Atividade Prática Unidade III

Autor:
Gabriel Otsuka
11721BCC018

Universidade Federal de Uberlândia

Sumário

1 Introdução

O exercício proposto tem como objetivo estudar na prática os vários tipos de IPCs estudados na disciplina de Sistemas Operacionais. De maneira geral, a estrutura da figura a seguir deve ser criada com o intuito final de que as três threads de $p7$ processem no total 10000 elementos, que estão no intervalo [1,1000], gerados aleatoriamente pelos processos $p1$, $p2$ e $p3$.

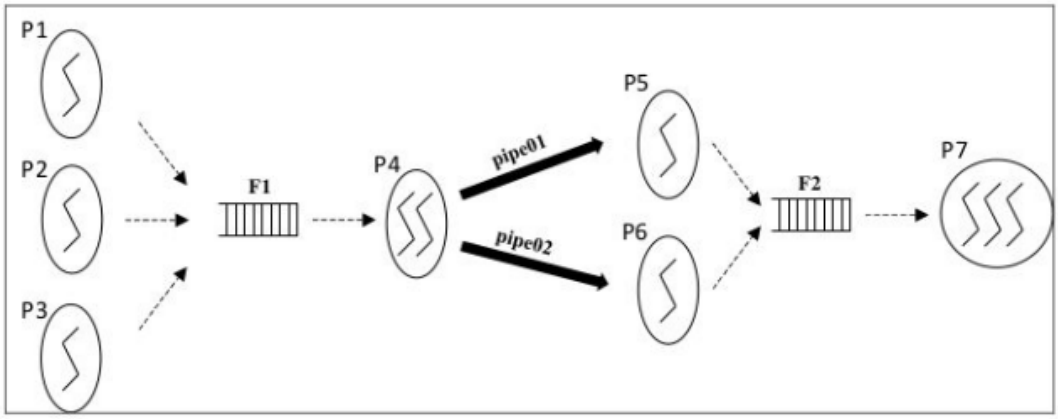

Figura 1: Estrutura do problema

Esse processamento é basicamente mostrar o dado retirado de $F2$ por $p7$ na saída padrão. Além disso, ao final do programa, deve-se mostrar um pequeno relatório com o tempo demandado para execução, quantos elementos $p5$ inseriu em $F2$, quantos elementos $p6$ inseriu em $F2$, o menor e o maior elemento processado por $p7$ e o elemento que foi mostrado na tela mais vezes (moda). Caso o conjunto de dados processados por $p7$ seja multimodal, apenas um desses valores deve ser impresso na tela.

Outros fatores importantes de se pontuar é que, os produtores ($p1$, $p2$, e $p3$) e o consumidor ($p4$) de $F1$ deverão atuar de forma alternada, ou seja, os produtores preenchem a fila por completo, o processo que inseriu o último elemento envia um *signal* para o $p4$, e os produtores esperam que o consumidor retire todos os elementos da $F1$ para que voltem a atuar. Esta fila deve proteger sua Região Crítica utilizando a estratégia de Semáforo.

Em contrapartida, a medida que os produtores ($p5$, $p6$) inserem elementos na $F2$, o consumidor $p7$ tenta retirar, sem ter que esperar a fila ser totalmente preenchida ou esvaziada. Esta fila deve proteger sua Região Crítica utilizando a estratégia de Busy Wait.

2 Definição do ambiente

Partindo do enunciado apresentado na introdução, sabe-se que será necessária a criação de 7 processos distintos e que cada um deles terá sua função. Estes processos deverão comunicar entre si e para que isso ocorra com sucesso, deve-se configurar todo o ambiente no qual eles executarão antes do processamento dos dados.

2.1 Structs

Primeiramente, as estruturas de dados que facilitam o controle do fluxo serão apresentadas para que o código fique mais claro posteriormente.

2.1.1 Estrutura de controle da F1

Todo fluxo que engloba a primeira fila será controlado utilizando os campos da seguinte struct

```
1 //Estrutura Fila 1
2 struct queue1_t {
3   sem_t mutex;
4   int fst, lst, count;
5   int F1[QUEUE_SZ];
6   int toggleAction; // 0: Produzir  |  1: Consumir
7   int sendSignal;   // 0: P4 nao pode receber sinal  |  1: P4
      pode receber sinal
8 };
9 typedef struct queue1_t * Queue1;
```

$F1$ é o vetor de tamanho $QUEUE\_SZ$ (constante correspondente à 10 para cumprir exigência do exercício proposto) e é o recurso compartilhado entre as threads dos processos $p1$, $p2$, $p3$ e $p4$. Nele ocorrem a inserção e remoção de dados, cujo controle é feito pelas variáveis fst, lst e $count$. Suas utilizações serão descritas posteriormente.

$mutex$ é uma variável do tipo sem_t e terá a responsabilidade única de não permitir que mais de uma thread acesse o recurso compartilhado $F1$ simultaneamente. Esta variável é utilizada para estratégia de exclusão mútua $Semaphore$, que evita insconsistência de dados em contextos multithreadeds.

$toggleAction$ é uma flag que terá dois possíveis valores, 0 ou 1. Quando 0 for atribuido a ela, $F1$ está vazia, e os processos $p1$, $p2$ e $p3$ poderão produzir os elementos para preencherem a fila, além disso, não permite que as threads

de $p4$ consumam esses valores. Em contrapartida, quando seu valor for 1, $p4$ terá permissão de consumir os dados disponíveis à ele em $F1$, assim como bloqueará os produtores para não inserirem novos elementos na fila.

sendSignal é outra flag utilizada para o controle do envio do sinal dos produtores para o $p4$. Sem este controle existe a possibilidade do sinal ser enviado sem que $p4$ esteja preparado para recebê-lo, gerando erros na execução do programa.

2.1.2 Estrutura de controle da F2

Assim como a struct $queue1\_t$, esta estrutura que será apresentada possibilitará o controle do fluxo em $F2$.

```
1 //Estrutura Fila 2
2 struct queue2_t {
3   int turn; // 0 = P5 | 1 = P6 | 2 = T1P7 | 3 = T2P7 | 4 = T3P7
4   int fst, lst, count;
5   int F2[QUEUE_SZ];
6 };
7 typedef struct queue2_t * Queue2;
```

Os campos descritos nas linhas 4 e 5 são equivalentes aos da struct $queue1\_T$. Apesar disso, no contexto da segunda fila, a exclusão mútua utilizará a variável *turn*, responsável pelo controle da estratégia de Busy Wait. Essa estratégia basicamente consiste no processo aguardar sua vez de executar a Região Crítica. Como definidos no comentário da linha 3, observa-se que 0 corresponde à vez do produtor $p5$, 1 à vez de $p6$, 2 à vez da primeira thread de $p7$ e assim por diante.

2.1.3 Estrutura de controle do Resultado

Os campos da estrutura descrita a seguir, têm a finalidade de armazenar os dados capturados para as métricas que o exercício propôs.

```
1 //Estrutura do relatorio resultante
2 struct report_t {
3   sem_t mutex;
4   int counterP5, counterP6; //Quantidade de elementos
      processados por p5 e p6
5   int counterTotal; //Quantidade de elementos processados por p7
6   int counterEach[INTERVAL+1]; //Elementos processados por p7
7 };
8 typedef struct report_t * Report;
```

Esta estrutura possui um semáforo *mutex* para proteger seus recursos compartilhados de possíveis acessos simultâneos.

