

Stefan Sokołowski

PROGRAMOWANIE OBIEKTOWE II

wykład 12

**Inst. Informatyki Stosowanej, ANS
Elbląg, 2024/2025**

Współbieżność

Komputerowa realizacja procesów działających jednocześnie

Współbieżność

**Komputerowa realizacja procesów
działających jednocześnie**

Jeśli są niezależne...

Współbieżność

Komputerowa realizacja procesów działających jednocześnie

Jeśli są niezależne...

...nie korzystają z tych samych zasobów

Współbieżność

Komputerowa realizacja procesów działających jednocześnie

Jeśli są niezależne...

- ... nie korzystają z tych samych zasobów,
- ... nie „odzywają się” do siebie

Współbieżność

Komputerowa realizacja procesów działających jednocześnie

Jeśli są niezależne...

- ... nie korzystają z tych samych zasobów,
- ... nie „odzywają się” do siebie,
- ... żaden nie korzysta z wyników działania innego

Współbieżność

Komputerowa realizacja procesów działających jednocześnie

Jeśli są niezależne...

- ... nie korzystają z tych samych zasobów,
- ... nie „odzywają się” do siebie,
- ... żaden nie korzysta z wyników działania innego,
- ... itp.

Współbieżność

Komputerowa realizacja procesów działających jednocześnie

Jeśli są niezależne...

- ... nie korzystają z tych samych zasobów,
- ... nie „odzywają się” do siebie,
- ... żaden nie korzysta z wyników działania innego,
- ... itp.

to nietrudne:

osobne procesory

Współbieżność

Komputerowa realizacja procesów działających jednocześnie

Jeśli są niezależne...

- ... nie korzystają z tych samych zasobów,
- ... nie „odzywają się” do siebie,
- ... żaden nie korzysta z wyników działania innego,
- ... itp.

to nietrudne:

osobne procesory, ew.
podział czasu.

Współbieżność

Komputerowa realizacja procesów działających jednocześnie

Jeśli są niezależne...

- ... nie korzystają z tych samych zasobów,
- ... nie „odzywają się” do siebie,
- ... żaden nie korzysta z wyników działania innego,
- ... itp.

to nietrudne:

- osobne procesory, ew.
- podział czasu.

Jeśli są zależne... to problemy z synchronizacją, które trzeba rozwiązać.

Współbieżność

Komputerowa realizacja procesów działających jednocześnie

Jeśli są niezależne...

- ... nie korzystają z tych samych zasobów,
- ... nie „odzywają się” do siebie,
- ... żaden nie korzysta z wyników działania innego,
- ... itp.

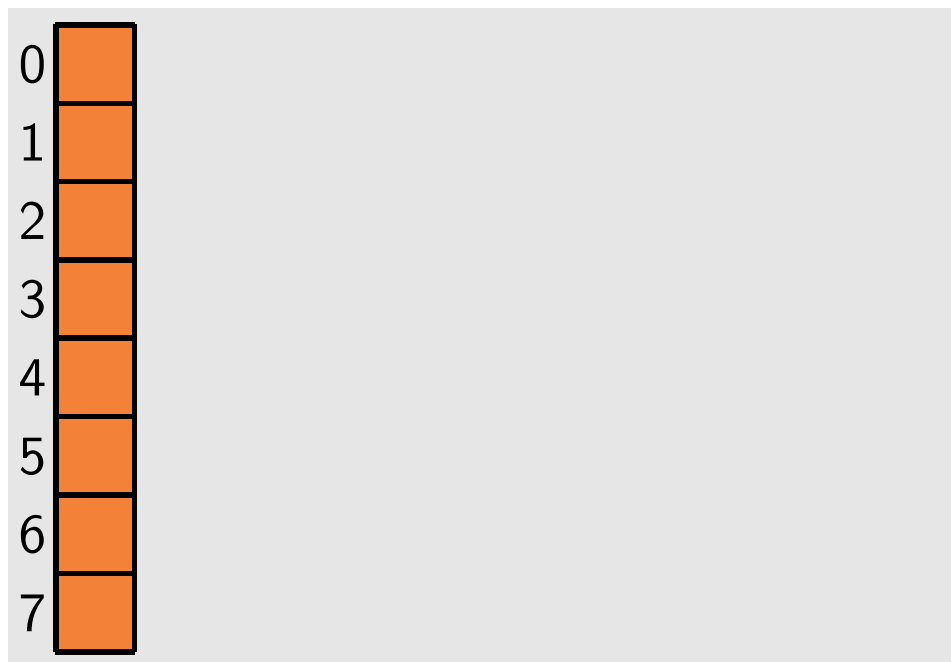
to nietrudne:

- osobne procesory, ew.
- podział czasu.

Jeśli są zależne... to problemy z synchronizacją, które trzeba rozwiązać.

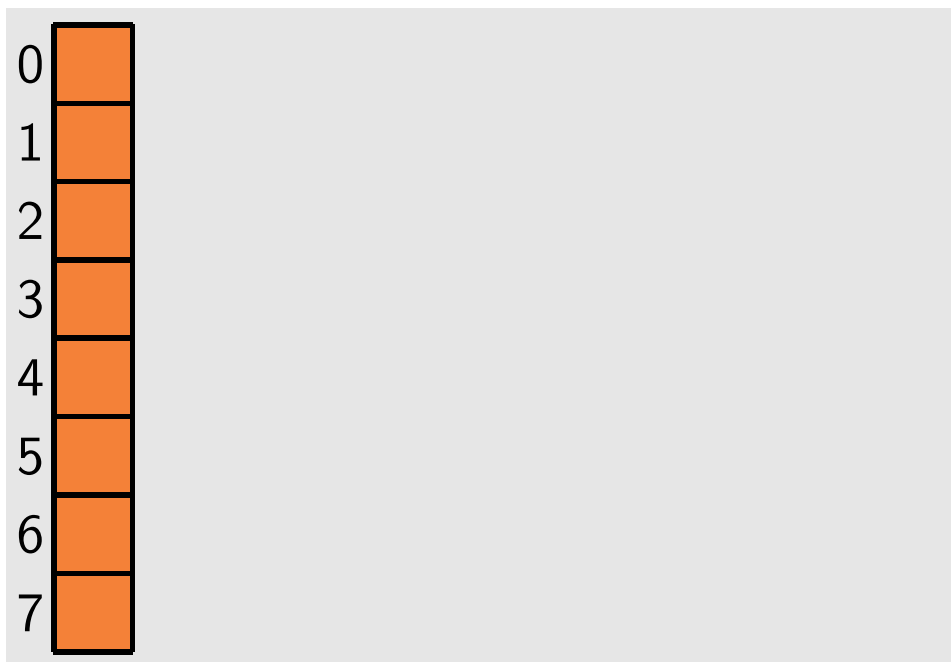
W Jawie — wątki (Thread).

Wielowątkowość — max



Wielowątkowość — max

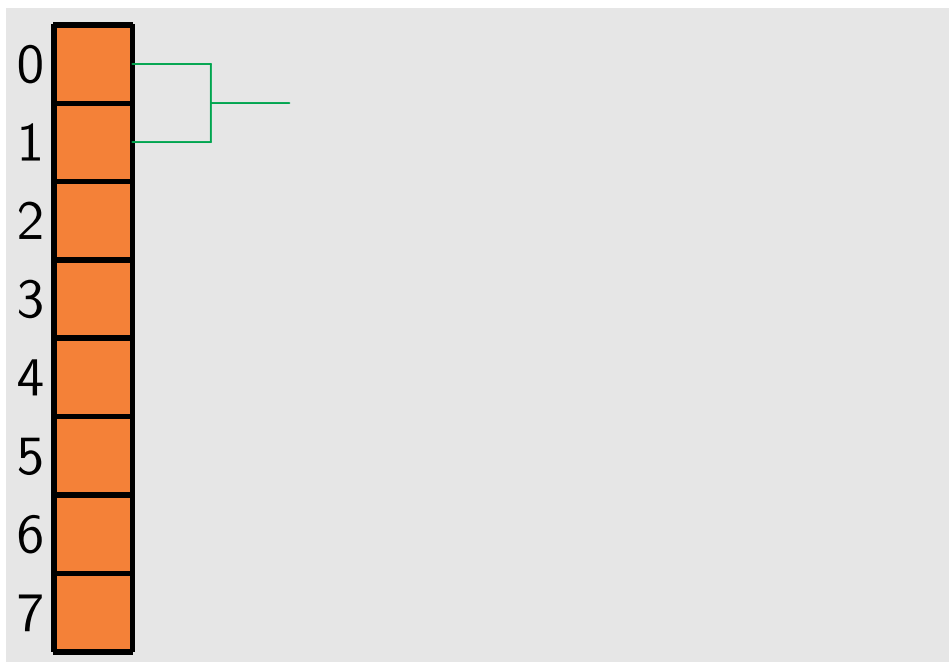
Liniowo:



```
max = a[0];  
for (int i=1; i<n; i++)  
    if (a[i] > max) max = a[i];
```

Wielowątkowość — max

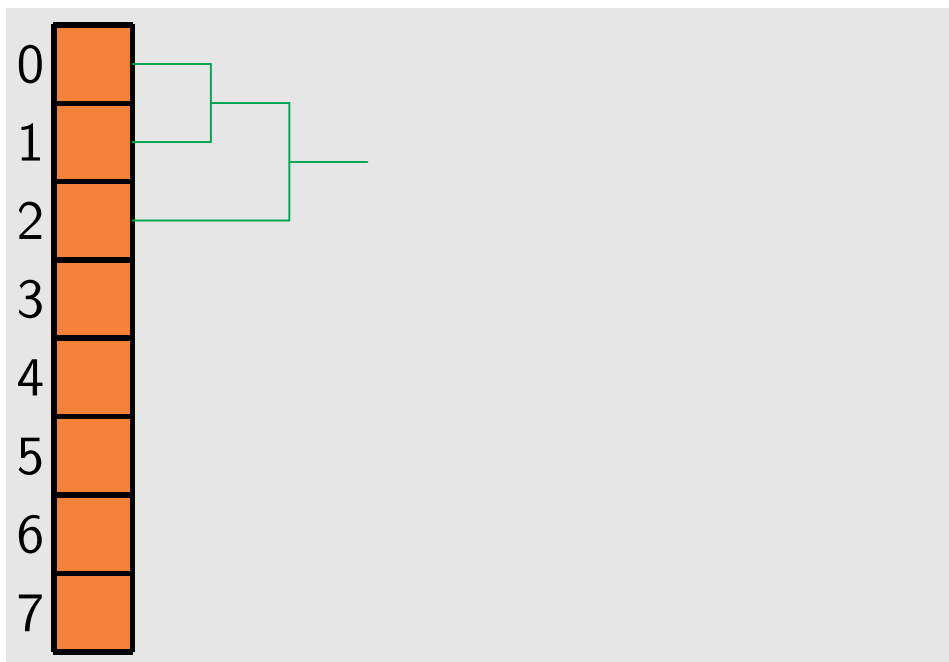
Liniowo:



```
max = a[0];  
for (int i=1; i<n; i++)  
    if (a[i] > max) max = a[i];
```

Wielowątkowość — max

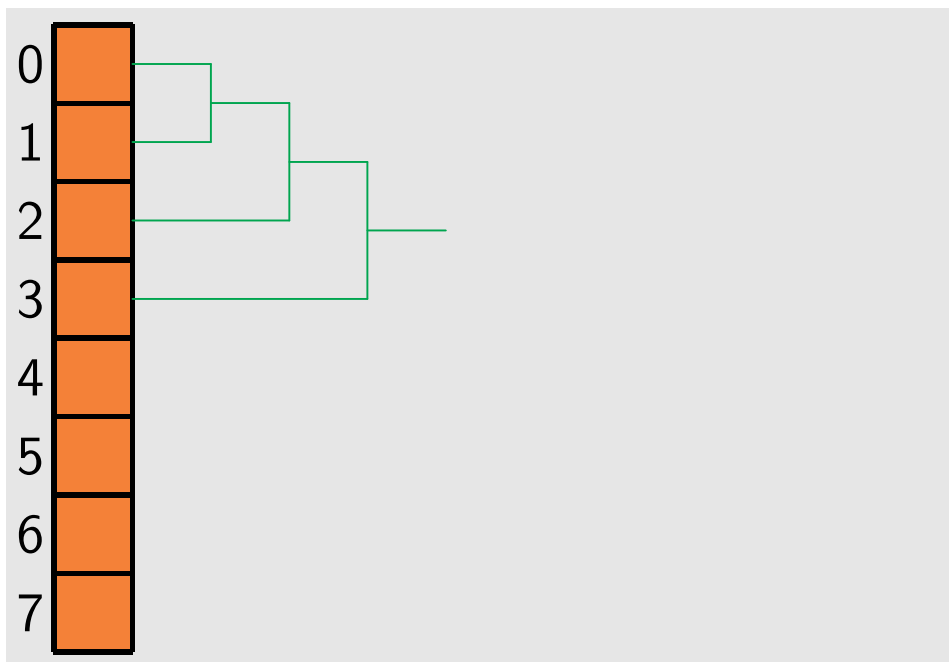
Liniowo:



```
max = a[0];  
for (int i=1; i<n; i++)  
    if (a[i] > max) max = a[i];
```

Wielowątkowość — max

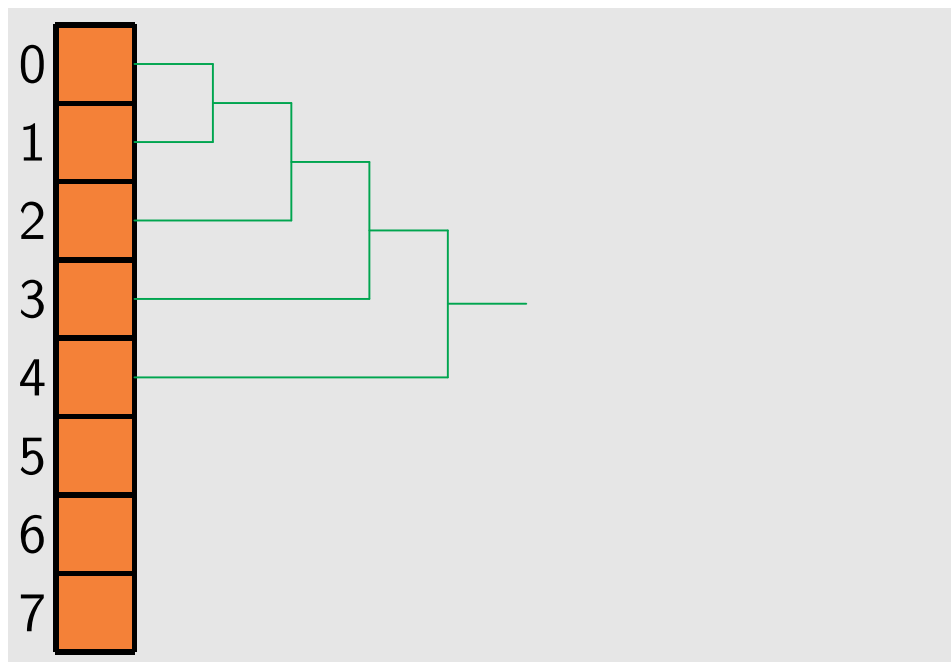
Liniowo:



```
max = a[0];  
for (int i=1; i<n; i++)  
    if (a[i] > max) max = a[i];
```

