Wojciech Celej
Nr. albumu: 271248

Ćwiczenie 4
Synchronizacja procesów z wykorzystaniem monitorów

I. Zadanie do zrealizowania

Bufor n-elementowy FIFO. Jest jeden producent i trzech konsumentów (A, B, C). Producent produkuje jeden element, jeżeli jest miejsce w buforze. Element jest usuwany z bufora, jeżeli zostanie przeczytany przez albo obu konsumentów A i B, albo przez obu konsumentów B i C. Konsument A nie może przeczytać elementu, jeżeli został on już przez niego wcześniej przeczytany, albo został przeczytany przez konsumenta C i na odwrót. Konsument B również nie może wielokrotnie czytać tego samego elementu. Ponadto, żaden konsument nie może dwa razy z rzędu usunąć elementu z bufora.

II. Proponowane rozwiązanie

1. Kolejka

Będzie to tablica o wymiarze N. Zaimplementowana będzie jako struktura danych w ramach monitora *MyMonitor:*

- usunięcie elementu będzie polegało na zwiększeniu indeksu el. początkowego o 1 (gdy dojdzie do ostatniego indeksu, wtedy przeskoczy na 0)
- dodanie nowego elementu będzie polegało na zwiększeniu indeksu el. początkowego o 1 (gdy dojdzie do ostatniego indeksu, wtedy przeskoczy na 0)
- semafory zliczające zapobiegną przepełnieniu tej tablicy

2. Klasa *MyMonitor*

Klasa ta publicznie dziedziczyć będzie od klasy monitor, zawierającej operacje signal() i wait() oraz chroniące metody tej klasy funkcjami enter() i leave () a także typ *Condition.* Zawartość klasy MyMonitor:

- int *buffer – wskaźnik na tablicę składującą elementy kolejki
- int buff_beg, buff_end, buff_size, num_el – do obsługi kolejki
- 2 zestawy flag:
    - unsigned char read[NUM_FLAGS] – pamiętają, czy dany wątek czytał ostatni element buffora
    - unsigned char flag[NUM_FLAGS] – pamiętają, czy dany wątek usuwał jako ostatni
- 3 zmienne warunkowe
    - Condition full – wątek producenta zatrzymuje się na niej, kiedy buffor jest pełny
    - Condition emptyAC – wątek konsumenta A/C zatrzymuje się na niej, gdy buffor jest pusty lub gdy konsument A/C czytał już dany element
    - Condition emptyB – analogicznie dla konsumenta B jak dla A/C

- Prywatne metody klasy:
  - enter_item() – wywoływana w zadaniu producenta, wstawia nowy element do bufora
  - read_item(unsigned char proc_id) – wywoływana w zadaniach konsumentów – czyta dany element z bufora i ustawia flagę read[proc_id] na 1
  - remove_item(unsigned char proc_id) - wywoływana w zadaniach konsumentów –usuwa dany element z bufora i ustawia flagi read na 0 i flagę flag[proc_id] na 1
  - wait_rand_time(int a, int b) – czekaj k sekund, gdzie k należy od a do b

- Publiczne metody klasy: będą to metody bezpośrednio wykonujące zadania wątków konsumentów i producentów. Do obsługi zadania konsumenta A i C użyję tej samej metody (można tak to rozwiązać, ponieważ zadania tych konsumentów są względem konsumenta B symetryczne – jeżeli czytał A, nie może czytać C – czyli flaga jest już ustawiona; żeby usunął A lub C – musi wcześniej odczytać B). Wątki A/C będą rozróżniane poprzez argument przekazywany do metody z poziomu main.

3. Struktura programu

Odpowiednie 3 wątki konsumentów będą wywoływane w funkcji main. Wątkiem głównym będzie wątek producenta. Producent produkuje n elementów, gdzie n zdefiniowane będzie jako makro. Po wyprodukowaniu wszystkiego, co miał wyprodukować, czeka 5 sekund i kończy pracę programu. Wątki konsumentów nie będą dołączane, przez co po wywołaniu return() przez wątek główny (producenta), program i wraz z nim wszystkie pozostałe wątki konsumentów zakończą działanie.

Deklaracja funkcji obsługiwanych przez wątki i obiektu monitora w *main.cpp:*

```
MyMonitor buffer(BUFF_SIZE);

void *consumerA(void *arg)
{
 for(;;)
 {
   wait_rand_time(2,3);
   buffer.consumerAC_task('A');
 }
}

void *consumerB(void *arg)
{
 wait_rand_time(3,4);
 for(;;) buffer.consumerB_task();
}
```

```
void *consumerC(void *arg)
{
  wait_rand_time(2,3);
  for(;;) buffer.consumerAC_task('C');
}

void producer(int n)
{
  for(int i=0; i<n; i++)
  {
    wait_rand_time(0,1);
    buffer.producer_task();
  }
  return;
}
```

4. Kody metod wywoływanych przez odpowiednie wątki

```
void consumerAC_task(char proc_id)
 {
   enter();
   if(num_el == 0 || read[CONS_AC]) wait(emptyAC);
   if(num_el != 0)
   {
     if(read[CONS_AC] != 1) cout <<"Consumer " <<proc_id <<" read item " <<read_item(CONS_AC)
<<endl;
     if(read[CONS_B])
       {
         if(flag[CONS_AC] != 1)
         {
           cout <<"Consumer " <<proc_id <<" remove item " <<remove_item(CONS_AC) <<endl;
           if(num_el == buff_size - 1) signal(full);
           signal(emptyB);
         }
         else signal(emptyB);
       }
     else signal(emptyB);
   }
   else signal(full);
   leave();
 }
```

```
void consumerB_task()
{
  enter();
  if(num_el == 0 || read[CONS_B]) wait(emptyB);
  if(num_el != 0)
  {
    if(read[CONS_B] != 1) cout <<"Consumer B read item " <<read_item(CONS_B) <<endl;
    if(read[CONS_AC])
      {
        if(flag[CONS_B] != 1)
        {
          cout <<"Consumer B remove item " <<remove_item(CONS_B) <<endl;
          if(num_el == buff_size - 1) signal(full);
          signal(emptyAC);
        }
        else signal(emptyAC);
      }
    else signal(emptyAC);
  }
  else signal(full);
  leave();
}
```

III. Testy

Odpowiednie metody sygnalizować będą swoje bieżące poczynania w konsoli. Proponuję dwa testy: pierwszy z szybkim producentem i wolnymi konsumentami oraz drugi z szybkimi konsumentami i wolnym producentem. Oba testy pozwolą nam stwierdzić poprawność rozwiązania – między innymi czy procesy konsumentów nie odwołują się do pustego buffora, a producent nie chce zapisywać do pustego buffora. Wątki będą budzić się do życia co pewien losowy czas (z ustalonego wcześniej przedziału).