Programowanie współbieżne
Rozwiązywanie problemu wzajemnego wykluczania

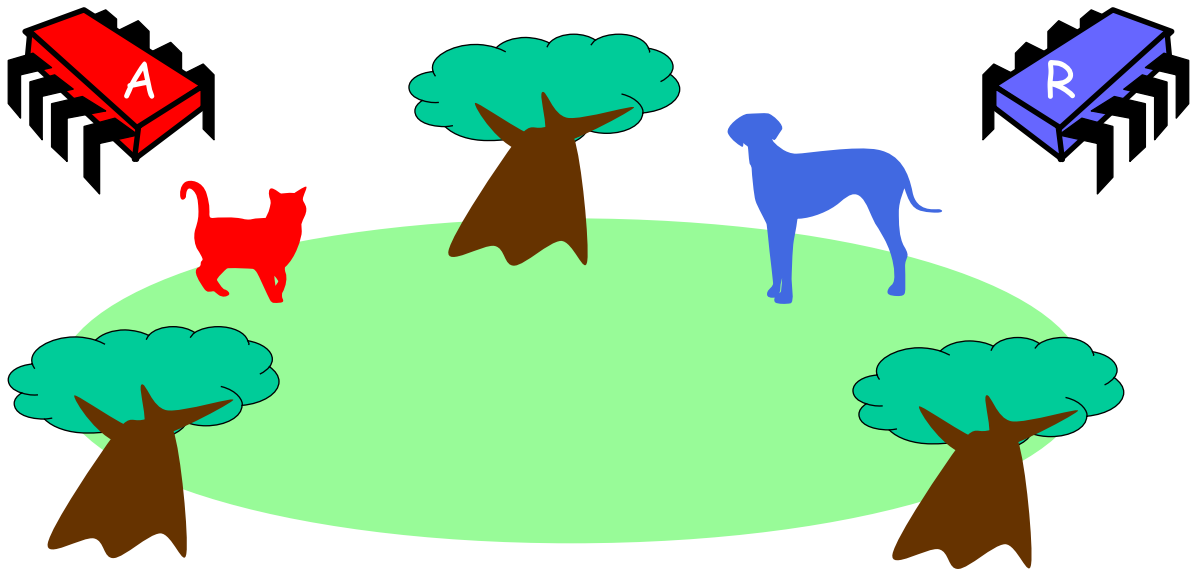
Rafał Skinderowicz

Na początek – bajka

- Zanim przejdziemy do omówienia problemu wzajemnego wykluczania w kontekście informatycznym zastanówmy się nad rozwiązaniem tego problemu w kontekście bajki / opowieści zaczerpniętej z książki: *Herlihy M., Shavit N., Sztuka programowania wieloprocesorowego, PWN.*

Alicja i Robert

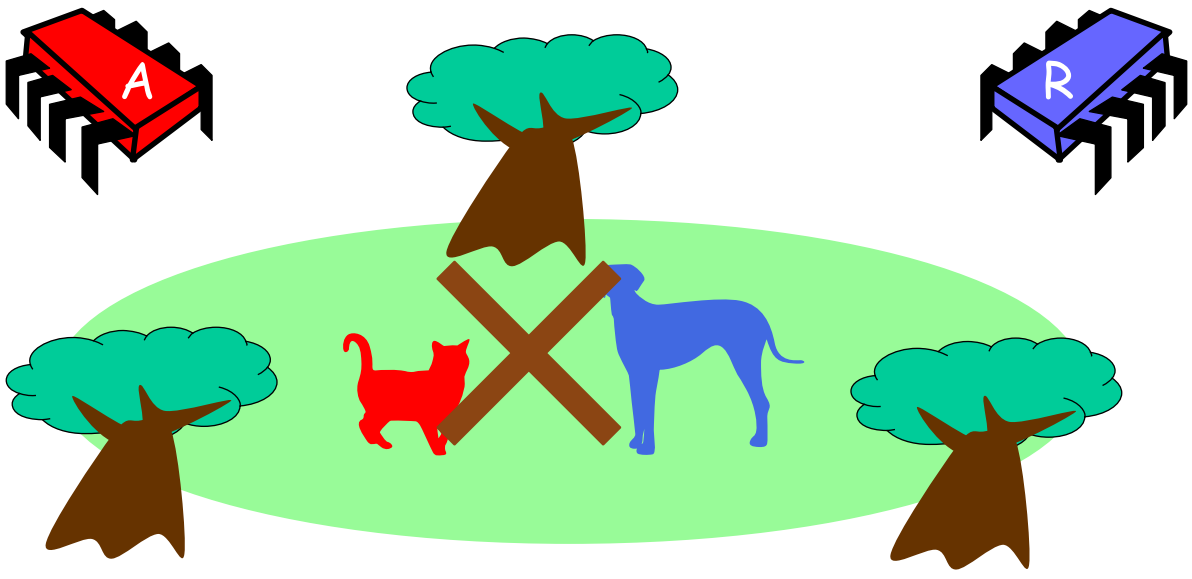
- Alicja i Robert mieszkają niedaleko od siebie:
- Alicja ma kota, Robert psa
- Zwierzętom pozwalają biegać po współdzielonym ogrodzie



Rysunek 1: Na podstawie: "Art of Multiprocessor Programming", M. Herlihy i N. Shavit (CC BY-SA 3.0) oraz "Cats" i "Dogs" Mauricio Duque (CC BY 3.0)

Problem

- Zwierzęta nie przepadają za sobą



Problem – sformułowanie

Chcemy ustalić wspólny sposób postępowania (protokół), który zapewni własności:

- Wzajemnego wykluczania
  - Oba zwierzaki nie mogą znaleźć się w ogrodzie tym samym czasie
  - Jest to własność bezpieczeństwa (dosłownie ☺)
- Braku zakleszczenia:
  - Jeżeli któraś z osób chce wypuścić swojego pupila, to jej się uda
  - Jeżeli obie chcą – jednej się uda
  - Jest to własność żywotności

Protokół – propozycja 1.

- Pomysł (protokół) nr 1:
  - Zerknij na ogród, czy jest pusty i jeżeli tak, to wypuść pupila

Protokół – propozycja 1.

- Pomysł (protokół) nr 1:
  - Zerknij na ogród, czy jest pusty i jeżeli tak, to wypuść pupila
- Problem:
  - Nie cały jest widoczny – drzewa zasłaniają widok – zwierzak może się „schować” za drzwiami i właśnie wchodzić do ogrodu

Protokół – propozycja 1.

- Pomysł (protokół) nr 1:
  - Zerknij na ogród, czy jest pusty i jeżeli tak, to wypuść pupila
- Problem:
  - Nie cały jest widoczny – drzewa zasłaniają widok – zwierzak może się „schować” za drzwiami i właśnie wchodzić do ogrodu

Interpretacja:

- Wątki (procesy) „nie widzą” tego co robią pozostałe wątki
- Niezbędna jest jawna komunikacja

Protokół – propozycja 2.

- Pomysł:
  - Robert dzwoni do Alicji z pytaniem (lub odwrotnie)

Protokół – propozycja 2.

- Pomysł:
  - Robert dzwoni do Alicji z pytaniem (lub odwrotnie)
- Problem:
  - Robert nie może odebrać telefonu
  - W telefonie Alicji rozładowała się bateria
  - Robert wyszedł na zakupy
  - I tak dalej ...

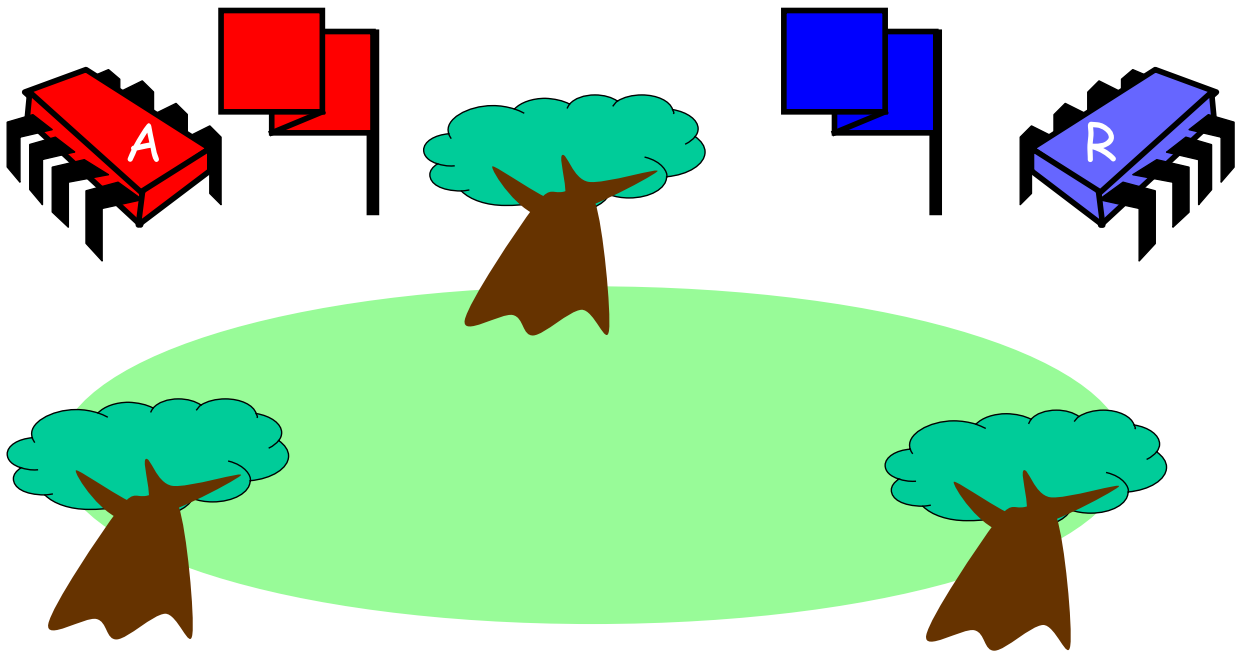
Propozycja 2. – interpretacja

- Komunikacja na zasadzie wymiany komunikatów nie działa
- Odbiorca może:
  - akurat nie słuchać
  - być niedostępny w danym momencie