Wielowątkowość — max

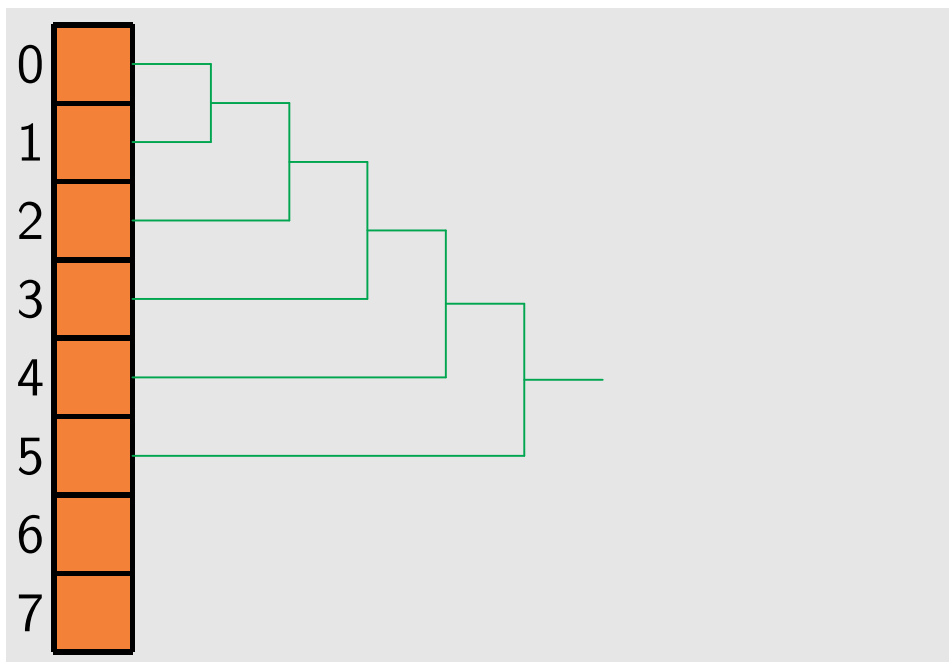
Liniowo:



```
max = a[0];  
for (int i=1; i<n; i++)  
    if (a[i] > max) max = a[i];
```

Wielowątkowość — max

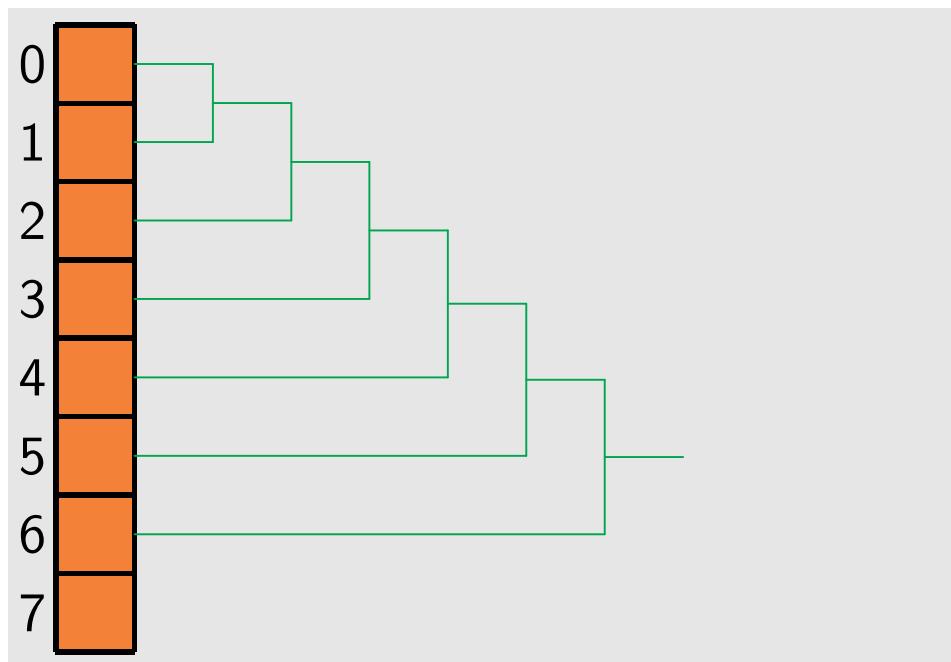
Liniowo:



```
max = a[0];  
for (int i=1; i<n; i++)  
    if (a[i] > max) max = a[i];
```

Wielowątkowość — max

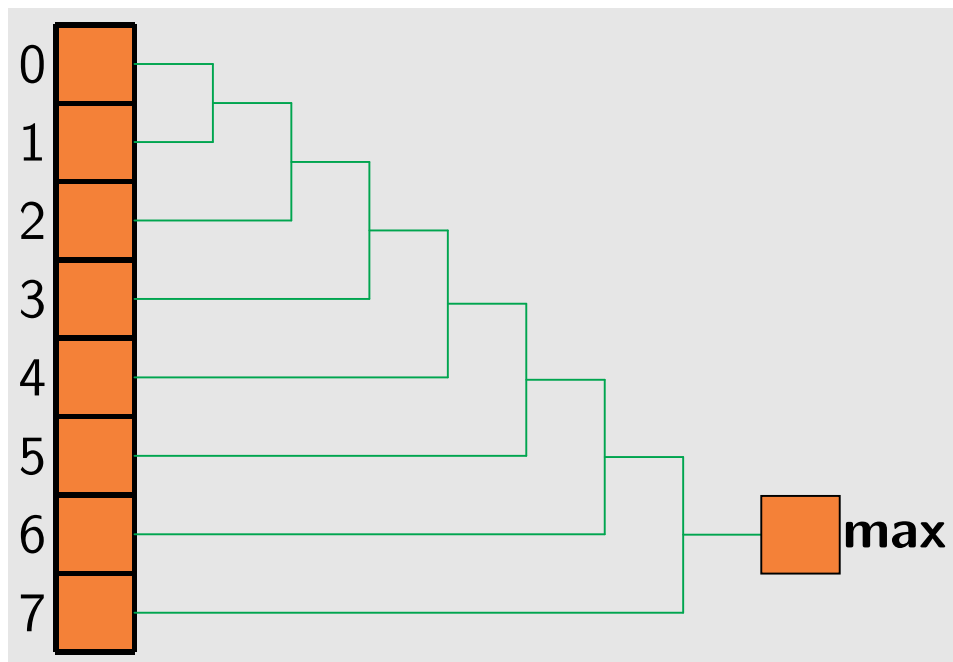
Liniowo:



```
max = a[0];  
for (int i=1; i<n; i++)  
    if (a[i] > max) max = a[i];
```

Wielowątkowość — max

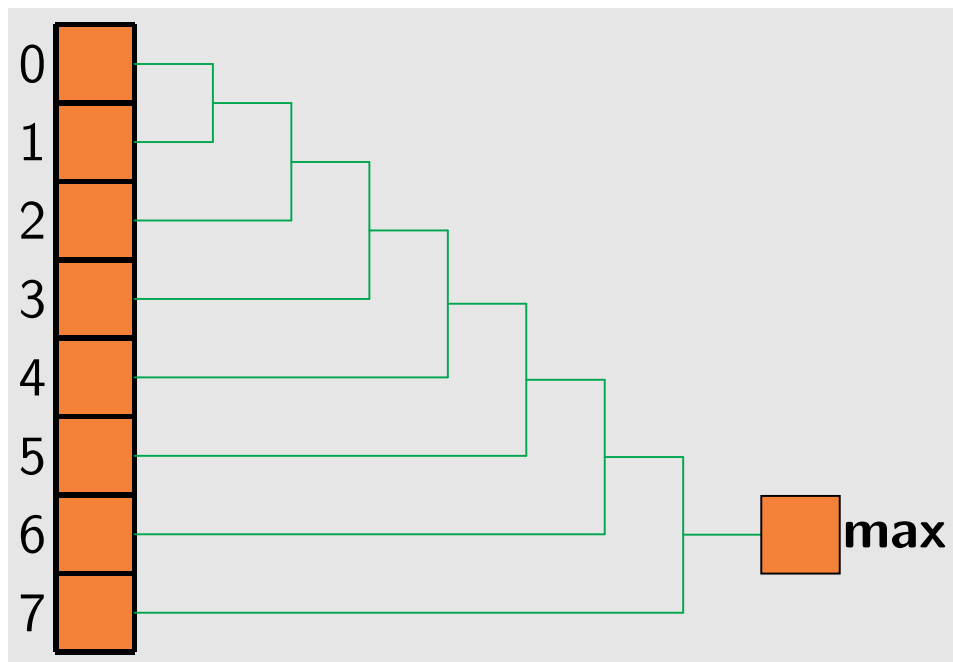
Liniowo:



```
max = a[0];  
for (int i=1; i<n; i++)  
    if (a[i] > max) max = a[i];
```

Wielowątkowość — max

Liniowo:

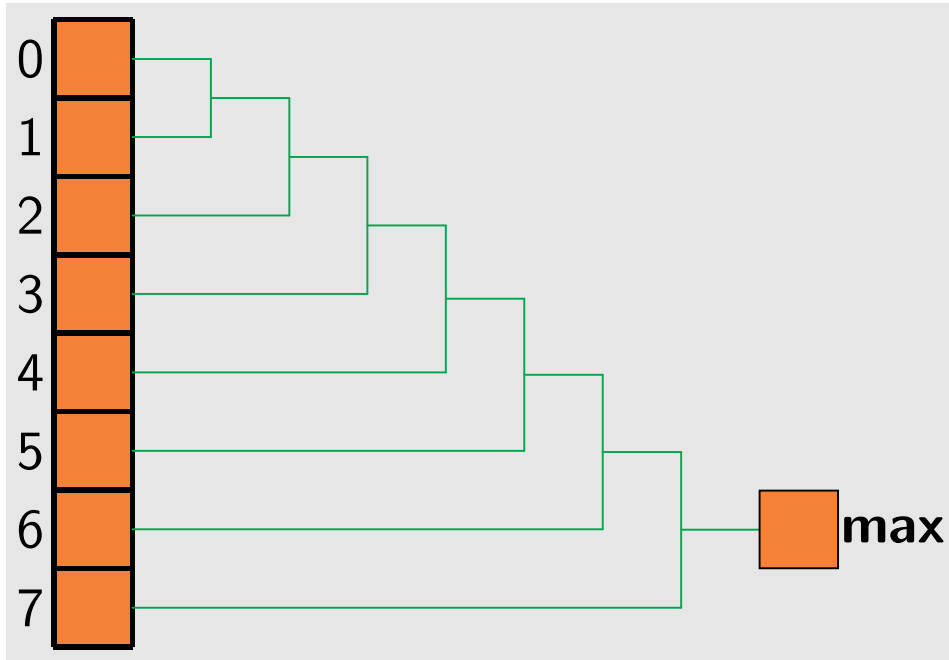


```
max = a[0];  
for (int i=1; i<n; i++)  
    if (a[i] > max) max = a[i];
```

Liczba porównań: $n - 1$

Wielowątkowość — max

Liniowo:



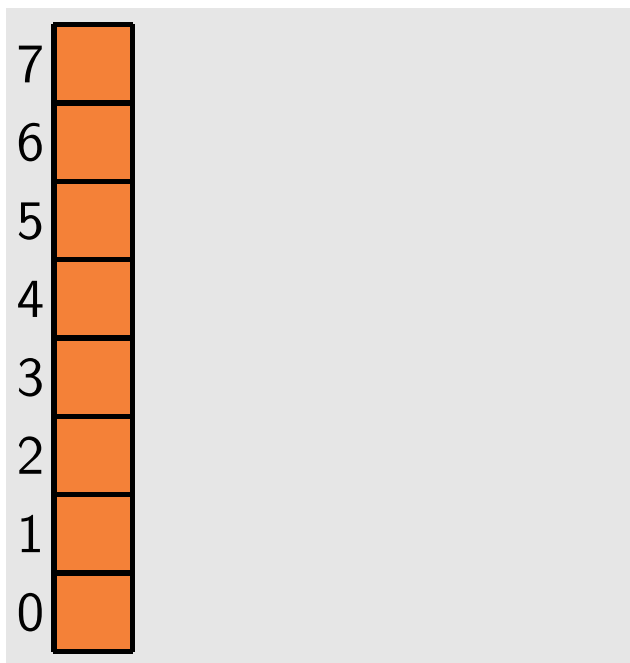
```
max = a[0];  
for (int i=1; i<n; i++)  
    if (a[i] > max) max = a[i];
```

Liczba porównań: $n - 1$

Do każdego porównania potrzebny jest wynik poprzedniego.

Wielowątkowość — max

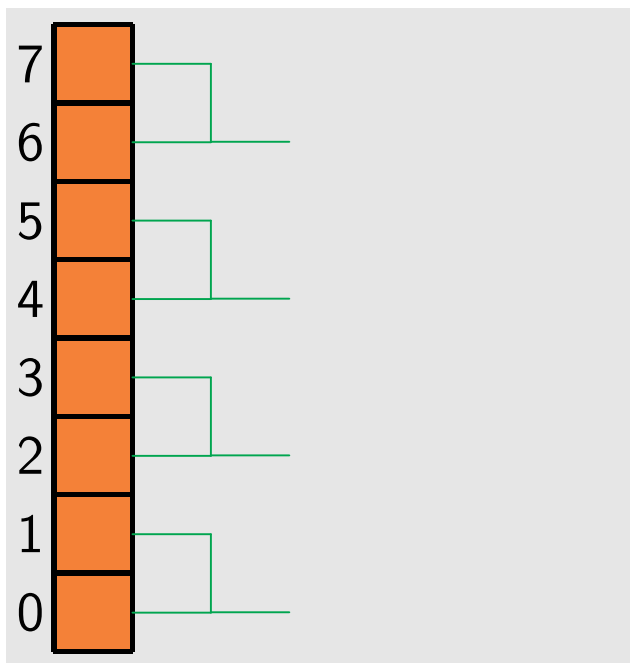
Pucharowo: (n jest potęgą 2)



```
k = 0;
while (2k < n) {
    for (int i=0; i+2k<n; i+=2k+1)
        if (a[i] < a[i+2k])
            zamień a[i] z a[i+2k];
    k++;
}
```

Wielowątkowość — max

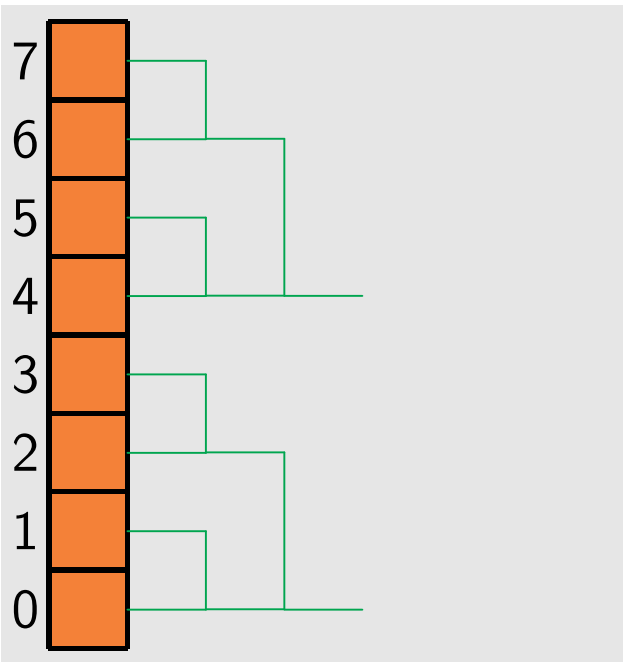
Pucharowo: (n jest potęgą 2)



```
k = 0;
while (2k < n) {
    for (int i=0; i+2k<n; i+=2k+1)
        if (a[i] < a[i+2k])
            zamień a[i] z a[i+2k];
    k++;
}
```

