NOTATKA: SYNCHRONIZACJA

wywołania sleep i wakeup – operacje, które w momentach, kiedy procesy nie mogą wejść do swoich sekrytów, blokują je, zamiast marnotrawić czas procesora. sleep to wywołanie systemowe, które powoduje zablokowanie procesu wywołującego – zawieszenie go do czasu, kiedy inny proces go obudzi. wakeup wybudza ze sleep.

semafor – zmienna służąca do zliczania liczby zapisanych sygnałów wakeup.

- wartości semafora:
  - 0 – brak zapisanych sygnałów wakeup
  - > 0 – gdy istnieje jeden lub więcej zaległych sygnałów wakeup
- operacje na semaforze (niepodzielne):
  - up – inkrementuje wartość wskazanego semafora. Jeśli na tym semaforze był uśpiony jeden proces lub więcej procesów, które nie mogły wykonać wcześniejszej operacji down, to system wybiera jeden z nich (np. losowo) i zezwala na dokończenie operacji down. Więc po wykonaniu operacji up na semaforze, na którym były uśpione procesy, semafor w dalszym ciągu będzie miał wartość 0, ale będzie na nim uśpiony o jeden proces mniej.
  - down – sprawdza wartość zmiennej
    - = 0 – proces jest przełączany na chwilę w stan uśpienia bez wykonywania operacji down
    - > 0 – dekrementuje wartość (tzn. wykonuje operację up z argumentem 1 dla zapisanych sygnałów wakeup) i kontynuuje.
- system operacyjny na czas sprawdzania semafora powinien zablokować przerwania, zaktualizować semafor i jeśli trzeba – przełączyć proces do stanu uśpienia.
- rodzaje semaforów:
  - **semafor binarny** (mutex) – semafor inicjowany wartością 1 i używany przez dwa lub więcej procesów po to, by uzyskać pewność, że tylko jeden z nich może wejść do sekrytu w tym samym czasie - wzajemne wykluczanie
  - **semafor zliczający** - synchronizacja – licznik zestawu dostępnych zasobów
- problem semaforów: trzeba przyjąć założenie, że przynajmniej do fragmentu współdzielonej pamięci ma dostęp wiele procesów. Semafory mogą być przechowywane w jądrze, a dostęp do nich być możliwy tylko za pomocą wywsysów. W większości sysopków istnieje mechanizm pozwalający procesom współdzielić pewną część swojej przestrzeni adresowej z innymi procesami.

mutex – uproszczona wersja semafora; nadaje się do zarządzania wzajemnym wykluczaniem niektórych współdzielonych zasobów lub fragmentów kodu (wykorzystywane, gdy nie jest potrzebna właściwość zliczania)

- zmienna, która może znajdować się w jednym z dwóch stanów: odblokowany lub zablokowany. Do jego zaprezentowania jest potrzebny jeden bit. W praktyce 0 – odblokowany, cokolwiek innego – zablokowany.
- procedury:
  - mutex_lock – kiedy wątek lub proces potrzebuje dostępu do sekryta. Jeśli muteks jest już odblokowany (sekryt dostępny), to wywołanie kończy się sukcesem i wątek wywołujący może wejść do sekrytu. Jeśli muteks jest zablokowany, wątek wywołujący zablokuje się do czasu, kiedy wątek znajdujący się w sekrycie zakończy w nim działania i wywoła mutex_unlock. Jeśli na muteksie zablokowanych jest wiele wątków, losowo wybieramy jest jeden z nich i otrzymuje zgodę na założenie blokady.
  - mutex_unlock – kiedy wątek zakończy swoje działania w sekrycie
- różnica między enter_region: kiedy mutex_lock nie uda się ustawić blokady, wywołuje funkcję thread_yield po to, by przekazać procesor do innego wątku. W konsekwencji nie ma aktywnego oczekiwania (powtarzanie testowania blokady). Kiedy wątek uruchomi się następnym razem, ponownie analizuje blokadę.

monitor – kolekcja procedur, zmiennych i struktur danych pogrupowanych ze sobą w specjalnym rodzaju modułu lub pakietu.

