

Lista 5 - Comp Prog

Alunos: Luiz Rodrigo Lacé Rodrigues (DRE:11804983)

Livia Barbosa Fonseca (DRE:118039721)

Questão 1)(40 pontos) Seja dado o programa abaixo:

```
#include <stdio.h>

int main(int argc, char *argv[], char *envp[]) {
    int i;
    for (i=0;argv[i] != NULL; i++)
        printf("argv[%2d]: %s\n",i, argv[i]);
}
```

a) (5) Crie o arquivo nome.c, compile as opções -m32 -fno-PIC -O2 -S para gerar o arquivo nome.s. Imprima este arquivo. Comente as linhas do código de montagem. A partir de nome.s crie um executável nome, rode e imprima a tela de saída do programa.

Compilando o código do arquivo nome.c com o comando **gcc -m32 -fno-PIC -O2 -S nome.c**

```
.file      "nome.c"
.text
.section   .rodata.str1.1,"aMS",@progbits,1
.LC0:
.string    "argv[%2d]: %s\n"
.section   .text.startup,"ax",@progbits
.p2align 4
.globl     main
.type      main, @function
main:
.LFB23:
.cfi_startproc
endbr32
leal      4(%esp), %ecx
    Fazemos %ecx = %esp+4, ou seja, apontamos para argc, que é uma
    posição acima da RIP
.cfi_def_cfa 1, 0
andl      $-16, %esp
    Fazemos o alinhamento com 16
pushl     -4(%ecx)
    Estamos dando um push em %ecx-4, ou seja, guardamos RIP na pilha
pushl     %ebp
    Aqui salvamos a base anterior
```

```

.cfi_escape 0x10,0x5,0x2,0x75,0
movl    %esp, %ebp
    Com %ebp na pilha, agora criamos uma nova base
pushl    %esi
    Salva %esi, obrigação de main (MEM[%ebp-4])
pushl    %ebx
    Salva %ebx, obrigação de main (MEM[%ebp-8])
pushl    %ecx
    Salva %ecx, obrigação de main (MEM[%ebp-12])
.cfi_escape 0xf,0x3,0x75,0x74,0x6
.cfi_escape 0x10,0x6,0x2,0x75,0x7c
.cfi_escape 0x10,0x3,0x2,0x75,0x78
subl     $12, %esp
    Abre 3 espaços na pilha
movl     4(%ecx), %esi
    Vamos mover o conteúdo de %ecx+4 para %esi. Copiamos argv para
%esi
pushl     $.LC0
    Vamos colocar .LC0 no topo da pilha
pushl     $1
    Vamos colocar o valor absoluto 1 no topo da pilha
call      __printf_chk
    Vamos chamar a função _printt_chk
movl     (%esi,%ebx,4), %eax
    Vamos colocar %ebx * 4 + %esi em %eax. %esi = (arv)

addl     $16, %esp
    Iremos devolver o espaço a pilha
testl    %eax, %eax
    Vamos fazer um teste lógico entre os registradores %eax, nesse
momento estamos testando se o ponteiro para o argumento é válido.
jne       .L3
    Iremos realizar um desvio caso o teste anterior não seja igual
.L2:
leal     -12(%ebp), %esp
    Faz %esp apontar para onde %ecx foi salvo
xorl     %eax, %eax
    Fazemos xor de %eax com ele mesmo, zeramos %eax
popl     %ecx
    Restaura %ecx
.cfi_restore 1
.cfi_def_cfa 1, 0

```

```

popl    %ebx
    Restaure %ebx
.cfi_restore 3
popl    %esi
    Restaure %esi
.cfi_restore 6
popl    %ebp
    Restaure %ebp
.cfi_restore 5
leal    -4(%ecx), %esp
    Fazemos %esp apontar para o RIP na pilha
.cfi_def_cfa 4, 4
ret
    Retorna
.cfi_endproc
.LFE23:
.size    main, .-main
.ident   "GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0"
.section .note.GNU-stack,"",@progbits
.section .note.gnu.property,"a"
.align 4
.long    1f - 0f
.long    4f - 1f
.long    5
0:
.string   "GNU"
1:
.align 4
.long    0xc0000002
.long    3f - 2f
2:
.long    0x3
3:
.align 4
4:
pushl    $.LC0
pushl    $1
call     __printf_chk
movl     (%esi,%ebx,4), %eax
addl     $16, %esp
testl    %eax, %eax
jne      .L3

```

```

.L2:
    leal    -12(%ebp), %esp
    xorl    %eax, %eax
    popl    %ecx
    .cfi_restore 1
    .cfi_def_cfa 1, 0
    popl    %ebx
    .cfi_restore 3
    popl    %esi
    .cfi_restore 6
    popl    %ebp
    .cfi_restore 5
    leal    -4(%ecx), %esp
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc

.LFE23:
    .size   main, .-main
    .ident   "GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0"
    .section .note.GNU-stack,"",@progbits
    .section .note.gnu.property,"a"
    .align 4
    .long    1f - 0f
    .long    4f - 1f
    .long    5
0:
    .string   "GNU"
1:
    .align 4
    .long    0xc0000002
    .long    3f - 2f
2:
    .long    0x3
3:
    .align 4
4:

```

Compilando para **nome.o** e depois para executável **nome**, temos:

```

luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ gcc -m32 -c nome.s -o nome.o
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ gcc -m32 nome.o -o nome
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ ./nome
argv[ 0]: ./nome
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$

```

b) Rode nome com gdb. Crie um ponto de parada em main. Você terá que usar comandos do gdb para responder às perguntas abaixo. Cada item deverá ser comprovado com a captura da tela dos comandos executados com o gdb. Você pode gerar todos os comandos no gdb necessários para a resposta a todas as perguntas abaixo e apresentar uma única tela de captura. Em cada item, responda o que foi pedido, citando a captura feita.

Rodando nome com gdb:

```

luiz@Jarvis: ~/Área de Trabalho/comp prog/codigos/lista 5
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ gdb nome
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from nome...
(No debugging symbols found in nome)
(gdb) break main
Ponto de parada 1 at 0x1090
(gdb) run
Starting program: /home/luiz/Área de Trabalho/comp prog/codigos/lista 5/nome

Breakpoint 1, 0x56556090 in main ()
(gdb)

```

```

(gdb) disas
Dump of assembler code for function main:
=> 0x56556090 <+0>:    endbr32
0x56556094 <+4>:    lea     0x4(%esp),%ecx
0x56556098 <+8>:    and     $0xffffffff0,%esp
0x5655609b <+11>:   pushl   -0x4(%ecx)
0x5655609e <+14>:   push    %ebp
0x5655609f <+15>:   mov     %esp,%ebp
0x565560a1 <+17>:   push    %esi
0x565560a2 <+18>:   push    %ebx
0x565560a3 <+19>:   push    %ecx
0x565560a4 <+20>:   sub     $0xc,%esp
0x565560a7 <+23>:   mov     0x4(%ecx),%esi
0x565560aa <+26>:   mov     (%esi),%eax
0x565560ac <+28>:   test    %eax,%eax
0x565560ae <+30>:   je      0x565560d3 <main+67>
0x565560b0 <+32>:   xor     %ebx,%ebx
0x565560b2 <+34>:   lea     0x0(%esi),%esi
0x565560b8 <+40>:   push    %eax
0x565560b9 <+41>:   push    %ebx
0x565560ba <+42>:   add     $0x1,%ebx
0x565560bd <+45>:   push    $0x56557008
0x565560c2 <+50>:   push    $0x1
0x565560c4 <+52>:   call    0xf7edfea0 <__printf_chk>
0x565560c9 <+57>:   mov     (%esi,%ebx,4),%eax
0x565560cc <+60>:   add     $0x10,%esp
0x565560cf <+63>:   test    %eax,%eax
0x565560d1 <+65>:   jne     0x565560b8 <main+40>
0x565560d3 <+67>:   lea     -0xc(%ebp),%esp
0x565560d6 <+70>:   xor     %eax,%eax
0x565560d8 <+72>:   pop     %ecx
--Type <RET> for more, q to quit, c to continue without paging--

```

```
--Type <RET> for more, q to quit, c to continue without paging--
0x565560dc <+76>:    lea     -0x4(%ecx),%esp
0x565560df <+79>:    ret
```

End of assembler dump.