Wielowątkowość — max

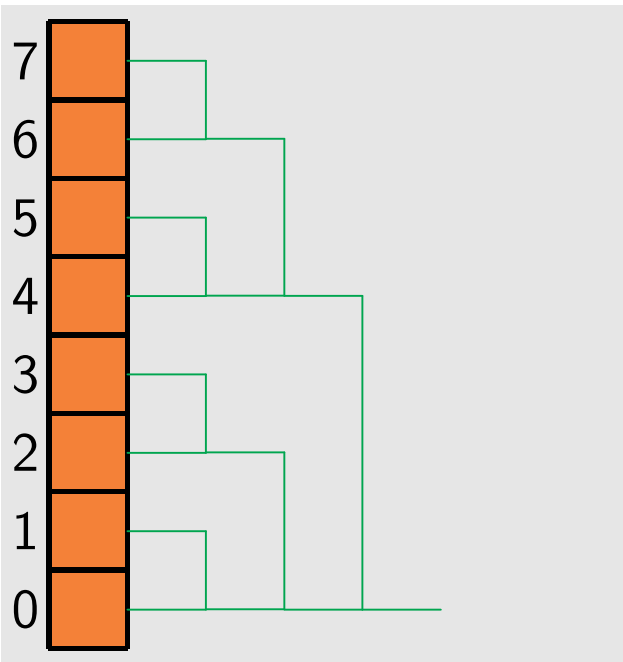
Pucharowo: (n jest potęgą 2)



```
k = 0;
while (2k < n) {
    for (int i=0; i+2k<n; i+=2k+1)
        if (a[i] < a[i+2k])
            zamień a[i] z a[i+2k];
    k++;
}
```

Wielowątkowość — max

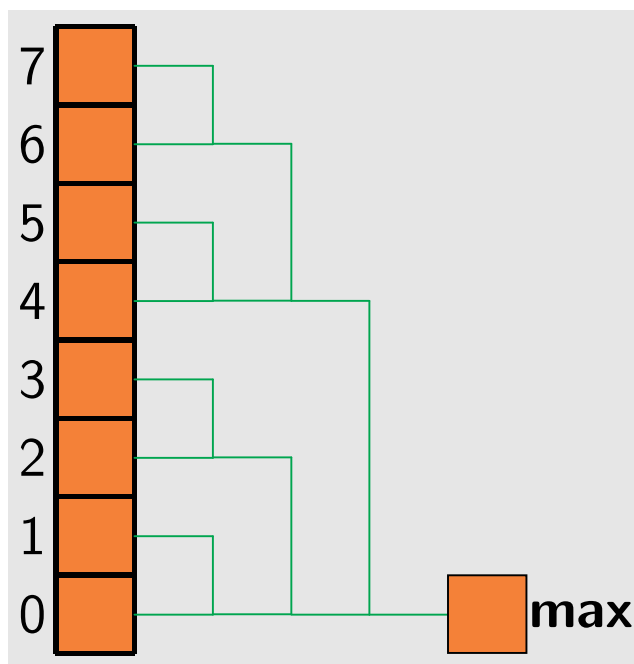
Pucharowo: (n jest potęgą 2)



```
k = 0;
while (2k < n) {
    for (int i=0; i+2k<n; i+=2k+1)
        if (a[i] < a[i+2k])
            zamień a[i] z a[i+2k];
    k++;
}
```

Wielowątkowość — max

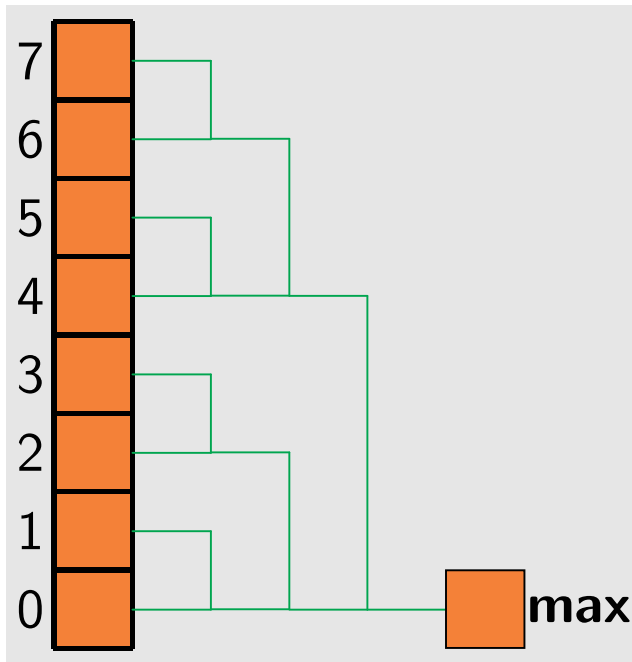
Pucharowo: (n jest potęgą 2)



```
k = 0;
while (2k < n) {
    for (int i=0; i+2k<n; i+=2k+1)
        if (a[i] < a[i+2k])
            zamień a[i] z a[i+2k];
    k++;
}
```

Wielowątkowość — max

Pucharowo: (n jest potęgą 2)



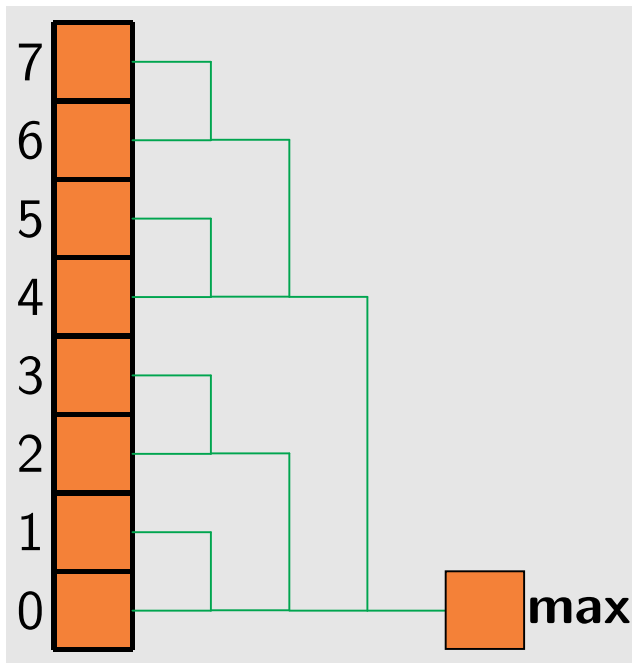
```

k = 0; krok = 1;
while (2k < n) {
    for (int i=0; i+2k<n; i+=2k+1)
        if (a[i] < a[i+2k])
            zamień a[i] z a[i+2k];
    k++; krok *= 2;
}

```

Wielowątkowość — max

Pucharowo: (n jest potęgą 2)



```

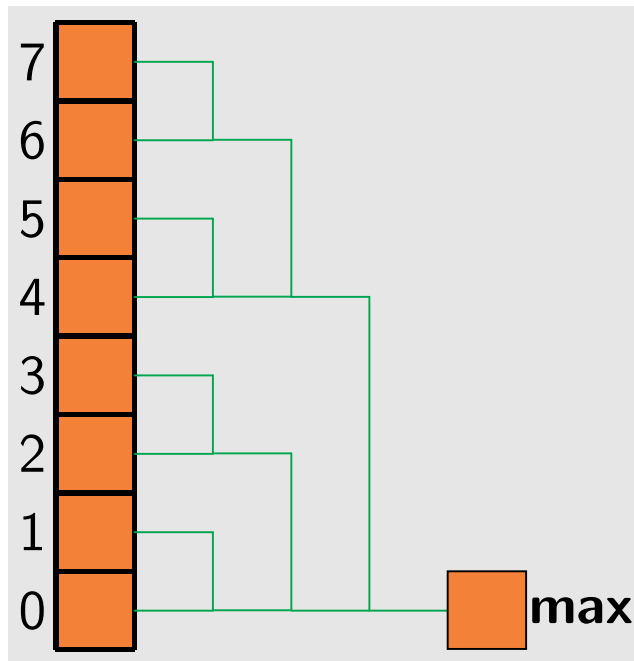
k = 0; krok = 1;
while (2k < n) {
    for (int i=0; i+2k<n; i+=2k+1)
        if (a[i] < a[i+2k])
            zamień a[i] z a[i+2k];
    k++; krok *= 2;
}

```

Liczba zamian: $\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \dots + 1 =$

Wielowątkowość — max

Pucharowo: (n jest potęgą 2)



```

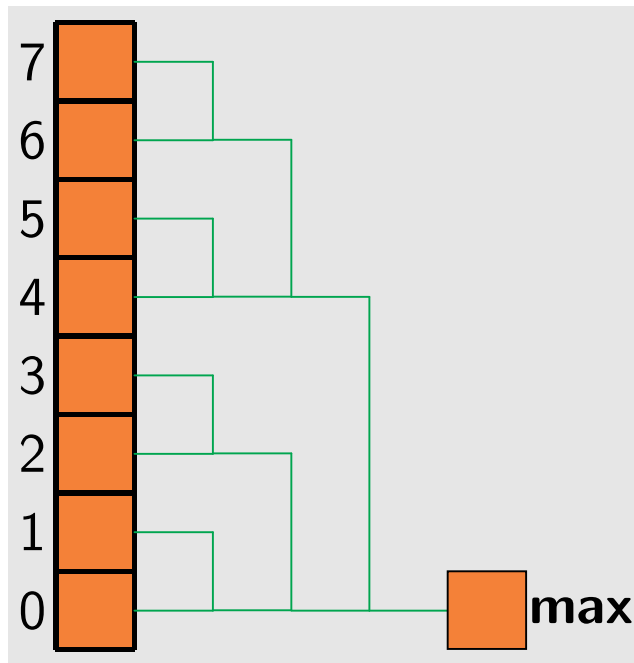
k = 0; krok = 1;
while (2k < n) {
    for (int i=0; i+2k<n; i+=2k+1)
        if (a[i] < a[i+2k])
            zamień a[i] z a[i+2k];
    k++; krok *= 2;
}

```

Liczba zamian: $\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \dots + 1 = n - 1$ (ciąg geometryczny)

Wielowątkowość — max

Pucharowo: (n jest potęgą 2)



```

k = 0; krok = 1;
while (2k < n) {
    for (int i=0; i+2k<n; i+=2k+1)
        if (a[i] < a[i+2k])
            zamień a[i] z a[i+2k];
    k++; krok *= 2;
}

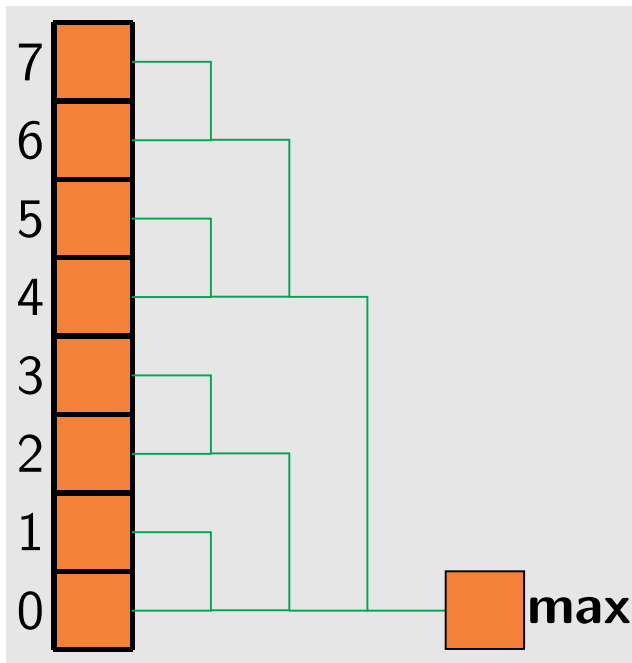
```

Liczba zamian: $\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \dots + 1 = n - 1$ (ciąg geometryczny)

Gdyby cały **for** na raz

Wielowątkowość — max

Pucharowo: (n jest potęgą 2)



```

k = 0; krok = 1;
while (2k < n) {
    for (int i=0; i+2k<n; i+=2k+1)
        if (a[i] < a[i+2k])
            zamień a[i] z a[i+2k];
    k++; krok *= 2;
}

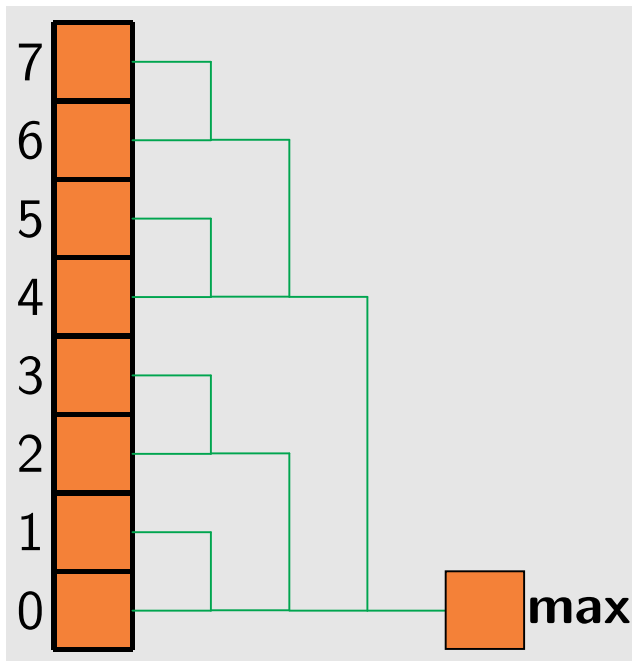
```

Liczba zamian: $\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \dots + 1 = n - 1$ (ciąg geometryczny)

Gdyby cały for na raz: $1 + 1 + 1 + \dots + 1 =$

Wielowątkowość — max

Pucharowo: (n jest potęgą 2)



```

k = 0; krok = 1;
while (2k < n) {
    for (int i=0; i+2k<n; i+=2k+1)
        if (a[i] < a[i+2k])
            zamień a[i] z a[i+2k];
    k++; krok *= 2;
}
  
```

Liczba zamian: $\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \dots + 1 = n - 1$ (ciąg geometryczny)

Gdyby cały for na raz: $1 + 1 + 1 + \dots + 1 = \log_2 n$

Wielowątkowość — max

W czym $\log_2 n$ jest lepszy od $n - 1$?

