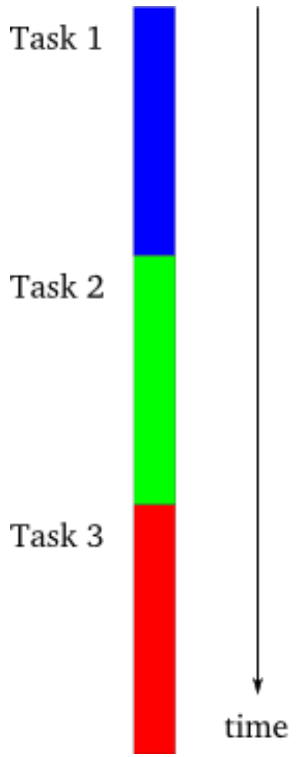
# Technologia Programowania 2017/2018
# Wykład 11
## Model asynchroniczy, Akka

**Jakub Lemiesz**

# Model jednowątkowy

Program ma do wykonania 3 zadania:

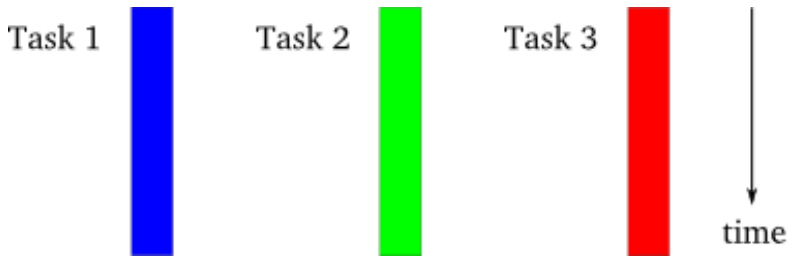
- 1 wątek = 1 zadanie w danej chwili, ustalona kolejność
- kolejne zadania mogą łatwo korzystać z wyników zadań wcześniejszych



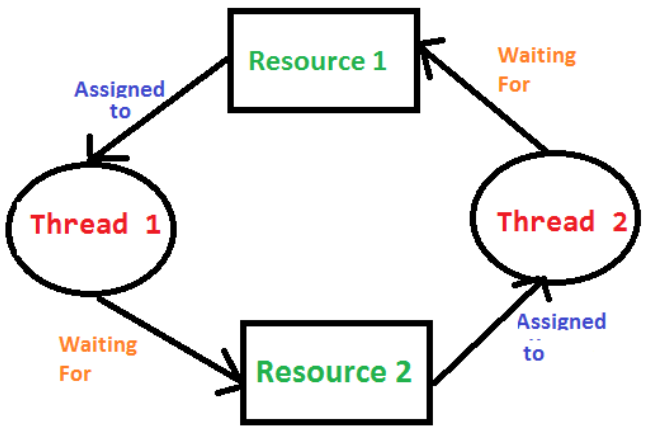
# Model wielowątkowy - przetwarznie równolegle

Każde zadanie w osobnym wątku:

- idealnie: niezależne sekwencje poleceń



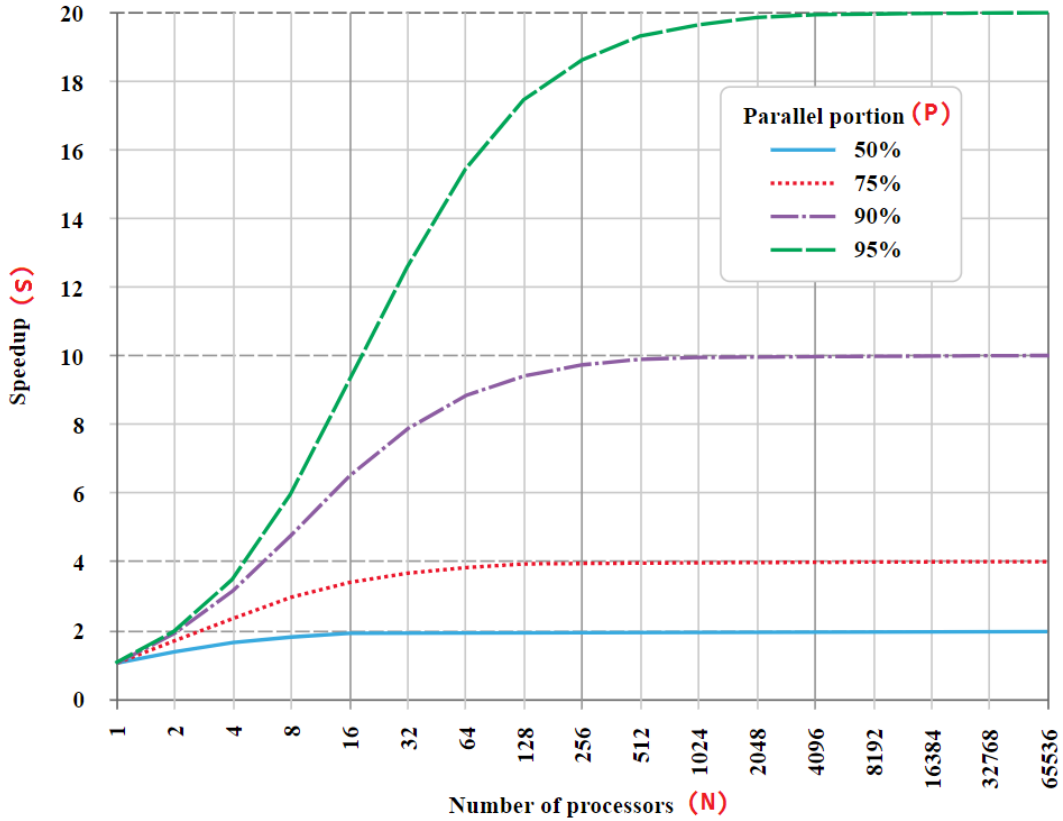

- w praktyce: zadania są powiązane
  (*synchronization, deadlock's, race condition itp.*)



# Przetwarznie równolegle: prawo Amdahla

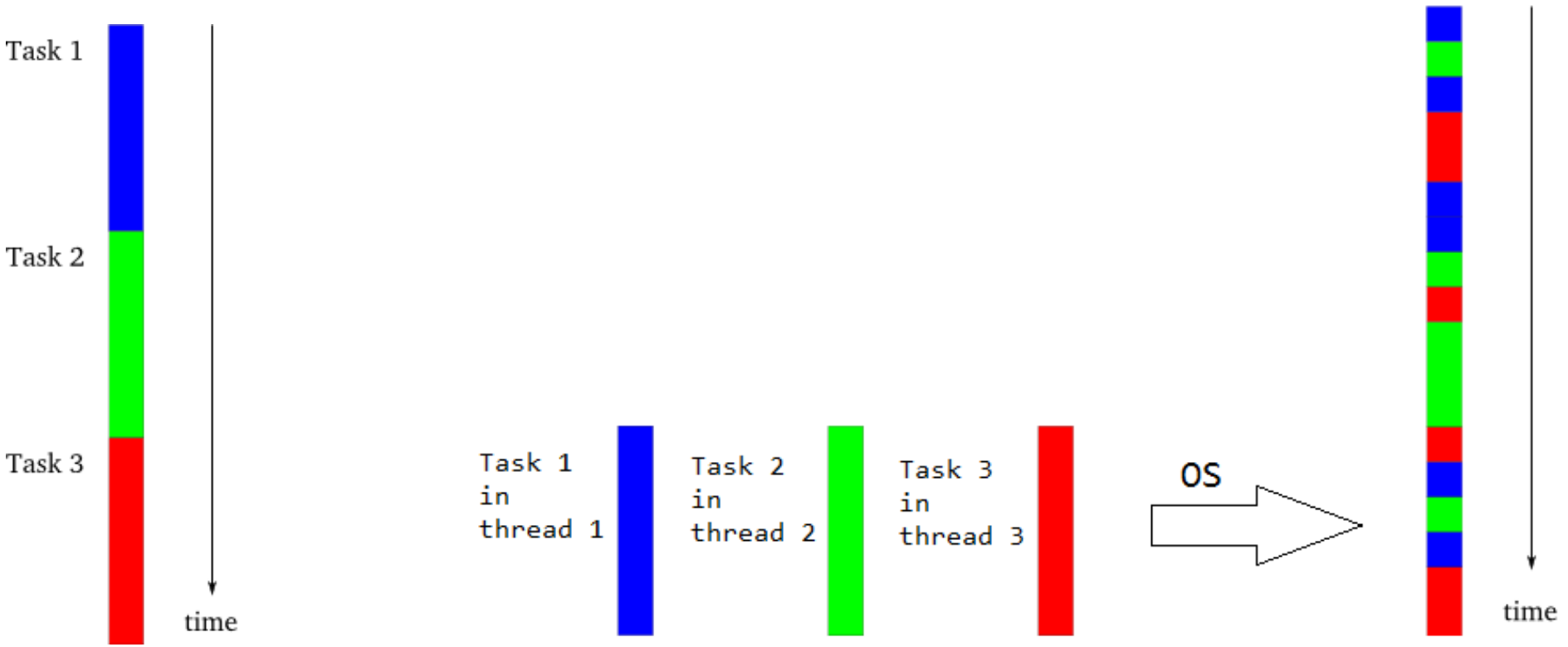
$$S = \frac{1}{(1-P) + \frac{P}{N}},$$

gdzie $S$ - przyspieszenie programu, $P$ - cześć programu, która może być zrównoleglona, $N$ - liczba procesorów



