Sztuka Wytwarzania Oprogramowania

Wykład 13 - współbieżność. Współbieżne wzorce projektowe, cz 1

Robert Nowak

25Z

Wzorce projektowe

- standardowe rozwiązania często pojawiających się problemów projektowych
- sprawdzone w praktyce

Przykłady: obiektowe wzorce projektowe (prototyp, kompozyt, adapter, leniwe tworzenie, leniwe kopiowanie, obserwator, fasada, fabryki, singleton, wizytator, wielometoda, most, komenda)

Plan wykładu

- procesy, wątki, wyścigi
- blokady
- skalowalność
- wzorce
  - zdobywanie zasobów jest inicjacją (RAII)
  - kończenie wątków
  - podwójne sprawdzanie
  - monitor (pasywny obiekty)
  - współbieżna blokada (czytelnicy/pisarze)

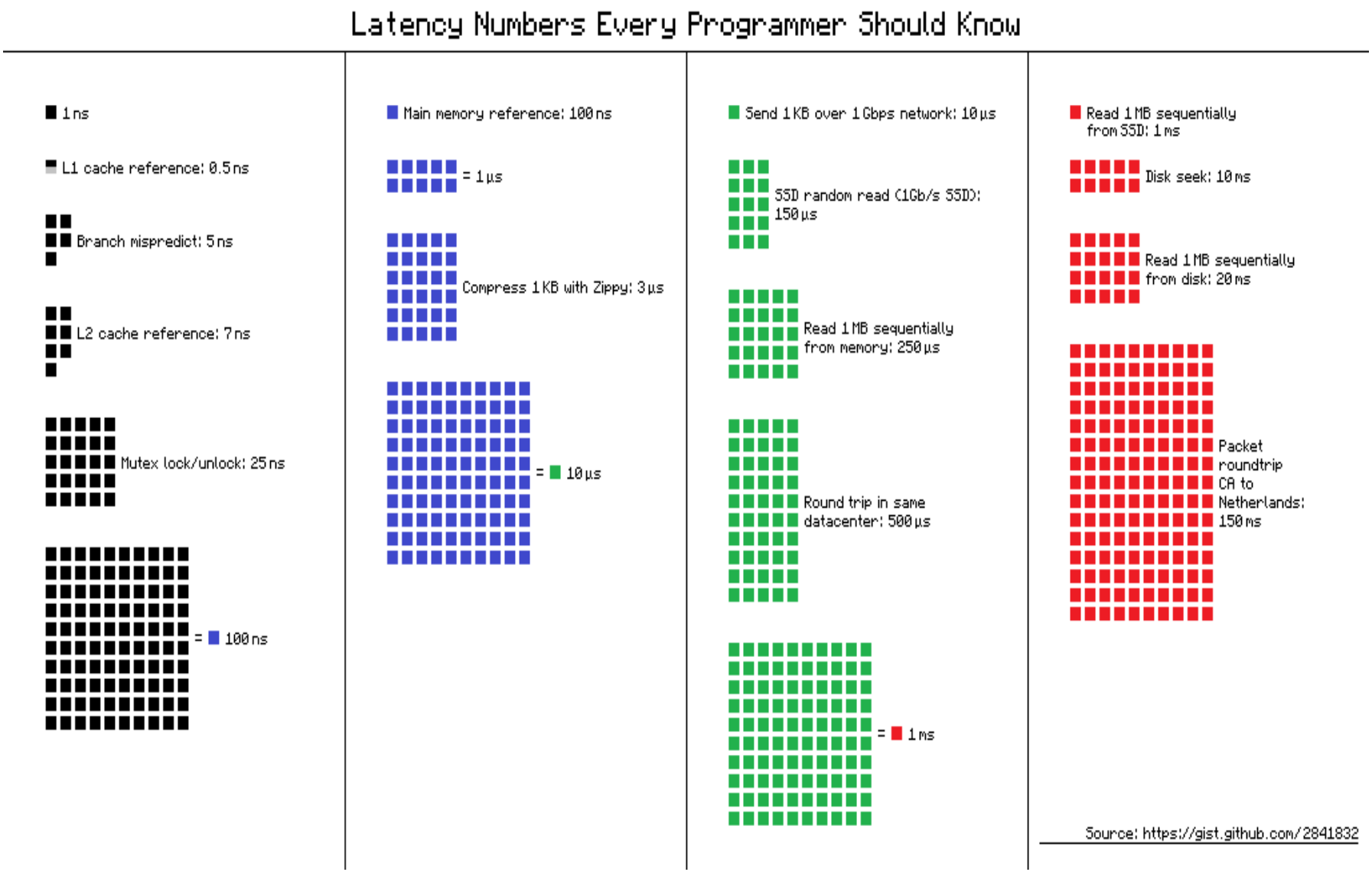
Równoległość - obliczenia podczas obsługi urządzeń

