

# Memorando

**De:** Pedro Afonso

**Nº de Matrícula:** 20220252

**Disciplina:** CPD

**Assunto:** Experiência da aula prática

**Data:** 16/03/24

## 1. Introdução

Este memorando descreve a execução de uma experiência laboratorial no desenvolvimento de aplicações em linguagem C no ambiente Linux. A experiência incluiu a utilização de ferramentas como gcc, gdb, make, e abordou técnicas de depuração e geração de executáveis. Foram realizados testes e manipulações de código, além de análise de erros comuns, como segmentation fault. As capturas de tela inseridas ao longo do memorando ilustram a execução dos passos descritos.

## 2. Experiências Realizadas

Durante a experiência, foram realizados os seguintes passos:

### Ciclo de Desenvolvimento e Geração do Executável:

- Descompactação do arquivo com o comando:
  - `$ tar -zxvf aula1-eg1.tgz`
- Compilação dos arquivos fonte:
  - `$ gcc -g -c list.c main.c`
- Ligação dos objetos para criar o executável:
  - `$ gcc -o main list.o main.o`
- Execução do programa:
  - `$ ./main`

### Utilização do Debugger GDB:

- Execução do programa com o gdb:
  - `$ gdb main`
- Colocação de breakpoint na função `insert_new_process` na linha 36:
  - `$ b list.c:36`
- Execução do programa até o breakpoint:
  - `$ r`
- Visualização de variáveis no escopo atual:
  - `$ p item`
  - `$ p *item`

**Nota:** O comando `display *item` só funcionará se, na sequência de execução, estivermos no escopo em que a variável `item` pertence.

1. Controle de execução com os comandos `step` e `next` para avançar pelo código.

### Análise de Erros:

- Definição do tamanho do arquivo `core dump`:

- \$ ulimit -c 10000000

**Nota:** O comando define o limite de 10 MB, mas o arquivo só será gerado se ocorrer um erro no programa.

- Análise de segmentation fault com o gdb:
- \$ gdb main core
- \$ bt

O comando backtrace (bt) mostra a pilha de chamadas até o ponto do erro.

### Utilização da Ferramenta Make:

- Criação do arquivo Makefile e execução do comando make para compilar o programa.
- Teste da recompilação ao modificar arquivos fonte ou cabeçalhos.
- Utilização da regra clean para limpar os arquivos objeto e o executável:
- \$ make clean

The screenshot shows a code editor with a C program named `list.c`. The code includes functions for inserting, updating, and printing a linked list. Below the code, a terminal window shows the execution of GDB. The user sets a breakpoint at line 40, runs the program, and the terminal output shows the program starting and then hitting the breakpoint. The GDB interface also displays the current state of the program, including the value of the `item` pointer and the `list` structure.

```

C main.c  C list.c
C list.c > insert_new_process(list_t*, int, time_t)
36 void insert_new_process(list_t *list, int pid, time_t starttime)
44     item->next = list->first;
45     list->first = item;
46
47
48
49 void update_terminated_process(list_t *list, int pid, time_t endtime)
50 {
51     printf("terminated process with pid: %d\n", pid);
52 }
53
54
55 void lst_print(list_t *list)
56 {
57     lst_iitem_t *item;

(gdb) b list.c:36
Ponto de parada 1 at 0x129e: file list.c, line 40.
(gdb) run
Starting program: /home/mv/Area de trabalho/main

This GDB supports auto-downloading debuginfo from the following URLs:
<https://debuginfod.ubuntu.com>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
Downloading separate debug info for system-supplied DSO at 0x7ffff7fc3000
Download failed: Não há rota para o host. Continuing without separate debug info for system-supplied DSO at 0x7ffff7fc3000
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
<<BEGIN>>

```

Figura 1 - Uso do gdb run, break

The screenshot shows the same code editor as Figure 1, but with the terminal window showing the GDB `next` command being used to step through the code. The output shows the program reaching the breakpoint at line 40, and the `next` command being used to move to the next line of code. The GDB interface also displays the current state of the program, including the value of the `item` pointer and the `list` structure.