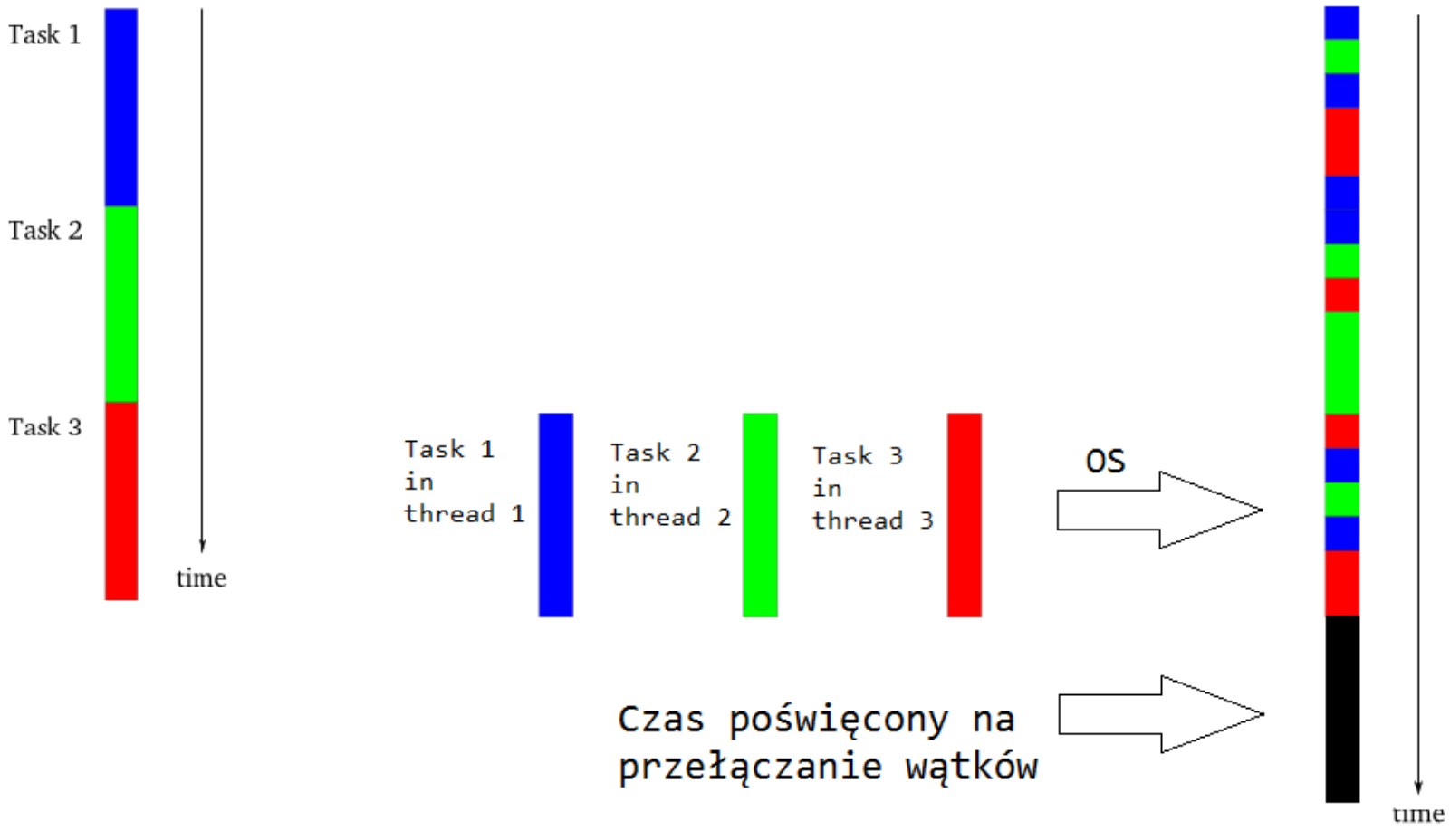
# Model wielowątkowy - przetwarzenie współbieżne

Wiele wątków, ograniczona liczba rdzeni: OS przełącza wątki
(*wiele wątków ale i tak przetwarzanie sekwencyjne*)



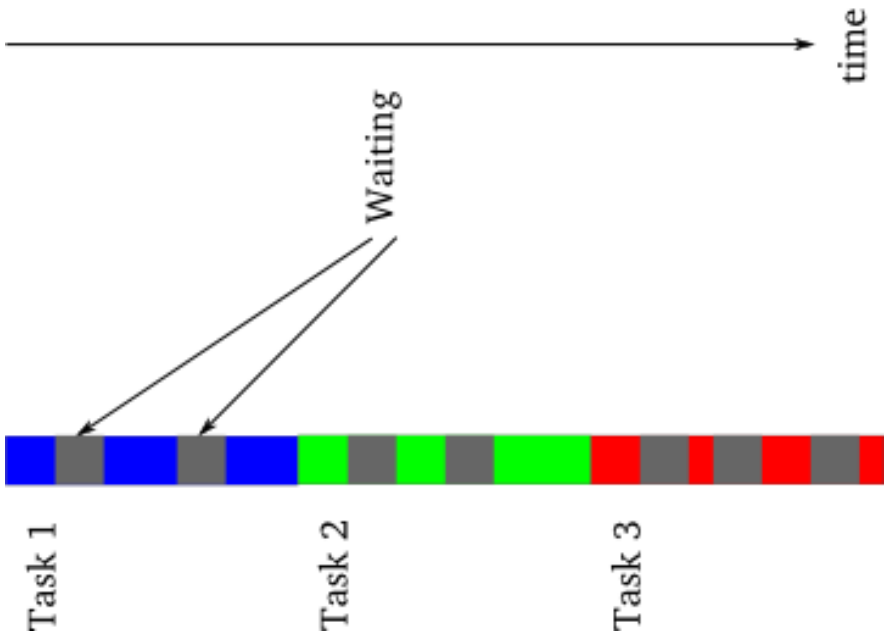
# Model wielowątkowy - przetwarzenie współbieżne

Zmiana kontekstu (*context switch*) – zmiana wątku wiąże się z dodatkowymi narzutami. Po co zatem wiele wątków?



# Model synchroniczny a blokowanie

- Zadania synchroniczne ∼ czekające na odpowiedź
- Np. czekają na zdarzenie, zakończenie operacji I/O (*z dysku, innego urządzenia, sieci ...* )
- Typowe CPU może przetwarzać dane znacznie szybciej niż są one dostarczane z dysku lub sieci
- Możliwe „blokowanie” procesora



# Model asynchroniczny

- Gdy mamy zadanie blokujące, przełączmy się na inne, które może zrobić postęp (*kontrola programisty*)
- Blokowanie jedynie, gdy żadne z zadań nie robi postępu
- Jeśli mamy wiele blokujących zadań model asynchroniczny będzie istotnie szybszy od synchronicznego

# Model asynchroniczny – motywacja

Model asynchroniczny sprawdza się gdy:

1. wiele zadań (*szansa, że jakieś nie będzie zablokowane*)
2. zadania wykonują wiele operacji I/O
3. zadania są w niewielki stopniu zależne od siebie (*nie muszą się nawzajem komunikować, czekać na siebie*)

Warunki te idealnie charakteryzują architekturę klient-serwer:

1. żądania klientów są przeważnie niezależne (*np. web-server*)
2. mamy bardzo wiele operacji I/O - klient wysyła żądanie i dostaje odpowiedź, którą długo przetwarza...

