# Programowanie współbieżne
# Wykład 5

Rafał Skinderowicz

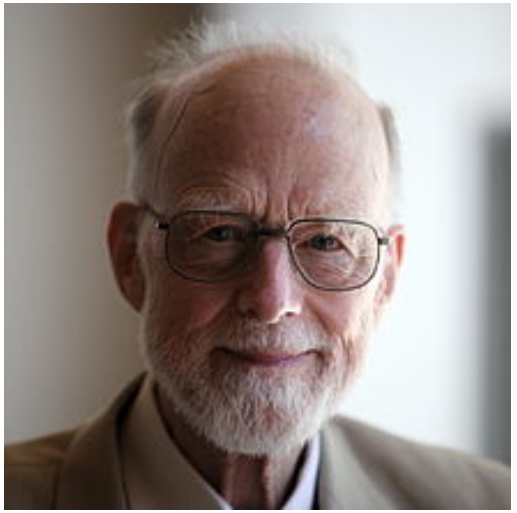
# Monitory – motywacje

- Mechanizmy synchronizacji takie jak
  - blokady (zamki)
  - semafory

  pozwalają efektywnie rozwiązywać dostępu do współdzielonych zasobów, jednak ich stosowanie staje się kłopotliwe w *złożonych* systemach.
- Sterowanie przydziałem do współdzielonych zasobów jest w takim przypadku *rozproszone* między wszystkie korzystające z niego wątki / procesy:
  - metody różnych klas zakładają jawnie te same blokady
  - kłóci się to z zasadą jednej odpowiedzialności (ang. *Single responsibility principle*)

# Monitory – motywacje

- Lepszym rozwiązaniem jest skupienie zarządzania dostępem do *danego* zasobu w jedną całość – monitor
- Monitory zostały zaproponowane przez Brincha Hansena, a następnie usprawnione przez Charlesa A.R. Hoare'a w 1974 r.



# Monitor

**Monitor**

**Monitor** to struktura danych oraz zbiór operacji (metod) służących do zarządzania dostępem do danego zasobu lub podzbioru zasobów.

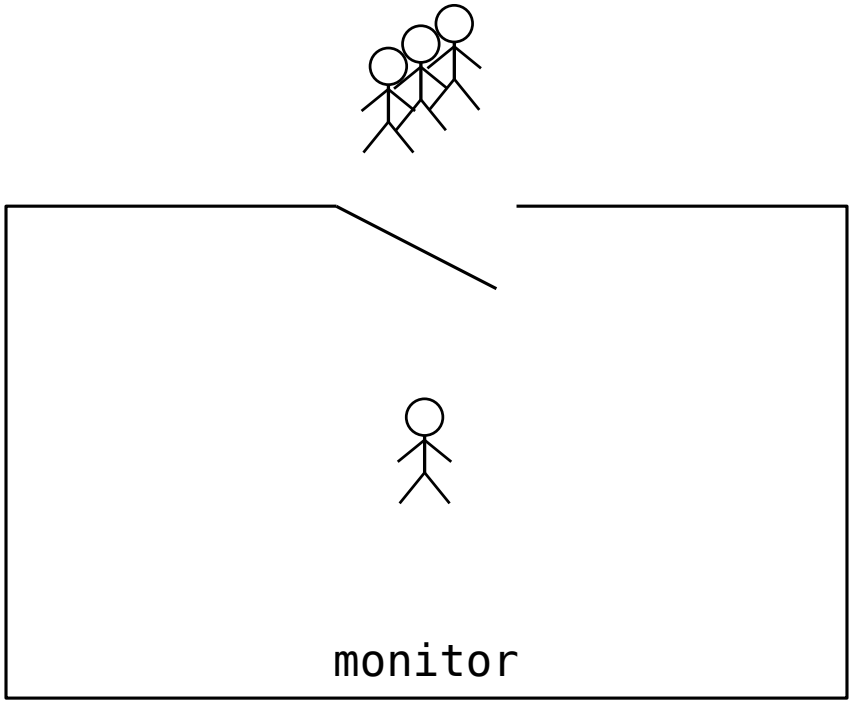
- W przypadku, gdy kilka procesów wywołuje jednocześnie operacje monitora, to będą one wykonane z zapewnieniem wzajemnego wykluczania.
- Mechanizmy konieczne do zapewnienia poprawnej synchronizacji *zamknięte* są wewnątrz monitora, stąd ich stosowanie jest wygodne dla klientów.

# Monitor – właściwości

Monitor grupuje dane (zmienne) oraz operacje na tych danych podobnie jak klasa w programowaniu obiektowym, jednak z pewnymi różnicami:

- Wszystkie pola monitora są prywatne, tzn. nie do nich bezpośredniego dostęp z zewnątrz monitora.
- Tylko jeden proces może jednocześnie wykonywać operację monitora, co gwarantuje zachowanie spójnych wartości pól monitora – wykonanie operacji monitora jest atomowe.
- Metody monitora nie powinny odwoływać się do żadnych zmiennych globalnych.

# Monitor – wyłączność dostępu




Ponieważ tylko jeden proces może wykonywać w danym momencie metodę monitora, to pozostałe procesy, które wywołały operację monitora muszą zostać wstrzymane i wstawione do kolejki oczekujących.

Po zakończeniu wykonywania operacji monitora budzony jest jeden z oczekujących procesów.

# Monitor – warunki

- W wielu przypadkach mogą zaistnieć dodatkowe powody, dla których proces powinien zostać wstrzymany już po wejściu do monitora
- Definiowane są tzw. warunki (ang. *conditions*), na których można wykonywać tylko dwie operacje:
  - wait() – usypia proces w oczekiwaniu na zajście warunku
  - signal() – sygnalizuje zajście warunku – powoduje wybudzenie procesu oczekującego na zajście warunku
- Z każdym warunkiem związana jest niejawna kolejka oczekujących procesów – początkowo pusta.
- Warunki czasem nazywane są *zmiennymi warunkowymi* – choć nie mają jawnej wartości, którą można odczytać – pełnią raczej rolę „pojemnika na wątki”

# Warunki monitorów a semafory

Semafor:

- `wait()` wstrzymuje proces lub nie
- `signal()` ma zawsze jakiś efekt – zmienia wartość semafora / budzi wątek
- proces wybudzany przez `signal()` zależy od implementacji semafora (silny albo słaby)

Warunek:

- `wait()` zawsze wstrzymuje proces
- `signal()` ma efekt przy niepustej kolejce
- `signal()` wznawia pierwszy proces z kolejki wstrzymanych

# Monitor – przykład

Przykład monitora dla rozwiązania problemu wzajemnego wykluczania.