- bardzo wolna reakcja człowieka
- wolne urządzenia wejścia - wyjścia (np. drukarki)
- bardzo szybkie procesory

	PC AT (1987)	PC (2025)	Poprawa
zegar procesora	6 MHz	3.9 GHz	**650x**
ilość rdzeni (procesorów)	1	24	**24x**
pamięć wielkość	1MB	64 GB	**64000x**
pamięć (czas dostępu)	200 ns	15 ns	**13x**
pamięć (czas dostępu/cykl)	1.4	54	**-39x**

rodzaj pamięci	wielkość	czas dostępu (cykle)
cache L1	64kB (na rdzeń)	2
cache L2	256kB (na rdzeń)	14
cache L3	16MB	16
DRAM	64GB	54
HDD	2000GB	**15000000**

Graficzne porównanie czasów realizacji operacji



[1]

[1] zakładając 1GB/sec SSD, dane z Peter Norvig, Teach Yourself Programming

Współbieżność

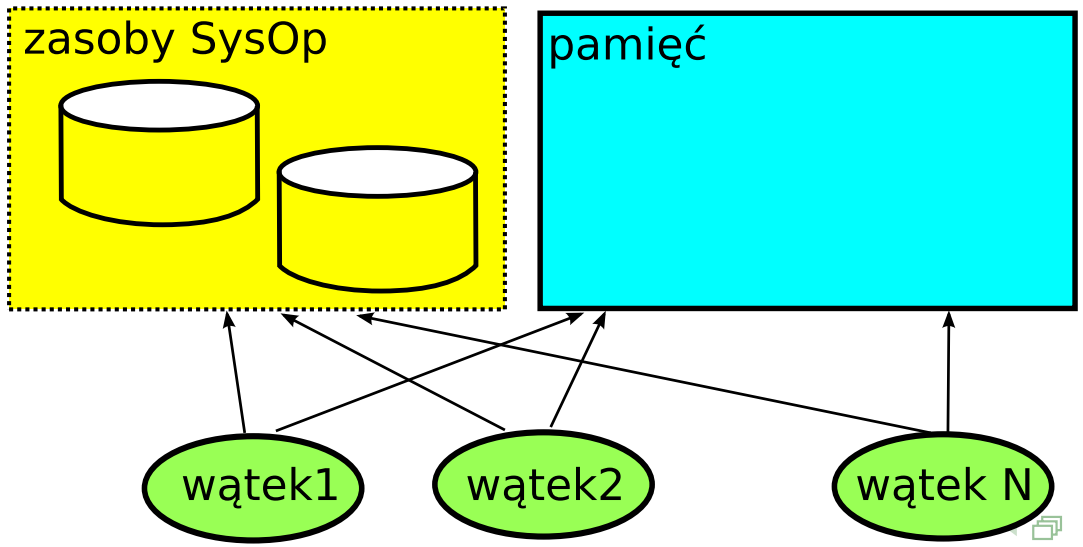
wieloprocesowe systemy operacyjne

- na platformach jedno-procesorowych (jedno-rdzeniowych)
- na platformach wieloprocesorowych (wielordzeniowych)

aplikacje wielowątkowe

Wątek („lekki proces”) realizuje niezależne ciągi instrukcji w ramach procesu.

- wątki współdzielą kod, dane oraz zasoby.
- mechanizm przełączania nie wprowadza dużych narzutów



Aplikacja wielowątkowa

- pozwala na reakcję na zlecenia użytkownika podczas przeprowadzania obliczeń
- lepiej wykorzystuje dostępną moc obliczeniową (szczególnie na platformach wieloprocesorowych)
- może obsługiwać wiele zleceń równolegle
- poprawnie zaprojektowana jest szybsza na platformach wieloprocesorowych (wielordzeniowych)

Procesy, wątki - były omawiane na przedmiocie 'Systemy Operacyjne'. Tutaj pokazujemy, jak używać tych mechanizmów w tworzeniu bibliotek i aplikacji.