# Podsumowanie

## Przetwarzanie współbieżne vs. równoległe

1. Parallelism - zadań są wykonywane równoległe
2. Concurrency - dwa albo więcej zadań może robić postępy, nawet jeśli nie są wykonywane równolegle

## Model synchroniczny vs. asynchroniczny

1. W modelu synchronicznym czekając na rezultat żądania blokujemy dostęp do procesora
2. W modelu asynchronicznym czekając na rezultat żądania nie blokujemy, powiadomienie gdy rezultat się pojawi
3. Asynchroniczność często jest preferowana, gwarantuje postęp (*lub wrażenie postępu np. w GUI*)

# Synchronizacja wątków

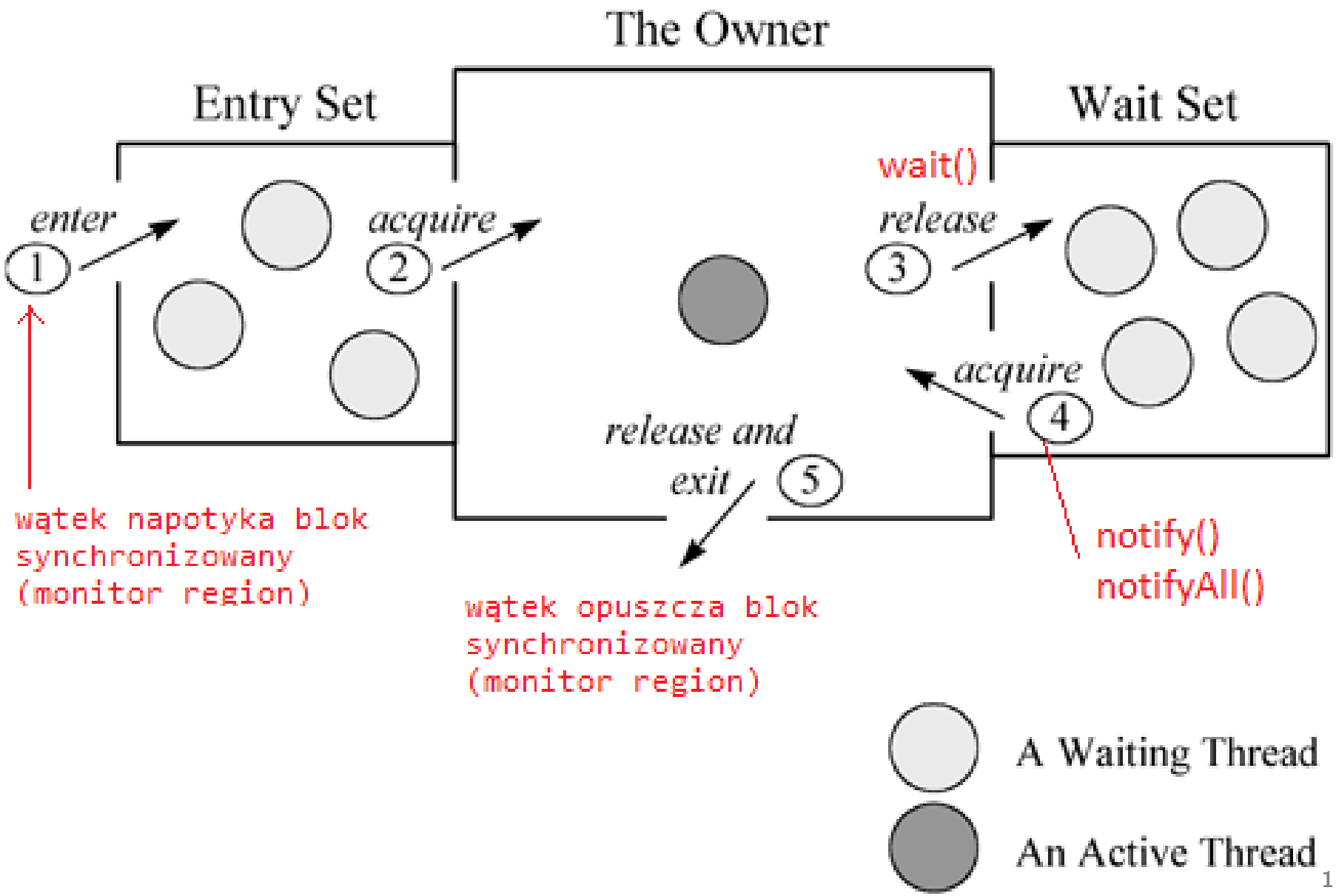
- W aplikacji wielowątkowej do kontroli działania wątków możemy wykorzystać blokady (*locks*) ograniczające dostęp do obiektów i metod
- Kontrola dostępu jest zazwyczaj potrzebna ze względu na gwarancje poprawności komunikacji między wątkami i poprawności działania programu
- W Java mamy słowo kluczowe synchronized, które gwarantuje dostęp na wyłączność jednemu wątkowi

# Synchronizacja wątków

- W JVM każdy obiekt „ma" swój monitor, wątki muszą posiąść monitor by uzyskać dostęp do obiektu
- Wątek prosi o przydzielenie monitora gdy trafi na blok lub metodę synchronizowaną (tzw. *monitor region*)
- *Monitor region* ma referencje obiektu, którego monitor jest potrzebny: *synchronized(someInstance){...}*
- Żadna instrukcja w *monitor region* nie może zostać wykonana dopóki monitor nie zostanie przydzielony
- Pola statyczne? - kiedy JVM ładuje plik klasy tworzy instancję klasy java.lang.Class. Blokowanie pól statycznych klasy to w istocie blokowanie instancji klasy Class