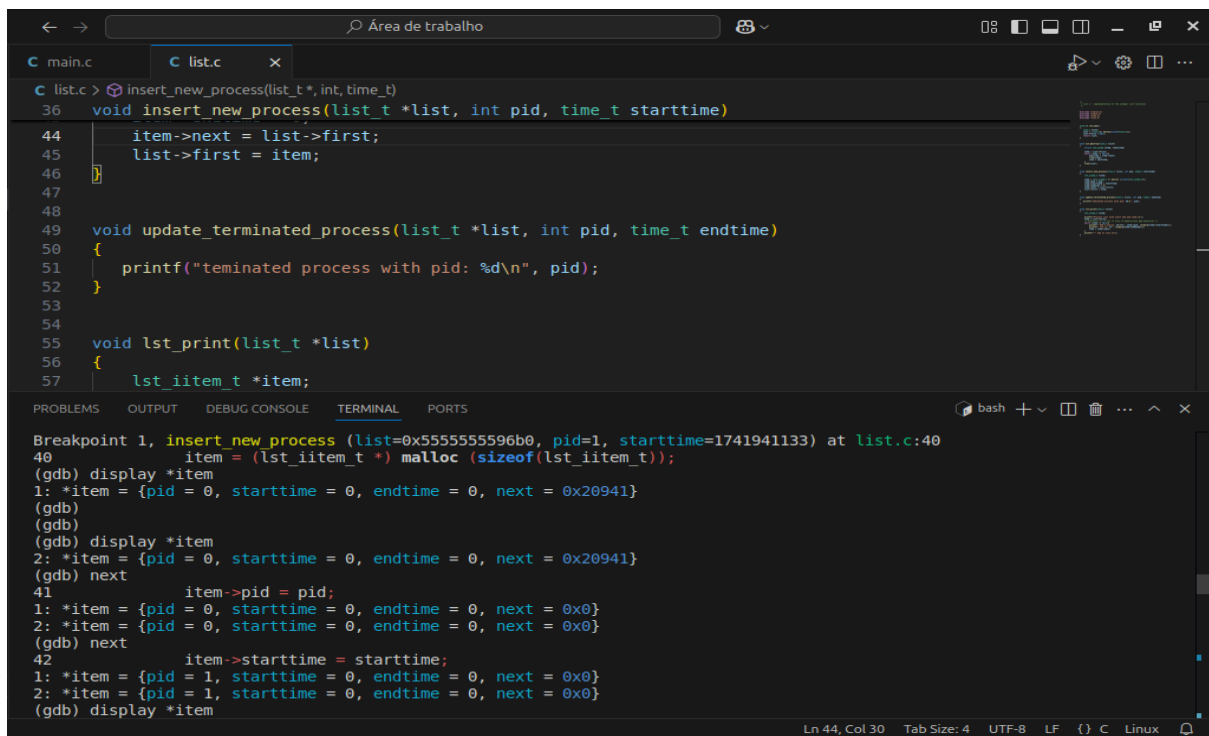
```

C main.c  C list.c
C list.c > insert_new_process(list_t*, int, time_t)
36 void insert_new_process(list_t *list, int pid, time_t starttime)
44     item->next = list->first;
45     list->first = item;
46
47
48
49 void update_terminated_process(list_t *list, int pid, time_t endtime)
50 {
51     printf("terminated process with pid: %d\n", pid);
52 }
53
54
55 void lst_print(list_t *list)
56 {
57     lst_iitem_t *item;

Breakpoint 1, insert_new_process (list=0x5555555596b0, pid=1, starttime=1741940486) at list.c:40
(gdb) print item
item = (lst_iitem_t *) malloc (sizeof(lst_iitem_t));
51 = (lst_iitem_t *) 0x5555555596b0
(gdb) print *item
52 = {pid = 0, starttime = 0, endtime = 0, next = 0x20941}
(gdb) next
41 item->pid = pid;
(gdb) next
42 item->starttime = starttime;
(gdb) next
43 item->endtime = 0;
(gdb) next
44 item->next = list->first;
(gdb) next

```

Figura 2 - Uso de gdb next e break



The screenshot shows a code editor with a C file named `list.c`. The code defines a linked list structure and functions to insert, update, and print processes. A breakpoint is set at line 40, which is the start of the `insert_new_process` function. The GDB terminal shows the execution state at this breakpoint. It displays the memory address of the list, the PID, and the start time. The terminal output shows the state of the list before and after inserting a new process, with the next pointer being updated.