Além disso, há três contadores, dentre eles, $counterP5$ e $counterP6$ utilizados para armazenar quantos elementos $p5$ e $p6$ inseriram em $F2$, além de um $counterTotal$ usado pelo $p7$ para encerrar os demais filhos quando atingir a meta de 10 mil elementos processados.

Já $counterEach$ é um array necessário para definir a moda dos elementos processados, assim como o menor e o maior número dentre eles. Posteriormente será explicado com mais detalhamento.

2.2 Variáveis globais

A solução sugerida exige a utilização da seguintes variáveis globais:

```
1 Queue1 queue1; //Ponteiro para estrutura da fila 1 (shared
    memory)
2 Queue2 queue2; //Ponteiro para estrutura da fila 2 (shared
    memory)
3 Report report; //Ponteiro para estrutura do relatorio resultante
    (shared memory)
4 int* pids; //Vetor com PIDs de todos os processos [pai,p1,p2,p3,
    p4,p5,p6,p7] (shared memory)
5 long int thread1p4Id; //TID da thread original do P4
6 int pipe01[2];
7 int pipe02[2];
```

Das linhas 1 a 3, temos ponteiros para as structs estudadas anteriormente. Na linha 4 há um ponteiro para inteiro, que representará um vetor com todos os PIDs dos processos envolvidos na solução, e será utilizado principalmente na troca de sinais entre os processos. Vale ressaltar que, como múltiplos processos acessarão esses ponteiros, todos deverão ser uma Shared Memory, já que processos distintos não compartilham a mesma pilha de memória.

$thread1p4Id$ é uma variável que armazena o TID da thread original de $p4$, e é utilizada em vários momentos do código.

Em sequência, os vetores $pipe01$ e $pipe02$ são utilizados para que os canais entre os processos $p4$, $p5$ e $p6$ sejam criados.

2.3 Métodos para criação do ambiente

Tendo em mente todos os tipos e variáveis que temos para configuração do ambiente, agora é necessário inicializá-lo.

2.3.1 Criação das Shared Memories

```
1  //Argumento type:
2  //  1 = Ponteiro para F1
3  //  2 = Ponteiro para vetor de PIDs
4  //  3 = Ponteiro para F2
5  //  4 = Ponteiro para Relatorio dos Resultados
6  void createSharedMemory (int type, int sharedMemorySize, int
       keySM) {
7    key_t key = keySM;
8    void *sharedMemory = (void *)0;
9    int shmid;
10
11   shmid = shmget(key, sharedMemorySize, 0666|IPC_CREAT);
12   if ( shmid == -1 ) {
13     printf("shmget failed\n");
14     exit(-1);
15   }
16
17   sharedMemory = shmat(shmid,(void*)0,0);
18
19   if (sharedMemory == (void *) -1 ) {
20     printf("shmat failed\n");
21     exit(-1);
22     }
23
24     if (type == 1) {
25       queue1 = (Queue1) sharedMemory;
26       createSemaphore(&queue1->mutex);
27     } else if (type == 2) {
28       pids = (int *) sharedMemory;
29       *(pids) = getpid(); // pids[0] = Pid do processo pai
30     } else if (type == 3) {
31       queue2 = (Queue2) sharedMemory;
32     } else if (type == 4) {
33       report = (Report) sharedMemory;
34       createSemaphore(&report->mutex);
35     }
36 }
```

Para a criação das Shared Memories, há esse método que recebe um tipo, o tamanho em bytes que deverá ter a Shared Memory e uma chave de identificação. O argumento type em especial define qual dos ponteiros globais mostrados no tópico anterior deverá ser inicializado.

2.3.2 Criação dos Semáforos e pipes

Estes métodos inicializam as pipes 1 e 2, assim como criam os semáforos já abertos.

```
1  //Inicializa semaforo
2  void createSemaphore (sem_t * semaphore) {
3    if ( sem_init(semaphore,1,1) != 0 ) {
4      printf("Semaphore creation failed\n");
5      exit(-1);
6    }
7  }
8
9  //Inicializa ambas pipes do projeto
10 void createPipes() {
11   if ( pipe(pipe01) == -1 ){ printf("Erro pipe()"); exit(-1); }
12   if ( pipe(pipe02) == -1 ){ printf("Erro pipe()"); exit(-1); }
13 }
```

2.3.3 Criação dos Processos Filhos

Agora que tudo que os processos utilizarão já foram criados, o próximo passo é criar os 7 processos filhos exigidos pelo enunciado.

```
1  //Processo pai cria todos os 7 filhos
2  int createChildren() {
3    pid_t p;
4    int id;
5
6    sem_wait((sem_t*)&queue1->mutex); //Pai fecha semaforo
7    for(id=1; id<=7; id++){
8      p = fork();
9      if ( p < 0 ) {
10       printf("fork failed\n");
11       exit(-1);
12     }
13     if ( p == 0 ) {
14       sem_wait((sem_t*)&queue1->mutex); //Filhos aguardam
       liberacao do pai
15       return id;
16     }
17     *(pids+id) = p; //Pai recebe PID do filho criado e insere
       valor no vetor pids
18   }
19
```

```
20    for (int i = 0; i < 8; ++i)
21      sem_post((sem_t*)&queue1->mutex); //Pai libera todos os
        filhos
22    for (int i = 0; i < 7; i++)
23      wait(NULL); //Pai espera o p7 encerrar todos os filhos
24
25    return 0;
26  }
```

Neste ponto, para que os processos aguardem a criação de seus irmãos, há uma sincronização na saída do método. Essa sincronização utiliza o semáforo da *queue*1. O processo pai na linha 1 fecha o semáforo com sem_wait, bloqueando todos os filhos que, na linha 14, tentam fechar o semáforo novamente. Quando o processo pai finaliza a criação dos filhos, lança um *sem_post* para cada *sem_wait* enviado. Outro fator importante de se pontuar, é que o preenchimento do vetor *pids* é feito na linha 17. Por fim, depois de criar todos os processos filhos, o pai aguarda o encerramento de todos eles para depois retomar a partir da linha 24. A ideia como um todo é fazer com que *p*7, ao processar todos os 10 mil dados, encerre a execução de todos os processos filhos, inclusive a sua própria e quando isso ocorrer, o processo pai é liberado para mostrar os resultados armazenados na shared memory report.

3 Primeiro contexto do problema

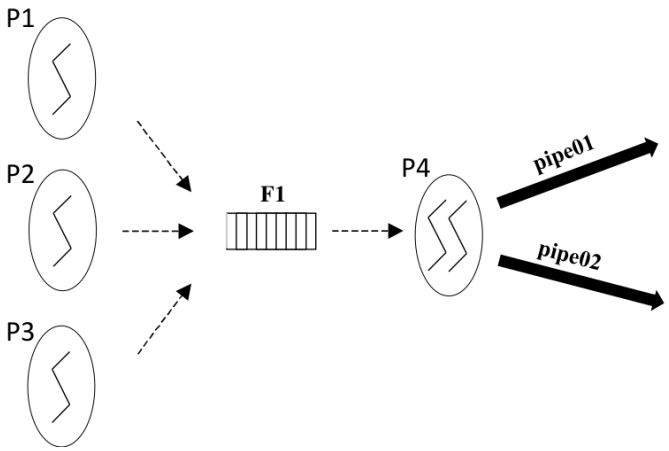

Figura 2: Primeira fila e threads envolvidas

Para um melhor entendimento do comportamento dos processos filhos, a proposta da solução será dividida em duas partes principais. Os códigos serão demonstrados de forma unitária e ao final, o programa como um todo estará descrito.