*synchronized(Name.class){...}*

# Monitor – właściciel może być tylko jeden



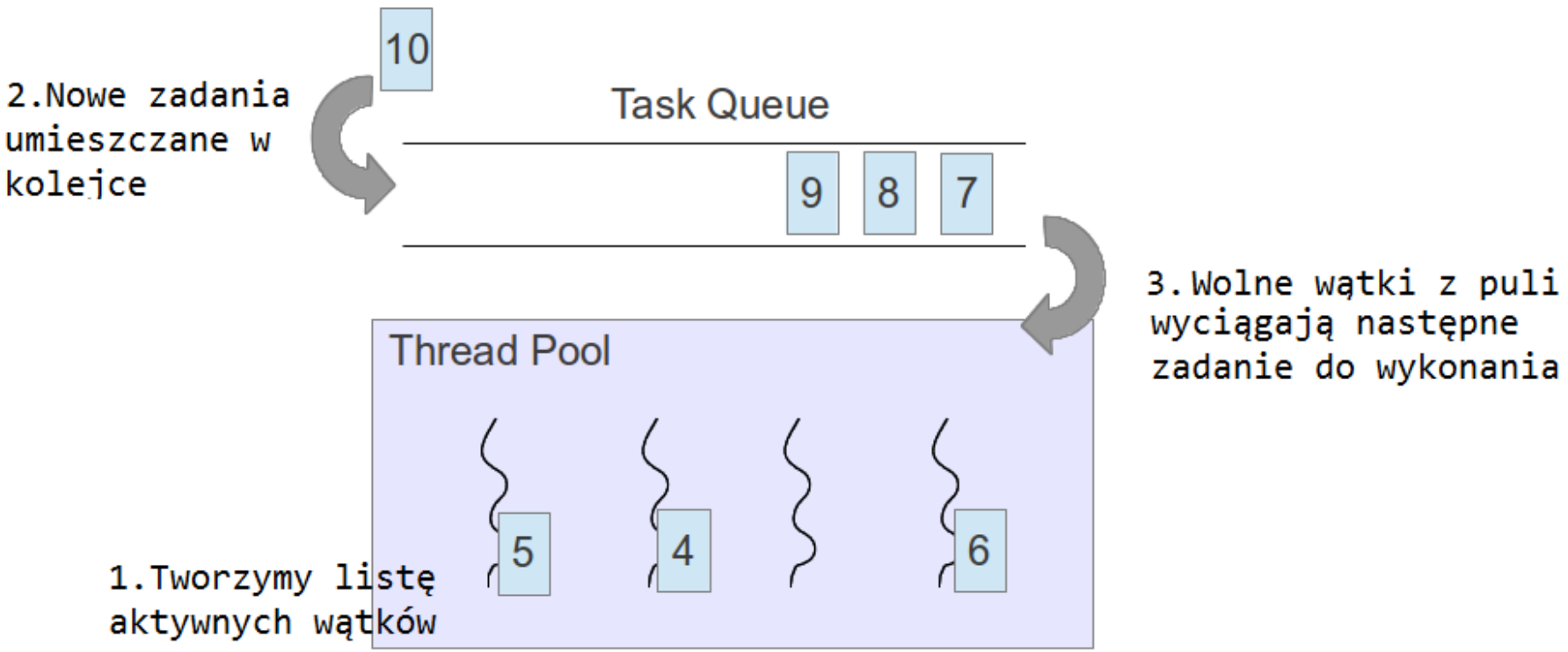
# Monitor - współpraca wątków

## http://www.artima.com/insidejvm/ed2/threadsynchP.html

1. A thread that currently owns the monitor can suspend itself inside the monitor by executing a wait() command.
2. If some owner execute notify(), then the entry set threads will have to compete with one or more threads from the wait set.
3. Java offers two notify commands: notify() and notifyAll(). A notify() command selects one thread arbitrarily from the wait set and marks it for resurrection. A notifyAll() marks all threads currently in the wait set for resurrection.
4. The manner in which a JVM implementation selects the next thread from the wait or entry sets is a decision of individual JVM implementation designers (e.g. FIFO)
5. As a programmer, you must not rely on any particular selection algorithm, at least if you are trying to write a Java program that is platform independent.

# Pula wątków (podstawowa wersja)

Koszty tworzenia i przełączania wielu wątków mogą być duże, często ograniczamy liczbę wątków stosując pulę wątków:

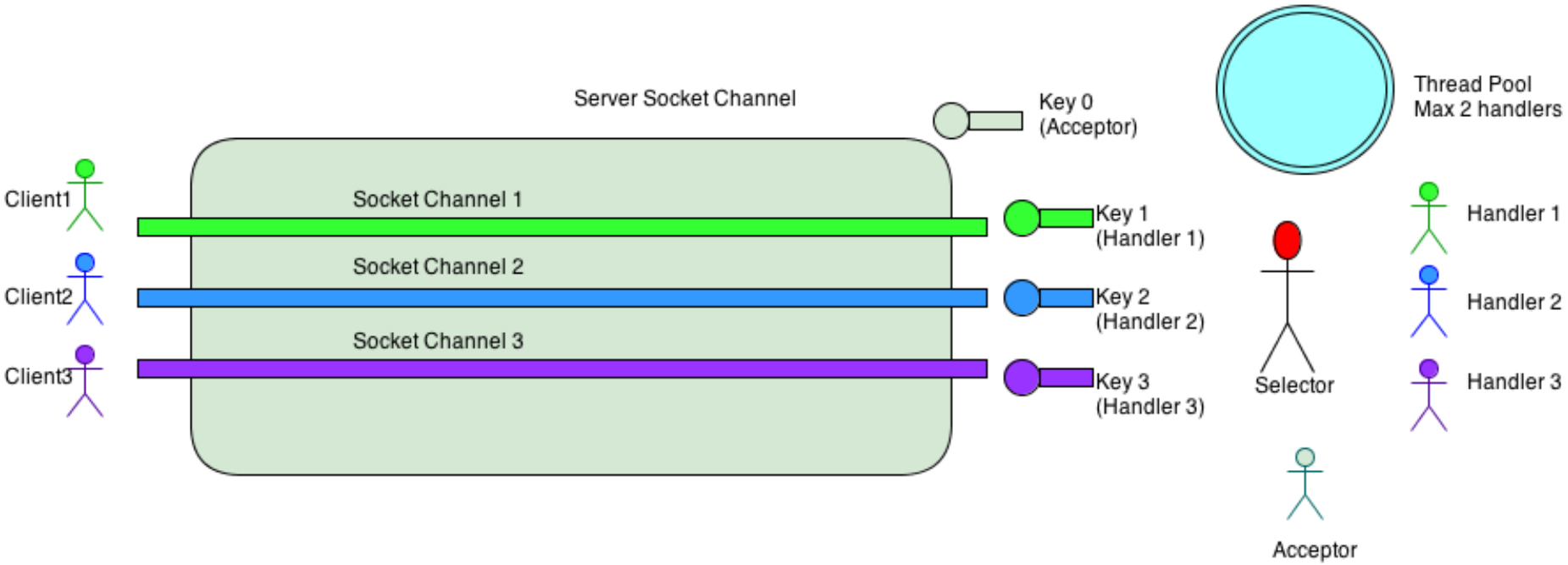


```
public class ThreadPool {
    List<ExecThread> threads = new ArrayList<ExecThread>();

    Queue tasks = ... //dostęp synchronizowany
                      //np.ConcurrentLinkedQueue

    public synchronized void  execute(Runnable task)
    {       //nowe zadanie
            tasks.enqueue(task);
    }

    //inicjalizacja
    for(int i=0; i<noOfThreads; i++)
        threads.add(new ExecThread(tasks));

    for(ExecThread thread : threads)
        thread.start();
```

```
public class ExecThread extends Thread {

    Queue tasks = null;

    ExecThread(SynchrQueue tasks) {this.tasks = tasks;}

    public void run(){
      while(!isStopped()){
       try{
         Runnable runnable = (Runnable) tasks.dequeue();
         runnable.run();
        }
        catch(Exception e)...
```

*Implementacja puli wątków dostępna w pakiecie java.util.concurrent*

# Architektura klient-serwer

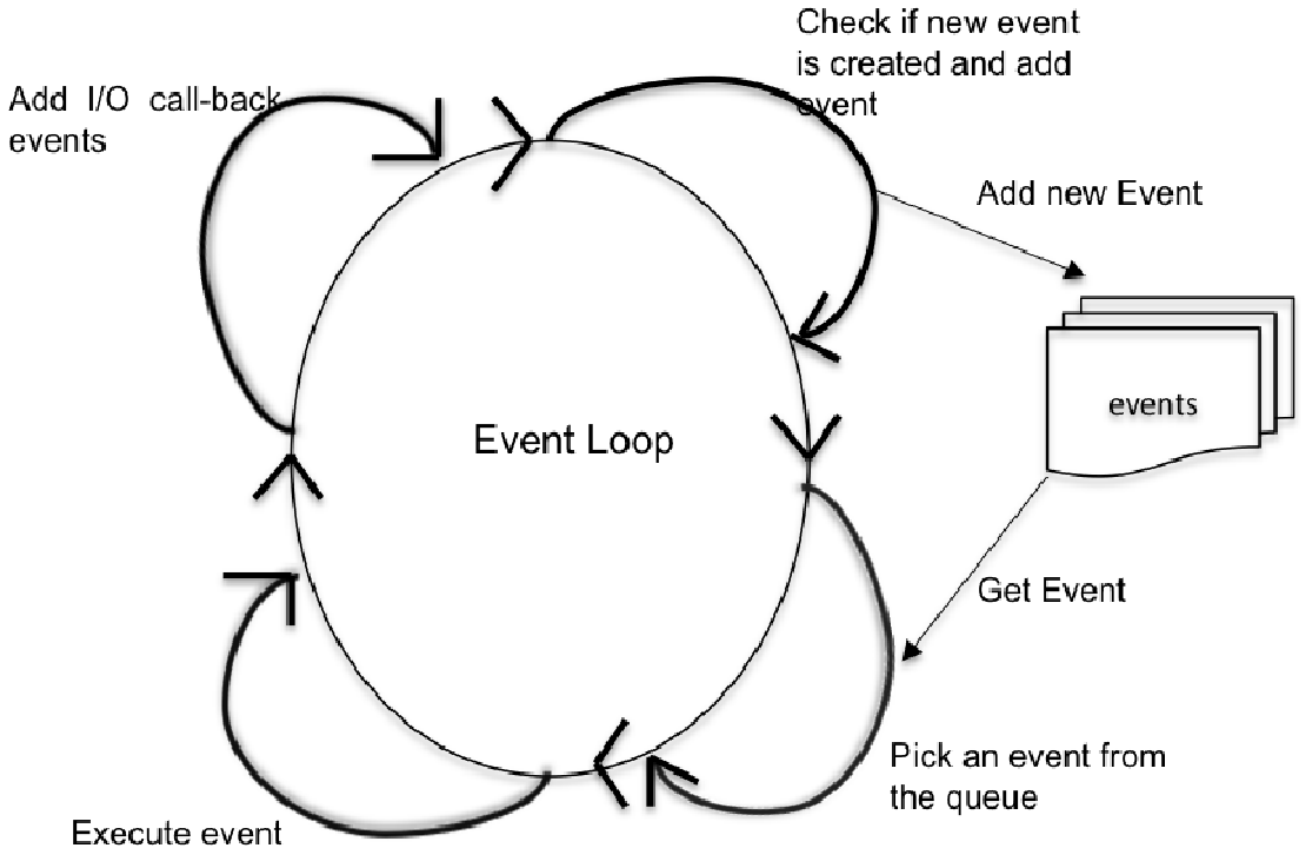



Jak radzić sobie z dużą liczbą połączeń w sposób możliwie „nieblokujący”? C10K Problem - 1999, C10M - 2010

1. Jak zorganizować pracę Selectora?
2. Agent (ang. handler) dla klienta czy dla typu zadania?
3. Osobny wątek dla każdego klienta? Thread Pool?

