Relatório – Prática 1

1.1 O programa inicia executando a chamada de “fork” que, em sistemas UNIX, cria um processo filho a partir de um processo pai, possuindo as mesmas variáveis de ambiente, mesmos arquivos e mesmos dados, porém em um espaço de endereçamento distinto. Em seguida, o programa verifica qual processo foi escalonado pelo sistema operacional para executar após o fork. Caso o filho tenha sido escolhido, uma instrução para dormir por 10 segundos é executada e após este período o processo termina. Se o pai tiver sido escalonado, é executada a chamada de sistema “waitpid”, que faz com que o processo atual seja bloqueado até que o processo com um determinado ID seja concluído. Após o término do processo filho, o processo pai escreve o código de status de saída do filho no terminal e finaliza a sua execução.

```
Pai com filho com PID 10787.
Aguardando término do filho!
Filho (PID 10787) vai dormir por 10 segundos.
Filho terminando a execução.
Pai viu filho com PID 10787 terminar.
Filho terminou com status = 0.
Pai terminando a execução.
```

1.2 O processo filho foi concluído antecipadamente pelo sinal enviado. No código há uma lógica que para verificar se o processo foi executado normalmente ou foi finalizado por um sinal (como o programa não inclui um tratador de sinais, um sinal pode provocar o término antecipado de sua execução).

```
Pai com filho com PID 11535.
Aguardando término do filho!
Filho (PID 11535) vai dormir por 10 segundos.
Pai viu filho com PID 11535 terminar.
Filho foi morto por sinal 2.
Pai terminando a execução.
```

2.1 O processo filho foi terminado antecipadamente pois recebeu o sinal SIGFPE do sistema operacional, dado que o código de saída do programa foi 8, de acordo com a mensagem escrita pelo processo pai. Este sinal é enviado para programas com erros em operações aritméticas com valores de ponto flutuante, como divisões por 0, por exemplo. Como no código do processo filho há uma divisão por 0, ele é encerrado pelo sinal SIGFPE.

```
Pai com filho com PID 11648.
Aguardando término do filho!
Pai viu filho com PID 11648 terminar.
Filho foi morto por sinal 8.
Pai terminando a execução.
```

2.2 O processo filho foi terminado antecipadamente pois recebeu o sinal SIGSEGV do sistema operacional, indicando que o programa tentou acessar um endereço de memória fora do espaço alocado para ele.

```
Pai com filho com PID 11739.
Aguardando término do filho!
Pai viu filho com PID 11739 terminar.
Filho foi morto por sinal 11.
Pai terminando a execução.
```

3.1 Na primeira execução, o programa se comporta da mesma maneira que o programa 1:

```
Pai com filho com PID 11979.
Aguardando término do filho!
Filho terminando a execução.
Pai viu filho com PID 11979 terminar.
Filho terminou com status = 0.
Pai terminando a execução.
```

Porém, na segunda execução, quando o sinal SIGINT foi enviado ao processo filho, uma mensagem é escrita no terminal informando que o sinal está sendo tratado e o programa continua sua execução.

```
Pai com filho com PID 12092.
Aguardando término do filho!
Tratando sinal SIGINT do processo 12092!
Na verdade, acho que não vou fazer nada! :-)
Filho terminando a execução.
Pai viu filho com PID 12092 terminar.
Filho terminou com status = 0.
Pai terminando a execução.
```

3.2 Não foi possível tratar o sinal pois o sinal enviado pelo sistema operacional devido a acesso indevido a memória não pode ser tratado por programas de usuário.

```
Pai com filho com PID 12227.
Aguardando término do filho!
Pai viu filho com PID 12227 terminar.
Filho foi morto por sinal 11.
Pai terminando a execução.
```

4.1 Na primeira execução o programa não foi encerrado antecipadamente após o envio do sinal SIGQUIT:

```
Pai com filho com PID 12744.
Aguardando término do filho!
Filho (PID 12744) vai dormir por 10 segundos.
Filho terminando a execução.
Pai viu filho com PID 12744 terminar.
Filho terminou com status = 0.
Pai terminando a execução.
```

Porém na segunda execução, o envio do sinal SIGTERM foi suficiente para encerrar o processo:

```
Pai com filho com PID 12881.
Aguardando término do filho!
Filho (PID 12881) vai dormir por 10 segundos.
Pai viu filho com PID 12881 terminar.
Filho foi morto por sinal 15.
Pai terminando a execução.
```

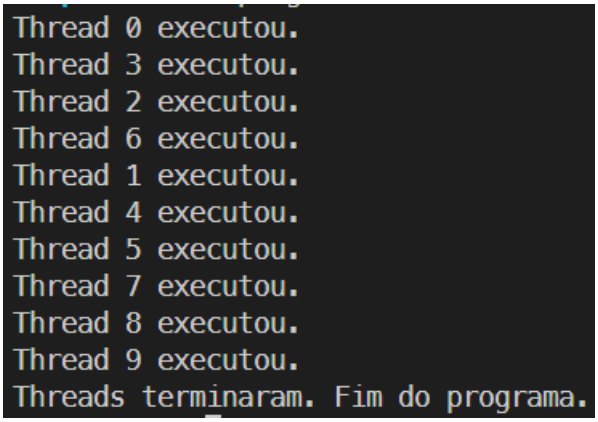
Esse comportamento é explicado pela máscara de sinal executada no código. Antes de executar a chamada de sistema “fork”, o programa instala uma máscara no sinal SIGQUIT com a chamada “sigaddset”, fazendo com que o sinal seja ignorado pelo processo e pelos seus descendentes.