- procesy mogą wywoływać procedury w monitorze, kiedy tylko tego chcą, ale z poziomu procedur zadeklarowanych poza monitorem nie mogą bezpośrednio korzystać z wewnętrznych struktur danych monitora
- w dowolnym momencie w monitorze może być aktywny tylko jeden proces
- zmienne warunkowe - sposób na to, by procesy blokowały się w czasie, gdy nie mogą kontynuować działania

zmienna warunkowa

- ma dwie operacje: wait i signal.
- nie są licznikami – nie akumulują sygnałów, jeśli zostanie wysłany sygnał do zmiennej warunkowej, na który nikt nie czeka, zostanie utracony na zawsze – operacja wait musi być wykonana przed operacją signal

- kiedy procedura monitora wykryje, że nie może kontynuować działania (np. producent wykryje, że bufor jest pełny), wykonuje operację wait na wybranej zmiennej warunkowej, np. full. operacja ta powoduje zablokowanie procesu wywołującego. Pozwala również innemu procesowi, który wcześniej nie mógł wejść do monitora, aby teraz do niego wszedł.
- inny proces, np. konsument, może obudzić swojego uśpionego partnera poprzez przesłanie sygnału z wykorzystaniem zmiennej warunkowej, na którą jego partner oczekuje. Aby uniknąć jednoczesnego występowania dwóch aktywnych procesów w monitorze, potrzebna jest reguła, która informauje o tym, co się dzieje po wykonaniu operacji signal.
    - Hoare zaproponował umożliwienie działania przebudzonemu procesowi i zawieszenie drugiego z nich
    - Hansen: wymaganie od procesu wykonującego operację signal natychmiastowego opuszczenia monitora
- różnica między sleep i wakeup a wait i signal: te pierwsze zawodzą, gdy jeden proces próbuje przejść w stan uśpienia, natomiast drugi próbuje go obudzić. W przypadku monitorów to nie może się zdarzyć – automatyczne wzajemne wykluczanie procedur.

L5.Z1. Wyjaśnij różnice między **zakleszczeniem** (ang. *deadlock*), **uwięzieniem** (ang. livelock) i **głodzeniem** (ang. *starvation*). W podręcznikach pojęcia te odnoszą się do **zasobów**. Pokaż, że podobne problemy występują w przypadku **przesyłania komunikatów.**

zasoby – wszystko to, co trzeba uzyskać, wykorzystać i zwolnić, np. urządzenia, rekordy danych, pliki itp.

- zasób z wywłaszczaniem – można odebrać procesowi korzystającemu z niego bez skutków ubocznych, np. pamięć
- zasób bez możliwości wywłaszczania – nie można go zabrać bieżącemu właścicielowi bez szkody dla obliczeń, np. nagrywarki Blu-ray

zakleszczenie (*deadlock*) **–** gdy każdy proces jest zablokowany, bo czeka na zdarzenie od innego zablokowanego procesu (np. na zwolnienie jakiegoś zasobu). Żadne ze zdarzeń nigdy nie nastąpi, więc zakleszczenie jest trwałe.

Przykład:

- korki w mieście
- problem ucztujących filozofów: pięciu filozofów siedzi przy stole i każdy wykonuje jedną z czynności: albo je, albo rozmyśla. Stół jest okrągły, przed każdym z nich znajduje się miska ze spaghetti, a pomiędzy każdą sąsiadującą parą filozofów leży widelec, a więc każda osoba ma przy sobie dwie sztuki – po swojej lewej i prawej stronie. Do zjedzenia potrzebne są dwa widelce. Ryzyko zakleszczenia, gdy każdy z nich zabierze lewy widelec i będzie czekał na prawy (lub na odwrót)

uwięzienie (ang. *livelock*) **–** sytuacja, w której dwa procesy zmieniają swój stan w odpowiedzi na zmianę tego drugiego bez wykonania postępu w pracy. Podobne do zakleszczenia, bo nie ma żadnego progresu, ale różni się tym, że procesy nie są blokowane i nie czekają na żadne zasoby.