Propozycja 2. – interpretacja

- Komunikacja na zasadzie wymiany komunikatów nie działa
- Odbiorca może:
  - akurat nie słuchać
  - być niedostępny w danym momencie
- Komunikacja musi byc trwała – jak „na piśmie”

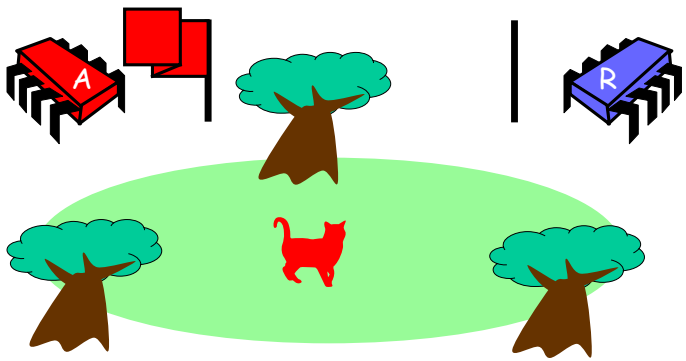
Propozycja 3. – protokół „flagowy”



- Alicja i Robert mają flagi do sygnalizacji swoich zamiarów drugiej stronie

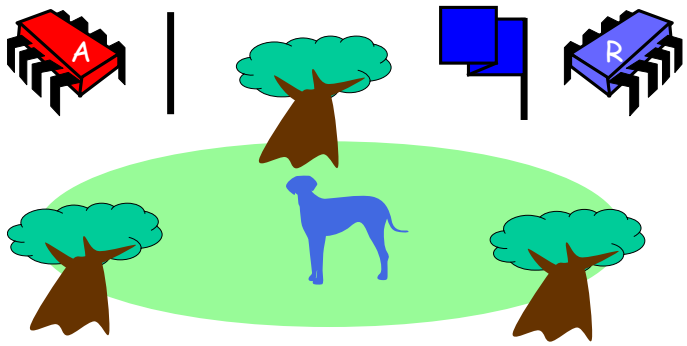
Propozycja 3. – protokół „flagowy”

Protokół Alicji:



1. Podnieś flagę
2. Czekaj, aż Robert opuści swoją
3. Wypuść swoje zwierzątko do ogrodu
4. Opuść flagę kiedy zwierzę wróci

Protokół Roberta:



1. Podnieś flagę
2. Czekaj, aż Alicja opuści swoją
3. Wypuść swoje zwierzątko do ogrodu
4. Opuść flagę kiedy zwierzę wróci

Protokół „flagowy” – problem

- Przedstawiony protokół może prowadzić do zakleszczenia:
  - Alicja i Robert podnoszą swoje flagi *jednocześnie*
  - Oboje czekają, aż druga strona opuści swoją
  - Niestety, nigdy się nie doczekają
- Problem ten można rozwiązać wprowadzając *asymetryczny* protokół dla Roberta

Protokół Roberta – wersja 2.0

1. Podnieś flagę
2. Dopóki flaga Alicji jest podniesiona:
   - Opuść flagę
   - Czekaj, aż Alicja opuści flagę
   - Podnieś flagę
3. Wypuść swoje zwierzątko do ogrodu
4. Opuść flagę kiedy zwierzę wróci

„Zasada flagi”

- Podnieś (ustaw) flagę
- Spójrz na flagę drugiej osoby
- Zasada flagi:
  - Jeżeli obie osoby podnoszą flagi i patrzą, wtedy
  - Ostatnia osoba, która patrzy musi widzieć *obie flagi podniesione*

Dowód wzajemnego wykluczania

- Spróbujmy przedstawić prosty *nieformalny* dowód na to, że przedstawiony protokół ma własność *wzajemnego wykluczania*
- Załóżmy, że **oba zwierzaki znalazły się jednocześnie** w ogrodzie pomimo stosowania się do protokołu
- Na podstawie założenia doprowadzimy do sprzeczności poprzez rozumowanie wstecz

Dowód wzajemnego wykluczania

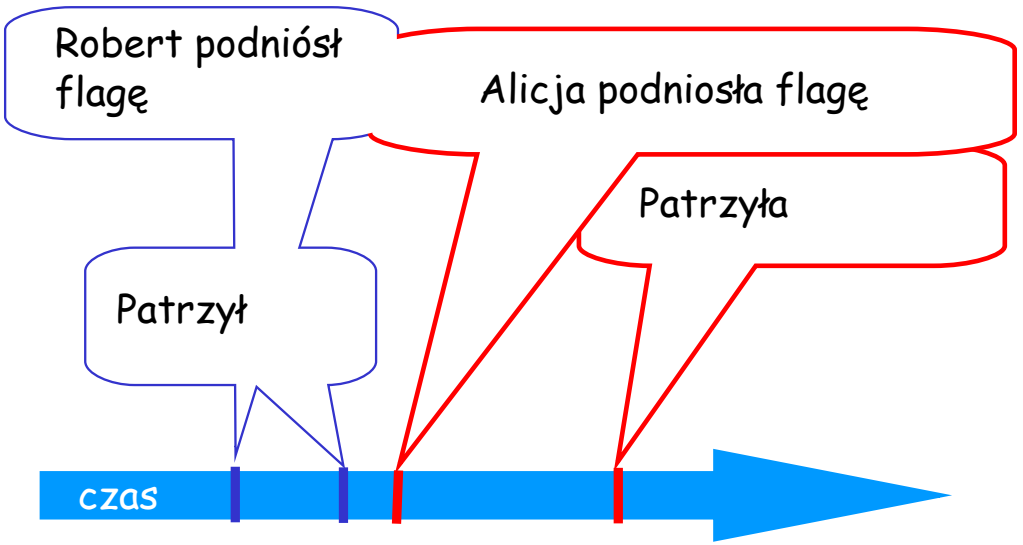
- Rozpatrzmy ostatnią sytuację, w której Alicja i Robert patrzyli w kierunku sąsiada tuż przed wypuszczeniem zwierząt
- Załóżmy, że ostatnia patrzyła Alicja ...
- Skoro Robert wypuścił psa, to znaczy, że gdy patrzył w kierunku domu sąsiadki Alicja jeszcze nie podniosła flagi
- Alicja patrzyła jako druga, a więc później niż Robert

Dowód wzajemnego wykluczania

- Działania Alicji i Roberta można uporządkować w czasie w jedyny zgodny z tokiem rozumowania scenariusz

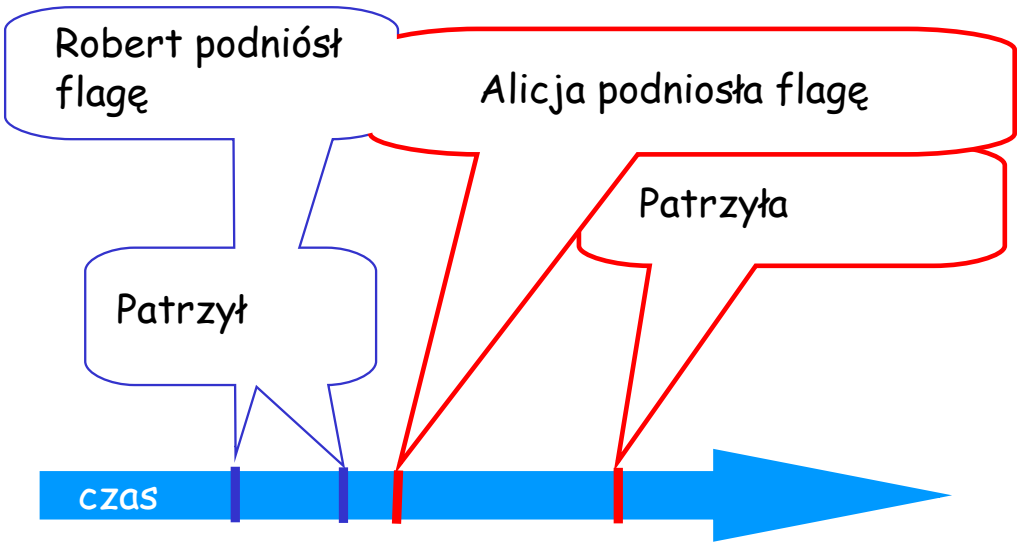
Dowód wzajemnego wykluczania

- Działania Alicji i Roberta można uporządkować w czasie w jedyny zgodny z tokiem rozumowania scenariusz