Wielowątkowość — max

W czym $\log_2 n$ jest lepszy od $n - 1$?

n	$n - 1$	$\log_2 n$
2	1	1
4	3	2
8	7	3

Wielowątkowość — max

W czym $\log_2 n$ jest lepszy od $n - 1$?

n	$n - 1$	$\log_2 n$
2	1	1
4	3	2
8	7	3
1 024	1 023	10

Wielowątkowość — max

W czym $\log_2 n$ jest lepszy od $n - 1$?

	n	$n - 1$	$\log_2 n$
	2	1	1
	4	3	2
	8	7	3
	1 024	1 023	10
(milion)	1 048 576	1 048 575	20

Wielowątkowość — max

W czym $\log_2 n$ jest lepszy od $n - 1$?

	n	$n - 1$	$\log_2 n$
	2	1	1
	4	3	2
	8	7	3
	1 024	1 023	10
(milion)	1 048 576	1 048 575	20
(miliard)	1 073 741 824	1 073 741 823	30

Wielowątkowość — max

W czym $\log_2 n$ jest lepszy od $n - 1$?

	n	$n - 1$	$\log_2 n$
	2	1	1
	4	3	2
	8	7	3
	1 024	1 023	10
(milion)	1 048 576	1 048 575	20
(miliard)	1 073 741 824	1 073 741 823	30
(bilion)	1 099 511 627 776	1 099 511 627 775	40

Wielowątkowość — max

W czym $\log_2 n$ jest lepszy od $n - 1$?

	n	$n - 1$	$\log_2 n$
	2	1	1
	4	3	2
	8	7	3
	1 024	1 023	10
(milion)	1 048 576	1 048 575	20
(miliard)	1 073 741 824	1 073 741 823	30
(bilion)	1 099 511 627 776	1 099 511 627 775	40

Ale...

...trzeba by mieć bardzo wiele procesorów.

Wielowątkowość — max

max — dodawanie współbieżności:

```
int krok=1;
while (krok < ile) {
    for (int i=0; i+krok<ile; i+=2*krok) {
        zamien(i, i+krok);
    }
    krok *= 2;
}
```

Wielowątkowość — max

max — dodawanie współbieżności:

```
int krok=1;
while (krok < ile) {
    for (int i=0; i+krok<ile; i+=2*krok) {
        zamien(i, i+krok);
    }
    krok *= 2;
}
```

Wielowątkowość — max

max — dodawanie współbieżności:

```
int krok=1;
while (krok < ile) {
    for (int i=0; i+krok<ile; i+=2*krok) {
        zamien(i, i+krok);
    }
    krok *= 2;
}
```

```
int krok=1;
while (krok < ile) {
    for (int i=0; i+krok<ile; i+=2*krok) {
        new Thread(new Zamien(i, i+krok)).start();
    }
    krok *= 2;
}
```

zamiast zamieniać —
tylko zainicjalizować zamianę



Wielowątkowość — max

max — dodawanie współbieżności:

Klasa Zamien:

```
class Zamien implements Runnable {  
    . . . . .  
    public void run() {  
        if (a[i] > a[j]) {  
            pom = a[i]; a[i] = a[j]; a[j] = pom;  
        }  
    }  
}
```

iis.ans-elblag.pl/~stefan/Dydaktyka/ProgObiII/Prog12/W1-Max/

Wielowątkowość — max

max — dodawanie współbieżności:

Ale:

- wersja współbieżna działałaby znacznie szybciej, gdyby nie ograniczona liczba procesorów/rdzeni

Wielowątkowość — max

max — dodawanie współbieżności:

Ale:

- wersja współbieżna działałaby znacznie szybciej, gdyby nie ograniczona liczba procesorów/rdzeni;
- na komputerach, które mam do dyspozycji, tego przyspieszenia nie udało się stwierdzić eksperymentalnie

Wielowątkowość — max

max — dodawanie współbieżności:

Ale:

- wersja współbieżna działałaby znacznie szybciej, gdyby nie ograniczona liczba procesorów/rdzeni;
- na komputerach, które mam do dyspozycji, tego przyspieszenia nie udało się stwierdzić eksperymentalnie; prawdopodobnie z powodu zbyt małej liczby procesorów.

Gra w piłkę

Dotąd: wystartowany wątek działa niezależnie od wszystkiego, aż dobiegnie do końca.

Potrzebne: wątek biorący pod uwagę zachowanie innych wątków.

Gra w piłkę

Dotąd: wystartowany wątek działa niezależnie od wszystkiego, aż dobiegnie do końca.

Potrzebne: wątek biorący pod uwagę zachowanie innych wątków.



Gra w piłkę

Dotąd: wystartowany wątek działa niezależnie od wszystkiego, aż dobiegnie do końca.

Potrzebne: wątek biorący pod uwagę zachowanie innych wątków.

- piłka **przelatuje** przez białą (otwartą) bramkę, a **odbija się** od czerwonej (zamkniętej)



Gra w piłkę

Dotąd: wystartowany wątek działa niezależnie od wszystkiego, aż dobiegnie do końca.

Potrzebne: wątek biorący pod uwagę zachowanie innych wątków.

- piłka **przelatuje** przez białą (otwartą) bramkę, a **odbija się** od czerwonej (zamkniętej);
- gracz może zamknąć swoją bramkę i jednocześnie otworzyć bramkę przeciwnika



Gra w piłkę

Dotąd: wystartowany wątek działa niezależnie od wszystkiego, aż dobiegnie do końca.

Potrzebne: wątek biorący pod uwagę zachowanie innych wątków.

- piłka **przelatuje** przez białą (otwartą) bramkę, a **odbija się** od czerwonej (zamkniętej);
- gracz może zamknąć swoją bramkę i jednocześnie otworzyć bramkę przeciwnika;
- wątki:
 - **program główny** porusza piłkę,
 - **gracz 0** obsługuje klawisz **Ctrl**,
 - **gracz 1** obsługuje klawisz **Shift**.



Gra w piłkę

Dotąd: wystartowany wątek działa niezależnie od wszystkiego, aż dobiegnie do końca.

Potrzebne: wątek biorący pod uwagę zachowanie innych wątków.

- piłka **przelatuje** przez białą (otwartą) bramkę, a **odbija się** od czerwonej (zamkniętej);
- gracz może zamknąć swoją bramkę i jednocześnie otworzyć bramkę przeciwnika;
- wątki:
 - program główny porusza piłkę,
 - gracz 0 obsługuje klawisz **Ctrl**,
 - gracz 1 obsługuje klawisz **Shift**.



Zawsze jedna bramka otwarta, a druga zamknięta.

Gra w piłkę

Dotąd: wystartowany wątek działa niezależnie od wszystkiego, aż dobiegnie do końca.

Potrzebne: wątek biorący pod uwagę zachowanie innych wątków.

- piłka **przelatuje** przez białą (otwartą) bramkę, a **odbija się** od czerwonej (zamkniętej);
- gracz może zamknąć swoją bramkę i jednocześnie otworzyć bramkę przeciwnika;
- wątki:
 - program główny porusza piłkę,
 - gracz 0 obsługuje klawisz **Ctrl**,
 - gracz 1 obsługuje klawisz **Shift**.



Zawsze jedna bramka otwarta, a druga zamknięta.

iis.ans-elblag.pl/~stefan/Dydaktyka/Prog0biII/Prog12/W2-Gra

Argument wywołania:

liczba całkowita, od której zależy prędkość piłki — im mniejsza tym szybciej.

Gra w piłkę

Gracz 0:

```
if (e.getKeyCode() == KeyEvent.VK_CONTROL) {  
    bk.zamk[0] = true;  bk.zamk[1] = false;  
}
```

Gracz 1:

```
if (e.getKeyCode() == KeyEvent.VK_SHIFT) {  
    bk.zamk[1] = true;  bk.zamk[0] = false;  
}
```

Gra w piłkę

Gracz 0:

```
if (e.getKeyCode() == KeyEvent.VK_CONTROL) {  
    bk.zamk[0] = true;  bk.zamk[1] = false;  
}
```

Gracz 1:

```
if (e.getKeyCode() == KeyEvent.VK_SHIFT) {  
    bk.zamk[1] = true;  bk.zamk[0] = false;  
}
```

Zawsze jedna bramka otwarta, a druga zamknięta.

Gra w piłkę

Gracz 0:

```
if (e.getKeyCode() == KeyEvent.VK_CONTROL) {  
    bk.zamk[0] = true;  bk.zamk[1] = false;  
}
```

Gracz 1:

```
if (e.getKeyCode() == KeyEvent.VK_SHIFT) {  
    bk.zamk[1] = true;  bk.zamk[0] = false;  
}
```

Zawsze jedna bramka otwarta, a druga zamknięta.

A co, jeśli...

wciśnięcie Ctrl

Gra w piłkę

Gracz 0:

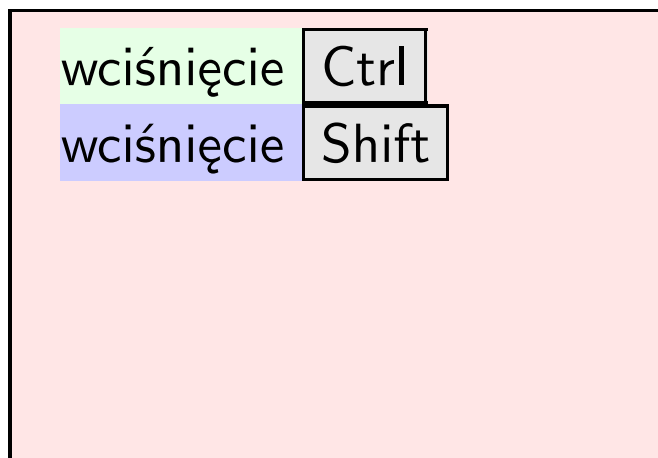
```
if (e.getKeyCode() == KeyEvent.VK_CONTROL) { ●  
    bk.zamk[0] = true;  bk.zamk[1] = false;  
}
```

Gracz 1:

```
if (e.getKeyCode() == KeyEvent.VK_SHIFT) { ●  
    bk.zamk[1] = true;  bk.zamk[0] = false;  
}
```

Zawsze jedna bramka otwarta, a druga zamknięta.

A co, jeśli...



Gra w piłkę

Gracz 0:

```
if (e.getKeyCode() == KeyEvent.VK_CONTROL) {  
    bk.zamk[0] = true;  bk.zamk[1] = false;  
}
```

Gracz 1:

```
if (e.getKeyCode() == KeyEvent.VK_SHIFT) {   
    bk.zamk[1] = true; bk.zamk[0] = false;  
}
```

Zawsze jedna bramka otwarta, a druga zamknięta.

A co, jeśli...

wciśnięcie

Ctrl

wciśnięcie

Shift

bk.zamk[0] = true;

Gra w piłkę

Gracz 0:

```
if (e.getKeyCode() == KeyEvent.VK_CONTROL) {  
    bk.zamk[0] = true;  bk.zamk[1] = false;  
}
```

Gracz 1:

```
if (e.getKeyCode() == KeyEvent.VK_SHIFT) {  
    bk.zamk[1] = true;  bk.zamk[0] = false;  
}
```

Zawsze jedna bramka otwarta, a druga zamknięta.

A co, jeśli...

wciśnięcie

Ctrl

wciśnięcie

Shift

bk.zamk[0] = true;

bk.zamk[1] = true;

Gra w piłkę

Gracz 0:

```
if (e.getKeyCode() == KeyEvent.VK_CONTROL) {
    bk.zamk[0] = true;  bk.zamk[1] = false; 
}
```

Gracz 1:

```
if (e.getKeyCode() == KeyEvent.VK_SHIFT) {
    bk.zamk[1] = true;  bk.zamk[0] = false;
}
```

Zawsze jedna bramka otwarta, a druga zamknięta.

A co, jeśli...

wciśnięcie

Ctrl

wciśnięcie

Shift

bk.zamk[0] = true;

bk.zamk[1] = true;

bk.zamk[1] = false;

Gra w piłkę

Gracz 0:

```
if (e.getKeyCode() == KeyEvent.VK_CONTROL) {  
    bk.zamk[0] = true;  bk.zamk[1] = false;   
}
```

Gracz 1:

```
if (e.getKeyCode() == KeyEvent.VK_SHIFT) {  
    bk.zamk[1] = true;  bk.zamk[0] = false;   
}
```

Zawsze jedna bramka otwarta, a druga zamknięta.

A co, jeśli...

wciśnięcie Ctrl

wciśnięcie Shift

bk.zamk[0] = true;

bk.zamk[1] = true;

bk.zamk[1] = false;

bk.zamk[0] = false;

Gra w piłkę

Gracz 0:

```
if (e.getKeyCode() == KeyEvent.VK_CONTROL) {
    bk.zamk[0] = true;  bk.zamk[1] = false; 
}
```

Gracz 1:

```
if (e.getKeyCode() == KeyEvent.VK_SHIFT) {
    bk.zamk[1] = true;  bk.zamk[0] = false; 
}
```

Zawsze jedna bramka otwarta, a druga zamknięta.

A co, jeśli...

wciśnięcie Ctrl

wciśnięcie Shift

bk.zamk[0] = true;

bk.zamk[1] = true;

bk.zamk[1] = false;

bk.zamk[0] = false;

...i obie otwarte...

Problemy programisty ze współbieżnością

Założenie: wątki (ew. procesy) działają **współbieżnie**, czyli programista nie może niczego zakładać o kolejności wykonywania ich komend.

Problemy programisty ze współbieżnością

Założenie: wątki (ew. procesy) działają **współbieżnie**, czyli programista nie może niczego zakładać o kolejności wykonywania ich komend.

Wtedy nawet wynik **pojedynczych przypisań** jest niepewny.

Problemy programisty ze współbieżnością

Założenie: wątki (ew. procesy) działają **współbieżnie**, czyli programista nie może niczego zakładać o kolejności wykonywania ich komend.

Wtedy nawet wynik **pojedynczych przypisań** jest niepewny.

Na przykład:

komenda	składa się z
<code>x = x+1;</code>	

Problemy programisty ze współbieżnością

Założenie: wątki (ew. procesy) działają **współbieżnie**, czyli programista nie może niczego zakładać o kolejności wykonywania ich komend.

Wtedy nawet wynik **pojedynczych przypisań** jest niepewny.

Na przykład:

komenda	składa się z
<code>x = x+1;</code>	pobrać x

Problemy programisty ze współbieżnością

Założenie: wątki (ew. procesy) działają **współbieżnie**, czyli programista nie może niczego zakładać o kolejności wykonywania ich komend.

Wtedy nawet wynik **pojedynczych przypisań** jest niepewny.

Na przykład:

komenda	składa się z
$x = x + 1;$	pobrać x powiększyć o 1

Problemy programisty ze współbieżnością

Założenie: wątki (ew. procesy) działają **współbieżnie**, czyli programista nie może niczego zakładać o kolejności wykonywania ich komend.

Wtedy nawet wynik **pojedynczych przypisań** jest niepewny.

Na przykład:

komenda	składa się z
<code>x = x+1;</code>	pobrać x powiększyć o 1 zapisać do x

Problemy programisty ze współbieżnością

Założenie: wątki (ew. procesy) działają **współbieżnie**, czyli programista nie może niczego zakładać o kolejności wykonywania ich komend.

Wtedy nawet wynik **pojedynczych przypisań** jest niepewny.

Na przykład:

komenda	składa się z
$x = x + 1;$	pobrać x powiększyć o 1 zapisać do x

Dwa wątki zawierające taką komendę mogą sobie przeszkadzać...

Problemy programisty ze współbieżnością

Założenie: wątki (ew. procesy) działają **współbieżnie**, czyli programista nie może niczego zakładać o kolejności wykonywania ich komend:

Wtedy nawet wynik **pojedynczych przypisań** jest niepewny.

