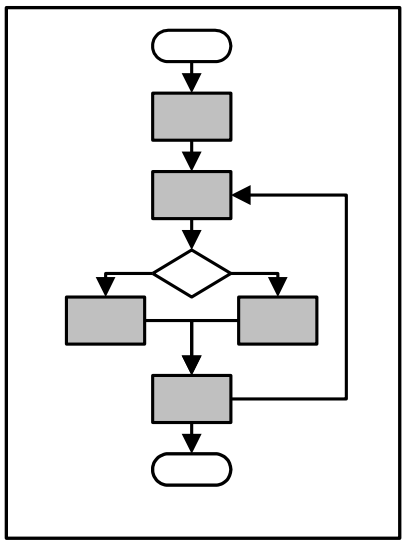
Wątki

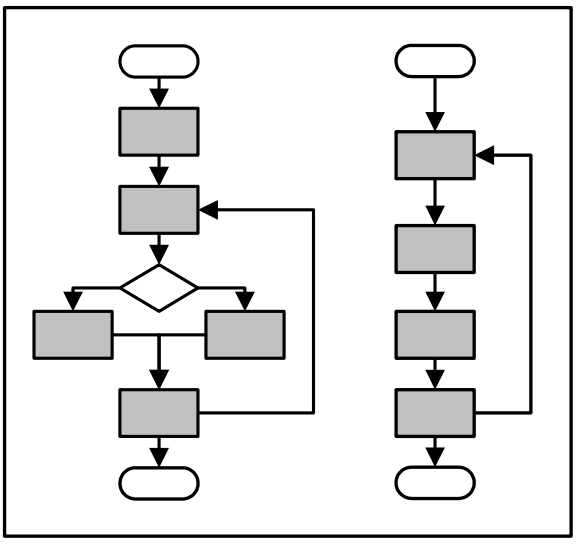
Wątki w języku Java są mechanizmem umożliwiającym pisanie programów współbieżnych, czyli programów, które równocześnie realizują pewną liczbę sekwencyjnych podprogramów.

Poza możliwością uruchamiania wątków Java dostarcza mechanizmów synchronizacji, wzajemnego wykluczania i komunikacji pomiędzy wątkami.

Programy sekwencyjne i programy współbieżne

Program *sekwencyjny* to program, który wykonuje jeden ciąg instrukcji. Ciąg ten może zawierać rozgałęzienia (instrukcje warunkowe) i pętle, ale w każdym momencie wykonania programu możemy wskazać aktualnie wykonywaną instrukcję identyfikowaną przez adres rozkazu załadowany do rejestru *IP* (*instruction pointer*) procesora.





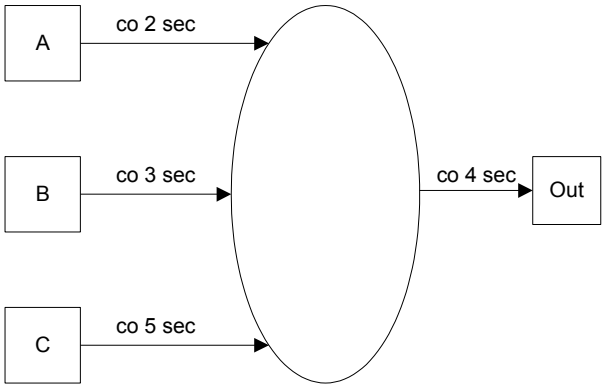
Program *współbieżny* (ang. *concurrent*) to program, który jest zaimplementowany w postaci zbioru sekwencyjnych podprogramów. Środowisko, w którym wykonywany jest program współbieżny zapewnia ich równoczesne wykonanie. Może być to realizowane w różny sposób. W systemie jednoprocesorowym następuje podział czasu procesora pomiędzy sekwencyjne podprogramy; w środowisku wieloprocesorowym podprogramy mogą być uruchamiane na różnych procesorach.

Dlaczego buduje się programy współbieżne

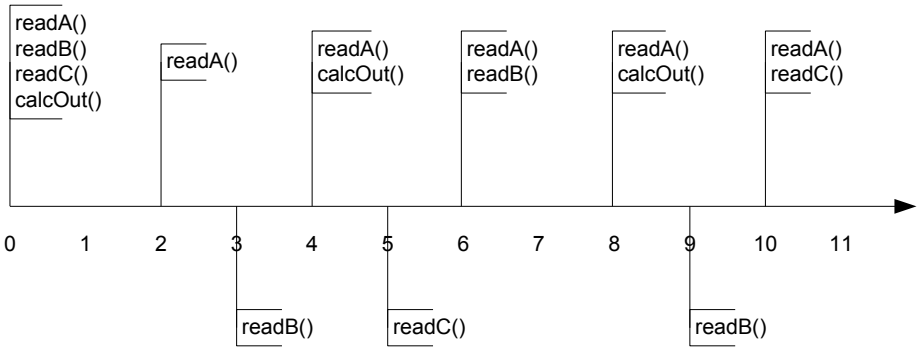
Pierwszą motywacją jest zastosowanie w sterowaniu.

Przykład

System okresowo odczytuje wartości z czujników A (co 2 sec.) B (co 3 sec) i C (co 5 sec) i oblicza wartość wyjściową (co 4 sec).



Na podstawie tych wymagań można zbudować program sekwencyjny realizujący opisane wymagania ustalając dla niego harmonogram wykonywanych operacji (ang. *schedule*).



Wynikowy program sekwencyjny będzie wiec realizował ciąg operacji na przemian z okresami bezczynności:

```
readA();readB();readC();calcOut();
sleep(2000); // uśpienie programu 2 sec
readA();
sleep(1000);
readB();
sleep(1000);
readA();calcOut();
// itd
```

Ten sam system może być zaimplementowany w postaci programu współbieżnego złożonego z 4 równocześnie wykonywanych podprogramów sekwencyjnych.

```
P1                P2                P3                P4
for(;;){          for(;;){          for(;;){          for(;;){
  readA();          readB();          readC();          calcOutA();
  sleep(2000);      sleep(3000);      sleep(5000);      sleep(4000);
}                 }                 }                 }
```

Programista nie zajmuje się dłużej problemem, w jakim czasie uruchomić poszczególne funkcje. Odpowiedzialność za harmonogram aktywacji podprogramów spoczywa zazwyczaj na specjalnie w tym celu skonstruowanym składniku środowiska wykonania programu -- planiście (ang. *scheduler*). Najczęściej planista jest częścią systemu operacyjnego.

Systemy reaktywne. Specjalną klasą systemów, które praktycznie zawsze są implementowane w sposób współbieżny są systemy reaktywne. Ich zadaniem jest reakcja na asynchronicznie pojawiające się zdarzenia wejściowe. Reakcja polega na zmianie stanu, wykonaniu pewnych operacji, obliczeniu wartości wyjściowych.

Zdarzeniem wejściowym może być pojawienie się lub zmiana danych na wejściu, przekroczenie pewnego progu wartości, akcja użytkownika, itd.

Zazwyczaj z każdym wejściem systemu związana jest wykonywana w pętli sekwencja instrukcji: oczekuj na pojawienie się zdarzenia, obsłuż zdarzenie.

```
for(;;){              for(;;){              for(;;){
   waitForA();           waitForB();           waitForC();
   handleA();            handleB();            handleC();
}                     }                     }
```

Reaktywny interfejs użytkownika. (ang. *responsive*) Tradycyjne programy sekwencyjne implementujące graficzny interfejs użytkownika działają według następującego schematu:

- czekaj na komendę użytkownika
- wykonaj polecenie

Taki tryb działania jest zadowalający, jeżeli czas wykonania poleceń jest krótki. W przypadku złożonego przetwarzania użytkownik oczekuje możliwości odczytu informacji o postępie operacji oraz możliwości jej przerwania. Nawet aplikacje biurowe zawierają operacje implementowane współbieżnie, np.: wydruk, dzielenie na strony w tle.

Współpraca z wolnym medium komunikacyjnym. Coraz większa liczba współczesnych aplikacji realizuje funkcje komunikacji sieciowej. O ile moc obliczeniowa serwerów i stacji roboczych stale rośnie, medium komunikacyjne ma zawsze ograniczoną przepustowość.

Implementacja współbieżna programów komunikacyjnych zwiększa komfort ich użytkowników: możliwe jest np.: równoczesne ładowanie większej liczby plików, uaktualnianie na bieżąco interfejsu użytkownika, itd.

Szczególną grupę programów stanowią serwery (www, ftp, mail). Serwer odbiera zlecenie i dane od klienta, uruchamia lokalne programy (np.: skrypty CGI) i przesyła dane z powrotem do klienta. Gdyby serwer był ograniczony do obsługi tylko jednego klienta, wówczas jego wydajność byłaby bardzo niska. Zazwyczaj serwer może obsłużyć większą liczbę zleceń jednocześnie, np.: kilkaset uruchamiając dla każdego zlecenia pojedynczy podprogram (wątek).

Mechanizmy współbieżne systemów operacyjnych

Współczesne systemy operacyjne umożliwiają równoczesne uruchamianie wielu programów .

Program, rozumiany jako zbiór rozkazów procesora i danych zapisany w postaci ciągu bajtów na dysku lub w stałej pamięci, w momencie uruchomienia staje się *procesem* (*zadaniem*).