Przykład:

- dwoje małżonków je obiad, ale mają jedną łyżkę. Każde jest bardzo miłe i daje drugiemu łyżkę, jeśli ta osoba jeszcze nie jadła.
- kod

```
var l1 = .... lock object like semaphore or mutex etc...
var l2 = .... lock object like semaphore or mutex etc...

// Thread1
Thread.Start( ()=>{
    while(true) {
        if(!l1.Lock(1000))
        {
            continue;
        }
        if(!l2.Lock(1000))
        {
            continue;
        }
        /// do some work
});

// Thread2
Thread.Start( ()=>{
    while(true) {
        if(!l2.Lock(1000))
        {
            continue;
        }
        if(!l1.Lock(1000))
        {
            continue;
        }
        /// do some work
});
```

głodzenie (ang. *starvation*) **–** sytuacja, w której dany proces jest pomijany przez planistę i mimo że ma możliwość działania, to nie ma dostępu do procesora lub współdzielonego zasobu.

Przykład:

- w systemie, w którym jeden z procesów ma do wydrukowania duży plik za każdym razem, gdy drukarka będzie wolna, system będzie próbował znaleźć proces, który ma najmniejszy plik do wydrukowania. W przypadku ciągłego napływu procesów z małymi plikami, procesom chcącym wydrukować duży plik nigdy nie zostanie przydzielona drukarka. Zagłodzą się na śmierć – będą w nieskończoność odraczane, mimo że nie są zablokowane.
- kod

```
Queue<int> q = .....

    while(q.Count>0){
```

```
4.          var c = q.Dequeue();
5.          ...
6.          ...
7.          ...
8.          // Some method in different thread accidentally
9.          // puts c back in queue twice within same timeframe
10.         q.Enqueue(c);
11.         q.Enqueue(c);
12.
13.         // leading to growth of queue twice then it
14.         // can consume, thus starving of computing
15.     }
```

komunikat – zawiera nagłówek identyfikujący proces wysyłający i odbierający oraz dane.
przesyłanie komunikatów – pomiędzy dwoma procesami o rozłącznych przadrach obejmuje ich kopiowanie z pamięci do pamięci; gdy między wątkami, to przadr jest wspólny

zakleszczenie komunikacyjne

- do innego rodzaju zakleszczeń dochodzi w systemach komunikacyjnych (np. sieciach), w których dwa lub więcej procesy komunikują się ze sobą poprzez przesyłanie komunikatów. W popularnym układzie proces A wysyła komunikat z żądaniem do procesu B, a następnie blokuje się do czasu, aż proces B zwróci komunikat z odpowiedzią. Przypuśćmy, że komunikat z żądaniem został utracony. Proces A blokuje się w oczekiwaniu na odpowiedź. Proces B blokuje się w oczekiwaniu na żądanie wykonania jakiejś operacji. Mamy zakleszczenie.
- przeciwdziałanie:
    - nie da się przeciwdziałać poprzez żądanie zasobów (ponieważ ich nie ma), nie da się także uniknąć ich przez uważne szeregowanie (nie ma takich momentów w czasie, kiedy można by opóźnić żądanie)
    - inna technika: limity czasu (ang. *timeouts*). Za każdym razem, gdy wysyłany jest komunikat wymagający odpowiedzi, jednocześnie uruchamia się licznik czasu. Jeśli licznik czasu dojdzie do zera, zanim nadejdzie odpowiedź, nadawca komunikatu zakłada, że komunikat został utracony i wysyła go ponownie.
    - jeśli pierwszy komunikat nie został utracony, a jedynie odpowiedź jest opóźniona, to zamierzony odbiorca może otrzymać komunikat dwa lub więcej razy, a konsekwencje mogą być niepożądane (np. system bankowości elektronicznej i komunikat zawierający instrukcje dokonania płatności

L5.Z2. Wymień cztery warunki konieczne do zaistnienia zakleszczenia. W jaki sposób programista może **przeciwdziałać** zakleszczeniom (ang. *deadlock prevention*)? Których z proponowanych rozwiązań nie implementuje się w praktyce i dlaczego?