(gdb) break *0x565560a4

Ponto de parada 2 at 0x565560a4

(gdb) continue

Continuing.

Breakpoint 2, 0x565560a4 in main ()

(gdb) disas

Dump of assembler code for function main:

```
0x56556090 <+0>:    endbr32
0x56556094 <+4>:    lea     0x4(%esp),%ecx
0x56556098 <+8>:    and     $0xffffffff0,%esp
0x5655609b <+11>:   pushl   -0x4(%ecx)
0x5655609e <+14>:   push    %ebp
0x5655609f <+15>:   mov     %esp,%ebp
0x565560a1 <+17>:   push    %esi
0x565560a2 <+18>:   push    %ebx
0x565560a3 <+19>:   push    %ecx
=> 0x565560a4 <+20>:   sub     $0xc,%esp
0x565560a7 <+23>:   mov     0x4(%ecx),%esi
0x565560aa <+26>:   mov     (%esi),%eax
0x565560ac <+28>:   test    %eax,%eax
0x565560ae <+30>:   je      0x565560d3 <main+67>
0x565560b0 <+32>:   xor     %ebx,%ebx
0x565560b2 <+34>:   lea     0x0(%esi),%esi
0x565560b8 <+40>:   push    %eax
0x565560b9 <+41>:   push    %ebx
0x565560ba <+42>:   add     $0x1,%ebx
0x565560bd <+45>:   push    $0x56557008
0x565560c2 <+50>:   push    $0x1
0x565560c4 <+52>:   call    0xf7edfea0 <__printf_chk>
```

```
luiz@Jarvis: ~/Área de Trabalho/comp prog/codigos/lista 5
0x565560c4 <+52>: call 0xf7edfea0 <__printf_chk>
0x565560c9 <+57>: mov (%esi,%ebx,4),%eax
0x565560cc <+60>: add $0x10,%esp
0x565560cf <+63>: test %eax,%eax
0x565560d1 <+65>: jne 0x565560b8 <main+40>
0x565560d3 <+67>: lea -0xc(%ebp),%esp
0x565560d6 <+70>: xor %eax,%eax
0x565560d8 <+72>: pop %ecx
0x565560d9 <+73>: pop %ebx
0x565560da <+74>: pop %esi
0x565560db <+75>: pop %ebp
--Type <RET> for more, q to quit, c to continue without paging--
0x565560dc <+76>: lea -0x4(%ecx),%esp
0x565560df <+79>: ret
End of assembler dump.
(gdb) print /x $ecx
$1 = 0xffffd110
(gdb) x/4b 0xffffd110
0xffffd110: 0x01 0x00 0x00 0x00
(gdb) print /x *(int*) ($ecx+4)
$2 = 0xffffd1a4
(gdb) x/4b 0xffffd1a4
0xffffd1a4: 0x4e 0xd3 0xff 0xff
(gdb) x/60b 0xffffd34e
0xffffd34e: 0x2f 0x68 0x6f 0x6d 0x65 0x2f 0x6c 0x75
0xffffd35e: 0x69 0x7a 0x2f 0xc3 0x81 0x72 0x65 0x61
0xffffd36e: 0x20 0x64 0x65 0x20 0x54 0x72 0x61 0x62
0xffffd37e: 0x61 0x6c 0x68 0x6f 0x2f 0x63 0x6f 0x6d
0xffffd38e: 0x70 0x20 0x70 0x72 0x6f 0x67 0x2f 0x63
0xffffd39e: 0x6f 0x64 0x69 0x67 0x6f 0x73 0x2f 0x6c
0xffffd3ae: 0x69 0x73 0x74 0x61 0x20 0x35 0x2f 0x6e
0xffffd3be: 0x6f 0x6d 0x65 0x00
(gdb) print /x "/home/luiz/Área de Trabalho/comp prog/codigos/lista 5/nome"
$3 = {0x2f, 0x68, 0x6f, 0x6d, 0x65, 0x2f, 0x6c, 0x75, 0x69, 0x7a, 0x2f, 0xc3, 0x81, 0x72, 0x65, 0x61, 0x20,
0x64, 0x65, 0x20, 0x54, 0x72, 0x61, 0x62, 0x61, 0x6c, 0x68, 0x6f, 0x2f, 0x63, 0x6f, 0x6d, 0x70, 0x20, 0x70,
0x72, 0x6f, 0x67, 0x2f, 0x63, 0x6f, 0x64, 0x69, 0x67, 0x6f, 0x73, 0x2f, 0x6c, 0x69, 0x73, 0x74, 0x61, 0x20,
0x35, 0x2f, 0x6e, 0x6f, 0x6d, 0x65, 0x00}
(gdb)
```

b1) (5) Documente a sequência de passos usada no gdb, capturando desde o disparo inicial do gdb, run nome e os comandos efetuados no gdb para obter &argc, argc, &argv[0] e argv[0].

Primeiramente, colocamos um break na função main com o comando “break main”. Depois damos “run” no código como podemos ver aqui:

```
luiz@Jarvis: ~/Área de Trabalho/comp prog/codigos/lista 5
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ gdb nome
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from nome...
(No debugging symbols found in nome)
(gdb) break main
Ponto de parada 1 at 0x1090
(gdb) run
Starting program: /home/luiz/Área de Trabalho/comp prog/codigos/lista 5/nome

Breakpoint 1, 0x56556090 in main ()
(gdb)
```

Depois damos o comando “disas” como podemos ver aqui:

```
luiz@Jarvis: ~/Área de Trabalho/comp prog/codigos/lista 5
(gdb) disas
Dump of assembler code for function main:
=> 0x56556090 <+0>:      endbr32
    0x56556094 <+4>:      lea     0x4(%esp),%ecx
    0x56556098 <+8>:      and     $0xffffffff0,%esp
    0x5655609b <+11>:     pushl   -0x4(%ecx)
    0x5655609e <+14>:     push    %ebp
    0x5655609f <+15>:     mov     %esp,%ebp
    0x565560a1 <+17>:     push    %esi
    0x565560a2 <+18>:     push    %ebx
    0x565560a3 <+19>:     push    %ecx
    0x565560a4 <+20>:     sub     $0xc,%esp
    0x565560a7 <+23>:     mov     0x4(%ecx),%esi
    0x565560aa <+26>:     mov     (%esi),%eax
    0x565560ac <+28>:     test    %eax,%eax
    0x565560ae <+30>:     je      0x565560d3 <main+67>
    0x565560b0 <+32>:     xor     %ebx,%ebx
    0x565560b2 <+34>:     lea     0x0(%esi),%esi
    0x565560b8 <+40>:     push    %eax
    0x565560b9 <+41>:     push    %ebx
    0x565560ba <+42>:     add     $0x1,%ebx
    0x565560bd <+45>:     push    $0x56557008
    0x565560c2 <+50>:     push    $0x1
    0x565560c4 <+52>:     call    0xf7edfea0 <__printf_chk>
    0x565560c9 <+57>:     mov     (%esi,%ebx,4),%eax
    0x565560cc <+60>:     add     $0x10,%esp
    0x565560cf <+63>:     test    %eax,%eax
    0x565560d1 <+65>:     jne     0x565560b8 <main+40>
    0x565560d3 <+67>:     lea     -0xc(%ebp),%esp
    0x565560d6 <+70>:     xor     %eax,%eax
    0x565560d8 <+72>:     pop     %ecx
--Type <RET> for more, q to quit, c to continue without paging--
```