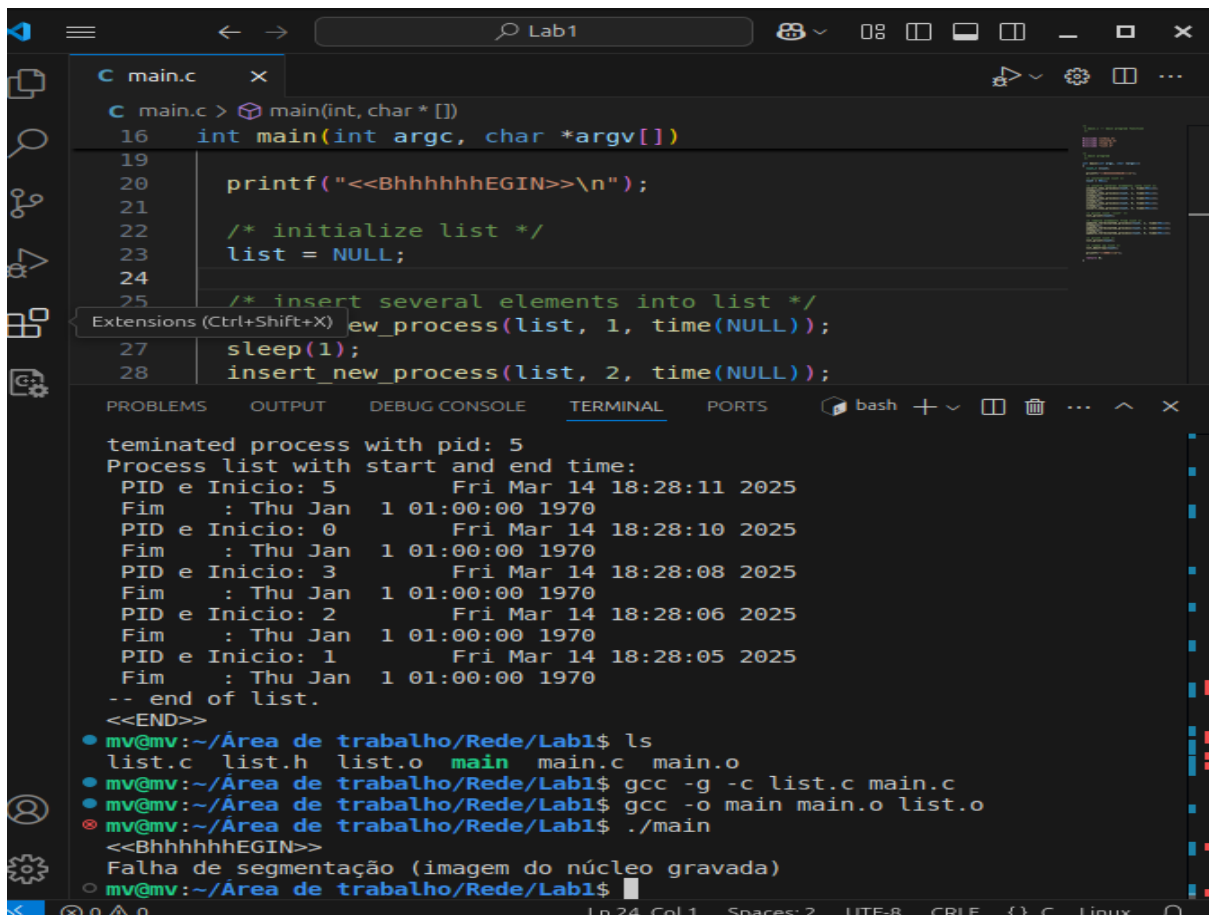
```
list.c:40 void insert_new_process(list_t *list, int pid, time_t starttime)
{
    item->next = list->first;
    list->first = item;
}

void update_terminated_process(list_t *list, int pid, time_t endtime)
{
    printf("teminated process with pid: %d\n", pid);
}

void lst_print(list_t *list)
{
    lst_iitem_t *item;
    while (item != NULL)
    {
        printf("PID e Inicio: %d %s\n", item->pid, item->start_time);
        item = item->next;
    }
    printf("-- end of list.\n");
}
```

Breakpoint 1, insert\_new\_process (list=0x5555555596b0, pid=1, starttime=1741941133) at list.c:40  
(gdb) display \*item  
1: \*item = {pid = 0, starttime = 0, endtime = 0, next = 0x20941}  
(gdb) display \*item  
2: \*item = {pid = 0, starttime = 0, endtime = 0, next = 0x20941}  
(gdb) next  
41: item->pid = pid;  
1: \*item = {pid = 0, starttime = 0, endtime = 0, next = 0x0}  
2: \*item = {pid = 0, starttime = 0, endtime = 0, next = 0x0}  
(gdb) next  
42: item->starttime = starttime;  
1: \*item = {pid = 1, starttime = 0, endtime = 0, next = 0x0}  
2: \*item = {pid = 1, starttime = 0, endtime = 0, next = 0x0}  
(gdb) display \*item

