Pedro Henrique Marra Araújo (12011BCC008)

Interprocess Communication

Uberlândia, Minas Gerais
2022

Sumário

Repositório

Todos os códigos foram disponibilizados em um repositório no GitHub, que pode ser acessado clicando aqui.

Fila Estática

Foi usado, nesse trabalho, um TAD de fila estática usando contator. Nesse TAD, encontram-se as funções `cria_fila`, `fila_vazia`, `fila_cheia`, `insere_fim` e `remove_ini` de implementação canônica, portanto não será comentada nesse relatório (até mesmo porque não é esse o propósito deste).

Comentários Gerais

Entre os arquivos, a implementação de todas restrições/comunicações entre processos explicitadas pela especificação da atividade foram feitas no arquivo `main.c`. Neste, toda a criação das *shared memories*, dos *pipes* e da lógica das rotinas de cada processo estão todos comentados minunciosamente. Dessa forma, esse relatório servirá para explicitar todas as escolhas que tomei no código para garantir que todas as restrições pedidas foram alcançadas. Assim, abaixo citarei trechos da restrição e partes do código que as garantem, comentando os motivos de cada escolha e qual foi meu processo de criação para chegar no código que escrevi.

(i) *"Os processos P1, P2 e P3 são singlethreads"*: todos os processos (exceto o processo principal da *main* foram criados por `fork` no `fork` e depois à cada processo foi delegado uma rotina específica (`if-then-elses` abaixo).

```
for(processo = 1; processo <= PROCS_NUMBER; processo++)
    if((pids[processo-1] = fork()) == 0) break;

if(processo == 1) process_4();
else if(processo == 2) process_123();
else if(processo == 3) process_123();
else if(processo == 4) process_123();
```

```
else if(processo == 5) process_56(P5_NUMBER, pipe_1,
↪  &f2->n_p5);
else if(processo == 6) process_56(P6_NUMBER, pipe_2,
↪  &f2->n_p6);
else process_7();
```

Veja que o primeiro processo é o P4 e nele, antes de delegar a rotina `process_4`, eu guardei o PID de P4 na *shared memory* da F1. Isso, por causa de *(ii)*. Invoquei `process_7` por último, pois preciso que a cópia do *array* dos PIDs desse processo tenha os PIDs de todos os processos do `fork`, pois preciso matar todos eles, como descrito em *(xiii)*.

(ii) *"Essa área compartilhada deve ser usada como uma fila (F1) do tipo FIFO (first-in-first-out), a qual armazena valores do tipo inteiro. A capacidade da fila é de 10 valores."*: a única restrição da *shared memory* de F1 era *(vii)* e que deveria ser uma fila do tipo FIFO (implementada por `f1->fila`) com capacidade máxima de 10 valores. Assim, validadas essas restrições, como há a restrição do sinal (iv), armazenei também o PID de P4 na *shared memory* de F1 para que seja facilmente mandado esse sinal (por `kill(f1->p4, SIGUSR1)` em P1, P2 e P3).

(iii) *"Os processos P1, P2 e P3 são produtores e o processo P4 é consumidor nessa comunicação."*: a produção do número randômico é gerado por `r = 1 + (rand() % MAX_N)`.

(iv) *"P4 sempre aguarda um sinal (SIGUSR1) para consumir dados da F1. Esse sinal é enviado para P4 quando F1 estiver cheia, ou seja, com 10 valores. O sinal deve ser enviado para P4 pelo processo que inseriu o décimo valor em F1. "*: como foi somente especificado que esse sinal deveria ser dado pelo processo que encheu F1 (P1, P2 ou P3), porém não foi especificado como ele deveria ser tratado, eu criei um chaveamento da F1 por meio de `f1->pronta`, que significa que F1 está pronta para P4 consumir todos seus elementos (de cheia para totalmente vazia). Assim, quando um dos processos produtores manda o seguinte sinal para P4:

```
if(!f1->pronta && insere_fim(&f1->fila, r) &&
↪  fila_cheia(&f1->fila)){
   kill(f1->p4, SIGUSR1);
```

```
    continue;
}
```

Esse sinal é tratado por P4 pela função `trigger_p4`, que muda o chaveamento da F1 para que P4 esteja pronta para consumir da fila. Perceba que esse `if` só ocorre quando P4 não está pronta para consumir, é possível inserir um elemento e então a F1 fica cheia, ou seja, essa foi a inserção do décimo elemento. Portanto, mandar sinal para P4 por SIGUSR1.

(v) *"Note que neste caso F1 somente receberá valores após P4 retirar o último (décimo) valor de F1, deixando-a vazia para receber novos valores."*: nesse caso, simetricamente ao chaveamento comentado acima, se F1 não está pronta para P4 consumir os elementos, então P1, P2 e P3 devem inserir elementos em F1. Para isso, como esse sinal não foi especificado, preferi por fazer, analogamente, de forma simétrica, pela utilização do `f1->pronta`. Isto é, se, em `process_123`, é aberto o semáforo, porém F1 está pronta para P4 retirar elementos, fecha-se o semáforo e não faz nada (não insere elemento). Esse chaveamento simétrico é realizo em `process_123` por:

```
if(!f1->p4 || f1->pronta) {
    sem_post(&f1->sem);
    continue;
}
```

(vi) *"Os valores inteiros inseridos em F1 devem ser gerados de forma aleatória na faixa de 1 até 1000."*: realizado por `r = 1 + (rand() % MAX_N)`, tal que `MAX_N` $= 1000$.