Em seguida, demos um break no ponto 0x565560a4 da função main com o comando “break *0x565560a4”:

```
--Type <RET> for more, q to quit, c to continue without paging--
    0x565560dc <+76>:     lea     -0x4(%ecx),%esp
    0x565560df <+79>:     ret
End of assembler dump.
(gdb) break *0x565560a4
Ponto de parada 2 at 0x565560a4
(gdb) continue
Continuing.

Breakpoint 2, 0x565560a4 in main ()
(gdb) disas
Dump of assembler code for function main:
    0x56556090 <+0>:      endbr32
    0x56556094 <+4>:      lea     0x4(%esp),%ecx
    0x56556098 <+8>:      and     $0xffffffff0,%esp
    0x5655609b <+11>:     pushl   -0x4(%ecx)
    0x5655609e <+14>:     push    %ebp
    0x5655609f <+15>:     mov     %esp,%ebp
    0x565560a1 <+17>:     push    %esi
    0x565560a2 <+18>:     push    %ebx
    0x565560a3 <+19>:     push    %ecx
=> 0x565560a4 <+20>:     sub     $0xc,%esp
    0x565560a7 <+23>:     mov     0x4(%ecx),%esi
    0x565560aa <+26>:     mov     (%esi),%eax
    0x565560ac <+28>:     test    %eax,%eax
    0x565560ae <+30>:     je      0x565560d3 <main+67>
    0x565560b0 <+32>:     xor     %ebx,%ebx
    0x565560b2 <+34>:     lea     0x0(%esi),%esi
    0x565560b8 <+40>:     push    %eax
    0x565560b9 <+41>:     push    %ebx
    0x565560ba <+42>:     add     $0x1,%ebx
    0x565560bd <+45>:     push    $0x56557008
    0x565560c2 <+50>:     push    $0x1
    0x565560c4 <+52>:     call    0xf7edfea0 <__printf_chk>
```


Em seguida, iremos passar o comando “print /x \$ecx” que irá imprimir o valor de argc, que é %ecx. Depois, vamos escrever o comando “x/4b 0xffffd110” e ele irá informar o conteúdo de memória. Assim, temos que “0xffffd110: 0x01 0x00 0x00 0x00” é o valor de argc. Depois, inserimos o comando “print /x *(int*) (\$ecx+4)” e printamos o valor de argv.

Inserimos o comando “x/4b 0xffffcfe4” e pegamos o valor de argv[0]. Podemos observar que é impresso 0xffffcfe4: 0xd1 0xd1 0xff 0xff. Em seguida, utilizamos o comando “x/60b 0xffffd34e” que lê o conteúdo de memória a partir de argv[0].

```

luiz@Jarvis: ~/Área de Trabalho/comp prog/codigos/lista 5
0x565560c4 <+52>: call 0xf7edfea0 <__printf_chk>
0x565560c9 <+57>: mov (%esi,%ebx,4),%eax
0x565560cc <+60>: add $0x10,%esp
0x565560cf <+63>: test %eax,%eax
0x565560d1 <+65>: jne 0x565560b8 <main+40>
0x565560d3 <+67>: lea -0xc(%ebp),%esp
0x565560d6 <+70>: xor %eax,%eax
0x565560d8 <+72>: pop %ecx
0x565560d9 <+73>: pop %ebx
0x565560da <+74>: pop %esi
0x565560db <+75>: pop %ebp
--Type <RET> for more, q to quit, c to continue without paging--
0x565560dc <+76>: lea -0x4(%ecx),%esp
0x565560df <+79>: ret
End of assembler dump.
(gdb) print /x $ecx
$1 = 0xffffd110
(gdb) x/4b 0xffffd110
0xffffd110: 0x01 0x00 0x00 0x00
(gdb) print /x *(int*) ($ecx+4)
$2 = 0xffffd1a4
(gdb) x/4b 0xffffd1a4
0xffffd1a4: 0x4e 0xd3 0xff 0xff
(gdb) x/60b 0xffffd34e
0xffffd34e: 0x2f 0x68 0x6f 0x6d 0x65 0x2f 0x6c 0x75
0xffffd35e: 0x69 0x7a 0x2f 0xc3 0x81 0x72 0x65 0x61
0xffffd36e: 0x20 0x64 0x65 0x20 0x54 0x72 0x61 0x62
0xffffd37e: 0x61 0x6c 0x68 0x6f 0x2f 0x63 0x6f 0x6d
0xffffd38e: 0x70 0x20 0x70 0x72 0x6f 0x67 0x2f 0x63
0xffffd39e: 0x6f 0x64 0x69 0x67 0x6f 0x73 0x2f 0x6c
0xffffd3ae: 0x69 0x73 0x74 0x61 0x20 0x35 0x2f 0x6e
0xffffd3be: 0x6f 0x6d 0x65 0x00
(gdb) print /x "basa/Área de Trabalho/comp prog/codigos/lista 5/comp"

```

Pela sequência executada com o gdb, temos que argc = 1, pois esse é o conteúdo apontado por %ecx. Como %ecx = 0xffffd110 = &argc, esse será o endereço de argc. Podemos observar que argc = 1, pois o arquivo foi executado sem argumentos.

Já argv está na posição de memória 0xffffd1a4. O valor desta posição de memória argv[0] = 0xffffd34e.

b2) (5) Liste em hexa o conteúdo de memória a partir do endereço argv[0], até encontrar byte NULL. Capture a tela do gdb.

Podemos observar que listamos 60 bytes da memória a partir do endereço 0xffffd34e, com ajuda do comando “x/60b”:

```

(gdb) x/60b 0xffffd34e
0xffffd34e: 0x2f 0x68 0x6f 0x6d 0x65 0x2f 0x6c 0x75
0xffffd35e: 0x69 0x7a 0x2f 0xc3 0x81 0x72 0x65 0x61
0xffffd36e: 0x20 0x64 0x65 0x20 0x54 0x72 0x61 0x62
0xffffd37e: 0x61 0x6c 0x68 0x6f 0x2f 0x63 0x6f 0x6d
0xffffd38e: 0x70 0x20 0x70 0x72 0x6f 0x67 0x2f 0x63
0xffffd39e: 0x6f 0x64 0x69 0x67 0x6f 0x73 0x2f 0x6c
0xffffd3ae: 0x69 0x73 0x74 0x61 0x20 0x35 0x2f 0x6e
0xffffd3be: 0x6f 0x6d 0x65 0x00
(gdb) print /x "/home/luiz/Área de Trabalho/comp prog/codigos/lista 5/nome"

```

b3) (5) Identifique a sequência de caracteres ASCII que corresponde ao conteúdo de memória, usando o comando print /x do gdb para criar a tradução em hexa e comprovar que é igual ao valor em memória. Capture a tela do gdb.