# Pomysł: model asynchroniczny, event-driven programming

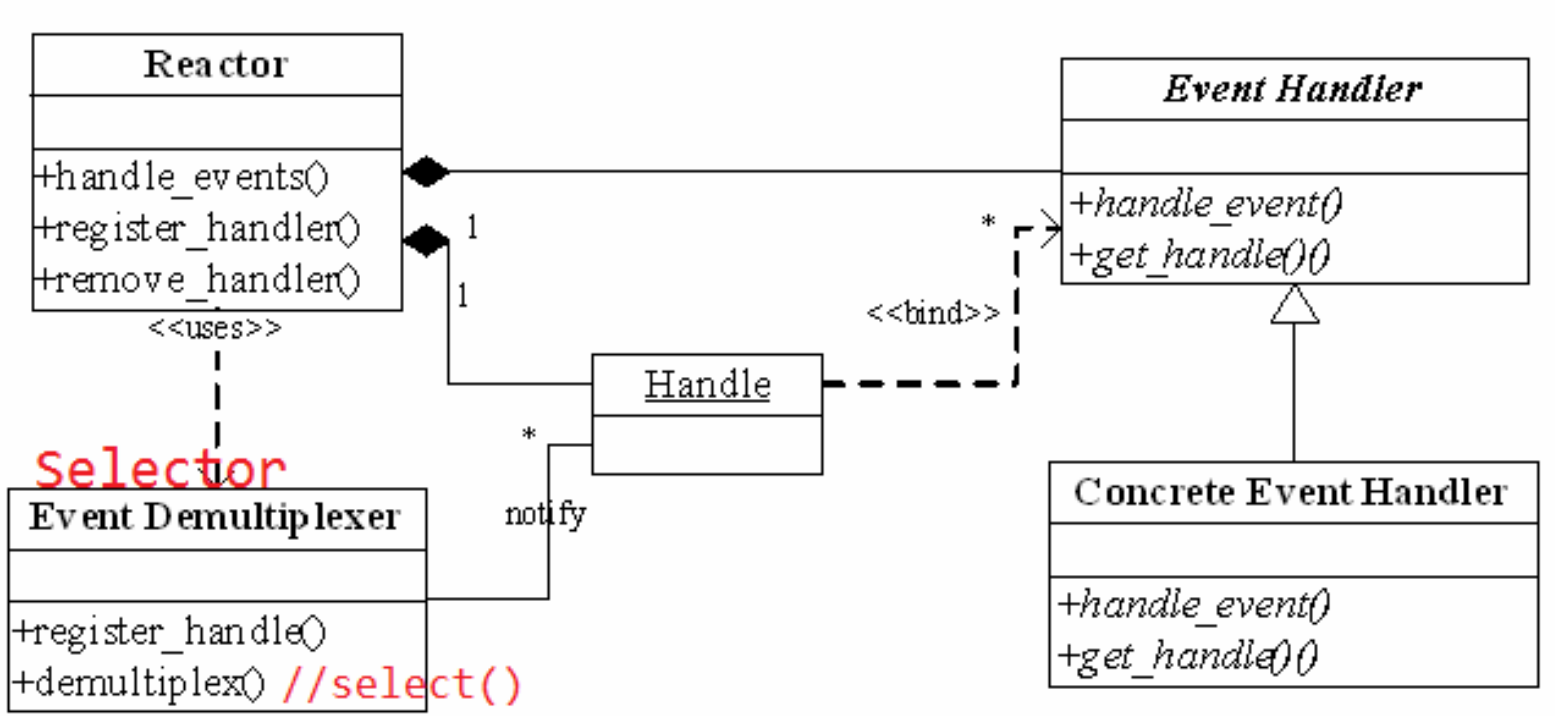
```
while ((e = GetEvent()) != 0)
{
  HandleEvent(e);
}
```

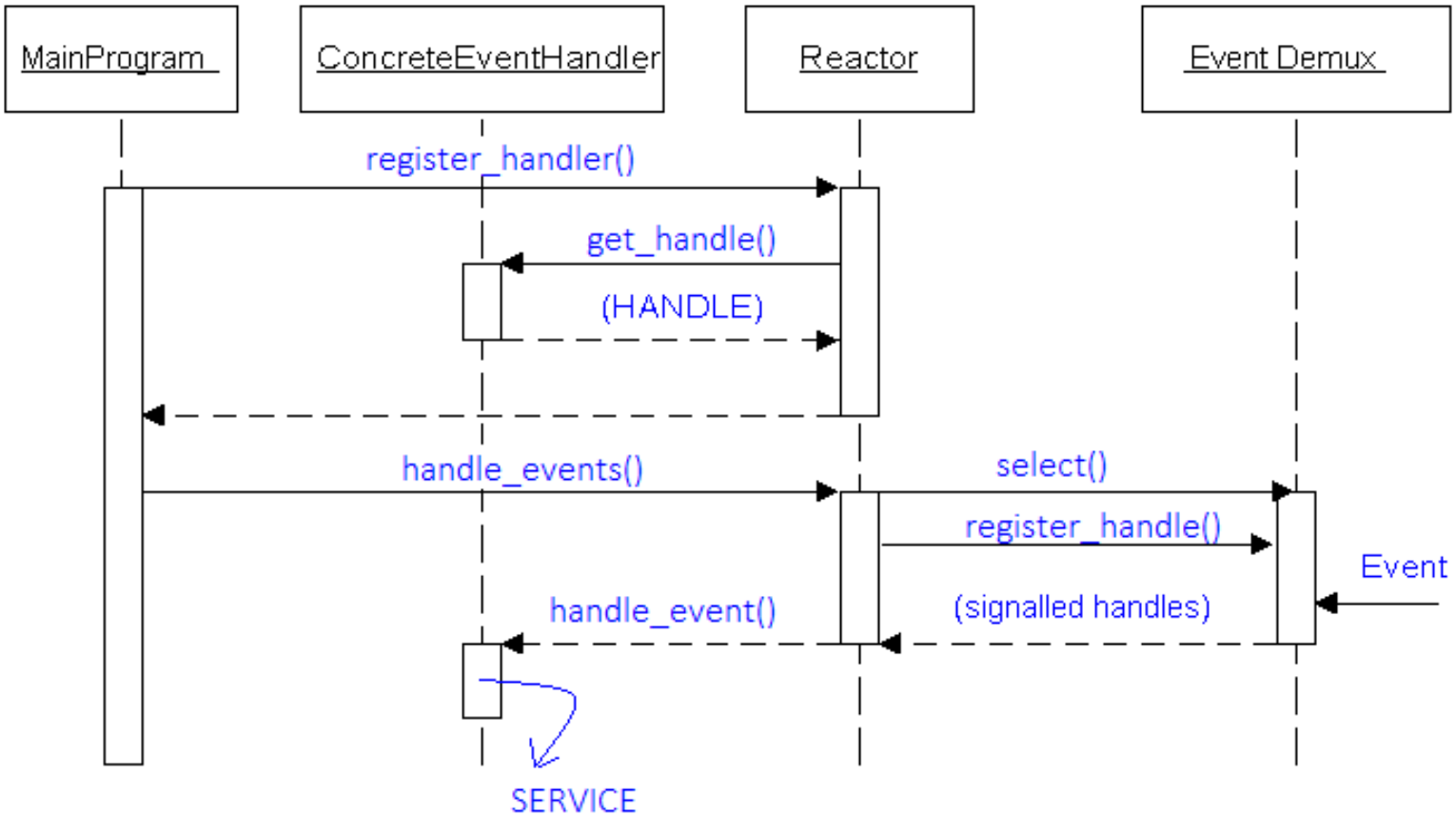


# Wzorzec Reactor: diagram klas

EventHandler: interfejs, konkretne *handler'y* służą do obsługi zdarzeń określonego typu. Są rejestrowane i wywoływane kiedy pojawi się odpowiednie zdarzenie (≈ *callbacks, handler==agent*)

