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.
   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. Punktem wyjścia powinien być program *pub_sym_1.c*
   - Aktywnymi elementami symulacji są klienci-wątki, pub jest reprezentowany przez odpowiednie zasoby - struktury danych. Struktury danych i związane z nimi konstrukcje programistyczne mają zapewniać bezpieczne (poprawne) korzystanie z zasobów pubu.
   - W pierwszej wersji parametrami symulacji są: liczba klientów oraz liczba kufli.
   - Zasobami pubu są kufle i krany z piwem. Początkowo założone jest, że liczba kranów jest na tyle duża, że nie ma problemu rywalizacji przy dostępie do kranów (każdy klient znajduje bez trudu wolny kran).
   - Problemem jest dostęp do kufli - sytuacja kiedy różni klienci/wątki sięgają po ten sam kufel.
   - Bezpieczeństwo (poprawność) polega na tym, że uniemożliwia się dwóm klientom pobranie jednocześnie kufla.
   - Pierwszym problemem do rozstrzygnięcia jest: jak ma wyglądać operacja pobrania kufla? To z kolei wymaga odpowiedzi na pytanie jaka jest reprezentacja pubu?

   Wskazówka: w pierwszej wersji do poprawnej obsługi pubu wystarcza jego reprezentacja w postaci aktualnej liczby dostępnych kufli (jako zmiennej globalnej, dostępnej dla wszystkich wątków)
4. W kodzie powinny znaleźć się sprawdzenia dwóch potencjalnych błędów programu:
   a) kiedy na skutek rywalizacji o kufle, całkowita liczba kufli w pubie zmienia się - ten warunek sprawdza się przez porównanie początkowej i końcowej liczby kufli w pubie (w funkcji *main*, przed zakończeniem pracy pubu)
   b) kiedy klient pobiera kufel, mimo że nie ma już wolnych kufli w pubie - ten warunek jest sprawdzany w funkcji klienta zaraz po pobraniu kufla

   (kod obu przypadków sprawdzenia powinien znaleźć się w sprawozdaniu)
5. Program *pub_sym_1.c* 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.
   a) Rozwiązywanie zadania należy rozpocząć od sytuacji kiedy liczba kufli przewyższa liczbę klientów
   - przy takich założeniach warunek liczby kufli pobranych mniejszej od liczby kufli dostępnych jest zawsze spełniony, jednak warunek tej samej liczby kufli na początku i końcu pracy pubu, bez odpowiednich mechanizmów wzajemnego wykluczania, już nie jest zagwarantowany
   - należy doprowadzić do błędnego działania kodu poprzez odpowiedni dobór parametrów i ewentualne modyfikacje kodu

- uwaga: uzyskanie błędnego działania kodu (różne liczby kufli na początek i na koniec programu) **może** (nie musi) wymagać kompilacji bez opcji optymalizacji (-O0 -g) i/lub **wykomentowania opóźnień w pracy wątków (większość *sleep*, wszystkie *printf*) oraz uruchomienia z dużą wartością parametru *ile_musze_wypic*, np. 333 lub więcej)**; pseudokod w takiej wersji może wyglądać następująco:

```
for(i=0; i<ile_musze_wypic; i++){
  ... // pobranie_kufla
  usleep(1); // może być potrzebne, żeby uniknąć optymalizacji usuwających błąd
             // braku synchronizacji
  ... // oddanie kufla
}
```

Od tego momentu praca powinna przebiegać na wersji dla której błędny kod skutkował błędnymi wynikami i polegać na usuwaniu błędów wykonania przez odpowiednie modyfikacje kodu

b) **Na wstępie należy tak zmodyfikować kod, aby pobieranie kufla było bezpieczne (nie ma wyścigu). W tym celu należy użyć (w pierwszej ocenianej wersji) wyłącznie muteksów i procedury *pthread_mutex_lock* .**

- Do oceny program należy uruchomić trzykrotnie:
  - dla liczby kufli większej od liczby klientów i niebezpiecznej wersji operacji pobrania i oddania kufla – błędne działanie programu prowadzące do różnej liczby kufli na początku i końcu funkcjonowania pubu
  - dla liczby kufli większej od liczby klientów i bezpiecznej wersji operacji pobrania i oddania kufla, dzięki wprowadzeniu mechanizmu wzajemnego wykluczania – działanie poprawne
  - z liczbą klientów większą od liczby kufli i bezpiecznej wersji operacji – działanie błędne, mimo mechanizmu wzajemnego wykluczania: klienci pobierają kufle, mimo, że nie ma już wolnych

(dla każdego z powyższych przypadków sprawozdanie powinno zawierać kod operacji pobierania i oddawania kufli oraz wydruki niepoprawnego lub poprawnego działania)

- *-> 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 (`do_something_else_or_nothing();`), 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();` jest 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)*

Na zakończenie, po uzyskaniu poprawnie działającego pubu, można przywrócić (odkomentować) pierwotne napisy (printf) i oczekiwania na wykonanie działań (sleep)

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

Rozszerzenia dla podwyższenia oceny:

1. Rozważenie wariantu *pub_sym_1.c* (z wystarczająco dużą liczbą kranów) 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).
   - fragment `do_something_else_or_nothing();` można zaimplementować nie jako funkcję, ale np. jako zwiększanie o 1 wartości pewnej zmiennej zdefiniowanej na początku funkcji wątku jako: *long int wykonana_praca = 0;*
   - każdy wątek może przed końcem pracy (w momencie opuszczania pubu) drukować wartość zmiennej *wykonana_praca*

- *-> 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)*
- *-> Czy użycie* trylock() *zwiększa ilość wykonanej pracy (w oczekiwaniu na picie piwa), czy wynik zależy od liczby klientów i proporcji liczby klientów do liczby kufli?*
- *-> 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 należałoby wykorzystać inne mechanizmy*

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

2. W związku z rosnącymi wymaganiami klientów, napisanie nowego programu (np. na podstawie dostarczonego szkieletu *pub_sym_2.c*) - wprowadzenie małej liczby **rozróżnialnych** kranów (np. z różnymi gatunkami piwa) 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"*.

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

Warunki zaliczenia:

- Obecność na zajęciach i wykonanie kroków 1-6.
- Oddanie sprawozdania, o treści i formie zgodnej z regulaminem ćwiczeń laboratoryjnych, z opisem zadania, kodem źródłowym programów **(wszystkie dokonane na kolejnych etapach rozwijania kodu modyfikacje, łącznie z fragmentami sprawdzania poprawności działania)** 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)