System operacyjny wiąże z każdym procesem zbiór danych nazywany *kontekstem*. Obejmuje on:

- stan procesu (nowy, gotowy, aktywny, zawieszony, oczekujący)
- licznik rozkazów – adres następnego rozkazu do wykonania
- rejestry procesora
- informacje sterujące przydziałem procesora (identyfikator procesu, priorytet, liczniki czasu)
- informacje o przydzielonej pamięci (stosu, sterty, wartości graniczne bloków pamięci)
- informacje o stanie wejścia/wyjścia (otwarte pliki, urządzenia przydzielone do procesu, otwarte połączenia sieciowe)
- informacje o prawach dostępu przypisanych procesowi
- informacje służące do raportowania (wykorzystany czas procesora, informacje o właścicielu procesu)

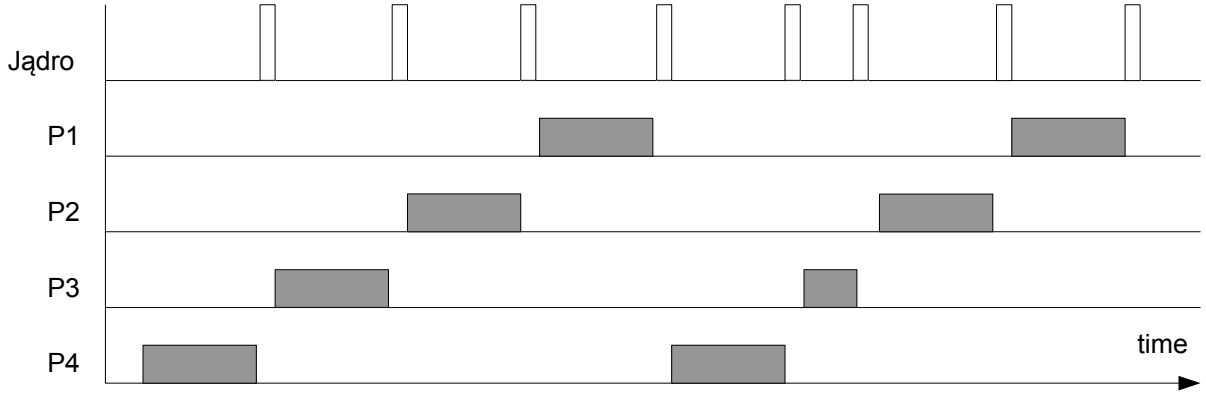
System operacyjny zarządza uruchomionymi procesami przydzielając im czas procesora. Bodźcem do zatrzymania bieżącego procesu i uaktywnienia innego mogą być rozmaite zdarzenia: przekroczenie przydzielonego czasu, gotowość urządzenia wejścia-wyjścia do dostarczenia lub przyjęcia kolejnej porcji danych, akcja użytkownika.

System operacyjny może realizować różne polityki przydziału procesora opierające się na równomiernym podziale czasu pomiędzy procesy i/lub priorytetach.

Z punktu widzenia użytkownika działanie systemu operacyjnego jest zadawalające, jeżeli wykonanie procesów sprawia wrażenie wykonania współbieżnego i system zachowuje się reaktywnie, tzn. w akceptowalnym czasie reaguje na akcje użytkownika.

Przełączanie procesów. Przełączenie procesów polega na zatrzymaniu bieżącego procesu, zapisaniu jego kontekstu w tablicach systemowych, wybraniu spośród procesów gotowych następnego do wykonania i załadowaniu jego kontekstu.

Wybrany proces kontynuuje działanie od miejsca w którym nastąpiło jego zatrzymanie. Odtwarzane są wartości jego rejestrów, w tym rejestru IP (*instruction pointer*). Równocześnie odtwarzane są wszystkie pozostałe elementy środowiska: przestrzeń adresowa pamięci, otwarte pliki, itd.



Czas przełączania kontekstu jest relatywnie długi (rzędu kilkudziesięciu mikrosekund lub nawet milisekund). Czas ten może być bardzo długi, jeżeli zachodzi konieczność załadowania do pamięci fizycznej stron pamięci zapisanych wcześniej na dysk.

Z punktu widzenia użytkownika systemu czas przełączania kontekstu jest czasem straconym, ponieważ realizowane są wówczas wyłącznie zadania systemu operacyjnego, a nie programy użytkownika.

Realizacja programu współbieżnego, w którym każdy sekwencyjny ciąg sterowania jest realizowany przez odrębny proces jest na ogół zbyt kosztowna.

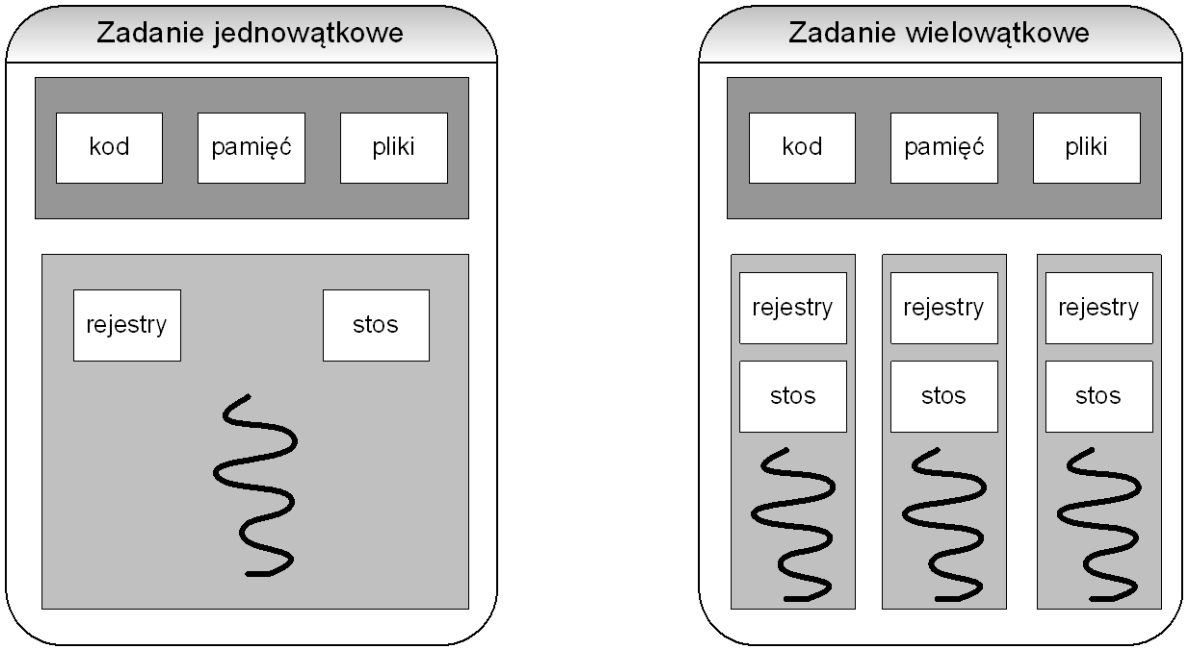
Z drugiej strony istnieją systemy operacyjne dedykowane dla aplikacji czasu rzeczywistego (VxWorks, pSOS, QNX), w których czas przełączania procesów ma gwarantowaną niską wartość. Równocześnie oferowany jest bogaty zestaw mechanizmów komunikacji, synchronizacji i wzajemnego wykluczania procesów.

Wątki w systemach operacyjnych. Wątki (ang. *thread*) są nazywane także procesami lekkimi (ang. *lightweight process*). Kontekst wątku jest znacznie mniejszy niż kontekst procesu:

- licznik instrukcji
- wartości rejestrów procesora
- stos

Wątki dzielą z innymi wątkami pozostałe zasoby: kod, pamięć, otwarte pliki i inne obiekty systemowe.

Zadaniem nazywany jest proces zawierający jeden lub więcej wątków oraz dzielone zasoby. Proces ciężki (ang. *heavy weight process*) odpowiada zadaniu z jednym wątkiem.



Zalety wątków:

- Wątki dzielą zasoby, więc nie jest konieczne ich przeładowywanie podczas przełączania kontekstu.
- Dzielenie zasobów powoduje mniejsze zużycie zasobów systemowych
- Programowanie wątków jest łatwiejsze – komunikacja odbywa się za pośrednictwem wspólnej pamięci, bez użycia specjalnych mechanizmów systemu operacyjnego
- Możliwa jest implementacja wieloprocesorowa

Implementacje wątków. W systemach operacyjnych wątki są implementowane na kilka sposobów:

- jako wątki poziomu użytkownika (ang. *user level threads*)

  W przypadku wątków poziomu użytkownika, każdemu zadaniu przydzielany jest jeden wątek jądra. Za przełączanie wątków w programie odpowiada specjalna dołączona biblioteka. (POSIX P-threads) Nie wymaga to ingerencji systemu operacyjnego.

  Zaletą jest krótki czas przełączeń i zwiększona wydajność; wadą jest niesprawiedliwy przydział czasu procesora: program wielowątkowy otrzymuje go tyle samo co jednowątkowy.

  W systemach UNIX, Linux, Solaris

- jako wątki jądra (ang. *kernel threads*)

  Każdemu wątkowi programu przydzielany jest jeden wątek jądra. Przełączanie pomiędzy wątkami jest obsługiwane przez system operacyjny.

