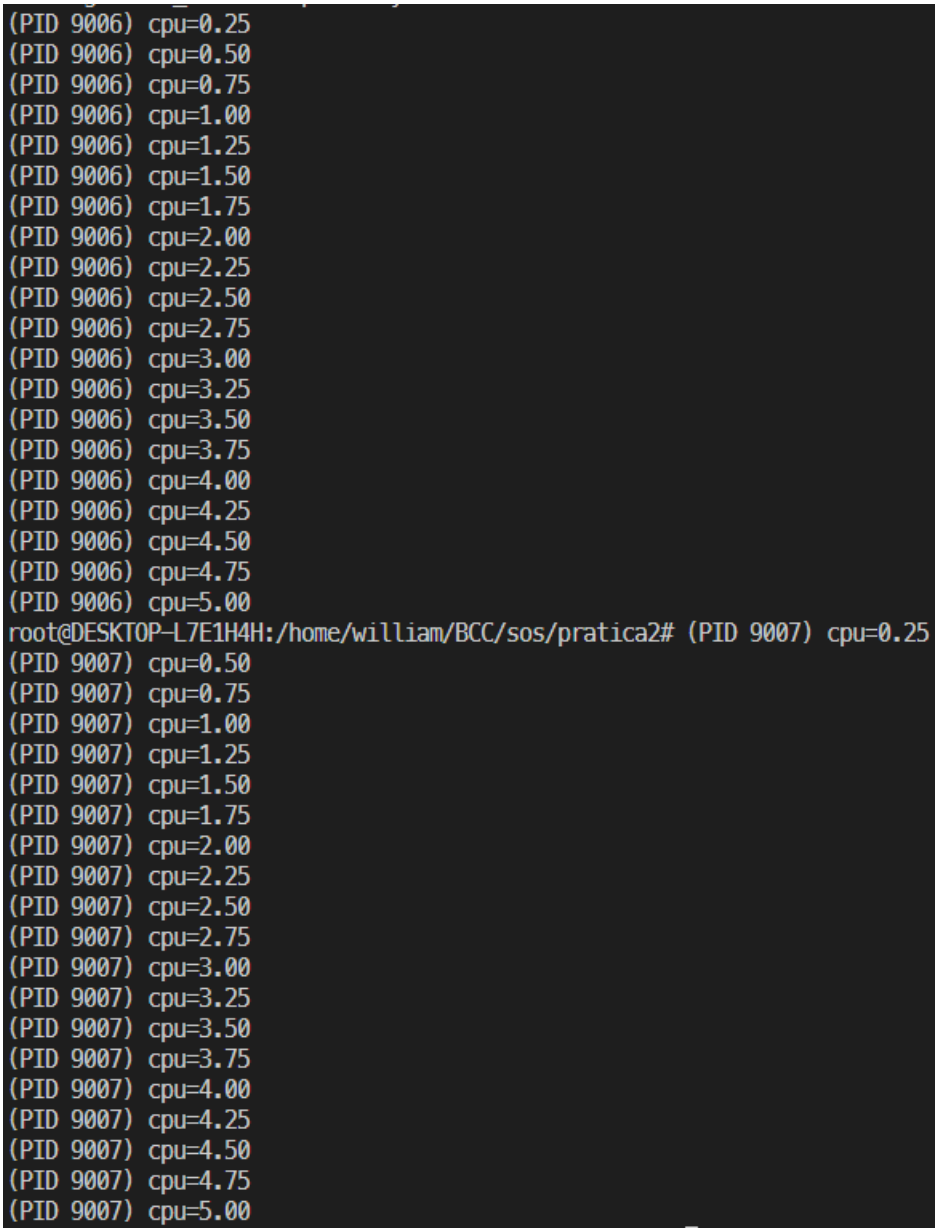
Relatório – Prática 2

1.1 Analisando-se a saída do programa, é possível notar que os processos criados pela chamada "fork" são executados um após o outro sem interrupções:



Isso pode ser explicado pelo tipo do escalonador utilizado e pela prioridade atribuída a cada processo. O escalonamento FIFO configurado no programa é do tipo não-preemptivo, ou seja, ele não bloqueia um processo para que outro possa utilizar a CPU, de modo que cada processo só inicia quando o anterior termina e os processos executam sem parar (exceto em caso de erros fatais ou sinais) até concluir. Como o mesmo nível de prioridade é atribuído a ambos os processos, qualquer um que seja escalonado pelo sistema após o "fork" executa até concluir, e em seguida o outro é acionado para fazer o mesmo.