Utilizando o comando “print /x” do gdb, podemos observar que os dois conteúdos são iguais, tanto o em memória quanto o ASCII correspondente ao caminho completo do executável.

```

(gdb) x/60b 0xffffd34e
0xffffd34e: 0x2f 0x68 0x6f 0x6d 0x65 0x2f 0x6c 0x75
0xffffd35e: 0x69 0x7a 0x2f 0xc3 0x81 0x72 0x65 0x61
0xffffd36e: 0x20 0x64 0x65 0x20 0x54 0x72 0x61 0x62
0xffffd37e: 0x61 0x6c 0x68 0x6f 0x2f 0x63 0x6f 0x6d
0xffffd38e: 0x70 0x20 0x70 0x72 0x6f 0x67 0x2f 0x63
0xffffd39e: 0x6f 0x64 0x69 0x67 0x6f 0x73 0x2f 0x6c
0xffffd3ae: 0x69 0x73 0x74 0x61 0x20 0x35 0x2f 0x6e
0xffffd3be: 0x6f 0x6d 0x65 0x00
(gdb) print /x "/home/luiz/Área de Trabalho/comp prog/codigos/lista 5/nome"
$3 = {0x2f, 0x68, 0x6f, 0x6d, 0x65, 0x2f, 0x6c, 0x75, 0x69, 0x7a, 0x2f, 0xc3, 0x81, 0x72, 0x65, 0x61, 0x20,
0x64, 0x65, 0x20, 0x54, 0x72, 0x61, 0x62, 0x61, 0x6c, 0x68, 0x6f, 0x2f, 0x63, 0x6f, 0x6d, 0x70, 0x20, 0x70,
0x72, 0x6f, 0x67, 0x2f, 0x63, 0x6f, 0x64, 0x69, 0x67, 0x6f, 0x73, 0x2f, 0x6c, 0x69, 0x73, 0x74, 0x61, 0x20,
0x35, 0x2f, 0x6e, 0x6f, 0x6d, 0x65, 0x0}
(gdb)

```

c) (10) Altere um único valor em nome.s (liste e mostre o que foi alterado) e gere nomem.s de modo a imprimir todas as variáveis de ambiente. Gere o executável nomem a partir de nomem.s, rode e imprima a tela, comprovando o sucesso da modificação.

Nesse exercício vamos listar as variáveis de ambiente, ao invés dos argumentos. Para isso, vamos mudar o código de montagem de nome.s e substituir argv por envp, visto que envp aponta para o vetor de ponteiros das variáveis de ambiente. Assim, vamos modificar a linha “movl 4(%ecx), %esi” por “movl 8(%ecx), %esi”, ou seja, estamos copiando envp para %esi.

Depois geramos um executável nomem a partir de nomem.s:

```

luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ gcc -m32 -o nomem nomem.s
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ objdump -d nomem

```

```
luiz@Jarvis: ~/Área de Trabalho/comp prog/codigos/lista 5
uesmontagem da seção .text:
00001090 <main>:
1090:      f3 0f 1e fb          endbr32
1094:      8d 4c 24 04          lea     0x4(%esp),%ecx
1098:      83 e4 f0             and     $0xffffffff0,%esp
109b:      ff 71 fc             pushl  -0x4(%ecx)
109e:      55                   push   %ebp
109f:      89 e5               mov     %esp,%ebp
10a1:      56                   push   %esi
10a2:      53                   push   %ebx
10a3:      51                   push   %ecx
10a4:      83 ec 0c             sub     $0xc,%esp
10a7:      8b 71 08             mov     0x8(%ecx),%esi
10aa:      8b 06               mov     (%esi),%eax
10ac:      85 c0               test    %eax,%eax
10ae:      74 23               je      10d3 <main+0x43>
10b0:      31 db               xor     %ebx,%ebx
10b2:      8d b6 00 00 00 00      lea     0x0(%esi),%esi
10b8:      50                   push   %eax
10b9:      53                   push   %ebx
10ba:      83 c3 01             add     $0x1,%ebx
10bd:      68 08 20 00 00        push    $0x2008
10c2:      6a 01               push    $0x1
10c4:      e8 fc ff ff ff        call    10c5 <main+0x35>
10c9:      8b 04 9e             mov     (%esi,%ebx,4),%eax
10cc:      83 c4 10             add     $0x10,%esp
10cf:      85 c0               test    %eax,%eax
10d1:      75 e5               jne     10b8 <main+0x28>
10d3:      8d 65 f4             lea     -0xc(%ebp),%esp
10d6:      31 c0               xor     %eax,%eax
10d8:      59                   pop     %ecx
10d9:      5b                   pop     %ebx
10da:      5e                   pop     %esi
10db:      5d                   pop     %ebp
10dc:      8d 61 fc             lea     -0x4(%ecx),%esp
10df:      c3                   ret
```

E obtemos a seguinte tela de saída:

```
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ ./nomem
argv[ 0]: SHELL=/bin/bash
argv[ 1]: SESSION_MANAGER=local/Jarvis:@/tmp/.ICE-unix/1984,unix/Jarvis:/tmp/.ICE-unix/1984
argv[ 2]: QT_ACCESSIBILITY=1
argv[ 3]: COLORTERM=truecolor
argv[ 4]: XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
argv[ 5]: XDG_MENU_PREFIX=gnome-
argv[ 6]: GNOME_DESKTOP_SESSION_ID=this-is-deprecated
argv[ 7]: LANGUAGE=pt_BR:pt:en
argv[ 8]: GNOME_SHELL_SESSION_MODE=ubuntu
argv[ 9]: SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
argv[10]: XMODIFIERS=@im=ibus
argv[11]: DESKTOP_SESSION=ubuntu
argv[12]: SSH_AGENT_PID=1918
argv[13]: GTK_MODULES=gail:atk-bridge
argv[14]: DBUS_STARTER_BUS_TYPE=session
argv[15]: PWD=/home/luiz/Área de Trabalho/comp prog/codigos/lista 5
argv[16]: LOGNAME=luiz
argv[17]: XDG_SESSION_DESKTOP=ubuntu
argv[18]: XDG_SESSION_TYPE=x11
argv[19]: GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
argv[20]: XAUTHORITY=/run/user/1000/gdm/Xauthority
argv[21]: WINDOWPATH=2
argv[22]: HOME=/home/luiz
argv[23]: USERNAME=luiz
argv[24]: IM_CONFIG_PHASE=1
argv[25]: LANG=pt_BR.UTF-8
argv[26]: LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:
su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc=01;31:*.arj=01;31:*.taz=0
1;31:*.lha=01;31:*.lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31:*.z=
01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31:*.zst=01;31:*.tzt=01;31:*.bz2=01;31:*.bz=01
;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=
01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.cab=01;31:*.wim=01;31:*.swm=01;31:*.dwm
=01;31:*.esd=01;31:*.jpg=01;35:*.jpeg=01;35:*.mjpg=01;35:*.mjpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35
:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;35:*.mng=0
1;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;35:*.mkv=01;35:*.webm=01;35:*.ogm=01;35:*.mp4=01;35:*.m
4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.
avi=01;35:*.fli=01;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*
```

O que temos nos prints é:

```

argv[ 0]: SHELL=/bin/bash
argv[ 1]:
SESSION_MANAGER=local/Jarvis:@/tmp/.ICE-unix/1984,unix/Jarvis:/tmp/.ICE-unix/1984
argv[ 2]: QT_ACCESSIBILITY=1
argv[ 3]: COLORTERM=truecolor
argv[ 4]: XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
argv[ 5]: XDG_MENU_PREFIX=gnome-
argv[ 6]: GNOME_DESKTOP_SESSION_ID=this-is-deprecated
argv[ 7]: LANGUAGE=pt_BR:pt:en
argv[ 8]: GNOME_SHELL_SESSION_MODE=ubuntu
argv[ 9]: SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
argv[10]: XMODIFIERS=@im=ibus
argv[11]: DESKTOP_SESSION=ubuntu
argv[12]: SSH_AGENT_PID=1918
argv[13]: GTK_MODULES=gail:atk-bridge
argv[14]: DBUS_STARTER_BUS_TYPE=session
argv[15]: PWD=/home/luiz/Área de Trabalho/comp prog/codigos/lista 5
argv[16]: LOGNAME=luiz
argv[17]: XDG_SESSION_DESKTOP=ubuntu
argv[18]: XDG_SESSION_TYPE=x11
argv[19]: GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
argv[20]: XAUTHORITY=/run/user/1000/gdm/Xauthority
argv[21]: WINDOWPATH=2
argv[22]: HOME=/home/luiz
argv[23]: USERNAME=luiz
argv[24]: IM_CONFIG_PHASE=1
argv[25]: LANG=pt_BR.UTF-8
argv[26]:
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc=01;31:*.arj=01;31:*.taz=01;31:*.lha=01;31:*.lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31:*.z=01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31:*.zst=01;31:*.tzst=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.cab=01;31:*.wim=01;31:*.swm=01;31:*.dwm=01;31:*.esd=01;31:*.jpg=01;35:*.jpeg=01;35:*.mjpg=01;35:*.mjpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;35:*.mkv=01;35:*.webm=01;35:*.ogm=01;35:*.