  Zaletą jest lepsze działanie algorytmów szeregowania – system przydziela czas procesora wątkom, a nie zadaniom. Wadą jest zwiększony czas przełączeń pomiędzy wątkami, a także zmniejszona wydajność przy większej liczbie wątków.

  W systemach: Windows, Linux, Solaris

- rozwiązania mieszane

  Wątek jądra wykonuje jeden lub kilka wątków użytkownika. Łączy zalety obu rozwiązań; wadą są bardziej złożone algorytmy odwzorowania wątków.

  Systemy: Solaris 2, HP-UX

Języki programowania

Języki programowania są narzędziem umożliwiającym tworzenie programów współbieżnych.

Możliwe są dwa podejścia:

- Funkcje odpowiedzialne za uruchamianie i sterowanie wykonaniem wątków oraz zadań są częścią zewnętrznej biblioteki, a nie języka.

  W zależności od systemu operacyjnego wykorzystywane są różne biblioteki. Tworzone są programy dedykowane dla danego systemu operacyjnego. Zazwyczaj biblioteka stanowi też interfejs umożliwiający tworzenie, usuwanie i korzystanie z obiektów systemowych zapewniających wzajemne wykluczanie oraz komunikację i synchronizację procesów: semaforów, kolejek komunikatów, potoków, itd.

  - Podejście to jest typowe dla języka C.
  - Wadą tego rozwiązania jest brak przenośności. Zaletą jest duża wydajność.

- Mechanizmy współbieżne są wbudowane bezpośrednio w język programowania. Język zawiera funkcje pozwalające na tworzenie i sterowanie wykonaniem wątków (procesów). Język zawiera też konstrukcje zapewniające wzajemne wykluczanie przy dostępie do dzielonych zasobów oraz wsparcie dla synchronizacji i komunikacji pomiędzy procesami.

  - Podejście to jest typowe dla języka Ada i Java.
  - Zaletą tego rozwiązania jest przenośność i zwiększone bezpieczeństwo działania aplikacji. Wadą może być mniejsza elastyczność i wydajność programów.

Realizacja wielowątkowości w maszynie wirtualnej Java zależy ściśle od jej implementacji w systemach operacyjnych, np.: na platformie Windows wątki Java będą odwzorowane w wątki jądra, natomiast na platformie Linux będą raczej wątkami użytkownika.

Szeregowanie wątków dla różnych platform także może się różnić (inni planiści -- *schedulers*) pewne parametry, np.: priorytety mogą być ignorowane.

Thread -- klasa wątków w języku Java

Każdy język dostarcza podobnych mechanizmów umożliwiających tworzenie wątków. Programista implementuje funkcję, w której umieszcza pewną sekwencję instrukcji i uruchamia ją jako wątek. Zazwyczaj funkcja ma postać pętli.

W języku Java realizacja wątku wymaga stworzenia klasy dziedziczącej po `java.lang.Thread` i przedefiniowania metody `run()`. Metoda `start()` klasy `Thread` uruchamia funkcję `run()` jako wątek.

Przykład

```
public class TestThreads {
  static int sharedVariable=0;
  public static void main(String[] args) {
    new Inc().start();
    new Dec().start();
  }
}

class Inc extends Thread
{
  public void run() {
    for(int i=0;i<1000;i++){
      TestThreads.sharedVariable++;
      System.out.println(
      "I"+TestThreads.sharedVariable);
    }
  }
}

class Dec extends Thread
{
  public void run(){
    for(int i=0;i<1000;i++){
      TestThreads.sharedVariable--;
      System.out.println(
      "D"+TestThreads.sharedVariable);
    }
  }
}
```

Rezultat: `I1 I2 .....D236 D235 D234 .... I-2 I-1 I0`

Alternatywną metodą jest utworzenie klasy realizującej interfejs `java.lang.Runnable`. Interfejs ten deklaruje jedyną metodę: `void run()`.

Aby uruchomić zaimplementowaną metodę `run()` jako wątek, należy utworzyć obiekt klasy `Thread` przekazując mu w konstruktorze referencję do klasy implementującej `Runnable`.

Przykład:

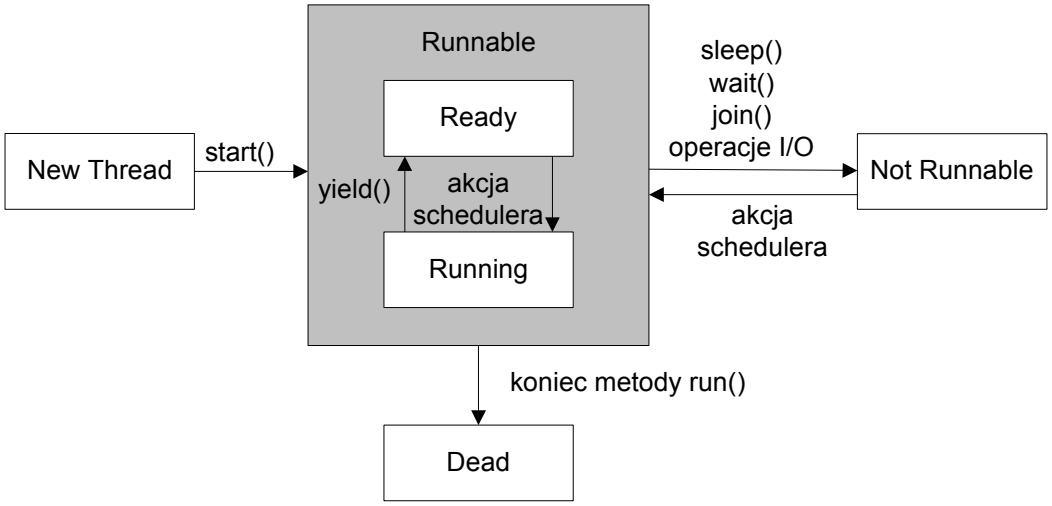
```
class Display implements Runnable
{
 public void run()
 {
   for(int i=0;i<1000;i++){
     System.out.println(
     TestThreads.sharedVariable);
   }
 }
}
```

```
public static void main(String[] args) {
  new Thread(new Display()).start();
}
```

Takie rozwiązanie może być preferowane w przypadku, kiedy klasa ma już swoje miejsce w hierarchii dziedziczenia (np.: `Display` dziedziczy już po jakiejś klasie interfejsu graficznego).

Diagram stanów wątku

Diagram stanów definiuje zachowanie wątku w odpowiedzi na wywołanie metod oraz akcji podjętych przez mechanizm szeregowania.



- *New Thread* –stan początkowy, który jest przypisany wątkowi bezpośrednio po utworzeniu obiektu.
- *Runnable* – w tym stanie wątek może być wykonywany. Stan *Runnable* ma dwa podstany: *Running* – wątek jest aktywny lub *Ready* – wątek jest gotowy do działania, ale inny wątek ma przydzielony czas procesora.
- *Not Runnable* – w tym stanie wątek nie może być wykonywany. Przyczyną może być:
  - uśpienie wątku za pomocą metody `sleep()`
  - przejście w stan oczekiwania na dostęp do chronionego monitorem obiektu, wywołanie metod `wait()` lub `join()`
  - blokada podczas operacji wejścia-wyjścia – np.: brak jest danych do odczytu lub medium nie może przyjąć kolejnej porcji danych
- *Dead* – wątek osiąga ten stan po zakończeniu metody `run()`.

Metoda currentThread

Jeżeli klasa nie jest potomną `Thread`, wtedy nie można w jej metodach wołać bezpośrednio metod klasy `Thread` , np.: `yield`, `sleep`,, itd.

Każda metoda jest jednak wykonywana przez jakiś wątek.
W momencie uruchamiania programu tworzony jest wątek, który wykonuje metodę `main()`. Metody wołane z `main()` są dalej wykonywane przez ten sam wątek.

Metoda `static Thread currentThread()` zwraca referencję do bieżącego wątku.

Przykład

```
public static void main(String[] args) {
   Thread ct = Thread.currentThread();
   System.out.println(ct.getName());
}
```

Wywołanie `Thread.currentThread()` umieszczone w dowolnej metodzie pozwala na uzyskanie dostęp do wykonującego ją wątku (podobnie jak `this` zwraca referencję do obiektu, którego metoda jest wykonywana).

Metoda `getName()` zwraca nazwę wątku. Dla wątku głównego będzie to *main*. Innym wątkom można nadać nazwy w konstruktorze. W przeciwnym razie nazwy będą wygenerowane (typu `Thread-n`).

Metoda yield

Wywołanie metody `yield()` klasy `Thread` powoduje wcześniejsze niż wynikałoby z przydzielonego dla wątku czasu procesora uruchomienie mechanizmu szeregowania (*schedulera*).

Spowoduje to wywłaszczenie bieżącego wątku i uaktywnienie innego spośród gotowych do wykonania (o ile takie istnieją).

Przykład

```
class Display implements Runnable
{
 public void run(){
    for(int i=0;i<100;i++){
      System.out.println(TestThreads.sharedVariable);
      Thread.currentThread().yield();
    }
 }
}
```

Wywołanie `yield()` umieszczone w metodzie `run()` klasy `Display` oraz analogiczne w pętlach klas `Inc` i `Dec` spowoduje zmianę sposobu szeregowania wątków. W klasach `Inc` i `Dec` metodę `yield()` można wołać bezpośrednio.