Ao alterar o programa para que a rotina “useCPU2” fosse executada, o processo só começa a executar quando o processo pai é morto manualmente:

```
PID 10983 FIFO running (n=-1100000000)
PID 10983 FIFO running (n=-1000000000)
PID 10983 FIFO running (n=-900000000)
PID 10983 FIFO running (n=-800000000)
PID 10983 FIFO running (n=-700000000)
PID 10983 FIFO running (n=-600000000)
PID 10983 FIFO running (n=-500000000)
Killed
root@DESKTOP-L7E1H4H:/home/william/BCC/sos/pratica2# PID 10984
FIFO running (n=100000000)
PID 10984 FIFO running (n=200000000)
PID 10984 FIFO running (n=300000000)
PID 10984 FIFO running (n=400000000)
PID 10984 FIFO running (n=500000000)
PID 10984 FIFO running (n=600000000)
PID 10984 FIFO running (n=700000000)
```

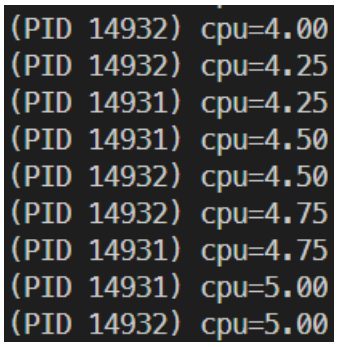
Se nenhum sinal é enviado para o processo pai, este executa sem parar, e o filho nunca ganha tempo de CPU.

1.2 De acordo com a saída do programa, o escalonador alterna entre os processos criados pelo programa a cada 0.25 segundos, de modo que os dois são executados praticamente ao mesmo tempo e concluem quase ao mesmo tempo:

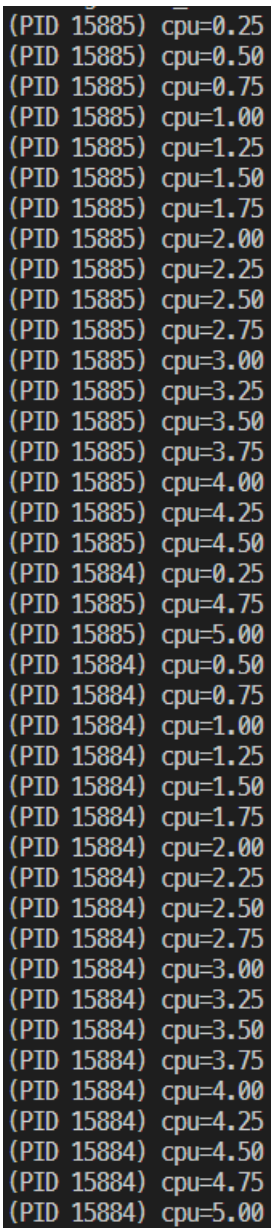
```
(PID 7223) cpu=0.25
(PID 7224) cpu=0.25
(PID 7223) cpu=0.50
(PID 7224) cpu=0.50
(PID 7223) cpu=0.75
(PID 7224) cpu=0.75
(PID 7223) cpu=1.00
(PID 7224) cpu=1.00
(PID 7223) cpu=1.25
(PID 7224) cpu=1.25
(PID 7223) cpu=1.50
(PID 7224) cpu=1.50
```

Este comportamento se deve ao escalonamento utilizado pelo programa. Com a estratégia *Round Robin* o escalonador é preemptivo, alternando entre os processos de modo que os processos com mesma prioridade ganhem o mesmo tempo de CPU.

1.3 Atribuindo-se uma prioridade maior para o processo filho, é possível ver pela saída do programa que ele ganhou mais tempo de CPU, dado que ele foi o que mais escreveu mensagens no terminal neste cenário:



1.4 O processo pai ficou sendo executado na CPU por 4 segundos sem ser interrompido, porém após este período o escalonador passou a alternar entre ele e o processo filho, com este último ganhando cada vez mais tempo de CPU:



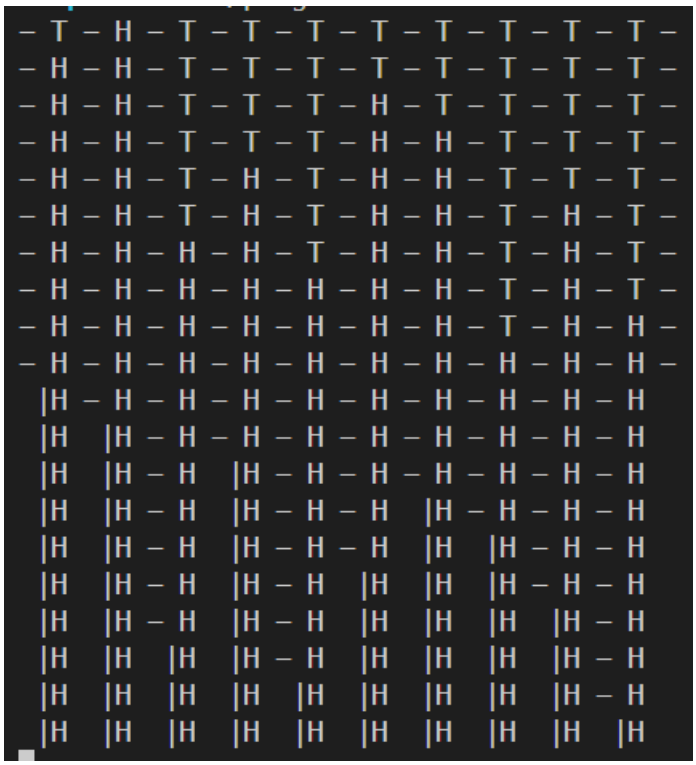
Este comportamento se verifica em escalonadores que implementam alguma metodologia de envelhecimento de processos. Conforme um processo é executado, a sua prioridade decai por um determinado fator, de forma que em algum momento os demais processos terão prioridades maiores e passarão a ganhar mais tempo de CPU.