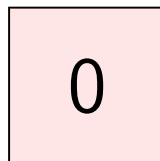
wątek niebieski

pobrać x
zwiększyć o 1
zapisać w x



x

0



wątek zielony

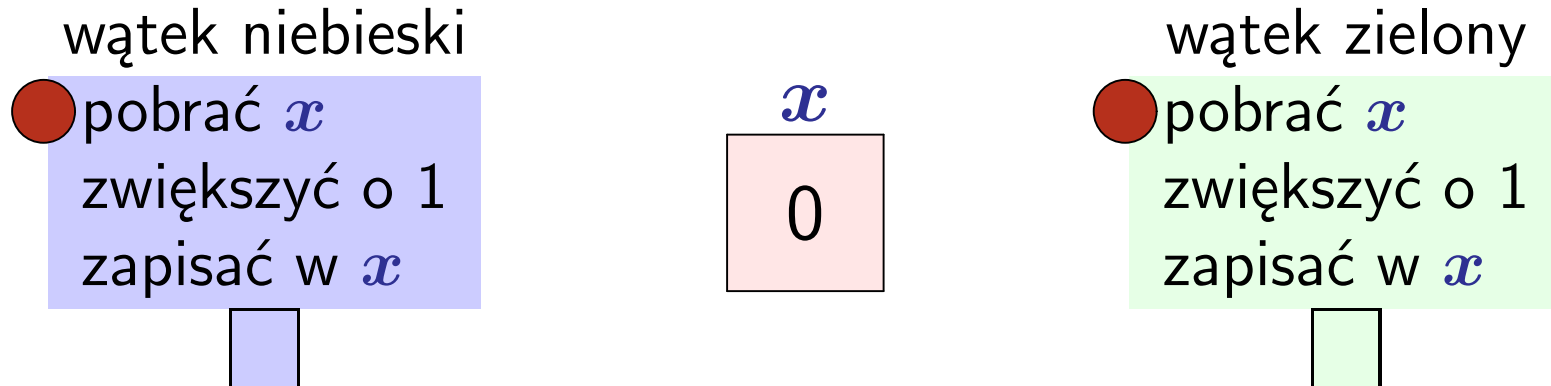
pobrać x
zwiększyć o 1
zapisać w x



Problemy programisty ze współbieżnością

Założenie: wątki (ew. procesy) działają **współbieżnie**, czyli programista nie może niczego zakładać o kolejności wykonywania ich komend:

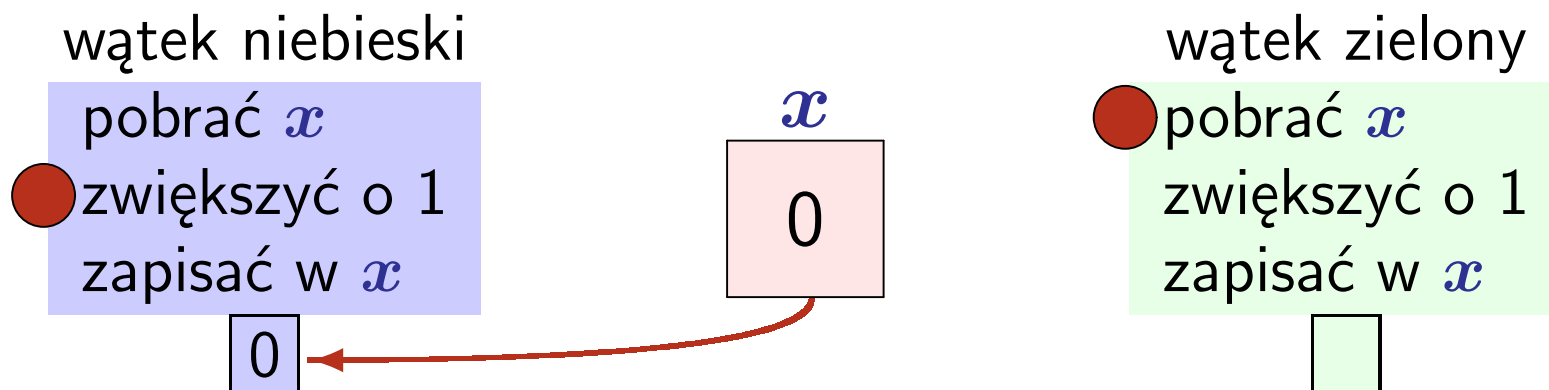
Wtedy nawet wynik **pojedynczych przypisań** jest niepewny.



Problemy programisty ze współbieżnością

Założenie: wątki (ew. procesy) działają **współbieżnie**, czyli programista nie może niczego zakładać o kolejności wykonywania ich komend.

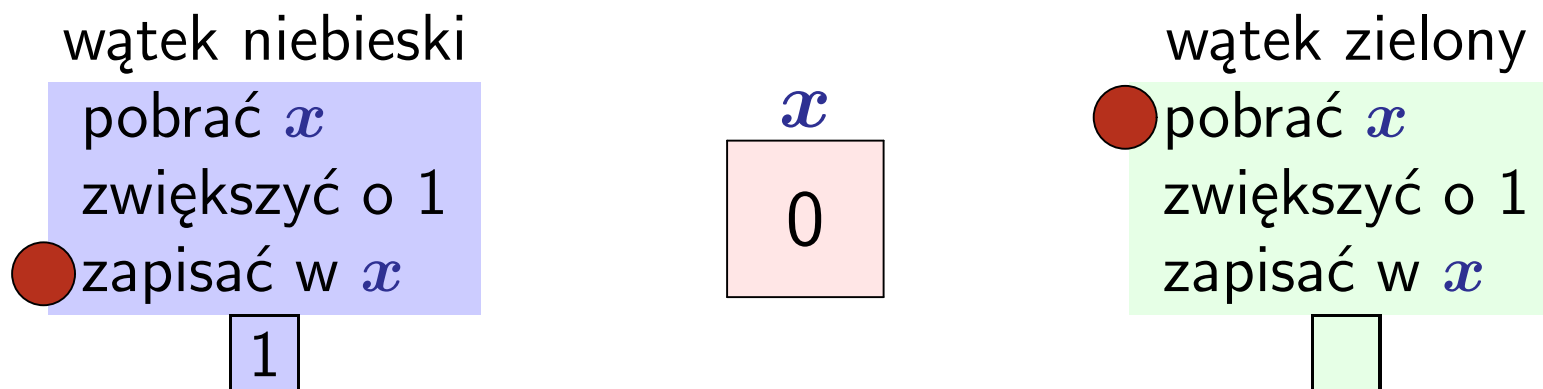
Wtedy nawet wynik **pojedynczych przypisań** jest niepewny.



Problemy programisty ze współbieżnością

Założenie: wątki (ew. procesy) działają **współbieżnie**, czyli programista nie może niczego zakładać o kolejności wykonywania ich komend.

Wtedy nawet wynik **pojedynczych przypisań** jest niepewny.



Problemy programisty ze współbieżnością

Założenie: wątki (ew. procesy) działają **współbieżnie**, czyli programista nie może niczego zakładać o kolejności wykonywania ich komend.

Wtedy nawet wynik **pojedynczych przypisań** jest niepewny.

wątek niebieski

pobrać x
zwiększyć o 1
zapisać w x ●

1

x

1

wątek zielony

● pobrać x
zwiększyć o 1
zapisać w x



Problemy programisty ze współbieżnością

Założenie: wątki (ew. procesy) działają **współbieżnie**, czyli programista nie może niczego zakładać o kolejności wykonywania ich komend.

Wtedy nawet wynik **pojedynczych przypisań** jest niepewny.

wątek niebieski

pobrać x

zwiększyć o 1

zapisać w x ●

1

x

1

wątek zielony

pobrać x

● zwiększyć o 1

zapisać w x

1



Problemy programisty ze współbieżnością

Założenie: wątki (ew. procesy) działają **współbieżnie**, czyli programista nie może niczego zakładać o kolejności wykonywania ich komend.

Wtedy nawet wynik **pojedynczych przypisań** jest niepewny.

wątek niebieski

pobrać x

zwiększyć o 1

zapisać w x ●

1

x

1

wątek zielony

pobrać x

zwiększyć o 1

● zapisać w x

2

Problemy programisty ze współbieżnością

Założenie: wątki (ew. procesy) działają **współbieżnie**, czyli programista nie może niczego zakładać o kolejności wykonywania ich komend.

Wtedy nawet wynik **pojedynczych przypisań** jest niepewny.

wątek niebieski

pobrać x

zwiększyć o 1

zapisać w x ●

1

x

2

wątek zielony

pobrać x

zwiększyć o 1

zapisać w x ●

2



Problemy programisty ze współbieżnością

Założenie: wątki (ew. procesy) działają **współbieżnie**, czyli programista nie może niczego zakładać o kolejności wykonywania ich komend.

Wtedy nawet wynik **pojedynczych przypisań** jest niepewny.

wątek niebieski

pobrać x

zwiększyć o 1

zapisać w x ●

1

x

2

wątek zielony

pobrać x

zwiększyć o 1

zapisać w x ●

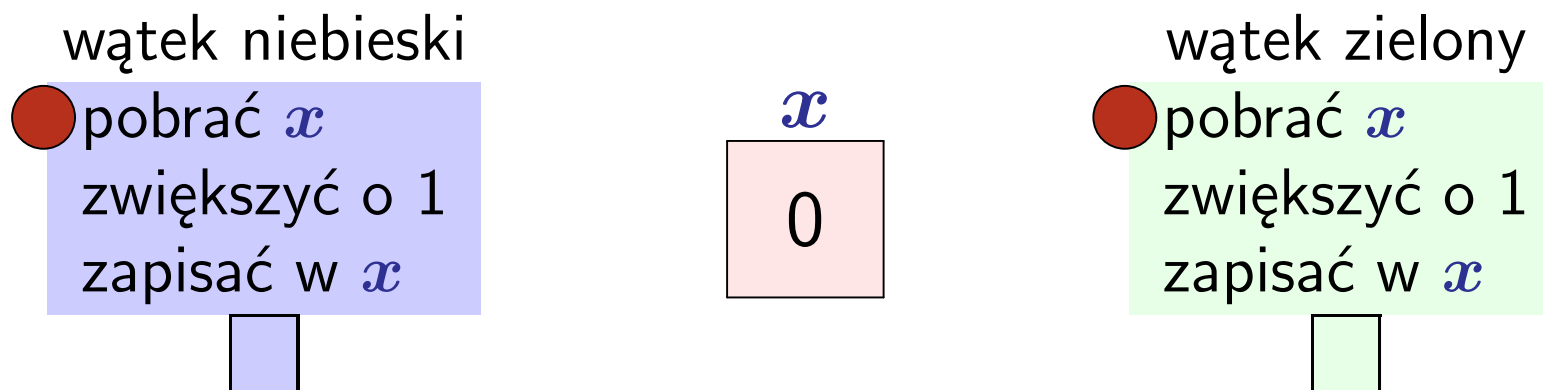
2

Po obliczeniu 1: $x = 2$

Problemy programisty ze współbieżnością

Założenie: wątki (ew. procesy) działają **współbieżnie**, czyli programista nie może niczego zakładać o kolejności wykonywania ich komend:

Wtedy nawet wynik **pojedynczych przypisań** jest niepewny.

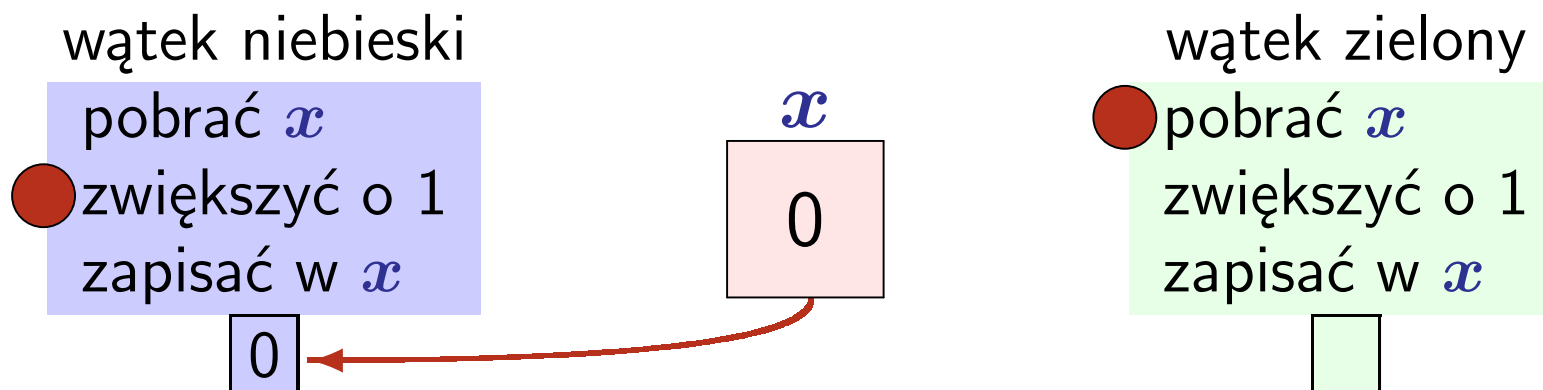


Po obliczeniu 1: $x = 2$

Problemy programisty ze współbieżnością

Założenie: wątki (ew. procesy) działają **współbieżnie**, czyli programista nie może niczego zakładać o kolejności wykonywania ich komend.

Wtedy nawet wynik **pojedynczych przypisań** jest niepewny.

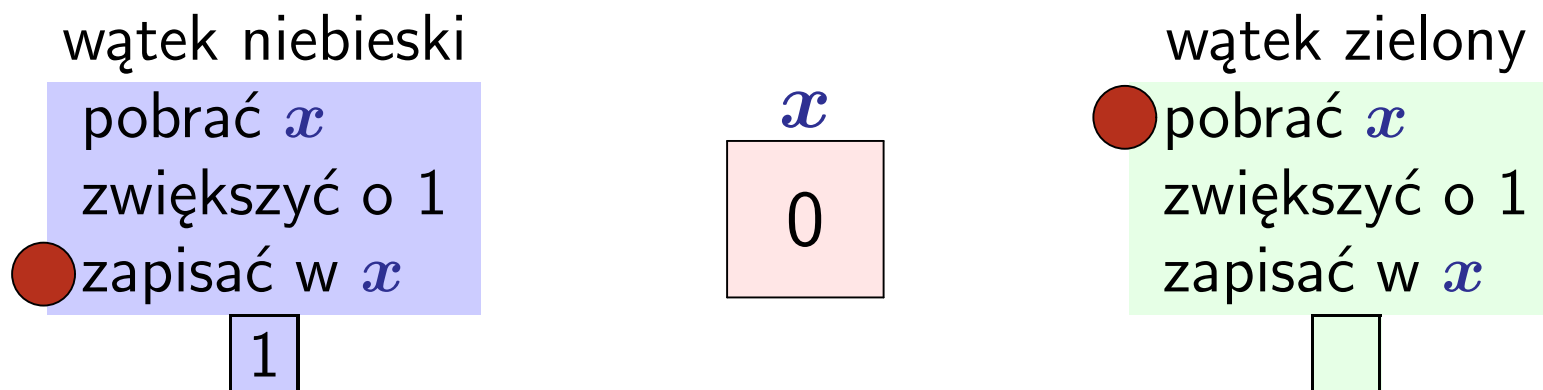


Po obliczeniu 1: $x = 2$

Problemy programisty ze współbieżnością

Założenie: wątki (ew. procesy) działają **współbieżnie**, czyli programista nie może niczego zakładać o kolejności wykonywania ich komend.