Na przykład wygenerowany ciąg będzie miał postać:

`I0 D0` `I1 D0 0 I1 D0 0 I1 D0 0...`

Można oczekiwać, że „ręczne sterowanie” mechanizmem szeregowania powinno zapewnić wygenerowanie ciągu trójek:
`< I1 D0 0 >`.

Jednak pierwsze dwa wyrazy ciągu mogą budzić niepokój. Analizując kod wątków możemy jednak odtworzyć, w jaki sposób powstał powyższy zapis z przebiegu:

Inc	Dec
`I1:for(;;){`	`D1:for(;;){`
`I2:  sharedVariable++;`	`D2:  sharedVariable--;`
`I3:..drukuj(sharedVariable);`	`D3:..drukuj(sharedVariable);`
`I4:  yield();`	`D4:  yield();`
`I5:}`	`D5:}`

Wykonane zostały instrukcje:
`D2`,wywłaszczenie wątku `Dec`, `I2`, `I3`, `I4`, `D3` i `D4`.

- Przykład pokazuje, że metody `yield()` nie należy stosować w celu uzyskania określonej kolejności wykonania wątków. Należy wtedy raczej skorzystać z ogólnych mechanizmów synchronizacji.
- Wywołania metody można wprowadzić w fazie optymalizacji działania aplikacji. Wątek może zrezygnować z czasu procesora, jeżeli stwierdzimy, że jego zadania zostały w danej chwili zrealizowane lub nie musi niczego wykonywać.

Przykład – uaktualnianie ekranu

```
for(;;){
    if(dane do wyswietlenia zostaly zmienione){
        przerysuj ekran
    }
    yield();
}
```

Metoda sleep

Statyczna metoda `sleep()` powoduje uśpienie (zatrzymanie) wątku. Argumentem jest czas uśpienia w milisekundach. Po jego upływie wątek przejdzie w stan gotowości i będzie mógł zostać uruchomiony. Rzeczywisty czas bezczynności wątku może dłuższy niż czas uśpienia.

Uśpienie może być przerwane przed czasem przez wywołanie metody `interrupt()` klasy `Thread`. (Wywołanie musi nastąpić z innego wątku). W takim przypadku wyrzucany jest wyjątek `InterruptedException`, dlatego wywołanie metody `sleep()` musi być umieszczone w bloku `try`.

```
try{
    sleep(1000); // 1 sekunda
}
catch(InterruptedException e){
 // co zrobić z InterruptedException?
 // throw new RuntimeException(e);
}
```

Metody `sleep` nie należy używać do wymuszenia określonego porządku szeregowania wątków, ale do implementacji operacji, które powinny być wykonywane periodycznie.

Priorytety

W maszynie wirtualnej Java zrealizowany jest algorytm szeregowania wątków oparty na priorytetach. Spośród gotowych do wykonania wątków uaktywniany jest zawsze ten o najwyższym priorytecie.

Priorytety są zwykle narzędziem optymalizacji działania aplikacji, na przykład wątek, który ładuje lub modyfikuje dane powinien mieć wyższy priorytet niż wątek je wyświetlający.

Priorytet jest liczbą z zakresu `Thread.MIN_PRIORITY` (1) i `Thread.MAX_PRIORITY` (10). Standardową wartością priorytetu nadawaną wątkom jest `Thread.NORM_PRIORITY` (5).

W momencie tworzenia wątek dziedziczy priorytet po wątku, w którym go utworzono. Priorytet może także zostać zmieniony za pomocą metody `setPriority()`.

Własności mechanizmu szeregowania (*schedulera*)

- Algorytm szeregowanie wątków zawiera mechanizmy wywłaszczania (ang. *preemption*) : jeżeli podczas wykonania wątku o niższym priorytecie wątek o wyższym priorytecie osiągnie stan gotowości, np.: minie jego czas uśpienia lub zostanie odblokowana jego operacja wejścia-wyjścia, wówczas wywłaszczy on bieżący wątek.
- Dla gotowych wątków o tym samym priorytecie realizowane jest szeregowanie karuzelowe (ang. *round-robin*), czyli wątki uruchamiane są kolejno.
- Specyfikacja maszyny wirtualnej Java nie narzuca stosowania mechanizmów szeregowania opierających się na podziale czasu (ang. *time slicing*). Szeregowanie z podziałem czasu polega na przydzielaniu wątkowi czasu procesora proporcjonalnego do priorytetu, a nawet dynamicznej zmianie priorytetów dla wątków, które nie były długo uruchamiane. Po upływie przydzielonego

czasu następuje uruchomienie algorytmu szeregowania i wybór następnego wątku.

W większości systemów operacyjnych algorytmy szeregowania implementują podział czasu, dlatego też w praktyce taki sposób szeregowania występuje w maszynach wirtualnych Java (chociaż nie jest zagwarantowany).

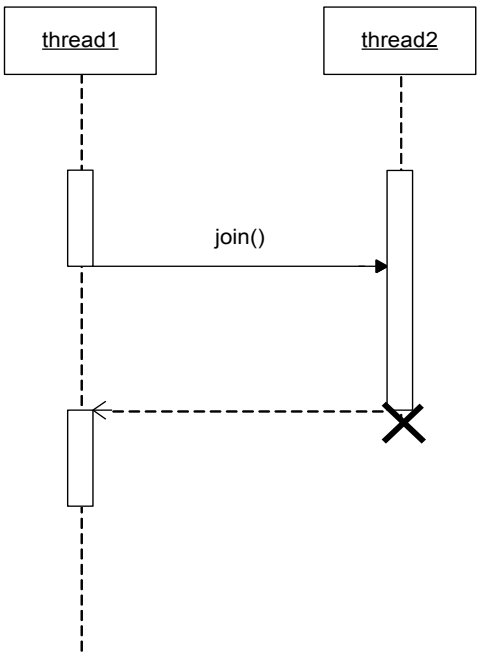
Tak więc aktywny wątek będzie wykonywany, aż do momentu:

- Kiedy wątek o wyższym priorytecie osiągnie stan gotowości i go wywłaszczy.
- Wywołana zostanie metodę `yield()`, `sleep()`, `wait()` lub jego metoda `run()` zakończy działanie
- Dla systemów z podziałem czasu wyczerpany zostanie przydzielony czas procesora.

W dokumentacji języka Java można znaleźć szereg ostrzeżeń przed założeniem, że wątki będą wykonywane w środowisku z podziałem czasu. Specyfikacja maszyny wirtualnej gwarantuje system szeregowania oparty na priorytetach z wywłaszczaniem. Dla wątków o tym samym priorytecie ma być stosowane szeregowanie karuzelowe, czyli wątki będą wykonywane, aż do osiągnięcia stanu, kiedy nie będą mogły kontynuować działania. Podział czasu jest opcjonalny, dlatego, zgodnie z dokumentacją, wątki nie powinny zachowywać się egoistycznie i możliwie często wołać metodę `yield()` umożliwiając częste przełączanie pomiędzy wątkami o tym samym priorytecie.

Metoda join

Wywołanie metody `join()` wskazanego wątku, np.: `thread2.join()` powoduje zatrzymanie bieżącego wątku, np.: `thread1`, aż do momentu kiedy drugi zakończy działanie (czyli zmieni stan na *Dead*).



Wątek wywołujący metodę `join()` przechodzi w stan *Not Runnable*.

Jego stan zawieszenia może zostać przerwany za pomocą wywołanej z zewnątrz metody `interrupt()` , dlatego metoda może wyrzucać wyjątek `InterrruptedException`.

Istnieją też przeciążone wersje metody `join()` – z argumentem liczbowym określającym maksymalny czas zawieszenia w milisekundach. Po upływie tego czasu wątek przechodzi w stan *Runnable* bez generacji wyjątku.

Przykład

```
public static void main(String[] args) {
  Inc inc = new Inc();
  Dec dec = new Dec();
  inc.start();
  dec.start();
  try{
    inc.join();
    dec.join();
  }
  catch(InterruptedException e){}
  System.out.println("Koniec...");
}
```

Wpierw czekamy na zakończenie wątki `Inc`, a potem `Dec`. Następnie drukowany jest tekst `Koniec...`

Wątki – demony

Demon jest wątkiem, który wykonywany jest w tle podczas działania programu, ale nie realizuje zadań pierwszoplanowych, dla których program został utworzony. Typowym przykładem może być wątek realizujący usuwanie obiektów (*garbage collection*).

Program będzie działał, dopóki aktywny jest co najmniej jeden wątek, który nie jest demonem. Jeżeli w uruchomionym programie aktywne są wyłącznie wątki—demony, program zakończy działanie.

Metoda `setDaemon(boolean on)` ustala status wątku. Dla argumentu `true`, wątek będzie demonem. Metoda musi być wywołana przed uruchomieniem wątku za pomocą metody `start()`.

Metoda `boolean isDaemon()` pozwala odczytać status wątku. Wszystkie wątki utworzone przez wątki--demony są również demonami.

Współdziałanie wątków

Wątki składające się na współbieżnie wykonywany program nie działają w oderwaniu od siebie. Muszą one:

- Komunikować się, czyli wymieniać informacje
- Wzajemnie wykluczać podczas dostępu do dzielonych zasobów
- Synchronizować swoje działania

Komunikacja pomiędzy wątkami jest stosunkowo prosta, ponieważ dzielą one pamięć. Mogą one wymieniać informacje za pomocą statycznych pól klas lub obiektów umieszczonych na stercie, które w każdym z wątków mogą być wskazywane przez referencje.

