Automatic Mutual Exclusion

Filip Rachwalak

10 grudnia 2014

Spis treści

Dlaczego nie mutexy?

- Częste i łatwe do popełnienia błędy programistów: zakleszczenia, zawieszenie GUI.
- Konieczność przestrzegania kolejności zajmowanych zamków.
- Im większy projekt, tym (znacznie) trudniejsze programowanie.

Dlaczego nie pamięć transakcyjna?

- Wciąż wymaga decyzji programisty, które bloki kodu muszą być chronione.
- Gry programista chce zawęzić zasięg bloku chronionego musi usunąć żądaną część kodu z bloku, a to wpływa na zrozumienie kodu np. przez następnego programistę (szczególnie w dużych projektach).

Główna idea AME

W aplikacji wielowątkowej:

- Domyślnie chroniony cały kod, deklarowanie kodu niechronionego.
- Poprawność ponad wydajnością.
- Łatwe zarządzanie kodem.

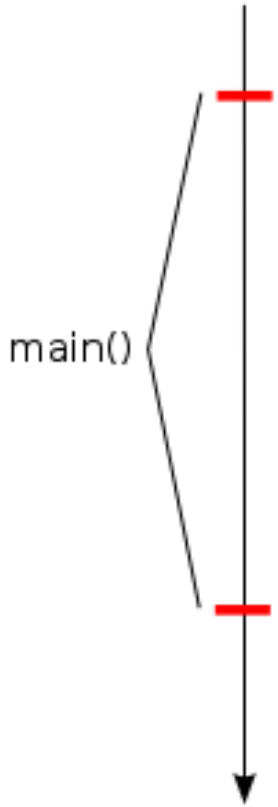
PODSTAWOWY MODEL SYSTEMU AME

Asynchroniczne wywołania metod

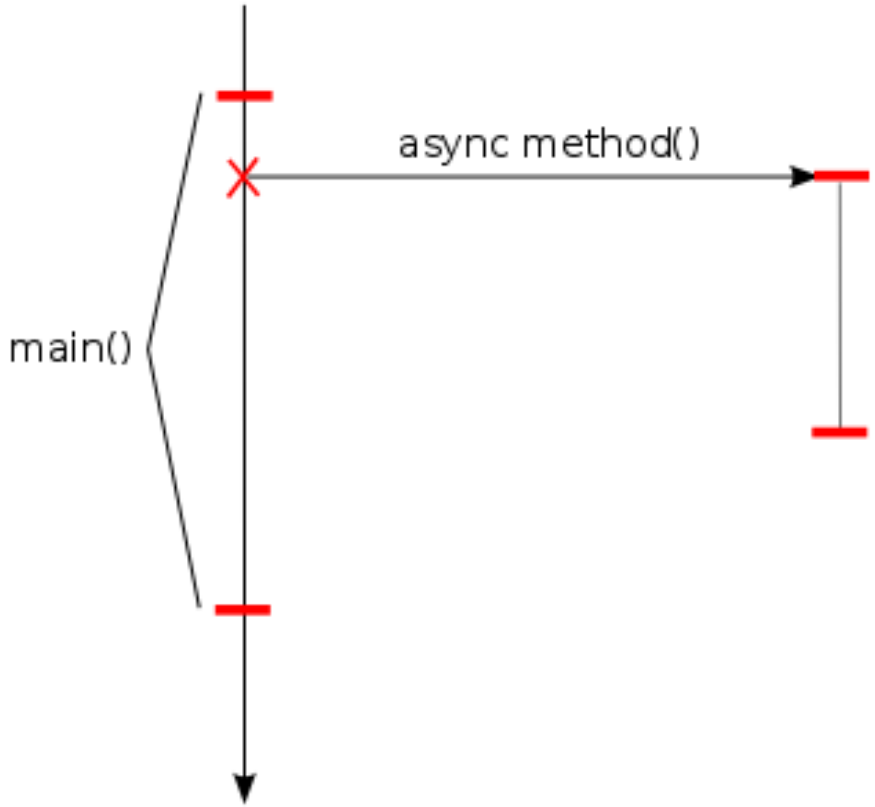
Automatic Mutual Exclusion:

- składa się z wywołań metod asynchronicznych,
- gwarantuje, że wynik wykonanych wywołań będzie równoważny wynikowi, który byłby osiągnięty poprzez wykonanie sekwencyjne (atomowość, niekoniecznie kolejność),
- kończy się, gdy wszystkie asynchroniczne metody zostaną wykonane,
- osiąga współbieżność poprzez wykonywanie wywołań niekonfliktowych.