2.1 As variáveis condicionais são usadas para que as threads executando o produtor ou o consumidor sedam voluntariamente o seu tempo de CPU na região crítica do programa para a outra thread quando alguma condição é satisfeita. Neste exemplo, quando o buffer de mensagens estiver cheio, o produtor não pode mais inserir itens e precisa esperar que o consumidor processe os itens existentes para liberar espaço. Analogamente, quando o buffer estiver vazio, o consumidor não tem nada a fazer até que o produtor insira novos dados no buffer. Assim, quando alguma destas condições é satisfeita, cada thread executa a operação

“wait” da variável condicional, permitindo que a outra thread entre na região crítica e execute o seu trabalho até que a outra condição seja satisfeita, momento no qual ela utiliza a mesma operação, resultando no envio de um sinal para que a primeira thread retome a sua execução.

```
Buffer vazio -- consumidor aguardando...
Produtor reiniciando...
item 0 inserido (valor=59)
item 1 inserido (valor=26)
item 2 inserido (valor=40)
item 3 inserido (valor=26)
item 4 inserido (valor=36)
item 5 inserido (valor=11)
item 6 inserido (valor=68)
item 7 inserido (valor=67)
Buffer cheio -- produtor aguardando...
Consumidor reiniciando...
item 7 removido (valor=67)
item 6 removido (valor=68)
item 5 removido (valor=11)
item 4 removido (valor=36)
item 3 removido (valor=26)
item 2 removido (valor=40)
item 1 removido (valor=26)
item 0 removido (valor=59)
```

3.1 O programa executa por um tempo porém trava quando chega a um estado em que todos os filósofos estão com um garfo na mão, de modo que nenhum consegue comer e o processo nunca avança. Este impasse se deve ao uso incorreto da estratégia de semáforo binário para resolução deste problema. Um filósofo não espera até que ambos os garfos estejam disponíveis antes de entrar em uma região crítica para pegá-los e comer, ele simplesmente pega um caso haja algum disponível, e em seguida tenta pegar o outro. Deste modo o programa chega em ponto em que não dois garfos disponíveis para um filósofo e todos ficam esperando até que um seja solto, o que nunca acontece



3.2 Este programa resolve o problema do jantar dos filósofos de forma correta, pois todos os filósofos que desejam comer conseguem fazê-lo em algum momento e o programa não trava com todos eles com fome. A solução se baseia em um teste que cada filósofo executa quando está com fome. Em vez de pegar um garfo e em seguida esperar até que o outro esteja disponível, um filósofo verifica se o filósofo a sua esquerda e o filósofo a sua direita estão comendo. Se for o caso, ele não faz nada e continua com fome, caso contrário ele pega os garfos e começa a comer. Esta operação é feita em uma região crítica, assim nunca ocorre o evento de dois filósofos próximos tentarem comer pegando um garfo ao mesmo tempo.

```
T H E T E T E T H E H
T H E T E T E T H T H
T H E T E T E T H T E
T H E T E T E T E T E
T H E H E T E T E T E
T H E H E H E T E T E
T H E H E H E H E T E
T H E H T H E H E T E
T H E H T H T H E T E
T H E H T E T H E T E
T H T H T E T H E T E
T H T E T E T H E T E
T E T E T E T H E T E
Total de refeicoes:
Filosofo 0: 9
Filosofo 1: 9
Filosofo 2: 9
Filosofo 3: 10
Filosofo 4: 9
Filosofo 5: 10
Filosofo 6: 9
Filosofo 7: 9
Filosofo 8: 9
Filosofo 9: 9
Filosofo 10: 8
```