Możliwe są również bardziej złożone mechanizmy komunikacji, np.: potoki.

Wzajemne wykluczanie. Wątki mogą wspólnie korzystać zasobów: obiektów umieszczonych w pamięci, ale także plików, połączeń sieciowych. Problemem jest to, że czasem usiłują z nich korzystać *równocześnie*, naruszając w ten sposób spójność danych. Należy bronić się przed tym używając odpowiednich algorytmów albo stosując ogólne mechanizmy wzajemnego wykluczania (ang. *mutual exclusion*).

Kod procesu (wątku), w którym realizowany jest dostęp do dzielonego zasobu w sposób niosący ryzyko utraty spójności danych traktowany jest jak sekcja krytyczna.

> *Sekcja krytyczna* to część procesu, która nie może być wykonana współbieżnie z sekcją krytyczną innego procesu. Jeśli dany proces wejdzie do sekcji krytycznej, wówczas dopóki jej nie opuści, żaden inny proces nie będzie mógł wejść do sekcji krytycznej. Pomiędzy sekcjami krytycznymi zachodzi wzajemne wykluczanie.

Typowa konstrukcja mechanizmu wzajemnego wykluczania, to użycie flagi (semafora) zezwalającej na wejście do sekcji krytycznej.

```
shared int semaphore=1;
for(;;){
    while(semaphore == 0);
    semaphore --;
    critical section;
    semaphore ++;
}
```

Procesy czekające na wejście do sekcji krytycznej mogą wykonywać nieskończoną pętlę – `while(semaphore == 0);` ale lepiej jest, jeżeli na czas oczekiwania przechodzą w stan blokady.

Problemem kluczowym dla implementacji mechanizmu wzajemnego wykluczania jest niepodzielność instrukcji – proces nie powinien zostać wywłaszczony pomiędzy stwierdzeniem, że zmienna `semaphore` ma wartość 1, a więc można wejść do sekcji krytycznej, a ustawieniem jej na 0.

Dlatego też najczęściej mechanizmy wzajemnego wykluczania implementuje się najczęściej na poziomie systemu operacyjnego jako wywołania procedur jądra.

W języku Java mechanizm wzajemnego wykluczania polega na czasowym przejęciu „własności” obiektu przez wątek i jest realizowany wewnątrz maszyny wirtualnej.

Synchronizacja pomiędzy procesami lub wątkami polega na określeniu warunków, kiedy wątek powinien zostać zawieszony, na przykład w oczekiwaniu na dane lub zakończenie operacji wykonywanej przez inny wątek oraz uaktywniony, kiedy dane staną się dostępne lub operacja zakończona.

Metoda `join()` jest typową metodą synchronizacji, ale ma ograniczone zastosowanie. W języku Java stosuje się parę metod `wait()` – czekaj na sygnał do reaktywacji i `notify()` – powiadom wątek, że można kontynuować działanie.

Problemy programowania współbieżnego

Podczas implementacji oprogramowania współbieżnego można napotkać na szereg typowych problemów.

- Wyścig, szkodliwe współzawodnictwo (ang. *race conditions*).

`Shared TYPE A = undetermined;`	
`Thread1`	`Thread2`

`void run(){`
`  initT1();`
`  calculateA();`
`}`
	`void run(){`
	`  initT2();`
	`  useA(A);`
	`  calculateB();`
	`}`

Uruchamiając wątki (procesy) oczekujemy, że wpierw wątek `Thread1` wyznaczy wartość `A` za pomocą funkcji `calculateA()`, a następnie drugi wątek `Thread2` wykorzysta tę wartość w `useA(A)`. Tak może się dziać w większości zaobserwowanych przebiegów programu, ponieważ, na przykład, operacja `initT2()` jest na ogół dłuższa niż `initT1()` i `calculateA()` lub wątek `Thread2` jest uruchamiany później.

Potencjalnie jednak istnieje takie szeregowanie, w którym `useA()` jest uruchamiane przed `calculateA()` i jest ono czasem (rzadko) obserwowane.

Rozwiązaniem jest synchronizacja działania wątków.

- Zakleszczenie (ang. *deadlock*).

Zakleszczenie jest to sytuacja, kiedy żaden z procesów (wątków) nie może kontynuować działania, ponieważ czeka na warunek lub zdarzenie, które pojawiłoby się tylko wtedy, gdyby inny proces mógł zakończyć wykonywaną operację.

Zakleszczenie może pojawić się w wyniku zastosowania mechanizmów synchronizacyjnych lub wzajemnego wykluczania.

Thread1	Thread2

`void run(){`
`  for(;;){`
`  waitForB();`
`  waitForA();`
`  useAB();`
`  releaseAB();`
`  }`
`}`
	`void run(){`
	`  for(;;){`
	`  waitForA();`
	`  waitForB();`
	`  useA();`
	`  useB();`
	`  releaseAB();`
	`  }`
	`}`

- Głodowanie (ang. *starvation*).

Głodowanie jest sytuacją, kiedy pewien proces czeka na warunek lub zdarzenie, które *może* nigdy nie nastąpić.

Na przykład proces nie może wejść do sekcji krytycznej, ponieważ zawsze jest wyprzedzany przez inne procesy – te o wyższym priorytecie.

Problemów głodowania na ogół nie rozwiązuje się algorytmicznie, ale raczej wbudowuje się odpowiednie zabezpieczenia w mechanizmy wzajemnego wykluczania i szeregowania procesów środowiska wykonania (systemu operacyjnego lub maszyny wirtualnej).

Typowym rozwiązaniem może być użycie kolejek. Wątki usiłujące wejść do zajętej sekcji krytycznej są rejestrowane w kolejce FIFO. Po zwolnieniu sekcji krytycznej uaktywniany jest pierwszy wątek z kolejki.

Innym przykładem może być dynamiczna zmiana priorytetów. Mechanizm szeregowania wraz z upływem czasu dynamicznie zwiększa priorytet głodującego wątku. Kiedy wzrośnie on wystarczająco by zrównać się z priorytetami innych działających wątków, będzie on uaktywniony.

W niektórych przypadkach zapewnienie sprawiedliwego przydziału zasobów musi jednak być implementowane algorytmicznie. Dotyczy to na przykład procesów działających w systemie rozproszonym, gdzie brak jest centralnego mechanizmu sterującego dostępem do dzielonych zasobów sieciowych.

W takim przypadku procesy nie powinny zachowywać się egoistycznie, dobrowolnie zezwalając innym na wejście do sekcji krytycznej. Częstym rozwiązaniem jest sprawdzenie, czy inny proces zgłosił chęć wejścia do sekcji krytycznej. Jeżeli lista procesów oczekujących nie jest pusta, wówczas proces jest usypiany na *losowo* wybrany przedział czasu przed złożeniem żądania wejścia, co pozwala innym na wcześniejsze osiągnięcie sekcji krytycznej.

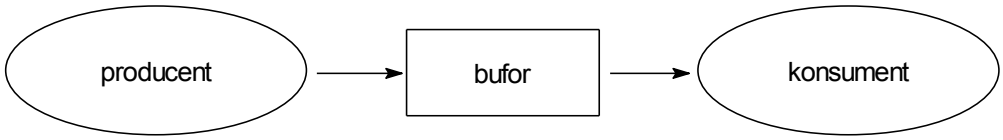
Zagadnienie producenta i konsumenta

W zagadnieniu występują trzy elementy:

- *Producent* – generuje ciąg danych i wprowadza je do bufora (kanału komunikacji)
- *Konsument* – pobiera kolejne dane z bufora i przetwarza je dalej
- *Bufor* – ma skończoną pojemność. Istnieją abstrakcje, gdzie zakłada się nieskończone pojemności bufora.

Podstawowe założenia:

1. Producent i konsument mogą pracować z różną szybkością
2. Poszczególne elementy danych muszą być otrzymane przez konsumenta w takiej kolejności, w jakiej wygenerował je producent.
3. Żaden z elementów nie może zostać zagubiony podczas przesyłu
4. Elementy nie mogą być zwielokrotniane (np.: otrzymywane dwukrotnie)



Ograniczenia synchronizacyjne:

- Reguła przyczynowości. Proces konsumenta nie może otrzymać danych, dopóki producent ich nie wygeneruje. Jeśli bufor jest pusty – proces konsumenta gotowy do pobierania danych powinien ulec zawieszeniu.
- Bufor ma skończoną pojemność. Producent nie może wysyłać kolejnych elementów, jeśli bufor jest pełny.