Handle: "uchwyt" do źródła zdarzenia (*np. gniazdo, widget, plik*)

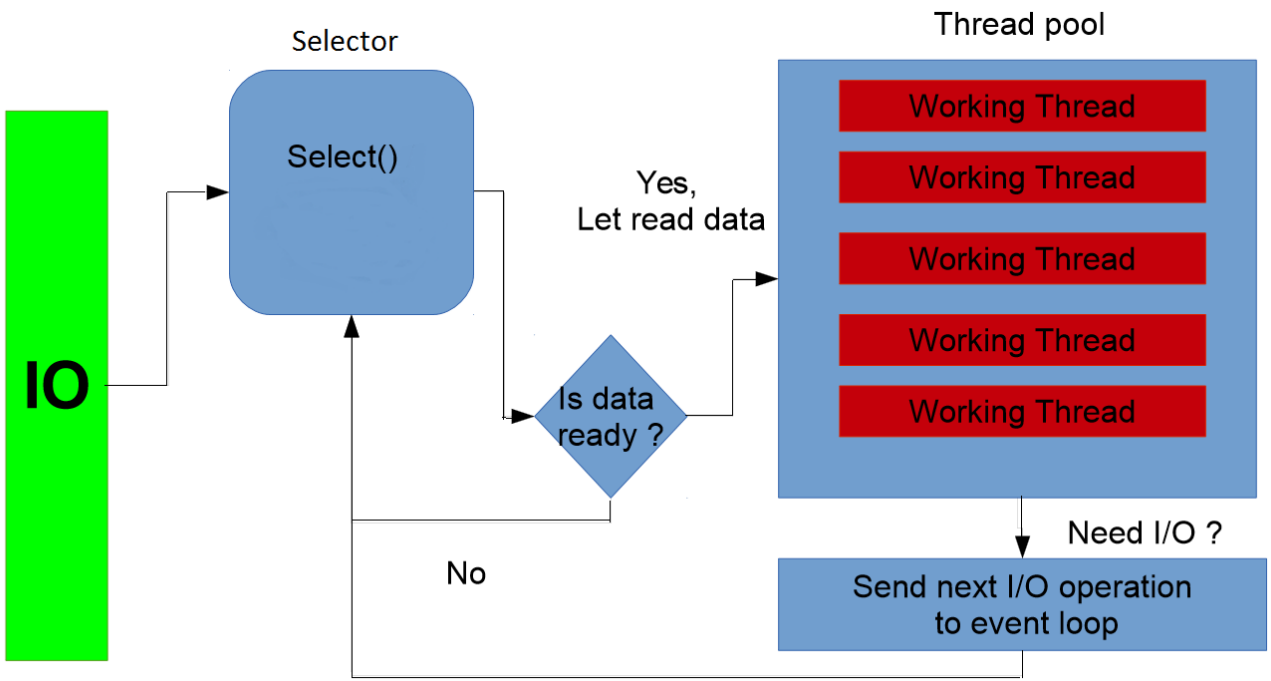





1. Rejestrujemy EventHandler z uchwytem (handle), np. gniazdem
2. Reactor pobiera uchwyty od zarejestrowanych EventHandlers
3. Reactor uruchamia Selectora (*EventDemux*) i przekazuje uchwyty
4. Selector informuje Reactor gdy któreś uchwyty gotowe (*np. gniazda*)
5. Reactor używa otrzymany od Selecotra podzbiór uchwytów do wywołania metod handle_event() w odpowiednich EventHadnler'ach
6. handle_event() gwarantuje odpowiednią obsługę (*np. w osobnym wątku z puli*), która może odpowiedzieć klientowi mając dostęp do uchwytu

# Wzorzec Reactor: przykład

1. Jedna pętla zdarz dla nadchodzących zdarzeń I/O (*monitorujemy zdarzenia w metodzie select*)
2. Zdarzenia zamieniane na zadania dla puli wątków
3. Gdy potrzebne nowe dane powrót do pętli zdarzeń




Źródło diagramu

# Więcej o wzorcu Reactor

Sensowna książka (*źródło powyższych diagramów*)
"Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects" by Schmidt et al., strona 159

Implementacje wzorca Reactor:

1. Minimalistyczna przykładowa implementacja:
   http://assorted.sourceforge.net/java-reactor/
2. Staranna implementacja z dobrym opisem:
   http://jeewanthad.blogspot.com/2013/02/reactor-pattern-explained-part-1.html
3. Reactor – implementacja we freameworku Spring
   https://spring.io/guides/gs/messaging-reactor/

# Reactor: podsumowanie

1. Korzyści: rozwiązanie efektywne, łatwa obsługa (*w praktyce programista definiuje event handler'y, resztę robią biblioteki*)
2. Wada: może być trudne do zdebugowania (*więcej: książka z poprzedniego slajdu*)
3. Inne, ogólniejsze rozwiązanie: system Akka