4.1 Durante a execução do programa, os leitores leram o mesmo valor do banco de dados diversas vezes e ficaram bloqueados pela operação dos escritores, que operam um por vez, sem poder ler os novos valores conforme eles eram escritos.

```
Leitor 142 leu o valor 0
Leitor 143 leu o valor 0
Leitor 144 leu o valor 0
Leitor 145 leu o valor 0
Leitor 146 leu o valor 0
Escritor 1 escreveu o valor 1
Escritor 2 escreveu o valor 2
Escritor 3 escreveu o valor 3
Escritor 4 escreveu o valor 4
Escritor 6 escreveu o valor 6
Escritor 5 escreveu o valor 5
Escritor 8 escreveu o valor 8
Escritor 9 escreveu o valor 9
Escritor 10 escreveu o valor 10
```

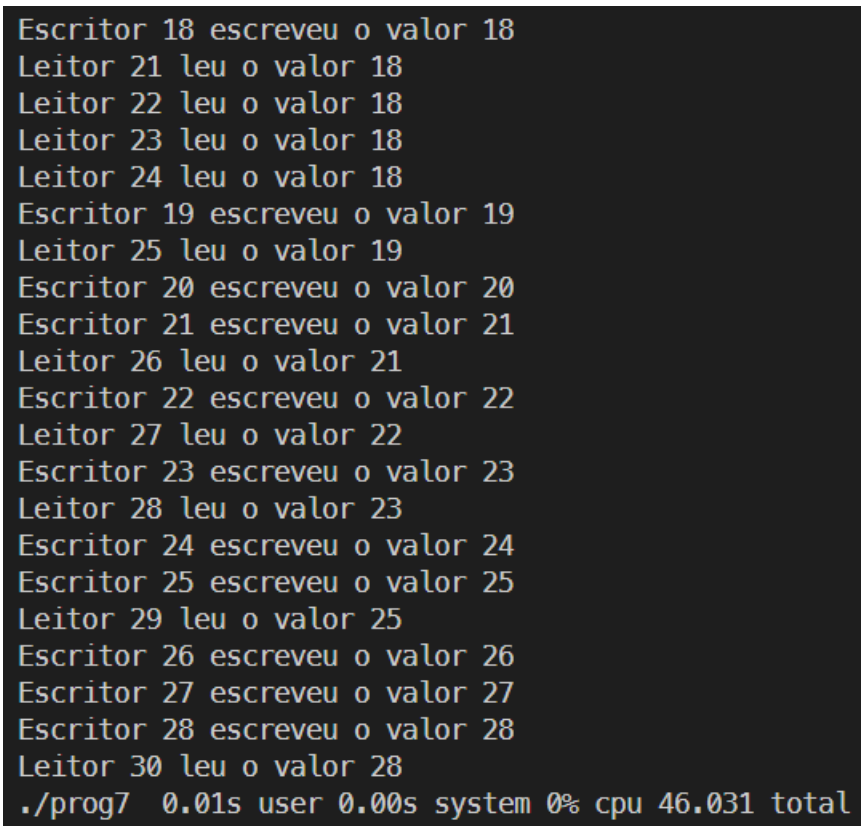
Isto se deve ao fato de que os leitores não leem o valor em uma região crítica, apenas decrementa o contador de leituras, de modo que quando este chega a zero os escritores bloqueiam a operação dos leitores.

4.2 Agora os leitores não são bloqueados pelos escritores e conseguem ler o valor no banco de dados paralelamente caso não haja um escritor querem alterar este valor.

```
Leitor 17 leu o valor 16
Leitor 18 leu o valor 16
Leitor 19 leu o valor 16
Escritor 17 escreveu o valor 17
Leitor 20 leu o valor 17
Escritor 18 escreveu o valor 18
Leitor 21 leu o valor 18
Leitor 22 leu o valor 18
Leitor 23 leu o valor 18
Leitor 24 leu o valor 18
Escritor 19 escreveu o valor 19
Leitor 25 leu o valor 19
Escritor 20 escreveu o valor 20
Escritor 21 escreveu o valor 21
Leitor 26 leu o valor 21
Escritor 22 escreveu o valor 22
Leitor 27 leu o valor 22
Escritor 23 escreveu o valor 23
Leitor 28 leu o valor 23
Escritor 24 escreveu o valor 24
Escritor 25 escreveu o valor 25
Leitor 29 leu o valor 25
Escritor 26 escreveu o valor 26
Escritor 27 escreveu o valor 27
Escritor 28 escreveu o valor 28
Leitor 30 leu o valor 28
./prog6  0.00s user 0.01s system 0% cpu 1:00.04 total
```

Contudo, a leitura do banco é bloqueante, de modo que apenas um leitor pode ler o banco por vez, o que prejudica performance do programa.

4.3 Houve uma melhoria de aproximadamente 25% no tempo de execução do programa (de 1 minuto para 46 segundos):



Este aumento na performance se deve a aplicação de exclusão mútua a partir de um mutex no programa, de tal modo que os leitores possam ler o valor no banco ao mesmo tempo.