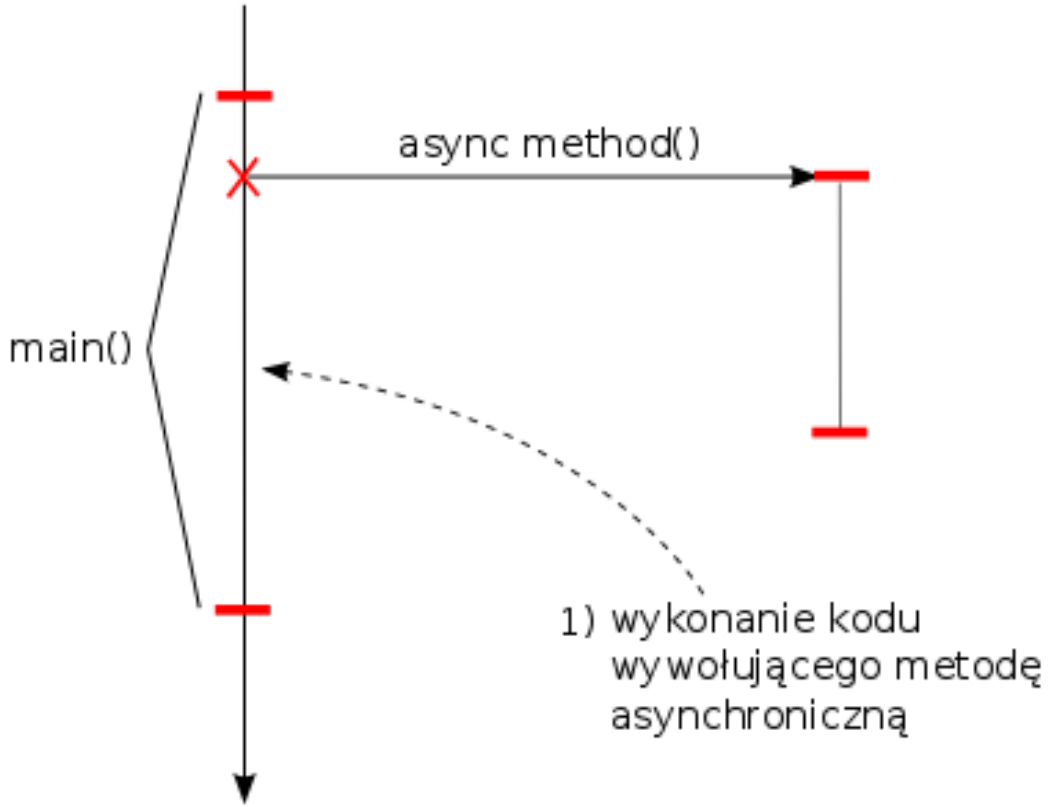
Działanie programu



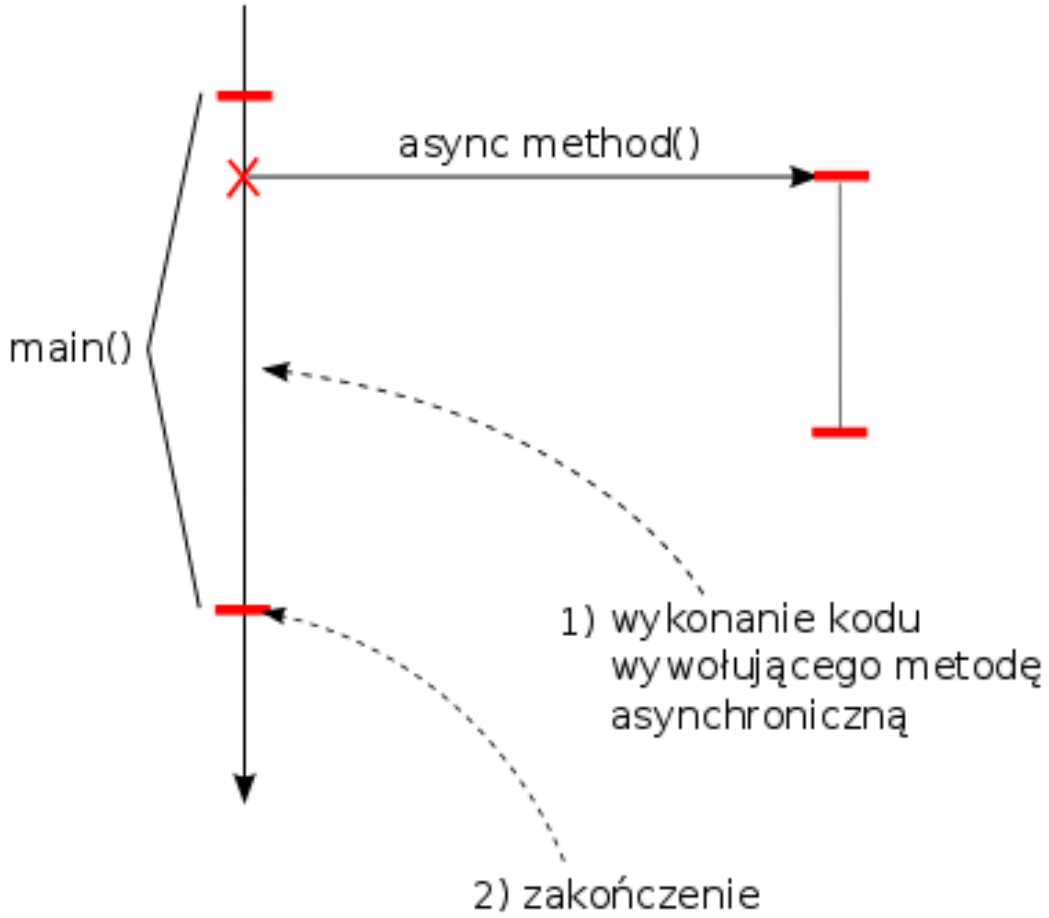
Działanie programu



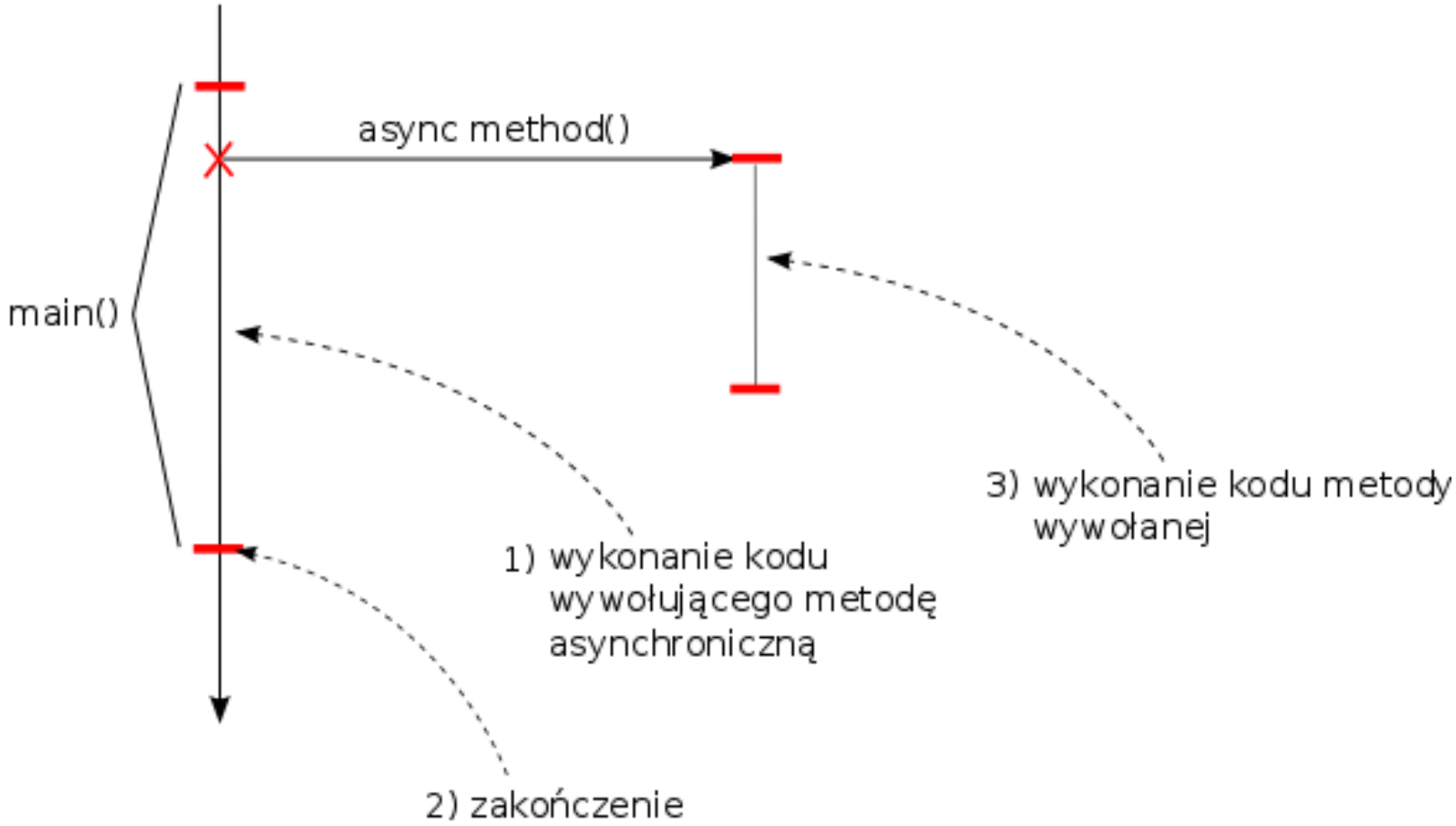
Działanie programu



Działanie programu



Działanie programu



Blokowanie wywołań

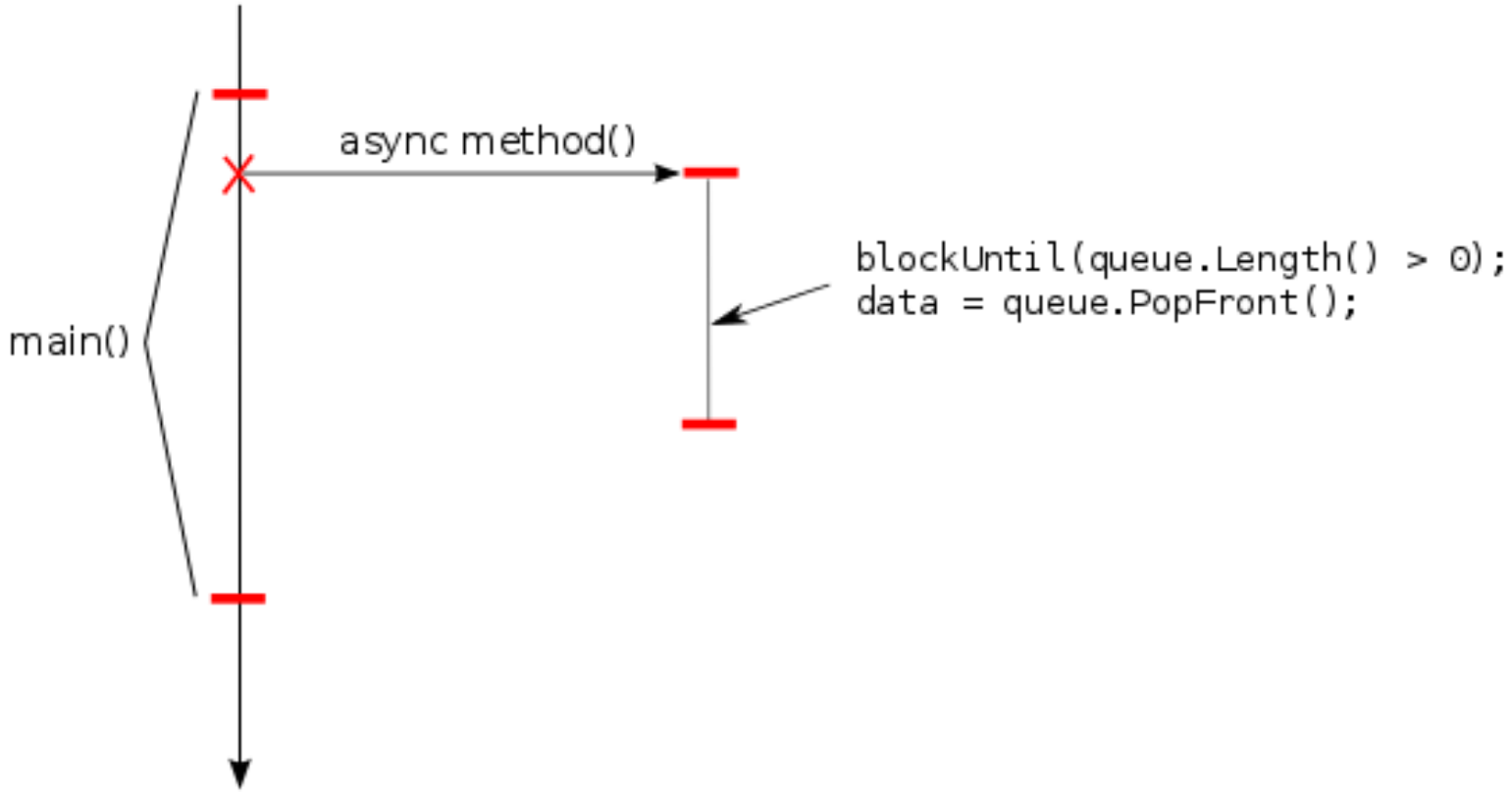
`BlockUntil(bool condition)`

Jeśli `condition = true`, to nic się nie dzieje (kontynuacja przetwarzania).
W przeciwnym wypadku wycofanie aktualnej transakcji i ponowna próba po pewnym czasie.