3.1 Produtores de F1

```
1  //p1, p2 e p3 produzem elementos aleatorios para F1
2  void producerF1() {
3    while(1) {
4      while(queue1->toggleAction != 0); //Controle para que nao
       produza quando p4 estiver consumindo
5
6      int response, random;
7      srand(getpid() + report->counterTotal - queue1->lst); //Seed
       para funcao random() sempre muda dessa forma
8      while(1) {
9
10       if (queue1->toggleAction == 1)
11         break; //Se a fila ja estiver sendo consumida, nao posso
         produzir
12
13       random = (rand()%INTERVAL)+1; //Gera numero aleatorio
       entre 1 e 1000
14       response = pushF1(random); //Tenta inserir na F1
15
16       if(response == 1) { //Ultimo elemento inserido na fila
17         while(queue1->sendSignal != 1); // Espero p4 estar
       pronto para receber sinal
18         while(kill(*(pids+4), SIGUSR1) == -1); //Tento enviar
       sinal para p4 consumir ate ter sucesso
19         break;
20
21       } else if (response == -1) //Fila cheia
22         break;
23     }
24   }
25 }
```

Os produtores de $F1$ tentarão gerar elementos infinitamente, até que $p7$ os encerre após processar os 10 mil elementos exigidos no enunciado.

Na linha 4 podemos observar o campo *toggleAction* em ação, evitando que produzam elementos enquanto $p4$ consome a fila. Na linha 10, há um

double check verificando novamente essa variável, pois ela pode ser alterada enquanto tentam preencher a fila. O ciclo de repetição da linha 7 encerrará assim que a fila for preenchida por completo.

Uma observação importante é o while da linha 17, o qual fará com que o processo que inseriu o último elemento aguarde a mudança da flag *sendSignal* por $p4$.

Observa-se que quando o retorno da função $pushF2()$ for igual a 1, o envio do sinal $SIGUSR1$ para $p4$ será realizado, assim que o campo *sendSignal* permitir, ou seja, assim que $p4$ estiver pronto para tratar o sinal.

3.1.1 Inserção de dados de F1

Em todos as tentativas de inserir ou remover elementos das filas, deverá existir uma estratégia de Exclusão Mútua, pois estes vetores ($queue1->F1$ e $queue2->F2$) são os recursos compartilhados entre os processos, logo as regiões em que exista manipulação desses campos são as Regiões Críticas do código. Como descrito no começo do relatório, $F1$ deve utilizar a estratégia de Semáforos.

```
1  //Tenta inserir elemento "value" na fila "queue"
2  int pushF1 (int value) {
3    sem_wait((sem_t*)&queue1->mutex);
4
5    if (queue1->count == QUEUE_SZ) { //Caso fila cheia
6      sem_post((sem_t*)&queue1->mutex);
7      return -1;
8    }
9
10   //Insiro novo elemento
11   queue1->F1[queue1->lst] = value;
12   queue1->lst = next(queue1->lst);
13   queue1->count++;
14
15   //Caso insercao encheu a fila, flagSendSignal == 1
16   int flagSendSignal = (queue1->count == QUEUE_SZ);
17
18   sem_post((sem_t*)&queue1->mutex);
19   return flagSendSignal;
20 }
21
22 //Calcula proxima posicao livre para realizar a insercao
23 int next (int position) {
24   return (position + 1) % QUEUE_SZ; //Insercao circular
25 }
```

Há três possibilidades de retorno na tentativa de inserir um elemento em $F1$.

-1: Não inseri elemento pois a fila já está cheia.

1: Inseri elemento com sucesso na última posição da fila.

0: Inseri elemento com sucesso nas demais posições da fila.

O retorno 1 é importante em $F1$ para que o produtor que inseriu o último elemento na fila saiba quando enviar o sinal para $p4$.

3.2 Consumidor de F1

$p4$ é o consumidor da primeira fila, e é um processo dualThread. Por este motivo, o primeiro passo que ele deve seguir é criar a sua thread secundária.

```
1  void createThreadP4() {
2    thread1p4Id = gettid(); //Defino TID da thread principal do p4
3
4    pthread_t thread2;
5    pthread_create(&thread2, NULL, p4SignalReceiver, NULL);
6    p4SignalReceiver();
7    pthread_join(thread2, NULL);
8  }
9
10 //Threads de p4 aguardam envio do sinal dos produtores
11 void* p4SignalReceiver() {
12   while(1) {
13     if (gettid() == thread1p4Id) {
14       signal(SIGUSR1, (__sighandler_t) setF1ToConsume);
15       queue1->sendSignal = 1; //Pronto para receber signal
16     }
17
18     while(queue1->toggleAction != 1); //Enquanto a fila nao
       estiver pronta para consumo, nao fazer nada
19
20     consumerF1();
21     setF1ToProduce();
22   }
23 }
24
25 //Bloqueia produtores de continuarem produzindo ao retirar
     elementos da F1
26 void* setF1ToConsume() {
27   queue1->toggleAction = 1; //Nao produza mais! F1 pode ser
     consumida
28 }
```

```
29
30 //Libera produtores para produzir elementos para F1
31 void setF1ToProduce() {
32   queue1->toggleAction = 0; //Produza mais! F1 nao pode ser
       consumida
33 }
```

Na linha 2, obverva-se onde a variável global *thread1p4Id* foi inicializada. Lembrando que não é necessário que seja uma Shared Memory, já que diferentes threads do mesmo processo compartilham da mesma pilha de memória.

Após isso, ambas threads chamam o método *p4SignalReceiver*(). Neste método, a thread primária de *p*4 configura o processo para que trate o sinal enviado pelos produtores. Um detalhe importante está na linha 15, na qual a flag *sendSignal* é configurada para que os produtores saibam que podem enviar o sinal. Posteriormente, quando o consumo é feito, 0 volta a ser atribuido a essa flag.

Ambas threads, ficam aguardando na linha 18 que o sinal seja recebido e a flag *toggleAction* as avise que o consumo pode ser feito. E posterior a esse consumo define *toggleAction* para produção novamente.

O método *consumerF*1() é onde exatamente as threads tentam retirar valores da fila e escrevê-los na pipe correspondente de cada thread.

```
1  //p4 (dualThread) consome F1 e escreve elementos nas pipes
2  void consumerF1() {
3    int response, value, resp;
4    while(1) {
5      response = popF1(&value);
6      if (response == 0) { //Se consegui retirar elemento
7
8        if (thread1p4Id == gettid()){ //Thread original n o
       permite envio de sinal
9          queue1->sendSignal = 0;
10         resp = write(pipe01[1], &value, sizeof(int)); //Envio
       para pipe01
11       } else {
12         resp = write(pipe02[1], &value, sizeof(int)); //Envio
       para pipe02
13       }
14
15       if(resp < 0) {
16         printf("Erro na escrita do pipe\n");
17         break;
```

```
18          }
19        } else if (response == -1)
20          break;
21      }
22  }
```

Mais uma vez, destava-se a importância de saber quem é a thread primária e quem é a secundária. Nesse caso, essa informação faz a distinção da pipe em que o valor lido será escrito.