Warunki konieczne do zaistnienia zakleszczenia

1. wzajemne wykluczenie – w danym czasie tylko jeden proces może korzystać z danego zasobu
2. wstrzymanie i oczekiwanie (*hold and wait*) – proces może trzymać dostęp do zasobów, czekając na otrzymanie dostępu do innych zasobów
3. brak wywłaszczania (*no preemption*) – zasoby przydzielone wcześniej nie mogą być przymusowo zabrane procesom. Muszą być jawnie zwolnione przez proces, który jest ich właścicielem.
4. cykliczne czekanie (*circural wait*) – łańcuch procesów, w którym każdy proces trzyma co najmniej jeden zasób potrzebny innemu procesowi

1-3: *może* zdarzyć się deadlock
1-4: *jest* deadlock

Zapobieganie zakleszczeniom (*deadlock prevention*)

1. wzajemne wykluczenie – w przypadku danych można nadać im status tylko do odczytu, tak aby procesy mogły jednocześnie z nich korzystać. Zapis zawsze musi być dokonywany tylko przez jeden proces.

2. trzymaj i czekaj (*hold and wait*) – wymuszenie na procesach, aby zamawiały zasoby tylko wtedy, gdy nie są w posiadaniu innych zasobów. Średnio w porządku, bo wiele procesów przed rozpoczęciem działania nie wie, ile i jakich zasobów będzie potrzebować. Dwa sposoby:
    a. wymaganie od procesu żądania wszystkich zasobów przed rozpoczęciem wykonywania. Jeśli wszystko jest dostępne, proces otrzyma wszystko, czego potrzebuje, i będzie mógł działać do końca. Jeśli jakiś zasób jest niedostępny, żaden zasób nie zostanie przydzielony i proces musi czekać.
    b. wymaganie od procesu zwolnienia wszystkich przetrzymywanych zasobów przed zamówieniem nowego

3. brak wywłaszczania (*no preemption*) - jeśli procesowi przydzielono drukarkę i jest on w trakcie drukowania swojego wyjścia, przymusowe zabranie mu drukarki ze względu na to, że potrzebny ploter jest niedostępny, będzie czasami niemożliwe, a na pewno trudne. Aby uniknąć tej sytuacji, można doprowadzić do wirtualizacji niektórych zasobów. Buforowanie wyjścia drukarki na dysk i zezwolenie na dostęp do fizycznej drukarki tylko demonowi eliminuje zakleszczeni związane z drukarką (tworzy możliwość zakleszczenia z powodu wyczerpania się miejsca na dysku). Nie wszystkie zasoby można jednak wirtualizować w taki sposób, np. rekordy w bazie danych lub tablice wewnątrz systemu operacyjnego.

4. cykliczne czekanie (*circural wait*)
    a. możemy zastosować regułę, zgodnie z którą w określonym momencie proces jest uprawniony tylko do jednego zasobu. Jeśli potrzebuje kolejnego, musi zwolnić poprzedni. Dla procesu, który wymaga skopiowania dużego pliku z taśmy na drukarkę, takie ograniczenie jest nie do zaakceptowania.
    b. inny sposób: nadanie każdemu zasobowi określony, unikatowy numer porządkowy (liczbę naturalną) i wymuszenie na procesach zamawiania zasobów, według rosnącej numeracji. Oznacza to, że proces, który ma już w swoim posiadaniu zasób nr 4, może zająć zasób nr 5, ale już nie może zająć zasoby nr 3, a nawet kolejnego egzemplarza zasobu nr 4. Alternatywnie można wymagać, aby proces posiadający zasób nr 4 zwolnił go przed zamówieniem zasobu nr 3.

Niech zapis *{P} I {Q}* oznacza, że formuły *P* i *Q* są prawdziwe odpowiednio przed i po wykonaniu instrukcji *I*. Formuły te nazywamy kolejno *warunkami wstępnymi* (ang. *preconditions*) i *warunkami końcowymi* (ang. *postconditions*). Zauważ, że z racji wywłaszczania dla programu {P1} I1 {Q1}; {P2} I2 {Q2} nie musi zachodzić Q1 ≡ P2!
UWAGA! W kolejnych zadaniach należy jawnie opisywać globalny stan przy pomocy formuł logicznych.