```

```

mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=
01;35:*.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv=0
1;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=
01;35:*.ogv=01;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.m4a=00;36:*.
mid=00;36:*.midi=00;36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=0
0;36:*.wav=00;36:*.oga=00;36:*.opus=00;36:*.spx=00;36:*.xspf=00;36:
argv[27]: XDG_CURRENT_DESKTOP=ubuntu:GNOME
argv[28]: VTE_VERSION=6003
argv[29]:
GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/screen/cc176e9b_88fb_44d6_
8bf4_6652dd4298a1
argv[30]: INVOCATION_ID=7590f23df1fd46f19ed35bbfabb79f36
argv[31]: MANAGERPID=1723
argv[32]: LESSCLOSE=/usr/bin/lesspipe %s %s
argv[33]: XDG_SESSION_CLASS=user
argv[34]: TERM=xterm-256color
argv[35]: LESSOPEN=| /usr/bin/lesspipe %s
argv[36]: USER=luiz
argv[37]: GNOME_TERMINAL_SERVICE=:1.390
argv[38]: DISPLAY=:0
argv[39]: SHLVL=1
argv[40]: QT_IM_MODULE=ibus
argv[41]:
DBUS_STARTER_ADDRESS=unix:path=/run/user/1000/bus,guid=328ff893a802838
be7e344c3614b367d
argv[42]: XDG_RUNTIME_DIR=/run/user/1000
argv[43]: JOURNAL_STREAM=8:51755
argv[44]:
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/local/share/:/usr/share/:/var/lib/snapd/desk
top
argv[45]:
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/g
ames:/snap/bin
argv[46]: GDMSESSION=ubuntu
argv[47]:
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus,guid=328ff893a8
02838be7e344c3614b367d
argv[48]: _=./nomem

```

d) (5) Rode nomem com gdb, indique o valor do endereço apontado por envp[0] e liste o endereço de memória a partir deste endereço e até encontrar o byte NULL.

Rodando nomem com o gdb e colocando um break na função main temos:

```
luiz@Jarvis: ~/Área de Trabalho/comp prog/codigos/lista 5
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ gdb nomem
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from nomem...
(No debugging symbols found in nomem)
(gdb) break main
Ponto de parada 1 at 0x1090
(gdb) run
Starting program: /home/luiz/Área de Trabalho/comp prog/codigos/lista 5/nomem

Breakpoint 1, 0x56556090 in main ()
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) n
Program not restarted.
(gdb) disas
```

Podemos observar que demos o comando “run” e em seguida imprimimos o código de montagem com o comando “disas”.

```
luiz@Jarvis: ~/Área de Trabalho/comp prog/codigos/lista 5
(gdb) break *0x565560aa
Ponto de parada 2 at 0x565560aa
(gdb) continue
Continuing.

Breakpoint 2, 0x565560aa in main ()
(gdb) disas
Dump of assembler code for function main:
   0x56556090 <+0>:    endbr32
   0x56556094 <+4>:    lea     0x4(%esp),%ecx
   0x56556098 <+8>:    and     $0xffffffff0,%esp
   0x5655609b <+11>:   pushl   -0x4(%ecx)
   0x5655609e <+14>:   push    %ebp
   0x5655609f <+15>:   mov     %esp,%ebp
   0x565560a1 <+17>:   push    %esi
   0x565560a2 <+18>:   push    %ebx
   0x565560a3 <+19>:   push    %ecx
   0x565560a4 <+20>:   sub     $0xc,%esp
   0x565560a7 <+23>:   mov     0x8(%ecx),%esi
=> 0x565560aa <+26>:   mov     (%esi),%eax
   0x565560ac <+28>:   test    %eax,%eax
   0x565560ae <+30>:   je      0x565560d3 <main+67>
   0x565560b0 <+32>:   xor     %ebx,%ebx
   0x565560b2 <+34>:   lea     0x0(%esi),%esi
   0x565560b8 <+40>:   push    %eax
   0x565560b9 <+41>:   push    %ebx
   0x565560ba <+42>:   add     $0x1,%ebx
   0x565560bd <+45>:   push    $0x56557008
   0x565560c2 <+50>:   push    $0x1
   0x565560c4 <+52>:   call    0xf7edfea0 <__printf_chk>
   0x565560c9 <+57>:   mov     (%esi,%ebx,4),%eax
   0x565560cc <+60>:   add     $0x10,%esp
   0x565560cf <+63>:   test    %eax,%eax
   0x565560d1 <+65>:   jne     0x565560b8 <main+40>
   0x565560d3 <+67>:   lea     -0xc(%ebp),%esp
   0x565560d6 <+70>:   xor     %eax,%eax
   0x565560d8 <+72>:   pop     %ecx
   0x565560da <+73>:   pop     %ebx
```

Com o comando “print /x *(int*) (\$ecx+8)”, podemos observar que o valor de envp = MEM[%ecx+8] e que &envp[0] = 0xffffd1ac. E com o comando “x/4b 0xffffd1ac”, podemos descobrir envp[0], onde, 0xffffd1ac: 0x89 0xd3 0xff 0xff é &envp[0], o end inicial do ambiente.

Em seguida, pintamos a lista de conteúdo de memória com ajuda do “x/60b 0xffffd389”. Aqui podemos observar a lista de endereços de memória a partir de 0xffffd389 até o byte NULL.

```

luiz@Jarvis: ~/Área de Trabalho/comp prog/codigos/lista 5
0x565560b2 <+34>: lea    0x0(%esi),%esi
0x565560b8 <+40>: push  %eax
0x565560b9 <+41>: push  %ebx
0x565560ba <+42>: add   $0x1,%ebx
0x565560bd <+45>: push  $0x56557008
0x565560c2 <+50>: push  $0x1
0x565560c4 <+52>: call  0xf7edfea0 <__printf_chk>
0x565560c9 <+57>: mov   (%esi,%ebx,4),%eax
0x565560cc <+60>: add   $0x10,%esp
0x565560cf <+63>: test  %eax,%eax
0x565560d1 <+65>: jne   0x565560b8 <main+40>
0x565560d3 <+67>: lea   -0xc(%ebp),%esp
0x565560d6 <+70>: xor   %eax,%eax
0x565560d8 <+72>: pop   %ecx
0x565560d9 <+73>: pop   %ebx
0x565560da <+74>: pop   %esi
0x565560db <+75>: pop   %ebp
0x565560dc <+76>: lea   -0x4(%ecx),%esp
0x565560df <+79>: ret
End of assembler dump.
(gdb) print /x *(int*) ($ecx+8)
$1 = 0xffffd1ac
(gdb) x/4b 0xffffd1ac
0xffffd1ac: 0x89 0xd3 0xff 0xff
(gdb) x/60b 0xffffd89
0xffffd89: Não é possível acessar a memória no endereço 0xffffd89
(gdb) x/60b 0xffffd389
0xffffd389: 0x53 0x48 0x45 0x4c 0x4c 0x3d 0x2f 0x62
0xffffd391: 0x69 0x6e 0x2f 0x62 0x61 0x73 0x68 0x00
0xffffd399: 0x53 0x45 0x53 0x53 0x49 0x4f 0x4e 0x5f
0xffffd3a1: 0x4d 0x41 0x4e 0x41 0x47 0x45 0x52 0x3d
0xffffd3a9: 0x6c 0x6f 0x63 0x61 0x6c 0x2f 0x4a 0x61
0xffffd3b1: 0x72 0x76 0x69 0x73 0x3a 0x40 0x2f 0x74
0xffffd3b9: 0x6d 0x70 0x2f 0x2e 0x49 0x43 0x45 0x2d
0xffffd3c1: 0x75 0x6e 0x69 0x78

```

e) (5) Identifique a sequência de caracteres ASCII que corresponde ao conteúdo desta lista de caracteres (string), usando o comando print /x do gdb para criar a tradução em hexa e comprovar que é igual ao valor em memória. Capture a tela do gdb.