**monitor** Dozorca
    boolean zajęty = false
    condition wolny
    **operation** przydziel()
        if zajęty
            wolny.wait()
        zajęty = true
    **operation** zwolnij()
        zajęty = false
        wolny.signal()

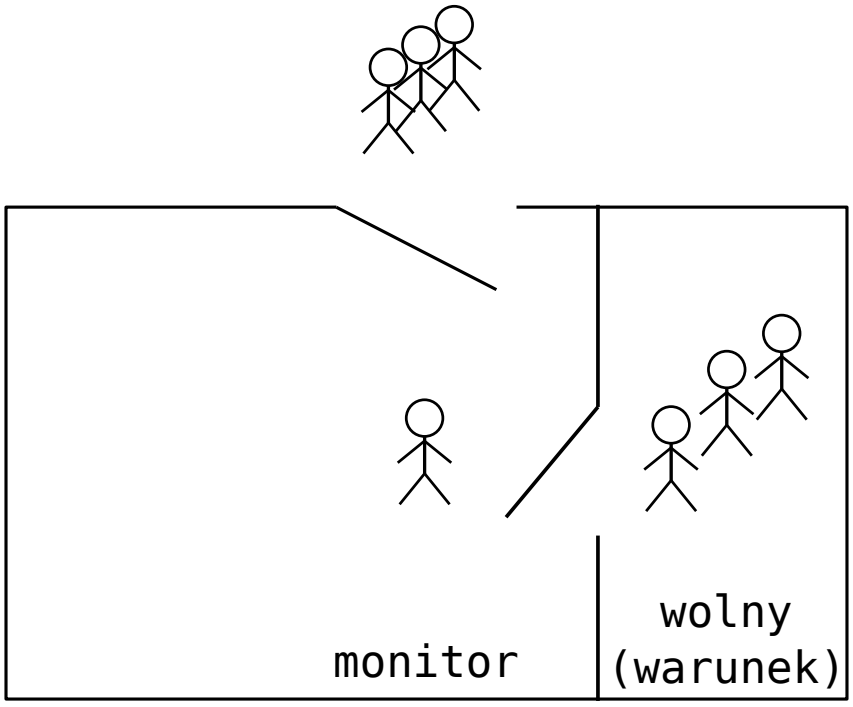
Metody `przydziel` i `zwolnij` odpowiadają znanym z blokad metodom `lock` i `unlock`

# Monitor – przykład

```
// Proces P
while (true) {
    sekcja lokalna
    Dozorca.przydziel()
    sekcja krytyczna
    Dozorca.zwolnij()
}
```

```
// Proces Q
while (true) {
    sekcja lokalna
    Dozorca.przydziel()
    sekcja krytyczna
    Dozorca.zwolnij()
}
```

# Monitor – przykład




Z każdym warunkiem związana jest niejawna kolejka oczekujących procesów.

# Monitor – kolejność wznawiana procesów

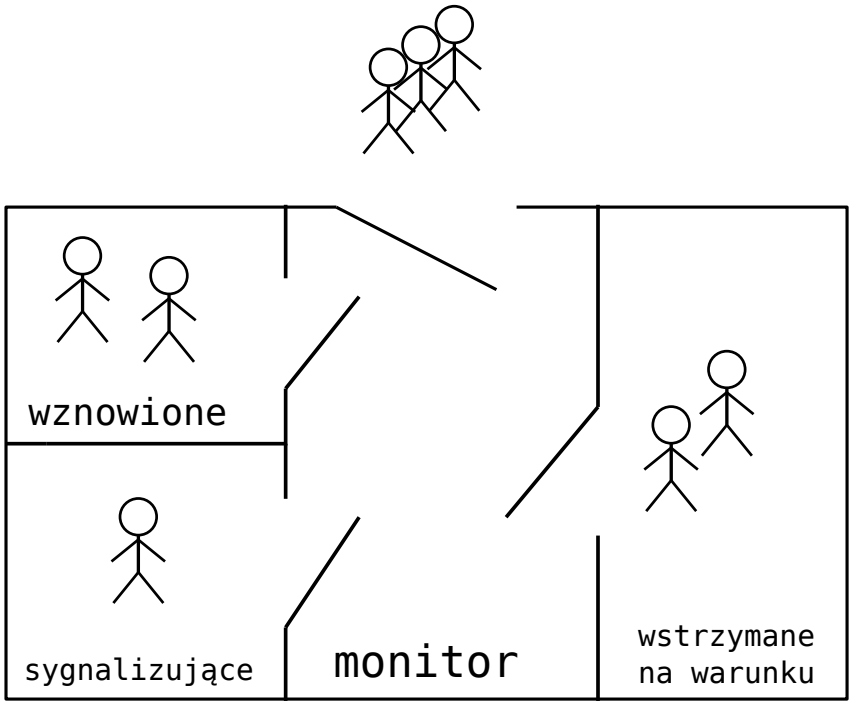
- Definicja monitora wymaga, by po sygnalizacji warunku natychmiast został wznowiony jeden z wstrzymanych na warunku procesów.
- Przykładowo, jeżeli proces $P$ wykonał operację `signal()` na warunku `wolny`, to automatycznie wznowiony powinien zostać proces $Q$ wstrzymany na tym warunku. Jednak oba procesy nie mogą wykonywać jednocześnie metod monitora, ponieważ ich wykonanie jest wyłączne – stąd jeden z nich musi zostać wstrzymany.
- Uzasadnienie – proces sygnalizujący zmienił stan monitora, w taki sposób, że warunek stał się prawdziwy, więc po wznowieniu proces oczekujący na warunku nie musi go sprawdzać (w teorii ☺)

# Kolejność wznawiana procesów

```
monitor Dozorca
    boolean zajęty = false
    condition wolny
    operation przydziel()
1       if zajęty
2           wolny.wait()
3       → zakładamy, że zajęty == false
4       zajęty = true
    operation zwolnij()
5       zajęty = false
6       wolny.signal()
```

Jeżeli po sygnalizacji warunku (wiersz 6) szansę wykonania operacji wait() otrzymałby proces inny, niż wcześniej wstrzymany na warunku, to ten ostatni po obudzeniu zastałby **zajęty = true** (wiersz 4)

# Kolejność wznawiana procesów

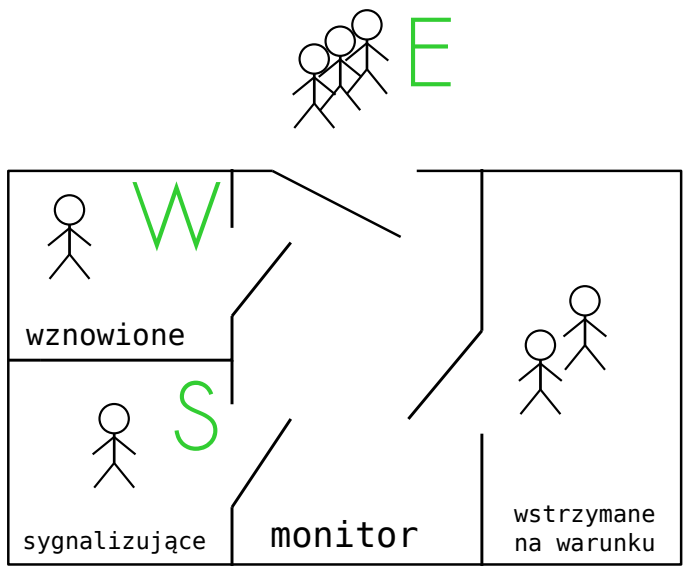



Ze względu na kilka kolejek wstrzymanych procesów należy ustalić priorytet ich wznawiania.