L5.Z3. W poniższym programie występuje **sytuacja wyścigu** (ang. *race condition*) na współdzielonej zmiennej «tally». Wyznacz jej najmniejszą i największą możliwą wartość. Dyrektywa «parbegin» rozpoczyna współbieżne wykonanie procesów.

```
1 const int n = 50;
2 shared int tally = 0;
3
4 void total() {
5   for (int count = 1; count <= n; count++)
6     tally = tally + 1; /* to samo co tally++ */
7 }
8
9 void main() { parbegin (total(), total()); }
```

Maszyna wykonuje instrukcje arytmetyczne wyłącznie na rejestrach – tj. kompilator musi załadować wartość zmiennej «tally» do rejestru, przed wykonaniem dodawania. Jak zmieni się przedział możliwych wartości zmiennej «tally», gdy wystartujemy *k* procesów zamiast dwóch? Odpowiedź uzasadnij.

2 podprocesy
Dolna granica = 2

podproces 1	podproces 2	wartość tally	komentarz
load (tally, reg1)		0	reg1 ma wartość 0 (początkową wartość tally)
	load (tally, reg2)	0	
	inc (reg2)	0	
	store (reg2, tally)	1	
	…	49	podproces 2 kontynuuje pracę aż do przedostatniej iteracji pętli
inc (reg1)		49	
store (reg1, tally)		1	wartość 0 trzymana w reg1 zostaje zinkrementowana i zapisana do tally
	load (tally, reg2)	1	
load (tally, reg1)		1	w obu rejestrach wartość tally wynosi 1
inc (reg1)		1	
store (reg1, tally)		2	
…		50	podproces 1 wykonuje pętlę do końca i kończy swoje działanie
	inc (reg2)	50	
	store (reg2, tally)	2	podobnie jak wcześniej, wartość 1 została zachowana w reg2, zinkrementowana i zapisana do tally

Wszystkie zakończyły podprocesy kończą swoje działanie i wartość końcowa tally wynosi 2.
Górna granica = 100

podproces 1	podproces 2	wartość tally	komentarz
	load (tally, reg2)	0	reg1 ma wartość 0 (początkową wartość tally)
	inc (reg2)	0	
	store (reg2, tally)	1	
	…	50	podproces 2 kończy się
load (tally, reg1)		50	podproces 2 zaczyna pracę
inc (reg1)		50	
store (teg1, tally)		51	
…		100	podproces 1 kończy się

k podprocesów

Dolna granica = 2

podproces 1	podproces 2	podprocesy (3 … k)	tally	komentarz
load (tally, reg1)			0	reg1 ma wartość 0 (początkową wartość tally)
	load (tally, reg2)		0	
	inc (reg2)		0	
	store (reg2, tally)		1	
	…		49	podproces 2 kontynuuje pracę aż do przedostatniej iteracji pętli
		…	x	podprocesy (3 … k) wykonują się do końca, modyfikując dowolnie zmienną tally
inc (reg1)			x	
store (reg1, tally)			1	wartość 0 trzymana w rejestrze 1 zostaje zinkrementowana i zapisana do tally, rezultat pracy podprocesów (3 … k) nie ma znaczenia
	load (tally, reg2)		1	
load (tally, reg1)			1	
inc (reg1)			1	
store (reg1, tally)			2	
…			50	podproces 1 wykonuję pętlę do końca i kończy swoje działanie
	inc (reg2)		50	
	store (reg2, tally)		2	podobnie jak wcześniej, wartość 1 została zachowana w reg2, zinkrementowana i zapisana do tally

Wszystkie zakończyły podprocesy kończą swoje działanie i wartość końcowa tally wynosi 2.