**Reactor vs. Akka - dyskusja**

http://stackoverflow.com/questions/16595393/akka-or-reactor

# Czym jest Akka?

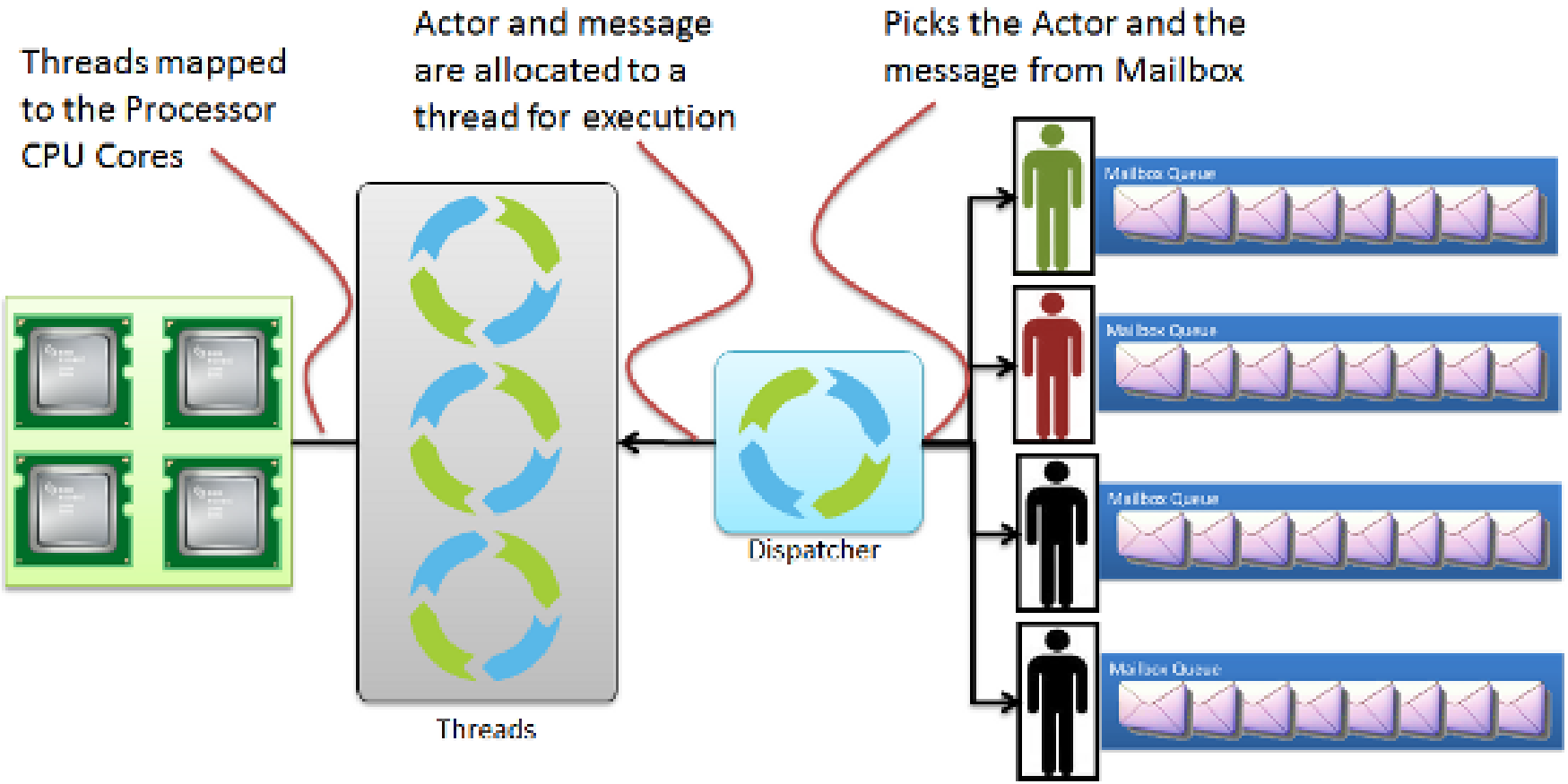
## Autrzy Akka

*„We think that writing correct concurrent and scalable applications is [now] too hard (...) Any system that have the need for high throughput and low latency is a good candidate for using Akka."*

Akka opiera się na systemie współpracujących aktorów:

- Idea: opakowujemy nasze klasy Java w "aktorów"
- Aktorzy porozumiewają się wyłącznie za pomocą "maili", żadne zmienne informacje nie są współdzielone (*wysokopoziomowa abstrakcja dla progr. współbieżnego* )
- W istocie model asynchroniczny, sterowany zdarzeniami

# Akka w dużym skrócie




```
public class Dispatcher extends MessageDispatcher {

//The event-based Dispatcher binds a set of Actors
//to a thread pool backed up by a BlockingQueue.
```

# Akka - system hierarechiczny

- Aktorzy tworzą system hierarechiczny
- Aktor może np. podzielić swoje zadanie na podzadania i przydzielić je kontrolowanym przez siebie aktorom

```
// Using the ActorSystem will create top-level actors,
// supervised by the system provided guardian actor,
// while using an actor's context will create a child actor.
// ActorSystem is a heavy object: create only one per app.

final ActorSystem system = ActorSystem.create("MySystem");

final ActorRef parent =
  system.actorOf(Props.create(Parent.class));

class Parent extends UntypedActor {
    final ActorRef child =
       getContext().actorOf(Props.create(Child.class));
```

# Czym jest aktor?

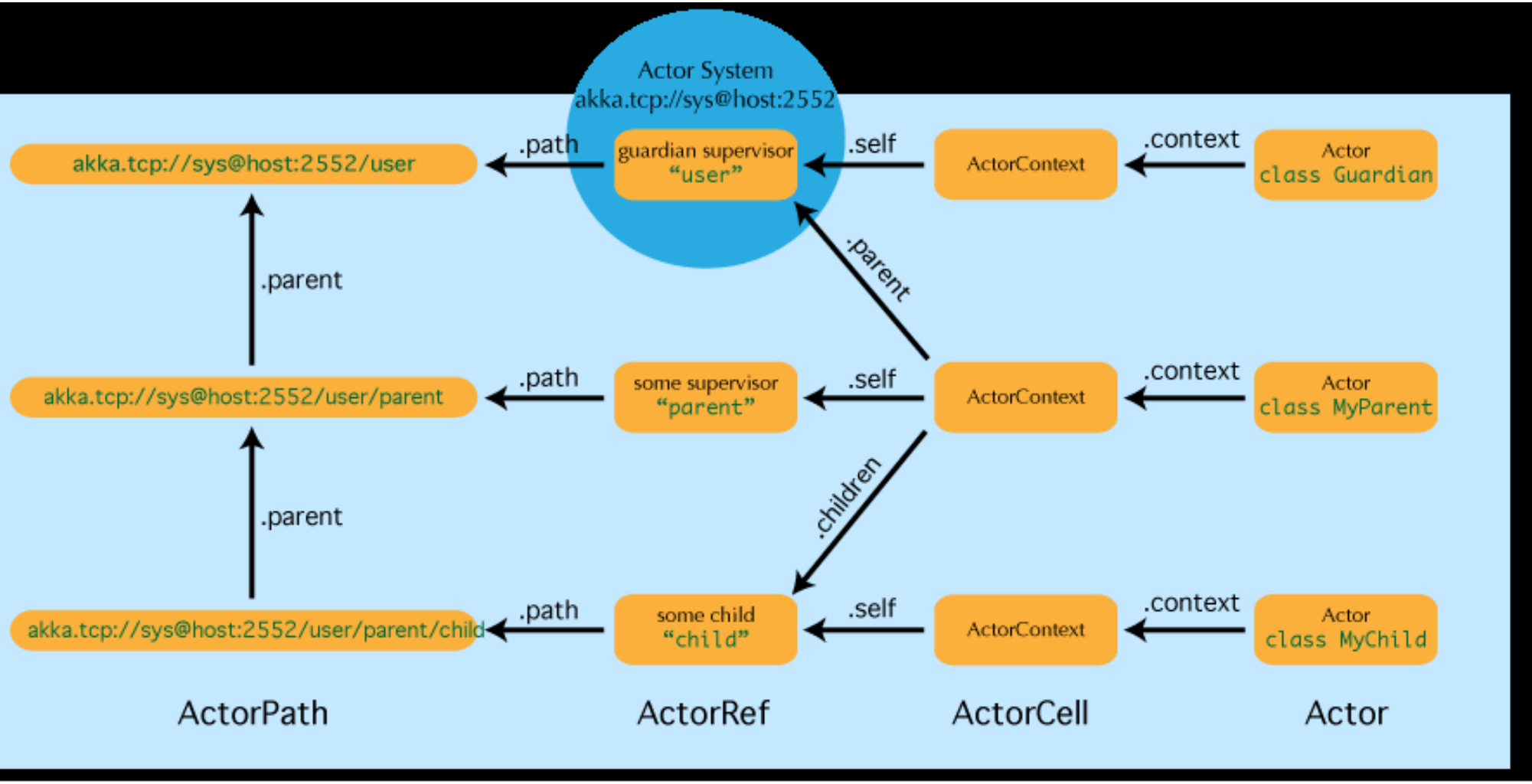
- Aktor to w istocie kontener, który zawiera:
  State (*prywatne pola*),
  Behavior (*onRecive + prywatne metody*),
  Mailbox, Childrens oraz Supervisor Strategy
- Aktorzy komunikują się wyłącznie poprzez wymianę wiadomości umieszczanych w Mailbox odbiorcy
- UntypedActor vs. TypedActor

```
public class MyUntypedActor extends UntypedActor {

public void onReceive(Object message) {
  if (message instanceof String) {
     ...
     getSender().tell(message, getSelf());
  }
  else ...
```

# Czym jest referencja do aktora?

1. Referencje do aktorów ActorRef służą głównie do wspierania komunikacji między aktorami
2. Każdy aktor ma dostęp do swojej referencji przez pole self oraz metodę getSelf()
3. Przy wysyłaniu komuniaktu referencja do nadawcy jest autmatycznie dołączana i można ją odczytać przez pole sender oraz metodę getSender()
4. Aktorzy mają również referencje do rodzica i do potomków

# Czym jest referencja do aktora?



# Remote Actors

Łatwe dodawanie aktorów działających na innych maszynach:

```
Address addr = new Address("akka", "remotesys", "host", 1234

ActorRef routerRemote = system.actorOf(
  new Props(Child.class).withRouter(
       new RemoteRouterConfig( new RandomRouter(5), addr )
     )
    );

//dalej komunikacja przez tell() i onRecive()
```

# Supervisor Strategy

- Aktor tworzy i nadzoruje swoich potomków
- Aktor może przekazywać swoje zadania potomkom (*np. może też podzielić zadanie na mniejsze zadania*)
- Router jest aktorem, który przekazuje nadchodzące wiadomości do swoich potomków w określony sposób: RoundRobinRouter, RandomRouter, SmallestMailboxRouter, ...

```
int nrOfInst = 5;
 ActorRef router = system.actorOf(
  new Props(Child.class).withRouter(new RandomRouter(nrOfInst)))

  router.tell(message, getSelf());
```