Wtedy nawet wynik **pojedynczych przypisań** jest niepewny.

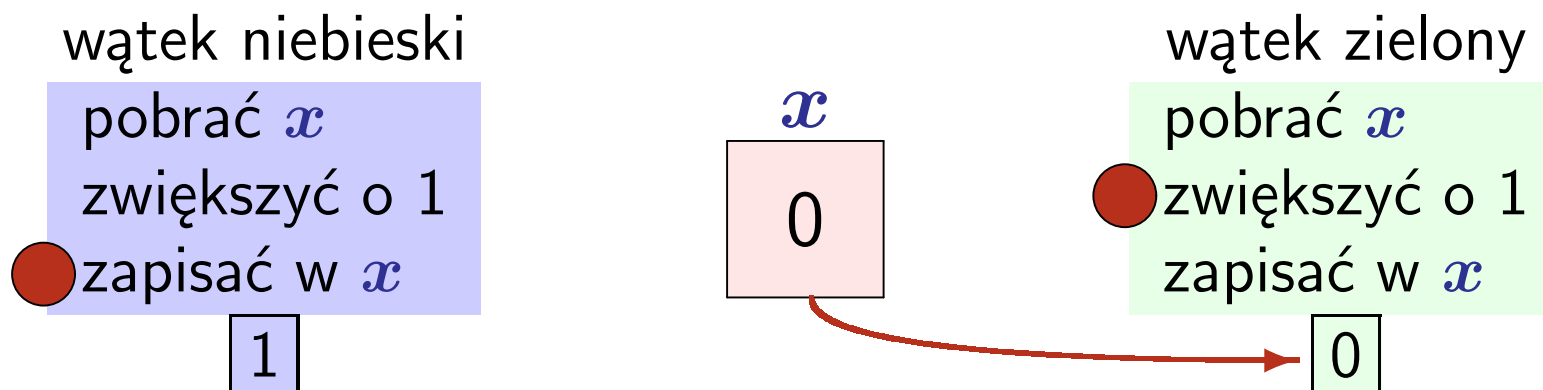


Po obliczeniu 1: $x = 2$

Problemy programisty ze współbieżnością

Założenie: wątki (ew. procesy) działają **współbieżnie**, czyli programista nie może niczego zakładać o kolejności wykonywania ich komend.

Wtedy nawet wynik **pojedynczych przypisań** jest niepewny.

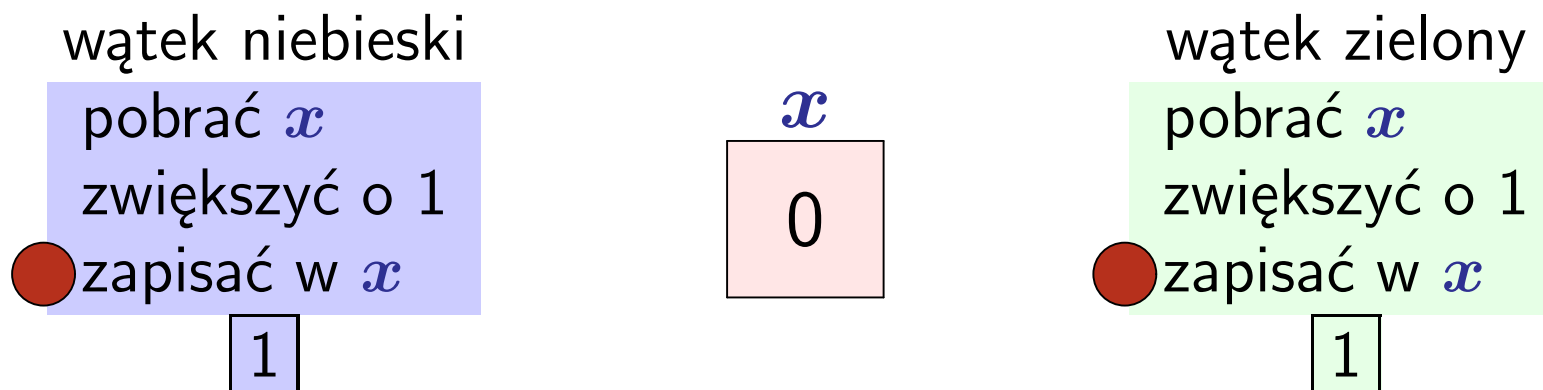


Po obliczeniu 1: $x = 2$

Problemy programisty ze współbieżnością

Założenie: wątki (ew. procesy) działają **współbieżnie**, czyli programista nie może niczego zakładać o kolejności wykonywania ich komend.

Wtedy nawet wynik **pojedynczych przypisań** jest niepewny.

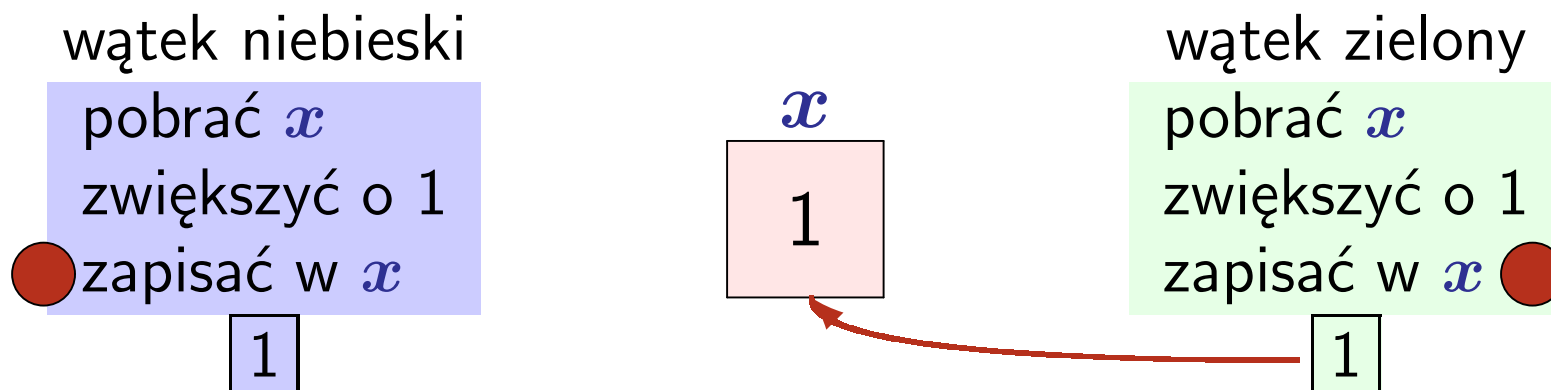


Po obliczeniu 1: $x = 2$

Problemy programisty ze współbieżnością

Założenie: wątki (ew. procesy) działają **współbieżnie**, czyli programista nie może niczego zakładać o kolejności wykonywania ich komend.

Wtedy nawet wynik **pojedynczych przypisań** jest niepewny.

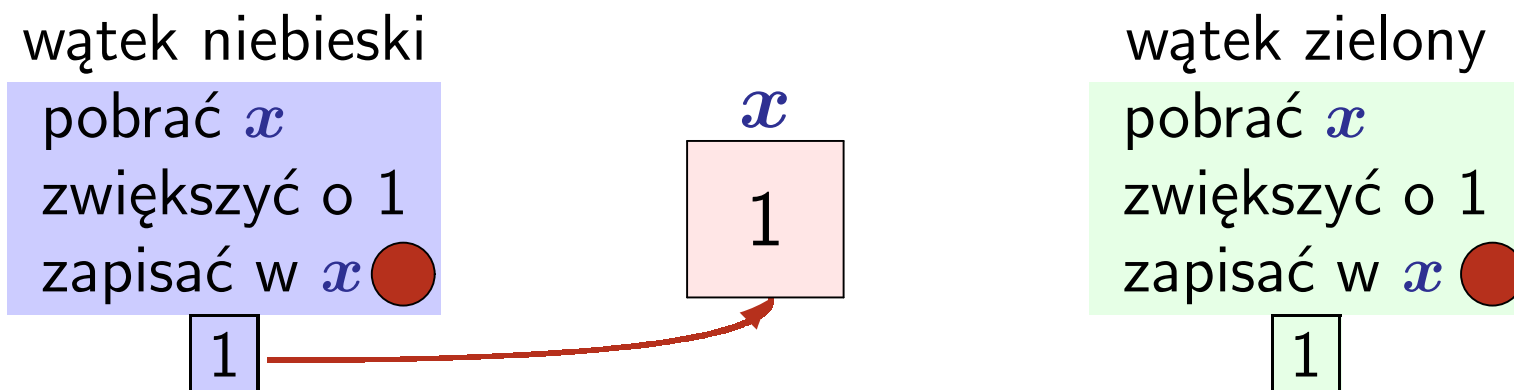


Po obliczeniu 1: $x = 2$

Problemy programisty ze współbieżnością

Założenie: wątki (ew. procesy) działają **współbieżnie**, czyli programista nie może niczego zakładać o kolejności wykonywania ich komend.

Wtedy nawet wynik **pojedynczych przypisań** jest niepewny.



Po obliczeniu 1: $x = 2$

Problemy programisty ze współbieżnością

Założenie: wątki (ew. procesy) działają **współbieżnie**, czyli programista nie może niczego zakładać o kolejności wykonywania ich komend.

Wtedy nawet wynik **pojedynczych przypisań** jest niepewny.

wątek niebieski

pobrać x

zwiększyć o 1

zapisać w x ●

1

x

1

wątek zielony

pobrać x

zwiększyć o 1

zapisać w x ●

1

Po obliczeniu 1: $x = 2$

Po obliczeniu 2: $x = 1$

Problemy programisty ze współbieżnością

```
class Zmienna {
    public static int x=0;
    public static void up() { x = x+1; }
}

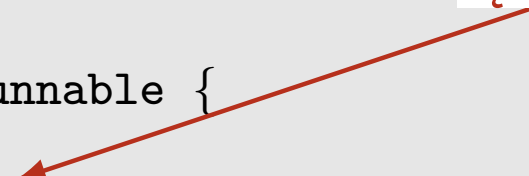
class WGore implements Runnable {
    public void run() {
        for (int i=0; i<100000; i++)
            Zmienna.up();
    }
}

class Test {
    public static void main(String[] args)
        throws InterruptedException {
        Thread niebieski = new Thread(new WGore());
        Thread zielony = new Thread(new WGore());
        niebieski.start(); zielony.start();
        niebieski.join(); zielony.join();
        System.out.println(" x == " + Zmienna.x);
    }
}
```

Problemy programisty ze współbieżnością

```
class Zmienna {  
    public static int x=0;  
    public static void up() { x = x+1; }  
}  
  
class WGore implements Runnable {  
    public void run() {  
        for (int i=0; i<100000; i++)  
            Zmienna.up();  
    }  
}  
  
class Test {  
    public static void main(String[] args)  
        throws InterruptedException {  
        Thread niebieski = new Thread(new WGore());  
        Thread zielony = new Thread(new WGore());  
        niebieski.start(); zielony.start();  
        niebieski.join(); zielony.join();  
        System.out.println(" x == " + Zmienna.x);  
    }  
}
```

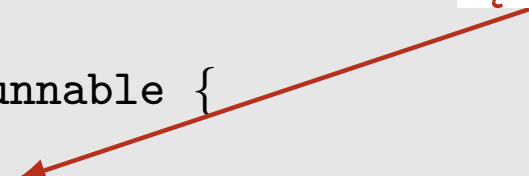
przy tylu powtórzeniach
są wysokie szanse, że paradoks
się ujawni



Problemy programisty ze współbieżnością

```
class Zmienna {  
    public static int x=0;  
    public static void up() { x = x+1; }  
}  
  
class WGore implements Runnable {  
    public void run() {  
        for (int i=0; i<100000; i++)  
            Zmienna.up();  
    }  
}  
  
class Test {  
    public static void main(String[] args)  
        throws InterruptedException {  
        Thread niebieski = new Thread(new WGore());  
        Thread zielony = new Thread(new WGore());  
        niebieski.start(); zielony.start();  
        niebieski.join(); zielony.join();  
        System.out.println(" x == " + Zmienna.x);  
    }  
}
```

przy tylu powtórzeniach
są wysokie szanse, że paradoks
się ujawni



Niezależnie działające wątki ścigają się
o „ostatnie słowo”
w modyfikacji tej samej zmiennej.

Problemy programisty ze współbieżnością

```
class Zmienna {  
    public static int x=0;  
    public static void up() { x = x+1; }  
}  
  
class WGore implements Runnable {  
    public void run() {  
        for (int i=0; i<100000; i++)  
            Zmienna.up();  
    }  
}  
  
class Test {  
    public static void main(String[] args)  
        throws InterruptedException {  
        Thread niebieski = new Thread(new WGore());  
        Thread zielony = new Thread(new WGore());  
        niebieski.start(); zielony.start();  
        niebieski.join(); zielony.join();  
        System.out.println(" x == " + Zmienna.x);  
    }  
}
```

przy tylu powtórzeniach
są wysokie szanse, że paradoks
się ujawni

Niezależnie działające wątki ścigają się
o „ostatnie słowo”
w modyfikacji tej samej zmiennej.

Problemy programisty ze współbieżnością

```
class Zmienna {  
    public static int x=0;  
    public static void up() { x = x+1; }  
}  
  
class WGore implements Runnable {  
    public void run() {  
        for (int i=0; i<100000; i++)  
            Zmienna.up();  
    }  
}  
  
class Test {  
    public static void main(String[] args)  
        throws InterruptedException {  
        Thread niebieski = new Thread(new WGore());  
        Thread zielony = new Thread(new WGore());  
        niebieski.start(); zielony.start();  
        niebieski.join(); zielony.join();  
        System.out.println(" x == " + Zmienna.x);  
    }  
}
```

przy tylu powtórzeniach
są wysokie szanse, że paradoks
się ujawni

Niezależnie działające wątki ścigają się
o „ostatnie słowo”
w modyfikacji tej samej zmiennej.