3.2.1 Remoção de dados de F1

Diferente da inserção, a retirada de elementos não precisa definir se o elemento retirado esvaziou a fila, portanto este método é mais simples, retornando apenas o sucesso ou fracasso da remoção.

```
1  //Tenta retirar um elemento da fila 1 e inserir em "value"
       passado por referencia
2  int popF1 (int * value) {
3    sem_wait((sem_t*)&queue1->mutex);
4
5    if (queue1->count == 0) { //Caso fila ja vazia retorno erro
6      sem_post((sem_t*)&queue1->mutex);
7      return -1;
8    }
9
10   *value = queue1->F1[queue1->fst];
11   queue1->fst = next(queue1->fst);
12   queue1->count--;
13
14   sem_post((sem_t*)&queue1->mutex);
15   return 0;
16 }
```

Vale lembrar que este método também é uma região crítica e está sendo protegido com a utilização de semáforos.

4 Segundo contexto do problema

Na segunda parte, todo o contexto da Segunda Fila é abordado, como mostra a Figura 3.

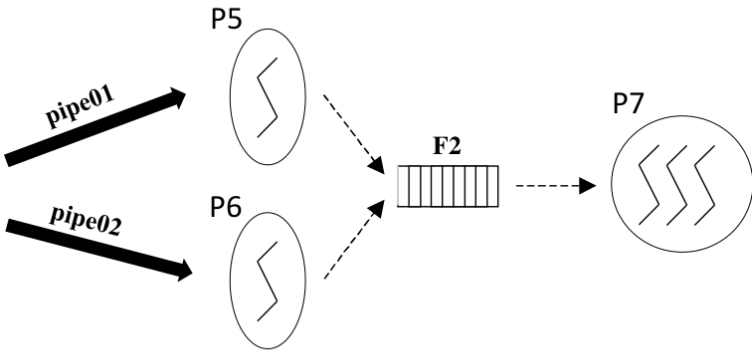

Figura 3: Segunda fila e threads envolvidas

4.1 Produtores de F2

Os processos $p5$ e $p6$ são os produtores de $F2$, e cada um é responsável por ler sua respectiva pipe e o dado retirado deve ser inserido na fila $F2$.

```
1  //Le elementos das pipes e insere em F2
2  void producerF2(int process) {
3
4    int value, resp, response;
5
6    while(1) {
7
8      if (process == 5)
9        resp = read(pipe01[0], &value, sizeof(int)); //Tentativa
       de leitura de pipe01
10     else if (process == 6)
11       resp = read(pipe02[0], &value, sizeof(int)); //Tentativa
       de leitura de pipe02
12
13     if(resp == -1) {
14       printf("Erro na leitura do pipe0%d\n", process-4);
15       break;
16     } else if (resp > 0) {
17       pushF2(value, process-5); //Tento colocar na F2
18     }
19   }
```

```
20 }
```

O código desses processos é bastante conciso e direto. Basicamente lerão da sua respectiva pipe e tentarão inserir em F2 repetidamente até que $p7$ processe todos os elementos exigidos pelo enunciado. O problema em si está no controle da inserção e remoção, e é o que será discutido a seguir.

4.1.1 Inserção de dados de F2

```
1  //Tenta inserir elemento "value" na fila "queue"
2  void pushF2 (int value, int turn) {
3    while (queue2->turn != turn); //Aguardo minha vez (busy wait)
4
5    if (queue2->count == QUEUE_SZ) {
6      queue2->turn = nextTurn(queue2->turn);
7      return;
8    }
9
10   //Se p7 ainda nao processou AMOUNT_DATA incrementa contadores
       de p5 e p6
11   sem_wait((sem_t*)&report->mutex);
12   if (report->counterTotal < AMOUNT_DATA)
13     (turn == 0) ? report->counterP5++ : report->counterP6++;
14   sem_post((sem_t*)&report->mutex);
15
16   queue2->F2[queue2->lst] = value;
17   queue2->lst = next(queue2->lst);
18   queue2->count++;
19
20   queue2->turn = nextTurn(queue2->turn);
21   return;
22 }
23
24 //Calcula qual thread vai processar regiao critica (busy wait)
25 int nextTurn(int turn) {
26   return (turn + 1)%5;
27 }
```

A grande diferença da estratégia Busy Wait pode ser observada na linha 3. Cada thread tem seu valor *turn* correspondente, e ao chegarem nesta linha, ficam aguardando até que outra thread passe sua vez a ela. Isso é controlado na variável *turn* da struct *queue*2, como citado anteriormente.

Dentre as linhas 11 e 14, há a contabilização de quantos elementos cada um dos produtores da segunda fila processou.

O cálculo para passar para próxima vez foi feito de duas maneiras. A descrita acima é a circular, ou seja, de 0 passa para 1, de 1 para 2, ..., de 4 para 0 novamente e assim por diante, até que $p7$ encerre os processos envolvidos. Utilizar essa estratégia circular para o cálculo da próxima thread a executar suas Região Crítica, implica em duas observações importantes. A primeira delas, é que os processos $p5$ e $p6$ sempre processarão a mesma quantidade de elementos, no caso 5 mil. A segunda observação, é que a terceira thread de $p7$ nunca processará nenhum elemento, já que são apenas 2 produtores e três threads, a fila sempre estará vazia na vez da terceira thread. Portanto, outra estratégia para a função $nextTurn()$ que evita essas observações é retornar um inteiro aleatório no intervalo $[0, 4]$ da seguinte forma:

```
1  //Calcula qual thread vai processar regiao critica. Aleatorio (
       busy wait)
2  int nextTurn2(int turn) {
3    return rand() % 5;
4  }
```

4.2 Consumidor de F2

Pelo fato do processo consumidor de $F2$, $p7$, possuir três threads, a primeira pendência a ser resolvida é a chamada dos $pthread_create$.

```
1  void createThreadsP7() {
2    pthread_t tid2, tid3;
3
4      pthread_create(&tid2, NULL, thread2p7, NULL);
5      pthread_create(&tid3, NULL, thread3p7, NULL);
6      consumerF2(2);
7
8    pthread_join(tid2, NULL);
9    pthread_join(tid3, NULL);
10 }
11
12 //Define segunda thread como vez 3
13 void * thread2p7() {
14   consumerF2(3);
15 }
16
17 //Define terceira thread como vez 4
18 void * thread3p7() {
19   consumerF2(4);
20 }
```