Język programowania, a tworzenie aplikacji wielowątkowej

- języki jedno-wątkowe: JavaScript
- języki wielo-wątkowe: C++, Java

Dla C++:

- należy używać bibliotek przeznaczonych do pracy wielowątkowej
- C++11 dostarcza mechanizmy tworzenia i synchronizacji wątków

Współdzielone	Niezależne
▶ obiekty globalne ▶ obiekty dynamiczne (adres unikalny w ramach procesu) ▶ zasoby systemu operacyjnego (np. pliki) ▶ kod wykonywany	▶ rejestry ▶ ciąg wyk. instrukcji ▶ stos ▶ obiekty automatyczne (dostęp przez stos)

Program ma zawsze jeden wątek (funkcja main()) - wątek główny lub inicjujący. Może tworzyć dodatkowe wątki.

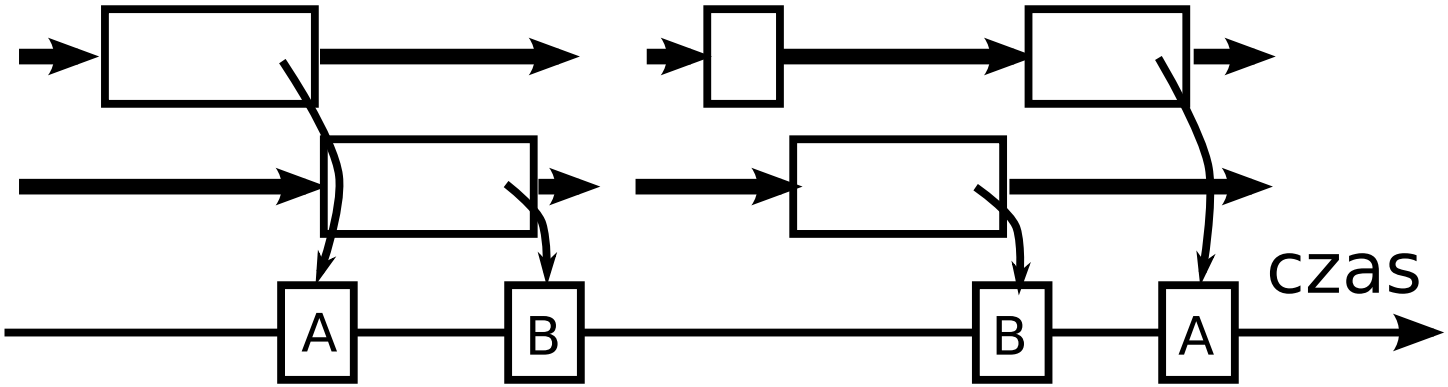
std::thread - wątki w C++

```
#include <thread>

//Funkcja główna wątku użytkownika
void my_thread() { /* ... */ }
//Można też użyć funktora
class MyThread {
  public:
  //tutaj implementacja funkcji wątku użytkownika
  void operator()() { /* ... */ }
};

int main() {
  std::thread thrd(&my_thread); //Utworzenie i uruchomienie wątku
  try {
    thrd.join(); //Bieżący watek czeka na zakończenie wątku thrd
  } catch(...) { } //wyjątek zgłaszany, gdy wątek już nie istnieje
  return 0;
}
```

Wyścigi (race conditions)

wiele wątków pisze lub czyta tę samą pamięć $\rightarrow$

- niewłaściwa wartość
- niezdefiniowane zachowanie



jedno z rozwiązań: synchronizacja (blokady)

problemy z usuwaniem błędów w aplikacjach wielowątkowych

- niektóre błędy są niepowtarzalne
- różne zachowanie się wersji debug od release
- testy na platformach z jednym procesorem (rdzeniem) mogą nie pokazywać błędów które wystąpią na platformach posiadających wiele procesorów (rdzeni)
- trudno testować wszystkie możliwe przebiegi sterowania

Blokady, mutex (mutual exclusion)