Dowód wzajemnego wykluczania

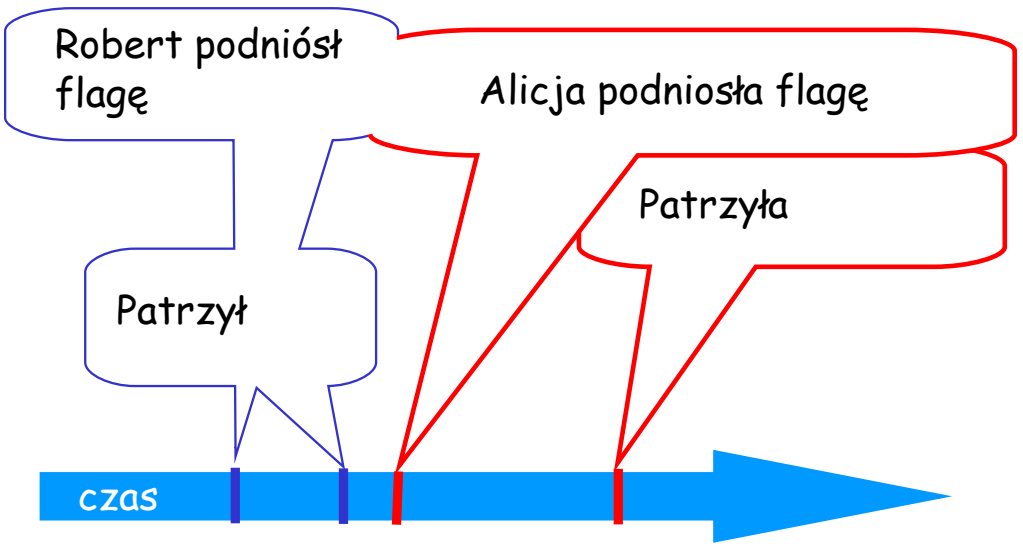
- Działania Alicji i Roberta można uporządkować w czasie w jedyny zgodny z tokiem rozumowania scenariusz



- Scenariusz ten pokazuje, że gdy Alicja patrzyła, to flaga Roberta była w górze

Dowód wzajemnego wykluczania

- Działania Alicji i Roberta można uporządkować w czasie w jedyny zgodny z tokiem rozumowania scenariusz



- Scenariusz ten pokazuje, że gdy Alicja patrzyła, to flaga Roberta była w górze

- Zatem Alicja musiała widzieć podniesioną flagę Roberta, więc nie mogła wypuścić swojego zwierzaka – sprzeczność z naszym założeniem, że oboje działali zgodnie z protokołem

Dowód braku zakleszczenia – uwagi

- Jeżeli tylko jeden zwierzak chce wejść – wchodzi

Dowód braku zakleszczenia – uwagi

- Jeżeli tylko jeden zwierzak chce wejść – wchodzi
- Zakleszczenie wymaga, aby obie strony próbowały uzyskać dostęp się do wspólnego zasobu jednocześnie

Dowód braku zakleszczenia – uwagi

- Jeżeli tylko jeden zwierzak chce wejść – wchodzi
- Zakleszczenie wymaga, aby obie strony próbowały uzyskać dostęp się do wspólnego zasobu jednocześnie
- Jeżeli Robert zobaczy flagę Alicji, daje jej pierwszeństwo, więc nie ma możliwości zakleszczenia

Uwagi

- Zaprezentowany protokół jest niesprawiedliwy:
  - Robert może nigdy nie otrzymać szansy na wpuszczenie swojego psa do ogrodu
- Protokół ten wymaga oczekiwania:
  - Jeżeli Robert podczas pobytu jego psa w ogrodzie zostanie zabrany do szpitala, to nie zdąży opuścić flagi – Alicja nie będzie mogła mieć szansy otrzymania wyłączności do ogrodu dla swojego zwierzaka, pomimo jego „gotowości”

Podsumowanie

- Problem wzajemnego wykluczania (WW) nie może zostać rozwiązany za pomocą:
  - nietrwałej komunikacji
  - przerwań
- Może natomiast zostać rozwiązany za pomocą:
  - współdzielonych rejestrów (zmiennych), pełniących rolę „flag”
  - na zmiennych tych musi być możliwy atomowy zapis i odczyt

Problem WW – terminologia

- Na N-wątkowy proces współbieżny można spojrzeć jak na nieskończoną pętlę złożoną z ciągów instrukcji dwóch rodzajów:
  - sekcji lokalnej (SL)
  - sekcji krytycznej (SK)
- SK to ciąg instrukcji, z których część odwołuje się do *współdzielonych zasobów* (zmiennych w pamięci, plików, baz danych, urządzeń we/wy) i powinna być wykonywana co najwyżej przez jeden wątek na raz
- SL to wszystkie pozostałe fragmenty, które nie wymagają *wyłączności*

Problem WW – terminologia

- Oznaczenia przydatne w późniejszej analizie algorytmów
- Niech $\mathrm{SK}_{\mathrm{i}}^{\mathrm{k}}$ oznacza *k*-te z kolei wykonanie sekcji krytycznej wątku *i*
- Niech $\mathrm{SK}_{\mathrm{j}}^{\mathrm{m}}$ oznacza *m*-te z kolei wykonanie sekcji krytycznej wątku *j*
- Wtedy zachodzi:
  - $\mathrm{SK}_{\mathrm{i}}^{\mathrm{k}} \rightarrow \mathrm{SK}_{\mathrm{j}}^{\mathrm{m}}$ lub $\mathrm{SK}_{\mathrm{j}}^{\mathrm{m}} \rightarrow \mathrm{SK}_{\mathrm{i}}^{\mathrm{k}}$
  - gdzie $\rightarrow$ oznacza relację poprzedzania w czasie

Poprawność programu współbieżnego – przypomnienie

- W celu poprawnego wykonania wątków muszą zostać spełnione warunki:
  - **wzajemne wykluczanie** (ang. mutual exclusion) – wykonanie sekcji krytycznej (tej samej) dwóch wątków nie może się przeplatać – tylko jeden wątek wykonuje w danym momencie sekcję krytyczną
  - **brak zakleszczenia / uwięzienia** (ang. freedom from deadlock / livelock) – jeżeli wątki usiłują wejść do swoich sekcji krytycznych to musi im się w końcu udać
  - **brak zagłodzenia** (ang. freedom from starvation) – każdy wątek, który chce wejść do sekcji krytycznej w końcu do niej wejdzie (mocna uczciwość)
- Wykonanie sekcji lokalnych *różnych* wątków może być przeplatane

Deadlock a livelock

Wątek (proces) jest zakleszczony lub uwięziony jeżeli nie jest w stanie wykonać kolejnych instrukcji zapewniających *postęp* obliczeń, ponieważ oczekuje na zajście warunku, który nigdy nie zostanie spełniony

- zakleszczenie (ang. deadlock) – wątek oczekuje na zajście warunku, ale nie zużywa czasu procesora (np. wait() w Javie)
- uwięzienie (ang. livelock) – wątek oczekuje na zajście warunku, ale zużywa przy tym czas procesora (najczęściej niesk. pętla)

Problem WW – protokoły

- Rozwiązaniem problemu wzajemnego wykluczania jest protokół synchronizacyjny gwarantujący spełnienie warunków poprawności
- Protokół ten składa się z:
  - instrukcji poprzedzających sekcję krytyczną – **protokół wstępny**
  - instrukcji następujących po sekcji krytycznej – **protokół końcowy**

Problem wzajemnego wykluczania

Struktura programu:

Proces P

```
1 zmienne globalne;
2 zmienne lokalne;
3 while (true) {
4     sekcja lokalna;
5     protokół wstępny;
6     sekcja krytyczna;
7     protokół końcowy;
8 }
```

Proces Q

```
1 zmienne globalne;
2 zmienne lokalne;
3 while (true) {
4     sekcja lokalna;
5     protokół wstępny;
6     sekcja krytyczna;
7     protokół końcowy;
8 }
```

Uwagi

- W protokołach można stosować zmienne lokalne i globalne, ale zmienne używane w protokołach nie mogą być stosowane w sekcjach (lokalnej i globalnej) i na odwrót
- Wykonanie sekcji krytycznej musi cechować **postęp** – jeżeli proces rozpocznie wykonywanie instrukcji w sekcji krytycznej, to zakończy je w *skończonym czasie*
- Wykonanie sekcji lokalnej **nie musi cechować się postępem** – jeżeli proces rozpocznie wykonywanie sekcji lokalnej, to może jej nigdy nie zakończyć:
  - zakończenie wykonywania na skutek błędu
  - wpadnięcie w *nieskończoną pętlę*

Dodatkowe pożądane właściwości protokołów WW

- Niewielkie zapotrzebowanie na pamięć
- Niewielka złożoność obliczeniowa – *szybki protokół* to taki, który umożliwia niemal natychmiastowe wejście do sekcji krytycznej w przypadku braku rywalizacji

Rodzaje rozwiązań problemu WW

- Rozwiązana programowe oparte na atomowych instrukcjach odczytu i zapisu pamięci – algorytmy Dekkera, Petersona, piekarniany
- Rozwiązania programowo–sprzętowe – wykorzystanie specjalnych rozkazów procesora, umożliwiających wykonywanie złożonych instrukcji atomowych (np. Test-and-Set)

Implementacje

Istnieją gotowe rozwiązania należące do obu wymienionych rodzajów, np.:

- systemy operacyjne udostępniają mechanizmy synchronizacji dostępu przez biblioteki systemowe (muteksy, semafory w unixie, linuxie)
- specjalne konstrukcje w językach programowania (np. synchronized w Javie)
- bibliotekach standardowe języków programowania i dodatkowe biblioteki zewnętrzne (np. pthreads, Boost.Thread)

Blokady

- Jedną z implementacji protokołu wzajemnego wykluczania jest mechanizm blokad (inaczej zamków, ang. lock)
- W Javie abstrakcję blokady można wyrazić za pomocą poniższego interfejsu

```
1 public interface Lock {
2     public void lock();
3     public void unclock();
4 }
```

- Metoda `lock()` wykonuje protokół wstępny alg. WW
- Metoda `unlock()` wykonuje protokół końcowy alg. WW

Blokady – zastosowanie

Typowy schemat stosowania blokady do rozwiązania problemu WW wygląda następująco na przykładzie współbieżnego licznika:

```
1  public class ConcurrentCounter {
2      private int value;
3      private Lock lock; // obiekt blokady / zamka
4      ...  // dodatkowe pola, metody
5      public void getAndIncrement() {
6          int temp;
7          lock.lock();       // wejdź do sekcji krytycznej
8          try {
9              temp = value;      // w sekcji krytycznej
10             value = value + 1; // w sekcji krytycznej
11         } finally {
12             lock.unlock(); // wyjdź z sekcji krytycznej
13         }
14         return temp;
15     }
16 }
```

Blokady – uwagi

- Każda sekcja krytyczna powinna być skojarzona z tylko jedną, unikalną blokadą
- Każdy wątek wchodząc do sekcji krytycznej korzysta z tej blokady – wywołując metodę `lock()`
- Po zakończeniu obliczeń w ramach sekcji krytycznej wątek wywołuje metodę `unlock()`

Blokady – własności

Brak zakleszczenia:

- Jeżeli wątek wywołuje metodę `lock()`
  - i wywołanie to nie zostaje zakończone
  - to znaczy to, że inny wątek wykonuje metody `lock()` i `unlock()` nieskończenie często
- Program jako całość cechuje się postępem obliczeń – mimo, że niektóre wątki nie dostają szansy wejścia do SK

Blokady – własności

Brak zakleszczenia:

- Jeżeli wątek wywołuje metodę `lock()`
  - i wywołanie to nie zostaje zakończone
  - to znaczy to, że inny wątek wykonuje metody `lock()` i `unlock()` nieskończenie często
- Program jako całość cechuje się postępem obliczeń – mimo, że niektóre wątki nie dostają szansy wejścia do SK

Brak zagłodzenia:

- Jeżeli wątek wywołuje metodę `lock()` , to kiedyś to wywołanie się kończy
- Poszczególne wątki cechują się postępem obliczeń

Alg. WW dla dwóch wątków

- A teraz czas na próbę zaprojektowania algorytmu wzajemnego wykluczania od podstaw, krok po kroku
- Algorytm wymaga jedynie atomowych opreacji odczytu / zapisu do pamięci (rejestrów)

WW dla dwóch wątków – algorytm 1.

```
1  class LockOne implements Lock {
2      private boolean[] wants = new boolean[2];
3      public void lock() {
4          int i = ThreadID.get(); // mój identyfikator
5          int j = 1 - i; // identyfikator drugiego wątku
6          wants[i] = true; // powiadamia o zamiarze wejścia do SK
7          while (wants[j] == true) { ; }
8      }
9      public void unlock() {
10         int i = ThreadID.get(); // mój identyfikator
11         wants[i] = false;
12     }
13 }
```

WW dla dwóch wątków – algorytm 1.

```
1  class LockOne implements Lock {
2      private boolean[] wants = new boolean[2];
3      public void lock() {
4          int i = ThreadID.get(); // mój identyfikator
5          int j = 1 - i; // identyfikator drugiego wątku
6          wants[i] = true; // powiadamia o zamiarze wejścia do SK
7          while (wants[j] == true) { ; }
8      }
9      public void unlock() {
10         int i = ThreadID.get(); // mój identyfikator
11         wants[i] = false;
12     }
13 }
```

- Uwaga: powyższa implementacja nie zadziała poprawnie w Javie
  - chyba, że tablica `wants` będzie składała się ze zmiennych ulotnych, jednak w Javie nie można tworzyć tablic zmiennych ulotnych – można użyć `AtomicIntegerArray` (metody `get()` i `set()`)

Algorytm 1. – analiza

- Tw. Algorytm `LockOne` zapewnia wzajemne wykluczanie dla dwóch wątków A i B o identyfikatorach 0 i 1
- Dowód (przez doprowadzenie do sprzeczności):
  - Załóżmy, że SK_A^j nakłada się z SK_B^k
  - Rozpatrzmy ostatnie odczyty i zapisy każdego wątku w metodzie `lock()`
  - Pokażmy istnienie sprzeczności z założeniem

Algorytm 1. – dowód WW

Z zapisu algorytmu wynika, że:

1. $\text{zapis}_A(\ \text{wants}[0] = \text{true}\ ) \rightarrow \text{odczyt}_A(\ \text{wants}[1] == \text{false}\ ) \rightarrow \text{SK}_\text{A}$
2. $\text{zapis}_B(\ \text{wants}[1] = \text{true}\ ) \rightarrow \text{odczyt}_B(\ \text{wants}[0] == \text{false}\ ) \rightarrow \text{SK}_\text{B}$

```
1 public void lock() {
2     int i = ThreadID.get(); // mój identyfikator
3     int j = 1 - i; // identyfikator drugiego wątku
4     wants[i] = true; // powiadamia o zamiarze wejścia do SK
5     while (wants[j] == true) { ; }
6 }
```

(Zapis x $\rightarrow$ y oznacza, że x poprzedza y)

Algorytm 1. – dowód WW

Z założenia wynika, że:

1. $\text{odczyt}_A(\ \text{wants}[1] == \text{false}\ ) \rightarrow \text{zapis}_B(\ \text{wants}[1] = \text{true}\ )$
2. $\text{odczyt}_B(\ \text{wants}[0] == \text{false}\ ) \rightarrow \text{zapis}_A(\ \text{wants}[0] = \text{true}\ )$

Z zapisu algorytmu wynika, że:

3. $\text{zapis}_A(\ \text{wants}[0] = \text{true}\ ) \rightarrow \text{odczyt}_A(\ \text{wants}[1] == \text{false}\ )$
4. $\text{zapis}_B(\ \text{wants}[1] = \text{true}\ ) \rightarrow \text{odczyt}_B(\ \text{wants}[0] == \text{false}\ )$

Algorytm 1. – dowód WW

Z założenia wynika, że:

1. $\text{odczyt}_A(\ \text{wants}[1] == \text{false}\ ) \rightarrow \text{zapis}_B(\ \text{wants}[1] = \text{true}\ )$
2. $\text{odczyt}_B(\ \text{wants}[0] == \text{false}\ ) \rightarrow \text{zapis}_A(\ \text{wants}[0] = \text{true}\ )$

Z zapisu algorytmu wynika, że:

3. $\text{zapis}_A(\ \text{wants}[0] = \text{true}\ ) \rightarrow \text{odczyt}_A(\ \text{wants}[1] == \text{false}\ )$
4. $\text{zapis}_B(\ \text{wants}[1] = \text{true}\ ) \rightarrow \text{odczyt}_B(\ \text{wants}[0] == \text{false}\ )$

Stąd:

- $\text{zapis}_A(\ \text{wants}[0] = \text{true}\ ) \rightarrow \text{odczyt}_A(\ \text{wants}[1] == \text{false}\ )$

Algorytm 1. – dowód WW

Z założenia wynika, że:

1. $\text{odczyt}_A(\ \text{wants}[1] == \text{false}\ ) \rightarrow \text{zapis}_B(\ \text{wants}[1] = \text{true}\ )$
2. $\text{odczyt}_B(\ \text{wants}[0] == \text{false}\ ) \rightarrow \text{zapis}_A(\ \text{wants}[0] = \text{true}\ )$

Z zapisu algorytmu wynika, że:

3. $\text{zapis}_A(\ \text{wants}[0] = \text{true}\ ) \rightarrow \text{odczyt}_A(\ \text{wants}[1] == \text{false}\ )$
4. $\text{zapis}_B(\ \text{wants}[1] = \text{true}\ ) \rightarrow \text{odczyt}_B(\ \text{wants}[0] == \text{false}\ )$

Stąd:

- $\text{zapis}_A(\ \text{wants}[0] = \text{true}\ ) \rightarrow \text{odczyt}_A(\ \text{wants}[1] == \text{false}\ )$
- $\rightarrow \text{zapis}_B(\ \text{wants}[1] = \text{true}\ )$

Algorytm 1. – dowód WW

Z założenia wynika, że:

1. $\text{odczyt}_A(\ \text{wants}[1] == \text{false}\ ) \rightarrow \text{zapis}_B(\ \text{wants}[1] = \text{true}\ )$
2. $\text{odczyt}_B(\ \text{wants}[0] == \text{false}\ ) \rightarrow \text{zapis}_A(\ \text{wants}[0] = \text{true}\ )$

Z zapisu algorytmu wynika, że:

3. $\text{zapis}_A(\ \text{wants}[0] = \text{true}\ ) \rightarrow \text{odczyt}_A(\ \text{wants}[1] == \text{false}\ )$
4. $\text{zapis}_B(\ \text{wants}[1] = \text{true}\ ) \rightarrow \text{odczyt}_B(\ \text{wants}[0] == \text{false}\ )$