As funções chamadas pela criação das threads basicamente definem o argumento *turn* do método *consumerF2*() para que lá seja feito o controle do busy wait de acordo com cada thread. Observa-se também que na linha 6 a thread principal chama a mesma função de consumo de $F2$ com *turn* igual a 2. Assim garante-se a ordem que foi estipulada e explicada anteriormente (0 = P5; 1 = P6; 2 = T1P7; 3 = T2P7; 4 = T3P7).

```
1  //Tento consumir valores de F2
2  void consumerF2(int turn) {
3    int value, response;
4    while(1) {
5
6      response = popF2(&value, turn); //Tento retirar elemento da
       fila
7
8      if (response == 0) { //Se consegui retirar elemento
9        sem_wait((sem_t*)&report->mutex);
10
11       report->counterTotal++; //Incrementa quantidade de
       elementos processados por p7
12       report->counterEach[value]++; //Incrementa 1 na posi   o
       correspondente ao elemento processado por p7
13       if (report->counterTotal >= AMOUNT_DATA) //Se processei
       quantidade total de elementos que desejo
14         for (int i = 1; i <= 7; ++i)
15           kill(*(pids+i), SIGTERM); //Encerro execucao dos
       processos exceto pai
16
17       printf("Elemento %d retirado de F2\n", value);
18       sem_post((sem_t*)&report->mutex);
19     }
20   }
21 }
```

O $p7$, como citado várias vezes durante o relatório, é responsável por definir quando os outros processos filhos se encerrarão. A cada elemento que processam e mostram na tela, as threads contabilizam na struct *report* a quantidade total de elementos no campo *counterTotal*, assim como incrementam 1 no index respectivo do valor processado.

Quando alguma das três threads de $p7$ processa o 10000º elemento, ele encerra a execução de todos os processos filhos, inclusive a sua própria, liberando o pai que até então aguardava o fim dos filhos em *wait*(). O pai por sua vez, mostra o pequeno relatório exigido pelo exercício.

4.2.1 Remoção de dados de F2

O *popF*2() é uma região crítica, assim como os demais métodos de manipulação dos campos das filas, e da mesma forma que o pushF1, utiliza do busy wait para garantir que não haja inconsistência de dados.

```
1  //Tenta retirar um elemento da fila 2 e inserir em "value"
       passado por referencia
2  int popF2 (int * value, int turn) {
3    while(queue2->turn != turn); //Espero vez da thread de p7
4
5    if (queue2->count == 0) { //Se fila vazia passo a vez (busy
       wait)
6      queue2->turn = nextTurn2(queue2->turn);
7      return -1;
8    }
9
10   *value = queue2->F2[queue2->fst];
11   queue2->fst = next(queue2->fst);
12   queue2->count--;
13
14   queue2->turn = nextTurn2(queue2->turn); //Passo a vez
15   return 0;
16 }
```

5 Processo Pai

De forma geral, praticamente tudo que o pai faz já foi discutido anteriormente no relatório e extraindo apenas o que é de sua responsabilidade temos o seguinte resultado

```
1  int main () {
2      clock_t begin = clock();
3
4    //Criacao e inicializacao das Shared Memories
5    srand(time(NULL));
6    createSharedMemory(1, SM_QUEUE1_SZ, random()); //queue1
7    createSharedMemory(2, SM_PIDS_SZ,   random()); //pids
8    createSharedMemory(3, SM_QUEUE2_SZ, random()); //queue2
9    createSharedMemory(4, SM_REPORT_SZ, random()); //report
10
11   //Inicializacao das pipes
12   createPipes();
13
```

```
14    //Cria processos filhos
15    int id = createChildren();
16
17    clock_t end = clock();
18    printResult((double)(end - begin) / CLOCKS_PER_SEC);
19
20    return 0;
21  }
```

O método *printResult*() é o responsável por mostrar o resultado final dos valores processados durante a execução do programa.

5.1 Impressão do resultado

```
1   //Imprime resultado do programa na tela
2   void printResult(double timeSpent) {
3     printf("\na)\n\t*Tempo de execucao do programa: %lf\n",
        timeSpent);
4
5     printf("\nb)\n\t*Quantidade de valores processados por p5: %d\
        n", report->counterP5);
6     printf("\t*Quantidade de valores processados por p6: %d\n",
        report->counterP6);
7
8     //Moda: Maior valor de counterEach, pois ao processar um
        elemento, p7 incrementa index dele
9     int mode = 0;
10    int higher = report->counterEach[0];
11    for (int i = 0; i <= INTERVAL+1; ++i) {
12      if (higher < report->counterEach[i]) {
13        higher = report->counterEach[i];
14        mode = i;
15      }
16    }
17    printf("\nc)\n\t*Moda: %d\n", mode);
18
19    //Min: Primeira observacao != 0 crescente
20    int min;
21    for (int i = 0; i <= INTERVAL+1; ++i) {
22      if (report->counterEach[i] > 0) {
23        min = i;
24        break;
25      }
26    }
27    printf("\t*Valor minimo: %d\n", min);
```

```
28
29    //Max: Primeira observacao != 0 descrescente
30    int max;
31    for (int i = INTERVAL+1; i >= 0; --i) {
32      if (report->counterEach[i] > 0) {
33        max = i;
34        break;
35      }
36    }
37    printf("\t* Valor maximo: %d\n", max);
38  }
```

Aqui pode-se observar todos os itens exigidos pelo relatório. Na linha 3, mostra-se na tela o tempo demandado durante a execução do programa.

Das linhas 9 à 17 é realizado o cálculo da moda, que basicamente é verificar qual é o maior valor dentro do vetor *counterEach*, pois assim basta imprimir na tela a posição em que este maior valor está, pois significa que foi incrementado mais vezes pelo processo consumidor da fila 2.

O cálculo do valor mínimo, é feito da linha 20 à 27, e apenas busca pelo primeiro valor em *counterEach* diferente de zero e imprime sua posição na tela.

Analogamente, o cálculo do valor máximo, feito dentre as linhas 30 a 37, busca o último valor em *counterEach* diferente de zero e imprime sua posição também.

6 Conclusão

Este código é extremamente interessante para o estudo de comunicação entre processos e várias outras estratégias de programação para vários problemas diferentes. Podemos listar aqui o uso de Shared Memory, Pipes, Semáforos, BusyWait, Sinais, processos multiThreads, multiprocessos, estrutura de dados, geração de números pseudo-aleatórios, dentre outros pormenores.

6.1 Resultados

A seguir podemos ver dois prints de resultados diferentes, o primeiro (Figura 4) utilizando função $nextTurn()$, e o segundo (Figura 5), utilizando a função $nextTurn2()$ debatidas durante o relatório.