Mutex to obiekt synchronizujący, pozwala tworzyć sekcje krytyczne

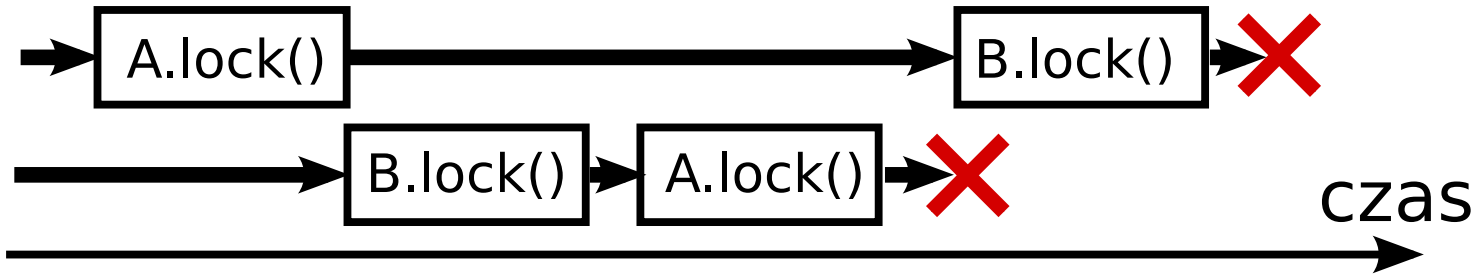
```
#include <mutex>
mutex.lock();
//Sekcja krytyczna, w danej chwili dostęp ma tylko jeden wątek
mutex.unlock();
```

Wzorzec projektowy - RAII - zdobywanie zasobów jest inicjacją

```
struct Lock { //Zdobywanie zasobów jest inicjowaniem RAII
  Lock(mutex& m) : m_(m) { m_.lock(); }
  ~Lock() { m_.unlock(); } //Destruktor wychodzi z sekcji
  mutex& m_; //Mutex, którym zarządza
};
//Przykład użycia
std::mutex m; //obiekt służącego do synchronizacji
{
  Lock guard(m); //lock_guard - RAII dla mutex
  //sekcja krytyczna, dostęp do zasobus
}//zwolnienie zasobu
```

Zakleszczenia (deadlock)

każdy z wątków czeka na jakiś inny (jest blokowany). Żaden z nich nie może dalej pracować.



Rozwiązanie:

- zajmowanie blokad zawsze w tej samej kolejności
- stosowanie innych mechanizmów synchronizujących

Kończenie wątku - tylko z wewnątrz

Błędne jest przerwanie wątku z zewnątrz, nieustalony stan aplikacji.

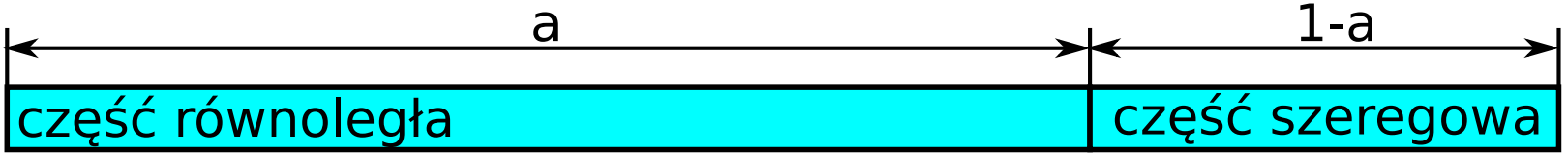
```
class MyThread {
public:
  MyThread() : finish_(false) {}
  //sygnalizacja, że wątek ma się zakończyć wcześniej
  void finish() { finish_ = true; }
  void operator()() {
    while(!finish_) {
      //część przetwarzania
      //a następnie sprawdzanie warunku !finish_
    }
  }
private:
  volatile bool finish_; //zabronione optymalizowanie tej składowej
};
```