Figura 3 - Uso do comando gdb display

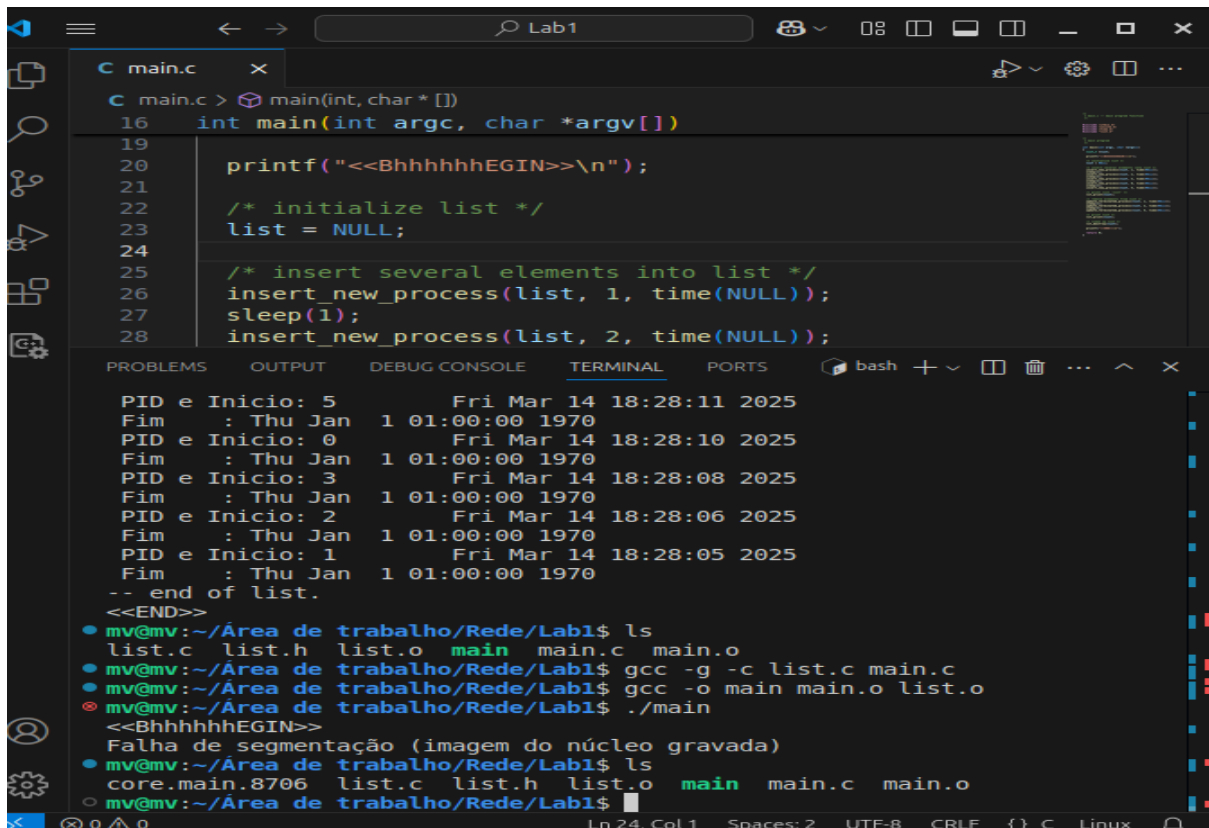


The screenshot shows a code editor with a C file named `main.c`. The code defines a `main` function that prints a header, initializes a list, inserts several elements, and prints the list. The terminal output shows the execution of the program, including the header, the list of processes, and the end of the list. The list contains five elements with PIDs 5, 0, 3, 2, and 1, all starting at the same time. The terminal output also shows the command prompt and the execution of the program.

```
main.c:16 int main(int argc, char *argv[])
{
    printf("<<BhhhhhhEGIN>>\n");
    /* initialize list */
    list = NULL;
    /* insert several elements into list */
    insert_new_process(list, 1, time(NULL));
    sleep(1);
    insert_new_process(list, 2, time(NULL));
}
```

teminated process with pid: 5  
Process list with start and end time:  
PID e Inicio: 5 Fri Mar 14 18:28:11 2025  
Fim : Thu Jan 1 01:00:00 1970  
PID e Inicio: 0 Fri Mar 14 18:28:10 2025  
Fim : Thu Jan 1 01:00:00 1970  
PID e Inicio: 3 Fri Mar 14 18:28:08 2025  
Fim : Thu Jan 1 01:00:00 1970  
PID e Inicio: 2 Fri Mar 14 18:28:06 2025  
Fim : Thu Jan 1 01:00:00 1970  
PID e Inicio: 1 Fri Mar 14 18:28:05 2025  
Fim : Thu Jan 1 01:00:00 1970  
-- end of list.  
<<END>>  
mv@mv:~/Área de trabalho/Rede/Lab1\$ ls  
list.c list.h list.o main main.c main.o  
mv@mv:~/Área de trabalho/Rede/Lab1\$ gcc -g -c list.c main.c  
mv@mv:~/Área de trabalho/Rede/Lab1\$ gcc -o main main.o list.o  
mv@mv:~/Área de trabalho/Rede/Lab1\$ ./main  
<<BhhhhhhEGIN>>  
Falha de segmentação (imagem do núcleo gravada)  
mv@mv:~/Área de trabalho/Rede/Lab1\$