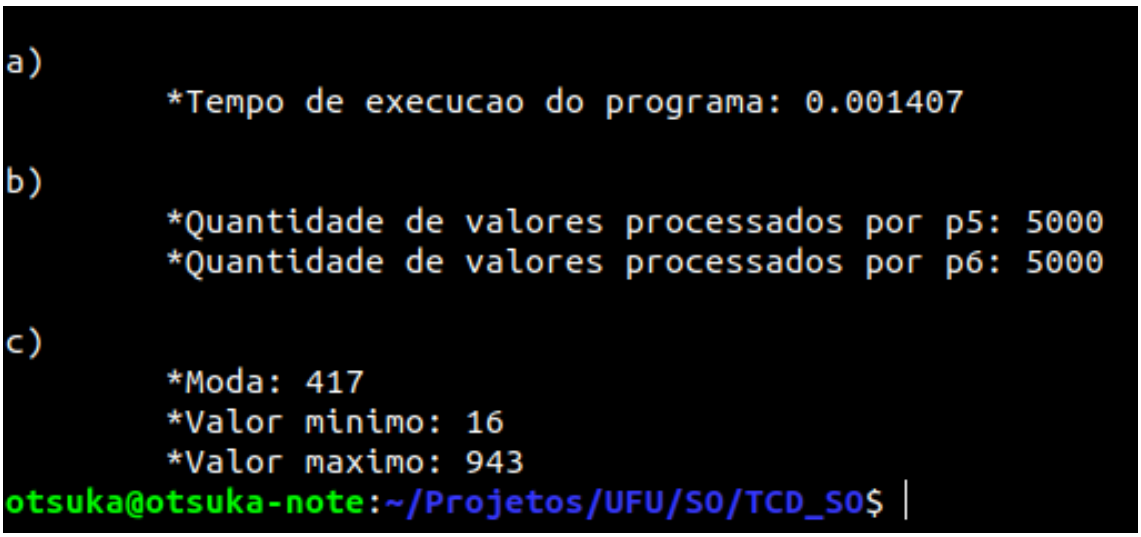

Figura 4: Utilização de nextTurn()

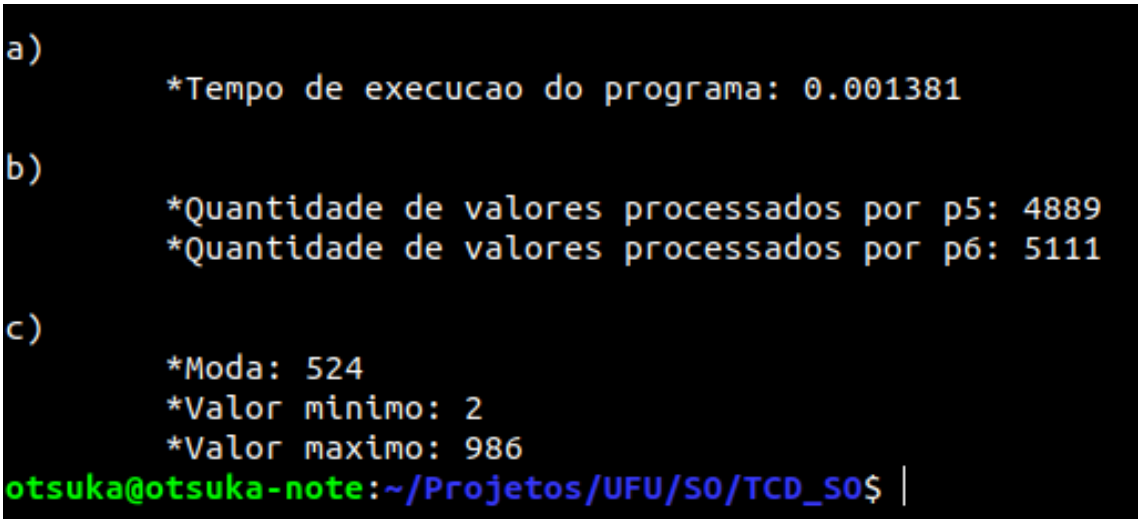

Figura 5: Utilização de nextTurn2()

6.2 Instruções de execução e código completo

O ambiente computacional utilizado para os testes foi o seguinte:

Kernel: 5.4.0-58-generic

Distro: Ubuntu Ubuntu 20.04.1 LTS

CPU: Intel® Core™ i7-8550U CPU @ 1.80GHz × 8

GPU: AMD® Iceland / Mesa Intel® UHD Graphics 620 (KBL GT2)

Memoria: 4GB

Para executar o código utiliza-se um Sistema Operacional Linux, e é necessário primeiramente ter instalado um compilador C. Nas instruções dadas a seguir, o compilador utilizado é o GCC (GNU Compiler Collection). Supondo que o arquivo fonte se chame *challengeIPC.c* e que esteja em um terminal no mesmo diretório que este arquivo, o comando utilzado para compilar o código é:

gcc challengeIPC.c -o exec -lpthread

Este comando gerará um arquivo executável de nome *exec* no mesmo diretório e para executá-lo, basta inserir o seguinte comando:

./exec

O código completo será descrito até o final do relatório.

```
1  #include <pthread.h>
2  #include <semaphore.h>
3  #include <signal.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <sys/shm.h>
7  #include <sys/syscall.h>
8  #include <sys/wait.h>
9  #include <threads.h>
10 #include <time.h>
11 #include <unistd.h>
12
13 #define gettid() syscall(SYS_gettid)
14 #define QUEUE_SZ 10 //Tamanho das filas (F1 e F2)
15 #define PIDS_SZ  8  //Tamanho do vetor com PIDs de todos os
       processos
16 #define AMOUNT_DATA 10000 //Quantidade de elementos que devem
       ser processados por p7
17 #define INTERVAL 1000
18
19 //Estrutura Fila 1
20 struct queue1_t {
21   sem_t mutex;
```

```
22    int fst, lst, count;
23    int F1[QUEUE_SZ];
24    int toggleAction; // 0: Produzir  |  1: Consumir
25    int sendSignal;   // 0: P4 nao pode receber sinal  |  1: P4
       pode receber sinal
26  };
27  typedef struct queue1_t * Queue1;
28
29  //Estrutura Fila 2
30  struct queue2_t {
31    int turn; // 0 = P5 | 1 = P6 | 2 = T1P7 | 3 = T2P7 | 4 = T3P7
32    int fst, lst, count;
33    int F2[QUEUE_SZ];
34  };
35  typedef struct queue2_t * Queue2;
36
37  //Estrutura do relatorio resultante
38  struct report_t {
39    sem_t mutex;
40    int counterP5, counterP6; //Quantidade de elementos
       processados por p5 e p6
41    int counterTotal; //Quantidade de elementos processados por p7
42    int counterEach[INTERVAL+1]; //Elementos processados por p7
43  };
44  typedef struct report_t * Report;
45
46  #define SM_QUEUE1_SZ sizeof(struct queue1_t)
47  #define SM_QUEUE2_SZ sizeof(struct queue2_t)
48  #define SM_REPORT_SZ sizeof(struct report_t)
49  #define SM_PIDS_SZ   PIDS_SZ*sizeof(int)
50  #define SM_SYNC_SZ   sizeof(sem_t)
51
52  Queue1 queue1; //Ponteiro para estrutura da fila 1 (shared
      memory)
53  Queue2 queue2; //Ponteiro para estrutura da fila 2 (shared
      memory)
54  Report report; //Ponteiro para estrutura do relatorio resultante
       (shared memory)
55  int* pids; //Vetor com PIDs de todos os processos [pai,p1,p2,p3,
      p4,p5,p6,p7] (shared memory)
56  long int thread1p4Id; //TID da thread original do P4
57  int pipe01[2];
58  int pipe02[2];
59
60  void  consumerF1();
```

```
61  void  consumerF2(int turn);
62  int   createChildren();
63  void  createPipes();
64  void  createSharedMemory (int type, int sharedMemorySize, int
        keySM);
65  void  createSemaphore (sem_t * semaphore);
66  int   next (int position);
67  int   nextTurn(int turn);
68  int   nextTurn2(int turn);
69  void* p4SignalReceiver();
70  int   popF1 (int * value);
71  int   popF2 (int * value, int turn);
72  void  printResult(double timeSpent);
73  void  producerF1();
74  void  producerF2(int process);
75  int   pushF1 (int value);
76  void  pushF2 (int value, int turn);
77  void* setF1ToConsume();
78  void  setF1ToProduce();
79  void* thread2p7();
80  void* thread3p7();
81
82  int main() {
83    clock_t begin = clock();
84
85    //Criacao e inicializacao das Shared Memories
86    srand(time(NULL));
87    createSharedMemory(1, SM_QUEUE1_SZ, random()); //queue1
88    createSharedMemory(2, SM_PIDS_SZ,   random()); //pids
89    createSharedMemory(3, SM_QUEUE2_SZ, random()); //queue2
90    createSharedMemory(4, SM_REPORT_SZ, random()); //report
91
92    //Inicializacao das pipes
93    createPipes();
94
95    //Cria processos filhos
96    int id = createChildren();
97
98    //Processo pai
99    if ( id == 0 ) {
100     clock_t end = clock();
101     printResult((double)(end - begin) / CLOCKS_PER_SEC);
102   }
103
104   //P1, P2, P3
```

```
105     else if ( id <= 3 ){
106       producerF1();
107     }
108
109     //P4
110     else if ( id == 4 ){
111       thread1p4Id = gettid(); //Defino TID da thread principal do
          p4
112
113       pthread_t thread2;
114       pthread_create(&thread2, NULL, p4SignalReceiver, NULL);
115       p4SignalReceiver();
116       pthread_join(thread2, NULL);
117     }
118
119     //P5, P6
120     else if ( id == 5 || id == 6){
121       producerF2(id);
122     }
123
124     //P7
125     else if ( id == 7 ){
126       pthread_t tid2, tid3;
127
128           pthread_create(&tid2, NULL, thread2p7, NULL);
129           pthread_create(&tid3, NULL, thread3p7, NULL);
130         consumerF2(2);
131
132       pthread_join(tid2, NULL);
133       pthread_join(tid3, NULL);
134     }
135
136     return 0;
137   }
138
139   //Argumento type:
140   //   1 = Ponteiro para F1
141   //   2 = Ponteiro para vetor de PIDs
142   //   3 = Ponteiro para F2
143   //   4 = Ponteiro para Relatorio dos Resultados
144   void createSharedMemory (int type, int sharedMemorySize, int
        keySM) {
145     key_t key = keySM;
146     void *sharedMemory = (void *)0;
147     int shmid;
```

```
148
149   shmid = shmget(key, sharedMemorySize, 0666|IPC_CREAT);
150   if ( shmid == -1 ) {
151     printf("shmget failed\n");
152     exit(-1);
153   }
154
155   sharedMemory = shmat(shmid,(void*)0,0);
156
157   if (sharedMemory == (void *) -1 ) {
158     printf("shmat failed\n");
159     exit(-1);
160     }
161
162     if (type == 1) {
163       queue1 = (Queue1) sharedMemory;
164       createSemaphore(&queue1->mutex);
165     } else if (type == 2) {
166       pids = (int *) sharedMemory;
167       *(pids) = getpid(); // pids[0] = Pid do processo pai
168     } else if (type == 3) {
169     queue2 = (Queue2) sharedMemory;
170     } else if (type == 4) {
171     report = (Report) sharedMemory;
172     createSemaphore(&report->mutex);
173     }
174 }
175
176 //Inicializa semaforo
177 void createSemaphore (sem_t * semaphore) {
178   if ( sem_init(semaphore,1,1) != 0 ) {
179     printf("Semaphore creation failed\n");
180     exit(-1);
181   }
182 }
183
184 //Inicializa ambas pipes do projeto
185 void createPipes() {
186   if ( pipe(pipe01) == -1 ){ printf("Erro pipe()"); exit(-1); }
187   if ( pipe(pipe02) == -1 ){ printf("Erro pipe()"); exit(-1); }
188 }
189
190 //Processo pai cria todos os 7 filhos
191 int createChildren() {
192   pid_t p;
```

```
193   int id;
194
195   sem_wait((sem_t*)&queue1->mutex); //Pai fecha semaforo
196   for(id=1; id<=7; id++){
197     p = fork();
198     if ( p < 0 ) {
199       printf("fork failed\n");
200       exit(-1);
201     }
202     if ( p == 0 ) {
203       sem_wait((sem_t*)&queue1->mutex); //Filhos aguardam
      liberacao do pai
204       return id;
205     }
206     *(pids+id) = p; //Pai recebe PID do filho criado e insere
      valor no vetor pids
207   }
208
209   for (int i = 0; i < 8; ++i)
210     sem_post((sem_t*)&queue1->mutex); //Pai libera todos os
      filhos
211   for (int i = 0; i < 7; i++)
212     wait(NULL); //Pai espera o p7 encerrar todos os filhos
213
214   return 0;
215 }
216
217
218 //p1, p2 e p3 produzem elementos aleatorios para F1
219 void producerF1() {
220   while(1) {
221     while(queue1->toggleAction != 0); //Controle para que nao
      produza quando p4 estiver consumindo
222
223     int response, random;
224     srand(getpid() + report->counterTotal - queue1->lst); //Seed
      para funcao random() sempre muda dessa forma
225     while(1) {
226
227       if (queue1->toggleAction == 1)
228         break; //Se a fila ja estiver sendo consumida, nao posso
      produzir
229
230       random = (rand()%INTERVAL)+1; //Gera numero aleatorio
      entre 1 e 1000
```

```
231         response = pushF1(random); //Tenta inserir na F1
232
233         if(response == 1) { //Ultimo elemento inserido na fila
234           while(queue1->sendSignal != 1); // Espero p4 estar
        pronto para receber sinal
235           while(kill(*(pids+4), SIGUSR1) == -1); //Tento enviar
        sinal para p4 consumir ate ter sucesso
236           break;
237