Implementacja

```
class BufferEmptyException extends Exception{}
class BufferFullException extends Exception{}

interface Buffer
{
  void put(int i)throws BufferFullException;
  int take()throws BufferEmptyException;
}
```

Klasy producenta i konsumenta

```
class Producer extends Thread
{
  Buffer buf;
  public void run(){
    for(int i=0;;i++){
      try{
        buf.put(i);
        sleep((int)(100*Math.random()));
      }
      catch(BufferFullException e){
        System.out.println( "buffer full...." );
        i--;
        yield();
      }
      catch(InterruptedException e){}
    }
  }
}
```

- Producent wykonuje pętlę nieskończoną. Umieszcza liczbę w buforze i jest usypiany na losowo wybrany przedział czasu.
- Jeżeli bufor jest pełny (`BufferFullException`), modyfikuje zmienną iteracyjną, aby nie tracić wartości i dobrowolnie zrzeka się przydzielonego czasu procesora – `yield()`.

```
class Consumer extends Thread
{
  Buffer buf;
  public void run(){
    for(;;){
      try{
        System.out.println( buf.take() );
        sleep((int)(100*Math.random()));
      }
      catch(BufferEmptyException e){
        System.out.println( "buffer empty...." );
        yield();
      }
      catch(InterruptedException e){}
    }
  }
}
```

Konsument zaimplementowany jest w analogiczny sposób: odczytuje liczby i drukuje je po czym usypia.

Klasa `ArrayBuffer` implementuje bufor w postaci tablicy.

```
class ArrayBuffer implements Buffer
{
static final int SIZE=10;
private int[] buf = new int[SIZE];
int count=0;
  public void put(int i)throws BufferFullException {
  ❶ if(count==SIZE)throw new BufferFullException();
  ❷ buf[count]=i;
  ❸ count++;
  }
  public int take() throws BufferEmptyException {
  ❹ if (count == 0) throw new BufferEmptyException();
  ❺ int retval = buf[0];
  ❻ System.arraycopy(buf,1,buf,0,count-1);
  ❼ count--;
  ❽ return retval;
  }
}
```

Program będzie drukował kolejne liczby (pojawiały będą się także napisy informujące o stanie bufora).

Niespójność

Analiza kodu wskazuje, że kod może doprowadzić do niespójności danych w buforze

Sytuacja początkowa:

12	13	14	15	16					

Konsument wykonuje ❹ ❺ ❻ i jest wywłaszczany

13	14	15	16	16					

Producent wykonuje ❶ ❷ ❸ i jest wywłaszczany

13	14	15	16	16	17				

Konsument wykonuje ❼ ❽

13	14	15	16	16	17				

W rezultacie wartość 17 jest tracona.

Problemem implementacji jest to, że operacje `put()` i `take()` korzystają ze wspólnej zmiennej `count`. Wywłaszczenie wątku w trakcie wykonywania metody może prowadzić do niespójności danych.

Możliwe rozwiązania:

- Implementacja bufora w sposób zapewniający działanie wątków na odrębnych zestawach zmiennych licznikowych.
- Zastosowanie mechanizmu wzajemnego wykluczania.

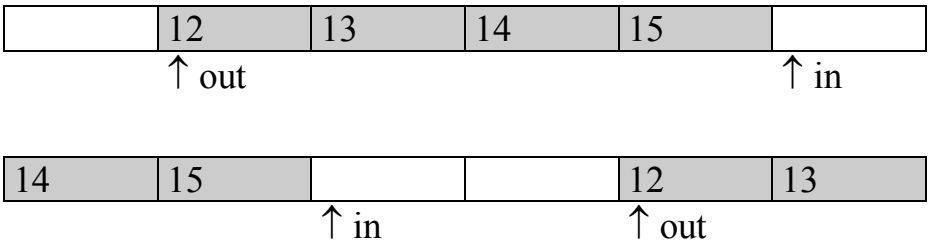
Bufor cykliczny

Bufor cykliczny jest powszechnie stosowaną strukturą danych w problemach producent—konsument.

```
class CyclicBuffer implements Buffer
{
static final int SIZE=10;
private int[] buf = new int[SIZE];
private int in=0,out=0;

  public void put(int i)throws BufferFullException {
    if((in+1)%SIZE == out)throw new BufferFullException();
    buf[in]=i;
    in=(in+1)%SIZE;
  }
  public int take() throws BufferEmptyException {
    if (in == out) throw new BufferEmptyException();
    int retval = buf[out];
    out=(out+1)%SIZE;
    return retval;
  }
}
```

Zaletą jest bufora cyklicznego jest brak kopiowania elementów i niezależność operacji na licznikach elementów.



Metody synchroniczne

Bufor cykliczny może zostać wprowadzony w stan niespójności danych w przypadku, uogólnionego zagadnienia, w którym występuje *n* producentów i *m* konsumentów.

Dlatego lepszym rozwiązaniem jest potraktowanie metod `put()` i `take()` bufora jako sekcji krytycznych należących do jednej *klasy* i zastosowanie mechanizmu wzajemnego wykluczania.

> *Klasa sekcji krytycznych* to zbiór sekcji krytycznych pomiędzy którymi musi zachodzić wzajemne wykluczanie. Na przykład kod realizujący dostęp do danych i kod metod działających na atrybutach obiektu `A` jest jedną klasą, kod realizujący dostęp do obiektu `B` drugą klasą.
>
> *Mechanizm wzajemnego wykluczania powinien zabezpieczać dostęp do danych obiektu, a nie kod.*

Przekształcenie metody w sekcję krytyczną jest w sensie składni języka zabiegiem bardzo prostym: polega na dodaniu do deklaracji funkcji słowa kluczowego `synchronized`.

```
class ArrayBuffer implements Buffer
{
// deklaracje
  public synchronized void put(int i)
    throws BufferFullException {
  // implementacja
  }
  public synchronized int take()
    throws BufferEmptyException {
  // implementacja
  }
}
```

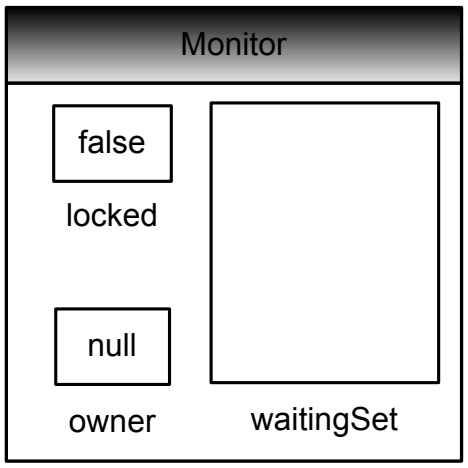
Zastosowanie metod synchronicznych zapewnia, że kiedy wątek `Producer` będzie wykonywał metodę `put()`, wątek `Consumer` nie będzie równocześnie wykonywał metody `take()`. Zostanie on zablokowany w momencie wywołania metody.

W każdy obiekt w języku Java wbudowana jest struktura danych nazywana *monitorem*. Monitor zapewnia wzajemne wykluczanie wątków przy dostępie do metod obiektu oraz pozwala na synchronizację wątków.

- Dokładnie jeden wątek może wykonywać metodę synchroniczną obiektu. Jest on tzw. *właścicielem monitora*.
- Pozostałe wątki mogą używać metod niesynchronicznych lub muszą czekać na zwolnienie monitora (wyjście właściciela z metody synchronicznej).
- Wątek będący właścicielem monitora może wywoływać inne metody synchroniczne. Bez tej własności monitora dochodziłoby do zakleszczeń.
- Wątek będący właścicielem monitora może zostać wywłaszczony w trakcie wykonywania metody synchronicznej. Kiedy jednak inny aktywny wątek będzie usiłował wywołać metodę synchroniczną obiektu, zostanie on zablokowany. Da to szansę powtórnego przejęcia procesora przez właściciela monitora i wykonania metody synchronicznej do końca.

Konstrukcja monitora

Idea konstrukcji monitora może być przedstawiona na poniższym schemacie



locked – atrybut przechowujący informację o stanie monitora

owner – referencja do wątku będącego właścicielem monitora

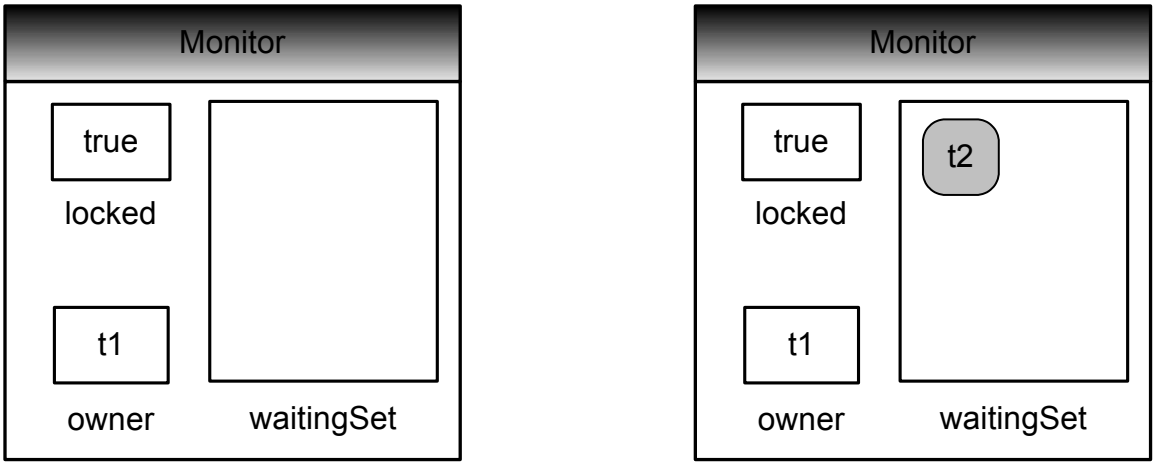
waitingSet – zbiór referencji do wątków oczekujących na przejęcie monitora.