4.2 O sinal SIGKILL não pode ser mascarado e nem tratado, de modo que um processo não pode impedir que o sistema operacional o encerre:

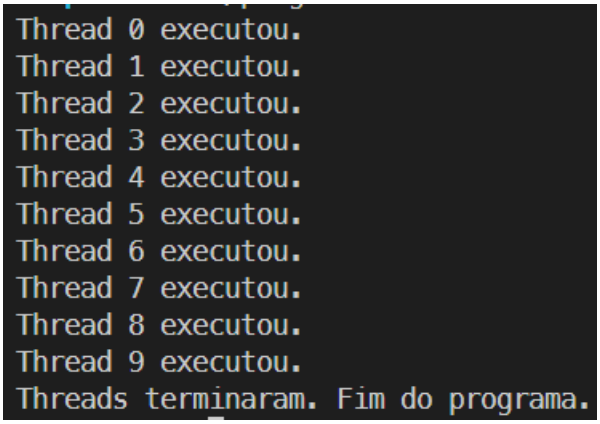
```
Pai com filho com PID 13110.
Aguardando término do filho!
Filho (PID 13110) vai dormir por 10 segundos.
Pai viu filho com PID 13110 terminar.
Filho foi morto por sinal 9.
Pai terminando a execução.
```

Por outro lado, o sinal SIGSTOP pode ser mascarado, de modo que é possível implementar um programa que impeça que outros processos o encerrem enviando este sinal.

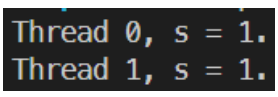
5.1 O programa inicia alocando estaticamente um vetor para *threads* e uma lista de inteiros para representar um contador. Em seguida ele cria 10 *threads*, identificando cada uma com números de 0 a 9, para executar uma função que imprime o identificador da *thread* atual. Por fim, o programa executa uma função para esperar que todas as 10 *threads* criadas finalizem sua execução.



Executando-se o programa diversas vezes, é possível notar que em cada vez a saída é diferente. Isso se deve ao fato de que em cada execução o sistema de tempo de execução de escalona as *threads* do programa em uma ordem distinta.



5.2 Executando-se o programa diversas vezes em sequência, nota-se que as duas *threads* sempre imprimem o mesmo valor da variável “s”, mesmo que cada *thread* atribua um valor distinto para ela e logo em seguida escreva no terminal o seu valor:



Este comportamento indica uma condição de corrida na variável “s”. Para resolver este problema, e necessário incluir no programa um procedimento para garantir exclusão mútua no acesso a variável, de modo que apenas uma *thread* escreva e leia a variável por vez.

6.1 Diferentemente do caso anterior, agora cada *thread* do programa sempre escreve no terminal o valor esperado da variável “s”, que é o valor atribuído durante a execução da *thread*, variando apenas a ordem com que as *threads* são escalonadas:

```
Thread 1, s = 1.
Thread 0, s = 0.
```

Isto se deve ao fato de que desta vez foi utilizado um sistema de exclusão mútua no código, onde cada *thread* espera até que a outra termine de utilizar a variável compartilhada antes de executar seus comandos.

7.1 O programa “prod” cria mensagens e as insere em um sistema de filas fornecida pelo sistema operacional, e o programa “cons” retira essas mensagens da fila e as processa, escrevendo o seu conteúdo no terminal. Quando este programa é executado quando a fila em questão está vazia, ou seja, quando o primeiro programa nunca foi executado ou todas as mensagens já foram consumidas, ele fica esperando até que uma mensagem seja produzida. Assim que o programa encontra uma mensagem na fila, ele a processa e termina sua execução.

```
erro na criacao de mailbox: Success
o resultado eh: 0
```

```
Erro na criacao de mailbox: Success
o tipo da mensagem eh: 1
o conteudo da mensagem eh: Teste  1
```

7.2 Saída do programa (prod | cons):

```
o tipo da mensagem eh: 2                                  o tipo da mensagem eh: 2
o conteudo da mensagem eh:                                o conteudo da mensagem eh:
o tipo da mensagem eh: 2                                  o tipo da mensagem eh: 2
o conteudo da mensagem eh:                                o conteudo da mensagem eh:
o tipo da mensagem eh: 2                                  o tipo da mensagem eh: 2
o conteudo da mensagem eh:                                o conteudo da mensagem eh:
o tipo da mensagem eh: 2                                  o tipo da mensagem eh: 2
```