(vii) *"O acesso a F1 deve ocorrer de forma exclusiva utilizando o mecanismo de semáforo entre os processos envolvidos."*: sempre que é usado F1, abre-se o semáforo por `sem_wait(&f1->sem)` e fecha-se assim que não é mais utilizado F1 por `sem_post(&f1->sem)`.

(viii) *"O processo P4 possui duas threads (t1 e t2), onde ambas as threads retiram valores de F1 e enviam para P5 (t1) e P6 (t2), respectivamente, utilizando o mecanismo de pipe."*: em `process_4`, cria-se uma thread secundária `pthread_create(&tid2, NULL, threads_4, pipe_2)`, onde

é passado o endereço do segundo pipe. A thread principal de P4 recebe, também, a rotina `threads_4(pipe_1)`, porém com o primeiro pipe.

(ix) *"Os processos P5 e P6 ao receberem valores de P4 os enviam para P7 utilizando da fila F2, também implementada como shared memory."*: em `process_56`, lê do respectivo pipe, caso sucesso, ainda faltam valores para serem lidos por P7 e F2 não está cheia, insere o elemento lido do pipe nela:

```
if(read(pipe[0], &elem, sizeof(int)) <= 0) f2->turn =
↪  next_thread();

if(f2->n_p7 == N_NUMBER) relatorio();

if(!fila_cheia(&f2->fila)){
    insere_fim(&f2->fila, elem);
    (*counter)++;
}
```

(x) *"Neste caso, o controle ao acesso de F2 deve ser implementado usando um mecanismo de exclusão mútua baseado em espera ocupada (busy wait)."*: como em `process_56` é passado a etiqueta do processo (para ver se é P5 ou P6) e, em `threads_7`, é passado a etiqueta da thread de P7 (para ver se é a thread principal ou as outras duas criadas por `pthread_create`), a espera ocupada é facilmente feita em `process_56` e `threads_7` respectivamente por:

```
while(f2->turn != p_number){} // Em process_56

while(f2->turn != t){} // Em threads_7
```

(xi) *"Diferente de F1, a fila F2 deve ser consumida na medida em que entram novos valores."*: em `process_56`, se foi possível inserir elemento em F2, passo a vez para uma das threads de P7 (para balancear as inserções e as remoções, para evitar que F2 fique muito tempo vazia ou cheia). Isso é feito por `f2->turn = next_thread_consumer()` no final da rotina. Essa função escolhe uma das threads de forma randômica:

```
int next_thread_consumer(){
    int r = rand() % 3;
```

```
    if(r == 0) return P7_NUMBER;
    if(r == 1) return P7_T2_NUMBER;
    else       return P7_T3_NUMBER;
}
```

(xii) *"Se a fila estiver cheia (máximo 10 valores), então os processos P5 e P6 devem aguardar a retirada de ao menos um elemento da fila, por P7, para inserirem novos valores. Todas as três threads de P7 retiram valores de F2 e os imprimem na tela."*: em `threads_7`, se a F2 é vazia, escolhe-se, randomicamente, P5 ou P6 para produzir mais elementos; caso contrário, remove elemento, imprime a remoção na tela, incrementa sua frequência e escolhe um novo produtor para a F2 (para balancear as remoções e inserções em F2, análogo à *(xi)*):

```
if(fila_vazia(&f2->fila)){
    f2->turn = next_process_producer();
    continue;
}

remove_ini(&f2->fila, &elem);
f2->freq[elem-1]++;
printf("P7(T\%d): removeu \%d de F2 - (\%d-ésimo
↪  elemento).\n", t_etiqueta, elem, f2->n_p7 + 1);
f2->n_p7++;

f2->turn = next_process_producer();
```

(xiii) *"Após P7 imprimir 10000 valores, o programa deve imprimir o seguinte relatório: [...]"*: sempre que passarem os 10000 elementos (`if(f2->n_p7 == N_NUMBER)`), é invocada a rotina `relatorio()`; essa busca no vetor de frequências do elementos o com menor, a moda e o com menor frequência, calcula o tempo de execução em milissegundos, imprime na tela os resultados, fecha todas as estruturas usadas no programa (os pipes e as *shared memories*) e então mata todos os processos.

(xiv) *"Tempo total de execução do programa."*: para calcular o tempo de execução, utilizei a estrutura `struct timespec` da biblioteca `time.h`, a qual, quando é invocada `int clock_gettime(clockid_t clockid,`

`struct timespec *tp)`, recebe o tempo em segundos e nanossegundos. Assim, crio duas dessas estruturas, uma no começo do programa e outra no final, calculo a diferença e então imprimo-a:

```
clock_gettime(CLOCK_REALTIME, &ti); // Tempo inicial
// Programa
clock_gettime(CLOCK_REALTIME, &tf); // Tempo final
```

Resultados

Esse código é correto e bastante eficiente, pois fiz os seguintes testes e obtive os seguintes resultados.

Comentei a linha a impressão dos elementos sendo retirados da F2 para que somente o relatório final seja impresso. Compilei os arquivos como descrito no `README.txt` e executei o seguinte comando:

```
for i in 1..2000; do ./pedroSO; done > resultados.txt
```

Ou seja, executei o programa duas mil vezes e o resultado pode ser visualizado clicando aqui. Perceba que em média, nos meus testes, cada iteração executou em $15ms$. Além disso, que os resultados se repetem bastante, isso ocorre, provavelmente, por causa da semente utilizada por cada iteração do algoritmo pseudoaleatório (`srand(time(NULL))` é muito próxima uma da outra, tornando-as equivalentes. Mas, para testes não automatizados, os resultados são mais imprevisíveis, aparecendo mais diversos resultados (tanto de menor, maior e moda dos números).