Wątki oczekujące mogą być w stanie *Runnable* lub *Not Runnable*.

Wątki w stanie *Runnable* współzawodniczą o przejęcie monitora. Wątki w stanie *Not Runnable* nie są brane pod uwagę -- czekają na zmianę stanu za pomocą `notify()` lub `notifyAll()`.

W momencie, kiedy wątek `t1` wywołuje metodę synchroniczna, a monitor jest wolny, staje się on właścicielem monitora.

Kiedy monitor jest w posiadaniu wątku `t1`, kolejny wątek `t2` zostanie zablokowany i trafia do kolekcji `waitingSet`.



Implementacja metody jako metody synchronicznej powoduje dodanie na jej początku i końcu niewidocznego dla użytkownika kodu prologu i epilogu.

```
synchronized foo(){
    prolog();
    body;
    epilog();
}
```

```
void prolog(){
  if(locked && owner==currentThread )return;
  if(locked ){
    zawieś currentThread
    waitingSet.add(currentThread);
  }else{
    locked=true;
    owner= currentThread;
  }
}
void epilog(){
  locked=false;
  monitorOwner=null;
  if(!waitingSet.empty()){
  // usuń wątek t z waitingSet
  // przydziel monitor lub powtorz prolog dla t
  }
}
```

Nakłady na implementację monitorów

- To, że metoda jest metodą synchroniczną jest cechą implementacji, a nie specyfikacji metod (słowo kluczowe synchronized nie zmienia sygnatury metody).
- Użycie metod synchronicznych kilkakrotnie zwiększa nakłady czasowe na wywołanie i wyjście z metody.
- Monitor jest częścią każdego obiektu. Może się to wydawać pewną rozrzutnością.

Sprawiedliwość

- Mechanizm wzajemnego wykluczania powinna cechować sprawiedliwość (ang. *fairness*). Powinien on zapewniać wątkom dostęp do chronionego zasobu w sposób nienarażający ich na głodowanie. Sprawiedliwość gwarantuje, że wątek zablokowany w oczekiwaniu na monitor w skończonym czasie stanie się jego właścicielem.
- Specyfikacja maszyny wirtualnej Java nie narzuca ograniczeń na kolejność w jakiej wątki oczekujące (umieszczone w `waitingSet`) otrzymują przydział monitora. Jest to pozostawione implementacji maszyny wirtualnej (FIFO, priorytety, wybieranie losowe)

Inne zastosowania monitorów

- Monitor może być także częścią klasy. Dzięki temu możemy używać statycznych metod synchronicznych i chronić dostęp do statycznych atrybutów.

```
class Foo
{
  static int v;
  synchronized static void f(...){...
      //dostep do atrybutów statycznych...
  }
  synchronized static void g(...){
      //dostep do atrybutów statycznych...
  }
}
```

- Oprócz metod synchronicznych, możliwe jest używanie bloków synchronicznych.

```
void f()
{
  //... niesynchroniczna część metody
  synchronized(anObject){
      //sekcja krytyczna
  }
  //... niesynchroniczna część metody
}
```

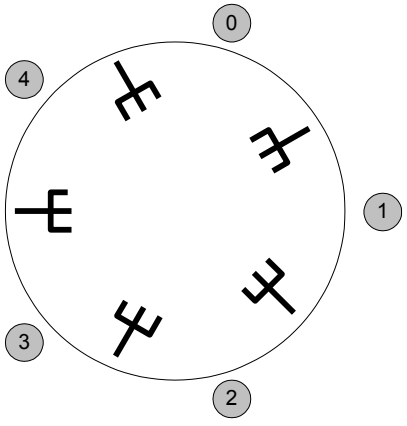
W takim przypadku pozostałe wątki będą mogły wykonywać równolegle niesynchroniczne fragmenty kodu; wzajemne wykluczanie będzie zachodziło dla sekcji krytycznej umieszczonej wewnątrz bloku.

Użycie bloku synchronicznego wymaga wskazania obiektu, którego monitor będzie używany do wzajemnego wykluczania. Może to być obiekt, którego metoda jest wykonywana (`this`) lub dowolny obiekt stworzony wyłącznie w celu synchronizacji.

Problem pięciu filozofów

Problem pięciu filozofów jest znanym przykładem zakleszczenia, które może powstać w wyniku zastosowania mechanizmów wzajemnego wykluczania podczas dzielenia zasobów.

Pięciu filozofów zgromadzono przy okrągłym stole, na którym stoi miska spaghetti. Każdy z filozofów ma stałe miejsce przy stole. Pomiędzy filozofami położono pięć widelców. Filozofowie na przemian myślą i jedzą. Aby przystąpić do jedzenia, filozof musi podnieść dwa widelce. Zakładamy, że robi to w ustalonej kolejności: wpierw sięga po lewy, a następnie po prawy widelec. Po zaspokojeniu głodu filozofowie odkładają widelce w odwrotnej kolejności.



Filozofowie dzielą widelce z sąsiadami. Jeśli któryś z widelców jest używany, filozof, który chciałby przystąpić do jedzenia musi czekać (być może z jednym widelcem w ręce), aż niezbędny widelec zostanie zwolniony.

Kod dla problemu

```
public class FivePhilo
{
  static class Fork{};
  static Fork[] fork = new Fork[5];
  static{
    for(int i=0;i<5;i++)fork[i]=new Fork();
  }
  public static void main(String args[])
  {
    for(int i=0;i<5;i++){
      new Philosopher(i).start();
      }
  }
}

class Philosopher extends Thread
{
  int i;
  Philosopher(int _i){i=_i;}
  public void run(){
    for(;;){
      System.out.println("Ph#"+i+" thinks");
      synchronized(FivePhilo.fork[i]){
        System.out.println(
          "\tPh#"+i+" waits for fork#"+(i+1)%5);
        yield();  // to generate deadlock
        synchronized(FivePhilo.fork[(i+1)%5]){
          System.out.println("Ph#"+i+" eats");
        }
      }
    }
  }
}
```

Przykład pokazuje zagnieżdżenie bloków synchronicznych. Obiektami użytymi do synchronizacji są dwa kolejne widelce.

Wywołanie `yield()` zapewnia szybkie osiągnięcie stanu zakleszczenia.

Synchronizacja wątków

Jeżeli wątek w trakcie wykonania napotka sytuację, kiedy nie powinien kontynuować działania, może zawiesić czasowo dalsze wykonanie wołając metodę `wait()`.

Inny wątek, po usunięciu przyczyny zawieszenia pierwszego wątku, może go odblokować za pośrednictwem metody `notify()` lub `notifyAll()`.

Wykorzystanie metod wymaga użycia monitora. Zawieszony wątek trafia do zbioru oczekujących wątków monitora; budzone są wątki oczekujące na monitorze.

Metody `wait()`, `notify()` i `notifyAll()` są metodami klasy `Object` i są dziedziczone przez wszystkie klasy Java. Wywołujący je wątek musi być właścicielem monitora, stąd ich wywołanie może się znaleźć w:

- metodzie synchronicznej i ma postać:

```
synchronized void foo(){
  try{
      wait(); // lub this.wait()
  }
  catch(InterruptedExcepion e){}
}
```

- bloku synchronicznym:

```
void foo(){
  // ...
  synchronized(anObject){
      anObject.notify();
  }
  // ...
}
```

Metoda wait

Wywołanie metody `wait()` klasy `Object` powoduje zawieszenie wykonującego ją wątku.

Istnieją trzy przeciążone wersje metody `wait()` – bezparametrowa i z parametrami podającymi czas zawieszenia.

Metoda `wait()` powoduje, że bieżący wątek *T*:

- zmienia stan na *Not Runnable,*
- zwalnia monitor obiektu, którego metoda była wołana,
- przechodzi do zbioru oczekujących wątków monitora (`waitingSet`).

Wątek *T* zostanie przebudzony (zmieni stan na *Runnable*) jeżeli:

- inny wątek wywoła metodę `notify()` tego samego obiektu i *T* zostanie wybrany do aktywacji ze zbioru `waitingSet`,
- wywołana zostanie metoda `notifyAll`() obiektu
- inny wątek wywoła metodę `interrupt()` wątku *T*, aby przerwać jego stan zawieszenia
- dla wersji metody `wait()` z parametrami określającymi maksymalny czas oczekiwania -- czas ten upłynie.

Po przebudzeniu wątek *T* będzie brany pod uwagę przez mechanizm szeregowania wątków i będzie współzawodniczył z innymi wątkami o przejęcie własności monitora.

Kiedy wątek *T* stanie się właścicielem monitora następuje powrót z wywołania metody `wait()`.

Wątek, którego zawieszenie zostanie przerwane wywołaniem metody `interrupt()` będzie również czekał na przejęcie monitora i dopiero wówczas zostanie wygenerowany wyjątek `InterruptedException`.