Podemos observar que a lista ASCII foi impressa utilizando o comando print /x do gdb que corresponde ao caminho do executável é igual ao conteúdo de memória:

```

(gdb) x/4b 0xffffd1ac
0xffffd1ac: 0x89 0xd3 0xff 0xff
(gdb) x/60b 0xffffd89
0xffffd89: Não é possível acessar a memória no endereço 0xffffd89
(gdb) x/60b 0xffffd389
0xffffd389: 0x53 0x48 0x45 0x4c 0x4c 0x3d 0x2f 0x62
0xffffd391: 0x69 0x6e 0x2f 0x62 0x61 0x73 0x68 0x00
0xffffd399: 0x53 0x45 0x53 0x53 0x49 0x4f 0x4e 0x5f
0xffffd3a1: 0x4d 0x41 0x4e 0x41 0x47 0x45 0x52 0x3d
0xffffd3a9: 0x6c 0x6f 0x63 0x61 0x6c 0x2f 0x4a 0x61
0xffffd3b1: 0x72 0x76 0x69 0x73 0x3a 0x40 0x2f 0x74
0xffffd3b9: 0x6d 0x70 0x2f 0x2e 0x49 0x43 0x45 0x2d
0xffffd3c1: 0x75 0x6e 0x69 0x78
(gdb) print /x "SHELL=/bin/bash"
$2 = {0x53, 0x48, 0x45, 0x4c, 0x4c, 0x3d, 0x2f, 0x62, 0x69, 0x6e, 0x2f, 0x62, 0x61, 0x73, 0x68, 0x00}
(gdb)

```


Questão 2) Compile o programa abaixo e gere um executável qfork.

```
#include <sys/wait.h>
#include <sys/types.h>
#include <stdlib.h>
#include <stdio.h>
int main () {
    int pid ;
    int status;
    pid = fork () ;

    if ( pid < 0 ) { perror ("Erro: "); exit (1) ;}
    else if ( pid > 0 ) {

        printf ("\nsou o processo pai, com pid = %d\n", getpid()) ;
        pid = waitpid(pid, &status, 0);
        printf ("pid do processo que liberou o pai = %d\n", pid);
        if (WIFEXITED(status)) printf ("término do filho = %d \n",
WEXITSTATUS (status));
        if (WIFSTOPPED(status)) printf ("filho parado= %d \n", WSTOPSIG
(status));
        if (WIFSIGNALED(status)) printf ("sinal que causou término do
filho = %d \n",
WTERMSIG (status));
        printf ("Tchau do pai\n");
        exit(0);
    }

    else { // processo filho
        printf ("sou o processo filho, com pid = %d\n", getpid()) ;
        sleep(10);
        printf("Tchau do filho\n");
        exit(2);}}
```

a) (5) Explique o que o programa faz, considerando todas as possibilidades de quem roda primeiro levando em conta as interações entre os processos.

Logo após a execução do fork, qualquer um dos processos, pai ou filho, podem rodar primeiro. Caso o processo pai venha a rodar primeiro, ele irá printar “sou o processo pai, com pid = [PID]” e irá ficar em espera bloqueante pelo término do processo filho.

Agora, caso o processo filho venha rodar primeiro, ele irá printar “sou o processo filho, com pid = [PID]” e irá dormir por 10 segundos. Nesse momento, quando o filho está dormindo, o pai consegue rodar. Como o filho rodou primeiro, o pai ficará em

espera bloqueante até o término do processo filho. Quando o filho acorda, ele imprime “Tchau do filho” e termina com status 200.

Depois o pai sairá da espera bloqueante ao receber o SIGCHLD e irá printar “término do filho = 200” e logo em seguida irá imprimir “Tchau do pai” e terminar o programa.

Executando o programa temos as linhas:

```
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ ./qfork
sou o processo pai, com pid = 83812
sou o processo filho, com pid = 83813
Tchau do filho
pid do processo que liberou o pai = 83813
término do filho = 200
Tchau do pai
```

b) (5) Você irá rodar este programa em background. Descubra como você pode, a partir da shell, parar o processo filho e depois fazê-lo continuar. Quais sinais são enviados e como são enviados? Imprime uma tela que mostra sua interação com o programa e os comandos executados na shell.

Primeiramente vamos rodar o programa e descobrir o pid de cada processo. Podemos observar que o pid do pai é 84235 e o pid do filho é 84236. Depois disso, vamos utilizar o comando “ps -l” para observar qual programa está parado e qual está em wait. Podemos observar que o processo filho está parado enquanto o processo pai está em wait.

Agora vamos fazer o processo filho parar executando o comando “kill -19 84236”. Após essa execução será printado as seguintes mensagens:

```
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ ./qfork&
[1] 84235
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$
sou o processo pai, com pid = 84235
sou o processo filho, com pid = 84236
kill -19 84236
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ ps -l
F S  UID      PID     PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    84214    84207  0  80   0 -  2895 do_wai pts/0        00:00:00 bash
0 S  1000    84235    84214  0  80   0 -   624 do_wai pts/0        00:00:00 qfork
1 T  1000    84236    84235  0  80   0 -   624 do_sig pts/0        00:00:00 qfork
4 R  1000    84237    84214  0  80   0 -  2879 -      pts/0        00:00:00 ps
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ kill -18 84236
Tchau do filho
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ pid do processo que liberou o pai = 84236
término do filho = 200
Tchau do pai
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$
```

Sabendo que o processo pai roda primeiro do que o processo filho, antes de esgotar o tempo de parada do filho, o SIGCONT é enviado junto com o comando kill e

observamos que o programa filho se encerra. Depois, o sinal SIGCHLD é enviado ao processo pai para resgatar o processo filho que já tinha encerrado. Dessa forma, o processo pai sai da espera e continua a execução até o seu término.

c) (5) Rode o programa em background e pare o processo filho. Agora envie um SIGINT para o filho. O quê acontece? O sinal SIGINT é entregue ao processo filho? Explique.

Podemos observar que ao enviar um sinal SIGINT a um processo filho que se encontra parado, nada acontece, pois o sinal fica pendente de entrega ao processo filho. Isso acontece, pois o sinal SIGINT não é entregue a processos que se encontram parados.

Podemos observar isso acontecendo no print abaixo quando fazemos o processo filho parar executando o comando “kill -19 84301”:

A terminal window titled 'luiz@Jarvis: ~/Área de Trabalho/comp prog/codigos/lista 5'. The user runs './qfork&' and the terminal shows '[1] 84300'. Then, the user enters 'kill -19 84301'. The terminal output shows 'sou o processo pai, com pid = 84300' and 'sou o processo filho, com pid = 84301'. The user then runs 'kill -2 84301' twice, which results in no visible output, indicating the signal is not delivered to the stopped process.

```
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ ./qfork&
[1] 84300
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$
sou o processo pai, com pid = 84300
sou o processo filho, com pid = 84301
kill -19 84301
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ kill -2 84301
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ kill -2 84301
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$
```

d) (5) Faça o filho continuar e verifique a saída do programa. Conclua sobre como ocorre a entrega de sinais SIGINT, SIGKILL e SIGSTOP a processos parados.