Ważne!

Jeśli metoda asynchroniczna (transakcja) zatrzyma się na którejś z kolei instrukcji `BlockUntil()`, wycofywana jest cała transakcja!

Blokowanie wywołań



Przykład Zombie

```
void startZombie() {
    Zombie z;
    z.initialize();
    async UpdateZombie(z);
}
```

```
void UpdateZombie(Zombie z) {
    Time now = GetTimeNow();
    BlockUntil(now - z.lastUpdate >
            z.updateInterval);
    z.lastUpdate = now;
    MoveAround(z);
    if (Distance(z, player) < DeathRadius) {
        KillPlayer();
    } else {
        async UpdateZombie(z);
    }
}
```

SZCZEGÓŁOWY MODEL SYSTEMU AME

I co dalej?

Problem

Długie metody z wieloma instrukcjami `BlockUntil()` będą częściej powodować wycofanie transakcji.

Rozwiązanie

Podział aktualnej transakcji za pomocą metody `Yield()` na atomowe bloki.

`BlockUntil()` zablokuje tylko fragment od poprzedniego wywołania metody `Yield()`, a jeśli będzie wznawiać działanie po aborcie, zrobi to od poprzedniego wywołania `Yield()`.

Używanie `Yield()`

Wymagane jest, aby metoda zawierająca `Yield()` deklarowała w swoim nagłówku słowo kluczowe `yields`:

```
returnType someMethod(args) yields { ... }
```

Natomiast każde wywołanie metody zawierającej `Yield()` powinno wyglądać następująco:

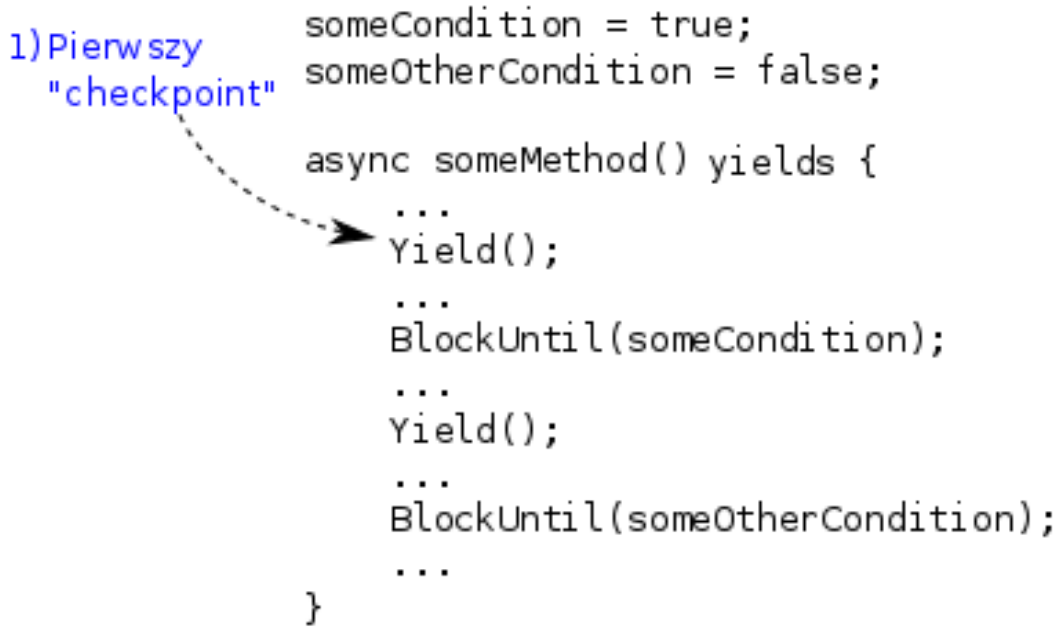
```
int foo = someMethod(x) yielding;
```