Początek i koniec wątku

trzeba

synchronizować

wątki

Początek i koniec wątku

trzeba
synchronizować
wątki

`t.start()` — rozpocznij działanie
wątku `t`

Początek i koniec wątku

trzeba
synchronizować
wątki

`t.start()` — rozpocznij działanie wątku `t`

`t.join()` — zaczekaj, aż `t` zakończy działanie

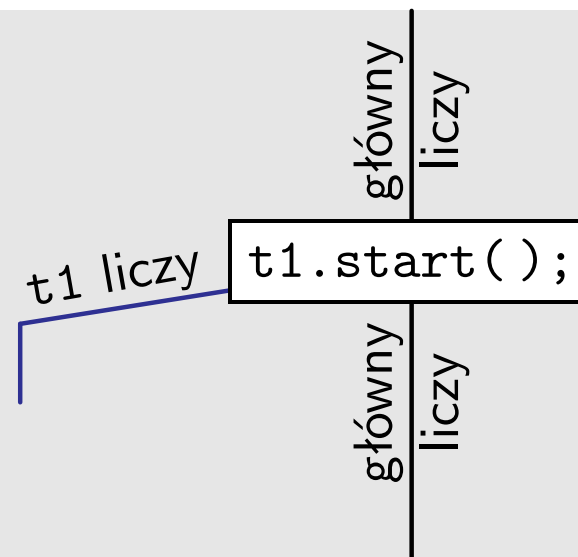
Początek i koniec wątku

główny
liczy

trzeba
synchronizować
wątki

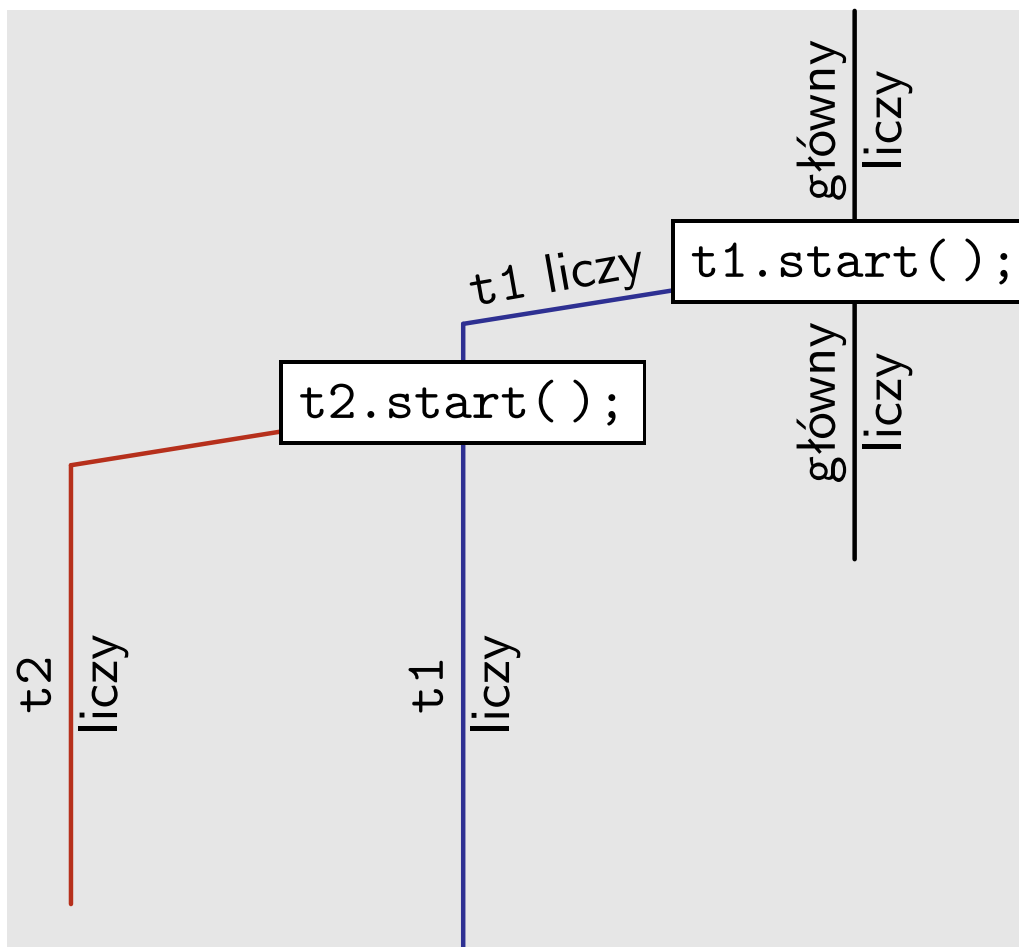
Początek i koniec wątku

trzeba
synchronizować
wątki



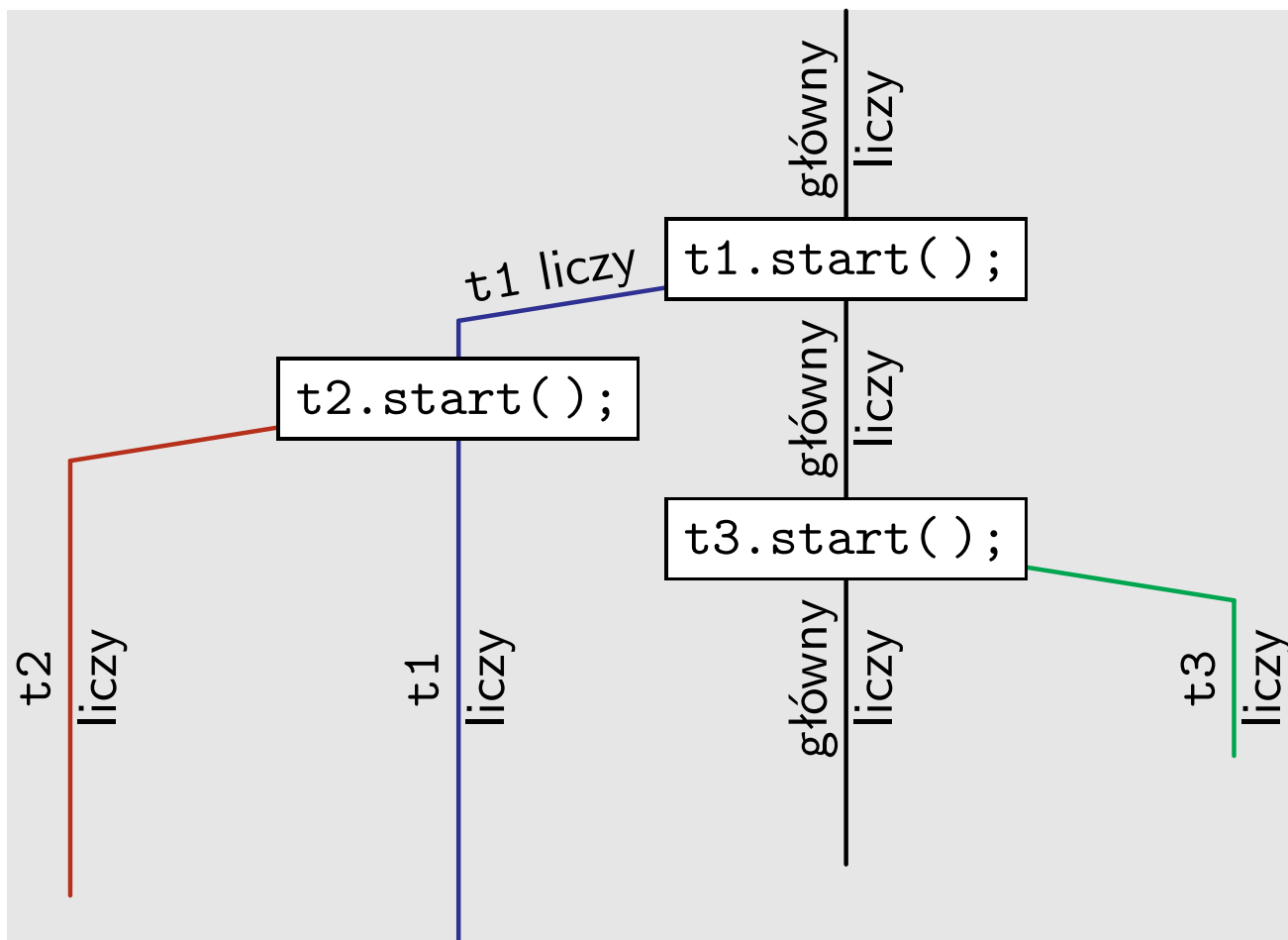
Początek i koniec wątku

trzeba
synchronizować
wątki



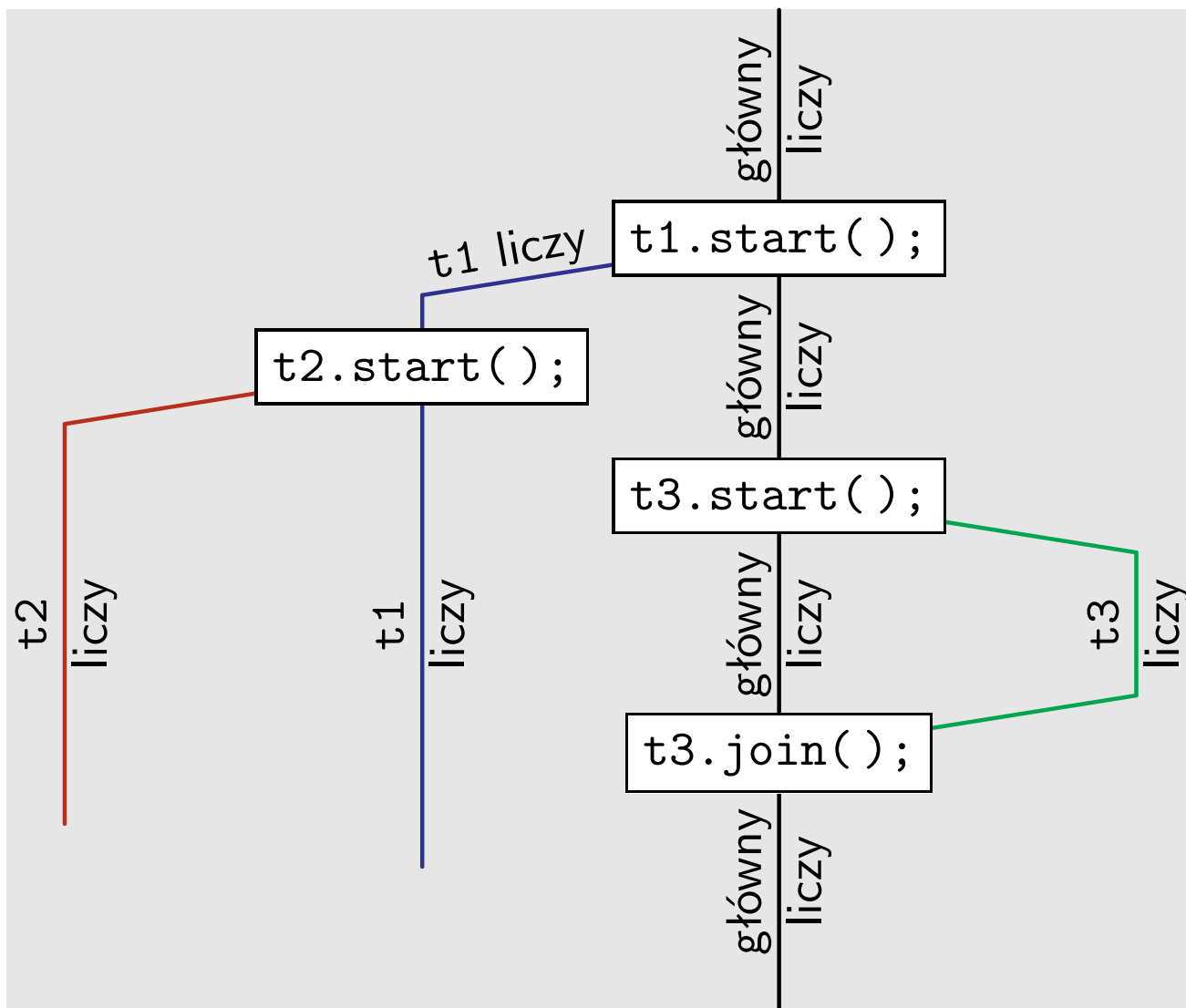
Początek i koniec wątku

trzeba
synchronizować
wątki



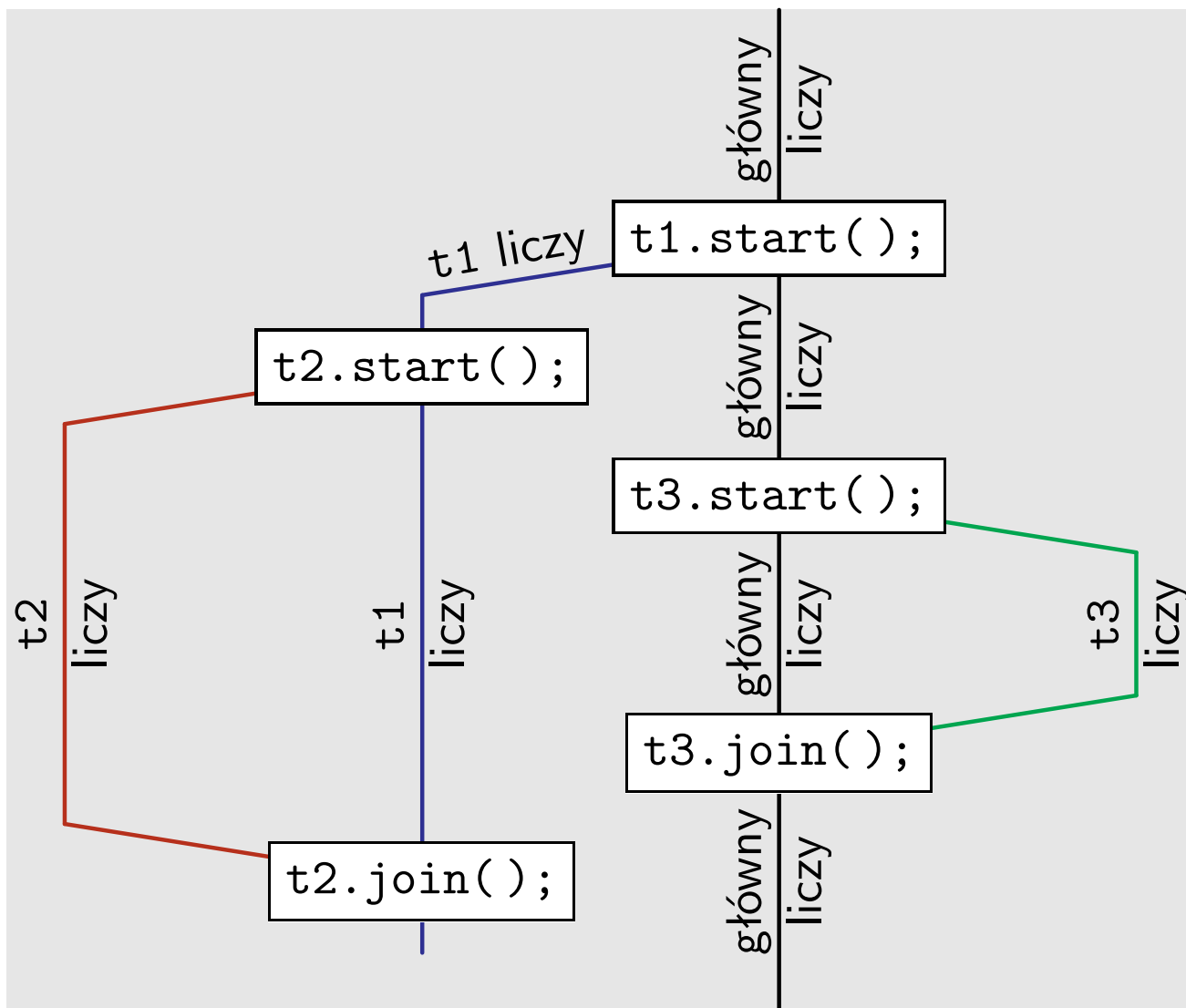
Początek i koniec wątku

trzeba
synchronizować
wątki



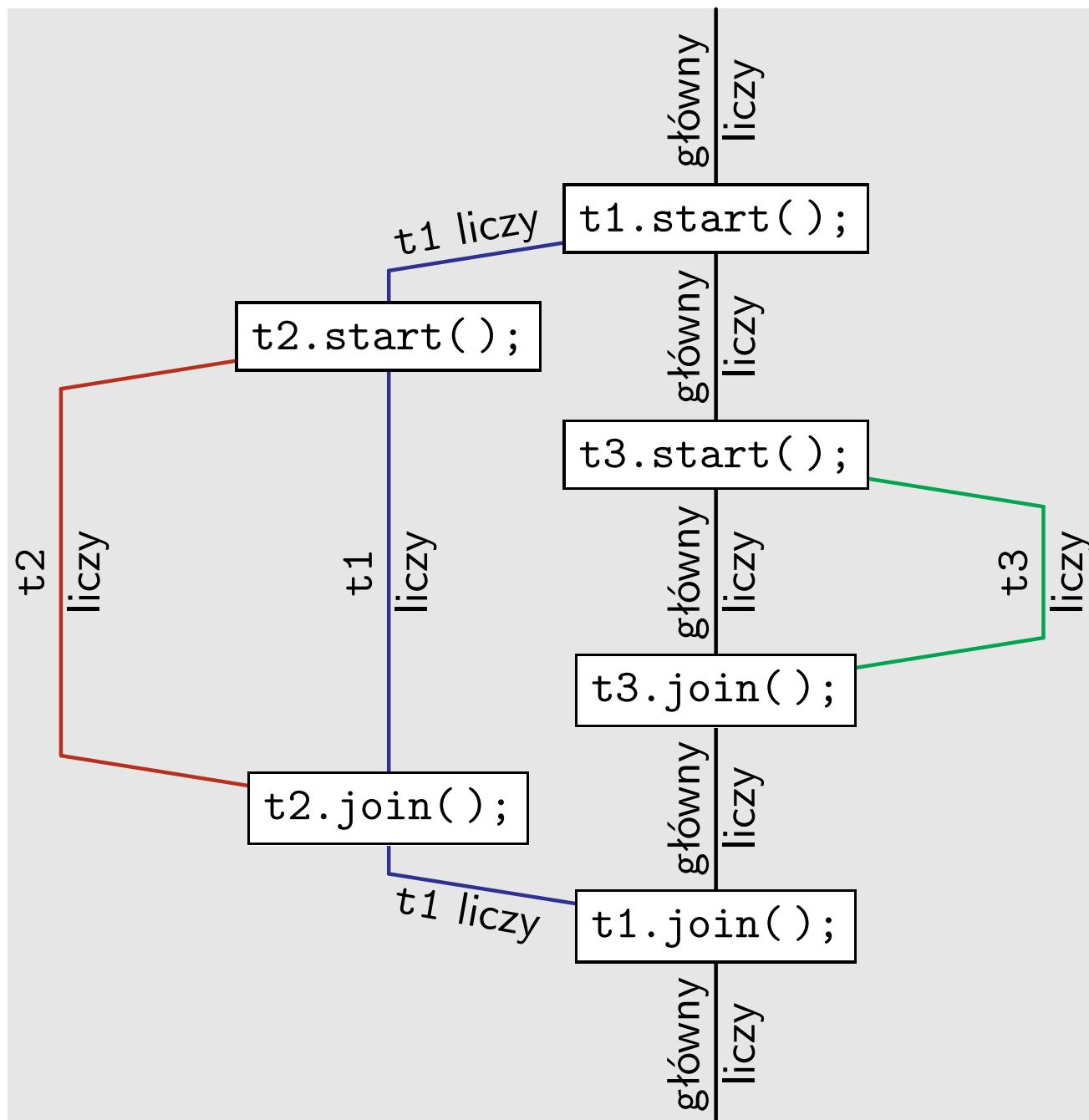
Początek i koniec wątku

trzeba
synchronizować
wątki



Początek i koniec wątku

trzeba
synchronizować
wątki



Wyścig (*race condition*)

— kiedy kilka współbieżnych wątków „ściga się” o dostęp do wspólnego zasobu.

Wyścig (*race condition*)

— kiedy kilka współbieżnych wątków „ściga się” o dostęp do wspólnego zasobu.

Skutkiem wyścigu może być zaburzenie integralności (sprzeczny stan) tego zasobu...

Wyścig (*race condition*)

— kiedy kilka współbieżnych wątków „ściga się” o dostęp do wspólnego zasobu.

Skutkiem wyścigu może być zaburzenie integralności (sprzeczny stan) tego zasobu...

Przykład:

Magazynier A:

Samochód

80 000 PLN

Spis towarów:

Magazynier B:

Maseczka

2 PLN

Wyścig (*race condition*)

— kiedy kilka współbieżnych wątków „ściga się” o dostęp do wspólnego zasobu.

Skutkiem wyścigu może być zaburzenie integralności (sprzeczny stan) tego zasobu. . .

Przykład:

Magazynier A:

Samochód

80 000 PLN

Spis towarów:

Maseczka

Magazynier B:

Maseczka

2 PLN

Wyścig (*race condition*)

— kiedy kilka współbieżnych wątków „ściga się” o dostęp do wspólnego zasobu.

Skutkiem wyścigu może być zaburzenie integralności (sprzeczny stan) tego zasobu...

Przykład:

Magazynier A:

Samochód

80 000 PLN

Spis towarów:

Samochód

Magazynier B:

Maseczka

2 PLN

Wyścig (*race condition*)

— kiedy kilka współbieżnych wątków „ściga się” o dostęp do wspólnego zasobu.

Skutkiem wyścigu może być zaburzenie integralności (sprzeczny stan) tego zasobu...

Przykład:

Magazynier A:

Samochód

80 000 PLN

Spis towarów:

Samochód

80 000 PLN

Magazynier B:

Maseczka

2 PLN

Wyścig (*race condition*)

— kiedy kilka współbieżnych wątków „ściga się” o dostęp do wspólnego zasobu.

Skutkiem wyścigu może być zaburzenie integralności (sprzeczny stan) tego zasobu. . .

Przykład:

Magazynier A:

Samochód

80 000 PLN

Spis towarów:

Samochód

2 PLN

Magazynier B:

Maseczka

2 PLN

Wyścig (*race condition*)

— kiedy kilka współbieżnych wątków „ściga się” o dostęp do wspólnego zasobu.

Skutkiem wyścigu może być zaburzenie integralności (sprzeczny stan) tego zasobu...

Przykład:

Magazynier A:

Samochód

80 000 PLN

Spis towarów:

Samochód

2 PLN

Magazynier B:

Maseczka

2 PLN

Żeby uniknąć wyścigu:

kiedy jeden wątek zacznie działać na zasobie,
inne wątki chcące na nim działać trzeba powstrzymać (muszą poczekać).

Wyścig (*race condition*)

— kiedy kilka współbieżnych wątków „ściga się” o dostęp do wspólnego zasobu.

Skutkiem wyścigu może być zaburzenie integralności (sprzeczny stan) tego zasobu...

Przykład:

Magazynier A:

Samochód

80 000 PLN

Spis towarów:

Samochód

2 PLN

Magazynier B:

Maseczka

2 PLN

Żeby uniknąć wyścigu:

kiedy jeden wątek zacznie działać na zasobie,
inne wątki chcące na nim działać trzeba powstrzymać (muszą poczekać).

W Jawie — metoda „*synchronized*”.

Wyścig (*race condition*)

```
class Zmienna {  
    public static int x=0;  
    public static void up() { x = x+1; }  
}
```

Wyścig (*race condition*)

```
class Zmienna {  
    public static int x=0;  
    public static void up() { x = x+1; }  
}
```



```
class Zmienna {  
    public static int x=0;  
    public static synchronized void up() { x = x+1; }  
}
```

Wyścig (*race condition*)

```
class Zmienna {  
    public static int x=0;  
    public static void up() { x = x+1; }  
}
```



```
class Zmienna {  
    public static int x=0;  
    public static synchronized void up() { x = x+1; }  
}
```

Teraz poprzedni program będzie drukował 200 000.

Wyścig (*race condition*)

```
class Zmienna {  
    public static int x=0;  
    public static void up() { x = x+1; }  
}
```



```
class Zmienna {  
    public static int x=0;  
    public static synchronized void up() { x = x+1; }  
}
```

Teraz poprzedni program będzie drukował 200 000.

Ale tracimy część współbieżności — kiedy wątki na siebie czekają, sumaryczny czas ich działania **rośnie**.

Wyścig (*race condition*)

Niepodzielność synchronized — odmiany

Wyścig (*race condition*)

Niepodzielność synchronized — odmiany

Synchronizowana **metoda w obiekcie**:

```
public synchronized typ_wynikowy nazwa { ciało }
```

— ciało tej metody może być jednocześnie wykonywane tylko przez **je-**
den wątek na obiekt; w całym programie tyle jednocześnie działających
wątków ile jest obiektów.

Wyścig (*race condition*)

Niepodzielność synchronized — odmiany

Synchronizowana **metoda w obiekcie**:

```
public synchronized typ_wynikowy nazwa { ciało }
```

— ciało tej metody może być jednocześnie wykonywane tylko przez **je-**
den wątek na obiekt; w całym programie tyle jednocześnie działających
