Przetwarzanie współbieżne. Programowanie równoległe i rozproszone
Laboratorium 8

Cel:

- doskonalenie umiejętności realizacji synchronizacji w języku C za pomocą zmiennych warunku oraz w programach obiektowych w Javie za pomocą narzędzi pakietu *java.util.concurrency*

Zajęcia:

1. Utworzenie katalogu roboczego (np. *lab_8*) i podkatalogu (np. *lab_8_bariera)*.
2. Na podstawie materiałów z wykładu utworzenie i zaimplementowanie (w osobnym pliku źródłowym *bariera.c)* algorytmu realizującego funkcję bariery: dowolny wątek może zakończyć realizację funkcji „bariera", dopiero po wywołaniu tej funkcji przez pozostałe wątki. Możliwy sposób rozwiązania:
   a) utworzenie zmiennej globalnej zliczającej liczbę wątków, które wywołały funkcję „bariera"
   b) związanie z tą zmienną zmiennej warunku i odpowiedniego muteksa
   c) opracowanie mechanizmu funkcjonowania bariery z wykorzystaniem powyższych zmiennych
   d) sugerowany interfejs:
      void bariera_init(int); - inicjowanie liczbą wątków w programie głównym
      void bariera(void); - wywołanie przez wątek
   e) przy implementacji zmienne globalne mogą zostać zastąpione przez statyczne
3. Uruchomienie z wykorzystaniem pliku *Makefile*
4. Przetestowanie działania funkcji "bariera" poprzez skompilowanie z dostarczonym programem "main.c" (korzystając z dostarczonego pliku Makefile) i uruchomienie – sprawdzenie niepoprawności działania bez bariery i poprawności działania z barierą
   a) uwaga na zapewnienie poprawności przy wielokrotnym wywołaniu funkcji bariera - co należy zrobić wewnątrz funkcji *bariera*, żeby umożliwić takie działanie? **(ocena)**
5. Utworzenie podkatalogu (np. *lab_8_pthreads*), rozpakowanie paczki *CzytPis_Pthreads.tgz* i uruchomienie kodu.
6. Przeanalizowanie pseudokodu monitora Czytelnia na slajdach z wykładów (wykład 6) oraz struktury kodu w paczce *CzytPis_Pthreads.tgz* (funkcja *my_read_lock_lock* ma odpowiadać funkcji *chcę_czytać* – czyli protokołowi wejścia do sekcji krytycznej dla czytelnika, *my_read_lock_unlock* protokołowi wyjścia i podobnie dla pisarza - *my_write_lock_lock* i *my_write_lock_unlock*)
7. Wykrycie błędu w kodzie:
   a) zrealizowanie pierwszego kroku implementacji protokołów wejścia i wyjścia do sekcji krytycznych pisania i czytania, w postaci śledzenia liczby czytelników i pisarzy w czytelni (wchodząc każdy zwiększa odpowiedni licznik, wychodząc zmniejsza)
      - liczby czytelników i pisarzy powinny być polami składowymi struktury z pliku *czytelnia.h*
   b) umieszczenie w procedurach pisania i czytania (*pisze(), czytam()* w pliku czytelnia.c) sprawdzenia warunków poprawnych wartości aktualnych liczb pisarzy i czytelników (w sekcji krytycznej!) oraz przerwania działania gdy warunki nie są spełnione
      - dobrze jest także dodać wydruki aktualnych wartości liczb
   c) uruchomienie kodu - z komunikatami o ewentualnych błędach **(ocena)**
8. Na podstawie pseudokodu monitora *Czytelnia* poprawienie kodu z paczki *CzytPis_Pthreads.tgz,* tak aby poprawnie rozwiązywać problem czytelników i pisarzy wykorzystując zmienne warunku
   a) uwaga 1: implementowane funkcj muszą w całości realizować wzajemne wykluczanie (tak jak w monitorze) - na początku zamykać mutex i zwalniać go tuż przed końcem
   b) uwaga 2: w bibliotece *pthreads* nie ma odpowiednika funkcji *empty()* z monitora - należy ją zastąpić sprawdzeniem wartości odpowiednio zaimplementowanej dodatkowej zmiennej
9. Przetestowanie działania kodu – w tym poprawności (jak w p. 3). Testowanie, zgodnie z wzorcem

w pliku *czyt_pis.c*, ma polegać na stworzeniu kilku wątków realizujących funkcje czytelnika i pisarza, które w nieskończonej (lub odpowiednio długiej) pętli będą kolejno realizowały swoje funkcje czytania i pisania z prawidłową realizacją wzajemnego wykluczania. **(ocena)**