Dzielenie metod – przykład

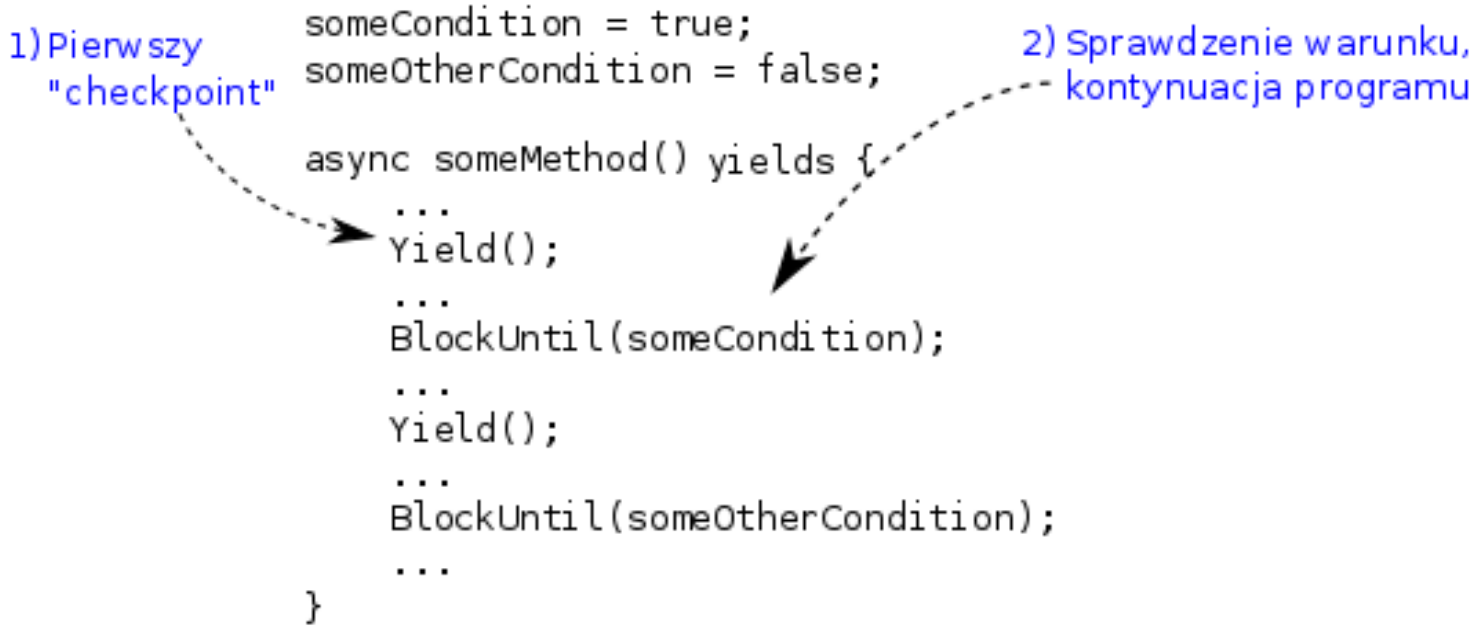
```
someCondition = true;
someOtherCondition = false;

async someMethod() yields {
    ...
    Yield();
    ...
    BlockUntil(someCondition);
    ...
    Yield();
    ...
    BlockUntil(someOtherCondition);
    ...
}
```

