Programowanie równoległe. Przetwarzanie równoległe i rozproszone.

Laboratorium 4

Cel: opanowanie umiejętności pisania programów z synchronizacją wątków.

Kroki:

1. Utworzenie katalogu roboczego (np. *lab_4*)
2. Zaprojektowanie symulacji pubu z następującą specyfikacją:
   a) W pubie jest *l_kf* nierozróżnialnych kufli 1 L oraz *l_kl* klientów.
   b) Klienci są reprezentowani przez wątki.
   c) Każdy klient pragnie wypić *ile_musze_wypic* kufli piwa.
   d) W pubie czekają przygotowane puste kufle do pobrania przez klientów.
   e) Napełnienie jednego kufla trwa kilka sekund (jest jeden kran!).
   f) Klient opróżnia kufel w kilkanaście sekund.
   g) Po wypiciu każdego litra klient oddaje stary kufel i pobiera nowy
   h) Po wypiciu *ile_musze_wypic* litrów piwa klient opuszcza pub.
   i) Pub otwarty jest do ostatniego klienta (ale nie wpuszcza nowych).
   j) Każdy klient podczas pobytu w pubie informuje (wypisując na ekranie) co robi w danej chwili
3. W pierwszej wersji należy zaprojektować i zaimplementować symulację pubu używając wyłącznie muteksów i procedury *pthread_mutex_lock* (sekundy trzeba zamienić na mikrosekundy – *usleep* lub *nanosleep* zamiast *sleep* – dla przyspieszenia testowania programu). Parametrami symulacji są: liczba klientów oraz liczba kufli.
   a) Punktem wyjścia powinien być program *pub_sym_1.c* . Program nie jest poprawny ponieważ w momencie kiedy klient pobiera kufel może dojść do wyścigu z innym klientem, który w tym samym momencie chce pobrać kufel. Należy doprowadzić do sytuacji kiedy pobieranie kufla jest bezpieczne (nie ma wyścigu).

      Wskazówka: rozpocząć od sytuacji kiedy liczba kufli przewyższa liczbę klientów i od rozwiązania problemu: jak ma zachowywać się klient (czyli nowo utworzony wątek), żeby nigdy nie pobierać kufla równocześnie z innym klientem/wątkiem (rozwiązanie problemu wymaga, aby najpierw rozstrzygnąć jak maja być reprezentowane kufle – uwaga kufle są nierozróżnialne).
   b) Mając rozwiązany (prosty) problem dla liczby kufli większej od liczby klientów, należy wprowadzić sprawdzenie poprawności działania:
      - w żadnym momencie nie może być tak, że liczba kufli pobranych jest większa od liczby kufli dostępnych (sprawdzenie wymaga umieszczenia właściwego wydruku w funkcji klienta)
      - na zakończenie pracy pubu liczba dostępnych kufli ma być identyczna jak na początku (należy wprowadzić odpowiednie wydruki w funkcji *main*)
        - uzyskanie błędnego działania kodu (różne liczby kufli na początek i na koniec dla programu bez odpowiednich mechanizmów wzajemnego wykluczania) może wymagać wykomentowania opóźnień w pracy wątków (*sleep, printf*) i uruchomienia z bardzo dużą wartością parametru *ile_musze_wypic*, np. 33333 lub więcej)

      (ocena)
      - Do oceny program należy uruchomić trzykrotnie (dla trzech różnych wersji, można je zgłaszać kolejno)
        - dla liczby kufli większej od liczby klientów – działanie niebezpieczne prowadzące do błędu różnej liczby kufli na początku i końcu funkcjonowania pubu

- dla liczby kufli większej od liczby klientów – działanie poprawne dzięki wprowadzeniu mechanizmu wzajemnego wykluczania
- z liczbą klientów większą od liczby kufli – działanie błędne, mimo mechanizmu wzajemnego wykluczania, klienci pobierają kufle, mimo, że nie ma już wolnych

◦ *-> jaka najprostsza reprezentacja pozwala na rozwiązanie problemu bezpiecznego korzystania z kufli w pubie w przypadku liczby kufli większej od liczby klientów (jeden kufel jest posiadany tylko przez jednego klienta)?*

c) W celu zapewnienia poprawności działania pubu, kiedy liczba klientów jest większa niż liczba kufli, należy wprowadzić mechanizm sprawdzania dostępności kufli – klient pobiera kufel tylko wtedy, kiedy jest jakiś dostępny, kiedy nie ma - musi poczekać (ewentualnie robiąc coś innego) i ponownie spróbować pobrać kufel (lub od razu ponowić próbę pobrania, żeby nikt go nie uprzedził). Nie mając narzędzi do oczekiwania na dostępność kufli w stanie uśpienia, klienci (wątki) muszą aktywnie sprawdzać dostępność kufli (realizując zaprojektowany wariant aktywnego czekania (*busy waiting*) – uwaga: klient oczekujący na pojawienie się dostępnych kufli musi umożliwić innym oddanie kufla). Zrealizowane rozwiązanie należy sprawdzić dla różnych liczb kufli i klientów – wykorzystując mechanizmy sprawdzania z poprzedniego punktu. **(ocena)**

