- **Primeiro Tópico (README): utilização do programa**

  a)Como fazer:

  - Para compilar o código pode-se usar:
    $ gcc -Wall vmm.c -o (nome_que_desejar)

  Obs. : -Wall é opcional.

  -Para executar:
    $ ./(nome_que_desejar) (nome_algoritmo) 10 < anomaly.dat

  Obs.: “nome_algoritmo” deve assumir, obrigatoriamente, “fifo” ou “random”. São as formas configuradas no código fonte para especificar os respectivos algoritmos (notar que os nomes estão em minúsculo).

  a.1) Exemplo na prática:

```
[Ricardo@localhost FIFO]$ gcc vmm.c -o vmm
[Ricardo@localhost FIFO]$ ./vmm random 10 < anomaly.dat
faults: 10
[Ricardo@localhost FIFO]$ ./vmm fifo 10 < anomaly.dat
faults: 9
```

  b) Falhas previstas:

  b.1) Chamada do algoritmo incorreta

  Como podemos ver a seguir caso o nome do algoritmo seja passado de maneira incorreta ele não é executado:

```
[Ricardo@localhost FIFO]$ ./vmm fif 10 < anomaly.dat
Please pass a valid paging algorithm.
[Ricardo@localhost FIFO]$ ./vmm FIFO 10 < anomaly.dat
Please pass a valid paging algorithm.
```

  b.2) Faltando um parâmetro

  Nos casos abaixo na primeira execução não consta a frequência de clock e na segunda não consta a especificação do arquivo “anomaly.dat”:

```
[Ricardo@localhost FIFO]$ ./vmm fifO < anomaly.dat
Usage ./vmm <algorithm> <clock_freq>
[Ricardo@localhost FIFO]$ ./vmm fifO
Usage ./vmm <algorithm> <clock_freq>
```

Obs. : como o FIFO não necessita do parâmetro referente à frequência clock adotei 10 como valor default para evitar falhas ocasionais não relacionadas ao algoritmo implementado por mim. Logo, falhas relacionadas à variações desse valor durante os testes de random e FIFO não foram aprofundadas.

- **Segundo Tópico: Comparando os mecanismos Random e FIFO para substituição de páginas**

a) O algoritmo FIFO (*first-in-first-out* – primeiro a entrar primeiro a sair):

Esse mecanismo possui como vantagens um baixo custo e implementação simples. Em um Sistema Operacional (S.O.), não em vmm.c, a implementação do FIFO consiste em uma lista de páginas (*frames*) na memória principal. Como o nome sugere, a página no começo da fila é a mais antiga e deve ser substituída. Com isso a página que estava no começo vai para o final, se tornando a mais recente.

Durante este projeto de simulação de memória virtual, uma FIFO foi implementada como uma função na linguagem C. Essa função fifo() visa simular o funcionamento do mecanismo original comparando uma variável (int fifo_frm – carrega o *frame* mais antigo) à pagina real que "guarda" a página virtual a ser substituída.

b) Tabela de resultados:

Execução n	Page Faults Random	Page Faults FIFO
0	9	10
1	9	10
2	9	8
3	9	7
4	9	7
5	9	10
6	9	10
7	9	8
8	9	8
9	9	8

Como podemos observar em algumas execuções a simulação do mecanismo de substituição randômica teve um número menor de falta de páginas. Esse acontecimento representa bem uma descrição de Tanenbaum em seu livro: "Quando se aplica o algoritmo FIFO em armazéns pode-se tanto remover itens pouco vendidos quanto itens muito vendidos , como farinha". Ou seja, é uma lógica que faz sentido, mas na realidade surgem casos como a farinha do exemplo. A página carrega algo importante, mas em um espaço determinado pela disponibilidade física e quantidade de requisições essa página virtual é descartada. Esse cenário é responsável pela quantidade considerável de *page faults* geradas pelo algoritmo, não só durante a simulação.

Enquanto isso o Random possui um resultado mais eficiente durante os testes. Se analisarmos por uma métrica muito simples (média aritmética), percebemos que o Random tem um número de 8,6 enquanto o FIFO de 9,0. Apesar de ser uma técnica simples e de ter demonstrado resultado melhor que a fila, o Random ainda é uma abordagem ineficiente. É fácil perceber isso comparando o número de requisições com o de falta de páginas. Ou seja, considerando os aspectos do cenário em que foi aplicado concluímos que realmente não é muito eficiente.