Stąd:

- $\text{zapis}_A(\ \text{wants}[0] = \text{true}\ ) \rightarrow \text{odczyt}_A(\ \text{wants}[1] == \text{false}\ )$
- $\rightarrow \text{zapis}_B(\ \text{wants}[1] = \text{true}\ )$
- $\rightarrow \text{odczyt}_B(\ \text{wants}[0] == \text{false}\ )$

Algorytm 1. – dowód WW

Z założenia wynika, że:

1. $\text{odczyt}_A(\ \text{wants}[1] == \text{false}\ ) \rightarrow \text{zapis}_B(\ \text{wants}[1] = \text{true}\ )$
2. $\text{odczyt}_B(\ \text{wants}[0] == \text{false}\ ) \rightarrow \text{zapis}_A(\ \text{wants}[0] = \text{true}\ )$

Z zapisu algorytmu wynika, że:

3. $\text{zapis}_A(\ \text{wants}[0] = \text{true}\ ) \rightarrow \text{odczyt}_A(\ \text{wants}[1] == \text{false}\ )$
4. $\text{zapis}_B(\ \text{wants}[1] = \text{true}\ ) \rightarrow \text{odczyt}_B(\ \text{wants}[0] == \text{false}\ )$

Stąd:

- $\text{zapis}_A(\ \text{wants}[0] = \text{true}\ ) \rightarrow \text{odczyt}_A(\ \text{wants}[1] == \text{false}\ )$
- $\rightarrow \text{zapis}_B(\ \text{wants}[1] = \text{true}\ )$
- $\rightarrow \text{odczyt}_B(\ \text{wants}[0] == \text{false}\ )$
- $\rightarrow \text{zapis}_A(\ \text{wants}[0] = \text{true}\ )$

Algorytm 1. – dowód WW

Z założenia wynika, że:

1. $\text{odczyt}_A(\ \text{wants}[1] == \text{false}\ ) \rightarrow \text{zapis}_B(\ \text{wants}[1] = \text{true}\ )$
2. $\text{odczyt}_B(\ \text{wants}[0] == \text{false}\ ) \rightarrow \text{zapis}_A(\ \text{wants}[0] = \text{true}\ )$

Z zapisu algorytmu wynika, że:

3. $\text{zapis}_A(\ \text{wants}[0] = \text{true}\ ) \rightarrow \text{odczyt}_A(\ \text{wants}[1] == \text{false}\ )$
4. $\text{zapis}_B(\ \text{wants}[1] = \text{true}\ ) \rightarrow \text{odczyt}_B(\ \text{wants}[0] == \text{false}\ )$

Stąd:

- $\text{zapis}_A(\ \text{wants}[0] = \text{true}\ ) \rightarrow \text{odczyt}_A(\ \text{wants}[1] == \text{false}\ )$
- $\rightarrow \text{zapis}_B(\ \text{wants}[1] = \text{true}\ )$
- $\rightarrow \text{odczyt}_B(\ \text{wants}[0] == \text{false}\ )$
- $\rightarrow \text{zapis}_A(\ \text{wants}[0] = \text{true}\ )$

Sprzeczność: $\text{zapis}_A(\ \text{wants}[0] = \text{true}\ ) \rightarrow \text{zapis}_A(\ \text{wants}[0] = \text{true}\ )$

Algorytm 1. – zakleszczanie

- Algorytm 1. zapewnia wzajemne wykluczanie,
- Jednak może prowadzić do zakleszczenia w przypadku jednoczesnego wykonania

Algorytm 1. – zakleszczanie

- Algorytm 1. zapewnia wzajemne wykluczanie,
- Jednak może prowadzić do zakleszczenia w przypadku jednoczesnego wykonania

```
1 public void lock() {
2     wants[i] = true;
3     while (wants[j] == true) { ; }
4 }
```

```
1 public void lock() {
2     wants[j] = true;
3     while (wants[i] == true) { ; }
4 }
```

Algorytm 1. – zakleszczanie

- Algorytm 1. zapewnia wzajemne wykluczanie,
- Jednak może prowadzić do zakleszczenia w przypadku jednoczesnego wykonania

```
1 public void lock() {
2     wants[i] = true;
3     while (wants[j] == true) { ; }
4 }
```

```
1 public void lock() {
2     wants[j] = true;
3     while (wants[i] == true) { ; }
4 }
```

- Jeżeli jednak wątki wykonują się *jeden po drugim* to algorytm się sprawdza

WW dla dwóch wątków – algorytm 2.

```
1  public class LockTwo implements Lock {
2      private int who_waits; // kto ustępuje pierwszeństwa
3
4      public void lock() {
5          int i = ThreadID.get(); // mój identyfikator
6          who_waits = i; // daj pierwszeństwo drugiemu wątkowi
7          while (who_waits == i) { ; } // czekaj na swoją kolej
8      }
9
10     public void unlock() {}
11 }
```

Algorytm 2. – analiza

- Algorytm zapewnia wzajemne wykluczanie:
  - Jeżeli wątek *i* jest wewnątrz swojej SK
  - To znaczy, że zmienna `who_waits == j`
  - Zmienna `who_waits` nie może mieć jednocześnie dwóch wartości

Algorytm 2. – analiza

- Algorytm zapewnia wzajemne wykluczanie:
  - Jeżeli wątek *i* jest wewnątrz swojej SK
  - To znaczy, że zmienna `who_waits == j`
  - Zmienna `who_waits` nie może mieć jednocześnie dwóch wartości
- Niestety, algorytm prowadzi do zakleszczenia:
  - Gdy jeden wątek wykonuje się w *całości* przed drugim
  - Jeżeli jednocześnie to OK

WW dla dwóch wątków – algorytm Petersona

- A może by tak połączyć Algorytm 1. z Algorytmem 2. ?

WW dla dwóch wątków – algorytm Petersona

- A może by tak połączyć Algorytm 1. z Algorytmem 2. ?

```
1  ...
2  public void lock()
3      ...
4      wants[i] = true;
5      who_waits = i;
6      while (wants[j] == true && who_waits == i) { ; }
7  }
8
9  public void unlock() {
10     ...
11     wants[i] = false;
12 }
```

WW dla dwóch wątków – algorytm Petersona

- A może by tak połączyć Algorytm 1. z Algorytmem 2. ?

```
1  ...
2  public void lock()
3      ...
4      wants[i] = true; // Chce wejść do SK
5      who_waits = i;     // Ustępuje pierwszeństwa
6      // Czeka dopóki
7      while (wants[j] == true // drugi również chce wejść do SK
8              && who_waits == i) // trzeba ustąpić pierwszeństwa
9      { ; }
10 }
11
12 public void unlock() {
13     ...
14     wants[i] = false; // Informuj o wyjściu z SK
15 }
```

Algorytm Petersona – WW

```
1 public void lock()
2     ...
3     wants[i] = true;
4     who_waits = i;
5     while (wants[j] == true && who_waits == i) { ; }
6 }
```

- Jeżeli wątek 0. jest w SK to:
  - `wants[0] = true`
  - `who_waits = 1`
- Jeżeli wątek 1. jest w SK to:
  - `wants[1] = true`
  - `who_waits = 0`

Algorytm Petersona – WW

```
1 public void lock()
2     ...
3     wants[i] = true;
4     who_waits = i;
5     while (wants[j] == true && who_waits == i) { ; }
6 }
```

- Jeżeli wątek 0. jest w SK to:
  - `wants[0] = true`
  - `who_waits = 1`
- Jeżeli wątek 1. jest w SK to:
  - `wants[1] = true`
  - `who_waits = 0`

Obie konfiguracje nie mogą zachodzić jednocześnie $\longrightarrow$ oba wątki nie mogą być jednocześnie w SK

Algorytm Petersona – brak zakleszczania

```
1 public void lock()
2     ...
3     while (wants[j] == true && who_waits == i) { ; }
4 }
```

- Wątki czekają tylko w pętli `while`
- Tylko jeden z nich może ustępować drugiemu (`who_waits = 0` albo `who_waits = 1`)
- Drugi wchodzi do sekcji krytycznej, którą musi zakończyć zgodnie z warunkiem postępu obliczeń SK

Algorytm Petersona – brak zagłodzenia

- Wątek *i* oczekuje, tylko jeżeli wątek *j* jest w sekcji krytycznej
- Przy próbie ponownego wejścia do SK, wątek *j* ustala `who_waits = j` , tak więc wątek *i* otrzymuje szansę wejścia do SK

```
1  public void lock()
2      wants[i] = true;
3      who_waits = i;
4      while (wants[j] == true
5              && who_waits == i)
6      { ; }
7  }
8  public void unlock() {
9      wants[i] = false;
10 }
```

Właściwości alg. Petersona

- Wzajemne wykluczanie: tak
- Brak zakleszczenia: tak
- Brak zagłodzenia: tak, pod warunkiem zastosowania **słabo uczciwego** szeregowania procesów

Słabo uczciwe szeregowanie (ang. weakly fair scheduling) gwarantuje, że jeżeli proces żąda dostępu do zasobu, to *w końcu*, tj. w skończonym czasie, ten zasób otrzyma.