238         } else if (response == -1) //Fila cheia
239           break;
240       }
241     }
242   }
243
244   //Tenta inserir elemento "value" na fila "queue"
245   int pushF1 (int value) {
246     sem_wait((sem_t*)&queue1->mutex);
247
248     if (queue1->count == QUEUE_SZ) { //Caso fila cheia
249       sem_post((sem_t*)&queue1->mutex);
250       return -1;
251     }
252
253     //Insiro novo elemento
254     queue1->F1[queue1->lst] = value;
255     queue1->lst = next(queue1->lst);
256     queue1->count++;
257
258     //Caso insercao encheu a fila, flagSendSignal == 1
259     int flagSendSignal = (queue1->count == QUEUE_SZ);
260
261     sem_post((sem_t*)&queue1->mutex);
262     return flagSendSignal;
263   }
264
265   //Calcula proxima posicao livre para realizar a insercao
266   int next (int position) {
267     return (position + 1) % QUEUE_SZ; //Insercao circular
268   }
269
270   //Threads de p4 aguardam envio do sinal dos produtores
271   void* p4SignalReceiver() {
272     while(1) {
273       if (gettid() == thread1p4Id) {
```

```
274         signal(SIGUSR1, (__sighandler_t) setF1ToConsume);
275         queue1->sendSignal = 1; //Pronto para receber signal
276       }
277
278       while(queue1->toggleAction != 1); //Enquanto a fila nao
        estiver pronta para consumo, nao fazer nada
279
280       consumerF1();
281       setF1ToProduce();
282     }
283   }
284
285   //Bloqueia produtores de continuarem produzindo ao retirar
        elementos da F1
286   void* setF1ToConsume() {
287     queue1->toggleAction = 1; //Nao produza mais! F1 pode ser
        consumida
288   }
289
290   //Libera produtores para produzir elementos para F1
291   void setF1ToProduce() {
292     queue1->toggleAction = 0; //Produza mais! F1 nao pode ser
        consumida
293   }
294
295   //p4 (dualThread) consome F1 e escreve elementos nas pipes
296   void consumerF1() {
297     int response, value, resp;
298     while(1) {
299       response = popF1(&value);
300       if (response == 0) { //Se consegui retirar elemento
301
302         if (thread1p4Id == gettid()){ //Thread original n o
        permite envio de sinal
303           queue1->sendSignal = 0;
304           resp = write(pipe01[1], &value, sizeof(int)); //Envio
        para pipe01
305         } else {
306           resp = write(pipe02[1], &value, sizeof(int)); //Envio
        para pipe02
307         }
308
309         if(resp < 0) {
310           printf("Erro na escrita do pipe\n");
311           break;
```

```
312         }
313       } else if (response == -1)
314         break;
315     }
316   }
317
318   //Tenta retirar um elemento da fila 1 e inserir em "value"
        passado por referencia
319   int popF1 (int * value) {
320     sem_wait((sem_t*)&queue1->mutex);
321
322     if (queue1->count == 0) { //Caso fila ja vazia retorno erro
323       sem_post((sem_t*)&queue1->mutex);
324       return -1;
325     }
326
327     *value = queue1->F1[queue1->fst];
328     queue1->fst = next(queue1->fst);
329     queue1->count--;
330
331     sem_post((sem_t*)&queue1->mutex);
332     return 0;
333   }
334
335   //Le elementos das pipes e insere em F2
336   void producerF2(int process) {
337
338     int value, resp, response;
339
340     while(1) {
341
342       if (process == 5)
343         resp = read(pipe01[0], &value, sizeof(int)); //Tentativa
          de leitura de pipe01
344       else if (process == 6)
345         resp = read(pipe02[0], &value, sizeof(int)); //Tentativa
          de leitura de pipe02
346
347       if(resp == -1) {
348         printf("Erro na leitura do pipe0%d\n", process-4);
349         break;
350       } else if (resp > 0) {
351         pushF2(value, process-5); //Tento colocar na F2
352       }
353     }
```

```
354 }
355
356 //Tenta inserir elemento "value" na fila "queue"
357 void pushF2 (int value, int turn) {
358   while (queue2->turn != turn); //Aguardo minha vez (busy wait)
359
360   if (queue2->count == QUEUE_SZ) {
361     queue2->turn = nextTurn2(queue2->turn);
362     return;
363   }
364
365   //Se p7 ainda nao processou AMOUNT_DATA incrementa contadores
      de p5 e p6
366   sem_wait((sem_t*)&report->mutex);
367   if (report->counterTotal < AMOUNT_DATA)
368     (turn == 0) ? report->counterP5++ : report->counterP6++;
369   sem_post((sem_t*)&report->mutex);
370
371   queue2->F2[queue2->lst] = value;
372   queue2->lst = next(queue2->lst);
373   queue2->count++;
374
375   queue2->turn = nextTurn2(queue2->turn);
376   return;
377 }
378
379 //Calcula qual thread vai processar regiao critica. Sequencial (
      busy wait)
380 int nextTurn(int turn) {
381   return (turn + 1)%5;
382 }
383
384 //Calcula qual thread vai processar regiao critica. Aleatorio (
      busy wait)
385 int nextTurn2(int turn) {
386   return rand() % 5;
387 }
388
389 //Define segunda thread como vez 3
390 void * thread2p7() {
391   consumerF2(3);
392 }
393
394 //Define terceira thread como vez 4
395 void * thread3p7() {
```

```
396   consumerF2(4);
397 }
398
399 //Tento consumir valores de F2
400 void consumerF2(int turn) {
401   int value, response;
402   while(1) {
403
404     response = popF2(&value, turn); //Tento retirar elemento da
      fila
405
406     if (response == 0) { //Se consegui retirar elemento
407       sem_wait((sem_t*)&report->mutex);
408
409       report->counterTotal++; //Incrementa quantidade de
      elementos processados por p7
410       report->counterEach[value]++; //Incrementa 1 na posi   o
      correspondente ao elemento processado por p7
411       if (report->counterTotal >= AMOUNT_DATA) //Se processei
      quantidade total de elementos que desejo
412         for (int i = 1; i <= 7; ++i)
413           kill(*(pids+i), SIGTERM); //Encerro execucao dos
      processos exceto pai
414
415       printf("Elemento %d retirado de F2\n", value);
416       sem_post((sem_t*)&report->mutex);
417     }
418   }
419 }
420
421 //Tenta retirar um elemento da fila 2 e inserir em "value"
      passado por referencia
422 int popF2 (int * value, int turn) {
423   while(queue2->turn != turn); //Espero vez da thread de p7
424
425   if (queue2->count == 0) { //Se fila vazia passo a vez (busy
      wait)
426     queue2->turn = nextTurn2(queue2->turn);
427     return -1;
428   }
429
430   *value = queue2->F2[queue2->fst];
431   queue2->fst = next(queue2->fst);
432   queue2->count--;
433
```

```
434   queue2->turn = nextTurn2(queue2->turn); //Passo a vez
435   return 0;
436 }
437
438 //Imprime resultado do programa na tela
439 void printResult(double timeSpent) {
440   printf("\na)\n\t*Tempo de execucao do programa: %lf\n",
       timeSpent);
441
442   printf("\nb)\n\t*Quantidade de valores processados por p5: %d\
       n", report->counterP5);
443   printf("\t*Quantidade de valores processados por p6: %d\n",
       report->counterP6);
444
445   //Moda: Maior valor de counterEach, pois ao processar um
       elemento, p7 incrementa index dele
446   int mode = 0;
447   int higher = report->counterEach[0];
448   for (int i = 0; i <= INTERVAL+1; ++i) {
449     if (higher < report->counterEach[i]) {
450       higher = report->counterEach[i];
451       mode = i;
452     }
453   }
454   printf("\nc)\n\t*Moda: %d\n", mode);
455
456   //Min: Primeira observacao != 0 crescente
457   int min;
458   for (int i = 0; i <= INTERVAL+1; ++i) {
459     if (report->counterEach[i] > 0) {
460       min = i;
461       break;
462     }
463   }
464   printf("\t*Valor minimo: %d\n", min);
465
466   //Max: Primeira observacao != 0 descrescente
467   int max;
468   for (int i = INTERVAL+1; i >= 0; --i) {
469     if (report->counterEach[i] > 0) {
470       max = i;
471       break;
472     }
473   }
474   printf("\t*Valor maximo: %d\n", max);
```

```
475  }
```