Górna granica = 50k
Podobnie jak w przykładzie dla 2 podprocesów, każdy podproces wykonuję się bez zamrożenia stanu i przełączenia się na inny podproces:

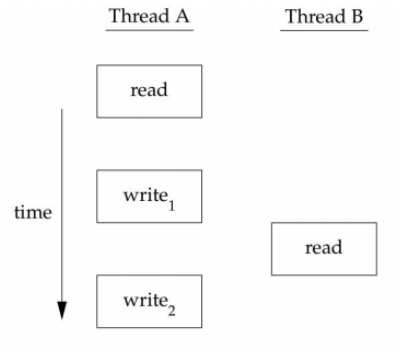
- podproces 1 wykonuję się od początku do końca (tally = 50)
- podproces 2 wykonuje się od początku do końca (tally = 2 * 50).
- …
- podproces k wykonuje się od początku do końca (tally = k * 50).

Dlaczego jest granica 50k to maksimum? Każdy podproces wykonuje maksymalnie 50 inkrementacji, które mogę zwiększyć o 1 zmienną tally. Poprocesów jest k, więc 50k to maksimum.

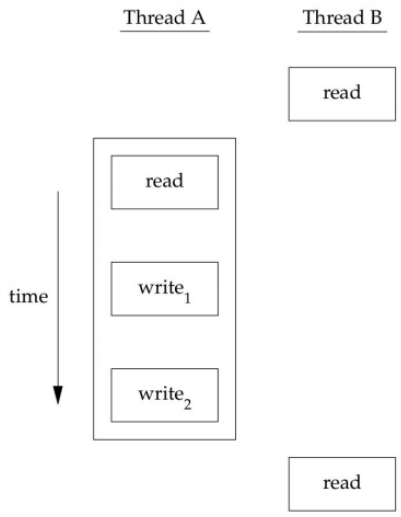
L5.Z4. Rozważmy poniższy kod ze slajdów do wykładu. Zakładamy, że kolejka «queue» przechowuje do *n* elementów. Wszystkie operacje na kolejce są **atomowe** (ang. *atomic*). Startujemy po jednym wątku wykonującym kod procedury «producer» i «consumer». Procedura «sleep» usypia wołający wątek, a «wakeup» budzi wątek wykonujący daną procedurę. Wskaż przeplot instrukcji, który doprowadzi do
(a) błędu wykonania w linii 6 i 13
(b) zakleszczenia w liniach 5 i 12.

```
1  def producer():
2    forever:
3      item = produce()
4      if queue.full():
5        sleep()
6      queue.push(item)
7      if not queue.empty():
8        wakeup(consumer)

9  def consumer():
10   forever:
11     if queue.empty():
12     sleep()
13   item = queue.pop()
14   if not queue.full():
15     wakeup(producer)
16   consume(item)
```



Operacja jest **atomowa** jeśli obserwator nie może zobaczyć wyników pośrednich.



operacje atomowe – obserwator operacji nie może zobaczyć wyników pośrednich. Atomiczność gwarantuje nam brak przerwań sprzętowych, sygnałów, izolację od innych wątków i procesów.

(a) błędu wykonania w linii 6 i 13

- Queue: 0, producer: **2** consumer: 10
- Queue: 0, producer: **3** consumer: 10
- Queue: 0, producer: **4** consumer: 10
- Queue: 0, producer: **6** consumer: 10
- Queue: **1**, producer: **7** consumer: 10 if jest true, nastepne wykona 8
- Queue: **1**, producer: 7 consumer: **11-13**
- Queue: 0, producer: 7 consumer: **14-16, 11,12**
- Queue: 0, producer: **8** consumer: 12 tu budzi consumer, gdy kolejka pusta
- Queue: 0, producer: 8 consumer: **13**

(b) zakleszczenia w liniach 5 i 12

- Queue: 0, producer: 2 consumer: **11** if jest true, nastepny bedzie sleep
- Queue: 0, producer: **3,6,7** consumer: 11
- Queue: **1**, producer: **3,6,7** consumer: 11
- Queue: **2**, producer: **3,6,7** consumer: 11
- …
- Queue: **n**, producer: **3,4** consumer: 11 nastepny bedzie sleep
- Queue: n, producer: 4 consumer: **12** idzie spac, mimo ze kolejka jest pelna
- Queue: n, producer: **5** consumer: 12 tez zasypia