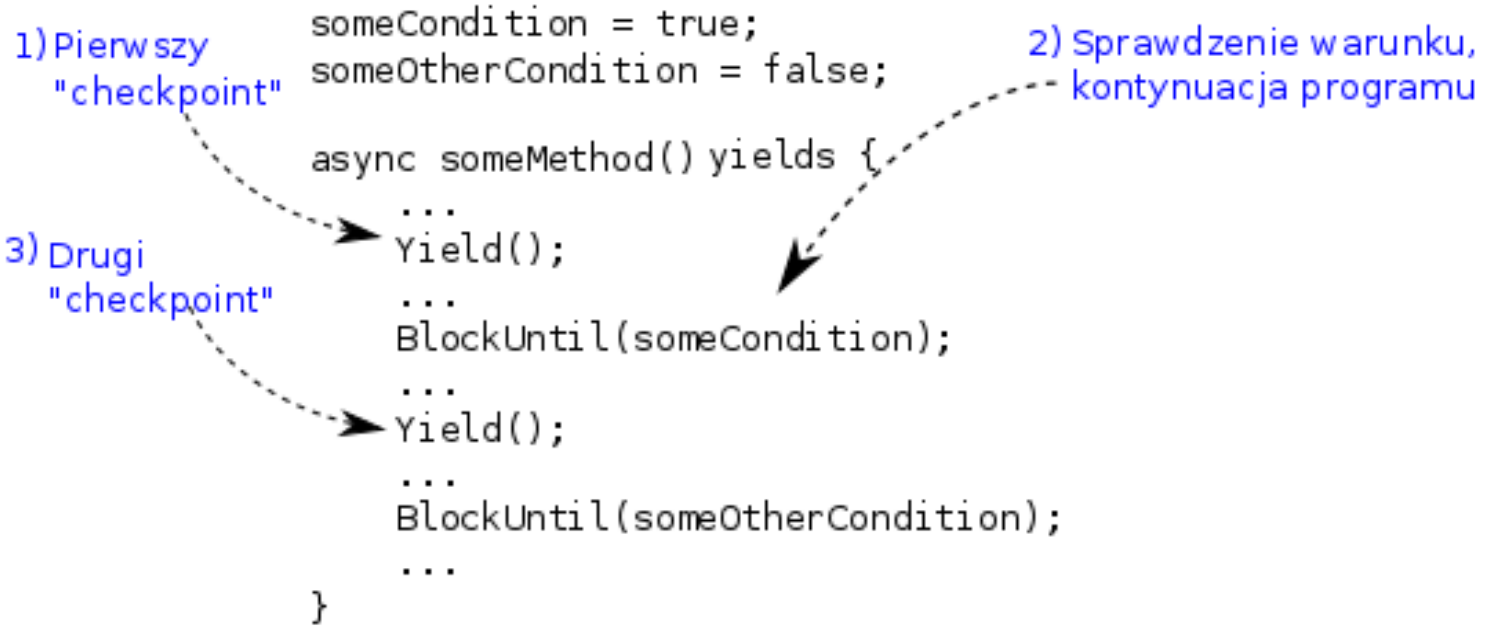
Dzielenie metod – przykład



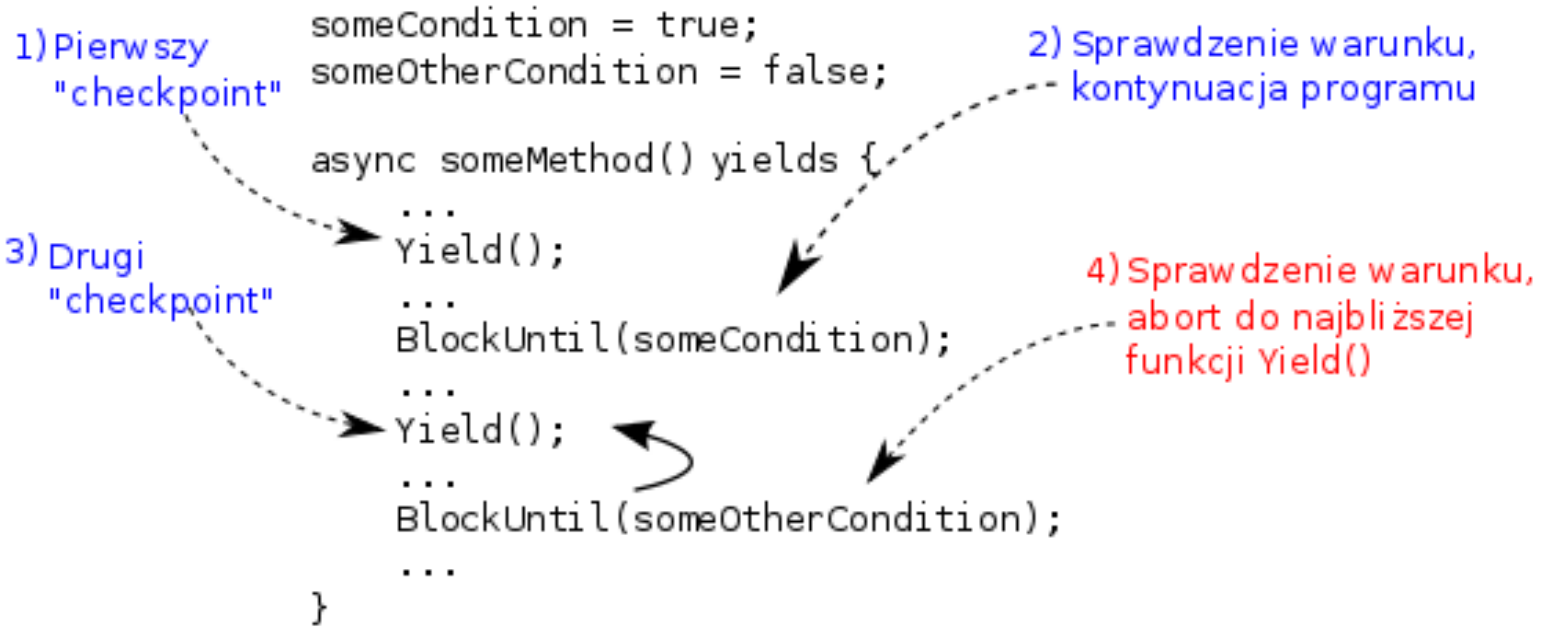
Dzielenie metod – przykład



Dzielenie metod – przykład



Dzielenie metod – przykład



Dzielenie metod – przykład Zombie

```
void startZombie() {
    Zombie z;
    z.initialize();
    async UpdateZombie(z);
}


void UpdateZombie(Zombie z) {
    Time now = GetTimeNow();
    BlockUntil(now - z.lastUpdate >
            z.updateInterval);
    z.lastUpdate = now;
    MoveAround(z);
    if (Distance(z, player) < DeathRadius) {
        KillPlayer();
    } else {
        async UpdateZombie(z);
    }
}
```

```
void RunZombie() yields {
    Zombie z;
    z.initialize();
    while (Distance(z, player) >=
           DeathRadius) {
        Yield();
        Time now = GetTimeNow();
        BlockUntil(now - z.lastUpdate >
                   z.updateInterval);
        z.lastUpdate = now;
        MoveAround(z);
        if (Distance(z, player) <
            DeathRadius) {
            KillPlayer();
        }
    }
}
```

`unprotected { ... }`

Użycie bloku `unprotected` powoduje zakończenie aktualnej transakcji (commit/yield), następnie wykonanie kodu wewnątrz bloku bez synchronizacji i rozpoczęcie nowej transakcji

Każda metoda korzystająca z `unprotected` musi mieć w nagłówku słowo kluczowe `yields`.

unprotected { ... }

```
void method yields {
    << someCode1 >>
    ...
    unprotected {                    // end of transaction
        unsynchronizedCode
    }                                // start of new transaction
    ...
    << someCode2 >>
}
```

Podsumowanie

We encourage correctness first, performance second and maintainability always.

System oparty zarówno na transakcjach jak i standardowych mutexach.

Przemyślany dla utrzymania systemów dużych i żywotnych.

Wyzwania

Scheduling w celu zminimalizowania ilości cofniętych transakcji.

Optymalizacje sytuacji, gdy `BlockUntil` stoi na początku fragmentu atomowego.

Wzbogacenie instrukcji `Yield` np. o deklarowanie zmiennych uaktualnianych tą funkcją, dzięki czemu wzrośnie poprawność.

Dziękuję za uwagę

DZIĘKUJĘ ZA UWAGĘ

Filip Rachwalak