---------- 3.0 ----------

10. Utworzenie podkatalogu (np. *lab_8_read_write_locks*), skopiowanie paczki *CzytPis_Pthreads.tgz*, rozpakowanie i ponowne uruchomienie kodu.
11. Zmodyfikowanie kodu, tak, żeby korzystać z interfejsu zamków do odczytu i zapisu *Pthreads* (*pthread_rwlock_rdlock, pthread_rwlock_wrlock, pthread_rwlock_unlock*):
    a) program *czyt_pis.c* powinien pozostać bez zmian
    b) struktura *czytelnia_t* musi teraz zawierać, zamiast zmiennych zliczających czytelników i pisarzy (które mogą pozostać w celu testowania poprawności kodu), zmienną typu *pthread_rwlock_t*
    c) w pliku *czytelnia.c* zamiast własnej implementacji zamków odczytu i zapisu powinny znajdować się wywołania odpowiednich funkcji *Pthreads*
       - każda z funkcji: *my_read_lock_lock, my_read_lock_unlock, my_write_lock_lock* i *my_write_lock_unlock* ma wywoływać właściwe funkcje dla zmiennej typu *pthread_rwlock_t*
    d) po implementacji kod powinien zostać przetestowany – najlepiej pozostawić zliczanie czytelników i pisarzy (chronione dodatkowym mutexem) i wypisywać aktualne liczby, z ewentualnymi komunikatami o błędach **(ocena)**
12. Modyfikacja funkcji *bariera* tak, aby umożliwić istnienie wielu barier w jednym programie:
    - wprowadzenie typu struktury zawierającej parametry bariery
    - modyfikacja inicjowania bariery, tak aby tworzyła nową barierę (nową strukturę)
    - modyfikacja wywołania funkcji bariery, tak aby przyjmowała jako argument konkretną barierę (strukturę) **(ocena)**

---------- 4.0 ----------

1. Utworzenie podkatalogu roboczego (np. *lab_8_java*)
2. W podkatalogu, na podstawie pseudokodu monitora *Czytelnia z*e slajdów na wykładzie, napisanie w Javie klasy *Czytelnia* pozwalającej na rozwiązanie problemu Czytelników i Pisarzy. Klasa powinna mieć metody *chcę_czytać, czytam, koniec_czytania, chcę_pisać, piszę, koniec_pisania* (oczywiście nazwy można dobrać dowolnie) oraz odpowiednie prywatne atrybuty pozwalające na poprawną (gwarantującą bezpieczeństwo i żywotność) implementację
   1. w kodzie należy wykorzystać interfejs *java.util.concurrent.locks.** i typy *Lock* oraz *Condition*. Należy użyć konstruktorów *ReentrantLock()* oraz *Lock.newCondition()* (oraz funkcji *lock.hasWaiters(condition)* do sprawdzenia czy kolejka uśpionych na danej zmiennej warunku wątków jest pusta).
3. Przetestowanie działania klasy poprzez stworzenie klasy testującej (z funkcją *main*) oraz kilku obiektów klas Czytelnik i Pisarz, które w nieskończonej (lub odpowiednio długiej) pętli będą kolejno realizowały swoje funkcje czytania i pisania. Jak zwykle kod powinien być wyposażony w wykrywanie i obsługę ewentualnych błędów. Do stworzenia tych klas można wykorzystać odpowiednio zmodyfikowany kod z paczki *ProdKons.tgz*
4. Zaimplementować w Javie mechanizm bariery – można posłużyć się uproszczonym interfejsem: synchronized, wait(), signal();

---------- 5.0 ----------

Warunki zaliczenia:

1. Obecność na zajęciach i wykonanie co najmniej kroków 1-9
2. Oddanie sprawozdania o treści i formie zgodnej z regulaminem ćwiczeń laboratoryjnych - z krótkim opisem zadania (cel, zrealizowane kroki, wnioski), kodem źródłowym programów oraz wydrukami wyników (wydruki wklejone jako obrazy z identyfikacją osoby przeprowadzającej obliczenia – zgodnie z regulaminem laboratoriów)