# Priorytety procesów

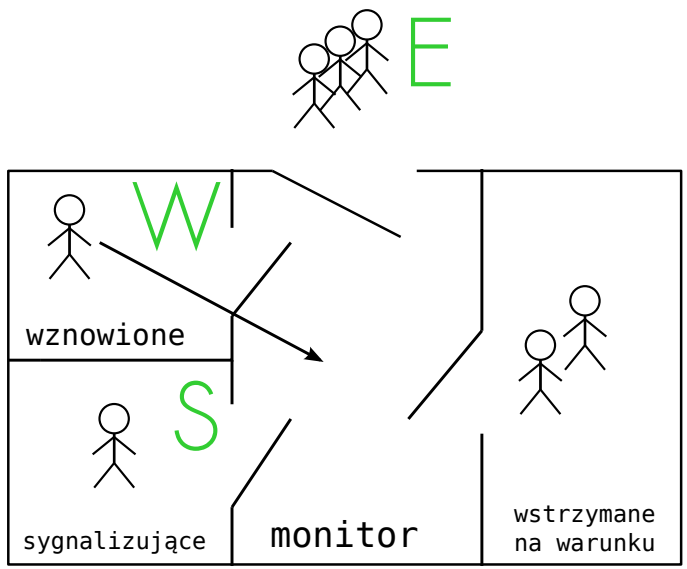
Procesy z pkt. widzenia monitora można podzielić na:

- wstrzymane na wejściu (priorytet E)
- sygnalizujące (priorytet S)
- wznowione z operacji wait() na warunku (priorytet W)



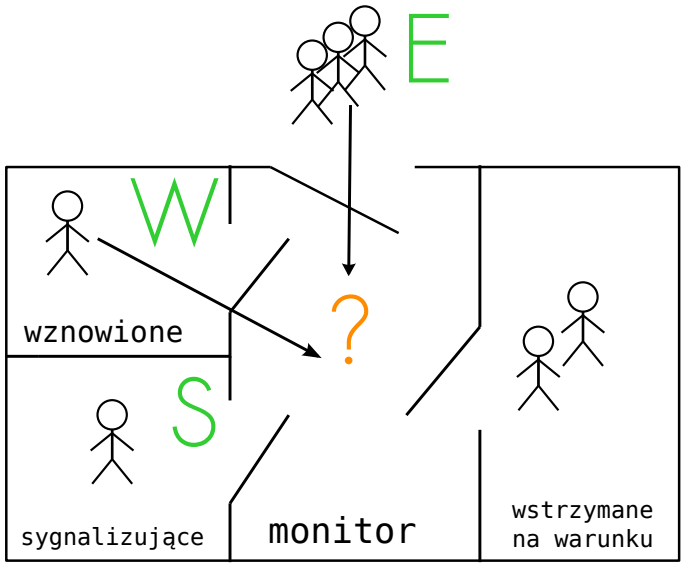
# Priorytety procesów

- W klasycznym monitorze zachodzi $E < S < W$, czyli największy priorytet mają procesy wznawiane z operacji wait().



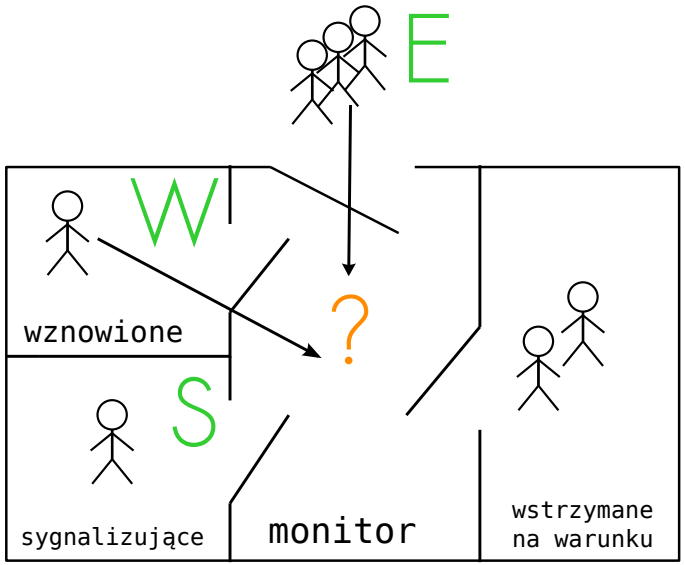
# Priorytety procesów

- Jeżeli warunek $E < S < W$ nie jest spełniony, to może to wymagać stosowania *dodatkowych operacji sprawdzania* stanu monitora. Przykładowo w Javie $E = W < S$.



# Priorytety procesów

- Jeżeli warunek $E < S < W$ nie jest spełniony, to może to wymagać stosowania *dodatkowych operacji sprawdzania* stanu monitora. Przykładowo w Javie $E = W < S$.