Metoda notify

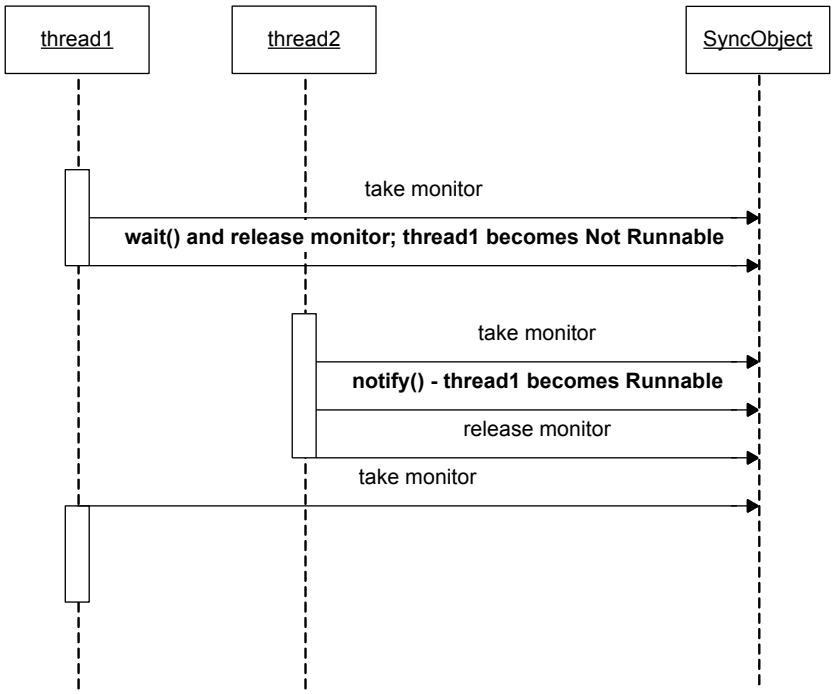
Wywołanie metody `notify()` obiektu powoduje, że arbitralnie wybrany wątek w stanie *Not Runnable* znajdujący się w kolekcji `waitingSet` monitora wbudowanego w obiekt zmieni stan na *Runnable*.

Metoda notifyAll

Wywołanie metody `notifyAll()` obiektu powoduje, że wszystkie wątki w stanie *Not Runnable* znajdujący się w kolekcji `waitingSet` monitora wbudowanego w obiekt zmienią stan na *Runnable*.

Wątki w stanie *Runnable* będą współzawodniczyły o przejęcie własności monitora zgodnie z implementacją zastosowaną w maszynie wirtualnej. W konsekwencji po zwolnieniu monitora (metody `notify()` i `notifyAll()` są zawsze wykonywane przez właściciela monitora) *jeden z nich* go przejmie i będzie mógł kontynuować działanie.

Mechanizm zastosowania metod `wait()` i `notify()` do synchronizacji pokazany jest na poniższym diagramie.



- Wątek `thread1` po wykonaniu `wait()` czeka na zmianę stanu z *Not Runnable* na *Runnable* dokonaną w wyniku wywołania `notify()` przez wątek `thread2`. Gwarantowane jest, że przejęcie monitora i uaktywnienie nie nastąpi przed wywołaniem `notify()`.
- Metody `wait()` i `notify()` są metodami obiektu służącego do synchronizacji: `SyncObject`.

Synchronizacja w problemie producenta i konsumenta

```
class SyncBuffer implements Buffer
{
static final int SIZE=10;
private int[] buf = new int[SIZE];
int count=0;

public synchronized void put(int i)
  throws BufferFullException {
  while(count==SIZE){  // if(count==SIZE)
    try{
      System.out.println( "producer waits... " );
      wait();
    }
    catch (InterruptedException e){ // rethrow
      throw new BufferFullException();
    }
  }
  buf[count]=i;
  count++;

  notifyAll();
}

public synchronized int take() throws BufferEmptyException {

  while(count == 0){ // if(count== 0)
    try{
      System.out.println( "consumer waits.... " );
      wait();
    }
    catch (InterruptedException e){ // rethrow
      throw new BufferEmptyException();
    }
  }

  int retval = buf[0];
  System.arraycopy(buf,1,buf,0,count-1);
  count--;

  notifyAll();
  return retval;
}
}
```

Dlaczego warunek jest testowany w pętli

- Jeżeli bufor jest pełny producent zawiesza działanie wołając `wait()`. Analogicznie działa konsument dla pustego bufora.
- W zagadnieniu, w którym występuje jeden producent i konsument testowanie warunku mogłoby się odbywać w instrukcji `if`:

```
if(count==0){ ... wait() ...}
```

- Dla *n* producentów i *m* konsumentów może się zdarzyć, że stan bufora znowu ulegnie zmianie *zanim* obudzony wątek przejmie monitor (*race condition*) . Z tego powodu warunek powinien być testowany w pętli

```
while(count==0){ ... wait() ...}
```

Dodatkowy parametr timeout

Podany kod łatwo zmodyfikować dodając parametr `timeout` dla operacji `put` i `take`.

```
public synchronized void put(int i)
throws BufferFullException {
  put(i,0);
}

public synchronized void put(int i,long timeout)
  throws BufferFullException {
  while(count==SIZE){
    try{
      wait(timeout);
    }
    catch (InterruptedException e){
      throw new BufferFullException();
    }
  }
// itd
}
```

Wywołanie `wait(0)` daje taki sam efekt, jak `wait()`. W wersji metody z parametrem `timeout` po upływie tego czasu stan oczekiwania będzie przerwany, wygenerowany zostanie wyjątek `InterruptedException` i następnie po przechwyceniu nastąpi konwersja na `BufferFullException`.

Wyjątki w metodach synchronicznych

Pojawienie się nieprzechwyconego wyjątku w metodzie synchronicznej obiektu zawsze powoduje zwolnienie monitora. Chroni to przed zakleszczeniem – gdyby monitor nie został zwolniony, wówczas żaden inny wątek nie mógłby uzyskać dostępu do obiektu.

Wyrzucając wyjątki należy zadbać, aby pozostawić dane obiektu w spójnym stanie.

```
class SharedResource
{
  synchronized void access() throws Exception
  {
    try{
      // inconsistent state
      if(condition)throw new Exception();
      // still inconsistent state
      // consistent state
    }
    catch(Exception e){
      // fix consistent state and rethrow
      throw (Exception)e.initCause(e);
    }
  }
}
```

Zatrzymywanie wątków

Wątek kończy działanie, po wyjściu z metody `run()` lub po pojawieniu się wyjątku w jednej z wykonywanych metod, który nie został przechwycony w metodzie `run()` i dotarł do szczytu stosu.

Metody, które „straciły wartość”

We wcześniejszych wersjach biblioteki zaimplementowano metody, które sterowały zatrzymaniem lub zawieszeniem wątków: `stop()`, `suspend()` i `resume()`.

Były one wzorowane na bibliotecznych metodach implementowanych w języku C/C++ przeznaczonych do sterowania wykonaniem wątków lub procesów w systemie operacyjnym.

Metody te są w dokumentacji oznaczone jako *deprecated*. Oznacza to, że stanowią one część interfejsu, pozostawiono je dla kompatybilności z poprzednimi wersjami biblioteki, ale ich użycie jest odradzane.

Wywołanie `stop()` zatrzymuje wątek wysyłając do niego wyjątek `ThreadDeath`. Dziedziczy on po klasie `Error`, czyli nie jest wyjątkiem weryfikowanym i nie powinien być przechwytywany.

```
public static void main(String args[])
{
  AnyThread s = new AnyThread();
  s.start();
  try{
    Thread.currentThread().sleep(1000);
  }
  catch(Exception e){}
  s.stop();
}
```

Metoda `stop()` może być wywołana z zewnątrz w dowolnym momencie, na przykład wtedy, kiedy wątek `AnyThread` jest właścicielem monitora dzielonego obiektu i obiekt jest w niespójnym stanie.

Propagacja wyjątku `ThreadDeath` w górę stosu powoduje zwolnienie wszystkich monitorów. Ponieważ wyjątek nie jest przechwytywany, wątek `AnyThread` może pozostawić obiekty w niespójnym stanie, co prowadzi do trudnych do wychwycenia i usunięcia błędów.

Metody `suspend()` – zawieś wątek -- nie należy używać z przeciwnego powodu: `suspend()` zawiesza wątek *nie zwalniając monitora*, co może prowadzić do zakleszczeń (podobnie jak w przykładzie pięciu filozofów.)

Metoda `resume()` – uaktywnij wątek zawieszony za pomocą `suspend()` jest metodą komplementarną.

Jak zatrzymać lub zawiesić wątek z zewnątrz

Najbezpieczniej jest dodać publiczną zmienną sterującą (np.: `boolean stop`), która będzie sprawdzana przez wątek w dogodnym miejscu, poza sekcją krytyczną.

W podobny sposób można bezpiecznie zawiesić wątek w miejscu, gdzie nie jest on właścicielem monitora chroniącego dzielony zasób.

```
class Suspendable extends Thread
{
boolean suspend=false;
boolean stop=false;

synchronized void wakeup(){
  suspend=false;
  notify();
}
void safeSuspend(){
  suspend=true;
}

public void run()
{
  for(int i=0;!stop;i++){
    System.out.println(i);

    synchronized(sharedObject){
      // critical section
    }

    synchronized(this){
      try{
        if(suspend){
          System.out.println("suspending");
          wait();
        }
      }
      catch(InterruptedException e){}
    }

  }
}
}
```

Wywołanie

```
public static void main(String args[])
{
  Suspendable s = new Suspendable();
  s.start();
  // ...
  s.safeSuspend();
  // ...
  s.wakeup();

  // ...
  s.stop=true;
}
```