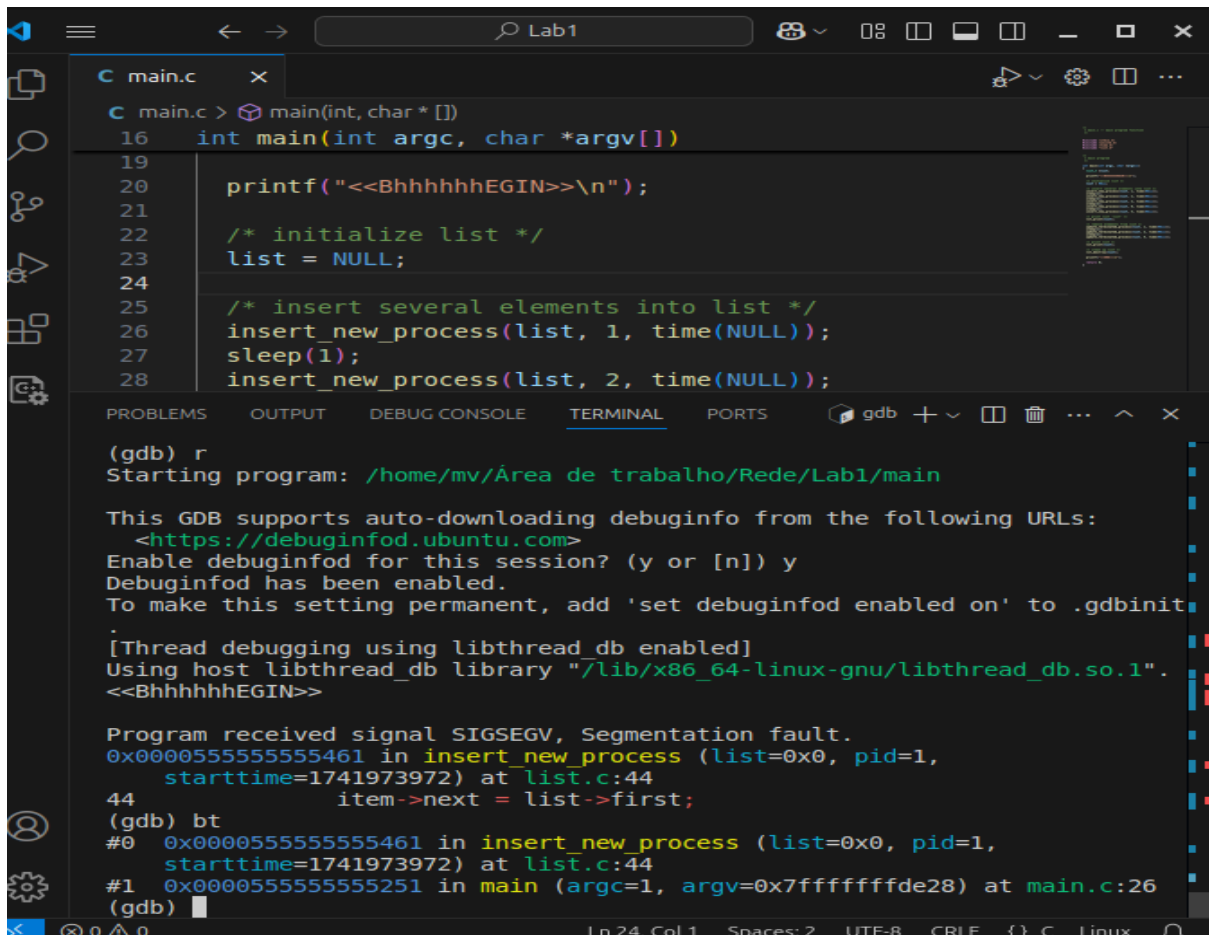
Figura 4 - Erro de falha de segmentação



```
C main.c > main(int, char * [])
16 int main(int argc, char *argv[])
19
20     printf("<<BhhhhhEGIN>>\n");
21
22     /* initialize list */
23     list = NULL;
24
25     /* insert several elements into list */
26     insert_new_process(list, 1, time(NULL));
27     sleep(1);
28     insert_new_process(list, 2, time(NULL));

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PID e Início: 5          Fri Mar 14 18:28:11 2025
Fim : Thu Jan 1 01:00:00 1970
PID e Início: 0          Fri Mar 14 18:28:10 2025
Fim : Thu Jan 1 01:00:00 1970
PID e Início: 3          Fri Mar 14 18:28:08 2025
Fim : Thu Jan 1 01:00:00 1970
PID e Início: 2          Fri Mar 14 18:28:06 2025
Fim : Thu Jan 1 01:00:00 1970
PID e Início: 1          Fri Mar 14 18:28:05 2025
Fim : Thu Jan 1 01:00:00 1970
-- end of list.
<<END>>
mv@mv:~/Área de trabalho/Rede/Lab1$ ls
list.c list.h list.o main main.c main.o
mv@mv:~/Área de trabalho/Rede/Lab1$ gcc -g -c list.c main.c
mv@mv:~/Área de trabalho/Rede/Lab1$ gcc -o main main.o list.o
mv@mv:~/Área de trabalho/Rede/Lab1$ ./main
<<BhhhhhEGIN>>
Falha de segmentação (imagem do núcleo gravada)
mv@mv:~/Área de trabalho/Rede/Lab1$ ls
core.main.8706 list.c list.h list.o main main.c main.o
mv@mv:~/Área de trabalho/Rede/Lab1$
```