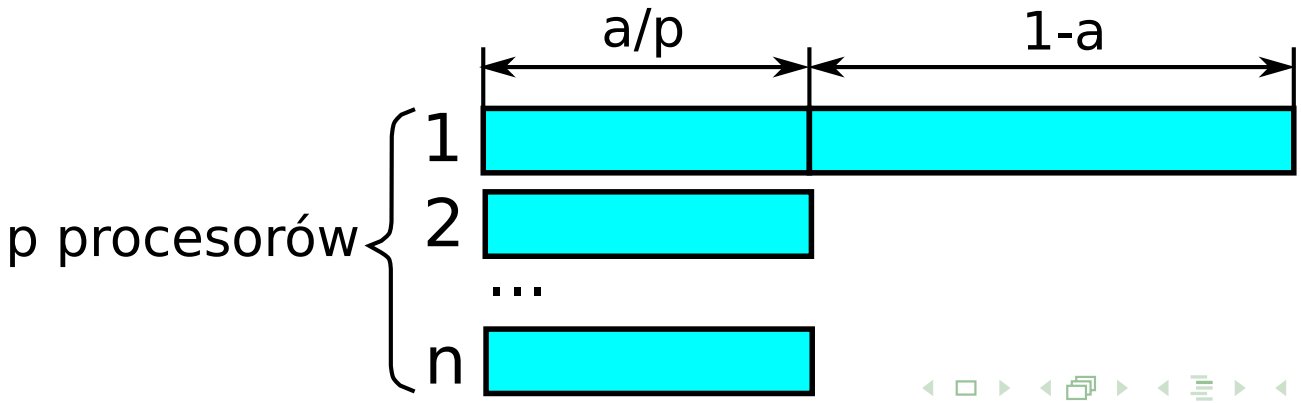
Skalowalność - wydajność przy zwiększaniu zasobów

Prawo Amdahla (przyspieszenie dla 'p' procesorów):

$$S(p) = \frac{1}{1 - a + \frac{a}{n}}$$

- ta sama ilość danych
- pomijamy narzuty współbieżności (np. czas przełączania)
- a - proporcja algorytmu, który może podlegać zrównolegleniu, $(1 - a)$ musi być wykonywane szeregowo

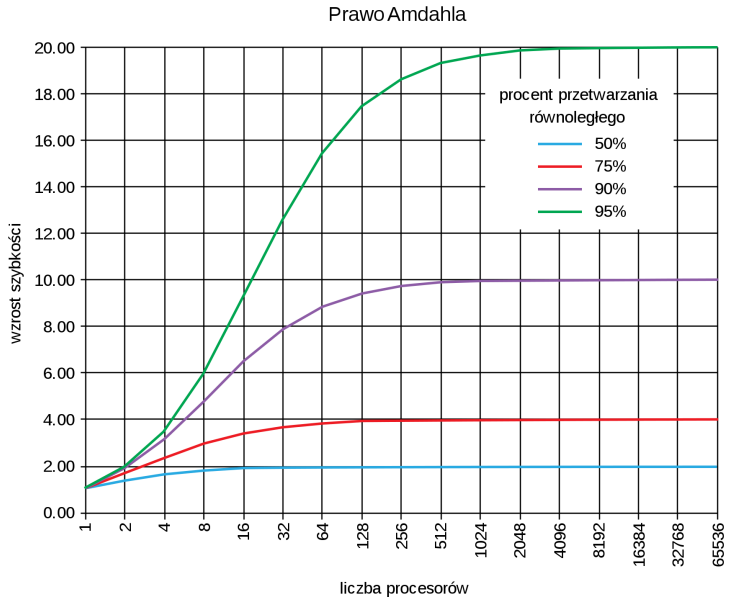




Prawo Amdahla i prawo Gustafsona-Barsisa

Dla stałej wielości problemu.

$$\lim_{p \to \infty} S(p) = \frac{1}{1 - a}$$



Prawo Gustafsona-Barsisa

$$S(p) = p - (1 - a)(p - 1), \text{ gdzie:}$$

p	ilość procesorów
$(1 - a)$	część procesu, która musi być wykonywana szeregowo

Ze wzrostem N wyrażenie $(1 - a)$ jest zbieżne i mniejsze od 1.
Wniosek: każdy wystarczająco duży problem da się zrównoleglić.

Wysoka skalowalność - wskazówki

Wątki mogą pracować niezależnie gdy:

- odczytują współdzielone dane
- zapisują własne (lokalne) dane

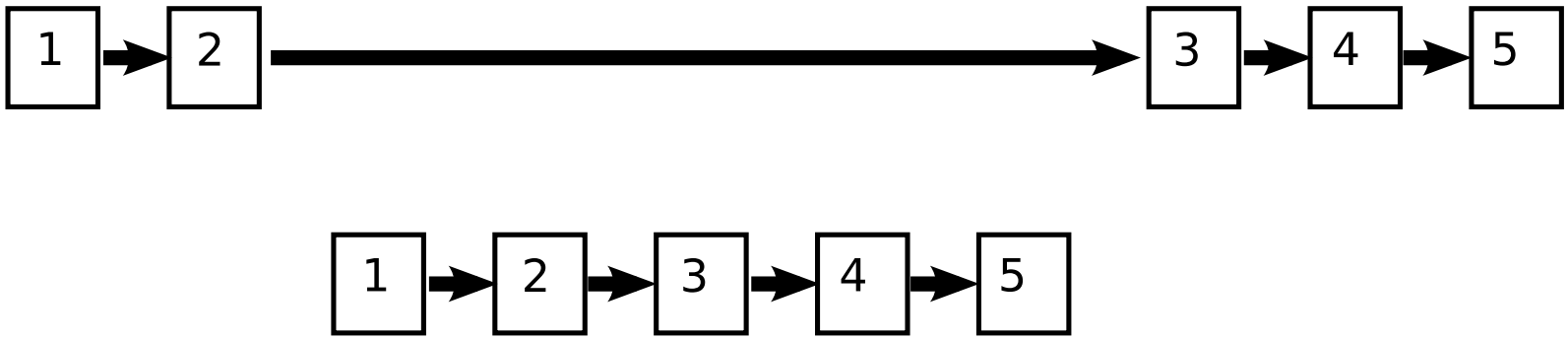
zapis współdzielonych danych jest kłopotliwy (spowalnia)

Wyróżnia się „szybką” i „wolną” ścieżkę w funkcjach wykonywanych w wątkach

- brak blokad na ”szybkiej ścieżce” (blokady, wykorzystywane nawet tylko do odczytu zapisują stan)
- usuwanie współdzielonych obiektów do zapisu (lepiej informację wyjściową przechowywać niezależnie dla każdego wątku, a później scalać wynik)

Podwójne sprawdzanie (double-checked locking)

```
class Singleton {
/*  patrz wzorzec Singeton  */
private:
   static Singleton* pInstance_ = nullptr;
};

Singleton& Singleton::getInstance() { //1
  if(!pInstance_) //2
    pInstance_ = new Singleton; //3
  return *pInstance_; //4
} //5
```



Podwójne sprawdzanie (2)

```
//Singleton poprawny, ale nieefektywny
Singleton& Singleton::getInstance() {
  std::lock_guard guard(mutex_); //tworzenie w sekcji krytycznej
  if(!pInstance_)
    pInstance_ = new Singleton;
  return *pInstance_;
}
//Singleton wielowątkowy efektywny
Singleton& Singleton::getInstance() {
  if(!pInstance_) { //wzorzec podwójnego sprawdzania
    std::lock_guard guard(mutex_); //tworzenie w sekcji krytycznej
    if(!pInstance_)
      pInstance_ = new Singleton;
  }
  return *pInstance_;
}
```

Singleton DCLP (Double checked locking pattern)

```
Singleton& Singleton::getInstance() {
  if(!pInstance_) { //wykorzystuje kolejność operacji w linii 5
    std::lock_guard guard(mutex_);
    if(!pInstance_)
    //zakłada kolejność: przydział pamięci, konstruktor, przypisanie
      pInstance_ = new Singleton;
  }
  return *pInstance_;
}
```

- może być niepoprawny (brak 'sequence point')
- próby poprawy są nieskuteczne np. dodatkowe instrukcje, wstawianie 'sequence point', oznaczanie obiektów jako zmienne (volatile)
- działa w praktyce

Wniosek
singletony inicjować w tym samym wątku (start aplikacji)

Wątki i mechanizm wyjątków

- wyjątki mogą być rzucane i wyłapywane w niezależnych wątkach
- wątek nie powinien rzucać wyjątku, który będzie wyłapywany w innym wątku

```
class MyThread {
  //...
  void operator()() { //funkcja wątku użytkownika
    try {
      //...
    } catch(...) { /* Wyłapuje wszystkie wyjątki */ }
  }
};
```

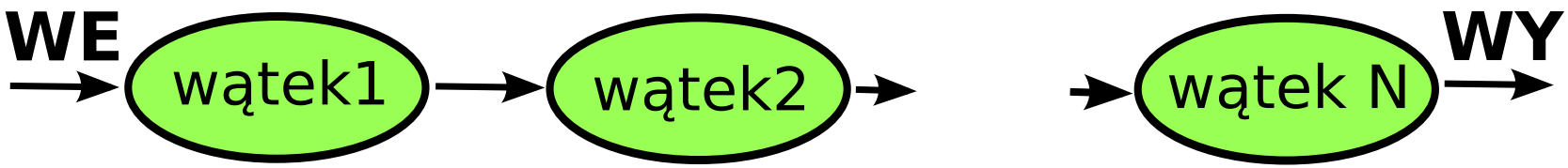
Wzorzec monitora (pasywny obiekt)