Após enviarmos o sinal SIGCONT (18) ao processo filho, podemos observar que o processo continuará, porém, o SO irá entregar o sinal SIGINT(2) ao processo. Isso vai forçar o término do processo filho. Sendo assim, não teremos um término normal do processo.

Podemos observar que o sinal SIGCONT fica pendente de entrega para o processo até que ele saia do estado parado. Os processos que se encontram parados, ficam com sinal pendente de entrega, incluindo o sinal SIGCONT. As únicas exceções são os sinais SIGKILL e SIGSTOP que são entregues imediatamente aos processos.

Vamos parar o processo filho e enviar um sinal SIGKILL para analisar o que acontece. Podemos observar que o estado de término do filho se dá devido ao sinal SIGKILL enviado.

```
luiz@Jarvis: ~/Área de Trabalho/comp prog/codigos/lista 5
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ ps -l
F S  UID      PID     PPID  C  PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S   1000    84294    84272  0   80   0  -  2722 do_wai pts/1        00:00:00 bash
0 S   1000    84300    84294  0   80   0  -   624 do_wai pts/1        00:00:00 qfork
1 T   1000    84301    84300  0   80   0  -   624 do_sig pts/1        00:00:00 qfork
4 R   1000    84356    84294  0   80   0  -  2879 -      pts/1        00:00:00 ps
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ kill -18 84301
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ pid do processo que liberou o pai = 84301
sinal que causou término do filho = 2
Tchau do pai
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ ps -l
F S  UID      PID     PPID  C  PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S   1000    84294    84272  0   80   0  -  2722 do_wai pts/1        00:00:00 bash
4 R   1000    84357    84294  0   80   0  -  2879 -      pts/1        00:00:00 ps
[1]+  Concluido      ./qfork
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$
```

e) (5) Modifique agora qfork.c para que o pai consulte o término do filho, mas de forma não bloqueante. Qual modificação foi feita? Rode algumas vezes o programa e analise o que é impresso pelo pai. Você diria que está correta e coerente a saída impressa pelo programa? Explique. Caso haja inconsistência, que modificações você faria para termos uma saída coerente? Explique sua modificação, se houver.

Agora modificamos o programa qfork.c para que o pai consulte o término do filho, de forma não bloqueante. Para isso, alteramos a seguinte linha:

```
pid = waitpid(pid, &status, 0);
```

Trocamos o '0' por '1' e ficamos com a seguinte linha:

```
pid = waitpid(pid, &status, 1);
```

Em seguida rodamos o código e obtemos:

```
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ ./qfork
sou o processo pai, com pid = 84450
pid do processo que liberou o pai = 0
sinal que causou término do filho = 64
Tchau do pai
sou o processo filho, com pid = 84451
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ Tchau do filho
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ ./qfork
sou o processo filho, com pid = 84458
sou o processo pai, com pid = 84457
pid do processo que liberou o pai = 0
sinal que causou término do filho = 80
Tchau do pai
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ Tchau do filho
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ ./qfork
sou o processo pai, com pid = 84464
pid do processo que liberou o pai = 0
sinal que causou término do filho = 112
Tchau do pai
sou o processo filho, com pid = 84465
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$ Tchau do filho
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 5$
```

Podemos observar que a seguinte linha:

```
if (WIFSIGNALED(status)) printf ("sinal que causou término do filho=%d\n", WTERMSIG (status));
```

É responsável pela impressão que possuímos sobre o sinal que causa o término do processo filho. O que é bem estranho, pois estamos tratando de uma chamada não bloqueante.

O processo filho estava dormindo quando o processo pai executou o `waitpid` não bloqueante. A impressão com valor de sinal diferente a cada rodada está errada, pois não tivemos nenhum sinal que tenha causado o término do processo filho.

Como a chamada não bloqueante retorna 0 quando ainda não temos o término de um filho. Neste caso, o teste para impressão do status só deveria ter sido feito quando o pid fosse diferente de 0. Para arrumar isso, podemos alterar a linha anterior da seguinte forma:

```
if (WIFSIGNALED(status) && pid) printf ("sinal que causou término do filho=%d\n", WTERMSIG (status));
```

Essa correção evita a impressão errada do programa.

f) (5) Modifique agora `qfork.c` para que o pai fique em espera pelo término ou parada do filho. Rode o programa em background e veja o que é impresso. Agora, após iniciar a execução, pare o processo filho e documente a saída do programa. Imediatamente ao término do pai, verifique se o programa filho ainda está no sistema. Comente e verifique se há processo zumbi no sistema.

Questão 3)

Para o programa `qfork.c` da questão anterior, foi gerado o código de montagem com opção `-m32 -fno-PIC -O2 -S`. Algumas linhas não essenciais foram suprimidas, incluindo diretivas para carga do código. Identifique no código as instruções que identificam qual condição garante que as macros `WIFEXITED(status)`, `WIFSTOPPED(status)` e `WIFTERM(status)` são TRUE, além de que bits ou bytes são retornados após as macros. `WEXITSTATUS(status)`, `WSTOPSIG(status)` e `WTERMSIG(status)`, respectivamente. Somente explique as passagens necessárias para justificar sua resposta. O código de montagem será bem longo, mas foque apenas nas linhas que implementam as macros. Apenas estas interessam.

Você deve determinar exatamente as condições dos bytes na variável `status` que identificam cada uma das três possíveis situações: término normal via `exit/return`, filho parado por causa de sinal e filho terminado por causa de sinal. Além disso, você tem que apontar os bits ou bytes de `status` que identificam as causas de

término ou os sinais envolvidos. Identifique pelo código de montagem os intervalos válidos para término normal de um processo e os valores possíveis para os sinais que causaram parada ou término do processo. Os valores retornados no código C são apenas exemplos. O que se quer são os valores possíveis que podem ser escolhidos ou identificados pelo SO em virtude do que é de fato analisado pelo código de montagem ao processar as macros correspondentes.

```
.file "qfork.c"
.LC0:
.string "Erro: "
.LC1:
.string "\nsou o processo pai, com pid = %d\n"
.LC2:
.string "pid do processo que liberou o pai = %d\n"
.LC3:
.string "t\303\251rmino do filho = %d \n"
.LC4:
.string "filho parado= %d \n"
.LC5:
.string "sinal que causou t\303\251rmino do filho = %d \n"
.LC6:
.string "Tchau do pai"
.LC7:
.string "sou o processo filho, com pid = %d\n"
.LC8:
.string "Tchau do filho"
main:
1 endbr32
2 leal 4(%esp), %ecx
3 andl $-16, %esp
4 pushl -4(%ecx)
5 pushl %ebp
6 movl %esp, %ebp
7 pushl %ebx
8 pushl %ecx          //Vamos salvar %ecx na pilha que será usado no
retorno
9 subl $16, %esp
10 movl %gs:20, %eax
11 movl %eax, -12(%ebp)
12 xorl %eax, %eax
13 call fork
14 testl %eax, %eax
15 js .L10
```

```

16 je .L3      // é feito um desvio caso pid == 0, que é o nosso
processo filho
17 movl %eax, %ebx    // pid do filho será salvo em %ebx
18 call getpid        // o programa pega o pid do processo
19 pushl %ecx
20 pushl %eax         // pid do nosso processo
21 pushl $.LC1
22 pushl $1           // parâmetro da função printf
23 call __printf_chk  // damos um print no pid do pai
24 addl $12, %esp
25 leal -16(%ebp), %eax    // calculamos &status
26 pushl $2
27 pushl %eax         //&status
28 pushl %ebx         // pid
29 call waitpid        //chamada de waitpid
30 addl $12, %esp
31 pushl %eax         // o pid é colocado na pilha
32 pushl $.LC2        // end da lista de impressão
33 pushl $1           // parâmetro da função printf
34 call __printf_chk  //chamada da função printf
35 movl -16(%ebp), %eax    // pega o status
36 addl $16, %esp      //devolve 4 espaços de 4 bytes a pilha

37 testb $127, %al      // máscara para os bits menos
significativos de status
38 je .L11             //Faz um desvio se todos os bytes forem
iguais a zero. Aqui temos o exit/return, ou seja, o término normal do
programa.
.L4:
39 cmpb $127, %al       //Testa se os bits menos significativos são
iguais a 1
40 je .L12             //Realiza um desvio se todos os bits forem
iguais a 1. Nesse momento, temos a parada de do filho
.L5: //
41 movl %eax, %edx       //Aqui pegamos o status
42 andl $127, %edx       // Realizamos uma mascara and com tudo em 1,
com os bits menos significativos do status.
43 addl $1, %edx         //Se for 0xEF vamos obter 0x40
44 subb $1, %dl          //Realiza uma subtração de 1 para que o
valor seja preservado
45 jle .L6              //Realiza um salto se for menor ou igual.Se
essa condição for satisfeita teremos o término do sinal.