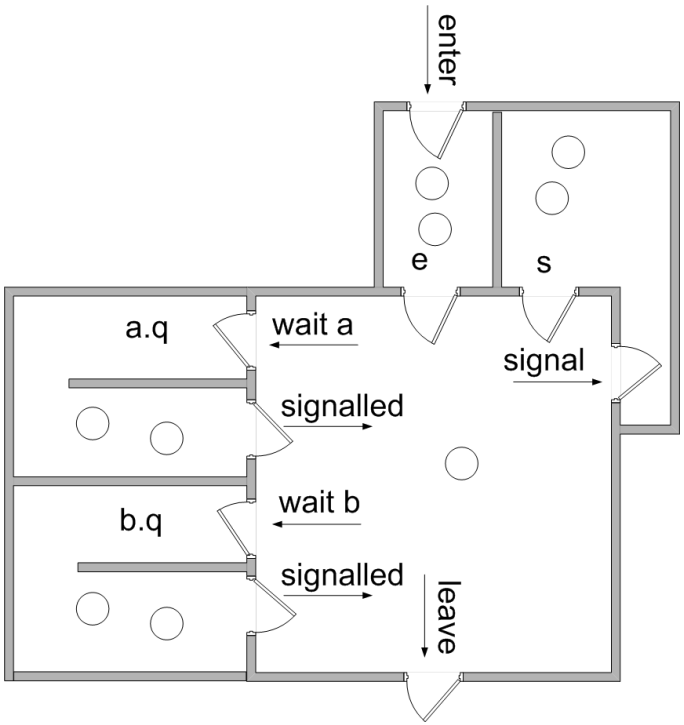

Zamiast:

```
1 if zajety
2     warunek.wait()
```

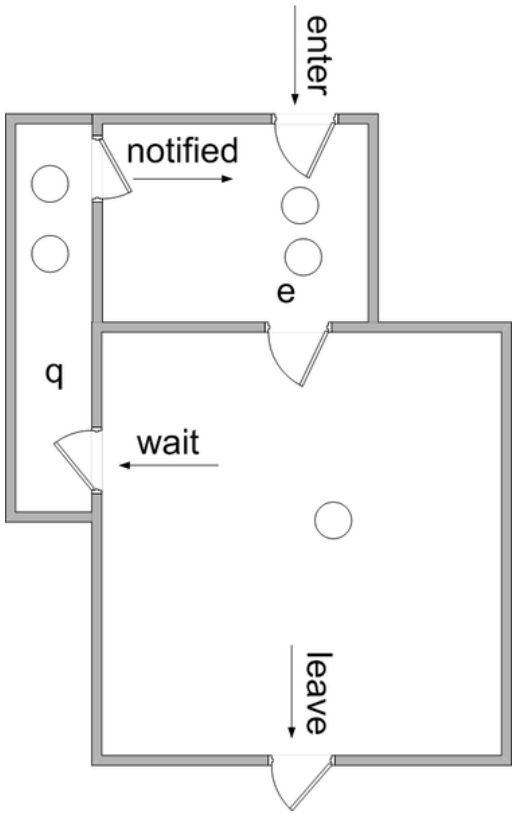
Trzeba użyć

```
1 while zajety
2     warunek.wait()
```

# Priorytety procesów w monitorze – Java




Rysunek 1: Monitor wg C.A.R. Hoarea (Autor: Theodore Norvell)




Rysunek 2: Monitor w Javie (Autor: Theodore Norvell)

# Problem czytelników i pisarzy

Dane są dwa rodzaje procesów: czytelnicy i pisarze, które mogą korzystać ze wspólnej czytelni.

- W czytelni jednocześnie mogą znajdować się jedynie procesy tego samego typu.
- Procesy czytelników mogą współdzielić czytelnię, ale wykluczają pisarzy.
- Procesy pisarzy żądają wyłącznego dostępu do czytelni, a więc wykluczają zarówno pozostałych pisarzy, jak i czytelników.

# Problem czytelników i pisarzy – rozwiązanie monitorowe

Najpierw sposób zastosowania monitora.

**Czytelnia** // monitor zarządzający czytelnią

```
// Proces Czytelnika
while (true) {
    Czytelnia.PoczątekCzytania()
    Czytaj()
    Czytelnia.KoniecCzytania()
}
```

```
// Proces Pisarza
while (true) {
    Czytelnia.PoczątekPisania()
    Pisz()
    Czytelnia.KoniecPisania()
}
```

# Problem czytelników i pisarzy – rozwiązanie monitorowe

```
1  Monitor Czytelnia
2      int czytelnicy = 0
3      int pisarze = 0
4      condition MożnaCzytać
5      condition MożnaPisać
6
7  operation PoczątekCzytania
8      if pisarze != 0
9          MożnaCzytać.wait()
10     czytelnicy = czytelnicy + 1
11     MożnaCzytać.signal()
12
13 operation KoniecCzytania
14     czytelnicy = czytelnicy - 1
15     if czytelnicy = 0
16         MożnaPisać.signal()
```

```
1  operation PoczątekPisania()
2      if pisarze != 0 or czytelnicy !=
         0
3          MożnaPisać.wait()
4      pisarze = pisarze + 1
5
6  operation KoniecPisania()
7      pisarze = pisarze - 1
8      MożnaCzytać.signal()
```

# Analiza

- Przestawione rozwiązanie jest poprawne:
  - zapewnia wzajemne wykluczanie w dostępie do czytelni
  - nie prowadzi do zakleszczenia
  - nie prowadzi do zagłodzenia czytelników

# Analiza

- Przestawione rozwiązanie jest poprawne:
  - zapewnia wzajemne wykluczanie w dostępie do czytelni
  - nie prowadzi do zakleszczenia
  - nie prowadzi do zagłodzenia czytelników
  - ale pisarze mogą być głodzeni

```
1 operation PoczątekCzytania
2     if pisarze != 0
3         MożnaCzytać.wait()
4     czytelnicy = czytelnicy + 1
5     MożnaCzytać.signal()
```

```
1 operation PoczątekPisania()
2     if pisarze != 0
3         or czytelnicy != 0
4         MożnaPisać.wait()
5     pisarze = pisarze + 1
```

# Monitory w Javie

- W Javie nie ma odrębnej konstrukcji do implementacji monitorów – implementacja za pomocą klas + blokady + warunki

# Warunki w Javie

- Warunki (ang. conditions) zawsze związane są z blokadą

```
1 public interface Lock {
2     void lock();
3     void lockInterruptibly();
4     boolean tryLock();
5     boolean tryLock(long time, TimeUnit unit);
6     void unlock();
7     Condition newCondition();
8 }
```

# Warunki w Javie – interfejs Condition

```
1  public interface Condition {
2      void await(); // odpowiednik wait()
3      boolean await(long time, TimeUnit unit);
4      long awaitNanos(long nanosTimeout);
5      void awaitUninterruptibly();
6      boolean awaitUntil(Date deadline);
7
8      void signal();
9      void signalAll();
10 }
```

- Metoda `await()` wstrzymuje bieżący wątek i dodaje go do kolejki oczekujących
- Metoda `signalAll()` budzi wszystkie wątki wstrzymane na warunku
- Metoda `signal()` budzi tylko jeden – najdłużej oczekujący (kolejność FIFO)

# Czytelnicy i pisarze w Javie

```
1  class Czytelnia {
2      private final Lock blokada = new ReentrantLock();
3      private final Condition wolne = blokada.newCondition();
4      private int pisarze = 0;
5      private int czytelnicy = 0;
6
7      void poczatekCzytania() throws InterruptedException;
8      void koniecCzytania();
9      void poczatekPisania() throws InterruptedException;
10     void koniecPisania();
11 }
```

- Klasa `Czytelnia` jest monitorem, poszczególne metody dbają o zapewnienie bezpiecznego dostępu do czytelni dla wątków czytelników i pisarzy