Algorytm Piekarniany (L. Lamport)

Opis algorytmu:

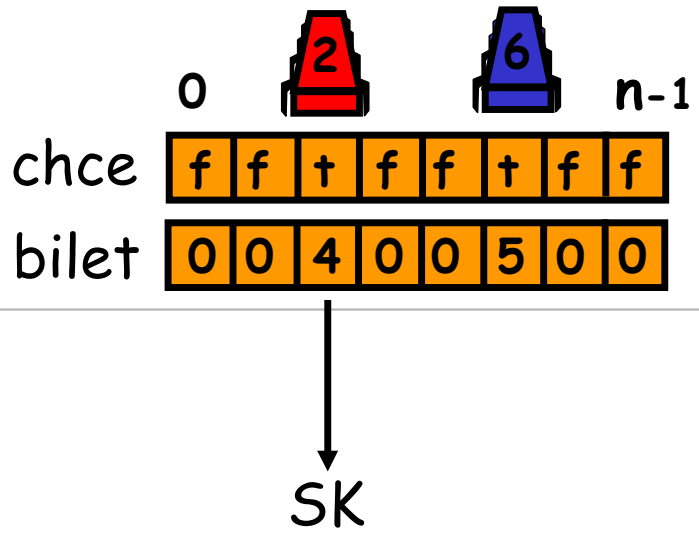
- Dostęp procesów do sekcji krytycznej regulowany jest na podstawie **biletów**
- Proces, który chce wejść do sekcji krytycznej musi pobrać **bilet** z numerem większym, niż każdy z dotychczas nieobsłużonych procesów
- Proces czeka, aż jego bilet będzie miał *najmniejszą* wartość spośród biletów *wszystkich* procesów, a następnie wchodzi do sekcji krytycznej
- Leslie Lamport w 2013 r. otrzymał nagrodę Turinga za wkład w teorię i praktykę systemów rozproszonych i współbieżnych

Algorytm Piekarniany

```
1  class Bakery implements Lock {
2      boolean[] chce;
3      Bilet [] bilet;
4      public Bakery (int n) { // Konstruktor
5          chce  = new boolean[n];
6          bilet = new Bilet[n];
7          for (int i = 0; i < n; i++) {
8             chce[i] = false;
9             bilet[i] = 0;
10         }
11     }
12 }
```

Algorytm Piekarniany

```
1  class Bakery implements Lock {
2      boolean[] chce;
3      Bilet [] bilet;
4      public Bakery (int n) { // Konstruktor
5          chce  = new boolean[n];
6          bilet = new Bilet[n];
7          for (int i = 0; i < n; i++) {
8              chce[i] = false;
9              bilet[i] = 0;
10         }
11     }
12 }
```



Algorytm Piekarniany – protokół wstępny

```
1 class Bakery implements Lock {
2 public void lock() {
3     chce[i]  = true;
4     bilet[i] = max( bilet[0], ..., bilet[n-1] ) + 1;
5
6     while ( Exists k: chce[k] == true
7                         (bilet[k], k) < (bilet[i], i) )
8    { ; }
9 }
```

- Uwaga: Porównanie w wierszu 7. dotyczy porządku *leksykograficznego*:
  - `(a, i) < (b, j)` gdy:
    - `a < b` lub
    - `a == b` i `i < j`

Algorytm Piekarniany – protokół końcowy

```
1 class Bakery implements Lock {
2     ...
3     public void unlock() {
4         chce[i] = false;
5     }
6 }
```

Algorytm Piekarniany dla 2 procesów

```
1 // zmienne globalne
2 bilet_a = 0
3 bilet_b = 0
```

Proces A

```
1 while (true) {
2     ...
3     bilet_a = bilet_b + 1;
4     while (bilet_b != 0
5         || bilet_a <= bilet_b)
6     { ; } // czekaj
7     ... // sekcja krytyczna
8     bilet_a = 0; // nie chce
9 }
```

Proces B

```
1 while (true) {
2     ...
3     bilet_b = bilet_a + 1;
4     while (bilet_a != 0
5         || bilet_b <= bilet_a)
6     { ; } //czekaj
7     ... // sekcja krytyczna
8     bilet_b; = 0 // nie chce
9 }
```

Algorytm Piekarniany dla 2 procesów ma własności wzajemnego wykluczania oraz nie prowadzi do zakleszczenia lub zagłodzenia.

Algorytm Piekarniany

Zalety algorytmu piekarnianego:

- ma własność wzajemnego wykluczania
- nie prowadzi do zakleszczenia
- jest **uczciwy** – procesy otrzymują pozwolenie na wejście do sekcji krytycznej zgodnie z kolejnością żądań (FIFO)
- jest prosty

Wady:

- numery biletów rosną nieograniczenie
- każdy proces musi odczytywać numery biletów wszystkich innych procesów zanim wejdzie do sekcji krytycznej

Wady rozwiązań programowych

Rozwiązywanie problemu wzajemnego wykluczania tylko za pomocą *atomowych* instrukcji odczytu i zapisu jest:

- wolne – procesy, które chcą wejść do sekcji krytycznej zużywają czas procesora niepotrzebnie, np. przez cykliczne sprawdzanie warunku, który nie może być spełniony – aktywne oczekiwanie (ang. spinlock)
- wymaga dodatkowej pamięci o rozmiarze liniowo rosnącym wraz z liczbą procesorów

Wady rozwiązań programowych c.d.

- Lepszym mechanizmem jest korzystanie z blokad (zamków), które powodują uśpienie wątku na pewien czas – np. muteksy, semafory, monitory
- Czasem, w systemach wieloprocesorowych, mechanizm aktywnego oczekiwania może być efektywniejszy – tańszy, niż przełączanie kontekstu, jeżeli aktywnych procesów jest nie więcej, niż procesorów

Rozwiązania sprzętowe

- Wyłączenie przerwań:
  - „Normalnie” proces wykonuje obliczenia dopóki nie zostanie przerwany – wywoła przerwanie systemowe lub zostanie wywłaszczony
  - Wyłączenie przerwań gwarantuje, że wykonanie procesu nie zostanie przerwane, więc nie ma problemu z wzajemnym wykluczaniem
  - Ograniczona możliwość szeregowania zadań
  - W systemach wieloprocesorowych wyłączenie przerwań pojedynczego procesora nie gwarantuje wzajemnego wykluczania

Rozwiązania sprzętowe

- Specjalne rozkazy maszynowe:
  - wykonanie w sposób **atomowy** bardziej złożonej operacji, niż tylko odczyt i zapisu komórki pamięci – rozkazy typu *odczytaj–zmodyfikuj–zapisz* (ang. Read-Modify-Write, RMW)
  - Przykładowe rozkazy: Test-And-Set, Fetch-And-Add, Compare-And-Swap

Test-And-Set

Programowy szkic implementacji instrukcji Test-And-Set:

```
1 boolean register;
2 boolean testAndSet() {
3     boolean previous = register; // zapisz poprzednią wartość
4     register = true;
5     return previous;
6 }
```

WW z użyciem Test-And-Set

```
1  boolean wants = false; // zmienna globalna
2  ...
3  while(true) {
4      sekcja lokalna
5      // Protokół wstępny:
6      while(wants.testAndSet(true) == true) {} // czekaj
7      // Sekcja Krytyczna
8      // ...
9      wants.set(false); // Protokół końcowy
10 }
```

- Ponieważ `testAndSet` jest instr. atomową, jeżeli kilka wątków jednocześnie wykona `wants.testAndSet(true)` tylko jeden z nich zobaczy wartość `false` i on wejdzie do sekcji krytycznej (wiersz 6)
- W Javie operacja Test-And-Set dostępna jest jako metoda `getAndSet` m.in. w klasach `AtomicBoolean`, `AtomicInteger` z pakietu `java.util.concurrent`

Mechanizmy zamków (blokad) w Javie

- `Lock`, `ReentrantLock`
- Konieczność stosowania tych samych zamków dla tych samych sekcji krytycznych i w tej samej kolejności
- Sekcje krytyczne niepowiązane ze sobą (zmienne nie zależą od zmiennych w innej sekcji bezpośrednio i pośrednio) mogą być realizowane współbieżnie jeżeli będą chronione *odrębnymi* blokadami

Lock c.d.