Figura 5 - Arquivo Core Dump gerado



```
C main.c > main(int, char * [])
16 int main(int argc, char *argv[])
19
20     printf("<<BhhhhhEGIN>>\n");
21
22     /* initialize list */
23     list = NULL;
24
25     /* insert several elements into list */
26     insert_new_process(list, 1, time(NULL));
27     sleep(1);
28     insert_new_process(list, 2, time(NULL));

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
(gdb) r
Starting program: /home/mv/Área de trabalho/Rede/Lab1/main

This GDB supports auto-downloading debuginfo from the following URLs:
<https://debuginfod.ubuntu.com>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
<<BhhhhhEGIN>>

Program received signal SIGSEGV, Segmentation fault.
0x0000555555555461 in insert_new_process (list=0x0, pid=1,
    starttime=1741973972) at list.c:44
44         item->next = list->first;
(gdb) bt
#0  0x0000555555555461 in insert_new_process (list=0x0, pid=1,
    starttime=1741973972) at list.c:44
#1  0x0000555555555251 in main (argc=1, argv=0x7fffffffde28) at main.c:26
(gdb)
```

Figura 6 - Visualizando o pilha de erro com bt

```
Makefile
1  main: list.o main.o
2      gcc -o main list.o main.o
3  list.o: list.c list.h
4      gcc -g -c list.c
5  main.o: main.c list.h
6      gcc -g -c main.c
```

```
-- end of list.
terminated process with pid: 1
terminated process with pid: 2
terminated process with pid: 5
Process list with start and end time:
PID e Inicio: 5          Fri Mar 14 19:59:53 2025
Fim      : Thu Jan  1 01:00:00 1970
PID e Inicio: 0          Fri Mar 14 19:59:52 2025
Fim      : Thu Jan  1 01:00:00 1970
PID e Inicio: 3          Fri Mar 14 19:59:50 2025
Fim      : Thu Jan  1 01:00:00 1970
PID e Inicio: 2          Fri Mar 14 19:59:48 2025
Fim      : Thu Jan  1 01:00:00 1970
PID e Inicio: 1          Fri Mar 14 19:59:47 2025
Fim      : Thu Jan  1 01:00:00 1970
-- end of list.
<<END>>
mv@mv:~/Área de trabalho/Rede/Lab1$ make
gcc -g -c list.c
gcc -o main list.o main.o
mv@mv:~/Área de trabalho/Rede/Lab1$
```

Figura 7 - Usu do make alterando o arquivo list.c sem o list.h

```
Makefile
1  list.o: list.c list.h
2      gcc -o main list.o main.o
3  list.o: list.c list.h
4      gcc -g -c list.c
5  main.o: main.c list.h
6      gcc -g -c main.c
```

```
Inferior 1 [process 9345] will be killed.

Quit anyway? (y or n) y
mv@mv:~/Área de trabalho/Rede/Lab1$ make
gcc -g -c main.c
gcc -o main main.o list.o
mv@mv:~/Área de trabalho/Rede/Lab1$ make
make: 'main' está atualizado.
mv@mv:~/Área de trabalho/Rede/Lab1$ ./main
<<BhhhhhHEGIN>>
Process list with start and end time:
PID e Inicio: 5          Fri Mar 14 19:59:53 2025
Fim      : Thu Jan  1 01:00:00 1970
PID e Inicio: 0          Fri Mar 14 19:59:52 2025
Fim      : Thu Jan  1 01:00:00 1970
PID e Inicio: 3          Fri Mar 14 19:59:50 2025
Fim      : Thu Jan  1 01:00:00 1970
PID e Inicio: 2          Fri Mar 14 19:59:48 2025
Fim      : Thu Jan  1 01:00:00 1970
PID e Inicio: 1          Fri Mar 14 19:59:47 2025
Fim      : Thu Jan  1 01:00:00 1970
```