# Czytelnicy i pisarze w Javie – main

```
1  public static void main(String[] args) throws InterruptedException {
2      final int n = 1000;
3      final int iluPisarzy = 2;
4      final int iluCzytelnikow = 10;
5      final Czytelnia czytelnia = new Czytelnia();
6      Thread pisarze[] = new Thread[iluPisarzy];
7      for (int i = 0; i < iluPisarzy; i++)
8          pisarze[i] = new Thread(new Pisarz(czytelnia, i, n));
9      Thread czytelnicy[] = new Thread[iluCzytelnikow];
10     for (int i = 0; i < iluCzytelnikow; i++)
11         czytelnicy[i] = new Thread(new Czytelnik(czytelnia, i, n));
12     for (Thread watek : pisarze)    { watek.start(); }
13     for (Thread watek : czytelnicy) { watek.start(); }
14     for (Thread watek : pisarze)    { watek.join(); }
15     for (Thread watek : czytelnicy) { watek.join(); }
16 }
```

# Czytelnicy i pisarze w Javie – pisanie

```
1  void poczatekPisania() throws InterruptedException {
2      blokada.lock();
3      try {
4          while (pisarze > 0 || czytelnicy > 0) {
5              wolne.await();
6          }
7          pisarze++;
8      } finally {
9          blokada.unlock();
10     }
11 }
12 void koniecPisania() {
13     blokada.lock();
14     try {
15         pisarze--;
16         wolne.signalAll();
17     } finally {
18         blokada.unlock();
19     }
20 }
```

# Czytelnicy i pisarze w Javie – czytanie

```
1  void poczatekCzytania() throws InterruptedException {
2      blokada.lock();
3      try {
4          while (pisarze != 0)
5              wolne.await();
6          czytelnicy++;
7      } finally {
8          blokada.unlock();
9      }
10 }
11 void koniecCzytania() {
12     blokada.lock();
13     try {
14         czytelnicy--;
15         if (czytelnicy == 0)
16             wolne.signalAll();
17     } finally {
18         blokada.unlock();
19     }
20 }
```

# Czytelnicy i Pisarze

## Wątek Pisarza

```
1  public void run() {
2  try {
3    for (int i = 0; i < n; i++) {
4      czytelnia.poczatekPisania();
5      System.out.println("Pisarz " +
        id + " pisze");
6      czytelnia.koniecPisania();
7    }
8  } catch (InterruptedException e)
9  { ; }
10 }
```

## Wątek Czytelnika

```
1  public void run() {
2  try {
3    for (int i = 0; i < n; i++) {
4      czytelnia.poczatekCzytania()
5      System.out.println("Czyt. " + id
         + " czyta");
6      czytelnia.koniecCzytania()
7    }
8  } catch (InterruptedException e)
9  { ; }
10 }
```

- Jak widać, wątki czytelników i pisarzy nie zajmują się bezpośrednio zakładniem blokad itp.
- Chociaż muszą wywołać parę metod, np. `poczatekPisania()` i `koniecPisania()`
- Lepszym rozwiązaniem byłoby scalić je, tzn. `czytaj(...)` i `pisz(...)`

# Monitory w Javie c.d.

- OK, ale w Javie z *każdym* obiektem związana jest *niejawna blokada*
- Z każdym obiektem związany jest również *niejawny warunek*
- Stąd proste monitory możemy implementować na dwa sposoby

Mechanizmy „niejawne”:

- `synchronized` lub `synchronized(this)`
- `this.wait()`
- `this.notify()`
- `this.notifyAll()`

Mechanizmy „jawne”:

- `blokada.lock()` oraz `blokada.unlock()`
- `warunek.await()`
- `warunek.signal()`
- `warunek.signalAll()`

# Czytelnia v2.0

## Poprzednia wersja

```
1  void poczatekPisania() throws
       InterruptedException {
2      blokada.lock();
3      try {
4          while (pisarze > 0 ||
       czytelnicy > 0) {
5              wolne.await();
6          }
7          pisarze++;
8      } finally {
9          blokada.unlock();
10     }
11 }
```

## Wersja druga

```
1  synchronized void poczatekPisania()
       throws InterruptedException {
2      while (pisarze > 0 || czytelnicy
         > 0) {
3          this.wait();
4      }
5      pisarze++;
6  }
```

# Czytelnia v2.0

## Poprzednia wersja

```
1 void koniecPisania() {
2     blokada.lock();
3     try {
4         pisarze--;
5         wolne.signalAll();
6     } finally {
7         blokada.unlock();
8     }
9 }
```

## Wersja druga

```
1 synchronized
2 void koniecPisania() {
3     pisarze--;
4     this.notifyAll();
5 }
```

# notify() czy notifyAll() ?

- Użycie `notify()` / `signal()` może prowadzić do trudnych do wykrycia błędów, dlatego jeżeli nie mamy pewności, że `notify()` wystarczy, to zalecane jest korzystanie z `notifyAll()` / `signalAll()`
- Wada – budzenie wszystkich wątków zamiast jednego jest mniej efektywne

# notify() czy notifyAll() – przykład

Rozpatrzmy fragment implementacji kolejki o ograniczonym rozmiarze

```
1  public synchronized void put(Object o) {
2      while (buf.size()==this.max_size)
3          wait();
4      buf.add(o);
5      notify(); // wybudź oczekujący wątek
6  }
7  public synchronized Object get() {
8      while (buf.size()==0)
9          wait();
10     Object o = buf.remove(0);
11     notify(); // wybudź oczekujący wątek
12     return o;
13 }
```

# notify() czy notifyAll() – przykład

- Załóżmy, że użytkownik ustalił `max\_size = 1`
- Rozpatrzmy scenariusz z dwoma producentami P1, P2, P3 oraz konsumentami K1 i K2

```
1  public synchronized
2  void put(Object o) {
3    while (buf.size()==max_size)
4        wait();
5    buf.add(o);
6    notify(); // wybudź
7  }           // oczekujący wątek
8  public synchronized
9  Object get() {
10     while (buf.size()==0)
11         wait();
12     Object o = buf.remove(0);
13     notify(); // wybudź
14     return o; // oczekujący w.
15 }
```

- P1 wykonuje `put()` - wstawia znak do bufora i kończy się

# notify() czy notifyAll() – przykład

- Załóżmy, że użytkownik ustalił `max\_size = 1`
- Rozpatrzmy scenariusz z dwoma producentami P1, P2, P3 oraz konsumentami K1 i K2

```
1  public synchronized
2  void put(Object o) {
3    while (buf.size()==max_size)
4        wait();
5    buf.add(o);
6    notify(); // wybudź
7  }           // oczekujący wątek
8  public synchronized
9  Object get() {
10     while (buf.size()==0)
11         wait();
12     Object o = buf.remove(0);
13     notify(); // wybudź
14     return o; // oczekujący w.
15 }
```

- P1 wykonuje `put()` - wstawia znak do bufora i kończy się
- P2 i P3 wykonują `put()` i czekają w wierszu 4.