L5.Z5. Przeczytaj rozdział 2 artykułu „*Beautiful concurrency*”. Na podstawie poniższego przykładu wyjaśnij czemu złożenie ze sobą poprawnych współbieżnych programów używających blokad nie musi dać poprawnego programu (tj. „*locks are not composable*”). Jak poprawić procedurę transfer? Czemu według autorów artykułu blokady nie są dobrym narzędziem do strukturyzowania współbieżnych programów?

Cel: napisz procedurę transferującą pieniądze z jednego konta bankowego do drugiego. Oba konta są trzymane w pamięci, nie ma operacji na bazach danych. Procedura musi poprawnie działać w programie wykonywanym współbieżnie, żaden wątek nie powinien widzieć pośrednich stanów pomiędzy opuszczeniem konta 1 a dotarciem na konto 2.

```
1  class Account {
2    int balance;
3    synchronized void withdraw(int n) { balance -= n; }
4    synchronized void deposit(int n) { balance += n; }
5  }
```

synchronized przy withdraw powoduje, że na czas trwania withdraw nałożona jest blokada
niepuci, bo we współbieżnym programie wątek 2 może zobaczyć operacje pomiędzy przerzuceniem pieniędzy z from do to. To, że są synchronized wiele nam nie daje, ponieważ pomiędzy wywołaniami from i to jest przerwa.

```
7  void transfer(Account from, Account to, int amount) {
8    from.lock(); to.lock();
9    from.withdraw(amount);
10   to.deposit(amount);
11   from.unlock(); to.unlock();
12 }
```

transfer jest podatny na zakleszczenia. Przykład: dwa wątki transferują pieniądze w przeciwnych kierunkach na dwa te same konta A, B. Jeden zakłada blokadę na konto A, drugi na B. Jeden wątek nie może przesłać pieniędzy na A, drugi na B, bo są zablokowane.

```
if from < to
then { from.lock(); to.lock(); }
else { to.lock(); from.lock(); }
```

Proponowane rozwiązanie: ustalamy blokady z pewnym liniowym porządkiem. Zadziała, gdy wiemy, ile blokad jest potrzebnych i musi być to znane z góry. Przykład: sytuacja - from.withdraw bierze pieniądze z from2, jeśli na from nie ma środków. Zanim nie przeczytamy from, nie wiemy, czy na from2 trzeba założyć blokadę. Może na założenie blokady na from2 jest już za późno, skoro składamy je w ustalonym porządku.

Blokady i zmienne warunkowe nie wspierają programowania modularnego – procesu budowania dużych programów poprzez sklejanie ze sobą mniejszych. Blokady to uniemożliwiają. Np. nie możemy użyć naszych (poprawnych) implementacji withdraw i deposit w niezmienionym stanie do implementacji transfer. Musimy ujawnić nasz protokół blokowania, abstrakcja nie jest zachowana.

L5.Z6. Poniżej znajduje się propozycja programowego rozwiązania problemu wzajemnego wykluczania dla dwóch procesów. Znajdź kontrprzykład, w którym to rozwiązanie zawodzi. Okazuje się, że nawet recenzenci renomowanego czasopisma „*Communications of the ACM*" dali się zwieść.

```
1  shared boolean blocked [2] = { false, false };
2  shared int turn = 0;
3
4  void P (int id) {
5    while (true) {
6      blocked[id] = true;
7      while (turn != id) {
8        while (blocked[1 - id])
9          continue;
10       turn = id;
11     }
12     /* put code to execute in critical section here */
13     blocked[id] = false;
14   }
15 }
16
17 void main( ) { parbegin (P(0), P(1)); }
```

Idea: blocked == true oznacza chęć wejścia do sekcji krytycznej. Po uruchomieniu P proces ustawia wartość blocked na true (L6). Jego tura nadejdzie (L10) dopiero wtedy, gdy uzyskamy pewność, że drugi proces nie jest zainteresowany sekcją krytyczną, skończył w niej pracę (L8). Przechodzimy do sekcji krytycznej (L12) i po wykonaniu pracy porzucamy zainteresowanie sekrytem (L13).