Figura 8 - Uso do make alterando o arquivo main.c

The screenshot shows a code editor with three tabs: `main.c`, `list.h`, and `resposta.txt`. The `list.h` tab is active, displaying the following C code:

```
1  C list.h > TotalActividades(list_t *)
2  #ifndef LIST_H
3
4
5  /* lst_new - allocates memory for list_t and initial
6  list_t* lst_new();
7  int TotalActividades(list_t *lista);
8
9  /* lst_destroy - free memory of list_t and all its i
10 void lst_destroy(list_t *);
11
12 /* insert new process - insert a new item with proce
```

Below the code editor, the `TERMINAL` tab is active, showing the output of the `make` command:

```
• mv@mv:~/Área de trabalho/Rede/Lab1$ make
make: 'main' está atualizado.
• mv@mv:~/Área de trabalho/Rede/Lab1$ make
gcc -g -c list.c
gcc -g -c main.c
gcc -o main list.o main.o
• mv@mv:~/Área de trabalho/Rede/Lab1$
```

Figura 9 - Usando o make para executar o programa

The screenshot shows a code editor with three tabs: `main.c`, `list.h`, and `resposta.txt`. The `resposta.txt` tab is active, displaying the following text:

```
14 os foram gerados?
15 o altera-se um arquivo de cabeçalho, o Makefile reco
16
17 le a alteração do ficheiro list.o. O que acontece qu
18 E se agora fizer make?
19 do executamos make list.o, o make verifica as depend
```

Below the code editor, the `TERMINAL` tab is active, showing the output of the `make list.o` command:

```
• mv@mv:~/Área de trabalho/Rede/Lab1$ make list.o
make: 'list.o' está atualizado.
• mv@mv:~/Área de trabalho/Rede/Lab1$
```