Interfejs `Lock` udostępnia również metodę `tryLock()`, która próbuje uzyskać zamek, ale jeżeli się to nie uda, to nie wstrzymuje wykonania wątku

Dowodzenie poprawności programów

- Jeżeli chcemy mieć pewność, że nasz program sekwencyjny jest poprawny, tj. nie zawiera błędów, to powinniśmy **udowodnić** jego poprawność
- Dowód poprawności pokazuje, że pewne „właściwości”, czyli **niezmienniki** naszego programu są prawdziwe w *każdym* możliwym stanie jego wykonania

Dowodzenie poprawności programów

- Jeżeli chcemy mieć pewność, że nasz program sekwencyjny jest poprawny, tj. nie zawiera błędów, to powinniśmy **udowodnić** jego poprawność
- Dowód poprawności pokazuje, że pewne „właściwości”, czyli **niezmienniki** naszego programu są prawdziwe w *każdym* możliwym stanie jego wykonania
- Stan procesu to wektor określający:
  - położenie w diagramie stanów (licznik programu)
  - aktualne wartości wszystkich zmiennych lokalnych i globalnych

Niezmienniki

- **Niezmiennikiem** programu nazywamy każdą formułę logiczną (stanowiącą o wartościach zmiennych i wskaźników instrukcji) prawdziwą w każdym *stanie* wykonania programu
- Dowodzenie poprawności programu wymaga dowiedzenia zachowania niezmienników we wszystkich możliwych stanach jego wykonania

Dowodzenie poprawności – metoda indukcji

- **Indukcyjne dowodzenie niezmienników** – dowód spełnienia wszystkich niezmienników o dotyczących warunków bezpieczeństwa i postępu wykonania algorytmu na podstawie:
  - rachunku predykatów
  - logiki temporalnej
  - dowodu nie wprost (łac. reductio ad absurdum) – sprowadzenie do sprzeczności

Dowodzenie poprawności – metoda indukcji

- **Indukcyjne dowodzenie niezmienników** – dowód spełnienia wszystkich niezmienników o dotyczących warunków bezpieczeństwa i postępu wykonania algorytmu na podstawie:
  - rachunku predykatów
  - logiki temporalnej
  - dowodu nie wprost (łac. reductio ad absurdum) – sprowadzenie do sprzeczności
- Pokazujemy, że pewne własności – **niezmienniki** są spełnione na początku programu
- Znajdujemy te fragmenty (instrukcje) naszego programu, w których któryś z niezmienników mógłby ulec zmianie (np. na skutek zmiany wartości zmiennych)

Dowodzenie poprawności – metoda indukcji

- **Indukcyjne dowodzenie niezmienników** – dowód spełnienia wszystkich niezmienników o dotyczących warunków bezpieczeństwa i postępu wykonania algorytmu na podstawie:
  - rachunku predykatów
  - logiki temporalnej
  - dowodu nie wprost (łac. reductio ad absurdum) – sprowadzenie do sprzeczności
- Pokazujemy, że pewne własności – **niezmienniki** są spełnione na początku programu
- Znajdujemy te fragmenty (instrukcje) naszego programu, w których któryś z niezmienników mógłby ulec zmianie (np. na skutek zmiany wartości zmiennych)
- Pokazujemy, że taka zmiana nie zachodzi

Dowodzenie poprawności – metoda indukcji

Niezmienniki dowodzi się przez *indukcję* po stanach obliczeń (wszystkich możliwych przejściach w diagramie stanów programu)

1. Udowodnij, że ϕ zachodzi w stanie początkowym – krok podstawowy indukcji
2. Załóż, że ϕ jest prawdziwe we wszystkich stanach, aż do bieżącego i wykaż, że ϕ jest prawdziwe w stanie następnym – krok indukcyjny

Jeżeli można udowodnić 1. i 2. to ϕ zachodzi dla wszystkich stanów obliczeń

Dowodzenie – programy współbieżne

- Opis programu składa się ze zbioru formuł logicznych
- Specyfikacja własności programu również stanowi formułę logiczną
- Weryfikacja poprawności programu współbieżnego sprowadza się do udowodnienia, że formuły określające właściwości **bezpieczeństwa** i **żywotności** są spełnione

Logika temporalna

- W dowodzeniu popr. PW przydatna jest **logika temporalna**
- Logika temporalna zakłada **zmienność w czasie** prawdziwości poszczególnych formuł atomowych. Przyjmujemy, że czas jest dyskretny (izomorficzny ze zbiorem liczb naturalnych)
- Składnia zdaniowej logiki temporalnej:
  - symbole formuł atomowych $p, q, r, \ldots$ są formułami
  - jeżeli ϕ, ψ są formułami to $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$ są formułami
  - jeżeli ϕ, ψ są formułami to $\Box\phi$, $\Diamond\phi$, $\bigcirc\phi$, $\phi\mathcal{U}\psi$ są formułami

Operatory temporalne

Operatory temporalne (czasu przyszłego):

- $\Box\phi$ – „zawsze ϕ" – formuła $\Box\phi$ jest prawdziwa w stanie s_i pewnego obliczenia, wtedy i tylko wtedy, gdy ϕ jest prawdziwa we wszystkich stanach s_j dla $j \geqslant i$.
- $\Diamond\phi$ – „kiedyś (w końcu) ϕ" – formuła $\Diamond\phi$ jest prawdziwa w stanie s_i, wtedy i tylko wtedy, gdy ϕ jest prawdziwa w pewnym stanie s_j dla $j \geqslant i$.

Operatory temporalne

- $\bigcirc\phi$ – „w następnej chwili ϕ" – formuła ϕ jest prawdziwa w stanie s_i, wtedy i tylko wtedy, gdy jest prawdziwa w stanie następnym, tj. s_{i+1}
- $\phi\mathcal{U}\psi$ – „ϕ jest prawdziwe, aż do ψ" – $\phi\mathcal{U}\psi$ jest prawdziwa w stanie s_i wtedy i tylko wtedy, gdy ψ jest prawdziwa w pewnym stanie $s_j, j \geqslant i$ oraz ϕ jest prawdziwa we wszystkich stanach $s_k, i \leqslant k < j$.

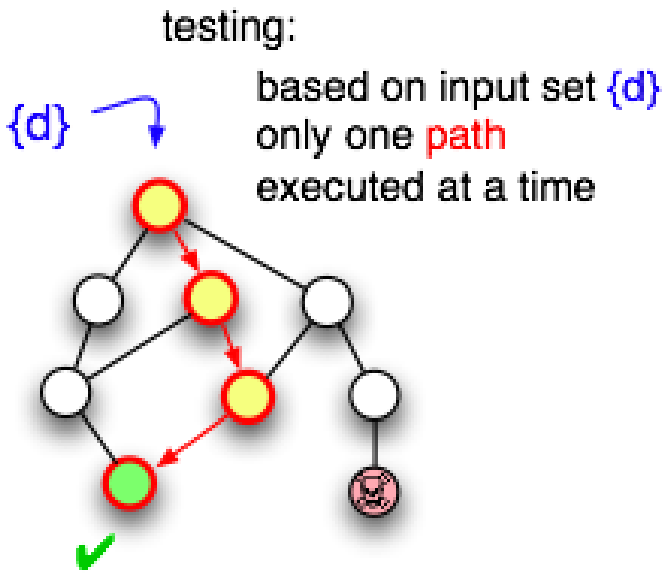
Dowodzenie – weryfikacja przez model

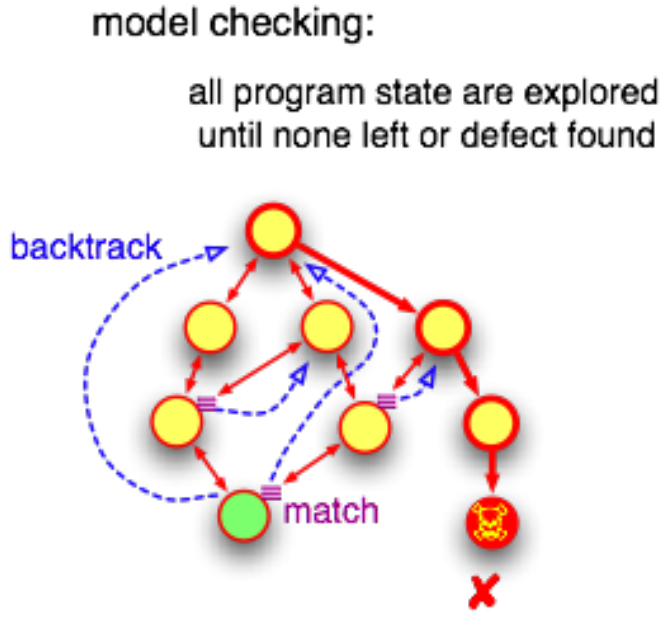
- **Weryfikacja przez model** – wykonanie modelu algorytmu w postaci maszyny stanów (np. NAS), w której poszczególne stany obrazują wszystkie możliwe etapy wykonania algorytmu i wykazanie, że program nie może znaleźć się w niepoprawnym stanie
- Tutaj również podajemy zbiór niezmienników, które powinny być zachowane
- Sprawdzanie wykonywane jest w sposób *automatyczny* za pomocą oprogramowania
- Weryfikacja przez model została zaproponowana przez Edmunda Clarke, Allen Emersona i Josepha Sifakisa, którzy w 2008 otrzymali za nią nagrodę Turinga

Testowanie a poprawność PW

- **Testowanie programu** przez wielokrotne uruchamianie nie jest dowodem jego poprawności:
  - Zazwyczaj nie da się zapewnić *wszystkich możliwych* przeplotów wykonania programu,
  - Zazwyczaj nie da się sprawdzić algorytmu dla wszystkich możliwych zestawów danych wejściowych

Testowanie a weryfikacja przez model





Przykład – współbieżne zwiększanie licznika

```
1  integer n = 0;
2
3  process P
4      integer regP = 0;
5      p1: load n into regP
6      p2: increment regP
7      p3: store regP into n
8      p4: end
9
10 process Q
11     integer regQ = 0;
12     q1: load n into regQ
13     q2: increment regQ
14     q3: store regQ into n
15     q4: end
```

Przykład zaczerpnięty z: Ben-Ari, Mordechai Moti. "A primer on model checking." ACM Inroads 1.1 (2010): 40-47.