# notify() czy notifyAll() – przykład

- Załóżmy, że użytkownik ustalił `max\_size = 1`
- Rozpatrzmy scenariusz z dwoma producentami P1, P2, P3 oraz konsumentami K1 i K2

```
1  public synchronized
2  void put(Object o) {
3    while (buf.size()==max_size)
4        wait();
5    buf.add(o);
6    notify(); // wybudź
7  }           // oczekujący wątek
8  public synchronized
9  Object get() {
10     while (buf.size()==0)
11         wait();
12     Object o = buf.remove(0);
13     notify(); // wybudź
14     return o; // oczekujący w.
15 }
```

- P1 wykonuje `put()` - wstawia znak do bufora i kończy się
- P2 i P3 wykonują `put()` i czekają w wierszu 4.
- K1 wykonuje `get()` i budzi P2

# notify() czy notifyAll() – przykład

- Załóżmy, że użytkownik ustalił `max\_size = 1`
- Rozpatrzmy scenariusz z dwoma producentami P1, P2, P3 oraz konsumentami K1 i K2

```
1  public synchronized
2  void put(Object o) {
3    while (buf.size()==max_size)
4        wait();
5    buf.add(o);
6    notify(); // wybudź
7  }           // oczekujący wątek
8  public synchronized
9  Object get() {
10     while (buf.size()==0)
11         wait();
12     Object o = buf.remove(0);
13     notify(); // wybudź
14     return o; // oczekujący w.
15 }
```

- P1 wykonuje `put()` - wstawia znak do bufora i kończy się
- P2 i P3 wykonują `put()` i czekają w wierszu 4.
- K1 wykonuje `get()` i budzi P2
- K2 wchodzi do `get()` (ma taki sam priorytet jak P2) i zatrzymuje się w wierszu 11. – czekają P3 oraz K2

# notify() czy notifyAll() – przykład

- Załóżmy, że użytkownik ustalił `max\_size = 1`
- Rozpatrzmy scenariusz z dwoma producentami P1, P2, P3 oraz konsumentami K1 i K2

```
1  public synchronized
2  void put(Object o) {
3    while (buf.size()==max_size)
4        wait();
5    buf.add(o);
6    notify(); // wybudź
7  }           // oczekujący wątek
8  public synchronized
9  Object get() {
10     while (buf.size()==0)
11         wait();
12     Object o = buf.remove(0);
13     notify(); // wybudź
14     return o; // oczekujący w.
15 }
```

- P1 wykonuje `put()` - wstawia znak do bufora i kończy się
- P2 i P3 wykonują `put()` i czekają w wierszu 4.
- K1 wykonuje `get()` i budzi P2
- K2 wchodzi do `get()` (ma taki sam priorytet jak P2) i zatrzymuje się w wierszu 11. – czekają P3 oraz K2
- P2 wstawia el. do bufora i wykonuje `notify()` – budzi P3

# notify() czy notifyAll() – przykład

- Załóżmy, że użytkownik ustalił `max\_size = 1`
- Rozpatrzmy scenariusz z dwoma producentami P1, P2, P3 oraz konsumentami K1 i K2

```
1  public synchronized
2  void put(Object o) {
3    while (buf.size()==max_size)
4        wait();
5    buf.add(o);
6    notify(); // wybudź
7  }           // oczekujący wątek
8  public synchronized
9  Object get() {
10     while (buf.size()==0)
11         wait();
12     Object o = buf.remove(0);
13     notify(); // wybudź
14     return o; // oczekujący w.
15 }
```

- P1 wykonuje `put()` - wstawia znak do bufora i kończy się
- P2 i P3 wykonują `put()` i czekają w wierszu 4.
- K1 wykonuje `get()` i budzi P2
- K2 wchodzi do `get()` (ma taki sam priorytet jak P2) i zatrzymuje się w wierszu 11. – czekają P3 oraz K2
- P2 wstawia el. do bufora i wykonuje `notify()` – budzi P3
- P3 próbuje wstawić el., ale bufor jest pełny, więc ponownie czeka

# notify() czy notifyAll() – przykład

- Załóżmy, że użytkownik ustalił `max\_size = 1`
- Rozpatrzmy scenariusz z dwoma producentami P1, P2, P3 oraz konsumentami K1 i K2

```
1  public synchronized
2  void put(Object o) {
3    while (buf.size()==max_size)
4        wait();
5    buf.add(o);
6    notify(); // wybudź
7  }           // oczekujący wątek
8  public synchronized
9  Object get() {
10     while (buf.size()==0)
11         wait();
12     Object o = buf.remove(0);
13     notify(); // wybudź
14     return o; // oczekujący w.
15 }
```

- P1 wykonuje `put()` - wstawia znak do bufora i kończy się
- P2 i P3 wykonują `put()` i czekają w wierszu 4.
- K1 wykonuje `get()` i budzi P2
- K2 wchodzi do `get()` (ma taki sam priorytet jak P2) i zatrzymuje się w wierszu 11. – czekają P3 oraz K2
- P2 wstawia el. do bufora i wykonuje `notify()` – budzi P3
- P3 próbuje wstawić el., ale bufor jest pełny, więc ponownie czeka
- K2 wciąż czeka, mimo że bufor jest pełny

# Warunki Cofmanna

Warunki, które muszą zostać jednocześnie spełnione, by doszło do zakleszczenia:

- **Wzajemne wykluczanie** (ang. mutual exclusion) – każdy z zasobów jest wolny albo zajęty przez jakiś proces (tylko jeden)
- **Przetrzymywanie i oczekiwanie** (ang. hold and wait) – proces posiadający zasób żąda dostępu do co najmniej jednego z pozostałych zasobów
- **Brak wywłaszczeń** (ang. no preemption) – zasoby nie podlegają wywłaszczeniu, czyli nie mogą być procesowi „odebrane", a jedynie przez niego zwolnione
- **Czekanie cykliczne** (ang. circular wait) – musi istnieć ciąg procesów $P = (P_1, P_2, \ldots, P_n)$, w którym proces $P_i$ czeka na zasób zajęty przez proces $P_j$, gdzie $j = i + 1$ dla $i < n$ lub $j = 1$ dla $i = n$

# Metody postępowania z zakleszczeniami

1. Zapobieganie zakleszczeniom – całk. eliminacja możliwości zakleszczenia
2. Unikanie zakleszczeń – zakleszczenie może wystąpić, ale do niego nie dopuszczamy
3. Wykrywanie i wychodzenie z zakleszczeń – działamy dopiero, gdy wykryjemy wystąpienie zakleszczenia