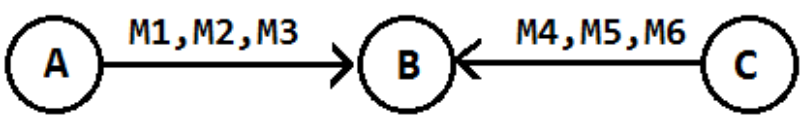
- Konfiguracja w pliku application.conf (*np. na ilu wątkach*)

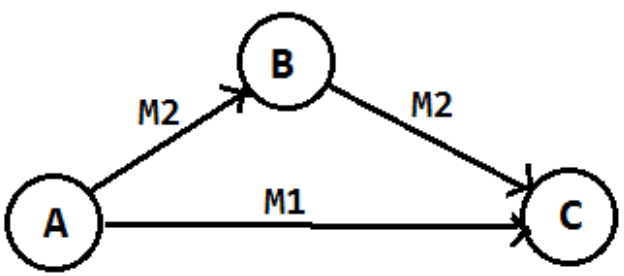
```
ActorRef router = getContext().actorOf(
      Props.create(PrintlnActor.class).withRouter(
                                  new RoundRobinRouter(5)));

for (int i = 1; i <= 10; i++)
       router.tell(i, getSelf());
```

..> Received message '1' in actor *a*
..> Received message '2' in actor *b*
..> Received message '3' in actor *c*
..> Received message '6' in actor *a*
..> Received message '4' in actor *d*
..> Received message '8' in actor *c*
..> Received message '5' in actor *e*
..> Received message '9' in actor *d*
..> Received message '10' in actor *e*
..> Received message '7' in actor *b*

# Mailbox - kolejność dostarczania

- $A1$ wysyła $M1, M2, M3$ do $A2$
  $A3$ wysyła $M4, M5, M6$ do $A2$
  1. jeśli $M1$ będzie dostarczone to dojdzie przed $M2$ i $M3$
  2. jeśli $M2$ będzie dostarczone to dojdzie przed $M3$
  3. jeśli $M4$ będzie dostarczone to dojdzie przed $M5$ i $M6$
  4. jeśli $M5$ będzie dostarczone to dojdzie przed$M6$
  5. kolejność między $M1, M2, M3$ a $M4, M5, M6$ nieustalona
- Zasada nie jest przechodnia:
  1. $A$ wysyła $M1$ do $C$
  2. $A$ wysyła $M2$ do $B$, $B$ przesyła $M2$ do $C$
  3. $C$ może otrzymać $M1$ i $M2$ w dowolnej kolejności



# Mailbox - gwarancje dostarczenia

- Akka nie daje żadnych gwarancji, że wysłana wiadomość dojdzie do odbiorcy, programista musi o to zadbać sam w wyższej warstwie (*fire-and-forget*). Dlaczego?
- Problem z ustaleniem, czy wystarcz sprawdzić, że wiadomość
  - doszła do serwera odbiorcy?
  - doszła do Mailbox odbiorcy?
  - została odebrana przez odbiorcę?
  - została poprawnie przetworzona przez odbiorcę?
- `tell()` preferowane, ale jest również `ask()`

```
String result = (String) Await.result(ask(Actor, message, timeou
```

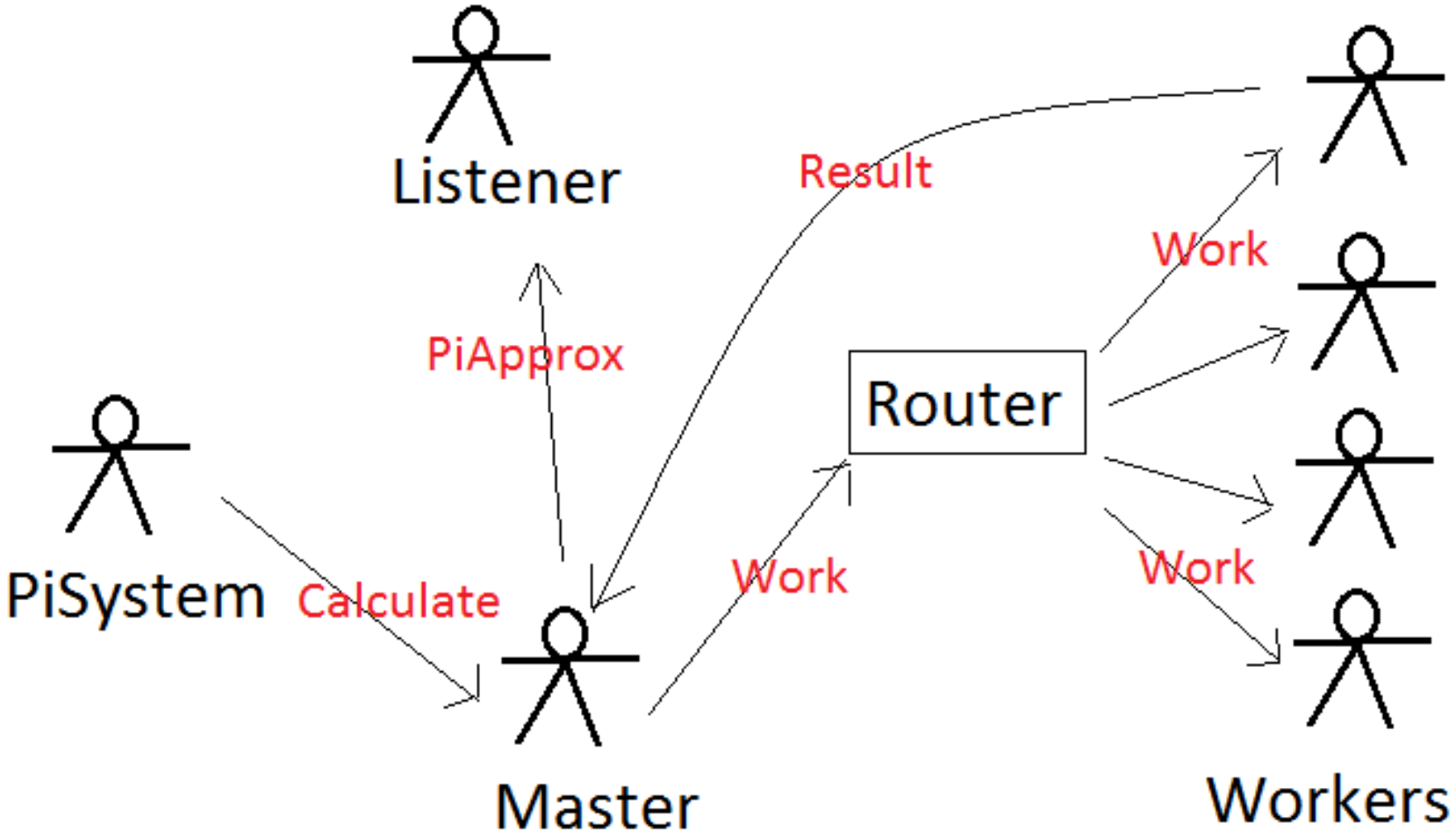
# Przykładowa aplikacja – przybliżanie liczby pi

- Wzór Leibniz'a:

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4}$$

- Im więcej wyrazów uwzględnimy tym lepsze przybliżenie $\pi = 3,141592653589793...$
- Zadanie łatwo jest przyspieszyć, możemy podzielić sumowanie na kilka niezależnych zadań, wykonać te zadania równolegle i połączyć wyniki
- Plik pom.xml dla aplikacji Pi, uruchamiamy:

```
mvn compile exec:java -Dexec.mainClass="$package$.Pi"
```




Jaka liczba Workerów w PiSystem jest optymalna?

# Podsumowując, co robimy?

1. Definiujemy niezmienne (ang. immutable) klasy wiadomości - wszystkie pola sa "final".
2. Definiujemy aktorów (extends UntypedActor) oraz przesłaniamy metodę onRecive(...), która określa akcje w zależności od rodzaju wiadomości
3. Określamy hierarchie aktorów (kto kogo tworzy)
4. Aktorzy komunikują się wysyłając wiadomości: polecenie tell() oraz metoda OnRecive() (*nie powinno być innych publicznych metod*)

Akka ma bardzo dobrą dokumentację
http://doc.akka.io/docs/akka/2.4/java.html
(*uwaga na wersję*)