Przykład – współbieżne zwiększanie licznika

```
1  integer n = 0;
2
3  process P
4      integer regP = 0;
5      p1: load n into regP
6      p2: increment regP
7      p3: store regP into n
8      p4: end
9
10 process Q
11     integer regQ = 0;
12     q1: load n into regQ
13     q2: increment regQ
14     q3: store regQ into n
15     q4: end
```

Przykład – współbieżne zwiększanie licznika

```
1  integer n = 0;
2
3  process P
4      integer regP = 0;
5      p1: load n into regP
6      p2: increment regP
7      p3: store regP into n
8      p4: end
9
10 process Q
11     integer regQ = 0;
12     q1: load n into regQ
13     q2: increment regQ
14     q3: store regQ into n
15     q4: end
```

Stan w tym programie składa się z pięciu elementów:

- wskaźników instrukcji dla obu procesów (wątków), np. `IP(P) = p1`, `IP(Q) = q3`
- wartości zmiennej `n`
- wartości rejestrów: `regP`, `regQ`

Przykład – współbieżne zwiększanie licznika

```
1  integer n = 0;
2
3  process P
4      integer regP = 0;
5      p1: load n into regP
6      p2: increment regP
7      p3: store regP into n
8      p4: end
9
10 process Q
11     integer regQ = 0;
12     q1: load n into regQ
13     q2: increment regQ
14     q3: store regQ into n
15     q4: end
```

Stan w tym programie składa się z pięciu elementów:

- wskaźników instrukcji dla obu procesów (wątków), np. `IP(P) = p1`, `IP(Q) = q3`
- wartości zmiennej `n`
- wartości rejestrów: `regP`, `regQ`

W skrócie:
(IP(P)=p3, IP(Q)=q1, regP=1, regQ=0, n=0)
lub
(p3, q1, 1, 0, 0)

Przykład – współbieżne zwiększanie licznika c.d.

```
1  integer n = 0;
2
3  process P
4      integer regP = 0;
5      p1: load n into regP
6      p2: increment regP
7      p3: store regP into n
8      p4: end
9
10 process Q
11     integer regQ = 0;
12     q1: load n into regQ
13     q2: increment regQ
14     q3: store regQ into n
15     q4: end
```

Stanem początkowym jest:

`(p1, q1, 0, 0, 0)`

Przykład – współbieżne zwiększanie licznika c.d.

```
1  integer n = 0;
2
3  process P
4      integer regP = 0;
5      p1: load n into regP
6      p2: increment regP
7      p3: store regP into n
8      p4: end
9
10 process Q
11     integer regQ = 0;
12     q1: load n into regQ
13     q2: increment regQ
14     q3: store regQ into n
15     q4: end
```

Stanem początkowym jest:

`(p1, q1, 0, 0, 0)`

Przykładowe wykonanie:

(p1,q1,0,0,0) $\rightarrow$ (p2,q1,0,0,0) $\rightarrow$
(p3,q1,1,0,0) $\rightarrow$ (p4,q1,1,0,1) $\rightarrow$
(p4,q2,1,1,1) $\rightarrow$ (p4,q3,1,2,1) $\rightarrow$
(p4,q4,1,2, **2**)

Przykład – współbieżne zwiększanie licznika c.d.

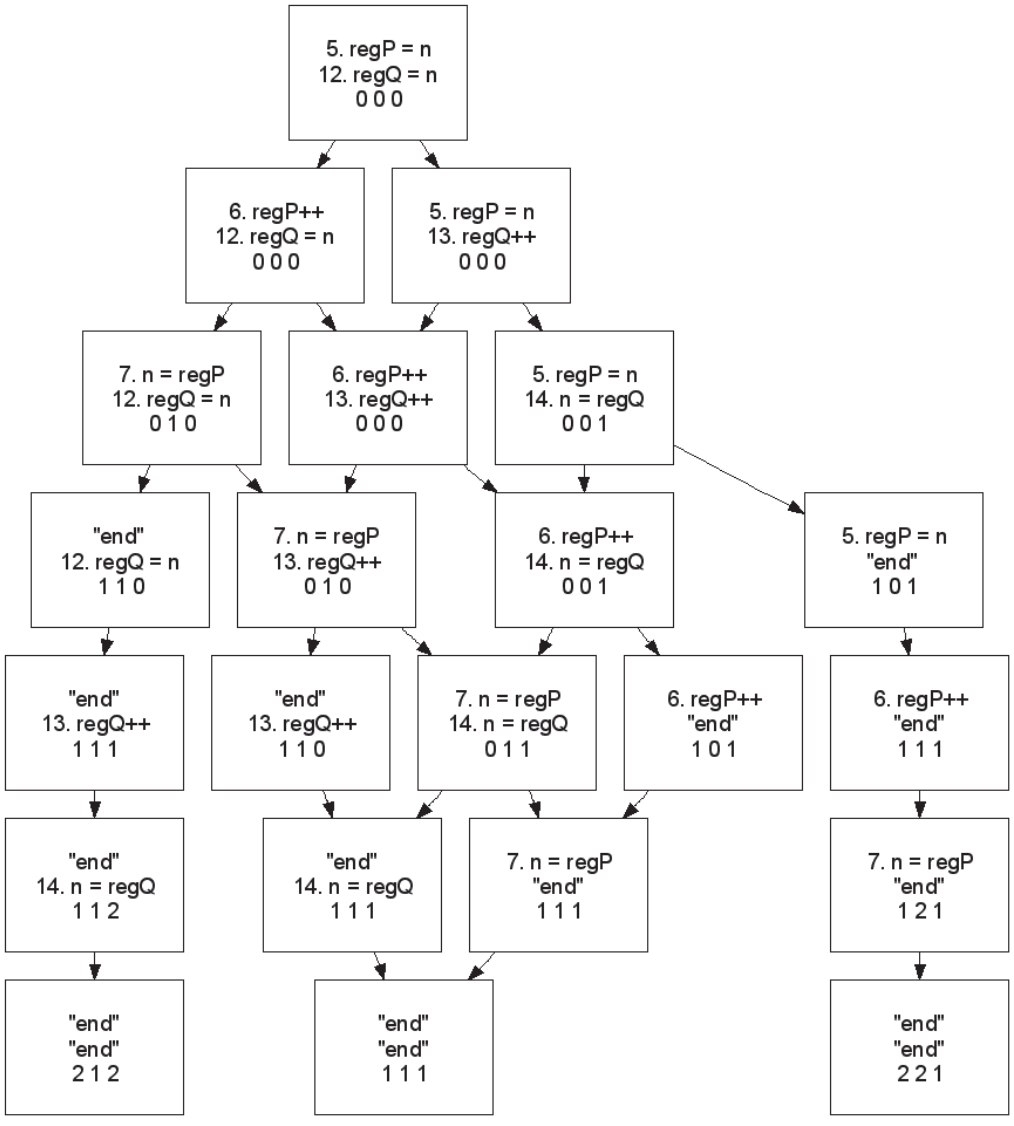
```
1  integer n = 0;
2
3  process P
4      integer regP = 0;
5      p1: load n into regP
6      p2: increment regP
7      p3: store regP into n
8      p4: end
9
10 process Q
11     integer regQ = 0;
12     q1: load n into regQ
13     q2: increment regQ
14     q3: store regQ into n
15     q4: end
```

Inne możliwe wykonanie:

(p1,q1,0,0,0) $\rightarrow$ (p2,q1,0,0,0) $\rightarrow$
(p2,q2,1,0,0) $\rightarrow$ (p3,q2,1,0,0) $\rightarrow$
(p3,q3,1,1,0) $\rightarrow$ (p4,q3,1,1,1) $\rightarrow$
(p4,q4,1,1,1)

W tym przypadku wyniki jest nieprawidłowy.

Przykład – współbieżne zwiększanie licznika c.d.



Oprogramowanie do weryfikacji przez model

- Spin – (http://spinroot.com) rozwijany przez Gerarda J. Holzmanna, NASA/JPL
- JSpin – Spin w wersji java
- JPF - Java Path Finder – http://babelfish.arc.nasa.gov/
- ThreadSafe http://www.contemplateltd.com/threadsafe

Spin

- Algorytm, który chcemy zweryfikować musimy zapisać w języku Promela
  - język o prostej składni, podobnej do Pascala
  - wzbogacony o elementy dotyczące współbieżności
  - niezmienniki określamy za pomocą asercji lub formuł liniowej logiki temporalnej
- Kod ten jest tłumaczony na kod w języku C, a następnie komilowany i uruchamiany – dzięki temu zachowana jest wysoka efektywność

Spin c.d.

- Spin można uruchamiać w kilku trybach:
  - tryb losowej symulacji – generator liczb pseudolosowych jest używany do symulacji niedeterminizmu wykonania programu współbieżnego
  - Tryb interaktywnej symulacji – użytkownik decyduje o ścieżce wybieranej przez program
  - tryb weryfikacji – Spin sprawdza wszystkie osiągalne stany w systemie i sprawdza, czy nie powodują one złamania któregoś z niezmienników

Spin – przykład

Algorytm w pseudokodzie

```
integer n = 0;

process P
    integer regP = 0;
    do 10 times
        load n into regP
        increment regP
        store regP into n
    end

process Q
    integer regQ = 0;
    do 10 times
        load n into regQ
        increment regQ
        store regQ into n
    end
```

Alg. w Promeli:

```
byte n = 0, finish = 0;

active [2] proctype P() {
    byte register, counter = 0;
    do :: counter = 10 -> break
       :: else ->
        register = n;
        register++;
        n = register;
        counter++
    od;
    finish++
}
active proctype WaitForFinish() {
    finish == 2;
    printf("n = \%d\n", n)
}
```