Wskazówka: Można wykorzystać prosty wzorzec poniżej (kropki oznaczają miejsca, gdzie musi zostać coś uzupełnione, fragment wykonywania innych operacji, kiedy nie można pobrać kufla, może być jakimś informującym napisem, ewentualnie może zostać pominięty):

```
int success = 0;
do{
   ..... // kiedy można bezpiecznie sprawdzać warunek?
   if ( condition_satisfied ) { ............; success = 1; }
   .....
   if ( success == 0 ) { // or just "else {"    |
       do_something_else_or_nothing();          |- this part can be omitted
   }                                            |
   .....
} while ( success == 0 );
```

Uwaga: czy po poprawnym (bezpiecznym) rozwiązaniu problemu fragment: `do_something_else_or_nothing();` będzie wykonywany wewnątrz sekcji krytycznej?

- *-> Jak wygląda rozwiązanie problemu bezpiecznego korzystania z kufli? Jak połączyć je ze sprawdzeniem dostępności kufli? Jaka jest wada rozwiązania z wykorzystaniem tylko muteksów? (czy zasoby procesora są wykorzystywane optymalnie? - nie zawsze program ma jakąś sensowną operacje do wykonania w obszarze* `do_something_else_or_nothing();` *, jeśli nie ma wtedy aktywne czekanie oznacza marnowanie zasobów sprzętowych na nieustające sprawdzanie warunku)*

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

Rozszerzenia dla podwyższenia oceny:

1. Rozważenie wariantu *pub_sym_1.c* (jeden kran) z aktywnym czekaniem, ale z wykorzystaniem *trylock*, tak żeby wątek nie czekał wewnątrz *pthread_mutex_lock*, kiedy w sekcji krytycznej jest inny wątek (muteks jest zamknięty) i mógł od razu przejść do fragmentu programu `do_something_else_or_nothing();` który będzie pełnić rolę działania w obszarze aktywnego czekania i będzie wykonywany, nawet jeśli sekcja krytyczna będzie zamknięta (można tu wykorzystać schemat ze slajdów z wykładu).

- *-> Jaki jest schemat działania pojedynczego wątku, który chce bezpiecznie korzystać z kufli, ale bez bezproduktywnego czekania na wejście do sekcji krytycznej? (należy podać kod i opisać go)*
- *-> Uwaga: użycie* trylock() *nie eliminuje pętli wielokrotnego sprawdzania warunku wewnątrz pętli* (do{....}while(sukces==0)) *– zmienia tylko sposób zarządzania wejściem do sekcji krytycznej; aby uniknąć tej pętli należy wykorzystać inne mechanizmy*

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

2. W związku z rosnącą popularnością pubu (rosnąca liczba klientów) napisanie nowego programu (np. na podstawie dostarczonego szkieletu *pub_sym_2.c*) - wprowadzenie większej liczby **rozróżnialnych** kranów oraz wykorzystanie procedury *trylock* do efektywnej obsługi możliwości korzystania z wielu kranów. Należy również zastosować aktywne czekanie, ale skoro krany maja być rozróżnialne, ich reprezentacja w programie musi być inna niż kufli (w pierwszej wersji można zostawić kufle nierozróżnialne i przenieść mechanizmy obsługi kufli z *pub_sym_1.c*) – program *pub_sym_2.c* zawiera sugestie możliwej reprezentacji kranów **(ocena)**

- *-> Jaka reprezentacja pozwala rozwiązać problem korzystania z rozróżnialnych kranów (i ewentualnie także kufli)? Jaki jest schemat działania pojedynczego wątku, który chce korzystać z kranu?*

3. Rozważenie przypadku rozróżnialnych kufli – zmiana reprezentacji, wprowadzenie nowych mechanizmów zabezpieczeń i nowych napisów informujących: *"piję piwo marki %s z kufla %s"*.

Warunki zaliczenia:

- Obecność na zajęciach i wykonanie kroków 1-3.
- Oddanie sprawozdania, o treści i formie zgodnej z regulaminem ćwiczeń laboratoryjnych, z opisem zadania, kodem źródłowym programów oraz analizą funkcjonowania dla różnych parametrów symulacji (z wklejonymi obrazami ilustrującymi działanie programu – zgodnie z regulaminem laboratorium).
- Symbol -> oznacza pytania, na które odpowiedzi ma dać laboratorium (odpowiedzi powinny znaleźć się w sprawozdaniu – najlepiej w punktach odpowiadających pytaniom)