# Zapobieganie zakleszczeniom

W celu zapobiegania zakleszczeniom wystarczy zapewnić, że przynajmniej jeden z warunków koniecznych zakleszczenia nie będzie spełniony

- Wzajemne wykluczanie
  - Żadne zasoby nie mogą być współdzielone w trybie „do modyfikacji”
- Przetrzymywanie i oczekiwanie
  - każdy z procesów musi zamawiać wszystkie niezbędne zasoby na raz
  - jeżeli proces nie otrzymał zasobu, to musi zwolnić wszystkie posiadane do tej pory i spróbować ponownie później
- Brak wywłaszczeń
  - Niejawne odebranie procesowi czekającemu na przydział wszystkich już posiadanych zasobów i oddanie ich innemu, a usunięte dopisane zostają do listy zamówień procesu

# Przykład

```
1  // Zapobieganie zakleszczeniom przez łamanie warunku przetrzymywania i
       oczekiwania
2  import java.util.concurrent.lock.ReentrantLock;
3  class NoDeadlock {
4      private int n;
5      protected ReentrantLock lock=new ReentrantLock();
6
7      private boolean getLocks(NoDeadlock other) {
8          Boolean myLock, yourLock; // initially false
9          try {
10             myLock = lock.tryLock()
11             yourLock = other.lock.tryLock()
12         } finally {
13             if (! (myLock && yourLock)) {
14                 if (myLock) { lock.unlock(); }
15                 if (yourLock) { other.lock.unlock(); }
16             }
17         }
18         return myLock && yourLock;
19     }
```

# Przykład c.d.

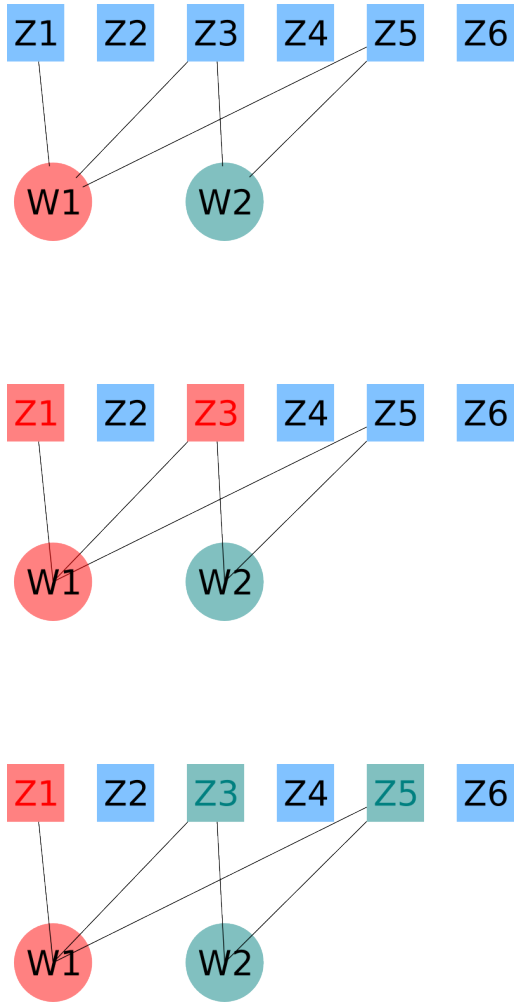
```
1 public boolean areEqual(NoDeadlock other) throws InterruptedException {
2     while(!getLocks(other)) {
3         Thread.sleep(1);
4     }
5     Boolean ret = (n==other.getN());
6     lock.unlock();
7     other.lock.unlock();
8     return ret;
9 }
```

- Jeżeli proces nie otrzymał wszystkich pożądanych zasobów – zwalnia, te które ma i próbuje ponownie później

# Zapobieganie zakleszczeniom

- Czekanie cykliczne
  - każdy zasób otrzymuje unikalny numer porządkowy (etykietę)
  - żądania do zasobów są zawsze wykonywane w kolejności rosnących numerów porządkowych
  - przykładowo, proces będący w posiadaniu zasobu nr 5 nie może żądać dostępu do zasobów o mniejszych numerach bez uprzedniego zwolnienia zasobu nr 5
  - numeracja nie musi być jawna, w praktyce chodzi o stosowanie zamków (ang. locks) lub semaforów zawsze w określonej kolejności

# Łamanie warunku oczekiwania cyklicznego



# Unikanie zakleszczeń

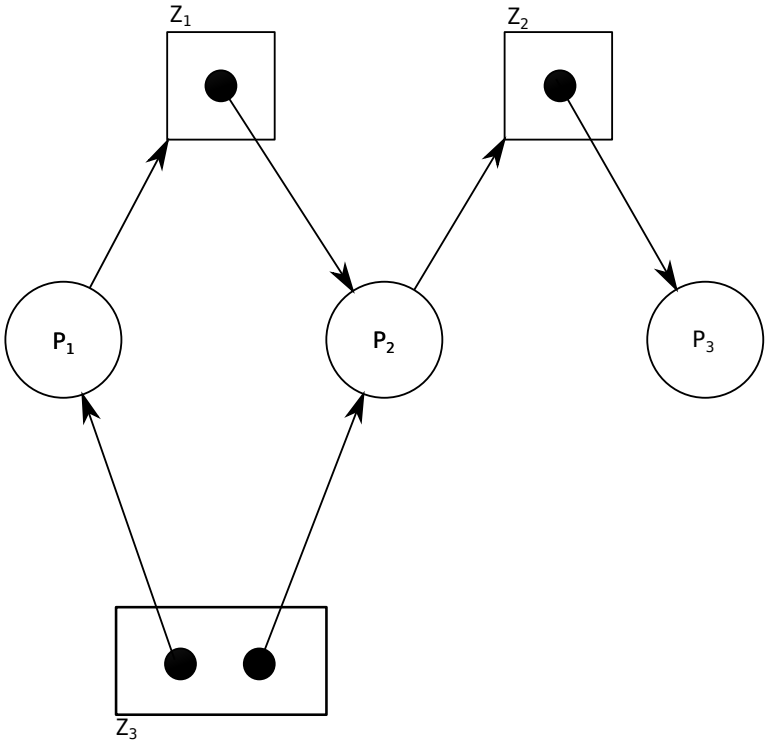
Metody zapobiegania zakleszczeniom mogą zmniejszać stopień wykorzystania zasobów i obniżać efektywność / przepustowość systemu.