Obiekt sam zapewnia, że metody mogą być wołane przez różne wątki.

```
class Queue {
  static const char END_TOKEN = '\0';
  std::queue<char> data_;
  std::mutex mutex; //wewnętrzny mutex
public:
  void put(char c) {
    std::lock_guard guard(mutex);
    data_.push(c);
  }
  bool get(char& c) {
    while(bool read = false; !read) {
      std::lock_guard<std::mutex> lock(mutex);
      if( ! data_.empty() ) {
        c = data_.front(); data_.pop(); read = true;
      }
    }
    return c != END_TOKEN;
  }
```

Przetwarzanie potokowe

algorytmy wykorzystujące przetwarzanie potokowe wygodnie tworzyć przy wykorzystaniu wątków



Współbieżna blokada

Problem czytelników i pisarzy (multiple-readers /single-writer locking pattern)

- czytelnicy: wątki nie wykluczające się nawzajem
- pisarze: wątki wykluczające każdy inny wątek, zarówno czytelnika jak i pisarza

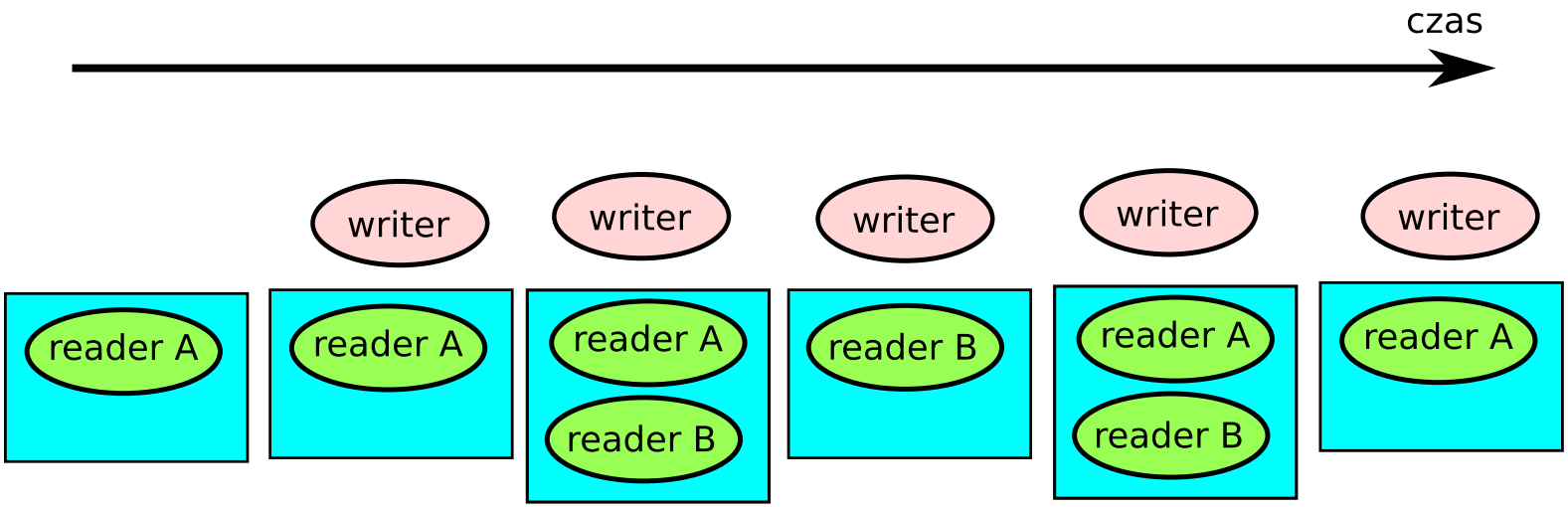
Bardziej skomplikowany od sekcji krytycznej.

- std::shared_mutex (C++14)
- boost::shared_mutex

- M::lock, M::try_lock, M::timed_lock
- M::lock_shared, M::try_lock_shared, M::timed_lock_shared

Zagłodzenia (starvation)

dany wątek nie może się wykonywać, ponieważ cały czas jest blokowany



Dziękuję

robert.nowak@pw.edu.pl