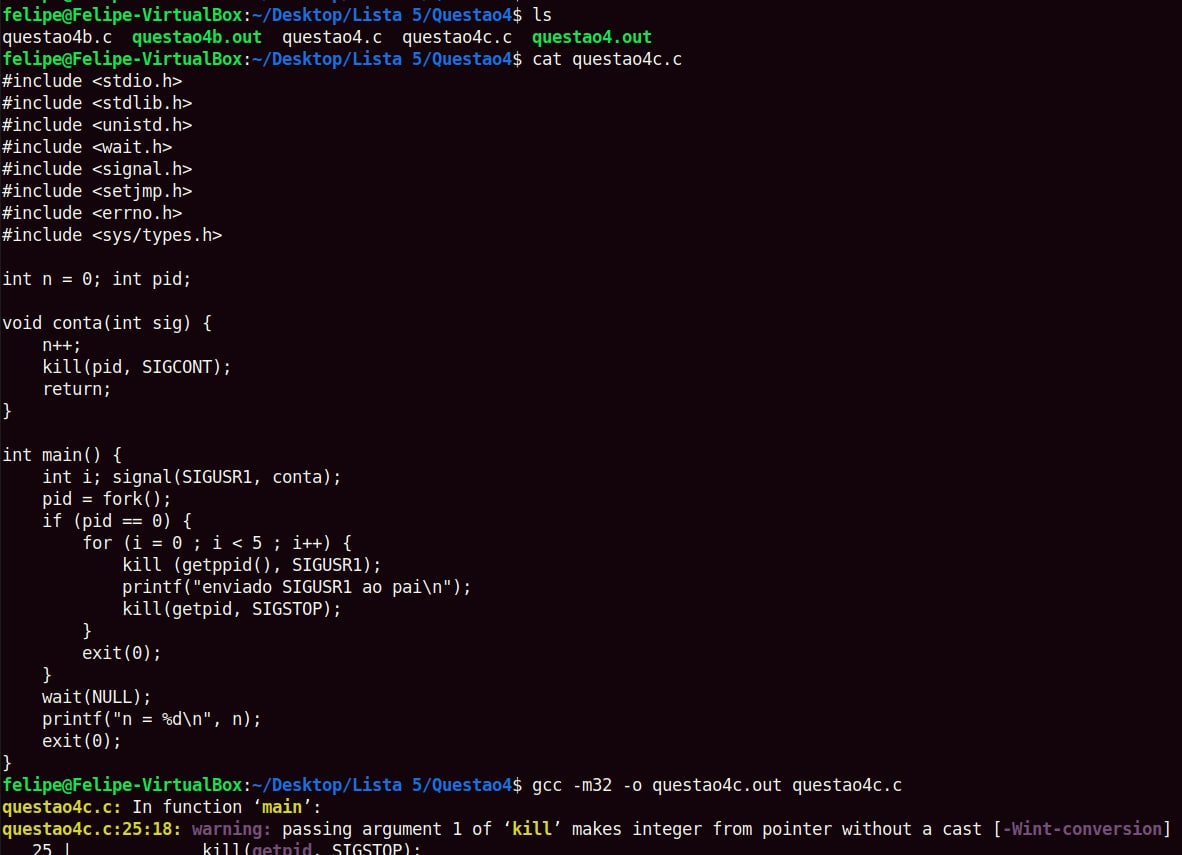




**Questão 4) c1) Explique a ideia do programador ao fazer a modificação no código. Isso vai funcionar na sua opinião?**

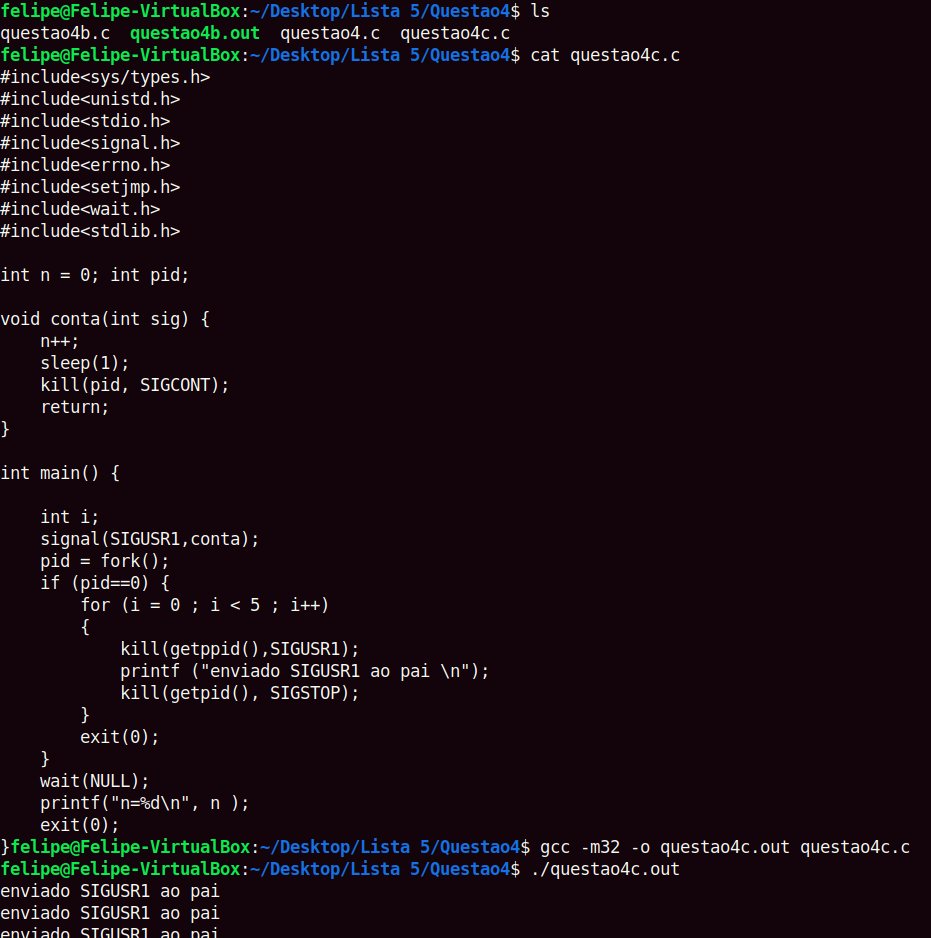
getppid() retorna o ID do processo pai do processo que chamou essa função, enquanto getpid() retorna o ID do processo atual. Possivelmente, a ideia do programador era tirar o sleep() para que as execuções ocorram simultaneamente e sinais não serem enviados em momentos errados. Acreditamos que isso não funcionará, pois o sleep() estava lá para garantir que o pai não enviasse SIGCONT ao filho antes de entrar em parada. Caso contrário, será possível que o pai envie SIGCONT antes do filho parar, o que fará o filho parar e nunca retomar.

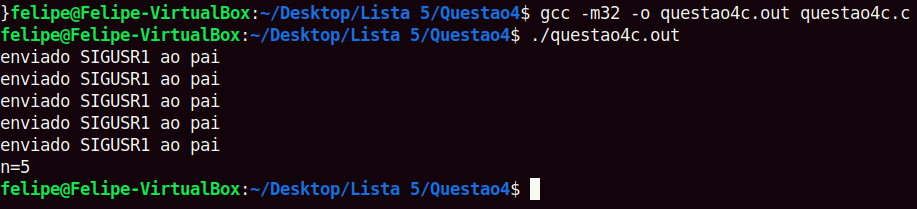
**Questão 4) c2) Capture numa mesma tela de terminal a compilação e execução do programa. Era o que seria esperado? Analise o programa, explique a interação entre os processos e justifique a saída obtida:**



O processo pai ao criar o filho, entra em espera bloqueante ao executar wait(NULL), esperando pelo término de qualquer um de seus filhos. Quando o sinal SIGCHLD é enviado ao pai (signal(SIGUSR1, conta)), o filho entra num loop com i = 0, envia SIGUSR1 ao pai. Ao receber SIGCONT, o filho continua no for incrementando i e repetindo a sequência. Cada SIGUSR1 entregue ao pai força ele a entrar em conta(), incrementando n, depois envia SIGCONT ao filho para que continue em execução. n será incrementado de 1 a 5 quando o filho finalmente terminar, o pai sairá do wait(NULL) bloqueante e imprimirá “n=5”.

**Questão 4) c3) Qual a modificação no código deste item C que você faria para acrescentar apenas um sleep(1) no código acima? Justifique seu raciocínio para a solução proposta, faça a alteração e veja se funciona:**





A justificativa para colocar sleep(1) em conta(), antes do kill(pid, SIGCONT) é para evitar deadlock entre os processos.