Przydział zasobów w systemie można opisać za pomocą **grafu przydziału zasobów**. Graf ten jest grafem skierowanym, w którym wierzchołki stanowią:

- wszystkie zasoby – $Z$
- wszystkie procesy – $P$

Krawędzie łączą procesy z zasobami – jeżeli proces $P_i$ oczekuje na zasób $Z_j$, to istnieje krawędź $(P_i, Z_j)$ nazywana *krawędzią zamówienia*. Krawędź prowadząca od zasobu do procesu jest *krawędzią przydziału*.

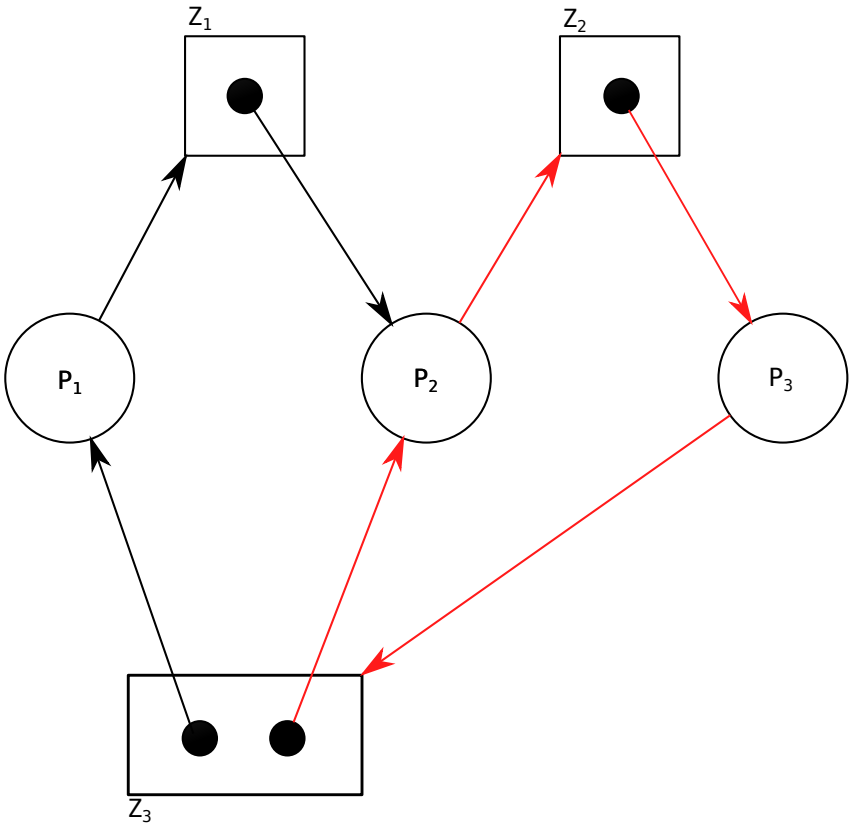
# Graf przydziału zasobów




Przykład grafu przydziału zasobów.

- Punkty wewnątrz wierzchołków zasobów symbolizują *egzemplarze* zasobu.
- Krawędzie łączą konkretny egzemplarz zasobu z procesem.
- W grafie nie ma *cyklu*, dlatego nie może dojść do zakleszczenia.

# Graf przydziału zasobów




Przykład grafu przydziału zasobów, w którym dochodzi do zakleszczenia. Występuje m.in. cykl $(P_2, Z_2, P_3, Z_3)$.

# Unikanie zakleszczeń c.d.

- Unikanie zakleszczeń można zrealizować analizując graf przydziału zasobów.
- System może znajdować się w trzech stanach:
  - bezpiecznym
  - zagrożonym
  - zakleszczenia.
- Unikanie zakleszczeń wymaga dokonywania takich przydziałów zasobów, aby system znajdował się *stale* w stanie bezpiecznym.
- Występuje konieczność analizowania żądań przydziału zasobów i wykrywania cykli w grafie przydziału zasobów.
- Jednym z przykładowych algorytmów unikania zakleszczeń jest *algorytm bankiera*.

# Algorytm bankiera

- Algorytm bankiera (ang. banker's algorithm) – zaproponowany przez E. Dijkstrę
- Algorytm zapobiega zakleszczeniom przez nadzorowanie przydziału zasobów poszczególnym procesom
- Każdy proces określa maksymalną liczbę zasobów, których będzie potrzebował
- Algorytm wyróżnia dwa stany systemu ze względu na przydział zasobów procesom

# Algorytm bankiera – stany systemu

- System może znaleźć się w jednym z dwóch stanów:
  - bezpiecznym (ang. safe) – nie może dojść do zakleszczenia
  - niebezpiecznym (ang. unsafe) – zakleszczenie jest możliwe, ale nie musi wystąpić

# Algorytm bankiera – stany systemu

- Do stwierdzenia, do której kategorii należy stan algorytm bankiera potrzebuje danych:
  - maksymalnej liczby zasobów wymaganej przez proces
  - całkowitej liczby zasobów dostępnych w systemie
  - łącznej liczby zasobów aktualnie przydzielonych procesom

# Algorytm bankiera – stany systemu

Tabela 1: Przykładowy stan bezpieczny

| Dostępne zasoby = 2 | | |
|---|---|---|
| Proces | Przydział | Maksimum |
| A | 1 | 6 |
| B | 1 | 5 |
| C | 2 | 4 |
| D | 4 | 7 |

- Proces C może otrzymać 2 brakujące zasoby i się zakończyć

Tabela 2: Przykładowy stan niebezpieczny

| Dostępne zasoby = 1 | | |
|---|---|---|
| Proces | Przydział | Maksimum |
| A | 1 | 6 |
| B | 1 | 5 |
| C | 2 | 4 |
| D | 4 | 7 |

- Żaden z procesów nie może otrzymać dostatecznej liczby zasobów

# Algorytm bankiera – idea

Ogólna idea działania alg. jest następująca:

1. znajdź proces z najmniejszą brakującą liczbą zasobów
2. przydziel zasoby temu procesowi i poczekaj na zakończenie
3. zasoby zakończonego procesu oznacz jako *dostępne*
4. wróć do kroku 1

# Wykrywanie i wychodzenie z zakleszczeń

Trzecia metoda radzenia sobie z zakleszczeniami polega na ich wykrywaniu i wychodzeniu z nich, co wymaga:

- algorytmu wykrywania zakleszczeń:
  - analiza grafu przydziału zasobów i wykrywanie cykli oznaczających zakleszczenie
- algorytmu wychodzenia z zakleszczeń:
  - kończenie zakleszczonych procesów
  - wywłaszczanie zasobów

Wadami tego rozwiązania jest dodatkowy narzut czasowy związany ze wspomnianym algorytmami. Muszą one być wykonywane:

- każdym żądaniu przydziału zasobu – duży narzut czasowy
- okresowo – brak możliwości wykrycia sprawców zakleszczenia
- po wykryciu spadku przepustowości w systemie