wątków ile jest obiektów.

Przykład:

```
class Konto {  
    private double stanKonta;  
    . . . . .  
    public synchronized void wplata(double k) { stanKonta += k; }  
}
```

Wyścig (*race condition*)

Niepodzielność synchronized — odmiany

Synchronizowana **metoda w obiekcie**:

```
public synchronized typ_wynikowy nazwa { ciało }
```

— ciało tej metody może być jednocześnie wykonywane tylko przez **je-**
den wątek na obiekt; w całym programie tyle jednocześnie działających
wątków ile jest obiektów.

Przykład:

```
class Konto {  
    private double stanKonta;  
    . . . . .  
    public synchronized void wplata(double k) { stanKonta += k; }  
}
```

— dwa różne wątki nie mogą jednocześnie wpłacać na to samo konto,
ale na różne konta mogą.

Wyścig (*race condition*)

```
class Konto {  
    .....  
    public synchronized void wplata(double k) { ... }  
    .....  
}
```

```
Konto konto1 = new Konto();  
Konto konto2 = new Konto();
```

Wyścig (*race condition*)

```
class Konto {  
    .....  
    public synchronized void wplata(double k) { ... }  
    .....  
}
```

```
Konto konto1 = new Konto();
```

```
Konto konto2 = new Konto();
```

jakiś wątek

konto1.wplata(1000.00)

inny wątek

konto2.wplata(1000.00)

Wyścig (*race condition*)

```
class Konto {  
    .....  
    public synchronized void wplata(double k) { ... }  
    .....  
}
```

```
Konto konto1 = new Konto();
```

```
Konto konto2 = new Konto();
```

jakiś wątek

konto1.wplata(1000.00)

jednoczesne
wykonanie
OK

inny wątek

konto2.wplata(1000.00)

Wyścig (*race condition*)

```
class Konto {  
    .....  
    public synchronized void wplata(double k) { ... }  
    .....  
}
```

```
Konto konto1 = new Konto();
```

```
Konto konto2 = new Konto();
```

jakiś wątek

konto1.wplata(1000.00)

jednoczesne
wykonanie
OK

inny wątek

konto2.wplata(1000.00)

jakiś wątek

konto1.wplata(1000.00)

inny wątek

konto1.wplata(1000.00)

Wyścig (*race condition*)

```
class Konto {  
    .....  
    public synchronized void wplata(double k) { ... }  
    .....  
}
```

```
Konto konto1 = new Konto();
```

```
Konto konto2 = new Konto();
```

jakiś wątek
konto1.wplata(1000.00)

jednoczesne
wykonanie
OK

inny wątek
konto2.wplata(1000.00)

jakiś wątek
konto1.wplata(1000.00)

jednoczesne
wykonanie
WYKLUCZONE

inny wątek
konto1.wplata(1000.00)

Wyścig (*race condition*)

Niepodzielność synchronized — odmiany

Synchronizowana **metoda statyczna**:

```
public static synchronized typ_wynikowy nazwa { ciało }
```

— ciało tej metody może być jednocześnie wykonywane tylko przez **jeden wątek na wszystkie obiekty**; w całym programie tylko jeden wątek.

Wyścig (*race condition*)

Niepodzielność synchronized — odmiany

Synchronizowana **metoda statyczna**:

```
public static synchronized typ_wynikowy nazwa { ciało }
```

— ciało tej metody może być jednocześnie wykonywane tylko przez **jeden wątek na wszystkie obiekty**; w całym programie tylko jeden wątek.

Przykład:

```
class ObslugaDrukarki {  
    public static synchronized void drukuj(String str) {  
        . . . . .  
    }  
}
```

Wyścig (*race condition*)

Niepodzielność synchronized — odmiany

Synchronizowana **metoda statyczna**:

```
public static synchronized typ_wynikowy nazwa { ciało }
```

— ciało tej metody może być jednocześnie wykonywane tylko przez **jeden wątek na wszystkie obiekty**; w całym programie tylko jeden wątek.

Przykład:

```
class ObslugaDrukarki {  
    public static synchronized void drukuj(String str) {  
        . . . . .  
    }  
}
```

— drukować jednocześnie może tylko jeden wątek w programie.

Wyścig (*race condition*)

Niepodzielność synchronized — odmiany

Synchronizowany **blok**:

```
synchronized(obiekt) { fragment_ciała_metody }
```

— z wielu wątków wykonujących jednocześnie ciało tej metody dla tego obiektu **tylko jeden** może znajdować się w tym fragmencie kodu.

Wyścig (*race condition*)

Niepodzielność synchronized — odmiany

Synchronizowany **blok**:

```
synchronized(obiekt) { fragment_ciała_metody }
```

— z wielu wątków wykonujących jednocześnie ciało tej metody dla tego obiektu **tylko jeden** może znajdować się w tym fragmencie kodu.

Przykład:

```
synchronized(this){ x+=w; y-=w; }  
.  
.  
.  
.  
.  
synchronized(this){ x-=w; y+=w; }
```

Wyścig (*race condition*)

Niepodzielność synchronized — odmiany

Synchronizowany **blok**:

```
synchronized(obiekt) { fragment_ciała_metody }
```

— z wielu wątków wykonujących jednocześnie ciało tej metody dla tego obiektu **tylko jeden** może znajdować się w tym fragmencie kodu.

Przykład:

```
synchronized(this){ x+=w; y-=w; }  
.  
.  
.  
.  
.  
synchronized(this){ x-=w; y+=w; }
```

— to zapewnia, że suma $x+y$ będzie stała, nawet jeśli każdy z jednocześnie działających wątków ma inne w .

Wyścig (*race condition*)

Niepodzielność synchronized — odmiany

Synchronizowany **blok**:

```
synchronized(obiekt) { fragment_ciała_metody }
```

— z wielu wątków wykonujących jednocześnie ciało tej metody dla tego obiektu **tylko jeden** może znajdować się w tym fragmencie kodu.

Przykład:

```
synchronized(this){ x+=w; y-=w; }  
.  
.  
.  
.  
.  
synchronized(this){ x-=w; y+=w; }
```

— to zapewnia, że suma $x+y$ będzie stała, nawet jeśli każdy z jednocześnie działających wątków ma inne w .

Należy synchronizować **każde** działanie na danych używanych przez więcej niż jeden wątek.