```

```

46 andl $127, %eax
47 pushl %ecx          // salva %ecx na pilha
48 pushl %eax          // salva a variável de status impressa
devido ao término
49 pushl $.LC5
50 pushl $1
51 call __printf_chk    // chama a função printf para imprimir o
valor do sinal que terminou o filho
52 addl $16, %esp
.L6:
53 subl $12, %esp
54 pushl $.LC6
55 call puts
56 movl $0, (%esp)      // faz o retorno de 0 do pai
57 call exit            // exit (0) no código c
.L3:                   // caso seja filho faz o desvio por aqui
58 call getpid
59 pushl %edx
60 pushl %eax
61 pushl $.LC7
62 pushl $1
63 call __printf_chk
64 movl $10, (%esp)
65 call sleep
66 movl $.LC8, (%esp)
67 call puts
68 movl $2, (%esp)
69 call exit
.L10:
70 subl $12, %esp
71 pushl $.LC0
72 call perror
73 movl $1, (%esp)
74 call exit
.L11:                  // Caso o término seja normal com exit ou return
75 movzbl %ah, %eax     // Segundo byte de status e o considera como um
inteiro
76 pushl %edx
77 pushl %eax
78 pushl $.LC3
79 pushl $1
80 call __printf_chk    // chama printf para imprimir o valor de término
do filho

```

```

81 movl -16(%ebp), %eax    //pega o status
82 addl $16, %esp
83 jmp .L4                // faz o desvio para o próximo teste
.L12:                    // Para situação de parada do filho
84 movzbl %ah, %eax        // pega o segundo byte de status e o trata
em magnitude
85 pushl %ebx
86 pushl %eax              // sinal que causa a parada do filho (está
em %ah
87 pushl $.LC4
88 pushl $1
89 call __printf_chk      // chama a função print para imprimir o
sinal que causou a parada do filho
90 movl -16(%ebp), %eax    // passa para %eax o status
91 addl $16, %esp
92 jmp .L5                // faz o desvio para trabalhar no caso do
sinal pego pelo filho

```

Questão 4) Considere o programa C abaixo.

```

pid_t pid;
sigjmp_buf buf;
int j;

void foo (int sig) {
    siglongjmp(buf, getpid());
}

void main() {
    sigset_t mask;
    sigemptyset(&mask);
    sigaddset(&mask, SIGUSR1);
    sigprocmask(SIG_UNBLOCK, &mask, NULL);
    signal (SIGUSR1, foo);
    j = sigsetjmp(buf, 0);
    printf("%d\n", j);
    pid = fork();
    if (pid == 0) {
        printf("%d \n", getpid());
        if (kill (getppid(), SIGUSR1) < 0) printf("erro");
        sleep(2);
        return;
    }
}

```


a) (10) Explique a interação entre os processos que serão criados e indique a saída que será impressa, assumindo que o processo inicial tem pid=100 e os processos filhos recebem pid crescente.

A primeira impressão do programa será '0', pois temos a função **sigsetjmp** e sabemos que o primeiro retorno dela é sempre um número zero. Nesse caso, temos "**j = sigsetjmp(buf,0);**", como o segundo argumento é zero, sabemos que o estado do vetor de sinais não será salvo no contexto e não será restaurado no retorno de **sigsetjmp**, assim, teremos um sinal bloqueante sendo enviado para o **SIGUSR1**. A cada vez que passamos por "**pid = fork();**", estamos criando um processo filho. E cada vez que o pid do processo em execução é igual a zero estamos executando um processo filho. Assim, se estivermos em um processo filho, será impresso o id do processo filho. Nesse momento, o processo filho emite um sinal forçando a execução do handler da função foo e iremos executar a seguinte linha "**siglongjmp(buf,getpid());**" que se encontra dentro da função. Com o **siglongjmp**, iremos "pular" a execução para a seguinte linha "**j = sigsetjmp(buf,0);**", que está dentro da função main, porém, neste momento não teremos mais o '0' como segundo argumento, e sim, o pid da função pai, que foi carregado no **siglongjmp** pelo **getpid()**.

Podemos observar que após o processo pai dorme por 2 segundos, o que dá tempo da função foo ser chamada e o pid do processo filho ser printado. Nesse momento, teremos o número "101" sendo printado na tela, visto que o processo pai é "100" e os filhos possuem pid crescente.

Após vamos passar pelo fork(), onde mais um processo filho será criado e entraremos no seguinte if "**if (pid==0)**" e em seguida repetimos os mesmos passos já mencionados anteriormente. Assim, a próxima impressão será a da id do processo pai (100) sendo printado pela linha "**printf("%d\n",j);**". Em seguida, será printado o id do processo filho criado (102) pela linha "**printf("%d \n", getpid());**".

Teremos a seguinte impressão:

```
0 // j = 0
101 // primeiro processo filho
100 // processo pai
102 //segundo processo filho
```

b) (10) Considere agora que, no código acima, substituímos **sigsetjmp(buf,0)** por **sigsetjmp(buf,1)**. Qual a implicação desta modificação na interação entre os processos e na saída que será impressa? Se não houver alteração, justifique o porquê.

Agora vamos trocar a linha **sigsetjmp(buf,0)** por **sigsetjmp(buf,1)**. Podemos observar que agora temos que o estado do vetor de sinais será salvo no

contexto e será restaurado no retorno de **sigsetjmp**, assim, teremos um sinal não bloqueante sendo enviado para o **SIGUSR1**.

Podemos observar que ao processo retornar do tratador, o processo filho irá emitir um sinal para o pai pela linha “**kill (getppid(),SIGUSR1)**” e iremos executar a função **foo**. Porém, nesse caso, estamos lidando com um sinal não bloqueante, ou seja, esse sinal é sempre emitido e a função fica em looping infinito.

A primeira impressão será ‘0’, pois o primeiro retorno da função **sigsetjmp** é sempre igual a zero. E teremos sendo impresso o primeiro id do primeiro processo filho (101) pela linha “**printf(“%d \n”, getpid());**”. Daí iremos imprimir o processo pai (100) pela linha “**printf(“%d\n”,j);**”. E o id do segundo processo filho pela linha “**printf(“%d \n”, getpid());**”.

Porém, isso nunca vai terminar, pois o sinal é não bloqueante, assim, teremos a mesma sequência de passos acontecendo infinitamente o que nós leva a seguinte impressão:

```
0 // j = 0
101 // primeiro processo filho
100 // processo pai
102 //segundo processo filho
100 //processo pai
103 // terceiro processo filho
100 // processo pai
104 //quarto processo filho
100 //processo pai
.
.
.
```