Kontrprzykład:

- stan początkowy:
  - blocked = {false, false}
  - turn = 0
- P1 wykonuje się
  - wchodzi do pętli L7 „while (turn != id)" (0 != 1)
  - nie wchodzi w do pętli L8 „while (blocked[1 - id])" (blocked[0] == false)
  - wstrzymuje wykonanie przed L10 „turn = id", zostaje wywłaszczony
- stan:
  - blocked = {false, true}
  - turn = 0
- P0 wykonuje się
  - nie wchodzi do pętli L7 „while (turn != id)" (0 != 0)
  - L12: zaczyna wykonywać sekcję krytyczną, zostaje wywłaszczony
- stan:
  - blocked = {false, true}
  - turn = 0
- P1 wznawia wykonanie
  - zaczyna od L10 „turn = id", wychodzi z pętli
  - L12: wchodzi w sekcję krytyczną

Mechanizm nie zawsze działa dobrze, ponieważ pozwala, żeby dwa procesy były w sekcji krytycznej.

L5.Z7. Podaj w pseudokodzie implementację **semafora** z operacjami «init», «down» i «up» używając wyłącznie muteksów i zmiennych warunkowych. Dopuszczamy ujemną wartość semafora.
Podpowiedź: Semaphore = {critsec: Mutex, waiters: CondVar, count: int, wakeups: int}

1. ----------
```
Semaphore = {
   critsec: Mutex,
   waiters: CondVar,
   count: int,
   wakeups: int,
}


void init(Semaphore &sem, int n):
   sem.critsec = init_mutex(sem.critsec)
   sem.waiters = init_condvar(sem.waiters)
   sem.count = n
   sem.wakeups = 0


void wait(Semaphore &sem):
   wait (sem.critsec)
   sem.count--

   if (sem.count < 0):
      do {
         cond_var_wait(sem.waiters,
sem.critsec)
      while (s.wakeups < 1):
      sem.wakeups--

   signal (sem.critsec)


void post(Semaphore &sem):
   wait (sem.critsec)
   sem.count++

   if (sem.count <= 0):
      sem.wakeups++
      cond_var_signal (sem.waiters)

   signal (sem.critsec)
```

2. ----------
```
struct semaphore {
   int value;
   int wakeups;
   mutex* mtx;
   condvar* cond;
}

void init(semaphore* s) {
   s->value = 0;
   s->pending_wakeups = 0;
   mutex_init(s->mtx);
   condvar_init(s->cond);
}

void wait(semaphore *s) {
   mutex_lock(s->mutex);
   s->value--;

   if (s->value < 0) {
      do {
         condvar_wait(s->cond, s->mtx);
      } while (s->wakeups < 1);
      s->wakeups--;
   }
   mutex_unlock(s->mutex);
}

void post(semaphore *s) {
   mutex_lock(s->mutex);
   s->value++;

   if (s->value <= 0) {
      s->wakeups++;
      condvar_signal(s->cond);
   }
   mutex_unlock(s->mutex);
}
```

3. ----------
```
Semaphore = {
   critsec: Mutex,
   waiters: CondVar,
   count: int,
   wakeups: int,
}


void init(Semaphore &sem, int n):
   sem.critsec = init_mutex(sem.critsec)
   sem.waiters = init_condvar(sem.waiters)
   sem.count = n
   sem.wakeups = 0


void wait(Semaphore &sem):
   wait (sem.critsec)
   sem.count--

   if (sem.count < 0):
      do {
         cond_var_wait(sem.waiters,
sem.critsec)
      while (s.wakeups < 1):
      sem.wakeups--

   signal (sem.critsec)


void post(Semaphore &sem):
   wait (sem.critsec)
   sem.count++

   if (sem.count <= 0):
      sem.wakeups++
      cond_var_signal (sem.waiters)

   signal (sem.critsec)
```