Figura 10 - Usando o make list.o para executar a regra list.o

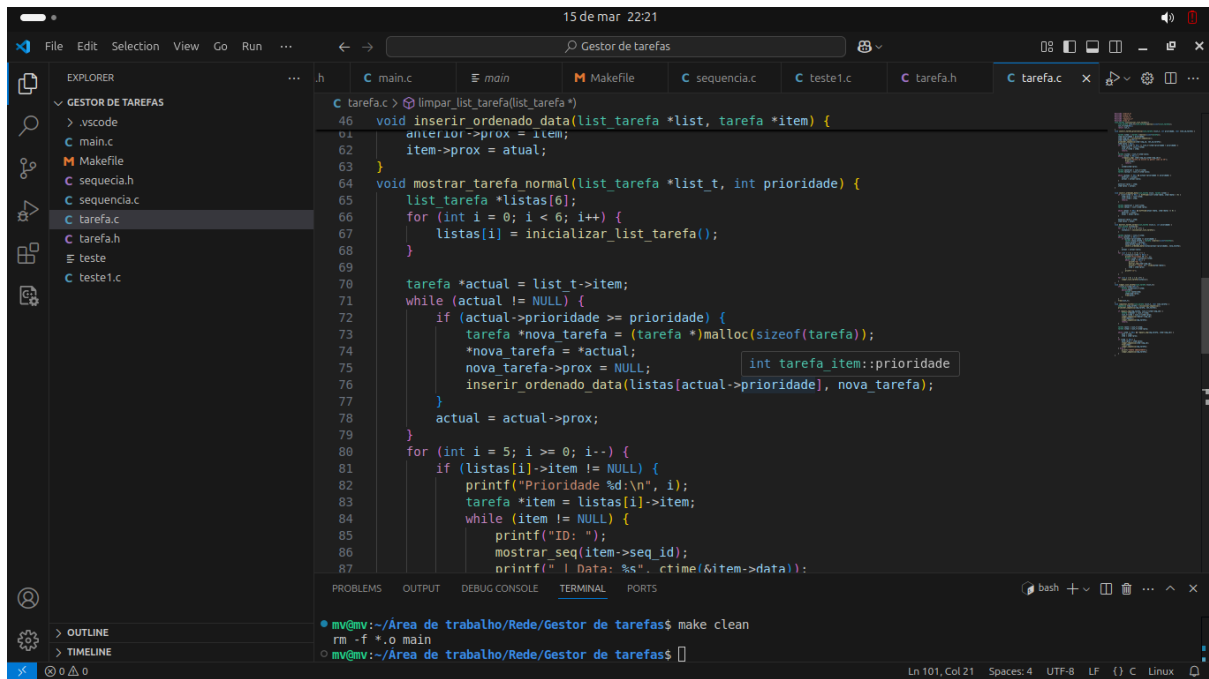


Figura 11 - Usando o make clean

## Solução da função update\_terminated\_process

1. void update\_terminated\_process(list\_t \*list, int pid, time\_t endtime) {
2. list\_t \*item = list->first;
3. while (item != NULL) {
4. if (item->pid == pid) {
5. item->endtime = endtime;
6. return;
7. }
8. item = item->next;
9. }
10. }
11. }

### 3. Desafios

Durante o desenvolvimento do projeto, alguns desafios foram enfrentados, como a compreensão das etapas de compilação e depuração no Unix e a análise de erros em tempo de execução.

a) Crie o ficheiro Makefile na sua área de trabalho e execute make. O que aconteceu?

R: Compilou o programa, executando os comandos `gcc -g -c main.c`, `gcc -g -c list.c`, depois linkou os arquivos .o que foram compilados.

b) Apague o ficheiro list.o. Re-execute make. Interprete o sucedido.

R: Como o arquivo list.o foi apagado, ao executar o comando make novamente, ele verifica que o arquivo objeto não está presente e, portanto, recompila o arquivo list.c para gerar o arquivo list.o novamente.

O Makefile possui dependências definidas na regra main, que dependem dos arquivos list.o e main.o. Sempre que um arquivo objeto (.o) necessário estiver ausente ou desatualizado, o make executa a regra correspondente para gerá-lo antes de criar o executável final.

c) Simule uma alteração ao ficheiro main.c com o comando seguinte e re-execute make. Compreenda o resultado.

R: Aconteceu algo parecido com a resposta anterior. O make percebeu que o arquivo main.c foi alterado e, por isso, recompilou apenas a dependência main.o para gerar um novo arquivo objeto atualizado. Em seguida linkou os arquivos object(.o).

d) Simule a alteração do ficheiro list.h e execute make. Porque razão todos os ficheiros foram gerados?

R: Quando altera-se um arquivo de cabeçalho, o Makefile recompila todos os arquivos que dependem desse cabeçalho, assim sendo foram recompilados o main.c e o list.c e depois linkados novamente.

e) Simule a alteração do ficheiro list.o. O que acontece quando faz make list.o? E se agora fizer make?

R: Quando executamos `make list.o`, o make verifica as dependências e, como houve alteração no list.c, recompila o arquivo com os dados atualizados do list.h e não relinka. Se rodarmos apenas make, ele vai relinkar não vai executar as dependências.

g) Adicione a regra seguinte no fim do ficheiro. O que descreve esta regra? Identifique: o alvo, as dependências e o comando.

R: Regra ou Alvo descreve o seguinte.

Um comando que apaga todos os arquivos do forma .o e o arquivo executavel main.

Alvo: clean.

Dependências: nenhuma

h) Execute `make clean`. O que aconteceu? Porque razão o comando é executado sempre que esta regra é invocada explicitamente?

R: Apagou os arquivos de formato .o e o arquivo main.



Ele acontece sempre que chamamos a regra clean explicitamente. Implementação da Função `update_terminated_process`

### **Solução da função `update_terminated_process`**

A função `update_terminated_process` foi desenvolvida para atualizar o tempo de término de um processo identificado pelo seu PID em uma lista encadeada. A função percorre a lista, localiza o processo correspondente e atualiza o campo `endtime`.

```
12. Código da função:
13. void update_terminated_process(list_t *list, int pid, time_t endtime) {
14.     lst_item_t *item = list->first;
15.     while (item != NULL) {
16.         if (item->pid == pid) {
17.             item->endtime = endtime;
18.             return;
19.         }
20.         item = item->next;
21.     }
22. }
23.
```

## **4. Referências Bibliográficas**

- Documentação oficial do GCC e GDB.
- Tutoriais sobre depuração e uso de Makefile no ambiente Unix.
- Apostilas e notas de aula fornecidas pelo professor.

## **5. Repositório GitHub**

[https://github.com/pedroaly/Memorando\\_1](https://github.com/pedroaly/Memorando_1).